THESE de DOCTORAT de l'UNIVERSITE PARIS VI

Informatique

Les Philosophies sur l'Interaction des Processus

(Process Interaction Models)

Steven Ericsson Zenith[1]

Ecole Nationale Supérieure des Mines de Paris
Centre de Recherche en Informatique
35 rue Saint-Honore 77305 Fontainebleau FRANCE.

soutenue le 1er Julliet 1992, 09:00h.

pour obtenir le titre de docteur de l'université de Paris VI.

Laboratoire
METHODOLOGIE & ARCHITECTURE DES
SYSTEMES INFORMATIQUES
Université Pierre et Marie Curie - PARIS VI

devant le jury composé de:

M. Claude Girault (President)	M. Ivan Lavallee (Raporteur)
M. Paul Feautrier (Directeur)	M. Henri Bal (Raporteur)
M. Suresh Jagannarthan	M. Pierre Jouvelot

[1]Funding for this work was provided in part by the Science Frontiers Institute, the Association pour la Recherche et le Développement des Méthodes et Processus Industriels (ARMINES). and USA Government NSF grant CCR–8657615 (at Yale University)

Résumé Etendu

La création des programmes comme un ensemble de types de comportements connus sous le terme *processus* et un certain mécanisme par lequel ils entrent en *interaction*, semble évoluer vers une solution d'ingénierie positive en matière de programmation des machines parallèles. Malgré les gros efforts effectués pour la compréhension des philosophies du processus (ex.: Hoare, Milner, Pratt et d'autres), ces philosophies qui sont les moyens d'interaction entre les processus, demeurent aujourd'hui un sujet à débattre.

Les philosophies sur l'interaction des processus sont une composante intégrale de la théorie et la pratique du processus parallèle, définissant les moyens d'interaction utilisés par les processus concourrants; par "interaction" on entend non seulement l'échange des données mais aussi la synchronisation inter-processus.

Etant à la recherche d'une philosophie passe-partout pour la programmation des machines parallèles, il est souhaitable de fournir une portabilité, un caractère expressif qui ne détournera pas l'esprit du programmeur de sa tâche et enfin une efficacité et cela indépendamment de l'architecture de la mémoire de la machine.

Le transfert de message fut pendant un temps, la philosophie préférée en matière d'interaction et cela du fait de sa simplicité et de la facilité de sa compréhension. Cependant, le Transfert Généralisé de Message, caractérisé par le langage Occam [May 88], est une philosophie inappropriée dans le domaine de programmation passe-partout des ordinateurs parallèles. Il crée des situations où le programmeur se trouve préoccupé par des problèmes complexes de distribution des données tandis que les implémentations sont contraintes de copier les données, qui dans le cas contraire, seraient échangées par référence.

La philosophie Linda [Gel85] annonçait une solution aux problèmes dues à la complexité de la distribution des données, en présentant le transfert généralisé de message. Les programmeurs n'ont pas à être concernés par des problèmes de distribution des données. Toutefois, la philosophie Linda est imparfaite et les sémantiques de cette philosophie sont imprévisibles; ce qui amène les programmeurs à écrire des codes en fonction de leur compréhension de la façon dont un optimiseur particulier ou encore un protocole correspondant sous-jacent se comporte; ce qui renverse toute conception de portabilité.

Cette thèse débat de ces problèmes et présente une nouvelle philosophie d'interaction. *Ease* est décrite comme étant un langage de programmation passe-partout, impérative et de haut niveau. Un programme est un ensemble de processus qui exécute de manière concourrante, créant et dialoguant par l'intermédiaire des structures des données partagées très spécifique qu'on appelle "Contexts".

Ease est nouveau dans le sens où un Context fournit un intermédiaire à vocation prioritaire, et introduit de manière stricte, dans lequel les structures des données distribuées sont construites et par lequel les processus peuvent entrer en interaction. *Ease* fournit des opérateurs reliés simples et symétriques qui permettent la construction et l'échange efficace des structures de données complexes; des constructions aussi bien pour la concurrence coopérative

que subordonnée, et un mécanisme pour créer des ressources statiquement réutilisables et virtuelles.

La définition du langage est utilisée comme un système de référence par la philosophie tandis que la philosophie est suffisamment distincte pour être ajoutée aux pratiques conventionnelles tout comme le transfert de message et Linda l'ont été par le passé; Une définition de C-avec-*Ease* est présentée. Cela illustre la manière dont les langages hybrides peuvent être créés. Un compilateur de prototype est également décrit.

Conclusion

L'objectif de cette thèse est de développer une philosophie et un langage pour simplifier le développement des programmes parallèles; elle est pilotée par l'observation des défauts dans les philosophies existantes qui détournent l'esprit de l'ingénieur. Ces points ont été soulevés dans l'introduction de cette thèse, à savoir essentiellement la confusion qui apparaît dans la distribution des données en ce qui concerne le transfert généralisé de message et la confusion dans la conception approximative de Linda en matière de la correspondance des valeurs associatives.

La solution à ses confusions est d'adopter une abstraction de l'espace de données qui cacherait la complexité de la distribution mais aussi énoncer la localisation directement pour permettre à l'ingénieur de se concentrer sur le développement de l'algorithme et la localisation des données. La solution présentée, *Ease* Contexts, est basée sur les structures des données partagées ayant des caractéristiques particuliers, et qui sont identifiées, par expérience, comme étant celles communément utilisées par les développeurs d'application.

L'objectif a également été de désigner une philosophie pouvant raisonnablement revendiquer une indépendance dans l'architecture; en d'autres termes, une philosophie qui fournit non seulement la portabilité du programme mais aussi l'uniformité de la performance à travers différentes architectures de la machine. Le transfert généralisé de message n'est pas uniforme car les structures significatives de message deviennent des opérations de reproduction lorsqu'elles sont retirées d'un système distribué et placées dans un système de mémoire partagée. Linda n'est pas uniforme car les optimisations Tuple Space ont des effets radicalement différents et une localisation de données implicite.

La solution à ces problèmes serait de fournir un mécanisme qui encapsule l'échange par référence pour les structures de données significatives mais dont les sémantiques sont valables pour les reproductions entre les espaces d'adresse disjoints, permettant ainsi une implémentation efficace, aussi bien sur les architectures de mémoire partagée que mémoire distribuée. Cette solution est apportée par des opérations simples et symétriques sur les Contexts qui fournissent une forte expression de localisation. Par ailleurs, cet objectif limite le choix des caractéristiques de structure des données, à savoir le choix accordé aux Contexts et aux opérations qui leur sont rattachées; à titre d'exemple, donner un type à un Context et un opérateur "test of presence" pourrait considérablement compliquer l'implémentation

(qui requiert des opérations de recherche) et l'uniformité du langage. Il vaut mieux laisser l'implémentation de ce genre de structure au soin de l'ingénieur ou au support de la bibliothèque.

Des objectifs secondaires ont été atteints dans la conception du langage complet. Occam était basé sur les principes mathématiques de CSP mais s'est avéré défectueux en tant qu'outil d'ingénierie dans bon nombre de cas en sus des problèmes associés au transfert généralisé de message. Dans la conception de *Ease*, j'ai tenté d'apporter une réponse aux frustrations jadis rencontrées par les programmeurs d'Occam bien que j'ai gardé la même base mathématique. Certaines sont tout simplement (bien que important) syntactiques mais il y a également d'autres réponses qui sont les suivantes: les solutions principales visent le système des types, la manipulation de structure des données, le support des systèmes intégrés, le temps réel et les ressources.

En conclusion, je présente des arguments sur l'importance globale de ces objectifs et la manière dont les solutions présentées apportent une aide. Je réponds aux questions posées par Hennessy et Patterson qui résument les problèmes traités dans cette thèse.

En considérant la difficulté que représente la programmation des machines parallèles, Hennessy et Patterson mettent en avant ce problème:

> "Why should it be so much harder to develop MIMD programs than sequential programs? One reason is that it is hard to write MIMD programs that achieve close to linear speed up as the number of processors dedicated to the task increases. ... think of the communication overhead for a task done by a committee .... While n people may have the potential to finish any task n times faster, the communication overhead for the group can prevent it from achieving this .... (Imagine the communication overhead going from 10 people to 1,000 people to 1,000,000)."
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 575.

Reste à savoir si *Ease* pourrait solidement contribuer à la diminution de cette complexité; toujours est-il que seule l'expérience d'application peut réellement nous le dire. C'est le jugement pragmatique des ingénieurs, et non le raisonnement, qui est décisif, quel que soit le niveau de sa réussite. J'ai soutenu que *Ease* réduit la complexité dans la programmation parallèle en supprimant la complication du transfert de message; les processus individuels se concentrent sur les données et non sur les autres processus qui partagent ces données, et par conséquent, la localisation apparaît de façon naturelle.

Je soutiens un fait qui est en rapport direct avec le sujet traité, car c'est une hypothèse fondamentale de cette thèse: programmer avec une composition parallèle est plus simple que programmer uniquement avec une composition séquentielle. Les programmeurs séquentiels sont préoccupés par l'imbrication des activités d'un programme et des approches orientés-objet se sont développées par suite de nécessité de traiter ce problème. L'évolution d'une philosophie orientée-objet vers des philosophies de processus n'est pas une projection déraisonnable.

Hennessy et Patterson poursuivent:

> "Another reason for the difficulty in writing parallel programs is how much the programmer must know about the hardware. On a uniprocessor, the high level language programmer writes his program ignoring the underlying machine organization — that's the job of the compiler. For a multiprocessor today, the programmer had better know the underlying hardware and organization if he is to write fast and scalable programs. This intimacy also makes portable parallel programs rare."
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 575.

C'est une observation importante. Une observation qui a entrainé un effort considérable dans le développement de la traduction automatique des programmes conventionnels vers des formes efficaces des machines parallèles. Mais le programmeur uniprocesseur y est allé doucement ces dernières années. La nature de la bête a été tolérante vis-à-vis d'une dépendance excessive des structures globales, et indulgente vis-à-vis des effets secondaires. De nouvelles philosophies de programmation doivent se développer; des philosophies qui désignent des localisations identifiables et qui sont dépourvues d'effets secondaires; *Ease* fournit ces deux caractéristiques. En outre, la philosophie de *Ease* fait abstraction de l'architecture de la mémoire sous-jacente de l'ordinateur en assistant le compilateur dans ses efforts, à savoir, réaliser des programmes parallèles portables avec une implémentation efficace.

Le rôle du compilateur doit être, finalement, de fournir l'emplacement efficace des données et des processus, permettant à des programmes évolutives rapides d'écrire sans se préoccuper de l'ordinateur sous-jacent. Cette tâche peut être fortement aidée par la philosophie du langage et de l'interaction — bien qu'aucune des deux ne peut apporter une solution directe puisqu'elles sont dépendantes de l'architecture de l'ordinateur. De nouveau, Hennessy et Patterson:

> "The real issues for future machines are these: Do problems and algorithms with sufficient parallelism exist? And can people be trained or compilers be written to exploit such parallelism?"
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 579.

La première question dépasse le sujet de cette thèse, bien que l'observation du monde naturel et les applications existantes laissent supposer que même s'il existe une certaine limite concernant le parallélisme dans les problèmes et algorithmes spécifiques, la composition parallèle élaborée de ceux-ci demeure utile.

Toutefois, c'est la seconde partie de cette question qui est diørectement traitée dans cette thèse. Former les ingénieurs requiert un investissement considérable. Les ingénieurs, autant

que possible, ont besoin des outils familiers. Occam fut confronté à des resistances, autant pour son notation idiosyncrasique que pour son introduction de parallélisme. Ce n'est pas tant le parallélisme qui est complexe, c'est plutôt la complexité de l'interaction du processus qui a présenté un obstacle. Dans *Ease*, cette complexité est considérablement réduite tandis qu'un style notationnel familier est conservé. Les compilateurs sont fortement assités dans leur tâche par la nature, dépourvue d'effect secondaire, du langage et l'abstraction du Context. Hennessy et Patterson approfondissent la question:

> "Compilers of the future have two challenges on machines for the future:
>
> 1. Lay out of data to reduce memory hierarchy and communication overhead, and
> 2. Exploitation of parallelism."
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 581.

Une première partie est traitée par l'augmentation du degré de localisation des données dans les programmes; la seconde est fournie par la nature des structures de Context, des expressions et des fonctions sans effets secondaires (qui permettent une identification simple du parallélisme subtil par le compilateur) et des constructions parallèles explicites.

En somme, un langage de type sûr avec une forte base mathématique qui ne s'impose pas à l'ingénieur, est généralement souhaitable, permettant ainsi l'application des méthodes conventionnelles, le cas échéant.

Contents

1 Introduction **17**

2 Process Interaction Models **27**

2.1 Processes and termination 28

2.2 Global memory interaction models 31

2.2.1 Semaphores — conditional precedence 31

2.2.2 Critical regions 34

2.2.3 Monitors 35

2.2.4 Threads and semaphores on the Encore Multimax 36

2.2.5 Distributed Shared Memory 38

2.3 High level process models 39

2.4 Communication 42

2.4.1 Synchronized message passing 43

2.4.2 Non–synchronized message passing 44

2.4.3 Express: an operating system expedient 45

2.5 Logically shared data structures 47

3 Implementing Interaction Models **51**

3.1 Implementation of scheduling 51

3.2 Implementing semaphores 53

3.3 Implementing communication 53

3.4 Implementation and optimization in Linda 58

3.4.1 Linda: Division of tuple space into distinct subsets 59

3.4.2 Linda: Implementation of distinct subsets 59
3.5 Implementation of logically shared data structures 62
3.6 Comparison 70
3.7 Consistency reviewed 72

4 Critique 75
4.1 What's wrong with global shared memory? 75
4.2 What's wrong with message passing? 76
4.3 What's wrong with Linda? 79

5 Semiotics 83
5.1 Performance semantics 83
5.2 *Semiotics* 84
5.3 *Semiotics* in Computer Science 85

6 Ease — A New Proposal 87
6.1 A new model arises 89
6.2 Contexts — shared data structures 89
6.3 Operations – actions on shared data 90
6.4 Uniformly building and using resources 91
6.5 The process model 91
6.5.1 Cooperating processes 91
6.5.2 Subordinate processes 92
6.5.3 Type associativity 92
6.6 Priority 93

7 Writing Programs with Ease 95
7.1 A brief introduction to the notion of "process" 95
7.2 A brief introduction to the notion of "data" 97
7.2.1 Names, environment and scopes 98
7.2.2 Expressions 98
7.2.3 Types 99

7.3 Actions . . . 99
7.3.1 Properties of termination . . . 99
7.3.2 Assignment . . . 101
7.3.3 Interaction . . . 101
7.4 Names, local and nonlocal data structures . . . 103
7.4.1 Constants . . . 103
7.4.2 Variables . . . 104
7.4.3 Scope . . . 105
7.4.4 Shared data structures . . . 106
7.4.5 Singletons and arrays . . . 106
7.4.6 Streams . . . 108
7.4.7 Bags – unordered sets . . . 110
7.4.8 Put – review . . . 110
7.4.9 Typing, elements and expressions . . . 111
7.4.10 Variants – type associative contexts . . . 113
7.4.11 Manipulating contexts . . . 113
7.4.12 Automatic termination of subordinates . . . 114
7.5 Composition . . . 114
7.5.1 Conditional processes . . . 114
7.5.2 Nondeterministic choice . . . 116
7.6 Resources . . . 118
7.6.1 Combinations . . . 118
7.7 Placement . . . 120
7.8 Undetermined values, invalidity and divergence . . . 120
7.8.1 Detectibility of undetermined values . . . 121
7.9 Exception handling . . . 122

8 Definition of Ease **123**
8.1 About the definition . . . 123
8.1.1 Syntax . . . 123
8.1.2 Validity statements . . . 125

8.1.3 Equivalence statements 125
8.1.4 Context free grammar 126
8.2 *Ease* pragmatics 126
8.2.1 The process model 126
8.2.2 Machine and process 126
8.2.3 Contexts and interaction 127
8.2.4 Expressing concurrency 127
8.2.5 The pragmatics of the parallel composition 129
8.2.6 Resources 129
8.2.7 Failure and error handling 129
8.2.8 Definitions and instances 130
8.2.9 Allocation 130
8.2.10 Typing 130
8.2.11 Expressions 130
8.2.12 Procedures 130
8.2.13 Lambda expressions (functions) 130
8.2.14 Modules 131
8.2.15 Stylistic conventions 131
8.2.16 Alien language processes 131
8.3 *Ease* syntax and semantics 131
8.3.1 Actions 132
8.3.2 Constructions 135
8.3.3 Types 144
8.3.4 Element 147
8.3.5 Expressions 150
8.3.6 Type constraint, assertion and casting 155
8.3.7 Keywords and names 156
8.3.8 Scope 157
8.3.9 Declaration 157
8.3.10 Definitions 159

8.3.11 Other recursive definitions 166
8.3.12 Comments 166
8.3.13 Modules 167

9 CSP 169
9.1 Communicating processes 169
9.2 Shared data structures 171
9.3 A brief overview of CSP 176
9.3.1 Events, alphabets and processes 177
9.3.2 Traces 179
9.3.3 Parallel processes 181
9.3.4 Communication 182
9.3.5 Subordination 182
9.3.6 Conditional 183
9.4 A CSP semantics of the *Ease* model 183
9.4.1 Singleton semantics 183
9.4.2 Stream semantics 184
9.4.3 Bag semantics 184
9.4.4 Context usage 184
9.4.5 Interaction operators 185
9.4.6 Shared resources — combinations 185
9.4.7 Parallel semantics 186

10 Ease Implementation 189
10.1 The implementation of Bags and Streams 189
10.1.1 Reference exchange 192
10.1.2 Replication 192
10.1.3 Bags on more than two distributed memory nodes 195
10.2 The implementation of Singletons 196
10.3 Type associative contexts 196
10.4 Placement 197

10.4.1 Manual placement . . . 197
10.4.2 Automatic placement . . . 197
10.4.3 Fine grain data parallelism. . . . 197
10.5 Program transformation . . . 197

11 Future work and directions 199
11.1 Prototype compilers . . . 199
11.2 Future directions . . . 201

12 Conclusion 203

A Programs in Ease 215
A.1 Conway's game of life. . . . 215
A.2 Stream exchange sort . . . 221
A.3 A shared stack . . . 223
A.4 The sieve of Eratosthenes . . . 225
A.5 Common cores . . . 227
A.5.1 Matrix multiply . . . 227
A.5.2 Gauss-Jordan . . . 228
A.6 Farming: master/worker . . . 229

B A YACC/Bison grammar 233

C A FLEX lexer 249

D C–with–Ease 255
D.1 *C–with–Ease* definition . . . 255
D.1.1 Process definition . . . 255
D.1.2 Parallel construction . . . 256
D.1.3 Replication . . . 257
D.1.4 Contexts – shared data structures . . . 257
D.1.5 Context type definition . . . 259
D.1.6 Context allocation . . . 259

D.1.7 Interaction 260
D.1.8 Resource 261
D.1.9 Scope 261

List of Figures

2.1 Semaphores provide guards for the implementation of critical regions; enforcing sequential synchronization. Here, although composed concurrently, P and Q are forced to be sequential since only one can succeed past $\mathcal{P}(S)$ before it must wait an instance of $\mathcal{V}(S)$ to reset the semaphore. 33

2.2 Critical regions: a composition of the regions P and Q acting on G must be sequential. 34

2.3 Message passing: a deceptively simple idea to sell. A process inputs a value output by some other process. 44

2.4 Linda: another deceptively simple idea to sell. Here we trace the state of tuple space through 4 changes. The effect of the "in" and "out" primitives are clearly illustrated. As a result of the second state change $i = 6$, "rd" has no affect on the state of tuple space but *reads* a tuple, since selection is nondeterministic $j = 9 \vee j = 3$. 50

3.1 Implementation of $\mathcal{P}$ and $\mathcal{V}$. 54

3.2 Point–to–point communication. A pending process. 55

3.3 Implementation of point–to–point output. 56

3.4 Implementation of point–to–point input. 57

3.5 Queues implement a shared data structure. 64

3.6 Implementation of a read operation on logically shared data. 65

3.7 Implementation of a get operation. 66

3.8 Implementation of write operation. 67

3.9 Implementation of persistence daemon. 69

4.1 Naming channels can be confusing. 78

7.1 Process concepts. 96

7.2 Data concepts 98

7.3 The actions. Although *Ease* has no conception of the pointers familiar in other languages, Put and Get encapsulate a mechanism that allows the exchange of data by reference; thus providing an abstraction from the underlying memory subsystem architecture and reducing copy operations in an implementation. . . 103

7.4 The scope of a name is the process following the specification. The above case illustrates three scopes, those of x, y and c, in addition the *environment* of this process includes the scope of the free name $\mathcal{K}$. 105

7.5 Stream context: The order of values in a stream is determined by the contributors. Here a single process (on the left of our picture) outputs three values $3, 2, 1$, inputs input the least recently output value (3 in this case). 109

7.6 Stream context: Multiple contributors still provide a stream guarantee – a process will not input a value from a contributor output earlier than a value previously input. 109

9.1 Partial order diagram for the interacting sequences: 173

9.2 Extended partial order diagram to illustrate the interleaving of: 174

9.3 A further illustration of the extended partial order relation 175

Dedicated to Sally.

Abstract

The construction of programs as a collection of behavior patterns called *processes* and some mechanism by which they *interact* is evolving as a positive engineering solution to programming parallel machines. While much work has been done on understanding process models (e.g., Hoare, Milner, Pratt and others), the models by which processes interact remain today a subject of debate.

Process interaction models are an integral component of parallel processing theory and practice, defining the means by which concurrent processes interact; where "interaction" means not only the exchange of data but also synchronization between processes. In searching for a general purpose model to program parallel machines, it is desirable to provide portability, an expressiveness which does not distract the programmer from the task in hand and efficiency independent of the memory architecture of the machine.

Message passing has, for sometime, been a favored interaction model on the basis that it is simple and readily understood. However, Generalized Message Passing, as typified by the language Occam[INM88], is an unsuitable model for general purpose programming of parallel computers. It causes programmers to be preoccupied with complex issues of data distribution and implementations are compelled to copy data that might usefully be exchanged by reference.

The Linda[Gel85] model promised a solution to the problems of data distribution complexity introduced by generalized message passing. Linda programmers need not be concerned by issues of data distribution. However, the Linda model is flawed; performance semantics in the model are unpredictable, leading programmers to write code based on an *understanding* of how a particular optimizer or underlying matching protocol behaves – subverting any meaningful portability, and data structures must be contrived to develop an uncertain construction of distributed data structures.

This thesis discusses these issues and presents a new process interaction model. *Ease* is described as a general purpose, high level, imperative programming language. A program is a collection of processes which execute concurrently, constructing and interacting via strictly typed shared data structures called "Contexts".

Ease is novel in the following regard: a Context provides a priority oriented and strictly typed intermediary in which distributed data structures are constructed and by which processes may interact. Ease provides simple and symmetric binding operators which allow complex data structures to be constructed and exchanged efficiently, constructions for both co-operative and subordinate concurrency, and a mechanism for constructing statically reusable and virtual resources.

The language definition is used as a reference vehicle for the model but the model is sufficiently distinct to be added to conventional practices in the way message passing and Linda have been in the past; a definition of C–with–Ease is presented. This illustrates how hybrid languages can be constructed. A prototype compiler is described.

Thanks

Special thanks to those who during the completion of this work found the time from their busy schedule to answer my questioning and to provide guidance: Paul Feautrier, Susan Flynn-Hummel, Guy Harriman, Suresh Jagannathan, Pierre Jouvelot, Vaughan Pratt. Thanks also to John Redman at the University of Western Australia for his useful feedback during the development of his *Ease* compiler.

Recognition and thanks to all the discussions with fellow "skunks" in the wake of the first transputer development; in particular between Jon Beecroft, Bob Krysiak, Guy Harriman, Hossein Yassaie and myself.

Thanks too to Dr.Noel Kantaris and Dr.Alan Lamb at the Camborne School of Mines, Cornwall, England for their long friendship and support: also Corrine Ancourt, Francois Irigion, Pierre Jouvelot, Michel Lenci, and others at Ecole des Mines de Paris.

My sincere thanks also to those people to whom my arrogance, ignorance, and questioning has often been a source of irritation. They taught me many lessons, for which I am sincerely grateful. In particular my thanks to Prof.David May FRS and Roger Shepherd at INMOS and to Prof.David Gelernter and Dr.Nick Carriero at Yale University, Connecticut, USA.

I am greatly indebted to the University of Paris, and in particular the University of Marie and Pierre Curie (PARIS VI) for the rare insight they have shown in accepting and encouraging this thesis. Not for its content but rather for the opportunity they provided for someone with only industrial experience and little conventional schooling.

Thanks also to my thesis committee Henri Bal, Prof.Claude Girault, Suresh Jagannathan, Pierre Jouvelot, Prof.Ivan Lavallee. My greatest respect and thanks to Prof.Paul Feautrier, my director, for his guidance and insight.

The first definition of Ease appeared in a rather hurried report published in July 1990 in a Yale University research report — RR809. That work was partly funded by a USA Government NSF grant CCR–8657615. My thanks to the NSF and to Prof.Gelernter for his support. Funding for 1991–1992 was also provided by ARMINES and Science Frontiers.

My thanks too to Lazarus Long and the Tertius family. My discovery of them over this same period kept me sane; thank you Robert Heinlein.

Blessed thanks to Lao Tzu whose simple words gave me the keys to enlightenment which — along with the loose change — rattle still in my trouser pocket. Finally my thanks to my beautiful wife Sally and our children, Celestial, Freedom, Mystical, and Zen; without you this would all be much less fun.

Preface

Summary of the contribution

The principal contribution of this thesis is to the field of parallel processing models and languages. The thesis discusses existing popular process interaction models, and makes a new proposal that arises from the rationale. The new model is simple in use, consistent and efficient in implementation. The model has important advantages over existing models for generalized process interaction such as Message Passing or Linda.

These advantages are two fold. First, the language allows the simple and explicit expression of locality by the construction of uniform shared data structures. In essence this provides a mechanism that frees the programmer of data distribution preoccupations and the need for contrived distributed data construction found in other models. Second, the model encapsulates a mechanism that allows an implementation to exchange data by reference. This is of particular advantage in the manipulation of non–trivial data structures and reduces data exchange and memory space costs in both shared memory, and distributed memory architectures.

The language presented is suitable for the expression of both fine and coarse grain parallelism. It contains simple mechanisms for resource management and a uniform approach to exceptional termination (including error handling) on parallel machines.

Structure of the thesis

Chapter one The thesis begins with a short, informal, and historical **introduction** that aims to enlighten the reader by giving a rationale for the direction taken in the work.

Chapter two considers the range of **process interaction models**; covering Global Shared Memory, Communication and Shared Data Structures.

Chapter three considers implementation issues. A particular implementation is considered for three fundamentals. Semaphores, which represent the fundamental of mechanistic

synchronization; Point–to–point message passing, which represents the fundamental of all message passing; and queues, representing a comparable mechanism for the implementation of logically shared data structures. A simple comparative assessment of the costs involved is given.

Chapter four contains a Critique that details problems with the models. In particular, a critique is made of the popular Shared Memory, Occam and Linda models.

Chapter five briefly discusses the objective of Semiotic definitions and the importance of making pragmatic statements in the language definition that allow the consistent and efficient use of a language.

Chapter six introduces the *Ease* model of interaction; a new proposal that addresses the issues raised by the Critique.

Chapter seven presents the full language *Ease* in an informal style.

Chapter eight provides a more formal definition of the *Ease* language.

Chapter nine provides a CSP (behavioral) semantics of the *Ease* model.

Chapter ten discusses current implementations of *Ease*; the implementation by the author and implementations completed by John Redman (University of Western Australia) and Tim MacKenzie (Monash); both Honours Students at an Australian Universities. Both these implementations generate C for the GNU C compiler. The author's version implements the language in a static style — the Redman implementation utilizes the uSystem kernel, the Mackenzie implementation uses a network of workstations.

Chapter eleven discusses future work and directions.

The thesis concludes with a conclusion of drawn upon the work and references.

Appendices provide examples written in *Ease*, a full YACC grammar for the language, a FLEX lexer, and a definition of C–with–Ease.

Related work

There is some historical and present related work in the area of programming models for parallel machines.

Occam and CSP

Occam was developed at INMOS by a team of engineers under the direction of David May at INMOS Limited's Bristol research center in England. Occam was first prototyped in the early 1980's[May83], and was fully developed in an Occam 2 definition released in October 1987.

Occam is an implementation of the ideas of Tony Hoare found in CSP[Hoa85]. Occam continues to be developed at INMOS.

CSP is the landmark work of Prof.C.A.R.Hoare and others at Oxford Programming Research Group, Oxford University, England. My own endeavour to escape from the influence of this work have inevitably served to strengthen my understanding of it and develop a realization of its magnitude and importance. If there is to be an "Elements" of computation I'm inclined to believe it will be founded on this work.

Linda

Linda was first suggested by Dave Gelernter in 1985. This model has had a significant effect on subsequent developments in the field — including the development presented here.

This development continues at Yale and elsewhere. Most notably the work has been extended into the domain of symbolic processing by Suresh Jagannathan, now at NEC Research labs in New Jersey, USA[Jag92].

Orca

Orca also develops the Linda ideas of shared data structures. This work was first undertaken by Henri Bal in Andy Tannenbaum's group in Amsterdam[Bal89] and is also a component of the Amoeba environment.

Ada and Pascal Plus

Ada and Pascal Plus implements the Remote Procedure Call (RPC) as the basic paradigm for processes and interaction. RPC is a simple variant of communicating sequential processes

modeled on the construction of procedures as processes whose formal inputs and outputs are communications.

Chapter 1

Introduction

(A little history)

Before getting into the core of my thesis I wish to take a few informal moments to detail a little history; history that has direct bearing on the subject of this thesis.

The construction of parallel computers continues a unique period of technological innovation in human history. In the last decade of the twentieth century we are no longer in awe of such machines. Rather we are in awe of how they might be utilized. This is perhaps analogous to the position at the end of the last century in the development of the internal combustion engine. The internal combustion engine had existed for several years. Few could begin to imagine in that final decade the extent to which that technology would impact on the following century. Fewer still could imagine that it would herald an age of motor vehicle in which international flight became an experience of mundane character.

What other revolutionary successes, analogous to the success of the Wright brothers or Henry Ford, await in some other discipline; waiting for the enormous power and versatility which can be found in the technology of parallel computers?

One observation is certain: Parallel Processing will become embedded as the core technology in most aspects of human technological endeavor. Be it in the so-called "grand challenges" such as simulation of the global climate, the implementation of satellite communications systems, or control and monitoring of the family motor vehicle. Parallel Processing will come to dominate the next century.

Yet our technology appears to differ significantly from that of the internal combustion engine in the order of its complexity. Actually, this is a statement I find difficult to justify. Is this apparent complexity any greater than the relative challenge internal combustion engines presented engineers during the latter half of the nineteenth century? I think not.

We are a sibling science, born of the same stuff as the internal combustion engine and the parent/sibling electronics. We are a child of human innovation. Yet we are surely the

youngest child in this family. We are clumsy, naive, often wrong and only just beginning to find our feet.

Like a child we stumble from one idea to the next and often find it difficult to maintain a focused attention. In one moment it is logic programming; in the next it is functional. One moment we are making over extensive use of grandiose mathematics, and in the next we make careless and ambiguous statements in common language.

Like all children we are developing methods to deal with our domain. Methods which enable us to execute a desired task in a timely and complete fashion, and also methods which enable us to explain to the other kids on the block just how we did it.

Sometimes the methods themselves lead us astray, and we find it necessary to modify our approach in the light of our experience. We are learning and growing; this is as it should be.

It is important to recall that our science[1] is an engineering science. The results of our research will be utilized by engineers and scientists trying to solve real problems with real machines. The measure of our success is how well we address the needs of these engineers.

Much of the direction taken in this thesis is motivated by first hand industrial experience. Experience that is difficult to quantify scientifically, with intuition difficult to assess qualitatively, though this thesis may be some measure. If I had the time and money I would gladly go back over the years and talk again, making a more detailed study, to each of the opinionated engineers whose irritation finally nagged me enough to present solutions founded on their loose remarks.

I was at the INMOS Research Center, in Bristol England, during the development of Occam and helped take that language from its prototype form to its complete form as Occam 2 (during 1985–1987). I worked on the detail of the language and its definition. The communicating process model of Occam is a simple copy of the concepts developed by Tony Hoare at Oxford University. Under the direction of Prof. David May F.R.S., I wrote the language definition as it is in the *Occam 2 Reference Manual*, published by Prentice Hall under the name "INMOS Limited".

Subsequently, I remained at INMOS for several years where I was a member of the Computer Architecture Group and Transputer Development Team. There I worked on the requirements for operating systems on parallel machines, and silicon support for various applications (including AI and graphics). I was directly involved in the design of two microprocessors — the T810 (a "skunk project" designed to enhance the T800[2]) and the H1/T9000[3].

There also, because of my detailed knowledge of the language, I played the role of "Occam Guru", listening to customers who were using (or abusing) the language. I listened with increasing disquiet to their comments. For more than 3 years INMOS very kindly paid for me

[1]That section of Computer Science concerned with Computer Architecture and tools for the engineer — such as programming languages and models.

[2]The T810 was canned in the end [mainly] because limitations in the T800 process technology restricted the eventual clock speed.

[3]I was a member of the T9000 design team in my last months at INMOS.

to travel across Europe and the United States — mainly I listened to engineers. They told what rapidly became a familiar story.

I began to suspect we had got something wrong. Occam was sold to the world as, and designed to be, a general purpose programming language for machines built with multiple microprocessor devices (in particular the transputer). In practice, the language (though not the devices) turned out to be rather special purpose.

How was this so when it directly countered the Occam design goal?

Engineers encountered two main problems. Programs often ran much slower than anticipated — too often the multiprocessor program ran slower than the sequential version (not good for business). Many engineers liked the new language yet project managers complain that Occam is too difficult to use in a large project.

Typically, and briefly since the thesis develops these points further, engineers were persuaded by us at INMOS to use the process model (parallelism) in the free expression of their problem. This ideal has many desirable characteristics. The expression of problems in the real world is naturally parallel as has been pointed out extensively in the literature [Hoa85, Gel85]. Certainly, it is at least simpler than expressing the interleaving of related activities in sequence as required by conventional programming. It is modular since processes become software components with well defined interfaces (at interaction points), and the program structure provides guidelines for the final hardware description of the ideal machine for the application.

Further, excessive parallelism ensures there is always a process which can be scheduled during internodal communication latency; i.e., while one process on a device is waiting for data other processes continue to utilize the CPU.

This rationale is indubitable in well balanced programs explicitly designed to take advantage of a particular machine's topology and network latency. Unfortunately this rationale is flawed in the context of a literally translated Generalized Message Passing model. Why?

One reason lies in the addition of new overheads introduced by the message passing programming model. Communication between processes and communication between devices is not necessarily well mapped. Indeed, without fore–knowledge of the target topology, it is hardly ever. Although, the program did describe a "perfect" message passing machine on which the application could execute, as written, few ever build applications that way.

The processes implementing the excess parallelism invariably need to communicate. In embedded systems cost engineering is a prime consideration. In practice this always means that the application must be scaled down to fit a target requirement of some number of processors less than "optimal".

Why should this cause a problem? Point–to–point communication is a copy operation — be it between processes communicating across a network or (and here the problem arises) between processes which share access to the same memory subsystem. Yet the implementation of communication across a network and communication within the same memory subsystem have very different performance effects upon the CPU.

The latency[4] over a network is much higher than the latency for the same operation implemented by copying data in the same address space. However, the actual CPU time involved in a non–trivial message communication implemented by a copy operation is significantly higher than the network counterpart. Excess parallelism can hide the network latency (on the transputer, for example) almost completely; however, the shorter latency of a communication implemented by a copy operation has a direct CPU cost.

So, communication of a block of data in the same address space has a significantly higher performance impact upon the CPU than the higher latency network communication. In the transputer architecture (which utilizes a RISC-like load/store model) this is because all data must pass through the CPU in the copy operation. Further, even with DMA, in cache architectures the source and destination of messages have poor spatial and temporal locality[5] causing an indirect impact on performance via the memory subsystem; whereas a network communication only has a single consistent effect at either the source or destination address block.

We can see why this is a significant issue in programming by considering the following case. Given some Occam program written in a general purpose style (i.e. without direct consideration of the architecture of the target machine) we are faced with several implementation options.

As a first option we might consider building the optimal distributed memory multi–processor described by the topology of the program. We could take a naive approach and simply designate one process to each processor; if processors are very cheap we might not care that some idle time occurs due to data dependencies. While this may not be considered an optimal use of hardware, it is a good implementation of the program. We have here a specialized machine well suited to our particular program.

A second option might be to take some random number of processors on a distributed memory multi–processor less than optimal which represents constraints imposed by the available resource, cost engineering or an operating system. This is a common case and very often the result of attempting program execution on a general purpose MIMD machine. The mapping here will require a *scaling down* of the program to fit the available resource.

The process of scaling down the program will cause collections of processes to share processors; communication between these processes will be implemented by copy operations. If we again consider the transputer implementation these operations require the involvement of the CPU. On the T4XX/8XX the cost is 2 cycles per word, the T9XXX is 1 cycle per word since the CPU can perform one read and one write in a cycle — this is best case with a well behaved memory subsystem and favorable locations for the source and destination. In practice the cost will be higher. Where the messages involved are non–trivial a significant performance penalty is incurred.

As a result of this effect engineers are shocked to discover that, counter to their intuition,

[4]Defined as the total time for the communication operation to complete.

[5]Good spatial and temporal locality is essential for efficient cache characteristics[Prz90].

their programs execute slower than expected[6] since the CPU is now busily copying data. This cost is so high in fact that I saw several cases in my time at INMOS where competent engineers had decomposed sequential programs only to find that the parallel version ran slower or with only marginal improvement over the sequential version.

This problem is at its worse when we consider a single processor implementation; this represents the worst case resource offering of the previous option. On a single device all communication between processes involves a memory to memory copy operation.

As a final option consider the effects of running our general purpose message passing program on a shared memory multiprocessor. The copy effect is now manifest on a parallel machine; for data which would benefit from being exchanged by reference must now be copied. This is one reason there is no interest in Occam for such architectures — again, the Occam model does not map well when processes share memory — an implementation is compelled to copy data that might otherwise be exchanged by reference.

This copy cost I call the *copy penalty* and while it is manifestly obvious on transputers it will be less obvious on other machines with hardware support for block move operations (i.e. via DMA) in the same address space where the effects will be manifest by side effects in the memory subsystem, or dwarfed by high communication start up caused by operating system overheads.

The copy penalty is often unnoticed or dismissed on non–transputer systems because many current communication architectures incur such high message start up latencies in operating systems. This cost in its turn has encouraged programmers to increase message sizes in an attempt to amortize the start up latency — this only serves to increase the copy penalty and is nothing short of a disaster for the future. Programs written today in a generalized message passing style, directed by attempts to amortize communication overheads, will find they hit new problems tomorrow. We can expect to see the start up cost of communication reduce as communication architecture evolves. Unfortunately the memory subsystem bottle neck will not disappear so easily.

Memory cache systems do not help us here. Indeed, a memory to memory copy operation can have devastating effects on cache memory hierarchy, particularly where multiple processors are involved. Messages have poor spatial and temporal locality (by definition). Implementations often require message placement in non–cached parts of the memory space causing performance side effects in memory subsystems either because message data is further copied into cached memory or because variables used in communication must be mapped into a non–cached space.

I do not wish to distort the importance of the copy penalty by contributing too lengthy an explanation — in fact, we shall see later, that distractions caused to the programmer by the message passing model are a far more significant issue. Any benefit resulting from reference exchange as opposed to copy exchange will vary according to machine architecture and application. However, it has been shown in even simple and regular cases to be of benefit.

[6] In many cases much slower than expected.

In the transputer case a 10% increase in effective bandwidth was seen by Hopkins and Welch[HoWe89] of the University of Kent, England, Computing Laboratory in 1989, when (by forced sequence in Occam) using exchange by reference. In a topology aware Data flow solution for systems of linear equations and using a hand crafted balanced pipeline using the exchange of pointers to manage buffers conventionally copied Hopkins and Welch noted only(sic) a 10% increase in effective bandwidth and 50% saving in memory requirements.

It is a general principle in the development of the new proposal described in this thesis that, for efficiency of performance and memory usage, and for uniform performance across architectures, a mechanism should be provided which allows data to be exchanged by reference where possible. Such a mechanism will reduce traffic in the memory subsystem and make the exchange of data in shared memory subsystems a fixed cost independent of the size of the data involved.

On current technology there is an additional overhead caused by operating system network subsystem software routing. This is because hardware architectures do not provide a fully connected network. This we can call the "routing overhead"; i.e., the cost of implementing data routing in software. This routing is often required to allow processes not directly connected by hardware to communicate. I will not be concerned by this overhead here since it is one that I anticipate will be solved by future systems architectures in hardware; i.e., I expect parallel machines and distributed architectures in the future to provide fully connected networks or routing which has no effect on intermediate processors. Evidence to support this contention exists in developing technologies[Pou90, TMC92, INTEL92]. Further, this issue is a general one which I delegate as the direct concern of a system level implementation and is thus not strictly a penalty caused by the programmers model (in that it is a performance issue common to them all on distributed systems). However, many of these same routing subsystems will benefit from the model proposed in this thesis since operating system services very often copy data which can be exchanged by reference. The thesis presents a model which allows this to be simply expressed.

The copy penalty manifest in the Generalized Message Passing model has a *semiotic effect*[7]. Occam programmers when they discover the copy penalty often overcome it by turning the Occam usage checker off. This allows programmers to pass data by reference circumventing the Occam usage rules[8]. In other cases the code is often rewritten (by the user) to provide a balanced process decomposition. In other words, efficient message passing programs can be written with a detailed awareness of the target topology — however the resulting code is highly machine/topology dependent, ergo specialized[9].

[7] Semiotics are explained later in the thesis, but account for the effect a language has upon the behavior of the user, i.e. here the way the engineer or scientist is forced to use the language.

[8] Hopkins and Welch, mentioned earlier in the text, have shown how forced sequencing via a manager/butler Occam process can be utilized in simple cases. Double/triple buffering techniques are commonly used in optimization and conform to Occam rules.

[9] Indeed, INMOS and the Programming Research Group at Oxford University are aware of the problem and extensive effort has been placed in the development of manual transformation tools[Gol88] which allow Occam programs to be algebraically transformed from a general form into a topology specific form.

These are efficiency problems and they have a direct effect upon the behavior of engineers. In practice there is a more remarkable semiotic effect which supports a contention that Occam is too difficult to use in a large project. This problem arises for several reasons, some have nothing to do with the language, none–the–less there is a common thread which indicates a language and model flaw, i.e.

Occam programmers become preoccupied with issues of data distribution.

Forget for a moment configuration issues; i.e., how to map an Occam program onto a target machine, consider only the issue of sharing data between parallel processes working together on a particular algorithm. Occam programmers become side tracked — nay, preoccupied — by issues of routing, buffering[10] and multiplexing data between the components of an algorithm. They expend disproportionate amounts of time on such issues which have nothing to do with the application problem they are trying to solve. As programs become larger the complexity of these issues grow rapidly.

In December 1989 I left INMOS and went to Yale University at the invitation of Prof.David Gelernter. I was intrigued by the activity of the Linda group and the simple model Linda promised. Why? What problems did Linda solve?

Linda's simple interaction model is as tempting and deceptive in its simplicity as the message passing story with two circles and an arrow. Linda promised a solution to the complexity of data distribution by providing a conceptually simple global space in which data can be placed and exchanged. Unlike Occam programmers, Linda programmers need not be concerned with issues of data distribution. Data objects, called *tuples* in Linda, are placed in a globally accessible space called *Tuple Space*, and tuples may be retrieved subsequently by value associative matching.

Linda's distinguishing characteristic is this value associative matching, which amounts to complex address translation in a global shared memory — this turns out to be an operation with unpredictable performance characteristics and this lack of predictability also has a semiotic effect.

Consider the case of the Linda optimizer, write a Linda program or Lindaize an existing C code. You cannot predict the performance of the program, and you will not know the performance until the task is complete — any empirical analysis you perform at any point in the development will likely be invalidated by subsequent changes to the program since the optimizer will almost certainly change strategy as you add new tuple types.

Ok, so the engineer learns everything there is to know about the optimizer — now you have subverted portability. In fact most "successful" Linda programs are either written within a few feet of the implementor of the optimizer or are so parallel and have such granularity that the Linda overhead disappears into insignificance and in such cases pretty much any model would do.

[10] Although buffering is almost entirely a performance related issue in use.

Like Occam programmers, Linda programmers have learnt ways to circumvent these problems and Linda programmers also do what amounts to turning off the usage checker in Occam. At Boeing where some work using Linda has taken place, they use tuples to exchange pointers into the shared address space of a large database. Is this a legitimate use or does it break the model? I would argue that it does break the model, though the engineers who wrote the code simply solved a problem with the tools at hand. From an engineers point of view that is legitimate use.

If it is a consistent programming model we need it should capture all that we seek to express. Our problem is then *to write efficient programs using either Generalized Message Passing or Linda on parallel machines today the engineer is often compelled to break the model.* Nor is this just a simple case of hacking for the last few drops of performance as might be the case on a uniprocessor, for the gains are so significant pragmatics present the engineer with no other option.

The lesson to learn from this precise is that the existing models are not meeting the true needs of the engineering and scientific community.

For completeness, I should not end this passage without mention of the various proposals made for Distributed/Shared Virtual Memory [RaTe et.al.89, RaKh88/50], least the reader think they be forgotten. Such proposals seek to provide a global address space for shared data management. Two central issues exist in such systems

- the expression of locality, and
- the reduction of copy operations within the same virtual address space.

If we consider the implementation of Linda and Generalized Message Passing models in such systems we find that we are confronted with the same set of problems mentioned in the foregoing discussion. The copy penalty remains problematic, perhaps more so. The preoccupation with data distribution remains. An implementation of Linda using such a system may be useful[RaKh88/38] but does not address the previous critique.

I here view Distributed/Shared Virtual Memory as an implementation technology and not as a programmers model; though in the thesis I shall consider global memory models of interaction and the problems associated with them. Distributed/Shared Virtual memory does not provide solutions to the classic critical region problems addressed by Occam and Linda.

Even if such systems provide guarantees of atomicity at some level, alone such memory systems provide no means to express locality — in fairness, such proposals have included proposals for programmers models, usually based on communicating objects or in the case of Ra[RaKh88/38] by implementing Linda. Thus such existing systems supplement the Virtual Memory system with the additional support of a programming model. I consider this a reasonable approach and expect the proposal given here to implemented on such systems. Such implementations will benefit from the features of the proposed model.

The future

Concurrency is being developed at many levels within the parallel machines under development today: *Super–scalar* designs for microprocessors with multiple execution units and long instruction words, *MIMD* and *SIMD* architectures, *shared* and *distributed* memory, *high connectivity* networks.

Some of these technologies are well understood and established, others are new. These latter developments are changing the fundamental nature of the machine. In particular, high connectivity networks, able to provide virtual connections between any two nodes in a network, will inspire an increased interest and utilization in the development of parallel algorithms on multiprocessor MIMD machines, whose processing elements will number many times greater than those on current machines.

It is a simple matter to project where all this might lead, and it is on the basis of such projections that the model presented here has evolved. My projected view sees a progressive integration of the varying parallel architectures into parallel machines which combine the architectural components previously described as, for example, machines where nodes consist of multiple super-scalar microprocessors which share local memory, interconnected to similar nodes via high connectivity networks; a combination of shared and distributed memory. In addition, nodes in such machines will include resources such as vector processing components, SIMD processors and other specialised hardware.

Instruction scheduling compilers will be required to exploit opportunities for parallelism at the instruction level; this is a very low level machine issue primarily aimed at hiding memory access latency, and fine grain data dependency.

Compilers will require much greater sophistication to transform a given program — be it sequential, implicitly or explicitly parallel — into a final form for efficient execution. Many of the criticisms arise here because compilers today perform literal translation, future work arising from this thesis demands a review of non–literal translation by optimization compilers in the light of interaction models.

I first made these remarks in my 1990 *Ease* report — announcements at the time of writing from machine manufactures have confirmed this intuition. In the Thinking Machine's CM5 each processor is surrounded by vector units[TMC92]. We have entered the world of parallel machines. Parallel in every sense, fine and coarse grain, implicit and explicit, it is all important.

Solutions are being sought at many levels to enable effective utilization of future and existing parallel machines.

An important effort is the development of tools which allow the transition of existing applications and skills to this new domain; e.g., Automatic parallel decomposition of FORTRAN code and other languages. Extensions to existing languages, such as Linda, provide simple extensions to current convention which do not require massive re-education of the existing

skill base.

These are *ad hoc* but necessary approaches to the problem of utilizing rapidly advancing machine architectures.

We are confronted with the following questions:

- How do we develop architecture independent algorithms which execute efficiently on a range of architectures (existing diverse architectures and the projected integration), taking advantage of the architectural features of each target? For example by limiting copy operations to communication between disjoint nodes.
- How may a common view of system resources on parallel machines (including specialised hardware components) be developed and simple access mechanisms be provided?
- How do we translate programs onto parallel machines to provide the most effective use and reduce overheads?
- How can failure and error be handled on parallel machines uniformly?

Caveat

To a degree the design of programming languages is a subjective affair. Occam was criticized extensively for its idiosyncratic style. The language presented here tries to be a little less idiosyncratic and to make allowances for the training of the existing engineer. For example, the primitives are predefined in a procedural form in the language in recognition that many USA observers disliked the query (?) and bang (!) notation; from a design point of view however it is desirable to maintain a clear notational distinction between procedural abstraction and fundamental primitives. In any case such aesthetics often determine the language of favor.

Chapter 2

Process Interaction Models

A Process Model describes a program as a collection of behavior patterns called "processes". A Process Interaction Model describes the means by which processes interact; i.e., the means by which processes synchronize and exchange data.

In this chapter I discuss Process Interaction Models; including the issues of process synchronization and data exchange. Several forms of process interaction concepts are currently in existence and these decompose into three categories:

- **communication** — the direct assignment of a value yielded in one process to a variable defined in another.

  The reader may be tempted to believe broadcast and buffering have been ignored in the above statement, this is not the case, a buffer is a simple process, an intermediary, that often implements (an indirect) nonsynchronized communication between two processes. A broadcast is similarly a process, again an intermediary, that implements multiple communications from a single source.

- **global memory shared data** — the extension of existing sequential models where defined variables are specified to be accessible to several or all of the processes composed in a program, and

- **logically shared data structures** — where shared data structures are distinctly defined and adopt characteristics (such as write order) not found in the global memory model.

It is not my aim here to provide a comprehensive rendition of historical and current interaction models. Such surveys are readily available and will not be improved by repetition here. For such perspectives I direct the reader to the excellent recent publication of Prof.Gregory Andrews book "Concurrent Programming: Principles and Practice" published by Benjamin/Cummings[And91]. This book provides an excellent and detailed overview with very good historical notes.

The purpose of this focused review is first to identify interaction models as a single issue in parallel processing; thus, I highlight that what many perceive as distinct models (e.g., barrier synchronization, communication and shared data structures) are, in fact, classes of a single model. The Interacting Process Model provides a common process model, mechanisms for synchronization and data exchange (sharing) between processes. In addition it is important to clearly identify issues effecting performance and functionality.

In this chapter I focus on fundamental characteristics of process interaction models. In the following chapter I shall discuss the implementation of these characteristics and, in particular, the focus is on three existing systems with some pedigree: Semaphores and global shared memory, Occam — an example of the direct communication method, and Linda — the most widely understood advocate of the logically shared data structure model. In the following chapter I consider some implementation requirements of these models in a manner that encourages a comparative assessment of the issues for data exchange in each model.

My objective is to lead the reader to a natural conclusion compatible with the views presented in the introduction; thus, I seek to persuade the reader that existing programming models for parallel machines address several issues badly. I will detail these inadequacies further in a following Critique and will present solutions later.

2.1 Processes and termination

All of the models considered allow for the explicit creation of processes; thus a program is a set of behavior patterns that interact via some interaction model. This set of processes may have been directly described by the programmer or may have been generated by a higher level compiler — such as a functional language compiler or a compiler which automatically decomposes FORTRAN — the origin of this decomposition is of no concern to us here.

In this section we consider processes and in particular the collective termination characteristics of process compositions.

There are two fundamental forms of process synchronization on termination in Interaction Models. Simply, process compositions either synchronize with all the other components of the composition or they do not.

Synchronized process termination assures that no subsequent process will continue before a composed set of processes has terminated. This is commonly called *barrier synchronization*. Barrier synchronization is present in many parallel models, such synchronization is an interaction between all the processes in the composition. Each process terminates only when all other processes in the composition have terminated and this requires the exchange of termination information between the processes in the composition.

Synchronized process termination manifests a synchronous process model; i.e., the barrier provides a step marker such that composed processes appear to have the same duration — though they, in fact, may not have.

Process termination without synchronization provides no assurance concerning the joint termination of a process composition. Each process in a composition terminates according to its natural duration; i.e., upon termination no interaction takes place between composed processes. Process termination without synchronization manifests an asynchronous process model; i.e., composed processes may have differing duration.

Process creation and synchronization interaction have long been supported by the mechanisms

- `Fork` — start a process, and
- `Join` — complete a forked process.

These mechanisms have proven to be the fundamental mechanisms of process creation and synchronization and continue to play a major role in implementation — although they are disappearing from the programmers model for higher level abstractions.

The primitive

$$\texttt{Fork}\ L$$

allows control flow to pass to the program statements identified by the label L (like "go to") but also allows the flow of control to continue to the program statements following the Fork; thus, establishing two "threads" of control. The primitive

$$\texttt{Join}$$

as the name suggests, brings together two such threads of control, by combining Forked threads.

In conventional multiprogramming `Fork` and `Join` combine to provide process creation and process termination synchronization; i.e.,

$$\begin{array}{rl} & \texttt{count} := 2 \\ & \texttt{Fork}\ L1 \\ & P \\ & \text{go to } L2 \\ L1: & Q \\ L2: & \texttt{Join count} \end{array}$$

creates a process labelled $L1$ whose behavior is described by Q, while a process described by P continues. This process pair is recombined by a `Join` statement that allows subsequent processes to begin only when `count` $= 0$; each instance of the `Join` instruction has atomically decremented `count`, thus `Join` provides an implementation of synchronized termination discussed in the previous section.

A version of `Fork` is still used today in the UNIX operating system. UNIX `Fork`, however, does not require a label and no `Join` equivalent exists. Library functions enable a process

to identify whether it is a child or parent process. A `Fork` instruction makes a complete copy of the memory allocated to a process and creates a new process that utilizes the copied "environment". The use of this copied environment by the newly created process avoids the dangers of data sharing (which I shall discuss in the following section). Both the child and the parent process continue execution from the point the `Fork` instruction was called. Unfortunately, because the environment is copied, UNIX `Fork` is an expensive operation and not to be used for the creation of non–trivial processes.

Rapidly, `Fork` and `Join` evolved into higher level compositional constructions allowing a structured process model to be considered; i.e.,

$$\texttt{cobegin}\ P;Q\ \texttt{coend}$$

provides a composition of the processes P and Q and is similar to that found later in Occam. The semantics of this composition can be implemented by the previous example of `Fork` and `Join`.

The construction of processes in this style was first proposed by E.W.Dijkstra and adopted by Hoare as the basis for the parallel construct in CSP.

The composition,

$$P\|Q$$

in Hoare's notation, behaves like P and Q concurrently and terminates when both P and Q terminate. Hoare rationalizes his choice in the following discussion ([Hoa85], page 226)

> "One great advantage of this structured notation is that it is easier to understand what is likely to happen [compared with `Fork` and `Join`], especially if the variables used in each of the [processes] are distinct from the variables used in the other ... In this case, the processes are said to be "disjoint", and (in the absence of communication) the concurrent execution of P and Q has exactly the same effect as their sequential execution in either order
>
> $$\texttt{begin}\ P;Q\ \texttt{end} = \texttt{begin}\ Q;P\ \texttt{end} = \texttt{cobegin}\ P;Q\ \texttt{coend}$$
>
> Furthermore, the proof methods for establishing correctness of parallel composition can be even simpler than the sequential case. That is why Dijkstra's proposal forms the basis for the parallel construct in [CSP]."

If we look at the case of processes without terminating synchronization we can follow Hoare's rationale more completely and find a means to address his concern. Restating the above, if P, Q and R are disjoint then

$$P;Q = Q;P = P\|Q,$$

where $P;Q$ denotes the sequential composition of P and Q. I further introduce the notation

$$//P$$

to mean the process creation (i.e., `Fork`) of P we find that

$$P \| Q; R \neq //P; //Q; R.$$

The distinction is in the terminating synchronization; i.e.,

$$//P; //Q; R = P \| Q \| R,$$

and we find that

$$P; //Q; R = P; R \| Q.$$

In processes that are not disjoint (i.e., they interact in some way) there is synchronization at the start of a forked process that defines the environment the created process inherits from the behavior of the creating (parent) process up to the process creation.

2.2 Global memory interaction models

Shared memory programming was the first practical model of parallel (though more commonly called "multi") computing. Global Memory interaction models extend conventional single address space programming models by adding mechanisms for process creation and synchronization (described in the previous section). Data exchange mechanisms are absent from traditional global memory models since all processes can access conventional variables that are within scope.

Global Memory models do, however, need to provide synchronization mechanisms to permit exclusive access to variables. These mechanisms manage the well known "critical region" problem. These problems are widely known and discussed and I refer the reader to the literature for deeper discussion. These problems have long been dealt with in operating system literature. Peterson and Silberschatz's classic text on "Operating System Concepts"[PeSi85] still provides a good overview of the issues.

2.2.1 Semaphores — conditional precedence

Semaphores were first introduced by Dijkstra[Dij65] as synchronization operations to manage the critical region problem.

A critical region is a process whose actions upon a variable set are exclusive; i.e., concurrent access by another process is forbidden since such access may produce a conflict. Semaphores provide the synchronization mechanism that permits exclusive access to be implemented.

As an example consider the deletion of a record from a linked list in a parallel system with a common address space. This problem will be of interest to us later when considering a shared process scheduling list. Each entry in the list has a next pointer pointing to the

following entry, and the list is implemented by a front and back pointer. The front pointer points to the first entry in the list; the back pointer points to the last entry, the behavior

```
back->next = new_entry; /* update pointer in the last entry */
back = new_entry;       /* update back pointer */
```

adds an entry to the list, and the behavior

```
entry = front;          /* get pointer to entry and */
front = front->next;    /* delete entry from list */
```

deletes an entry.

A problem will arise in both cases if we consider what may happen if two processes attempt to add or delete entries to the list at the same time. Consider a deletion where two processes simply interleave the assignments required for deletion:

```
P0: entry = front;
P1: entry = front;
P0: front = front->next;
P0: front = front->next;
```

a process reads the value of the pointer to the first entry but before the front pointer is updated a second process reads the same front pointer. Subsequently the two processes each update the front pointer. Now, by this description, both processes point to a single deleted entry while two entries have been deleted, and this is certainly not the desired effect. We need to ensure that the two assignments required to implement the operations we have defined occur without conflicting interruption. In a single processor multitasking system it is sufficient to ensure that no interrupt (e.g., a time slice) can occur during the update.

The transputer instruction set solves this problem by allowing interrupts only at well defined points in the instruction stream, such as at a loop end[1], and was designed with the above consideration in mind for uniprocessor multitasking.

Unfortunately the issue is a little more problematic when multiple independent processors share the same address space; and this is where the use of a semaphore comes in.

A semaphore S is an integer variable that, after assignment of an initial value, can only be accessed by the atomic operations $\mathcal{P}$ and $\mathcal{V}$. The classic definitions of $\mathcal{P}$ and $\mathcal{V}$ behavior are[2]

$$\begin{aligned}\mathcal{P}(S) &= \texttt{while } S \leq 0 : \texttt{skip} \\ & S := S - 1\end{aligned}$$

[1] Given today's super scalar pipeline architectures loop ends will not always be such a good choice for performance reasons. Many small loops are able to utilize the pipeline very efficiently; e.g., for like vector operations. A time slicing scheduler would have a rather profound effect on the performance of such operations.

[2] The *busy* nature of the definition of P is unimportant here, needless to say, the nonbusy implementation has the same behavior without the performance hit of continuously looping.

$$\mathcal{V}(S) \quad = \quad S := S + 1 \ .$$

Conceptually semaphore embodies a fundamental notion of synchronization, simply, a flag. A process wishing to enter a critical region waits on a flag value before proceeding; just as a train waits on a rail side signal before proceeding. Thus, if `S` is a semaphore that protects the list in the previous example then to allow the safe deletion of an item from the list our delete operation must become

```
P(S);                   /* wait until it is safe to proceed
                           and set semaphore */
entry = front;          /* get pointer to entry and */
front = front->next;    /* delete entry from list */
V(S);                   /* reset semaphore */
```

Other processes must cooperate; i.e., they too must use $\mathcal{P}(S)$ and $\mathcal{V}(S)$ to access critical variables protected by S. Thus concurrent processes are excluded from access to critical variables by surrounding the statements (the *region*) in a process that accesses them with $\mathcal{P}(S)$ and $\mathcal{V}(S)$ (figure 2.1).

declare semaphore $S = 0$

cobegin
$(\mathcal{P}(S); P; \mathcal{V}(S))$
$(\mathcal{P}(S); Q; \mathcal{V}(S))$
coend

Figure 2.1: Semaphores provide guards for the implementation of critical regions; enforcing sequential synchronization. Here, although composed concurrently, P and Q are forced to be sequential since only one can succeed past $\mathcal{P}(S)$ before it must wait an instance of $\mathcal{V}(S)$ to reset the semaphore.

Semaphores work well as long as the programmer observes the discipline of $\mathcal{P}$ and $\mathcal{V}$ around each critical region. However, if a $\mathcal{P}$ or $\mathcal{V}$ is neglected or placed in the wrong order chaos will result. Such errors can be non–obvious and difficult to detect.

Barrier synchronization (i.e., `Join`) between processes is defined by the semaphore notion also. If the semaphore NP is set to indicate the number of processes in a parallel composition then `Join` is simply the sequence

$$\mathcal{V}(NP); \mathcal{P}(NP); \mathcal{V}(NP).$$

The first V instance counts the number of terminating processes, each P will terminate when all the processes terminate, and as each terminates it permits the termination of each of the

other processes. We can rewrite our earlier example that used `Join` as

```
      declare semaphore NP := −1
      Fork L1
      P
      go to L2
L1 :  Q
L2 :  V(NP)
      P(NP)
      V(NP).
```

2.2.2 Critical regions

Critical regions can be specified as a higher level construct implemented by semaphores. This has the significant advantage that the semaphore operations are automatically placed around the region and in the correct order. Given P and Q as the critical regions acting on the variable set G (figure 2.2) a composition of processes P and Q may be sequentially composed in either order (i.e. $P;Q$, or $Q;P$) but not concurrent.

$$\texttt{region}(G) : P$$
$$\texttt{region}(G) : Q$$

Figure 2.2: Critical regions: a composition of the regions P and Q acting on G must be sequential.

Critical regions were first introduced as a language construct by Brinch Hansen and Hoare[Hoa72]. Hoare also introduced the notion of a conditional critical region

$$\texttt{region}\ G\ \texttt{when}\ B : P$$

where B is a boolean expression. P is performed only if B is true; otherwise, the region is restarted.

Again, semaphores prove to be the primitive notion. Regions are simply abstractions for

$$\texttt{region}(G) : P = \mathcal{P}(S); P(G); \mathcal{V}(S)$$

where S is the semaphore that guards the usage of G. Regions possess the additional advantage that the compiler can simply spot and check the usage of G; i.e., ensure that G is not used outside of a region.

Conditional Regions are considerably more complex to implement since they require that the condition be tested not only on entry to the region but also as each releasing process terminates since that process may have cause the condition to change. Even so, it is the semaphore that is the principal component of the implementation ([PeSi85], page 380).

2.2.3 Monitors

Monitors were later developed by Hansen and Hoare[Hoa74] as a further abstraction of critical regions. By Hoare's account ([Hoa85] page 228) Monitors were inspired by the class concept in SIMULA 67, classes themselves being a generalization of ALGOL 60 procedures.

A *monitor* provides safe access to a set of abstract data types by parallel processes. Monitors are very much like class objects in the object oriented model; providing a guaranteed mutual exclusion, for example, a monitor

```
monitor count
var n
proc *up{n := n + 1}
{n := 0; #; print n
}
```

provides an operation `up` that increments a variable, # is replaced in an instance of the monitor by the body of the specified procedure; i.e.,

```
instance count P
```

being equivalent to

```
monitor count
var n
proc *up{n := n + 1}
{n := 0; P; print n
}
```

within the body of P no reference can be made directly to n but the starred procedure `up` may be called. Monitors provide even greater containment than regions by providing stronger locality; making the job of the compiler in identifying shared variables even easier and enabling the implementation to safely place semaphores to protect them.

Monitors alone do not though provide all the synchronization required (see Peterson [PeSi85] for a more complete discussion) and thus condition typed variables were introduced. These provide a further form of synchronization which allow instances of a monitor to be descheduled (placed in a dormant state) or rescheduled (woken from a dormant state). Given a condition variable C the action

C.`wait`

deschedules the current monitor,

C.`signal`

reschedules a monitor previously descheduled by C.`wait`.

Condition typed variables go beyond the semaphore and provide mechanisms to directly manipulate the scheduling characteristics of an implementation.

None–the–less, as in all the above cases, it is the notion of conditional precedence that is fundamental; i.e., in essence the semaphore. Region constructions simply provide a higher level abstraction based upon these primitives. Similarly, monitors provide a high level programming abstraction which guarantees consistent access. None–the–less maintaining consistency (i.e., sharing of data) is complex in all the above cases; very often leaving the consistency issue completely in the programmers hands. In programming any non–trivial project this complexity is difficult to manage and error prone.

2.2.4 Threads and semaphores on the Encore Multimax

In this section I take a brief look at an existing system on a shared memory multiprocessor. The system is Encore Parallel Threads (EPT) based on work done at Brown University.

EPT provides Monitors, Semaphores, Threads and Micro Threads. A Thread is a `Fork` like process except the user can specify the portion of the environment to be copied. In practice very little of the parents environment is copied and thus remains directly accessible to subsequent Threads. In addition a certain efficiency is gained by the absence of memory bounds checking. The programmer explicitly states the stack size to be associated with the Thread and is expected not to exceed this limitation.

Micro Threads are designed to implement only fine grain parallel processes such as those forming the body of loops. Micro Threads are pro–active in their scheduling; i.e., they constantly loop checking for activation. They consume processor time while pending.

EPT is implemented as a C library the Thread calls of which are

THREADcreate	–create a new Thread.
THREADcurrent	–id of the current Thread.
THREADgo	–initialize the Thread system
THREADreschedule	–start next process.
THREADkill	–terminate a specified Thread and descendants.
THREADreturnvalue	–return the value of a terminated Thread.
THREADjoin	–the join instruction.
THREADreadytorun	–test for a waiting Thread.
THREADcreatequeue	–create a new Thread (wait) queue.
THREADrun	–add Thread to run queue.
THREADwait	–add Thread to a specified wait queue.
THREADdequeue	–delete Thread from its current queue.
THREADnext	–next Thread on a specified queue.
THREADrunning	–head of run queue.
THREADstartclock	–start preemptive scheduling.
THREADstopclock	–stop preemptive scheduling.
THREADfreeze	–generate a signal and start a new thread.
THREADfrozen	–test specified Thread is dormant.
THREADfrozenparent	–returns id of dormant parent of Thread.

Micro Threads have a small set of functions

uTHREADcreate	–create n Micro Threads.
uTHREADgo	–start Micro Thread group.
uTHREADjoin	–Join operation.
uTHREADkill	–delete Micro Thread group.
uTHREADpark	–stop Micro Thread group.
uTHREADready	–ready Micro Thread group.

Semaphores are simply what we expect

THREADseminit	–initialize semaphore.
THREADpsem	–semaphore P operation.
THREADvsem	–semaphore V operation.

Monitors are

THREADmonitorentry	–enter monitor when no other Thread is active in it.
THREADmonitorexit	–exit monitor.
THREADmonitorinit	–create a monitor.
THREADmonitorsignalandexit	–generate signal and exit.
THREADmonitorsignalandwait	–generate signal and wait.
THREADmonitorwait	–wait for a signal.
THREADmonitorwaitevent	–wait for a specific signal.

The purpose of this rendition is not to give a detailed specification of the Encore Parallel

Threads package. The purpose is simply to illustrate the degree of complexity the engineer must confront when building systems on many existing machines. Even so, the rendition given here is simplified. Some subset of these functions do prove useful as a target for higher level abstractions (such as that specified later in this thesis) but the whole is a shocking mess to program with. I would hope that to scan this plethora of functions will convince the reader of the need for simple high level models of programming multiprocess applications.

The value of high level models is often questioned on the basis of performance cost. However, such models *can* be constructed to incur no additional performance cost. In the following sections we shall consider several high level models which would at first seem to contradict this statement; Occam: a model that is in many respects efficient but introduces complexity into the programming model and lacks practical functionality, and Linda: providing a degree of functionality but with weak locality and at the cost of efficiency.

None–the–less, the later proposal will present a new model (*Ease*) which is as efficient as Occam with the functionality, and more, of Linda. What is more, the reader should understand why this is so.

2.2.5 Distributed Shared Memory

The advent of distributed memory computing has rather complicated the issues surrounding the global memory model. On such machines the ability to share conventional variables is not (without support) possible. Conventional implementations of semaphores and, consequently, `Join` are difficult on such machines for this reason. Conventional implementation is dependent on access to a counter variable which must be shared between many, perhaps distributed, processes. Proposals exist that extend virtual memory concepts across distributed processor machines[RaKh88/50, RaTe et.al.89]. These provide virtual address spaces in which the Global Memory model can be implemented. These concepts are of interest to us here because they often provide mechanisms to assist in the management of consistency.

The most referenced proposal is probably Kai Li's[Li89] "shared virtual memory". In this system virtual memory pages can be distributed among many processors (processes). Virtual memory pages can be arbitrarily duplicated by read operations provided that all such copies remain read only. A write operation to a page ensures that all such copies are first eliminated; thus consistency is assured. Li's model is, in fact, a slight variation of the Berkeley protocol for multiprocessor cache consistency. However, this simple mechanism can be effectively used to simulate a global memory model on a distributed machine and it has been further elaborated in operating system circles. In particular in Mach[RaTe et.al.89] and Ra[RaKh88/50].

At a low level I suspect we shall see increasing support for this model on distributed machines. The primary interest of programming models built upon this architecture is the expression of locality and the reduction or elimination of copy operations in the virtual address space. My later proposal will provide solutions to both these issues.

2.3 High level process models

One of the clearest exponents of a high level process model is Occam. Occam in turn derives its process model from CSP and what is said in the following of one in general applies to the other. In this section I briefly reiterate the notion of high level processes suggested in the earlier section, outline why the reiteration is necessary and highlight the advantages of a well defined process model.

In UNIX systems and those parallel machines supporting UNIX (nearly all such machines at this time) there is constant reference made to Daemons, Programs, Tasks, and Threads; often with so–called "lightweight" and "heavyweight" characteristics. To the community of engineers educated in the European engineering culture of the 1980's, a process is all of these things.

An operating system is a process, which in turn consists of processes providing support to other processes — often called "user programs". A vending machine is a process, which interacts with another process called a "human being". So, to clarify what has gone before, we speak of processes and their composition; very often using the notation P and Q to denote processes. In CSP P composed with Q in sequence is denoted

$$P; Q.$$

P composed with Q in parallel is

$$P \| Q.$$

This is a subtly distinct notion from that of "process creation" fostered with `Fork` in UNIX or Threads of control. Although, as I have shown earlier, simple process creation can be simplified to a form equivalent to CSP parallel composition.

To further clarify this let us look closer at the Occam process model. In Occam, assignment, input and output comprise the most primitive processes. Programs are built from compositions of processes formed by construction; i.e., larger processes are built by combining smaller processes. A construction builds a process which is of one of the forms

`SEQ`	sequence
`IF`	conditional
`CASE`	selection
`WHILE`	loop
`PAR`	parallel
`ALT`	alternation

A sequential process is built by combining processes in a sequence, conditional or selection construction. Loops are constructed by combining processes in a `WHILE` construction. Concurrent processes by parallel and alternation constructions.

The constructions `SEQ`, `IF`, `PAR` and `ALT` can all be replicated; i.e., specified to duplicate the constructed process some number of times.

Occam uses indentation to denote construction grouping so a sequence

```
SEQ
  P
  Q
```

composes two processes `P` and `Q`. `Q` is executed if and when `P` terminates successfully. A replication

```
SEQ i = 0 FOR N
  P(i)
```

creates `N` copies of `P` each with a distinct index constant `i`.

A conditional combines a specified number of processes each of which is guarded by a boolean expression. The conditional evaluates each boolean expression in sequence. If a boolean is true the associated process is performed, and the construction terminates. If none of the booleans are true the construction does not terminate; i.e., no process later in a sequence containing the construction is performed.

```
IF
  b
    P
  c
    Q
```

is a conditional which performs `P` if `b` is true, and `Q` is `b` is false and `c` is true, and stops otherwise.

Like sequence, a conditional may be replicated. A replicated conditional constructs a number of similar choices; i.e.,

```
IF i = 0 FOR N
  b(i)
    P(i)
```

creates a number of similar choices each guarded by a boolean expression `b(i)`. The replication may be expanded to show its meaning; i.e., where `N` is 2

```
IF
  b(0)
    P(0)
  b(1)
    P(1)
```

A selection combines a number of options, one is selected by matching the value of a selector with the value of a constant expression.

```
CASE s
  e
    P
  f
    Q
```

is a selection that performs `P` if `s` is `e`, performs `Q` if `s` is `f`, and stops otherwise. `e` and `f` must be distinct values. Case is a special form of conditional.

A loop repeats a process while an associated boolean is true.

```
WHILE b
  P
```

is a loop that performs `P` if `b` is true and repeats `P` if `b` remains true after `P`.

A parallel combines some number of processes concurrently.

```
PAR
  P
  Q
```

is a parallel process combining `P` and `Q` concurrently.

A parallel can be replicated, in the same way as sequences and conditionals. A replicated parallel constructs a number of similar concurrent processes; i.e.,

```
PAR i = 0 FOR N
  P(i)
```

creates `N` copies of the process `P`.

An alternation[3] combines a number of processes guarded by inputs and performs the process associated with a guard that is ready.

```
ALT
  c ? v
    P
  k ? v
    Q
```

[3]The "alternation" in Occam is badly named and should really be called "choice".

is a choice between the inputs `c ? v` and `k ? v`. If the input on `c` is ready, the input then `P` are performed, if the input on `k` is ready the input then `Q` are performed, if both are ready then one of the inputs and its associated process are performed; the choice being nondeterministic.

An alternation can be replicated in the same way as sequences, conditionals and parallels.

```
ALT i = 0 FOR N
  I(i)
    P(i)
```

performs a ready input `I(i)` and then the process `P(i)`.

The strong sense of process should be clear from this description. The distinction between a notion of process composition and the notion of Threads of control often associated with `Fork`[4] should also be clear.

In many ways `Fork` and/or Threads compound sequential thinking, where, process composition compounds parallel thinking. The latter is preferable when constructing programs for parallel machines.

A first order process model with well defined composition has a number of other attractions. Chief among these are true modularity, the inherent parallel nature and openness to mathematical treatment.

All that remains is to answer the question of how composed processes should interact.

2.4 Communication

Communication has arisen in programming models over the past decade.

[4]In many respects, there has been a transatlantic cultural divide over process models. Many USA engineers are unfamiliar with the Occam like process model and perceive of processes in the UNIX context. Similarly many European engineers were unfamiliar with the paradigms of `Fork` and Threads of control discussed in the previous sections. Though this is less true in the more formal communities on both sides of the Atlantic. The reason is historical and has much to do with the success of UNIX. UNIX has only in recent times come to play a significant role in European engineering, whereas there are two decades of pervasive experience with UNIX in the USA.

A brief historical anecdote will prove to illustrate this divide. The first implementation of protoOccam ran under a system know as the UCSD P-System and INMOS made Occam available under this system on the Apple II computer in 1984. It was, still, not at all clear to us in the England at that time that C and UNIX were a force to be reckoned with in the engineering community. In 1985, when the first transputers became available, development systems (with Occam and no C) still ran on top of the UCSD P-System on Sage computers. It wasn't until 1987 that INMOS began to support the IBM PC as a development platform and this was still (in retrospect) hopelessly out of touch with what was happening in the engineering community in the USA. In fact it wasn't until 1989 that UNIX began to see support at INMOS and then only with tools and compilers that ran on transputer boards plugged into UNIX work stations. By 1989 all the opportunities in the USA had passed and the market was lost, perhaps forever, to INMOS. As a measure of the perceived priorities a realistic C based development system has only come from INMOS in recent years.

The advent of communication in high level programming coincides with the advent of distributed memory machines. Communication models primitive actions in distributed computing and its generalization was a natural step. Communication between computing nodes is a fundamental primitive on distributed machines, and as such can be considered at the same level of abstraction as `Fork`, `Join` and Semaphore.

Just as the shared global storage programming model was suggested by the hardware of the then existing machines, so too communication is suggested by the evolution of networks and today's communication technology. In the following sections I briefly outline the basic communication primitives considering both synchronized and nonsynchronized message passing.

2.4.1 Synchronized message passing

For simplicity in this section I focus on Occam. Occam is a known and integrated message passing model for programming. Occam message passing is synchronized and point–to–point. In other words communication is via an object shared between only two processes.

Occam is significant in that it is based on the well formed mathematical principles laid down by Hoare in CSP. The central message of Occam is Communication and Concurrency. Values are passed between concurrent processes by communication on "channels". Each channel provides non–buffered, unidirectional point-to-point communication between two concurrent processes. The format and type of communication on a channel is specified by a channel "protocol" given in the declaration of a channel. Two actions exist in Occam to perform communication on a channel, they are: input and output.

An input receives a value from a channel and assigns the received value to a variable; e.g.,

$$c?v$$

receives a value from the channel c and assigns the value to the variable v. The input waits until a value is received. The value input must be of the same data type as the variable to which it is assigned, otherwise the input is not valid.

An output transmits the value of an expression to a channel; i.e.,

$$c!e$$

transmits the value of the expression e to the channel c. The output waits until the value has been received by a corresponding input.

Point–to–point communication, as found in Occam, benefits significantly from simplicity. Point–to–point communication is both

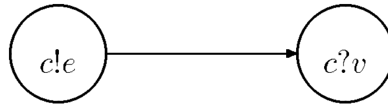


Figure 2.3: Message passing: a deceptively simple idea to sell. A process inputs a value output by some other process.

- simple to implement, and
- simple to explain.

The advantage of simple implementation has been exploited on the transputer [May88]. Consider just how simple a notion point–to–point communication is (figure 2.3). Channels are synchronized such that an outputting process will wait until the inputting process is ready to receive the output, an inputting process will wait until the outputting process is ready. This synchronization characteristic Occam message passing allows the exchange to be implemented without buffering. The destination space for the message is present in the receiving process.

2.4.2 Non–synchronized message passing

Non–synchronized message passing appears in several languages and operating systems (refer to Andrews[And91] for a detailed historical analysis of such systems). Like Occam, such systems provide an input and output primitive; however, an output does not wait for a corresponding ready input.

Essentially, non–synchronized communication provides a buffering mechanism. Such buffering can be easily implemented, using the concepts introduced so far, by the addition of an intermediate process; i.e., simply

$$\mathrm{in}?v; \mathrm{out}!v$$

a process that copies its input to its output is a simple buffer and this may be arbitrarily extended.

Very often such models are implemented as queues. Queuing and intermediate processes also allow several other possible implementations of message passing. Most notable being multiple destination, multiple source paradigms.

The primary interest of simple communication is its low level primitive nature. In practice this advantage suffers from the pitfalls of all similar primitives (such as “go to” and

semaphore); in this case leading programmers, as the introduction revealed, to a preoccupation with data distribution in the large.

2.4.3 Express: an operating system expedient

Express is a commercial product[5] which derives from work begun at CalTech. It seeks to provide supplementary facilities to a familiar OS environment at the operating system level for today's engineers programming parallel machines — as such it does an admirable job; though it is a slight diversion from the main theme of this chapter which is aimed at fundamentals in high level programming.

In essence Express provides operating system extensions to conventional languages such as C and FORTRAN. Here I shall detail the support provided for message driven processes. Express provides support for internodal communication on distributed memory processors and for the creation of processes which act upon incoming messages. These features are of particular use in the implementation of shared data models (and, indeed, are used in the implementation of the later proposal).

The three message passing operations I wish to highlight are

- exread — read a message of from a specified node of a specified type, and
- exwrite — write a message to a specified node of a specified type.
- exhandle — message driven scheduling,

In the following discussion I have taken a few liberties with the respective functions, their parameters (some of which may be pointers) and use of C syntax in the cause of clarity.

`exread(message, maxlength, source, type)` is a function which returns the length of a message received from `source` and is of type `type`. The incoming message is truncated, if necessary to `maxlength`.

`exwrite(message, length, destination, type)` is a function which returns the length of the message successfully sent to `destination`. The `type` and desired `length` are specified.

`exhandle(function, source, type)` executes `function` upon receipt of a message of type `type`. The incoming message is passed to the specified function as a parameter.

An excellent example of the usefulness of this facility is given in the Express manual[Par88] and in the following I present an abbreviated version of it.

[5]Available from Parasoft Corp. 27415 Trabuco Circle, Mission Viejo, CA 92692, USA.

The example illustrates the implementation of a global read only memory; but serves to illustrate how distributed shared data management can be implemented.

Each node executes a function

```
exhandle(memory, any, read)
```

The type of the incoming memory request is a structure

```
struct memory_request {
     int address;
     int length
};
```

which specifies the desired address and the length of contiguous memory required. The

function `memory` then is

```
memory(struct memory_request *request, l, source, t)
{
    exwrite(request->address, request->length, source, ack);
}
```

where `ack` is the type indicating a satisfied request.

A function to read this global memory is then

```
read(node, address, length, target)
{
    struct memory_request request;

    request.address = address;
    request.length  = length;

    exwrite(request, sizeof(request), node, read);
    exread (target,  length, node, ack);
}
```

Why introduce such an example just here? Firstly it is important to emphasise that message passing is a low level primitive on machines with distributed memory and that implementation contrasts with the higher level programming model. Secondly it is useful to understand that an elaboration on the above theme allows these higher level models to be implemented with reasonable efficiency. As has already been stated: such an elaboration is used in the implementation of the later proposal on such machines.

2.5 Logically shared data structures

I now consider the shared data structure (space) model, its features and advantages.

Logically shared data structures differ from conventional global memory models by providing a set of basic primitives which distinguish such structures from local variables. In addition, shared data structures exist in a logical *space* which have characteristics not common to conventional variables. In particular an assignment primitive may not replace the previous value associated with the structure but simply add a value to one of the possible values to be yielded by it.

For simplicity I focus on Linda. Linda is not representative of all Shared Data Models. However, Linda is well known and widely used or considered. In many respects it represents the opposite end of a spectrum in which Occam exists. It was among the first to consider a shared data structure model as defined above and in many ways has been the innovator behind many recent proposals in the field (Orca[Bal89], Swarm[RoCu90] and others) though the formal basis of Linda is weak by comparison to Occam.

The central message of Linda is Coordination and Concurrency. A message which is subtly distinct from the message of Occam.

Linda is based on the concept of "generative communication", which attempts to unify the concepts of process creation and communication. Linda was first described by David Gelernter [Gel85], and the first implementation is described by Nick Carriero [Car87]. The Linda language combines with some associate language, typically a conventional sequential language such as C, to form a language for the expression of parallel algorithms. The associate language provides the semantics of computation while Linda provides the semantics for concurrency and communication.

Linda utilizes a concept known as *tuple space*. Tuple space is a global associative memory, which stores objects called *tuples*. A tuple consists of a sequence of typed fields; i.e,

```
("foo", 6, 23.5)
```

is a tuple which is a sequence of values; a string `foo`, an integer value `6`, and a floating point value `23.5`. It is distinct from the following tuples

```
("foo", 6, 23.5, 32.5)
(6, 23.5, "foo")
(4, 5)
```

These are *passive* tuples; i.e., passive data objects. A tuple may also contain fields which are processes evaluated subsequent to entering tuple space. These are known as *active* tuples. It is easier at this stage to think of tuple space as a bag of objects. Linda provides four basic primitive operations which act upon tuple space:

`out(t)` to put tuple `t` into tuple space (i.e. to put an object into the bag).

`in(t)` to get tuple `t` from tuple space (i.e. to pick an object from the bag).

`rd(t)` to read tuple `t` in tuple space (i.e. to look at an object without removing it from the bag).

`eval(t)` to evaluate tuple `t` (i.e. put an object into the bag for evaluation).

`out(t)` and `eval(t)` place a tuple (`t`) into tuple space and then terminate. `in(t)` removes some tuple `t` from tuple space and then terminates. `rd(t)` reads the value of some tuple `t` and then terminates.

This definition naturally implies that if there is no tuple which initially matches `t` present in tuple space then the primitive `in` or `rd` will not terminate until it acquires a tuple `t` subsequently added to tuple space.

`eval(t)` acts like `out(t)`, except that `t` is evaluated subsequent to its entry to tuple space and will typically transform into a passive data tuple, for example

```
eval (P(), Q())
```

creates processes `P()` and `Q()` which are placed in tuple space and are evaluated concurrently. `P()` and `Q()` may themselves interact with tuple space, and leave results (as tuples) in tuple space. If `P()` and `Q()` are functions which return the integer values `6` and `7` respectively, then the *active* tuple `(P(), Q())` will transform into the *passive* tuple `(6, 7)`. Thus is born the term *generative communication*.

Note that the current definitions of the process model vary and are confused about the scoping of free variables (and pointers) in evaled processes. In the C–Linda implementation developed jointly by Yale and SCIENTIFIC Computing Associates `eval(P)` amounts to a specialized fork of `P`.

Tuples have no physical or virtual *address* in tuple space. A tuple is selected by `in` or `rd` by *associative matching*.

Each field of a tuple may contain an *actual* or *formal*, for example, if `N` is a variable of integer type

```
(6, ?N)
```

contains a formal `N` and will match with any of the following tuples

```
(6, 7)
(6, 8)
(6, 1024)
```

The input `in(6, ?N)` will select a matching tuple from tuple space, and perform the actual to formal assignment; if several matching tuples exist in tuple space then an arbitrary selection is made.

The output `out(6, ?N)` will place a tuple in tuple space, and may be selected by an input which has an actual of integer type in the place of the formal, e.g. `in(?I, 11)` or `in(6, 23)`.

More recently Linda has acquired two variant forms of `in` and `rd`. These are the primitives `inp` and `rdp`, predicate functions which test for presence; i.e.,

```
if (inp(t)) found = true
```

`inp(t)` attempts to remove some tuple `t` from tuple space and then terminates. `rdp(t)` attempts to read the value of some tuple `t` and then terminates. In both cases the predicate is true if the function succeeds in its attempt and is false otherwise. In addition, the predicate functions side effect; i.e., they will behave like their non–predicate equivalents if their result is true.

Thus, a shared data structure in Linda is a "Tuple Space" (see figure 2.4). Tuple Space acts as an intermediary in which data structures are created and by which processes can interact.

A tuple space is initially an empty set of values. I shall simplify the by focusing on the three operators that act upon this set.

- `out` — adds a value to the set,
- `read` — reads a value in the set,
- `in` — acts like read but also deletes the value from the set.

This description is a generalization of the tuple space model and the reason it is introduced here will become clear later.

Although some consideration has been given to multiple tuple spaces[6] I have focused on the single tuple space common to those Linda implementations at Yale University. Much work has been done on Linda both in the research community and in industry[Lel90].

The key advantage of shared data structures is their conceptually powerful nature. They provide an alleviation of concern about data distribution between processes and simplify coherency and synchronization.

[6]Indeed, I was involved with such considerations at Yale in 1990 — none of these bought a consensus. However, I would point to the work of Suresh Jagannarthan now at NEC as the most considered of these efforts and note that these same deliberations pointed me in the direction presented in this thesis.

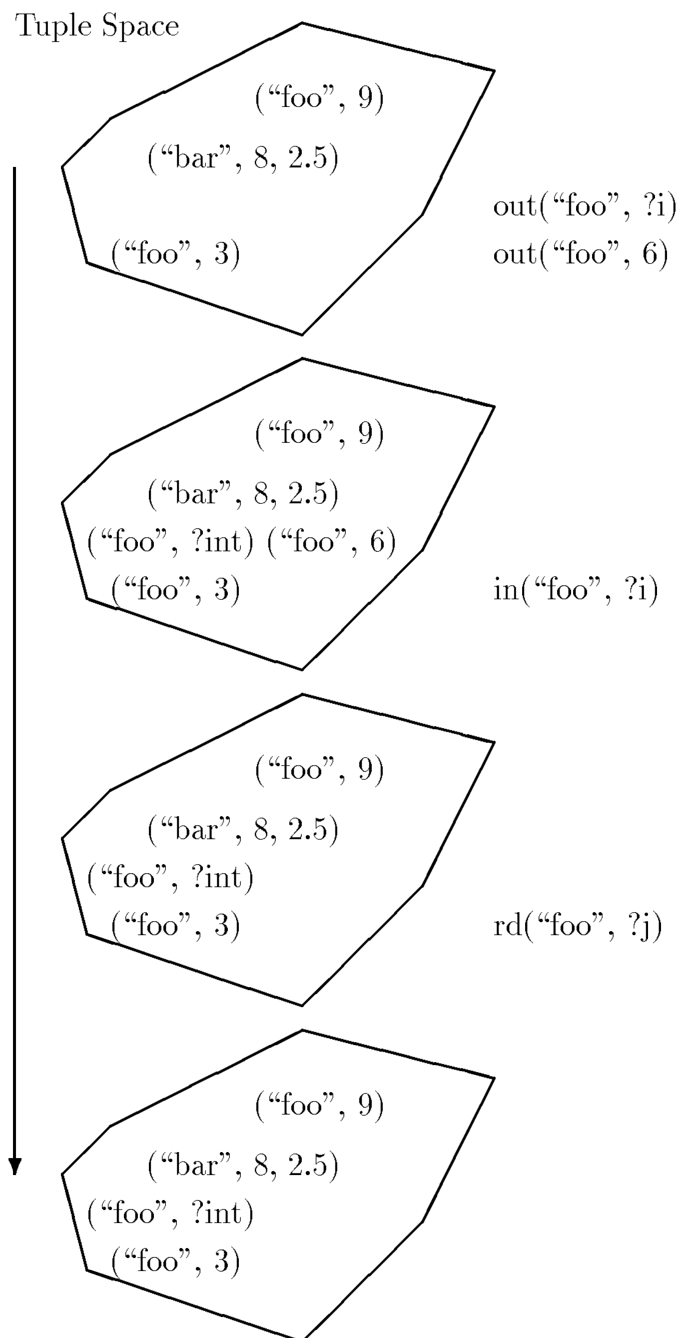


Figure 2.4: Linda: another deceptively simple idea to sell. Here we trace the state of tuple space through 4 changes. The effect of the "in" and "out" primitives are clearly illustrated. As a result of the second state change $i = 6$, "rd" has no affect on the state of tuple space but *reads* a tuple, since selection is nondeterministic $j = 9 \vee j = 3$.

Chapter 3

Implementing Interaction Models

In this chapter I take a look at the efficient implementation of the process interaction model fundamentals we have considered. In each case and on each architecture, there are choices to be made. To focus our discussion and to allow a comparative assessment I consider an efficient implementation that in each case takes a similar approach.

In the following discussion I shall use C as a notational convenience to model processes and the data structures involved in the implementation of interaction; though I shall adhere to no standard and ignore the fact that several of the functions used would, in fact, be very difficult to implement in real C (such as `nextprocess` in the following section). I use C notation simply because I anticipate a broad understanding of it.

Each operation is described in such detail to allow the reader to gain a direct sense of the potential implementation costs involved; such as the number of comparisons, loads and stores involved in each operation.

3.1 Implementation of scheduling

In the following implementations I shall consider a machine with a simple and single process scheduling queue. This scheduling queue is a linked list of process control blocks. Each process control block will contain three pointers for

- source or destination,
- state, and
- the next process.

A process control block is modeled by the data structure

```
struct process {
     data   *source_destination;
     struct pc      state;
     struct process *next;
};.
```

I need not explore the structure of the type `pc`, for it will certainly differ in each microprocessor architecture. It contains the state that needs to be maintained for the successful reinstantiation of a process and at least includes the machine's Program/Instruction Counter.

The scheduling list itself can simply be implemented in several ways (FIFO, LIFO, priority etc..). In the following consideration we simply assume a consistent implementation; e.g.,

```
struct list {
     struct process *pending;
     struct process *tending;
} scheduling_list ;
```

or

```
struct list {
     struct process *pending;
} scheduling_list ;
```

However, in the abstract, I shall take the function

```
tolist(struct list l, struct process *p)
```

to be the atomic function which adds the process `p` to the list `l`, and

```
fromlist(struct list l)
```

to be the atomic function that removes a process from the list `l` and returns a pointer to the removed process control block. In addition, I shall take the function

```
schedule(struct process *p)
```

to be the atomic function that specifically adds `p` to the scheduling list, and the function

```
nextprocess()
```

to be the atomic function that takes the next process from the scheduling list and begins its execution[1]. It is necessary to assume some unit of atomicity in our discussion, that unit will certainly vary from machine architecture to machine architecture, to constrain our discussion to the realms of reason I choose to define all the functions in this chapter as atomic in nature. This atomicity makes the discussion applicable to uniprocessor multiprogramming and closely coupled shared memory multiprocessor multiprogramming.

The issues of process placement and distribution in a distributed memory multiprocessor I shall leave to a higher level decision making paradigm, and I shall not consider the issue of process migration a necessary part of this discussion. I shall also use `PID` to mean the pointer to the process control block of the current process.

The scheduling model illustrated here should hold no surprises. It is in common use and closely resembles the scheduling mechanisms implemented in microcode on the transputer and described in Peterson and Silberschatz[PeSi85].

3.2 Implementing semaphores

Semaphore is the basic synchronization primitive for implementing many of the shared memory paradigms discussed earlier. In this section we take a look at how such primitives can be implemented efficiently.

Given our scheduling mechanism a semaphore can be modeled as a simple data structure consisting of a counter and a process list. Such a structure can be specified by

```
struct semaphore {
    int  count;
    list *processes;
} s { 0, EMPTY };
```

All semaphores being initialized according to this definition.

Semaphores require the implementation of the $\mathcal{P}$ and $\mathcal{V}$ operations specified in the previous chapter — though I prefer a non–busy implementation. These implementations are described in figure 3.1.

3.3 Implementing communication

Let us now consider the implementation of communication,in addition to synchronization this involves data exchange. The following description closely resembles the implementation of

[1]This function is not readily described in C directly and I abstract away from stack maintenance issues in this implementation.

$\mathcal{P} =$

- if the semaphore count is zero
  - decrement the semaphore count.
- if the semaphore count is non–zero
  - decrement the semaphore count,
  - add the current process to the scheduling list, and
  - start the next process.

```
P (struct semaphore *s)

     if (s->count == 0)
        s->count--;
     else {
        s->count--;
        tolist(s->processes, PID);
        nextprocess();
     }
```

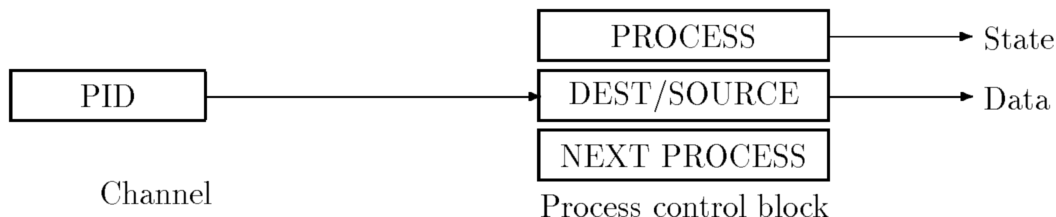
$\mathcal{V} =$

- if the pending process list is empty
  - increment the semaphore count.
- if the pending process list is not empty
  - increment the semaphore count,
  - add a pending process to the scheduling list.

```
V (struct semaphore *s)

     if (s->processes == EMPTY)
        s->count++;
     else {
        s->count++;
        schedule(s->processes);
        fromlist(s->processes);
     }
```

Figure 3.1: Implementation of $\mathcal{P}$ and $\mathcal{V}$.



1. The first process ready saves a pointer to itself in a channel location and stores the message source or destination in the process control block and starts the next available process,
2. the corresponding process retrieves the message source or destination, copies the message, initializes the channel and continues.

Figure 3.2: Point–to–point communication. A pending process.

---

channels on the transputer[INM90], a machine designed to implement the features of Occam.

A communication *channel* is modeled by a simple data structure, a single location in memory, whose value is either the special value `NULL` or is a pointer to a process ready to communicate, such a structure can be specified by

```
struct process *c = NULL;
```

All channels being initialized according to this definition.

A process that outputs on a channel knows

- the channel — identified by the location of the above data structure in memory,
- the message — which is identified by its location in the local memory of the sending process.

As in the semaphore case the outputting process does not know the identity of the concurrent (receiving) process or the destination location of the message. An implementation is shown in figure 3.3.

A similar process (figure 3.4) describes the behaviour of the input.

Two very significant advantages exist for the implementation illustrated,

- simplicity, and
- no buffering.

- if the channel is not "ready"
  - place the ID of the process control block in the channel location,
  - store the message location in the process control block,
  - start the next process.
- if the channel is "ready"
  - retrieve the destination from the inputting process control block,
  - copy the message to the destination,
  - add the inputting process to the scheduling list, and
  - initialize the channel for subsequent use.

```
output (process *channel, data *message)

    if (channel == NULL) {
        PID->source_destination = message;
        channel = PID;
        nextprocess();
    } else {
        destination = channel->source_destination;
        memcpy(destination, message, sizeof(data));
        schedule(channel);
        channel = NULL;
    }
```

Figure 3.3: Implementation of point–to–point output.

- if the channel is not "ready"
  - place the ID of the process control block in the channel location,
  - store the destination location in the process control block,
  - start the next process.
- if the channel is "ready"
  - retrieve the message location from the outputting process control block,
  - copy the message to the destination,
  - add the outputting process to the scheduling list, and
  - initialize the channel for subsequent use.

```
input (process *channel, data *destination)

    if (channel == NULL) {
        PID->source_destination = destination;
        channel = PID;
        nextprocess();
    } else {
        message = channel->source_destination;
        memcpy(destination, message, sizeof(data));
        schedule(channel);
        channel = NULL;
    }
```

Figure 3.4: Implementation of point–to–point input.

Simplicity is a virtue for its own sake, the no buffering requirement means that no demands are made on a dynamic memory management system. Indeed, it is difficult to think of a more elegant and efficient solution. However, problems do present themselves. This scheme does require that both sender and receiver have read and write access to the memory of the respective processes. On systems other than the transputer this would perhaps be difficult, undesirable, or both. However, this issue is a general one, not particular to Occam.

This implementation requires a single address space, but the versions required to cater for memory mapped communication devices are not dissimilar, and again the transputer implementation illustrates this. The transputer uses a few words in the lower address space which map onto the on chip communication links, the same instruction is used whether the channel uses the links or otherwise. The microcode detects the difference and uses the appropriate mechanism to copy the message.

We must now confront a central issue raised during the design of next generation transputers and, indeed, all current parallel systems architecture. The problem is one of how to provide sufficient connectivity to implement a general set of programs.

First generation transputers have only four hardware links able to implement at most four channels in each direction. This is a limiting factor in Occam implementations and would prove so too in our implementation here of the new proposal.

Virtual hardware channels can be provided by system support that multiplex data between nodes. Second generation transputer hardware (if it appears) provides support for multiple channels in the form of a hardware multiplexer. Intel's iWarp device also provides virtual connectivity. Such multiplexing, or virtual communication, support will be required, either in hardware or systems software.

Programming models and implementations must reduce copy operations effecting the memory subsystem, thus the location of buffering is a major systems issue. Hardware can help here by providing operating systems with instructions which enable the construction of various multiplexing strategies. Without going into detail, it is not clear that a hard wired multiplexer of the kind currently being developed at INMOS for the H1/T9000[INM91] can provide the flexibility demanded across systems.

3.4 Implementation and optimization in Linda

Tuple Space operations are not difficult to explain but are less than simple to implement. Linda compilers (such as the C–Linda compiler designed by David Gelernter and Nick Carriero [Car87]) perform several transformations on Linda tuple space operations, which optimize access and implementation of value associative matching and poor expression of locality.

A Linda program is analyzed by the compiler and the tuple space operations are transformed into simpler operations which require little or no run–time matching and implied locality grouping identified.

Linda optimizations take several forms and these are considered in the following section.

3.4.1 Linda: Division of tuple space into distinct subsets

Tuples can immediately be divided into sets based on their structure, i.e. the number, type and order of their fields. We know that tuples of one type structure can never match a tuple of a differing structure. Therefore, inputs and outputs can be distinguished as operations on one of these sets. Further, these distinct sets can themselves be divided, by usage of formal and actual data.

Consider the tuple set of type

$$(\mathrm{STRING}, \mathrm{INT}, \mathrm{INT})$$

It's common in Linda programs to find that one or more of the fields in a tuple of this type structure is a common constant (a constant used in both inputs and outputs of the tuple). In which case, operations on the set of tuples of this structural type can be further divided. Let us assume that the analyzer discovers that in all operations on tuples of this type structure the first field is some common constant. For example, assume these constants are the strings, "foo" and "bar", the tuple set described can be further divided into the subsets

$$(\text{“foo”}, \mathrm{INT}, \mathrm{INT})$$
$$(\text{“bar”}, \mathrm{INT}, \mathrm{INT})$$

tuples of the first type structure distinguished by common constants. Again, since operations on one subset can never match a tuple affected by an operation on a tuple of the other subset we can focus tuple space operations on the relevant sets.

3.4.2 Linda: Implementation of distinct subsets

Let us further assume, for the purposes of illustration, that the types discussed in the previous section are the only types of tuples which appear in a Linda program. The inputs and outputs in the program can now be divided into operations upon these distinguished sets. Such that we now have two distinct sets of type

$$(\mathrm{INT}, \mathrm{INT})$$

and the constant fields can be discarded since their value is now represented by selection

of one subset or the other.

Once analysis of the constants in a tuple is discerned, inspection of the usage of tuples on input and output enables a decision to be made about the method of implementation for each set.

In addition to position and type, each field in a tuple has a further characteristic we have yet to discuss. That is whether the field is an *actual* or *formal* (known as the *polarity* of the field). When matching the respective fields of an input with existent tuples there are four possible polarity combinations to consider

- ? both fields are actual
- $\times$ both fields are formal
- $\surd$ input field is actual and the respective field is formal
- $\surd$ input field is formal and the respective field is actual

where ? indicates a comparison of the fields value must be made for equality, $\times$ indicates that a comparison of the fields always yields the value false, and $\surd$ indicates that a comparison of the fields always yields the value true.

To summarize, in the first case, where both fields are actual values, the fields match if their values are equal. In the second case, where both fields are formal, no match can be made. In the third case, where the input field is actual and the corresponding field is formal, a match is always made. In the final case, where the input field is a formal and the corresponding field is an actual, a match is always made, and provided all the component fields of the tuple match, an actual to formal assignment is required.

There are four possible usage patterns for our given example subset of tuple space, (INT, INT),

- the remaining two fields are formal,
- the remaining two fields are actual data,
- one field is formal, the other is actual.

Here we are not concerned with the values of the particular fields, since we have already established there is no commonality (by constant).

If, by analysis of the inputs and outputs of tuples in the distinguished set, it can be seen that inputs of the set are always formal and outputs of the set are always actual, then no run–time matching is required to satisfy an input, since corresponding fields will always match. We can implement such a set as a simple queue. In our example case, an output adds two integer values to a queue, an input removes or reads two integer values from a queue, and subsequently performs an actual to formal assignment.

If, however, analysis shows we are not so fortunate, but that one of the fields in the tuple is always actual, then that field provides a criterion which can be used to select a match at run–time, that is, a *key* which can be used to locate a matching tuple. In such cases, where a

key is *always* provided, the set can be implemented as a hash table, which may be distributed. In fact, for efficiency, the implementation provides a single common hash table for such cases.

In our example case — let us say the first field is always actual on input and output — we can always use this field to select the correct hash slot for matching tuples. This leaves the last field to match, if this is a formal, the first value we find will do, and we can return this value for assignment.

Complex cases may arise in analysis. Where every field is actual in every output tuple, but not so for every input tuple, a key is not always available. In such cases, it will be necessary at times to perform an exhaustive search of the set. To optimize such cases, such a set is implemented as a private hash table, that is, a single hash table for the distinct set. Distributing a hash table which may require exhaustive searching, raises coordination and consistency issues which in implementation are avoided by not distributing the table.

When no key is available we are compelled to search the whole set. We could choose to implement a list and to search the whole list for each input. In fact, however, such instances are rare, and occur only in some cases where formals appear in outputs. This usage of tuples is so idiosyncratic that no compiler has implemented the case.

There remain some special cases to consider. Conversely, to the first case considered, although bizarre, if it can be seen that outputs of the type structure are always formal and inputs of the type are always actual, then again no run–time matching is required to satisfy an input, since corresponding fields will always match. Further, because no actual to formal assignment is required we simply need to keep track of the instances of each input or output, and we can implement such a set as a counting semaphore.

A simpler and more realistic case exists. That is the case where all the fields are constant. We might expect tuples which have a single string structure to be such a case. Consider a program whose tuple space operations are all on a single tuple

("STOP")

No value requires storage, and we only need to keep a count of the number of instances of this tuple in tuple space. Typically, such tuples are used to perform coordination between processes. Operations on such a set can be modeled by a single counting semaphore. An output increments the count, an input either decrements the count or tests for a value greater than zero. The analysis simplifies what is intuitively a complex feature of the Linda model — namely value associative matching. Complexity remains, though reduced to manageable levels.

Linda implementations can reduce matching costs down to a hashing operation, frequently to the manipulation of queues and occasionally to operations on counting semaphores. Though we must observe that each of these mechanisms have radically different cost and this factor will be a source of some criticism later.

This analysis also requires the complete program, thus is completed on a once and for all basis at run–time. One can conceive of a dynamic system which uses the information from

earlier compilations but this would constrain the optimizer significantly to decisions made early in the process. Clearly these optimizations demand later analysis to be consistent.

3.5 Implementation of logically shared data structures

I have spent some time on the subject of shared data structures in Linda since the proposal made in following chapters is a refinement of this model. The logically shared data structure model, compared to semaphores and message passing, is recent and thus the implementation is less well understood.

To continue to allow a comparative assessment with the other models here I shall focus on the implementation of shared data structures using queues. My reason for doing so is that not only do queues play a central role in the implementation of some Linda structures, they play an analogous role in the implementation of non–synchronized communication and they shall play *the* central role in the implementation of the later proposal. As such it is very important that we gain an insight to the comparative performance of this primitive structure.

In this section I consider the queuing mechanism in detail in a way that allows us to compare it directly with that of the implementation of semaphores and point–to–point communication presented earlier. We shall discover, understandably, that the implementation is more complex. The reason for this additional complexity is the need to allow for dynamic allocation of memory space. The semaphore and point–to–point communication models deal in absolutes; semaphores allow the direct manipulation of a shared address space, point–to–point communication is between only two processes.

Queues are interesting mechanisms since operations upon them display consistent characteristics, i.e. the addition and deletion of entities on the queue have a similar and consistent cost. I shall later present a comparison of these costs.

For simplicity, I again consider an implementation of the essential queue mechanism for a machine in which all the functions described are atomic; i.e. a machine with a single uniformly accessible address space and a process queue similar to the one described in earlier discussion.

The three operations we must implement on a shared data structure I will call

- **write** — write a value,
- **read** — read a value, and
- **get** — read and delete a value.

I shall call the data structure to be considered a "bag" of values, each value of the same type.

As stated, our implementation is more complex than that of channels, though perhaps surprisingly, not significantly so. One major additional requirement for shared data is the occasional necessity to allocate memory for shared data that persists. It is desirable to reduce

the instances of such memory allocation. In the implementation described here this is achieved by deferring the allocation to a point after potential consumers of the data have been given an opportunity to copy the data directly; as the implementation of channels does.

Consider a single shared data structure identified, as were channels, by a pointer to a data structure representing it. This data structure is more complex than the channel one. In place of the single word utilized by channels we use four. Each word being the head pointer of a process queue. The four queues represent pending operations as I have above described; i.e., one each for read and get operations, and two for write operations. This use of extra store is a direct trade off against the read memory and computational overhead of having to make comparisons to detect the operation type.

The four queues are

- **write** — pending write operations,
- **free** — pending allocations waiting to be moved to the write queue,
- **read** — pending read operations,
- **get** — pending get operations.

This structure can be described as

```
struct queue {
    struct list *write;
    struct list *free;
    struct list *read;
    struct list *get;
} q {EMPTY, EMPTY, EMPTY, EMPTY}
```

Each queue is initialized according to this definition.

Each queue points, as for channels, to a process control block which contains a pointer to the source or destination of the data in the operation. The control block in turn either points to the next pending process or it contains the value "EMPTY". The chaining operation replaces the channel initialization required for "ready" channels. We must also add a chaining operation when no data is available; an operation not required in the channel case.

The instructions to implement the read operation are not dissimilar from those for channels. In fact the read operation is marginally more efficient in the "ready" state since it doesn't require channel initialization. The implementation pays a cost in the "non–ready" state for a chaining operation to manage the queue. I shall take a closer look at this later. A Read operation (figure 3.6) simply looks for a "ready" Write, if one is available it copies the value, otherwise it adds itself to the Read pending queue and starts the next available process.

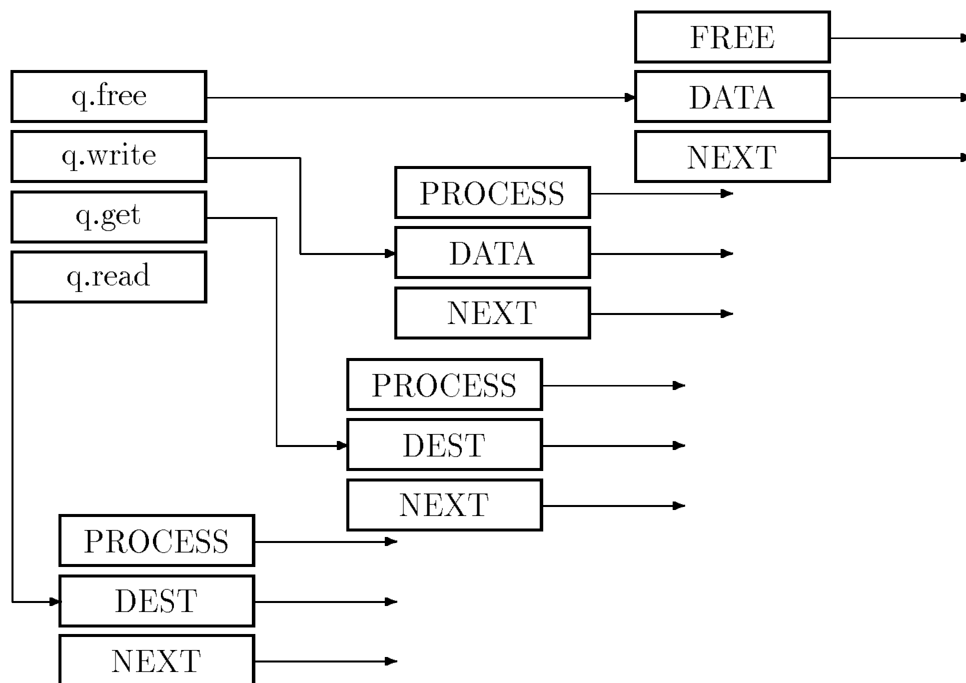


Figure 3.5: Queues implement a shared data structure.

- if no write is pending
  - save the destination in the process control block,
  - add current process to the read pending queue,
  - and start the next process.
- if a write is pending
  - retrieve the data source from the write process control block,
  - copy the data to destination required.

```
read (struct queue *q, data *destination)

    if (q.write == EMPTY) {
        PID->source_destination = destination;
        tolist(q.read, PID);
        nextprocess();
    } else {
        source = q.write->source_destination;
        memcpy(destination, source, sizeof(data));
    }
```

Figure 3.6: Implementation of a read operation on logically shared data.

- if no write is pending
  - save the destination in the process control block,
  - add current process to the get pending queue,
  - and start the next process.
- if a write is pending
  - retrieve the data source from the write process control block,
  - copy the data to destination required,
  - add the writing process to the scheduling list,
  - delete the writing process from the write pending queue.

```
get (struct queue *q, data *destination)

    if (q.write == EMPTY) {
        PID->source_destination = destination;
        tolist(q.get, PID);
        nextprocess();
    } else {
        source = q.write->source_destination;
        memcpy(destination, source, sizeof(data));
        schedule(q.write);
        fromlist(q.write);
    }
```

Figure 3.7: Implementation of a get operation.

The Get operation (figure 3.7) is also similar to the input channel implementation. Again the simple distinction is the requirement for queue management.

Like Read, the Get operation looks for a “ready” Write, if one is available it copies the value and reschedules the writing process. Why is a writing process “pending”; shouldn’t the writing process have been allowed to continue? All will be revealed shortly.

If no Write is available to satisfy the Get operation the Get process adds itself to the Get pending queue and the next available process is started.

The Write operation (figure 3.8) first checks to see if a Get operation is pending, if there is one the Write can simply satisfy the Get, schedule the pending process and continue, if a Get operation is not pending then the Write will simply add itself to the Write pending queue

- if no get is pending
  - save the source in the process control block,
  - add current process to the write pending queue,
  - and start the next process.
- if a get is pending
  - retrieve the destination from the get process control block,
  - copy the data to destination required,
  - add the getting process to the scheduling list,
  - delete the getting process from the get pending queue.

```
write (struct queue *q, data *source)

    if (q.get == EMPTY) {          PID->source_destination = source;
        tolist(q.write, PID);
        nextprocess();
    } else {
        destination = q.get->source_destination;
        memcpy(destination, source, sizeof(data));
        schedule(q.get);
        fromlist(q.get);
    }
```

Figure 3.8: Implementation of write operation.

---

and start the next process on the scheduling queue. By descheduling itself in this way the writing process gives all the other scheduled processes a first stab at reading the source data directly and perhaps rescheduling the writer without requiring a memory allocation.

Since now writing processes are apparently blocked it may happen that the consuming processes will appear starved and the scheduling list will appear empty. In particular, Reading processes are not satisfied by Writes directly. This circumvents any requirement for iteration in the instructions, since many Reads may be satisfied by a single write. Avoiding iteration in these primitive instructions provides them with uniform performance characteristics.

With knowledge of this implementation in any particular instant an intelligent scheduler would do well to place producers at the head of a queue and consumers at the tail. However, the dynamics of the non–particular in any given program are likely to make such an advantage transitory unless compiler directed assessment is allowed to control such dynamics.

We leave the satisfaction of pending Reads to be handled by a system level daemon. This process (figure 3.9) is evoked by the "nextprocess" instruction whenever the scheduling queue becomes empty. This is the last possible point at which we are compelled to allocate new memory for the shared source structure.

Even so there is hope we may avoid the memory allocation yet. Two possible states exist for us here, i.e.

- some number of reading processes may be pending, or
- some number of writing processes may remain blocked.

We would prefer writing processes to be rescheduled by a consuming process in anticipation that a subsequent Get operation will reschedule the writer (circumventing a memory allocation). Thus, if one is available, an available Read process is satisfied and run.

In the last resort we must allocate new memory and allow a writing process to continue since clearly the program expects the written source to persist. In this case new memory is allocated and the "source" copied. In fact we create a new process, for along with the new data allocation we associate a process that will free the allocated memory when a Get acts upon the data — thus completing our deletion semantics.

The function

```
allocfree(data *source, size_t size)
```

creates a process containing a pointer to allocated memory of the specified size. The "allocfree" process returns a pointer to a new process control block which contains a destination pointer to newly allocated space. This new process will be scheduled by a Get operation and when run will free the memory allocation and, in effect, deallocate itself.

In addition to those functions mentioned earlier, the function

```
freed(struct process *p)
```

is true if `p` is an allocated free process; i.e., a process created by the persistence daemon.

The design of the persistence daemon clearly has to cater for all the shared data structures on a given node; where a node is a machine with a common address space. This can be easily considered by the replication of this daemon for each data structure. In a distributed system the daemon may communicate with other nodes in the system if the data structure is distributed between nodes. By daemon replication communication latency can be hidden.

We saw in the previous chapter how in Express a message driven process can handle such remote requests. In a distributed system the implementation of the daemon allows for the addition of such processes. Thus, the next process executed by the daemon may have been contributed by such an internodal message and not be the process scheduled by the daemon.

- if the write queue is empty
  - add those values on the free queue to the write queue,
  - run the next process.
- if there is a read pending
  - retrieve the destination from the read process control block,
  - retrieve the source from the write process control block,
  - copy the data to destination required,
  - delete the reading process from the read pending queue,
  - run the next process.
- if the process on the write queue is not an allocation
  - allocate a new free process,
  - retrieve the address of the newly allocated data space,
  - copy the data into the new allocation,
  - add the new allocation to the free queue,
  - run the next process.

```
daemon ()
  if (q.write == EMPTY) {
      q.write = q.free;
      q.free  = EMPTY;
      nextprocess();
  } else if (q.read  != EMPTY) {
      destination = q.read->source_destination;
      schedule(q.read);
      fromlist(q.read);
      nextprocess();
  } else if (q.write != freed(q.write)) {
      tolist(q.free,
             allocfree(q.write->source_destination,
             sizeof(data)));
      schedule(q.write);
      fromlist(q.write);
      nextprocess();
  }
```

Figure 3.9: Implementation of persistence daemon.

While the implementation described for shared data structures is clearly more complex than that for channels, shared data structures are significantly more flexible. So there is a trade off here to consider between functionality and complexity. Is the trade off a significant one?

The implementation described here is similar to that use in the implementation of shared data structures in the later proposal. There remain opportunities for significant performance and functionality improvements and I shall discuss these in coming chapters and in more detail in the later chapter describing the particular implementation.

3.6 Comparison

I have introduced the implementation of four fundamentals in Process Interaction Models. The implementations are general purpose; we would not use them in specialized cases such as the automatic decomposition and scheduling of fine grain parallelism such as loop small loop iterates (for that there are other solutions). The four I have considered are

- process scheduling,
- semaphores
- point–to–point message passing, and
- queues.

Process scheduling enables multiprogramming and multiprocessor scheduling in closely coupled common address spaces, Semaphores are the fundamental primitive of synchronization and frequently used for maintaining consistency is shared address spaces; providing support for the implementation of critical regions and monitors, Point–to–point communication is the fundamental primitive of the message passing model, and Queues are an important mechanism in the implementation of non–synchronized message passing paradigms and logically shared data structures.

Data exchange. Of the three primitives focused on the sharing of data between processes we see increasing functionality.

A semaphore does not itself provide data exchange so there are no copy operations involved however a pending process list must be maintained. Channel communication, being exclusively between two processes, requires no pending process list but requires a memory copy operation to implement a data exchange. A queue implementing a shared data structure requires both a data exchange and pending process list.

In an attempt to gain a general feel for the differing real cost let us consider the distinctions in the implementations described. Each of the primary operations described

$\mathcal{P}$
$\mathcal{V}$
output
input
read
get
write

has the same conditional form which we shall treat as equal; i.e., each performs a single conditional test. I shall ignore any calling overhead on the basis that I regard the functions specified as macro definitions. I shall consider an assignment including a dereference to have a cost d and direct assignment (such as occurs in input and output) to have a cost a.

Semaphore is the only operation that requires the use of an arithmetic operation; this in the increment and decrement of the associated count. Even so I shall equate this as equal to d.

The functions shall have cost values represented by their name but I will state the following equality

$$\texttt{tolist} = \texttt{fromlist} = \texttt{schedule}$$

Finally I shall combine the cost of the operation independent of the conditional test result.

By this criteria we get the following

$\mathcal{P}$	=	$2d+$ `tolist` + `nextprocess`
$\mathcal{V}$	=	$2d+$ `fromlist` + `schedule`
`output`	=	$2d+2a$ `schedule` + `memcpy` + `nextprocess`
`input`	=	$2d+2a$ `schedule` + `memcpy` + `nextprocess`
`read`	=	$2d+$ `tolist` + `memcpy` + `nextprocess`
`get`	=	$2d+$ `tolist` + `fromlist` + `memcpy` + `nextprocess`
`write`	=	$2d+$ `tolist` + `fromlist` + `memcpy` + `schedule` + `nextprocess`

By simplification this we can illustrate the difference as

$\mathcal{P}$	=	0
$\mathcal{V}$	=	`schedule` − `nextprocess`
`output`	=	$2a+$ `memcpy`
`input`	=	$2a+$ `memcpy`
`read`	=	`memcpy`
`get`	=	`fromlist` + `memcpy`
`write`	=	`tolist` + `memcpy` + `schedule`

By comparison semaphores prove to be very efficient but do not account for any copy operations introduced by the programmer. Focus though on the distinction between point–to–point communication and shared data structures implemented by queues. Eliminate the memcpy cost for a moment, if we look inside the list functions we shall see that the cost of these (depending whether we have chosen LIFO of FIFO scheduling) is either d or $2d$.

Since LIFO scheduling is simple to implement, requires less memory and provides superior spatial and temporal characteristics (essential for cache memory subsystems) my preference will be for the cheaper function. This comes at the expense of fairness; but I see this as no cause for concern since fairness is not required by the semantics, and nor is fairness an issue in the point–to–point primitives against which we are making this comparison.

If we further accept that it is valid in this analysis to sum the total for each model to derive an average cost for each operation and take $2a = d$ as true then we can see that the point–to–point primitives have the same cost as those specified for the shared data structure implementation.

So far no mention has been made of the shared data structure daemon also described in the implementation; I shall argue here for its continued exclusion from my analysis. Here is why: I simply regard the daemon as a process with which the primitives interact equivalent to a process a message passing programmer will be forced to construct for himself in practice to provide anything like the same functionality. Thus, I feel justified in excluding it from my analysis on the basis that it will improperly distort the results.

One final point can be seen in this comparison. The function `memcpy` has a cost proportional to the size of the data involved in the data exchange. It would be most desirable to find a mechanism that would enable the reduction of this cost to a small constant value. This can be achieved if we provide a mechanism that allows for the exchange of data by reference. The later proposal does indeed provide such a mechanism.

3.7 Consistency reviewed

I have endowed myself with the luxury in this chapter of functions that I have defined to be atomic, but not accounted for the cost of maintaining this atomicity. As I pointed out earlier the level of atomicity is very architecture dependent. On some systems semaphores and the channel support described here have already become available in hardware and, indeed, the foregoing can be considered (in each case) a valid description of the hardware behavior and costs.

Of the software implementation consider the following.

On a uniprocessor machine with no interrupts (i.e., all scheduling is non–preemptive) I can be forgiven with grace since no consistency problem can occur.

On a uniprocessor with preemptive scheduling I shall have to insist the machine follow

the wisdom of the transputer such that interruption occurs only at well defined points in a program — this is also desirable to reduce the amount of state that needs to be saved on a context switch. These points (e.g., loop end, jmp, return) do not cause problems in the functions specified in this chapter.

On a multiprocessor machine a central scheduling list can be the source of much contention. To alleviate this contention, to speed scheduling and to provide the atomicity required I shall insist that the machine contain a small additional scheduling and consistency processor (a shared coprocessor) that is the only processor allowed to manipulate the scheduling list. This processor is specially designed to minimize response time during periods of high contention. Thus, the functions in this chapter are executed solely by this single processor for the other processors present in the system.

Where the hardware solution is not provided I am forced to ensure the consistency of these functions by the application of appropriate semaphores.

It should be noted that I have avoided "spin lock"techniques. For a description of such technique for semaphores the reader is referred to Hennessy and Patterson([Hen90] page 471). I disregard this approach first on aesthetic grounds — busy wait offends my sensibilities; wasting cycles that might otherwise be used, second it does not scale well to many processors because of memory subsystem traffic when the "lock" is released , third non–busy solutions (such as that illustrated here) provide a viable and efficient solution.

Chapter 4

Critique

We have considered interaction models; i.e., models that embody synchronization and exchange of data between *processes*. To simplify this analysis I have focused on fundamentals of process models and interaction. Semaphores, embodying the notion of conditional precedence, as a fundamental primitive of synchronization, point–to–point communication as the fundamental of message passing data exchange and queues as the basis of a fundamental for the implementation of Logically Shared Data Structures. Further we have focused on three specific existing models for programming with concurrency.

- Global shared memory — evolved primarily by extensions to earlier sequential models with regions and monitors as the high level aspects of the programming model.
- Generalized message passing, epitomised by Occam, with a focus on "Concurrency and Communication"; i.e., the *direct* interaction of processes by point–to–point communication, and barrier synchronization.
- Linda, whose message is subtly distinct, purports a focus on "Concurrency and Coordination"; i.e., the *indirect* interaction of processes via an intermediate shared data structure.

All three models address the single issue of process interaction. In the following discussion I take a critical look and these models in the anticipation that by developing an understanding of their critical aspects we might evolve a useful general purpose model.

4.1 What's wrong with global shared memory?

For a model that has received so much criticism over recent years global shared memory has been surprisingly resilient. The principal reason for this resilience is, I believe, that it has proven to be very effective on small scale parallel machines. In addition the concepts

associated with it are familiar to many engineers since it has been prevalent in the operating systems of uniprocessor machines for some time. In particular Fork and semaphore operations are an integral part of today's systems (such as UNIX).

Here I briefly reiterate the regular criticisms of the global shared memory model.

Recall that semaphores are difficult to use and error prone. The programmer must be well disciplined in their use to ensure that every access to shared variables is constrained by semaphore operations $\mathcal{P}$ and $\mathcal{V}$. Recall also how high level programming structures, designed mainly by Dijkstra and Hoare, had been designed to alleviate these problems.

Yet these solutions have proven inadequate as Hoare points out in his book on CSP. Sharing is complex, conditional critical regions and monitors are inefficient due to repeated testing of entry conditions. More elaborate schemes for monitors have evolved such as those with the range of features illustrated in the earlier chapter for the Encore Multimax but as Hoare says([Hoa85] page 230) "... the extra complexity is hardly worthwhile".

A case was made for high level process models in chapter 2.

4.2 What's wrong with message passing?

If it is not clear yet it is important here to understand the distinction between message passing as a component of parallel machine architecture and "Generalized Message Passing" as a programming model.

Concern for the characteristics of communication between nodes of a machine is an important and significant issue. Operating systems and VLSI must continue to provide internodal connectivity, in part by message passing, to support higher level models. These are not the issues being addressed here.

The issues we are concerned with here are those of message passing as a programming model. How programmers may conceive and construct parallel programs whose performance semantics may be well understood and remain efficient. For instance we must consider how efficiency can be made *uniform* regardless of the architecture of the machine's memory subsystem.

The Occam model of message passing certainly enables simplicity in implementation. However, this simplicity has a cost, and the cost is significant.

The primary goals of the Occam model were

- simple implementation,
- Generalized Message Passing as the basis of a general purpose parallel programming model, and
- efficiency.

The first goal was undoubtedly achieved in the manifestation of the transputer [INM90]. However, the remaining goals are incompatible and both fail as a result.

The failure of the model as general purpose in practice has already been highlighted in the Introduction. Against message passing I make two major contentions, message passing is

1. not general purpose, and
2. preoccupies programmers with issues of data distribution.

The efficiency failure occurs in the context of this generalization and is in essence caused by the copying of data which might otherwise be passed by reference.

The criticism here of message passing does not apply in such cases where the model is tied closely to specialized machine architecture. Specialized applications of message passing, such as systolic algorithms for systolic arrays are simply outside this criticism.

Generalized Message Passing implements communication between processes in the same address space as a memory to memory copy operation. This increases traffic in the most significant bottleneck in modern machine architecture — the memory subsystem. Indeed, modern cache memory and load and store CPU architecture conspire against the efficiency of memory to memory copy operations.

The model does not map well across the range of MIMD parallel machine architectures for this reason. A message passing program on a shared memory multiprocessor would certainly pay a performance penalty for these copy operations and other programming models, which do allow the exchange of data by reference (such as those mentioned for the global memory model), would be, indeed are, preferred on such machines.

The real disadvantages of this simple mechanism lie at the higher level. The programmer is forced to consider, in some detail, multiplexing and routing issues when distributing data among groups of processes.

In addition, when programming in this channel model (as in Occam), providing sensible names for channels proves to be difficult, problematic, and a further preoccupation for the programmer; introducing increased levels of complexity as a program evolves.

Consider a typical, trivial example of an Occam process:

```
PAR
  r ? buffer[0]
  P  (buffer[1])
  l ! buffer[2]
```

where `r` and `l` are channels, and `P` is some process. The process illustrates overlapping communication and computation, in a very expressive manner. However, real programs rapidly increase the complexity of a programmers naming scheme for channels and carefully thought out channel names begin to introduce confusion in the source.

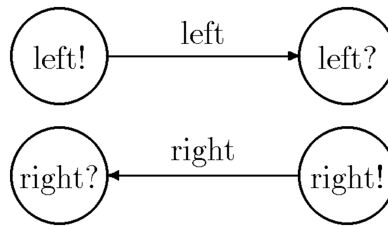


Figure 4.1: Naming channels can be confusing.

```
PROC R (CHAN l, r)
  PAR
    r ? buffer[0]
    P  (buffer[1])
    l ! buffer[2]
:
CHAN left, right:
PAR
  R (left, right)
  R (right, left)
```

This is a trivial, and introvert, piece of code to illustrate a point. We have combined two processes of the previous example by procedural abstraction and allocated the necessary channels to connect our processes, what appears as `right` in one process instant means `left` in another (figure 4.1).

Consider a real piece of Occam code, that serves to illustrate the point further. Substantial pieces of Occam provide commensurate degrees of complexity and ambiguity.

```
CHAN OF [pktSize]INT clockwiseIn, clockwiseOut,
                     NclockwiseIn, NclockwiseOut:
PAR
  LinkGuardian (clockwiseOut, clockwiseIn,
                clockwiseRingIn, clockwiseRingOut)
  LinkGuardian (NclockwiseOut, NclockwiseIn,
                NclockwiseRingIn, NclockwiseRingOut)
  UserGuardian (userIn, userOut,
                clockwiseIn, clockwiseOut,
                NclockwiseIn, NclockwiseOut)
```

In fact, the programmer has here adopted their own convention of placing channel pairs that are formals in order, input first, output second. This style of programming could be

described as *patch panel* programming, since it is synonymous with patching wires on a telephone exchange panel.

No doubt some readers will have in their mind implementations of message passing that are more dynamic than Occam. In particular, we can envisage those implementations of message passing that provide non–blocking (nonsynchronized) outputs (often referred to, incorrectly, as "asynchronous" message passing).

Even so, it should be clear that the same contentions made against Occam apply equally to these forms of message passing, perhaps more so. Recall, they are

- not general purpose, and
- preoccupy programmers with issues of data distribution.

To conclude, Generalized Message Passing is an unsuitable model for general purpose programming of parallel computers[1]. I do not contend that message passing is an unimportant component of the machine architecture, on the contrary, *communication* is an essential component of current parallel machine architectures.

Generalized Message Passing is an unsuitable model for general purpose programming of parallel computers since it causes engineers to be preoccupied with issues of data distribution and compels the implementation to copy data that might otherwise be exchanged by reference.

It is possible to write message passing programs that are efficient but only if the engineer writes topology specific code with a detailed awareness of the target machine. Thus it can be concluded that *message passing is a suitable model for specialized applications* (e.g. systolic) if closely mapped to a compatible target machine architecture.

4.3 What's wrong with Linda?

In addition to the work described here, several other groups have been prompted to examine the ideas behind this loosely defined model. Most notable among the recent work — and representative of the broad spectrum of interest — are the UNITY like SWARM[RoCu90] and Orca[Bal89], a component of the Amoeba[TaMu81] distributed operating system.

The important concepts introduced by Linda are a distinct shared data space and simple operations to change the state of that space. In many ways Linda has changed the understanding of how data objects can be *addressed* in parallel machines.

Linda concepts appear powerful. However, they are certainly too abstract in the context of the common host language C.

[1] This is a conclusion many would consider foregone. However, the view that generalized message passing is a suitable general purpose model has become a common mythology of European computer science.

The model cannot be applied to a broad spectrum of applications on parallel machines. The model is unsuitable for programming real–time or embedded systems since the performance semantics of tuple space operations are difficult to predict. Linda programming is heavily dependent on the program optimization we have seen earlier.

All modern compilers utilized optimization and these optimizations have performance effects. However, the Linda optimizations are of a radical nature, with such different performance characteristics, ranging from the cost of a simple counting semaphore to the complex cost of a distributed hash table and exhaustive searching. The introduction to a program of a single new tuple type may cause the optimizer to change strategy with remarkable effects, invalidating any earlier empirical analysis by the programmer. This leads programmers to develop techniques and conventions founded on an understanding of the behavior of a particular optimizer or underlying matching protocol, subverting any meaningful portability.

The extensive analysis of Linda programs cannot be applied at run time, indeed they must be applied to the whole program on a once and for all basis, thus requiring dynamic implementations to take very different approaches, with very different performance characteristics associated with each Linda operation.

Linda does not provide opportunities for exchanging data by reference. Each operation in the search for a match may involve several more copy operations than the equivalent operations in the message passing model. In short, the value matching overhead, for all the optimization, can still be significant.

Of particular concern to scientific applications is that the expression of distributed matrices cannot be achieved simply. Programmers are required to contrive a tuple structure to meet the requirement.

The Linda model makes it necessary for programmers to contrive naming schemes within tuple space to allow an uncertain association between processes. The complete disjoint nature of Linda processes makes it difficult to consider well formed process structures; e.g., such as those that might describe the behavior of robots or other control systems.

Let us consider these issues in a little more detail. I have said that with the extensive optimization it is impossible for a programmer to predict the performance of a Linda program, without detailed empirical analysis. The problem is significant. You cannot predict the performance of the Linda program, and you will not know the performance until the task is complete — any empirical analysis the programmer performs at any point in the development will likely be invalidated by subsequent changes to the program since the optimizer will almost certainly change strategy as new tuple types are introduced.

In applications and certainly in low level programming (as encouraged by Linda's principal partner to date, C) this will incline programmers to adopt programming conventions based on their developed understanding of a particular implementation. Programs designed to work well under one implementation will have unpredictable performance characteristics in some other implementation.

Programmers *will* write programs which exploit some facet of the way the optimizer behaves, thus subverting the very conceptual elegance the Linda abstraction seeks to provide in the first instance.

The Linda convention for accessing components of distributed arrays; e.g.,

```
in("foo", i, ? v)
```

optimizes to a hash table access. It may be inferred therefore that access to distributed arrays is inefficient, a serious disappointment for the scientific community where applications depend on such structures.

It should be possible to provide analyses that highlight opportunities for passing data by reference, just as is should be possible for Occam (though no compiler in either case does this), but currently copying data is always required by an interaction. This means that in both cases manipulation of complex data structures is an expensive operation.

One of the most significant practical advantages of Linda is that it is combined with a conventional language and thus does not require users to relearn language skills, of course, this same advantage is gained by Message Passing in the same way.

The Linda model arose from the shared memory culture prevalent in the USA at the time of the proposal. What was sought in the proposal was an addressing abstraction, and this was found in value associative matching.

It is also clear from the Linda proposal of value associative matching that a principle concern of Linda philosophy is interaction in systems with disjoint name spaces; i.e., a system in which no one process inherits knowledge of the names specified in any other. Thus several of the proposals for the manipulation of multiple tuple spaces (e.g. [Hup90]) include operations that ensure this imperative is preserved. This has complicated the task of specifying a Linda system with multiple tuple spaces. Let us for a moment focus on the demands of this imperative on Linda as we have seen it.

A name space provides a mechanism by which a name, an identifying entity, is associated with a value. Linda isolates explicit naming to that provided by the host language – which usually means explicit naming of local data. *Yet any process acting upon a tuple set has in any case to share knowledge of the tuple structure upon which it wishes to act*. This amounts to an implicit naming scheme shared by processes that the programmer must be aware of yet the Linda model cannot explicitly express.

In Linda we often find contrivances of the form

```
out("matrixA", i, 6)
```

and

```
in("matrixA, 0, ?v)
```

This tuple structure has, in essence, associated a name (the string `"matrixA"`) with a set of tuples whose type is

$$(\text{integer}, \text{integer}).$$

While it is true that these names can be read; i.e.,

```
rd(?name, ?v, ?v)
```

where `name` is a string variable, this mechanism adds little (other than complexity) when knowledge of the tuple structure is required by the process *a priori*. Indeed, in some implementations of Linda a first string field constant is a requirement.

We leave the naming issue with the following observation. Value associative matching does not provide an elegant solution to the issue of identifying values in systems with disjoint name spaces since processes must concur on both the tuple structure and any contrived naming by mechanisms outside the model. Further, such concurrence is impossible to check in a compiler without such a mechanism.

Chapter 5

Semiotics

Conventional programming language definitions have been primarily concerned with the development of *Syntax* and *Semantics* in the context of some infinite abstract machine model. For sequential languages this abstraction is the Turing Machine (an infinite resource uniprocessor). For newly evolving languages dealing with concurrency based on interacting sequential processes, the abstract machine has been an extension of this model — an infinite number of interconnected Turing Machines.

Abstract, infinite resource, machines enable the application of many mathematical techniques to computer programs. Unfortunately, the programmer finds herself in the classic engineering predicament of applying theory to practice. The programmers universe is a finite one, yet when seeking guidance in this universe none can be found in the formal language definition. The programmer is alone, or at best clutching a poorly specified and informally described system manual which has poor relation to the language used for programming, and is often in conflict with it. No mention is made of the effects that resource limitations may have on program behaviour, nor is mention made of the pragmatics that exist to caused the invention of some language feature.

In the past pragmatic issues have been left to an uncertain tutorial style which usually takes the form of an informal introduction to the language syntax and semantics, and some sketchy detail of implementation restrictions.

5.1 Performance semantics

In many conventional programming languages experienced programmers adopt a hidden semantics based on their understanding of the behaviour of a particular or general implementation issue. These semantics I name *performance semantics*, since they derive from a particular use of the language in anticipation that such use will result in some performance benefit.

Examples of such usage in conventional sequential languages[1] are

- use of shift operators for positive power of two multiplication and division,
- use of in line code in preference to procedure or function calls, and
- use of a certain loop construct on the understanding that an optimizer will vectorize the code and use an available vector processor.

Examples of performance semantics in parallel programming are prolific. Indeed, the very nature of some explicit parallel constructions is pragmatic. Replication, for example, is utilized when the programmer has an understanding that to use the construction will provide some performance benefit; i.e., the use of the construction does not contribute to the solution (is not an intrinsic of the algorithm).

None–the–less, general parallel construction need not be seen this way. Algorithmic decomposition using parallel constructions is a modular programming methodology not dissimilar from such methodologies as structured programming and object oriented programming. Thus general parallel construction need not be regarded as introducing performance semantics but rather be seen as a modular method of program construction; i.e., that such a program may be run on a machine with multiple processors is incidental.

The increasing introduction of performance optimizing compilers can be a source of great semantic abuse. Where the style and actual form of a program is directly affected by the programmer's understanding of the optimizing behavior of a particular implementation.

5.2 *Semiotics*

The term *semiotic* in linguistics and philosophy is used in reference to the complete study of signs. The study in these disciplines consists primarily of the three forms *syntax*, *semantics* and *pragmatics*. In this thesis I adopt the term in a particular sense. More usually found in linguistics or philosophy the term refers to "a general philosophical theory of signs and symbols that deals esp. with their function in both artificially constructed and natural languages and comprises the three branches of syntactics, semantics, and pragmatics"(Webster's Third New International Dictionary). In the manner I use the term here semiotics, in addition to considering the semantics and syntax of a programming language considers the effect the language has upon the behavior of the programmer, and, in particular, the pragmatic statements required for the programmer to make consistent and efficient use of the language.

The syntax and semantics of existing programming languages have inherited much from the development of mathematical formalism. In pure mathematics there are simple pragmatics. These pragmatics consist of

[1]These examples are simply those drawn from my own observation of programmer behavior.

- knowledge that use of the formalism is correctly applied (i.e. degree of confidence in the ability of the mathematician), and
- interpretation of the expressions by the human observer.

In mathematics both these pragmatics are dealt with by cross referencing with the competence of other mathematicians. However, it is important to observe that without the application of these pragmatics the validity of any mathematical result is in question.

The role of pragmatics in mathematics is an important one. The incorrect application of a formalism produces incorrect or meaningless results. A misconception means to interpret the behavior of an expression incorrectly. Incorrect application of the formalism implies the results are incorrect or unexpected. Calculus, geometry, whatever mathematical form, is just so much scribble without these pragmatics.

The pragmatics in mathematics rests on the integrity of the mathematician doing the work, the ability to cross reference with other mathematicians who can verify the correctness of the application and interpretation. This is an entirely reasonable assumption among mathematicians.

Plato formalised these mathematical pragmatics in a single statement written over the door of the Academy

> *"Let only geometers enter."*

Indeed, these pragmatics are a pragmatic of the scientific community.

5.3 *Semiotics* in Computer Science

For the programmer dealing with large, non–trivial, problems a central and unavoidable issue is performance. It is this pragmatic of performance that most distinguishes the engineering sciences (including Computer Science) from Mathematics.

Syntax and semantics are becoming well understood in the Computer Science community, they are issues of great complexity and form. I am concerned here not with those issues but rather the effect that the language and its implementation have upon the behavior of the programmer. As an example imagine a language where the use of procedural abstraction incurred a significant performance cost. This will effect the behavior of the programmer in time. The programmer learns through experience to avoid using procedural abstraction for trivial pieces of code.

We have seen mentioned in the earlier discussion several instances of this *semiotic* effect. In Occam, programmers soon learn that to over use communication in programs incurs a significant penalty due to an increase in the copy penalty, programmers begin programming with

the free expression of parallelism but pragmatics soon bring constraint and the programmer's behavior is modified; modified by the performance semantics of the language.

In generalized message passing a preoccupation with routing and multiplexing is a semiotic effect that directly changes the way the programmer behaves. In this case the effect serves to distract the programmer from the real task: algorithmic development.

In Linda the performance semantics of each primitive operation is not uniformly unpredictable because of the requirement for run time matching and optimization, and again the programmer's behavior is modified (in the ways discussed in the introduction).

Performance semantics are less of an issue in sequential programming since a sequential machine generally has a balanced nature with consistent performance semantics. However, for parallel and distributed programming no such balance exists in current and foreseeable technology. The more computer technology becomes parallel the greater this issue will become. Performance is an architectural complexity dependent on the balanced nature of the machine.

So what are we to do? We must come to understand these issues. This understanding will itself have an effect on the behavior of programming language designers.

In the following proposal I have made a first attempt at providing a truly semiotic definition by providing a section of explicit pragmatics that will enable the programmer to make efficient and consistent use of the language described.

Chapter 6

Ease — A New Proposal

Let us concede then that a process model is both a useful and desirable programmer's model and target model for higher level applications and systems. To support this concession we may point to the popularity of such models in systems architecture and languages such as Occam. I make no further case for it here, it is an underlying assumption to the work presented and I do not wish to distract the reader with a discussion concerning explicit versus implicit parallelism. In return I concede that models such as the functional and symbolic programming models may be, indeed are, preferred in some circumstances.

Thus, in pursuit of the thesis, I now consider how an expressive, general purpose, model for interaction between concurrently executing processes can be formed. One that refines the models we have discussed, preserving desirable characteristics and overcoming the deficiencies highlighted.

The solution proposed arises from the earlier debate, from that debate we derive the following objectives for our interaction model, such an interaction model should provide

- simple mechanisms with consistent performance semantics, and
- functionality that does not distract the programmer.

Thus, we wish to avoid specifying mechanisms that are difficult to implement; i.e., those that have varying performance semantics in implementation such as those specified in the Linda model. We also seek facility that does not distract the programmer in the way message passing does.

We are still left with a fundamental imbalance in implementation caused on many of today's architectures by the dramatic difference between the addressing of data held in a directly addressable store and the addressing of data held in an indirectly addressable store. This difference corresponds to the distinction between local memory and non–local (remote) memory access in distributed systems.

This problem is one of communication latency and can only ultimately be solved, for a fully general purpose system where the unit access time to all store is uniform. Such a model requires the rethinking of modern computer architecture or the invention of a new, as yet unknown, technology and is beyond our scope here.

The latency problem can be reduced (perhaps overcome) by ensuring sufficient parallelism exists to maintain computational activity whilst non–local data is fetched. This is not a general purpose solution and is of value only if the creation of such parallelism does not itself introduce a significant performance cost. This cost may be high if the processes involved have to interact extensively as a result of the extra decomposition. None–the–less, excess parallelism does provide a partial solution on conventional machine architectures.

We have also seen that to generalize the communication mechanism so that it also provides all interaction between the processes of a program is problematic. Cited in our earlier discussion were the copy penalty and programmer preoccupation with data distribution issues such as routing and multiplexing.

In the following presentation a new model is presented designed according to these objectives. The proposed model has been given the name *Ease*. It is a model that maintains the process model and provides a set of simple and symmetric interaction primitives. These primitives act upon strictly typed shared data spaces called "Contexts".

As expected, contexts provide a means of describing data structures whose components may be shared and manipulated by many processing elements. Later a complete programming language is illustrated that incorporates the model, and further, a definition is also given for the incorporation of the model into the language C.

The language presented here is close to the spirit of CSP. CSP provides a mathematical foundation for *Ease* since the process model is similar to Occam and Contexts can be simply defined as processes.

The full language addresses several significant systems issues, by providing simple mechanisms for the construction of statically reusable and virtual resources and providing a uniform means for handling failure and error.

Ease programs are described as collections of processes that execute concurrently, constructing and interacting via strictly typed shared data structures. A context provides a *priority* oriented, intermediary in which shared (perhaps distributed) data structures are constructed and by which processes interact. Unlike the generalised Tuple Space of Linda, *Ease* contexts are strictly typed and perform no matching at run time. Processes are distinct entities that interact with and via distinct contexts. A process is well defined and cannot side affect other processes.

Ease, like the language Occam, attempts to follow the principle of the Franciscan philosopher, William of Ockham, and creates no more entities than *necessity* requires. The identification of this necessity is as perceived in the context of the stated requirements. I adopt the tenet that elegance in a programming language is found in a symmetry that allows a useful

and intuitive expression of intent.

6.1 A new model arises

The remainder of this chapter gives an overview of the *Ease* model and concentrates on those aspects of the model that distinguish *Ease* as a unique model for programming parallel machines.

The model provides a type associative storage that can be efficiently implemented on a range of machine memory architectures through the provision of mechanisms that allow the exchange of data by reference. In implementation the methods discussed in the earlier chapter can be utilized and the efficiency enhanced by simple exchange of references in place of the copy operations specified, as we shall see.

The model is sufficiently distinct to be added to a conventional language as message passing primitives and, indeed, Linda primitives have been in the past.

The model enables simpler implementation and thus greater efficiency than Linda value associativity by obviating all run time matching. The model maintains, and enhances, the richness of shared data spaces as the basis for expressing interactions between processes. The model provides for efficient exchange of data by encapsulating a mechanism for exchange by reference and thus reducing the copy cost discussed earlier. This encapsulation enables the model to be implemented independent of machine memory architecture, yet remain efficient when compiled for either shared or distributed memory machines.

A program is described as a collection of processes that execute concurrently, constructing and interacting via strictly typed shared (distributed) data structures called *Contexts*.

Ease is novel in the following regard: a *context* provides a *priority* oriented and strictly typed intermediary in which distributed data structures are constructed and by which processes may interact.

The model provides simple and symmetric operators — read and write, get and put. The process model provides constructions for both cooperative and subordinate concurrency and a mechanism for building statically reusable and virtual resources on parallel and distributed machines.

6.2 Contexts — shared data structures

A *Context* is a typed shared data structure, either

- a **bag** — an unordered set of some type.

- a **stream** — a serially ordered set of some type where the least recently output value is the value input.
- a **singleton** — a single distinct object or array of objects that may be selected by subscription.
- a **call–reply** — a type providing guaranteed call reply semantics

Contexts of distinct types may be gathered under a single name enabling a single shared *space* of multiple types to be constructed. Operations on a context space are type associative (name equivalent); i.e., operations are valid if the type of the value or variable is one of the types specified for the space.

6.3 Operations – actions on shared data

There are four simple, symmetric, operations on contexts. They are

- `write (c, e)` — copies the value of the expression `e` to the context `c`.
- `read  (c, v)` — copies a value from the context `c` to a variable `v`.

- `put (c, n)` — moves the value associated with the name `n` to the context `c`.
- `get (c, n)` — moves a value from the context `c` and binds it to the name `n`.

Write and read are copy operations. Put and get are binding operators. The synchronization characteristics of the operations are similarly symmetric

- get and read block if data is not existent,
- write and put are non-blocking.

Consider how these operations change the state of a program.

Write changes the state of a context, leaving the local state unchanged. Read changes the local state whilst leaving the context state unchanged.

Put changes both the context state and local state; i.e., subsequently the value associated with the variable name used in the operation is undefined. Get also changes both the context state and the local state; i.e., the value bound to the variable name used in the operation is removed from the context.

6.4 Uniformly building and using resources

The construction of and interaction with resources has special requirements. To enable the simple and uniform view of resources in parallel and distributed environments, *Ease* provides *combinations*.

A *combination* provides guaranteed call–reply semantics via some context. A process that outputs a request to some resource that has access to a shared context is guaranteed to receive the corresponding reply to that request; thus two particular processes synchronize. A combination consists of two associated operations.

- a **call** – behaves like an output followed by a get, and
- a **resource** – behaves like a get, a process and subsequently an output.

The value output by the resource is guaranteed to satisfy the corresponding get of the associated call. This call–reply guarantee allows the simple creation of statically reusable and virtual resources.

A *statically reusable* resource is a process that manages direct access to the actual resource. A vector processor may be considered a statically reusable resource since the user process must await its turn before use. A simulation of the resource behavior may not be useful.

A *virtual resource* is a process that "pretends" to be the actual resource. A disc cache can be considered to provide virtual resource since it returns to the user immediately as though the requested action had been completed on the actual resource.

6.5 The process model

Ease provides two forms of process creation that differ in their synchronization characteristics.

6.5.1 Cooperating processes

A cooperation, creates some number of *cooperating* parallel processes.

$$\|P\|Q;$$

is the combination of two processes P and Q. The cooperation terminates when all the processes have terminated.

Cooperations thus represent processes that cooperate closely; multitasking processes on a single node of a uniprocessor or perhaps processes on a shared memory multiprocessor.

A special shorthand, a *replication*, allows many similar processes to be created, i.e.

$$\|i \text{ for } n : P(i);$$

creates a cooperation of n processes where each has an index i in its scope.

6.5.2 Subordinate processes

A subordination, creates one or more *subordinate* processes.

$$//P;$$

creates a single process P. Unlike cooperation subordination terminates immediately; i.e., the subordinate process is created and the creating process continues.

Again, a replication allows many similar processes to be created:

$$//i \text{ for } n : P(i);$$

creates `n` processes, each of which has an argument index i with a distinct value from 0 to $n-1$.

Subordinate processes are "process creation" thus they continue with a disjoint scope. If a subordinate interacts with a context whose parallel scope has terminated (because the process that defined the scope has terminated), the subordinate terminates. This mechanism is useful since it enables reasoning about speculative computation and provides the automatic release of allocated resources.

A concurrent process may access any context names in scope at the point of instantiation, but may not include references to any non-local variables.

6.5.3 Type associativity

A simple shared data structure might be a 2 dimensional matrix, the value of a component of such a matrix can be explicitly written to by

$$\texttt{write}[x][y](k, e)$$

where e is an expression, k the name identifying the context and x and y the subscript identifiers of the component.

We may specify contexts with multiple types, such that the name k in our example could be a "space" consisting of both a 2 dimensional matrix of components and a bag of components with a different type. A write

$$\text{write}(k, e')$$

will place the value of e' in the bag provided it is a compatible type. This amounts to a shorthand method for gathering different shared data structures under a single name where the particular context is selected by the type associated with the expression used in the operation. Such selection adds no overhead to the implementation since the compiler can infer the particular context at compile time.

The typing here is name equivalent which allows the programmer to build sophisticated, though well typed, shared data spaces.

6.6 Priority

A further characteristic is added to each component data structure of a shared space, that of priority; i.e., for each data structure specified under a single name a priority can be associated such that operations upon those data structures possessing a higher priority are completed in preference to those with a lower priority.

This priority aspect of *Ease* is experimental and its usefulness must be held in question until further experience is gained. The intent is to associate priority with data rather than associate priority with process (the case in Occam for example).

Chapter 7

Writing Programs with Ease

Ease is an Algorithmic Language[1], as such it belongs to the same class of languages as FORTRAN, C and Algol. Algorithms are expressed in the language using explicit control statements and actions upon data whose value is respecified by assignment. The following sections will take easy steps through the major concepts of the language though some knowledge of other imperative languages is supposed.

The following sections will also help develop an understanding of the definitions and methods used to describe the language and model. Each section begins with a simple precise and often continues with a detailed description or rationale. Readers may wish to pass by these details on first reading.

To keep forward references to a minimum the exposition begins with a precise but brief introduction to the central concepts of process and data. A more detailed explanation of the full language follows.

7.1 A brief introduction to the notion of "process"

A *process* is an action or some composition of processes. An *action* is an assignment to a variable or context (a "shared data structure"). Actions are composed to form process *compositions* such as sequences, conditionals, loops and parallels, that can be further composed. The assignment syntax

$$x := 1$$

represents the simplest kind of process called an action, it "assigns" the value represented by 1 to the variable x.

[1] Imperative Programming Language.

- An **action** is an assignment to a variable or shared data structure.
- A **process** is an action or combination of processes such as a sequence, conditional, loop or parallel.

Figure 7.1: Process concepts.

If P and Q are processes, a sequence

$$\{\ P \\ \quad Q\ \}$$

composes P and Q such that if P terminates successfully then Q is performed.

If b is a boolean expression (i.e. either true or false) then the sequence

$$\texttt{test}\ b : P \\ \qquad \texttt{else}\ Q;$$

is a sequence that behaves like the process P if b is true, otherwise it behaves like the process Q.

Processes can also be composed in parallel; i.e.,

$$||P \\ ||Q;$$

composes P and Q in parallel. Parallel algorithms often make use of "replication"; i.e., multiple copies of the same process.

$$||i\ \texttt{for}\ n : P(i);$$

composes n copies of the process $P(i)$, where i names an index value that may be used in P.

Each of the above parallel compositions share a "barrier synchronization"; i.e., the composition terminates when all the composed processes terminate. The sequence

$$\{\ //P; \\ \quad Q\ \}$$

is also a parallel composition, but P is said to be *subordinate* to the sequence containing Q; i.e., the sequence begins by creating a process P and then immediately continues with Q, both the "principal" sequence and the "subordinate" P terminate independently.

Similarly, the sequence

$$\{ \ //i \texttt{ for } n : P(i); \quad Q \ \}$$

is a parallel composition in which n copies of P are subordinate, the "principal" sequence and each copy of P terminate independently. To summarize,

- a **sequence** composes processes sequentially,
- a **cooperation** composes processes in parallel form such that subsequent processes continue only when all the processes in the composition have terminated, and
- a **subordination** composes processes in a parallel form concurrent to the subsequent processes; i.e., the parallel processes are "created" and the principal (i.e., the creating) process continues.

Let us end here this brief introduction to the notion of *process* and have a similarly brief look at the concepts of data.

7.2 A brief introduction to the notion of "data"

Since *Ease* is a parallel/distributed programming language with a strong process model it conceives of data as having one of two principal properties. It is either "local" (private to a process) or it is "nonlocal" (shared by concurrent processes). Local variable data is represented in a process by *variables*, nonlocal variable data is represented by shared structures called *Contexts*.

The behavior of an action is reflected in the effect it has on it's environment. An *environment* is the set of scopes whose names are valid in a process. A *scope* defines in which process a name is valid.

A *variable* is an element (such as a variable name) whose initial value is determined by its specification and whose subsequent value is defined by the actions in the sequential process for which it is specified.

A *context* is an element (such as a context name) whose initial value is empty and whose subsequent value is defined by the actions in the concurrent processes for which it is specified.

An *element* is a name, a subscripted name or an aggregate representing a set of distinct variables or contexts. In particular, an element can be an array or tuple. An *array* is an aggregate of distinct variables or contexts of the same type each distinguished by a unique subscript. A *tuple* is an aggregate of distinct variables of differing type each distinguished by a unique subscript.

Actions specify the value associated with an element in an *assignment*. Thus assignment is the primitive notion in *Ease*. In effect, each assignment "respecifies" the environment.

- **Local data** — *Variables*, may be acted upon by only one, sequential, process.
- **Shared data** — *Contexts*, may be acted upon by several processes concurrently.
- **Scope** — the process in which a name is valid.
- **Environment** — a set of scopes whose names are valid in a given process.

Figure 7.2: Data concepts

An assignment to a variable is an "internal" action; i.e., its effect is only witnessed by processes subsequent to it that share the scope of the variable, not by any process concurrent to that sequence. An *interaction* is an assignment to a context and is an "external" action; i.e., its effect may be witnessed by any concurrent (or subsequent) process that shares the scope of the context acted upon.

7.2.1 Names, environment and scopes

Each name used in a process must be explicitly specified and is considered unique; i.e., a name cannot have two meanings. Specifying an existent name will have the effect of masking the old meaning of the name for the duration of the new scope. We shall cover the details of how to specify names in the following sections.

Names represent constant values, variables, contexts, types, procedures and functions.

A *scope* is the process in which a name is valid. An *environment* is the set of scopes whose names are valid in a given process.

7.2.2 Expressions

An expression has a value, a determined value given by the evaluation of the expression. Variables may appear in expressions, and so may contexts under circumstances that will be explained later. Expressions and thus the functions that appear within them have no side–effects.

7.2.3 Types

All data is typed so that the types can be determined at compile time. The typing is polymorphic — in particular, literals have a polymorphic type; i.e., in the expression

$$1.0 + 1.0$$

the literal 1.0 has a default floating point type and this expression is itself considered a literal of that default type. All literals are specified to have a weak default type. However, in the expression

$$x + 1$$

where x is a variable or a constant (or, similarly, a function) the literal 1 will be a type compatible with x. Similarly, in the assignment

$$v := 1$$

the literal 1 is a type compatible with v. An assignment is defined to be an assignment of an expression of a compatible type to the given variable; i.e.,

$$v := x + 1$$

is valid only if the expression $x + 1$ is compatible with the type of v.

This typing system is flexible and enables the concise expression of programs using polymorphisms, however the use of literals should be reasonable and consistent; i.e.,

$$1.0 = 1$$

would produce a type error since it is not possible to determine the type of the expression, but

$$(x + 1.0) + 1$$

would not, since the type of 1 may be inferred from the type of $(x+1.0)$ which is dependent on x and not 1.0. It might not seem important here to allow such mixed usage, and whilst most sensible programmers are unlikely to produce an expression of the above sort — a mechanical source transformation system or high level application may.

With this brief overview of process and data let us now consider these concepts in greater detail.

7.3 Actions

7.3.1 Properties of termination

The simplest component of an *Ease* program is an *action*.

An action has properties of termination; i.e., either an action terminates or it does not terminate. Actions that have solely this behavior are denoted using the keywords `skip` and `stop`.

- `skip` — denotes an action that does nothing and then terminates.
- `stop` — denotes an action that does nothing and does not terminate.

These properties are important since they allow us to reason about *progress* and give meaning to actions and their composition. The actions Skip and Stop are analogous to the concept of a *point* in space–time. Just as a geometric point possesses no meaning except in its relation to other points, Skip and Stop actions have no meaning except in their relation to other actions. Indeed, Skip and Stop simply represent the termination characteristics of a point in computational space–time.

To continue the analogy beyond the simple one above an action can be considered an event in computational space–time described by the language semantics that reduces to Skip or Stop. An action has a semantic value that is an abstraction of the actions behavior, including the conditions under which subsequent events occur.

This view is abstract and some distance from the real machine. Given the objectives stated earlier we would like some way to make pragmatic statements about events. It is useful therefore to have some conception of "duration" for the behavior of an action.

We shall therefore consider that each action has a pragmatic value which we shall call it's "behavioral complexity". *Behavioral complexity* is a measure that accounts for the operational costs of an action, and is itself an abstraction of performance cost[2].

I shall not go into detail concerning behavioral complexity in this thesis. It is useful to us here in our semiotic endeavor to enable statements to be made that allow the programmer to make efficient and consistent use of the language.

The behavior complexity of Skip or Stop can be described as[3]

$$\mathcal{B}(Skip) = 0$$

$$\mathcal{B}(Stop) = 0$$

where $\mathcal{B}$ is a function that returns the measure of the operations involved.

[2]Behavioral complexity may not have a direct relationship to the performance cost in a particular implementation but simply be a further abstraction; an approximation of that cost.

[3]Note: $Stop \neq \infty$ since Stop has no behavior other than it's non–termination. In particular it is not equivalent to the *divergent process* which is equal to infinity. A later section describes Stop in relation to error, and divergence.

7.3.2 Assignment

An assignment assigns the value of the associated expression to the associated variable and terminates. The assignment

$$v := 1 + 1$$

assigns the value 2 to the variable called v; i.e., the assignment respecifies v to have the value 2.

A strict description of the above assignment is to say that the action "behaves like" Stop until the value of the variable named v is the value of the expression $1 + 1$, then it behaves like Skip. We view the action as a respecification of the name v for the remainder of its scope.

The behavioral complexity of the example action can be described as

$$\mathcal{B}(v := 2) + \mathcal{B}(1 + 1)$$

i.e., the behavior complexity of the assignment plus the behavior complexity of the expression.

7.3.3 Interaction

Interactions act upon shared data structures (Contexts).

A variable has a single value; i.e., an assignment to a variable will, in effect, replace the preceding value.

Unlike a variable, all contexts are considered sets and initially have the value of the empty set. Subsequently a context has a set of values defined by the actions upon it; i.e., an assignment to a context may simply add a value to the set of existing values.

Interactions are called "inputs" and "outputs". An *output* is either a Write or a Put. An *input* is either a Read or a Get.

A Write assigns the value of the associated expression to the context and terminates. The Write

$$\mathcal{K} \; ! \; 1 + 1$$

assigns (copies) the value 2 to the context called $\mathcal{K}$.

A Put assigns the value of the associated variable to the context, an undefined value is assigned to the variable, and the action terminates. The Put

$$\mathcal{K}!^{*}v$$

assigns (moves) the value of v to the context called $\mathcal{K}$ and then assigns a valid undefined value to v.

Unlike Write, a Put does not act upon general expressions and only upon a subset of variables; i.e., variable names, not subscripted names (components of arrays or tuples), or segments (subsets of arrays or tuples).

A Read assigns a value of the context to the associated variable and terminates. The Read

$$\mathcal{K}?v$$

copies a value of the context $\mathcal{K}$ to the variable called v.

A Get deletes a value from the context, assigns the value to the associated variable, and terminates. The Get

$$\mathcal{K}?^{*}v$$

deletes a value from $\mathcal{K}$ and assigns (moves) it to the variable called v.

Unlike Read, a Get does not act upon general expressions and only upon a subset of variables; i.e., variable names, not subscripted names (components of arrays or tuples), or segments (subsets of arrays or tuples).

Since a context may be empty the above defines the blocking behavior of an input, since an input cannot assign a value from an empty set.

It will be noted that whilst Read is a respecification (assignment) of a variable, and Write is a respecification of a context, Get and Put are respecifications of both a variable and a context.

Even so, the behavioral complexity of a Get or Put will often be much less than the equivalent Read or Write since Get and Put encapsulate a mechanism for exchanging values by reference (where possible), so give a particular advantage when managing non–trivial data structures. In an implementation the behavioral complexity of Get and Put should not exceed that of and equivalent Read or Write.

An interaction may also be written in a procedure style

$$\begin{array}{ll} \texttt{write}(\mathcal{K}, 1+1) \\ \texttt{put}\ \ (\mathcal{K}, v) \\ \texttt{read}\ (\mathcal{K}, v) \\ \texttt{get}\ \ (\mathcal{K}, v) \end{array}$$

each are predefined and exactly equivalent[4] to the previous syntax. Why not simply define interaction this way? The reason is a wish to maintain a distinction between procedures and actions — therefore the query(?) and pling(!) notation commonly used in CSP to express input and output are used. However, many programmers find this notation disconcerting thus these predefined procedures are provided.

[4]Even though the semantics will replace the instance of the expression with it's evaluation since these are also equivalent.

- **Assignment** — assign a variable the value of an expression.
- **Write** — assign the value of an expression to the context set.
- **Read** — assign a variable a value contained in the context set.
- **Put** — assign the value of a variable to the context set, and the value of the variable to be undefined.
- **Get** — assign a variable a value contained in the context set, and delete that value from the set.

Figure 7.3: The actions. Although *Ease* has no conception of the pointers familiar in other languages, Put and Get encapsulate a mechanism that allows the exchange of data by reference; thus providing an abstraction from the underlying memory subsystem architecture and reducing copy operations in an implementation.

7.4 Names, local and nonlocal data structures

As mentioned in the introductory remarks each name used in a program must be unique and thus first be specified in a *specification*.

There are two classes of specifications

- **declarations** give names to constants, variables and shared data spaces.
- **definitions** give names to types, procedures and functions.

The immediately following sections will discuss the first of these specification classes and type definitions.

At the heart of any programming language is the manipulation and expression of data. This section takes a comprehensive look at the data concepts that are central to the *Ease* language.

7.4.1 Constants

A *constant* name is specified in a *declaration*; i.e.

$$\texttt{let}\ \boxed{i} = 1 + 1$$

is a declaration that declares the name i, the *meaning* of this name is the value and type of the associated expression — in this case the value is 2 and has the default type, integer.

The meaning of a constant is easily determined. Wherever the constant appears it may simply be replaced by *the value of* the expression associated with it; i.e., given the above declaration is valid in our environment

$$\texttt{let } c = i$$

is exactly equivalent to

$$\texttt{let } c = 2.$$

A declaration

$$\texttt{let } i = x + 1$$

where x is a variable, is valid only if the variable x is not assigned to in the scope of i, even so

$$\texttt{let } c = i$$

is NOT equivalent to

$$\texttt{let } c = x + 1$$

since the instance of x may be invalid[5].

Constant names declared in this way can only appear in expressions. Whilst the use of a variable is invalid in any process concurrent to the sequential process for which it is specified, constant names may be used freely in all processes within the scope. Thus in addition to the conventional use of constant names (such as in Pascal) a constant name may, in fact, provide what is in essence read only access to variables in cooperations, or a snapshot copy of a principal's variable in a subordinate.

7.4.2 Variables

A *variable* is specified in a declaration called an *allocation*; i.e.,

$$\texttt{let } \boxed{v} := 0$$

is a specification that allocates a variable named v. The type and initial value of the variable is that of the associated expression — which in this case is 0 and has a default type, integer. An allocation is a special form of assignment that specifies a new name, so yes, it is also an action.

[5] As the following sections repeat the scope of a variable is invalid in a concurrent process — this may confuse some formalists and implementors who will suspect transformation problems. They may be assured that they are permitted to dismantle such problems by the introduction of additional variables or, if necessary, context invariants allowing them to isolate the effect. Programmers should note that by using variables in constant declarations they may endow the declaration with a behavioral complexity greater than zero.

```
let c = 1:
┌──────────────────────────────┐
│ ||let x := 0 : │{K?x ...}│    │
│ ||let y := 0 : │{K?y ...}│    │
└──────────────────────────────┘
```

Figure 7.4: The scope of a name is the process following the specification. The above case illustrates three scopes, those of x, y and c, in addition the *environment* of this process includes the scope of the free name $\mathcal{K}$.

A variable must always be given an initial value, if an undefined value is required then the form

$$\texttt{let } v := __- > \texttt{int}$$

specifies explicitly that the initial value of the variable is an undefined integer, that is because

$$__- > \texttt{int}$$

is an expression whose type is `int` and whose value is undefined (represented by the underscore "__").

Each specified name has associated with it a well defined *scope*.

7.4.3 Scope

Specifications can be placed almost anywhere in a program, but the part of the program in which the name has meaning is only that part of the program known as the name's "scope".

The *scope* of a specification is exactly the process that follows it and, in the case of a variable, that process must be sequential; i.e.,

```
let v := 0
:
┌───┐
│ P │
└───┘
```

where P is a sequential process; v may not be used in cooperative and subordinate processes within P. The colon ":" represents the binding of the specification "block" to the process. The scopes of other names, such as those of constants and contexts are not limited to sequences (illustrated in figure 7.4).

Ease scope is similar to the scope concepts found in languages such as Algol, Pascal, but is in many ways different from that found in Occam. In Occam variables remain in scope in parallel processes, constrained by a set of rules to maintain consistency – the first

definition of *Ease* shared these rules. It was modified for two reasons – first experience showed that programmers found these rules confusing, second since the central objective of *Ease* is to specify and efficiently construct shared data structures it was desirable to encourage programmers to utilize a single mechanism and to avoid confusing exceptions in the model.

Several specifications specified in the same specification block introduce those names simultaneously though mutual recursion is forbidden. A *specification block* is simply the text from the first `let` to a colon; i.e.,

```
let c = 1
let v := 0
:
```
P.

It is therefore considered an error to specify the same name more than once in a specification block. The naming is thus canonical (i.e., the simplest form possible), a name can only have one meaning in any given scope. The declaration of an extant name supersedes the old meaning for the duration of the new scope.

A specification block may follow another specification block, this has the effect of placing the specifications in sequence, thus

```
let c = 1 :
let v := 0 :
```
P.

introduces c and v sequentially where the earlier version introduced c and v simultaneously.

7.4.4 Shared data structures

Ease provides separately for the creation of data that can be operated upon concurrently. Each shared data structure is called a "context".

A context can be operated upon by concurrent processes that share the scope in which the data was specified. This allows the concurrent manipulation of shared data and facilitates interaction between (concurrent) extant processes.

7.4.5 Singletons and arrays

At its simplest a shared data structure is similar to a variable. The type of a shared data structure must be specified (though an implementation will provide several as predefined) since specifications will require named types; i.e.,

```
type c context [single] int
```

defines a type called c that is a context consisting of a single integer.

A data item of this type is *allocated* in a specification in a similar way to the variable specified in the previous section; i.e.,

$$\texttt{let } \mathcal{K} := c$$

specifies a name $\mathcal{K}$ of the context type c just specified. All contexts are initially assigned the value of the empty set[6]. The action

$$\mathcal{K}!e$$

Writes a value e such that the value of $\mathcal{K}$ becomes $\{e\}$. Singletons are such that, like variables, a subsequent assignment to $\mathcal{K}$ will, in effect, replace the previous value in the set $\mathcal{K}$. The action

$$\mathcal{K}?v$$

Reads that value and assigns it to the variable v. The action

$$\mathcal{K}?^*v$$

not only assigns the value to v but also deletes the value assigned from $\mathcal{K}$; i.e., subsequently $\mathcal{K} = \{\}$. Other inputs, of course, will not terminate until $\mathcal{K} \neq \{\}$.

Usefully, shared data structures that are multidimensional arrays can also be described as an aggregate of singleton contexts; i.e.,

$$\texttt{type } m \texttt{ context}[X][Y]\texttt{single int}$$

defines a type that is a shared 2–Dimensional matrix, where the components of the matrix are single integers. In fact the components may be any type including arrays, tuples or, indeed, other contexts; i.e.,

$$\texttt{type } l \texttt{ context}[X][Y]\texttt{single } (\texttt{bool}, \texttt{float64})$$

similarly defines a matrix in which the components are tuples that consist of a boolean and a floating point value.

The property described is called the *singleton* property and describes single and, importantly, data items addressable by subscription; i.e., given the allocation

$$\texttt{let } \mathcal{K} := m$$

the action

$$\mathcal{K}[x][y]?v$$

Reads the value x, y of the shared (perhaps distributed) 2D matrix $\mathcal{K}$.

[6] Thus a context allocation, like a variable allocation, is an action.

7.4.6 Streams

In describing streams I shall use the notation

$$[\pi, \mathcal{K}]$$

to indicate a totally ordered set I will call a "list", where π represents the last (or I shall say "most recent") value added to the set, and $\mathcal{K}$ represents all subsequent components of the set that may, of course, be the empty set. Similarly,

$$[\mathcal{K}, \pi]$$

will indicate a list such that π represents the first or "least recent" value added to the set and $\mathcal{K}$ represents all preceding components of the set [7].

A stream is a shared data structure the components of which are ordered according to that defined by the *contributing* processes; i.e., if a single sequential process outputs to a stream an inputting process will input values in the order defined by that sequence.

A definition

`type` c `context` `stream` `int`

defines a type called c that is a context consisting of a list of integers. An allocation

`let` $\mathcal{K} := c$

specifies a name $\mathcal{K}$ of the new type c. As before, the context is initially assigned the value of the empty set. The action

$$\mathcal{K}!e$$

Writes a value e such that the value of $\mathcal{K}$ becomes

$$[e, \mathcal{K}'].$$

As we have seen actions on streams construct ordered sets (lists) of values. An output adds a value to the set, an input Reads and may delete the least recent value; i.e., the action

$$\mathcal{K}?v$$

assigns the least recent value in the set to the variable v. The action

$$\mathcal{K}?^*v$$

not only assigns the least recent value to v but also deletes that value from $\mathcal{K}$; i.e., subsequently $\mathcal{K} = \mathcal{K}'$, where $[\mathcal{K}', \pi]$ represents the value of $\mathcal{K}$ preceding[8] the output, and π represents the value assigned to v.

[7] The concepts "most recent", "least recent" and "preceding" will be explained in more detail in a later explanation of order and interleaving.

[8] See preceding footnote.

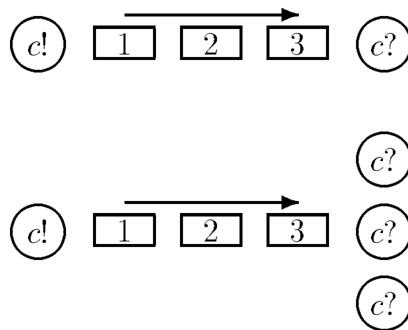


Figure 7.5: Stream context: The order of values in a stream is determined by the contributors. Here a single process (on the left of our picture) outputs three values $3, 2, 1$, inputs input the least recently output value (3 in this case).

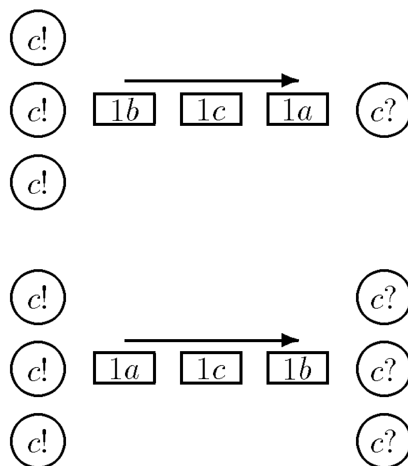


Figure 7.6: Stream context: Multiple contributors still provide a stream guarantee – a process will not input a value from a contributor output earlier than a value previously input.

7.4.7 Bags – unordered sets

Without the explicit specification of singleton or stream, by default, all contexts are unorder, just like regular sets. These contexts are called *bags*, a shared data structure the components of which are unordered; i.e., an input will make a nondeterministic selection of one of the values in the bag.

A definition

$$\texttt{type}\ c\ \texttt{context int}$$

defines a type called c that is a context consisting of a bag of integers. An allocation

$$\texttt{let}\ \mathcal{K} := c$$

specifies a name $\mathcal{K}$ of the new type c. As before, the context is initially assigned the value of the empty set. The action

$$\mathcal{K}!e$$

Writes a value e such that the value of $\mathcal{K}$ becomes $\{e, \mathcal{K}'\}$, where $\mathcal{K}'$ represents the value of $\mathcal{K}$ preceding the output.

Regular set notation is used to describe the value of bags.

Actions on bags construct (unordered) sets of values. An output adds a value to the set, an input Reads and may delete a selected value; i.e., the action

$$\mathcal{K}?v$$

assigns a value in the set to the variable v. The action

$$\mathcal{K}?^*v$$

not only assigns a value to v but also deletes the value assigned from $\mathcal{K}$; i.e., subsequently $\mathcal{K} = \mathcal{K}'$, where $\{\mathcal{K}', \pi\}$ represents the value of $\mathcal{K}$ preceding the output, and π represents the value assigned to v.

To avoid confusion it should be noted that the sets referred to are sets of distinct value bearing components; several or all of these components may hold the same value.

7.4.8 Put – review

In the preceding descriptions of actions upon contexts little mention has been made of Put. For a context a Put is equivalent to a Write, as for a variable a Get is equivalent to a Read.

The Put

$$\mathcal{K}!^*v$$

assigns an undefined value to v; i.e., a valid but unknown value of the type is assigned to v.

These semantics allow an implementation to exchange the value of v with the context by reference. Simple analysis permits an implementation to know if a memory allocation is required for the local variable that will permit subsequent use. However it will more often be the case that such variables appear in Put and Get actions in such a fashion that no memory allocation is required; i.e., a Put of a variable will often be followed by a Get with no intervening action upon the variable. Programmers can reduce the behavioral complexity of such operations by consistent use of variables that appear in Put and Get.

7.4.9 Typing, elements and expressions

I will not go into tutorial detail here of typing, elements and expressions. The full definition gives adequate coverage for our purposes here. In the following subsections I shall cover the distinguishing characteristics of these features in *Ease*, I shall not, for example, go into a detailed explanation of the mathematical operations available in the language.

Types

Values[9] are classified by their type. A type determines the set of values that may be represented by entities of that type. All typing must be determined before execution; no ambiguity is permitted to remain. All typing is name equivalent; e.g.,

```
type days is int

enum from 1 -> days
     Sunday,
     Monday,
     Tuesday,
     Wednesday,
     Thursday,
     Friday,
     Saturday

let today := Sunday

procedure timepasses ()
    test today = Saturday: today := Sunday
    else                   today := today + 1 ;
```

[9]Including those associated with variables and contexts.

Elements

An element is either

- a variable name,
- a context name,
- a subscripted variable name,
- a subscripted context name,
- or a segment.

Subscripts and segments select components of arrays. A subscript

$$a[i]$$

selects the ith component of the array. The component selected may itself be an array having one dimension less than its type for each subscript. A segment

$$a[b \texttt{ for } n]$$

is certainly an array, selecting n components of the array a from b. There are several syntactic forms of segments illustrated in the definition.

Renaming

The name identifying an element is changed by renaming; i.e.,

$$\texttt{let } n \fbox{\texttt{rename}} E$$

is a declaration that declares the name s renames the element E. In the scope of the declaration the name n is used in place of the element E. The element and any component of it may not be referred to except by using the new name in the scope.

The new name is the type and value of the element.

Expressions

Expressions have a value and a type. The value of an expression is the evaluation of the equation it expresses in the mathematical sense. Thus, expressions do not have side effects on other values.

The behavioral complexity of an expression is that of the operations within it.

The behavioral complexity of expressions is trivially extended to account for those including functions, that themselves involve actions – a function simply has the accumulated complexity of the expression and process it represents.

Although we reason about behavioral complexity here as an abstraction of performance cost in expressions, the semantics do not require the conceptualization of *computation*; i.e., the operational process of evaluation. An expression simply has a pragmatic value (distinct from it's semantic value) associated with the behavioral complexity of the operations involved in it.

7.4.10 Variants – type associative contexts

So far we have looked only at shared data structures with a single invariant type. However, a context may be defined to have several (perhaps many) types, the operations on such contexts being type associative. Given the type

```
type M context chocolate
               stream money
```

which describes a shared data structure whose components are a stream of type "money", and a bag of type "chocolate" representing a simple vending machine. The process

```
let choc := 0 -> chocolate
let coin := 0 -> money
let vm   := M
:
{   vm ?* coin
    vm !  choc
    vm ?* coin
    vm !  choc
    stop
}
```

serves two customers and then Stops. Examples like this derive from the work of C.A.R.Hoare. Here the example illustrates a context that has two types. In fact this is simply a shorthand for two distinct contexts, the appropriate context is selected for each operation by the type of the variable or expression used.

This feature of contexts allows very sophisticated construction of shared data structures.

7.4.11 Manipulating contexts

A context type may itself be a context and thus be manipulated by interaction; i.e., a context may be copied (by Read or Write) or moved (by Get or Put).

7.4.12 Automatic termination of subordinates

The scope of a context terminates as specified; i.e., the context terminates when the process the specification precedes terminates. Yet we have seen that a subordinate process may continue beyond the termination of the creating process, and thus beyond the termination of a context scope the process defines. Is the scope of the context maintained in the subordinate?

Bag and singleton contexts are deemed to terminate with the termination of the scope for which they are defined; thus if a subordinate attempts to interact with such a context it will terminate automatically. The same applies to subordinates on output to a stream component of a context.

Stream inputs are treated as an exception to the above because of the directional nature of streams. Subordinate inputs from a stream will only cause the automatic termination of the subordinate if the stream context has terminated *and is not ready*.

7.5 Composition

Compositions form processes into sequential or parallel form. Simple sequences and parallel forms have been mentioned. Recall,

- a **sequence** combines processes sequentially,
- a **cooperation** combines processes in parallel form such that subsequent processes continue only when all the processes in the composition have terminated, and
- a **subordination** composes processes in a parallel form concurrent to the subsequent processes; i.e., the parallel processes are "created" and the principal (i.e. the creating) process continues.

7.5.1 Conditional processes

Ease provides a guarded process form that allows the construction of conditional processes.

A Test is the principal form of conditional process, we saw briefly in the introduction to this chapter. It consists of processes "guarded" by boolean expressions. The process associated with the first true guard is performed, or none of them are performed; i.e., the composition behaves like Stop if no process guard evaluates true.

$$\texttt{test}\ b : P;$$

is P if b is true, and is Stop otherwise, whereas

```
test  b : P
      c : Q
      d : R;
```

is P iff b, Q iff $\neg b \wedge c$, R iff $\neg b \wedge \neg c \wedge d$, and is Stop otherwise.

A default may be specified, so that

```
test  b : P
      c : Q
      d : R
else  S;
```

is P iff b, Q iff $\neg b \wedge c$, R iff $\neg b \wedge \neg c \wedge d$, and is S otherwise.

A Selection is a special form of conditional process similar to conventional "case" constructions; i.e.,

```
select s
         x : P
         y : Q
         z : R;
```

is P iff $s = x$, Q iff $s = y$, R iff $s = z$, and is Stop otherwise. Similarly

```
select s
         x : P
         y : Q
         z : R
else     S;
```

is P iff $s = x$, Q iff $s = y$, R iff $s = z$, and is S otherwise.

A While is a recursive form of conditional process and is simply defined as

$$\texttt{while}\ bP \equiv \begin{cases} \texttt{test} & b : \{P\ \texttt{while}\ bP\} \\ & (\neg b) : \texttt{skip} \\ ; & \end{cases}$$

and

$$\texttt{do}\ P\ \texttt{until}\ b \equiv \begin{cases} \{ & P \\ & \texttt{while}\ (\neg b)P \\ \} & \end{cases}$$

7.5.2 Nondeterministic choice

Nondeterminism is introduced into a program by the nondeterministic selection of values from a bag context. The programmer may also introduce nondeterminism via the use of an explicit construct of nondeterministic choice. The choice

```
choice  k?v : P
        l?v : Q;
```

is the sequence

```
{ k?v
  P   }
```

if the input $k?v$ can be satisfied (we say "is ready"), or is the sequence

```
{ l?v
  Q   }
```

if the input $l?v$ is ready. If both inputs are ready then the choice behaves like only one of the possible sequences — which one is nondeterministic. If neither choice is ready the composition behaves like Stop until one or more possible sequences become ready and a choice can be made. A default may be specified as for Test; i.e.,

```
choice  k?v : P
        l?v : Q
else    R;
```

is R iff neither of the inputs is initially ready.

Full nondeterminism can be introduced by the construction

```
choice  |P
        |Q
        ;
```

in this case the components $|P$ and $|Q$ are always ready. This construction is particularly useful in applications that contain two or more implementations of the same behavior and the compiler or run time system may choose which one of them to use; such applications can reasonably be expected to choose the most efficient functioning component, indeed

```
choice  |P     = P.
        |stop
        ;
```

Complex nondeterministic constructions are easily built using choice and they should be treated with care. For example, in the above case P and Q may both start with an output (recall, an output is always ready). The programmer may expect that during the course

of a program both outputs will occur. This is not the case. Since choice selection is fully nondeterministic an implementation may choose to implement P and not bother with Q, or vice versa. Similarly, an implementation may always prefer to select the input $k?v$ over $l?v$, illustrated in the earlier example, when both are ready.

This issue is often referred to as "fairness"; i.e., that a choice should prefer the selection of the least frequently instanced process. Such choices can be constructed by using boolean guards; i.e.,

```
choice  a <= b|a ++
        b <= a|b ++
        ;
```

implements a fair choice between its components; where initially $a = b$. A false boolean guard excludes the component from the choice; so that, even though the first component may always be chosen when $a = b$, it is excluded from the choice on the subsequent instance allowing the second component to be performed.

The programmer is also cautioned of the distinction

```
choice  k?v : P  ≠  choice  |  {  k?v
        l?v : Q                   P    }
        ;                   |  {  l?v
                                  Q    }
                            ;
```

where the selection on the right is independent of the readiness of the inputs; and for the same reason

```
choice  | {  k!v       ≠  choice  |  {  k!v     .
             P    }                     P    }
        l?v : Q                   |  {  l?v
        ;                               Q    }
                                  ;
```

A choice that includes

```
|skip
```

or

```
b|skip
```

is invalid. It is also worth noting that

```
choice  |P
else    Q;
```

is always P.

7.6 Resources

The construction of reusable resource processes is an essential requirement in any system. Resources may manage specific, specialized, aspects of a system such as a printer, database or, perhaps, a vector processor.

Utilization of resources by specific processes demands, in fact, close synchronization. This synchronization provides claim/release mechanisms and ensures that responses issued by the resource return to the correct process.

Such resources have two forms. *Statically reusable* resources require management that ensures that their use is *pro rata*. *Virtual resources* behave like the resource but are actually either simulations of the behavior of the real resource, utilizing the real resource when available, or provide additional access to the real resource.

A statically reusable resource is usually one whose simulated behavior is not useful or whose shared access is either undesirable or impossible, for example, a vector processor. Such a resource must synchronize with the using process, act upon the data provided in a prescribed way and perhaps return a result.

A virtual resource is a process, whose behavior may be a delegation of usage of the real resource, or may provide additional access to a real resource. A print spooler common to most operating systems may be said to be a virtual resource. A database system that manifests processes providing additional access on demand may also be considered a virtual resource.

7.6.1 Combinations

Combinations provide the essential ingredients of resource access, and subordinate processes provide the dynamic process creation required for the dynamic creation of virtual resources.

A call behaves like an output to a context followed by a get; i.e., the call

$$\mathcal{K}!e? * v$$

behaves like the sequence

$$\{ \quad \begin{array}{l} \mathcal{K}!e \\ \mathcal{K}?^{*}v \quad \}. \end{array}$$

A resource behaves like a get from a context followed by a process and a subsequent output; i.e., the resource

$$\texttt{resource}\ \mathcal{K}?^{*}v : P!e$$

behaves like the sequence

$$\{\ \mathcal{K}?^*x \quad P(x) \quad \mathcal{K}!e\ \}.$$

In addition these combinations provide a guarantee that the output from a resource will satisfy the get of the call that supplied the input data.

$$\texttt{resource}\ \mathcal{K}?^*x!f(x)$$

is a convenient abbreviation defined for a common form of combination that uses functions in place of procedures. It is equivalent to the combination

$$\texttt{resource}\ \mathcal{K}?^*x : y := f(x)!y.$$

The reader will note a strong similarity to concepts of remote procedure call; indeed, combinations are sophisticated mechanisms providing similar functionality.

A call that first behaves like a put to a context followed by a get; i.e., the call

$$\mathcal{K}! * x? * x$$

behaves like the sequence

$$\{\ \mathcal{K}!^*v \quad \mathcal{K}?^*v\ \}$$

allows the exchange of data between the resource and the call to be implemented as exchange by reference.

A resource that finishes with a put completes the exchange; i.e.,

$$\texttt{resource}\ \mathcal{K}?^*x : P! * x$$

behaves like the sequence

$$\{\ \mathcal{K}?^*x \quad P(x) \quad \mathcal{K}!^*x\ \}.$$

Combinations can prove extremely effective where exchange by reference of non–trivial data structures is a benefit.

7.7 Placement

Contexts and subordinates can be explicitly placed on the nodes of a machine. A directive

$$\texttt{let}\ k := c\ \texttt{on}\ n\ \texttt{at}\ a$$

allocates the context k on node n at address a. This provides important support for embedded systems and in particular allows a singleton to be identified with a memory mapped device. An implementation may provide for the specification of several nodes by using a table type, so

$$\texttt{let}\ k := c\ \texttt{on}\ [0, 1, 3]$$

allocates the context k and distributes its implementation over nodes 0, 1, and 3.

$$//\texttt{on}\ i : P;$$

and,

$$//\texttt{on}\ i\ \texttt{for}\ n : P(i);$$

allows subordinate processes to be explicitly placed on remote nodes of a machine, where a node is regarded as a part of the machine with a common address space; regardless of the number of processors that share that space. Cooperations are required to be instanced on a single node. Thus

$$//\texttt{on}\ i : ||j\ \texttt{for}\ n : P(j);$$

can be used to describe the parallel execution of P on a remote SIMD node (i).

Explicit placement is provided as a first order feature of the language to provide support primarily for automatic source–to–source transformation. An implementation of *Ease* may provide such optimizations for each target implementation to map a program to available resources. Higher level languages that target *Ease* may choose to use the same optimizers or introduce placement directives explicitly.

7.8 Undetermined values, invalidity and divergence

Processes that include expressions (assignment, Write and guarded processes) behave like Stop if the value of the associated expression is undetermined (denoted by the symbol $\perp$). The following paragraphs define the meaning of undetermined values.

An undetermined value is the result of an expression that is *invalid*. An invalid expression is one that has

- a typographical syntax error,

- a type incompatibility,
- a use of an unspecified name[10],
- a value not included in the range expressible by a type[11][12], or
- a function that *diverges* or *deadlocks*[13].

Divergence is a *racing process*; e.g.,

$$\texttt{while}\ \mathit{true} : \texttt{skip}$$

A *divergent process* behaves like an infinite number of internal actions.

Where these actions are hidden this divergence will often take the form of *live lock*, where some set of processes interact infinitely between themselves.

Deadlock is a *stopped process*; e.g., a process that infinitely waits for an input to be satisfied.

An observer is unable to distinguish between deadlock and divergence since the divergent process will never interact with the observer. Nor is it possible for the observer to detect the fact.

Divergence is discussed in detail in [Ros86b], and to a lesser degree in [Ros86a].

7.8.1 Detectibility of undetermined values

Detectability of undetermined values is a pragmatic concern. The degree of successful detection is entirely dependent on the sophistication of an implementation.

Invalid expressions are usually detectable. A parser will detect typographical errors, usage checking will detect type incompatibility and unspecified variable names. Run time exception handling is often required to detect values not defined for the range of a type, for example in the case of arithmetic overflow or division by zero.

It is impossible to detect an undetermined value that results from the divergence of a function since even a privileged observer, i.e. one able to "peek inside" the process, sees progress. The observer cannot discount therefore that the process might yet terminate or interact.

---

[10]Including the use of a name in scope but which is *not specified* for use in a parallel construction; i.e., use of a free variable in a parallel construction.

[11]This statement encompasses arithmetic overflow, underflow, division by zero etc..

[12]The IEEE/ANSI 754 floating point definition provides values specified for the behavior of floating point operations (such as Not-A-Number). These values are considered *valid* for the purposes of this definition, and are represented in an implementation of the language by a compatible set of predefined constants.

[13]A function can only deadlock or diverge internally.

7.9 Exception handling

Where undetermined values can be detected it is often desirable to define alternative behavior. Such detectable instances at run time are called *exceptions* and can often be predicted by programmer concern.

To allow the programmer to express such concern *Ease* provides for the definition of alternative behavior. The sequential process

$$\{\ \texttt{on stop}\ E \quad P\ \}$$

performs E if P should stop in a way that is detectable. E and P share the same scope and thus have access to exactly the same free environment. The behavior of P up to the Stop action is not discarded and will be reflected in changes to that environment. Consider the example

```
{ on stop write(signal, "Expression error on %"(node))
  x = m * c + y
}
```

or

```
{ on stop fptrap()
  ...
  test e = NaN : stop else skip
}
```

Chapter 8

Definition of Ease

8.1 About the definition

The definition of *Ease* is constructed through the use of several interrelated *models* which constitute the semiotic definition of the language.

These models are respectively *syntactic*, *semantic* and *pragmatic* in nature. The syntactic model defines valid syntactic constructions of the language. The semantic model defines the meaning[1] of syntactic constructions. The pragmatic model provides a measure of performance cost and makes statements that help the programmer to make consistent and efficient use of the language.

- the **syntactic model** is defined by a BNF grammar and a set of *validity rules*.
- the **semantic model** is defined here by a set of simple informal statements and more formally by a CSP description.
- the **pragmatic model** consists of a set of informal statements designed to help the programmer make consistent and efficient use of the language.

8.1.1 Syntax

The BNF grammar which characterizes *Ease* and is presented in the following sections does not, indeed cannot, fully characterize all valid the syntactic constructions of the language since it is unable make statements which capture such things as

- A name used in an expression is only valid if the name has been *specified*.

[1] More accurately, describes the *behavior* represented by the syntactic constructions.

Therefore, the BNF characterization of the language syntax is accompanied by a set of *validity rules* which constrain the possible syntactic constructions defined by the BNF notation.

The BNF context free grammar used to characterize the syntax (and presented in full in the appendix) of the language presented here generates a LR parser from YACC or Bison without the use of precedence rules and with no reduction conflicts.

Modified BNF

The modified BNF[2] definitions specify a context free grammar which describes the syntax of the language.

In the following modified BNF

$$\langle_n item\rangle$$

means, **a textual sequence of** n **or more** *item*.

$$\langle_n \boxed{,}\ item\rangle$$

means, **a textual sequence of** n **or more** *item* **separated by commas.**

A circle ($\circ$) represents newline. Thus

$$\langle_n \circ\ item\rangle$$

means, **a textual sequence of** n **or more** *item* **separated by newlines.**

$$\begin{aligned} construction = \ & sequence \\ \mid\ & test \\ \mid\ & selection \\ \mid\ & combination \\ \mid\ & choice \\ \mid\ & repetition \end{aligned}$$

means, **a** *construction* **is a** *sequence*, **or a** *test*, **or a** *selection*, **or a** *combination* **or a** *choice*, **or a** *repetition*. It is equivalent to

$$\begin{aligned} construction &= sequence \\ construction &= test \\ construction &= selection \\ construction &= combination \\ construction &= choice \\ construction &= repetition \end{aligned}$$

[2]Backus Naur Form.

Keywords are not case sensitive in the language, but appear in the definition in upper case and boxed to distinguish them. With the exception of newlines all symbols of the language appear boxed. For example, in the definition

$$
\begin{array}{|lcl}
\textit{test} & = & \boxed{\texttt{TEST}}\ \langle_0 \circ \textit{conditional}\rangle\ \boxed{;} \\
 & | & \boxed{\texttt{TEST}}\ \langle_0 \circ \textit{conditional}\rangle\ \boxed{\texttt{ELSE}}\ \textit{process}\ \boxed{;} \\
\textit{conditional} & = & \textit{boolean}\ \boxed{:}\ \textit{process} \\
 & | & \textit{boolean} \circ \textit{process} \\
 & | & \textit{test}
\end{array}
$$

`TEST` and `ELSE` are keywords, "∘" represents a newline, symbols are highlighted in boxes; e.g., ";" marks the end of a test construction. When keywords and symbols appear in the accompanying text and mathematics they appear in a `typewriter` font.

The vertical bar which appears along the left side of a modified BNF definition is a loose grouping construct only, whose primary purpose is to distinguish such definitions from the main body of text.

8.1.2 Validity statements

In addition to the BNF syntactic descriptions validity rules define valid sentences in the language; i.e., the validity statement

example
Validity 1 (of assignment) *An assignment is valid only if the type of the variable and the expression are the same.*

constrains the syntax of assignment to values of the same type.

8.1.3 Equivalence statements

The meaning of some syntactic descriptions are defined equivalent to more primitive syntactic components; i.e., the equivalence

example

Equivalence 1 *The assignment*

$$v++$$

where v is a scalar variable of integer or float type is an abbreviation defined by

$$v{+}{+} \;\equiv\; v := v + 1$$

defines the meaning of $v++$ in terms of a more primitive assignment.

8.1.4 Context free grammar

The following definition includes a complete context free grammar; a version of this grammar is suitable for use with YACC[3] or Bison parser generators.

The complete grammar appears in the appendix and includes precise specifications for the appearance of line breaks in a program.

8.2 *Ease* pragmatics

This section introduces aspects of the language and how they relate to the programmer.

8.2.1 The process model

A process is a behavior pattern which consists of an action or combination of actions in local scopes.

8.2.2 Machine and process

A *machine*[4] is a device which provides the *resource* required to manifest the *behavior* of a *program*.

A *program* is a collection of *processes* combined to express the behavioral description of some *application*.

Resource includes the physical manifestation of the machine and the exact implementation of the language described in the following sections.

[3] Although use with YACC may require that you rebuild your version of YACC with larger table space.
[4] Such as a general purpose or specialized computer or robot.

Aspects of a machine's physical resource are finite. A program may behave incorrectly if the machine provides inadequate resource. To contribute to the correct behavior of a program, more than adequate resource must be guaranteed during the interval of its manifestation.

An *application* consists of some composite of *algorithms*. An application may have temporal requirements which cannot be met by the available resource, though a program can complete its manifestation. Such instances demonstrate an inadequacy in the specification of the machine.

A *process* is the expression of an algorithm or some component of an algorithm.

8.2.3 Contexts and interaction

A *context* is a distributed data structure, which has a well defined scope. The components of a context are some well defined type.

Write and Read operations copy values to and from a context.

Put and Get operations move value representations (variables and contexts) to and from a context.

Entities moved by *put* and *get* can be considered distinct. In implementation this enables data which exists in a shared address space to be passed from one process to another by reference.

Interaction decouples the sender and receiver of a value.

Groups of processes construct, creating and modifying, data structures distributed among them. Groups of processes interact uniformly via contexts.

8.2.4 Expressing concurrency

Programs are expressed in parallel form to

- enable simple elucidation of the disjoint components of the program, or to
- to introduce some particular performance semantics.

Algorithmic decomposition

A program can be expressed as the component, interacting, algorithms of an application.

Performance semantics

Performance semantics derive from the use of a construction in anticipation that such use will benefit performance of an algorithm.

Performance semantics are introduced to a program in three forms

1. Overlapping interaction and computation.
2. Computation decomposition.
3. Priority.

The use of the first form, overlapping interaction and computation, derives from an *understanding* that interaction on the particular structure is not an instantaneous operation (e.g., it is known to be on a remote node in a distributed system) and there exists computation which has no dependency on the data involved in the interaction.

The use of the second form, computation decomposition, derives from an *understanding* that concurrent computation can be divided and distributed across available resource and that there exists no direct dependency between each component of the decomposition. Indirect dependencies, such as those in a *pipeline*, are expressed as interactions in one or more *contexts*.

The use of the third form, priority, derives from a *requirement* that interactions on some data structures be completed in preference to interactions on others.

Programs dependent on performance semantics will behave differently according to the architecture and resource of a particular machine.

The performance semantics of a program can be altered by program transformation.

Cooperation and subordination

Concurrency can be expressed as either cooperative or subordinate. Cooperation parallelism is characterized by parallel operations on shared local data. Subordinate parallelism, is characterized by subordinate operations on globally (i.e. some broader locality) shared data.

These two forms are pragmatically distinct and they require separate syntactic constructions to express their semantics.

Cooperative parallelism is manifest as concurrency between processes which share a common address space. Such processes are commonly implemented on uniprocessor or shared memory multiprocessors.

Subordinate parallelism is manifest as concurrent management of resources, and concurrency between processes not sharing local data dependency, acting upon global, perhaps distributed, data.

Automatic release and speculative processing

Subordinate processes terminate if they attempt to interact with a context whose scope has terminated or whose value has been output. This provides two powerful pragmatics:

- automatic release of allocated resources, and
- speculative computation.

8.2.5 The pragmatics of the parallel composition

Parallel composition involves a pragmatic consideration since there will be sequences existent which possess complete independence from those specified (perhaps the execution of some other program, perhaps the power supply of a machine). The pragmatic demands that these sequences are "well behaved"; i.e., they do not, under any circumstance, affect the behavior of the parallel[5].

8.2.6 Resources

The construction of and interaction with resources has special requirements.

A *combination* provides guaranteed *call reply* semantics via some context. Access to system resources is provided by use of combinations.

Resources are either *statically reusable* or *virtualized*. The pragmatic distinction between the two types carries a performance semantic. Static resources synchronize the behavior of the actual resource with the calling process. Virtual resources provide resource response to a process whilst the actual resource may be *busy*, and the actual resource behavior may be delayed.

8.2.7 Failure and error handling

Detectable errors and failures may be gathered in a global *signal* context. Global error handlers can be simply specified to act on instances of error data.

Local error handlers are built using special sequence constructions.

[5] Clearly this is not an assurance against the "acts of some god" or, indeed, errant programs in the same system but it is a pragmatic which the programmer must accept.

8.2.8 Definitions and instances

A definition provides an algebraic abstraction whose meaning is defined in an instance of the name defined.

8.2.9 Allocation

An allocation allocates memory of the type specified.

8.2.10 Typing

All typing information can be detected by a translator at the time of translation. Typing is name equivalent.

8.2.11 Expressions

The type of an expression is determined in it's instance. For example, $1 + 2$ defines an expression. Let i be a variable of integer type, then in the instance

$$i := 1 + 2$$

the expression $1 + 2$ has the value 3 and is of type integer. Thus, let r be a variable of float type, then in the instance

$$r := 1 + 2$$

the expression $1 + 2$ has the value 3 and is of type float.

8.2.12 Procedures

Procedures name processes. A procedure is defined by textual substitution of the process named.

A procedure may be translated by either substitution of its textual translation, or where it is used more than once in a sequential construction or cooperation, as a closed subroutine.

8.2.13 Lambda expressions (functions)

Functions name lambda expressions. A function is defined by textual substitution of the lambda expression named.

Lambda expressions do not produce side effects. Lambda expressions which use subordination, interaction or choice internally are nondeterministic.

8.2.14 Modules

Modules are defined by textual substitution. The names specified in the hide portion of a module are given a canonical form, that is, they are renamed to unknown names which do not conflict with names in the current or subsequent scope.

8.2.15 Stylistic conventions

Programs must be written clearly, with useful supporting comments.

A program may be considered invalid if its lay out is ambiguous or the associated comments vague.

Automatic generation of *Ease* is also required to follow these pragmatics.

8.2.16 Alien language processes

Processes written in a conventional language may take the place of any process in *Ease*, provided they meet the rules for parallel processes.

Such processes may perform interaction, using variants of interactions which are semantically equivalent to those defined by *Ease*.

Alien languages which exist in the *Ease* abstract system, are *modified* to comply with *Ease* reference semantics. Aspects of the alien language which cannot be defined in terms of the *Ease* reference semantics must be omitted from the implementation or constrained to meet the requirements of *Ease* semantics.

Alien languages may be extended to include constructors and interaction operations which are semantically equivalent to those specified by the *Ease* reference specification. Names of keywords are implementation specific; however, the *Ease* aspects of the implementation must be some semantic subset of the *Ease* definition. Supersets must be defined to be algebraically equivalent to the reference semantics. Types and type definition must have defined equivalence to the *Ease* reference semantics.

8.3 *Ease* syntax and semantics

The following sections define the language syntax supplemented by semantic statements. *Ease* constructions have semantics compatible with CSP mathematics (as defined in[Hoa85]) for similar constructions — a CSP definition of the *Ease* interaction model is given separately in CSP.

The syntax is described using a modified BNF[6] plus additional validity rules which cover typing and usage requirements.

8.3.1 Actions

$$\begin{aligned} action = \; & assignment \mid interaction \\ & \mid skip \mid stop \end{aligned}$$

Stop and Skip

$$skip = \boxed{\texttt{SKIP}}$$
$$stop = \boxed{\texttt{STOP}}$$

Assignment

$$assignment = variable \; \boxed{\texttt{:=}} \; expression$$

$$variable = element$$

An assignment assigns the value of the expression to the variable.

Validity 8.1 (of assignment) *An assignment is valid only if the type of the variable and the expression are the same.*

Abbreviations for increment and decrement

$$\begin{aligned} assignment = \; & variable \; \boxed{\texttt{++}} \\ \mid \; & variable \; \boxed{\texttt{--}} \end{aligned}$$

Equivalence 8.1 *The assignment*

$$v++$$

where v is a scalar variable of integer or float type is an abbreviation defined by[7]

$$v{+}{+} \stackrel{def}{=} v := v + 1$$

[6] Backus-Naur Form.

[7] Note that literals are polymorphic.

Equivalence 8.2 *The assignment*

$$v--$$

where v is a scalar variable of integer or float type is an abbreviation defined by

$$v-- \stackrel{def}{=} v := v - 1$$

Interaction

$$
\begin{array}{lll}
interaction & = & input \mid output \\
\\
input & = & read \mid get \\
output & = & write \mid put \\
\\
read & = & context \; \boxed{\texttt{?}} \; variable \\
write & = & context \; \boxed{\texttt{!}} \; expression \\
\\
get & = & context \; \boxed{\texttt{?*}} \; reference \\
put & = & context \; \boxed{\texttt{!*}} \; reference \\
\\
context & = & element \\
reference & = & name
\end{array}
$$

A *read* assigns a value from the context to the variable.

A *write* assigns the value of the expression to the context.

A *get* binds a value in the context to the name, removes the binding from the context (and thereby the value).

A *put* binds the value of the name to the specified context, removes the binding from the name. The value of the name is subsequently undefined.

Validity 8.2 (of interaction) *An interaction is valid only if the type of the name, variable or expression is a member of the type set defined for the context.*

Abbreviations for read and write

The behavior of a write $k \,!\, e$ where k is a context, and e is an expression, is equivalent to the behavior of an assignment $k := e$ (given the relevant type characteristics).

Equally, the behavior of a read $k\ ?\ v$ where k is a context, and v is a variable, is equivalent to the behavior of an assignment $v := k$.

In practice the validity rules only allow a context of invariant type to appear in an assignment since there is no variant first class data type.

The following equivalences therefore define useful abbreviations.

Equivalence 8.3 *An assignment $k := e$ where k is a context of invariant type, and e is an expression of that type, is an abbreviation equivalent to a write in the context k. Thus*

$$k := e \quad \stackrel{def}{=} \quad k\ !\ e$$

Equivalence 8.4 *An assignment $v := e[k]$ where v is a variable and k is a context of invariant type which appears in the expression e, is an abbreviation equivalent to a read from the context k to a temporary variable of the type defined by the context and a subsequent assignment. Thus*

$$v := e[k] \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll} \{ & k\ ?\ t \\ & v := e[t] \quad \} \end{array} \right.$$

where t is a temporary variable of a type compatible with the invariant type of k. A context may only appear once in such an expression.

Thus an *assignment* $k := e[k']$ where k is a context of invariant type, and k' is a context of invariant type which appears in the expression e is defined by

$$k := e[k'] \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll} \{ & k'\ ?\ t \\ & k := e[t] \quad \} \end{array} \right. \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll} \{ & k'\ ?\ t \\ & k\ !\ e[t] \quad \} \end{array} \right.$$

where t is a temporary variable of a type compatible with the invariant type of k'.

Abbreviations for sequences of similar actions

$$\begin{array}{|lcl} read & = & context\ \boxed{?}\ \langle_2 \boxed{,}\ variable \rangle \\ write & = & context\ \boxed{!}\ \langle_2 \boxed{,}\ expression \rangle \\ & & \\ get & = & context\ \boxed{?*}\ \langle_2 \boxed{,}\ name \rangle \\ put & = & context\ \boxed{!*}\ \langle_2 \boxed{,}\ name \rangle \end{array}$$

Equivalence 8.5 *A sequence of interactions on the same context and of the same operation can be abbreviated. Let χ be an interaction operator such that $\chi \in \{!, !^*, ?, ?^*\}$. Let k be a context and a, b and c be compatible expressions, variables or names, then*

$$k\ \chi\ a, b, c \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll} \{ & k\ \chi\ a \\ & k\ \chi\ b \\ & k\ \chi\ c \quad \} \end{array} \right.$$

8.3.2 Constructions

Processes

$$process = action \mid construction$$

Sequential construction

$$\begin{aligned} construction = \; & sequence \\ \mid \; & test \\ \mid \; & selection \\ \mid \; & combination \\ \mid \; & choice \end{aligned}$$

Sequence

$$sequence = \boxed{\{} \; \langle_0 \circ process \rangle \; \boxed{\}}$$

A *sequence* performs each of its components sequentially.

Test

$$\begin{aligned} test \; = \; & \boxed{\texttt{TEST}} \; \langle_0 \circ conditional \rangle \; \boxed{;} \\ \mid \; & \boxed{\texttt{TEST}} \; \langle_0 \circ conditional \rangle \\ & \boxed{\texttt{ELSE}} \; process \; \boxed{;} \\ conditional \; = \; & boolean \; \boxed{:} \; process \\ \mid \; & boolean \circ process \\ \mid \; & test \end{aligned}$$

A test is the sequential composition of guarded processes called *conditionals*. A conditional is a process *guarded* by a boolean expression.

Conditionals are tested in sequence; if the guard is true the associated process is performed, if the guard is false the subsequent conditional is tested. If there is no subsequent conditional the construction behaves like stop or the default component if one exists.

A test which includes an `ELSE` specifies a default component process, which is performed if no conditional is performed.

Selection

$$
\begin{array}{lcl}
\mathit{selection} & = & \boxed{\texttt{SELECT}}\ \mathit{selector}\ \boxed{:}\ \langle_1 \circ \mathit{option}\rangle\ \boxed{;} \\
 & | & \boxed{\texttt{SELECT}}\ \mathit{selector} \circ \langle_1 \circ \mathit{option}\rangle\ \boxed{;} \\
 & | & \boxed{\texttt{SELECT}}\ \mathit{selector}\ \boxed{:}\ \langle_1 \circ \mathit{option}\rangle \\
 & & \boxed{\texttt{ELSE}}\ \mathit{process}\ \boxed{;} \\
 & | & \boxed{\texttt{SELECT}}\ \mathit{selector} \circ \langle_1 \circ \mathit{option}\rangle \\
 & & \boxed{\texttt{ELSE}}\ \mathit{process}\ \boxed{;} \\
\mathit{selector} & = & \mathit{expression} \\
\mathit{option} & = & \mathit{match_list}\ \boxed{:}\ \mathit{process} \\
 & | & \mathit{match_list} \circ \mathit{process} \\
\mathit{match_list} & = & \langle_1 \boxed{,}\ \mathit{expression}\rangle
\end{array}
$$

A *selection* is a sequential composition of options; *options* are processes guarded by a match list.

The value of the *selector* is used to select an option.

Each expression of a match list must be type compatible with the type of the selector. All the expressions used in the match lists of a selection must be distinct[8].

A selection which includes an `ELSE` specifies a default component process.

Equivalence 8.6 *Let s be an expression, e^n be an expression list (i.e., match list) with n components, i be an index, and P be a process, then*[9]

$$\texttt{SELECT}\ s : e^n P; \quad \stackrel{def}{=} \quad \texttt{TEST}\ i\ \texttt{FOR}\ n : (s = e^i)P;$$

Let Q be a process, then

$$\begin{array}{ll}\texttt{SELECT}\ s: & e^n P \\ & \texttt{ELSE}\ Q;\end{array} \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll}\texttt{TEST} & i\ \texttt{FOR}\ n : (s = e^i)P \\ & \texttt{ELSE}\ Q;\end{array}\right.$$

Let e^m be an expression list with m components, then

$$\begin{array}{ll}\texttt{SELECT}\ s: & e^n P \\ & \vdots \\ & e^m Q;\end{array} \quad \stackrel{def}{=} \quad \left\{ \begin{array}{l}\texttt{TEST} \\ \qquad \texttt{TEST}\ i\ \texttt{FOR}\ n : (s = e^i)P; \\ \qquad \vdots \\ \qquad \texttt{TEST}\ i\ \texttt{FOR}\ m : (s = e^i)Q; \\ ;\end{array}\right.$$

---

[8]This requirement allows selection to be efficiently implemented by table look up.

[9]There is a forward reference here to replication.

and

$$\texttt{SELECT } s: \begin{array}{l} e^nP \\ \vdots \\ e^mQ \\ \texttt{ELSE } R; \end{array} \quad \stackrel{def}{=} \quad \left\{ \begin{array}{ll} \texttt{TEST} & \\ & \texttt{TEST } i \texttt{ FOR } n : (s = e^i)P; \\ & \vdots \\ & \texttt{TEST } i \texttt{ FOR } m : (s = e^i)Q; \\ & \texttt{ELSE } R \\ ; & \end{array} \right.$$

Combination

$$\begin{array}{lcl} \textit{combination} & = & \textit{call} \mid \textit{reply} \\ \textit{call} & = & \textit{context} \ \boxed{\texttt{!}} \ \textit{expression} \ \boxed{\texttt{?*}} \ \textit{name} \\ & & \mid \textit{context} \ \boxed{\texttt{!*}} \ \textit{name} \ \boxed{\texttt{?*}} \ \textit{name} \\ \textit{reply} & = & \boxed{\texttt{RESOURCE}} \ \textit{context} \ \boxed{\texttt{?*}} \ \textit{name} \ \boxed{\texttt{:}} \\ & & \textit{process} \\ & & \boxed{\texttt{!}} \ \textit{expression} \\ & & \mid \boxed{\texttt{RESOURCE}} \ \textit{context} \ \boxed{\texttt{?*}} \ \textit{name} \circ \\ & & \textit{process} \\ & & \boxed{\texttt{!}} \ \textit{name} \\ & & \mid \boxed{\texttt{RESOURCE}} \ \textit{context} \ \boxed{\texttt{?*}} \ \textit{name} \ \boxed{\texttt{:}} \\ & & \textit{process} \\ & & \boxed{\texttt{!*}} \ \textit{name} \\ & & \mid \boxed{\texttt{RESOURCE}} \ \textit{context} \ \boxed{\texttt{?*}} \ \textit{name} \circ \\ & & \textit{process} \\ & & \boxed{\texttt{!*}} \ \textit{name} \end{array}$$

A *combination* synchronizes two processes interacting via a common context.

A *call* behaves like an output and get in sequence.

A *reply* behaves like a get, process and output in sequence.

Validity 8.3 *The process associated with the reply is invalid if it contains references of a call reply type to the context associated with the combination.*

The behavior of a combination is given a more formal description in the following section's CSP semantics.

Equivalence 8.7 *A convenient abbreviation is*

$$RESOURCE\ c?^{*}x!f(x) \stackrel{def}{=} RESOURCE\ c?^{*}x : y := f(x)!y$$

Choice

$$
\begin{array}{lcl}
choice & = & \boxed{\texttt{CHOICE}}\ \langle_0 \circ alternative \rangle\ \boxed{;} \\
 & | & \boxed{\texttt{CHOICE}}\ \langle_0 \circ alternative \rangle \\
 & & default\ \boxed{;} \\
 & & \\
alternative & = & determined \\
 & | & boolean\ \boxed{:}\ determined \\
 & | & boolean \circ determined \\
 & | & nondetermined \\
 & | & booleannondetermined \\
 & | & choice \\
 & & \\
default & = & \boxed{\texttt{ELSE}}\ process \\
 & | & \boxed{\texttt{ELSE}}\ \boxed{\texttt{AFTER}}\ time\ \boxed{:}\ process \\
 & | & \boxed{\texttt{ELSE}}\ \boxed{\texttt{AFTER}}\ time \circ process \\
 & & \\
determined & = & input\ \boxed{:}\ process \\
 & | & input \circ process \\
 & | & reply \\
 & & \\
nondetermined & = & \boxed{|}\ process \\
 & & \\
time & = & expression
\end{array}
$$

A choice is the composition of a number of alternative processes. A choice performs one, and only one, of its ready *alternatives*. A nondeterministic choice is made between ready components.

A *nondetermined* alternative is always ready.

The readiness of a *determined* alternative is determined by an input guard. An input guard is ready if the input can be satisfied.

An alternative guarded by a boolean is excluded from the choice if the boolean is false.

A selected *determined* alternative behaves like the input and the associated process in sequence.

If no alternative is ready the choice behaves like Stop until an alternative is ready.

A choice may include a default process. This process is chosen if no other component of the choice is ready.

A choice which includes an `ELSE AFTER` *time* default specifies a default component process. A *time* is a float expression which represents a period of time which limits the continuation of a choice; 1.0 = one millisecond. If none of the alternatives is ready within the specified time, the default process is performed.

Equivalence 8.8 *In choice*

$$\texttt{ELSE}\ P \stackrel{def}{=} \texttt{ELSE AFTER}\ 0 : P$$

Validity 8.4 *A choice is invalid if one of its alternatives is*

$$|SKIP.$$

The readiness of an alternative that is itself a choice is determined by its alternative components which may include a default; i.e.,

$$\begin{array}{ll} \texttt{CHOICE} & A \\ & \texttt{CHOICE}\ B\ \texttt{ELSE}\ P(); \\ & \texttt{ELSE}\ Q(); \end{array} \stackrel{def}{=} \begin{array}{ll} \texttt{CHOICE} & A \\ & B \\ & \texttt{ELSE CHOICE}\ |P()|Q();; \end{array}$$

Concurrent construction

$$\left|\begin{array}{lcl} construction & = & cooperation \\ & & |\ subordination \end{array}\right.$$

Concurrent constructions cause their components to be performed simultaneously.

Cooperating parallel processes

$$\left|\begin{array}{lcl} cooperation & = & \langle_1 associate\rangle \boxed{;} \\ associate & = & \boxed{||}\ process \end{array}\right.$$

The components of a *cooperation* start simultaneously and continue together; a cooperation terminates when all the components of the cooperation have terminated.

Validity 8.5 *A cooperation is invalid if it contains references to free variables (elements other than contexts).*

Subordinate processes

$$
\begin{array}{lcl}
\mathit{subordination} & = & \langle_1 \mathit{subordinate} \rangle \boxed{;} \\
\mathit{subordinate} & = & \boxed{//}\ \mathit{process}
\end{array}
$$

The components of a *subordination* start simultaneously and continue independently; a subordination terminates when all the components of the subordination have started.

Validity 8.6 *A subordinate is invalid if*

1. *it contains references to free variables (elements other than contexts), or*
2. *reference is made to a free context on the right side of an interaction.*

Otherwise all constant, procedure and function names free in the subordinate are valid, and remain valid in the scope of the subordinate, if they were valid at the point of subordination.

A subordinate process automatically terminates if it attempts to interact with a bag or singleton context whose *scope* has terminated or whose value has been *output*. An output to a stream context whose scope has similarly terminated will cause the subordinate to terminate. An input from a stream context will cause a subordinate to terminate if the stream has terminated and is "empty"[10].

Repetition

$$
\begin{array}{lcl}
\mathit{construction} & = & \mathit{repetition} \\
\mathit{repetition} & = & \boxed{\texttt{WHILE}}\ \mathit{boolean}\ \boxed{:}\ \mathit{process} \\
 & | & \boxed{\texttt{WHILE}}\ \mathit{boolean} \circ \mathit{process} \\
 & | & \boxed{\texttt{DO}}\ \mathit{process}\ \boxed{\texttt{UNTIL}}\ \mathit{boolean}
\end{array}
$$

A *repetition* is defined by

$$
\texttt{WHILE}\ bP \stackrel{def}{=} \left\{ \begin{array}{ll} \texttt{TEST} & b\{P\ \texttt{WHILE}\ bP\} \\ & (\sim b)\mathit{skip} \\ ; & \end{array} \right.
$$

and

$$
\texttt{DO}\ P\ \texttt{UNTIL}\ b \stackrel{def}{=} \left\{ \begin{array}{ll} \{ & P \\ & \texttt{WHILE}\ (\sim b)P \\ \} & \end{array} \right.
$$

[10] Treat this with some care. In streams which have multiple contributors data will almost certainly be lost where one of the contributors is itself a subordinate.

Replication

Sequential replication

$$
\begin{array}{lcl}
\textit{sequence} & = & \boxed{\{}\ \textit{replicator}\ \boxed{:}\ \textit{process}\ \boxed{\}} \\
 & | & \boxed{\{}\ \textit{replicator}\ \circ\ \textit{process}\ \boxed{\}} \\
\textit{test} & = & \boxed{\texttt{TEST}}\ \textit{replicator}\ \boxed{:}\ \textit{conditional}\ \boxed{;} \\
 & | & \boxed{\texttt{TEST}}\ \textit{replicator}\ \circ\ \textit{conditional}\ \boxed{;} \\
\textit{choice} & = & \boxed{\texttt{CHOICE}}\ \textit{replicator}\ \boxed{:}\ \textit{alternative}\ \boxed{;} \\
 & | & \boxed{\texttt{CHOICE}}\ \textit{replicator}\ \circ\ \textit{alternative}\ \boxed{;}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{replicator} & = & \textit{name}\ \boxed{\texttt{FOR}}\ \textit{count} \\
 & | & \textit{name}\ \boxed{\texttt{FOR}}\ \textit{count}\ \boxed{\texttt{FROM}}\ \textit{base} \\
 & | & \textit{name}\ \boxed{\texttt{FOR}}\ \textit{count}\ \boxed{\texttt{FROM}}\ \textit{base}\ \boxed{\texttt{BY}}\ \textit{step} \\
\textit{base} & = & \textit{expression} \\
\textit{count} & = & \textit{expression} \\
\textit{step} & = & \textit{expression}
\end{array}
$$

In the following, let n be a name, c be a count, b be a base, and s be a step.

A replicator

$$n \texttt{ FOR } c \texttt{ FROM } b \texttt{ BY } s$$

specifies a set of c names $n = (s * i + b)$ each with a distinct scope and value such that each value is the result of $s * i + b$, where i is the set of integer literals in the range $0..c-1$. The type of n is derived from the type of the expression $s * i + b$; i.e., it may be an integer (signed or unsigned) or float[11].

Equivalence 8.9 *Abbreviated replicators mean*

$$n \texttt{ FOR } c \texttt{ FROM } b \stackrel{def}{=} n \texttt{ FOR } c \texttt{ FROM } b \texttt{ BY } 1$$

$$n \texttt{ FOR } c \stackrel{def}{=} n \texttt{ FOR } c \texttt{ FROM } 0$$

Equivalence 8.10 *The meaning of a sequential replication is defined by*

$$\{i \texttt{ FOR } c : P(i)\} \stackrel{def}{=} \{P(0)\ P(1)\ \ldots\ P(c-1)\ \}$$

[11] See the definition of expressions for validity.

where i is a name specified by the replication. The process P is replicated c times. The scope of i for each replication is the associated replication of the process P.

$$\texttt{TEST}\ i\ \texttt{FOR}\ c\ : C(i); \stackrel{def}{=} \texttt{TEST}\ C(0)\ C(1)\ \ldots\ C(c-1);$$

where i is a name specified by the replication. The conditional C is replicated c times. The scope of i for each replication is the associated replication of the conditional C.

$$\texttt{CHOICE}\ i\ \texttt{FOR}\ c : A(i); \stackrel{def}{=} \texttt{CHOICE}\ A(0)\ A(1)\ \ldots\ A(c-1);$$

where i is a name specified by the replication. The alternative A is replicated c times. The scope of i for each replication is the associated replication of the alternative A.

Abbreviations of sequential replications

test	=	[TEST] *replicator* [:] *conditional*
		[ELSE] *process* [;]
	|	[TEST] *replicator* ∘ *conditional*
		[ELSE] *process* [;]
choice	=	[CHOICE] *replicator* [:] *alternative*
		default [;]
	|	[CHOICE] *replicator* ∘ *alternative*
		default [;]

Equivalence 8.11 *The following are valid abbreviations of test and choice replications.*

$$\texttt{TEST}\ i\ \texttt{FOR}\ c : C(i)\ \texttt{ELSE}\ P(); \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{TEST} \\ \qquad \texttt{TEST}\ i\ \texttt{FOR}\ c : C(i); \\ \qquad \texttt{ELSE}\ P(); \end{array} \right.$$

and

$$\texttt{CHOICE}\ i\ \texttt{FOR}\ c\ : A(i)\ \texttt{ELSE}\ P(); \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{CHOICE} \\ \qquad \texttt{CHOICE}\ i\ \texttt{FOR}\ c : A(i); \\ \qquad \texttt{ELSE}\ P(); \end{array} \right.$$

Concurrent replication

associate	=	[||] *replicator* [:] *process*
	|	[||] *replicator* ∘ *process*
subordinate	=	[//] *replicator* [:] *process*
	|	[//] *replicator* ∘ *process*

Equivalence 8.12 *The meaning of a concurrent replication is defined by*

$$||i \text{ FOR } c : P(i) \stackrel{def}{=} ||P(0)||P(1) \ldots ||P(c-1)$$

$$//i \text{ FOR } c : P(i) \stackrel{def}{=} //P(0)//P(1) \ldots //P(c-1)$$

where i is a name specified by the replication, the type of i is derived as before and the value is in the interval $0..c-1$. The process P is replicated c times. The scope of i for each replication is the associated replication of the process P.

A cooperation with no components terminates immediately.

$$||i \text{ FOR } 0 : P(i) \stackrel{def}{=} ||skip$$

A subordination with no components terminates immediately.

$$//i \text{ FOR } 0 : P(i) \stackrel{def}{=} //skip$$

Placement

$$\begin{array}{lcl} subordinate & = & \boxed{//}\ placementprocess \\ & | & \boxed{//}\ replicator\ \boxed{:}\ placementprocess \\ placement & = & \boxed{\text{ON}}\ node\boxed{:} \\ & | & \boxed{\text{ON}}\ node \circ \\ node & = & expression \end{array}$$

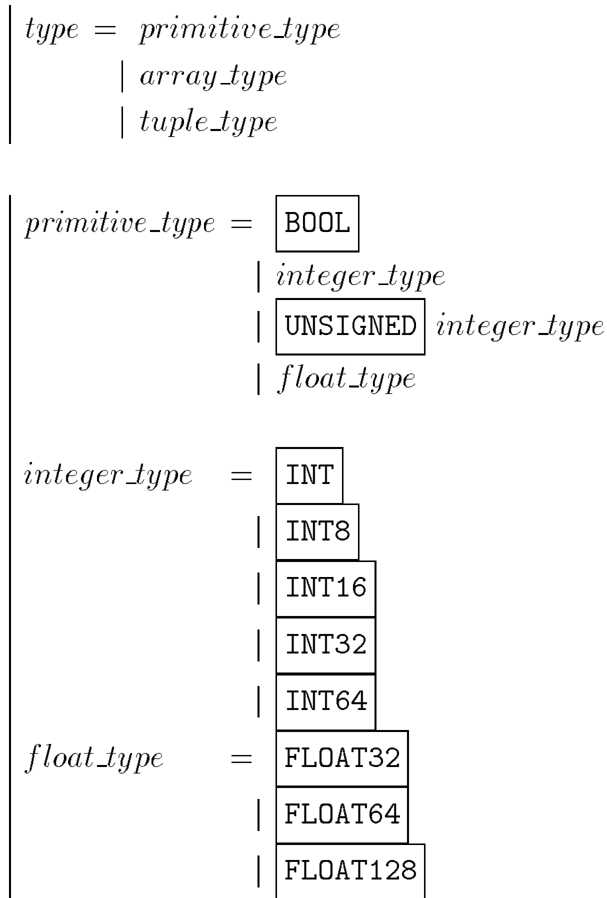
Placement provides directives to the implementation specifying the node on which a subordination should be placed. An implementation may choose to ignore these directives. The value and type of the node expression is implementation dependent.

On stop alternative sequence (error handling)

$$\begin{array}{lcl} sequence & = & \boxed{\{}\ \boxed{\text{ON}}\ \boxed{\text{STOP}}\ process \circ \\ & & \langle_0 \circ process\rangle\ \boxed{\}} \end{array}$$

An On Stop behaves like the first process if the sequence behaves like stop. Changes made to the environment by the sequence preceding the Stop are maintained.

8.3.3 Types



The interpretation of values is defined by a data type.

Boolean types

A value of type `BOOL` is either true or false.

Integer types

A value of integer type `INT` is a signed integer in the range

$$-2^{N-1}..(2^{N-1}-1)$$

where N is the number of bits natural to the particular implementation.

A value of integer type `INT`N is a signed integer in the range

$$-2^{N-1}..(2^{N-1}-1)$$

where N is the number of bits used to represent the type.

A value of unsigned integer type is an integer in the range

$$0..2^N$$

where N is the number of bits used to represent the type.

Float types

Floating point numbers use the ANSI/IEEE Standard 754–1985 representation.

A value of type `FLOAT32` is represented using a sign bit, an 8 bit exponent and a 23 bit fraction. A value is positive if the sign bit is 0 and negative if the sign bit is 1. Let m be the magnitude, f be the fraction and e represent the exponent, then

$$\begin{array}{ll} m = 2^{e-127} * 1.f & if\ 0 < e\ and\ e < 255 \\ m = 2^{-126} * 0.f & if\ e = 0\ and\ f <> 0 \\ m = 0 & if\ e = 0\ and\ f = 0 \end{array}$$

Similarly, a value of type `FLOAT64` is represented using a sign bit, an 11 bit exponent and a 52 bit fraction. A value is positive if the sign bit is 0 and negative if the sign bit is 1. Let m be the magnitude, f be the fraction and e represent the exponent, then

$$\begin{array}{ll} m = 2^{e-1023} * 1.f & if\ 0 < e\ and\ e < 2047 \\ m = 2^{-1022} * 0.f & if\ e = 0\ and\ f <> 0 \\ m = 0 & if\ e = 0\ and\ f = 0 \end{array}$$

Rounding and truncation

The rounding of floating point numbers occur in arithmetic expression evaluation, in explicit type conversions, and also when literals are converted to the IEEE representation.

All floating point issues defer to the ANSI/IEEE Standard 754-1985, representation and operations.

Array types

| *array_type* = `[` *expression* `]` *type*

An array type is a homogeneous ordered group of components of the same type. The size of the group is specified by the associate expression, which must be of integer type.

Validity 8.7 *Let e be an expression and t be a type, then the type $[e]t$ is valid iff $e > 0$.*

$$\begin{array}{|lcl} \textit{array_type} & = & \textit{integer_type}\ \boxed{\texttt{::}}\ \boxed{\texttt{[}}\ \textit{count}\ \boxed{\texttt{]}}\ \textit{type} \\ \textit{count} & = & \textit{expression} \end{array}$$

A counted array type is a *count* component of an integer type, followed by an array type. The size of the sequence is bounded to be not greater than the value specified by the associated count expression.

Validity 8.8 *Let e be an expression, I an integer type and t be some type, then the type $I :: [e]t$ is valid iff $e > 0$.*

Strings and characters

$$\begin{array}{|lcl} \textit{integer_type} & = & \boxed{\texttt{CHAR}} \\ \textit{array_type} & = & \boxed{\texttt{STRING}} \end{array}$$

Equivalence 8.13

$$\texttt{CHAR} \stackrel{def}{=} \texttt{UNSIGNED INT8}$$

$$\texttt{STRING} \stackrel{def}{=} \texttt{UNSIGNED INT8::[256]CHAR}$$

Tuple types

$$\begin{array}{|lcl} \textit{tuple_type} & = & \boxed{\texttt{(}}\ \langle_2 \boxed{\texttt{,}}\ \textit{type}\rangle \boxed{\texttt{)}} \end{array}$$

A tuple type is a heterogenous ordered group of components of some type.

Type deduction

$$\begin{array}{|lcl} \textit{type} & = & \boxed{\texttt{TYPE}}\ \boxed{\texttt{OF}}\ \textit{expression} \\ & | & \boxed{\texttt{TYPE}}\ \boxed{\texttt{OF}}\ \textit{context} \end{array}$$

The type `TYPE OF` *expression* is the type of the associated expression.

8.3.4 Element

An element enables the selection of components of arrays and tuples.

$$
\begin{array}{lcl}
element & = & element \; \texttt{[} \; subscript \; \texttt{]} \\
 & | & element \; \texttt{[} \; base \; \texttt{..)} \\
 & | & element \; \texttt{(..} \; limit \; \texttt{]} \\
 & | & element \; \texttt{[} \; base \; \texttt{FOR} \; count \; \texttt{]} \\
 & | & element \; \texttt{[} \; base \; \texttt{..} \; limit \; \texttt{]} \\
 & | & name \\
subscript & = & expression \\
base & = & expression \\
limit & = & expression \\
count & = & expression
\end{array}
$$

Let N be the name of an element, then the type of that element is the type specified in the *allocation* of N.

In the following, let s, b, l, and c be a subscript, base, limit, and count respectively, and let E be an array of type $[n]\tau$, where n is the size of the array and τ is the type of its components.

An element $E[s]$ selects the sth component of the array E and is of type τ.

An element $E[b \; \texttt{FOR} \; c]$, selects a *segment* (several contiguous components) of the array E, and is of type $[c]\tau$.

Equivalence 8.14

$$E(..l] \stackrel{def}{=} E[0..l]$$

$$E[b..) \stackrel{def}{=} E[b..n-1] \stackrel{def}{=} E[b \; \texttt{FOR} \; n-b+1]$$

Validity 8.9

1. *A subscript, base, limit, or count is valid iff the subscript, base, limit, or count expression is of integer type.*
2. *The element $E[s]$ is valid iff $0 \leq s < n$.*
3. *The element $E[b..l]$ is valid iff $0 \leq b \leq l < n$.*

4. The element $E[b$ `FOR` $c]$ is valid iff $0 \leq b$ and $1 \leq c \leq n$[12]*.*

element	=	*tuple*
tuple	=	`(` $\langle_2$ `,` *tuple_component*$\rangle$ `)`
tuple_component	=	*element* | `_`

A tuple is a heterogenous ordered group of elements.

A tuple component selects the field of a tuple.

In the following, let E and D be elements of type τ and θ respectively. Let T be a tuple.

The element (E, D) is a tuple of type (τ, θ).

The element $(E, _)$ is a tuple of type (τ, ω), where ω is an undefined type (i.e. compatible with any type).

An element $T[s]$ selects the sth component of the tuple, and is the type of the selected component.

An element $T[b$ `FOR` $c]$, selects a *segment* of the tuple T, and is a tuple with the type of the selected components.

element	=	*table*
table	=	`[` $\langle_2$ `,` *table_component*$\rangle$ `]`
table_component		| *element*

A table is a homogenous ordered group of elements of the same type.

A table component selects the field of a table.

In the following, let E^1 and E^2 be elements of type τ. Let T be a table.

The element $[E^1, E^2]$ is an array of type $[2]\tau$.

An element $T[s]$ selects the sth component of the table, and is of the type τ.

An element $T[b$ `FOR` $c]$, selects a *segment* of the table E, and is an array of type $[c]\tau$.

These principles are trivially extended for all tuples and tables with more than two more components.

An element has a value and a type. The value of an element may be changed by assignment or input. The initial value of an element is defined by its declaration.

Let e and e' be expressions, then:

[12]Note this definition does not allow elements with zero component arrays, counted arrays are provided for this purpose.

Let v be a variable of type t, then the assignment $v := e$ is valid iff e is of type t, the value of v is replaced by the value of e.

Let v be a variable of type $[n]t$, and s be a subscript of integer type, then the assignment $v[s] := e$ is valid iff e is of type t, the value of the component $v[s]$ is replaced by the value of e.

Let v be a variable of type $[n]t$, and $b..l$ be an interval, then the assignment $v[b..l] := e$ is valid iff e is of type $[n']t$, where $n' = l - b + 1$, the value of each component of $v[b..l]$ is replaced by the value of the respective component of e.

Let v be a variable of type $[n]t$, and $b\,\texttt{FOR}\,c$ be an interval, then the assignment $v[b\,\texttt{FOR}\,c] :=$ e is valid iff e is of type $[c]t$, the value of each component of $v[b\,\texttt{FOR}\,c]$ is replaced by the value of the respective component of e.

Let v be a variable of type $t :: [n]t'$, then the assignment $v := e$ is valid iff e is of type $t :: [n]t'$, the value of v is replaced by the value of e.

Let v be a variable of type $t :: [n]t'$, and $c :: e$ be an expression where c is an integer and e is an expression of type $[m]t'$, then the assignment $v := c :: e$ is valid iff $n \geq c \leq m$, the value of v is replaced by the value of $c :: e[0\,\texttt{FOR}\,c]$.

Let c be an integer variable, and v be a variable of type $[n]t'$, then the assignment $c :: v := e$ is valid iff e is of type $t :: [m]t'$, where $m \leq n$, the value of c is replaced by the size component of e, and then the value of $v[0\,\texttt{FOR}\,c]$ is replaced by the first c components of the associated array in e.

Let c and c' be integer variables, and v be a variable of type $[n]t$, then the assignment $c :: v := c' :: e$ is valid iff e is of type $[m]t$, where $m \leq n$, the value of c is replaced by the value of c', and then the value of $v[0\,\texttt{FOR}\,c]$ is replaced by $e[0\,\texttt{FOR}\,c']$.

Let v be a variable of type (t, t'), then the assignment $v := e$ is valid iff e is of type (t, t'), the value of each component in v is replaced by the value of the respective component of e.

Let (v, v') be a tuple of type (t, t'), and (e, e') be an expression of type (t, t'), then the assignment $(v, v') := (e, e')$, assigns the value of each component of (v, v') is replaced by the value of the respective component of (e, e'), such that $v := e$ and $v' := e'$.

Let $(v, _)$ be a tuple of type (t, u), where u is undefined, and (e, e') be an expression of type (t, t'), then the assignment $(v, _) := (e, e')$, is valid. The value of each component of $(v, _)$ is replaced by the value of the respective component of (e, e'), except where the component is undefined, such that $v := e$.

It follows that these principles hold for all tuples; a tuple can have two or more components, and zero or more of those components may be undefined.

8.3.5 Expressions

An expression has a value and a type. All expressions have a type which can be deduced by the compiler, although the type need not be explicitly stated where a type can be directly implied.

expression	=	*element*
		| *table*
table	=	`[` $\langle_1$ `,` *expression*$\rangle$ `]`

Let T be a table $[e^0, e^1, .., e^n]$, where each e denotes an expression of type t, then T is an array of type $[n]t$, where the value of each component is the value of the respective expression e. T is invalid iff any one of the components of T is of some type other than t.

Let e and e' be expressions of type t and t' respectively, then the value of (e, e') is a tuple of type (t, t'), where the value of each component is the value of e and e' respectively.

Let e be an expression of type t, then the value of $(e, _)$ is a tuple of type (t, t'), where t' is undefined, the value of each component is the value of e and an undefined value respectively.

Let $(e, _)$ be a tuple of type (t, u), where u is undefined, and (v, v') be an element of type (t, t'), then the assignment $(v, v') := (e, _)$, is valid. The value of each component of (v, v') is replaced by the value of the respective component of $(e, _)$, except where the component is undefined, such that $v := e$.

These principles hold for all tuples; a tuple can have two or more components, and zero or more of those components may be undefined.

expression	=	*monadic operand*
		| *operand dyadic operand*
operand	=	*expression*
		| `(` *expression* `)`

Both operands of a dyadic must be of the same type, the result is of the same type as the operands. All operands must have a defined value.

```
dyadic     = arithmetic
           | relation
           | logical
           | bitwise
arithmetic = + | - | ^ | * | / | %
relation   = = | > | < | >= | <= | <>
logical    = AND | OR | XOR
bitwise    = /\ | \/ | >< | >> | <<
```

The arithmetic operators are

+	addition
−	subtraction
^	power
*	multiplication
/	division
%	remainder

Arithmetic operators perform an arithmetic operation upon operands of the same integer or real data type. The result is a value of a type defined by the type of the operands.

The relation operators are

=	equality
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
<>	not equal

Relation operators perform a relational operation upon operands of the same data type (i.e. all types except contexts). The result is a value of boolean type. The relational operators are defined for non–scalar types. Non–scalar types are equal if all the components of the type are equal. A non–scalar type is greater or lesser than another according to the value of the first non–equal component. Non–scalar operands must be of the same type (which implies also the same length).

The logical operators are

`AND`	logical and
`OR`	logical or
`XOR`	logical exclusive or

Logical operators perform a logical operation upon operands of the boolean type. The result is a value of boolean type.

The bitwise operators are

`/\`	bitwise and
`\/`	bitwise or
`><`	bitwise exclusive or
`>>`	shift right
`<<`	shift left

Bitwise operators perform an operation on the bit pattern of a value of integer type. The result is a value of an integer type defined by the operands.

monadic	=	*minus*
	|	*complement*
	|	*negation*
minus	=	`−`
plus	=	`+`
complement	=	`~`
negation	=	`NOT`

The expression $-x$ has the value $0 - x$.

The expression $+x$ has the value $0 + x$.

The expression $\sim x$ is the complement of x; x must be an integer and the result is a value of the same integer type.

The expression `NOT`b is the negation of the boolean b. The result is boolean.

Expression evaluation

Expressions are evaluated left to right, dependent on operator priority. The priority is defined in the accompanying grammar.

Literals

$$expression = literal$$

$$
\begin{aligned}
literal = \; & char \\
| \; & string \\
| \; & integer \\
| \; & float \\
| \; & \boxed{\texttt{TRUE}} \; | \; \boxed{\texttt{FALSE}}
\end{aligned}
$$

A literal is an expression (it has a value and type).

$$
\begin{aligned}
char &= \boxed{\texttt{'}} \; character \boxed{\texttt{'}} \\
string &= \boxed{\texttt{"}} \; \langle_0 character\rangle \boxed{\texttt{"}}
\end{aligned}
$$

A *char* is an expression of the predefined type `CHAR`, a character is a member of the printable ASCII defined set or a special character.

A *string* is an expression of the predefined type `STRING`.

$$
\begin{aligned}
integer \;&=\; \langle_1 digit\rangle \\
&\;|\; \boxed{\texttt{\#}} \; \langle_1 digit\rangle \\
&\;|\; base \boxed{\texttt{\#}} \; \langle_1 digit\rangle \\
float \;&=\; \langle_1 digit\rangle \boxed{\texttt{.}} \; \langle_1 digit\rangle \\
&\;|\; \langle_1 digit\rangle \boxed{\texttt{.}} \; \langle_1 digit\rangle \boxed{\texttt{E}} \; exponent \\
exponent \;&=\; \boxed{\texttt{+}} \; \langle_1 digit\rangle \\
&\;|\; \boxed{\texttt{-}} \; \langle_1 digit\rangle
\end{aligned}
$$

The literals `TRUE` and `FALSE` represent the boolean values *true* and *false* respectively.

The type of an integer literal is `INT`, unless the type of an associated operand or element constrains it to a value of another integer type.

The type of a float literal is `FLOAT32`, unless the type of an associated operand or element constrains it to a value of another float type.

Let l be an integer literal, e an expression of integer type I, and Δ a valid dyadic, then in the expression $e\Delta l$ the literal l is constrained to be of type I.

Let l be an integer literal, and v a variable of integer type I, then in the assignment $v := l$ the literal l is constrained to be of type I, excepting if the assignment is in a declaration then l is the default type `INT`.

Let l be a float literal, e an expression of float type F, and Δ a valid dyadic, then in the expression $e\Delta l$ the literal l is constrained to be of type F.

Let l be a float literal, v a variable of float type F, then in the assignment $v := l$ the literal l is constrained to be of type F, excepting if the assignment is in a declaration then l is the default type `FLOAT32`.

Let l and l' be integer literals, and Δ a valid dyadic, then the expression $l\Delta l'$ is also an integer literal.

Let l and l' be float literals, and Δ a valid dyadic, then the expression $l\Delta l'$ is also a float literal.

Let l be an integer literal, and ϕ a valid monadic, then the expression ϕl is also an integer literal.

Let l be a float literal, and ϕ a valid monadic, then the expression ϕl is also a float literal.

Conditional expressions

| *expression* = `IF` *boolean* `:` *expression* `ELSE` *expression*

Let b be a boolean, let e and e' be expressions then the conditional expression

$$\texttt{IF}\ b\ \texttt{:}\ e\ \texttt{ELSE}\ e'$$

is the value of e if the value of b is true, is the value of e' if the value of b is false.

Merge expressions

| *expression* = *string* `(` $\langle_1$ `,` *expression*$\rangle$ `)`

Let S be a string. A string of the form $S(e^0, ...e^n)$, is a *merge* expression where each instance of the special character % in S is replaced by a string which represents the respective value of e, if e is a number, or the string if e is a string.

The format of numbers merged with a string in this way can be specified by the special character sequences $\%fd$ or $\%fd.d$ or $\%fd.d\mathtt{E}d$, where d is an integer value representing the number of components to add to the string, and f is either `r`, `l` or `c`, and indicates, right, left or centre alignment respectively.

String concatenation

$$
\begin{array}{lcl}
\mathit{expression} & = & \mathit{concatenation} \\
\mathit{concatenation} & = & \langle_1 \boxed{\backslash}\ \mathit{string} \rangle
\end{array}
$$

The expression $s\backslash t$ concatenates the strings s and t; i.e.,

"hello"\" world" = "hello world"

and

"hello"\
" world" = "hello world"

8.3.6 Type constraint, assertion and casting

A type constraint constrains the type of an expression to a specified type. If the value is not that specified then the value is coerced to a value of the type.

$$
\begin{array}{lcl}
\mathit{expression} & = & \mathit{constraint} \\
\mathit{constraint} & = & \mathit{expression}\ \boxed{\texttt{->}}\ \mathit{type} \\
 & | & \mathit{expression}\ \boxed{\texttt{->}}\ \boxed{\texttt{ROUND}}\ \mathit{type} \\
 & | & \mathit{expression}\ \boxed{\texttt{->}}\ \boxed{\texttt{TRUNC}}\ \mathit{type}
\end{array}
$$

Let e be an expression and t be a type, then a constraint $e -> t$ constrains the value of the expression e to be the type t. The value of e is converted to a value of the compliant C.

Conversions from integer to floating point values (and vice versa) can specify whether the result of the conversion is to be rounded or truncated. By default such conversions are rounded; i.e., the value is rounded to the nearest value of the specified type. Where two values are equally near the value is rounded to the nearest even number. A truncated value is rounded toward zero.

A boolean value can only be constrained to a boolean type.

Tuples and arrays can only be constrained to tuples and arrays with an equal number of components.

$$
\begin{array}{lcl}
element & = & element \ \boxed{\texttt{=>}} \ compliant \\
expression & = & expression \ \boxed{\texttt{=>}} \ compliant \\
compliant & = & type \\
 & | & \boxed{\texttt{[]}} \ type
\end{array}
$$

A compliance e => C is defined by

$$e \texttt{=>} C \stackrel{def}{=} e$$

and is valid iff

$$\texttt{TYPEOF}\ e = C$$

.

A compliance e => $[]C$ is defined by

$$e \texttt{=>} []C \stackrel{def}{=} e$$

and is valid iff e is an array with components of type C.

A compliance (type assertion) causes the associated action to stop if the compliance is not compatible.

$$
\begin{array}{lcl}
expression & = & cast \\
cast & = & expression \ \boxed{\texttt{>|}} \ type
\end{array}
$$

A cast forces a value to be the type specified but makes no alteration to the bit pattern use to represent the value of the expression's defined type.

The bit size of the cast type must be equal to the bit size of the original type. In particular tuples and arrays can be cast provided this constraint is met.

8.3.7 Keywords and names

Keywords are distinct and may not be used as names. Keywords are not case sensitive.

Names may be any length, include any of the following characters but must begin with a *letter*.

$$
\begin{array}{lcl}
letter & = & a..z \mid A..Z \\
digit & = & 0..9 \\
symbol & = & _ \mid '
\end{array}
$$

Names are case sensitive.

8.3.8 Scope

Names in *Ease* may strictly have only one meaning in any particular scope. Therefore, subsequent specification of a name already specified gives that name a new meaning for the scope of the specification, and *hides* the old meaning of the name for the duration of its scope.

$$
\begin{array}{lcl}
process & = & specification_block \; \boxed{:} \; scope \\
scope & = & process \\
 & | & conditional \\
 & | & option \\
 & | & alternative \\
 & & \\
specification_block & = & \langle_1 specification \rangle \\
specification & = & definition \\
 & | & declaration
\end{array}
$$

Associated with each name specified is a region of the program in which the name is valid, called the *scope* of the name. The scope of a name is the *specification block* in which it is specified and the action, construction, conditional, option or alternative which immediately follows the *specification block*.

A name may not be specified twice in the same *specification block*.

8.3.9 Declaration

$$declaration = \boxed{\texttt{LET}} \; \langle_1 \; specifier \rangle$$

Constants

$$
\begin{array}{lcl}
specifier & = & name \; \boxed{=} \; expression \\
constant & = & name
\end{array}
$$

A constant specifier declares a name whose value and type is that of the associated expression.

Validity 8.10

- *A name specified in a constant declaration may only appear in expressions.*
- *The specifier is invalid if an element that appears in the expression is assigned to in the scope of the name.*

Renaming

$$
\begin{array}{lcl}
\textit{specifier} & = & \textit{name}\ \boxed{\texttt{RENAME}}\ \textit{element} \\
 & | & \textit{name}\ \boxed{\texttt{RENAME}}\ \textit{context}
\end{array}
$$

An renaming specifier declares a name which *renames* the associated element or context. An element or context may be renamed only once in a specification block.

Variables and contexts

$$
\begin{array}{lcl}
\textit{allocation} & = & \textit{name}\ \boxed{\texttt{:=}}\ \textit{expression} \\
 & | & \textit{name}\ \boxed{\texttt{:=}}\ \textit{context_type_name} \\
 & | & \textit{name}\ \boxed{\texttt{:=}}\ \textit{context_type_name}\ \boxed{\texttt{ON}}\ \textit{node} \\
 & | & \textit{name}\ \boxed{\texttt{:=}}\ \textit{context_type_name}\ \boxed{\texttt{ON}}\ \textit{node}\ \boxed{\texttt{AT}}\ \textit{address} \\
\textit{specifier} & = & \textit{allocation} \\
\textit{address} & = & \textit{expression}
\end{array}
$$

An allocation specifies a name which is *assigned* the value and type of the associated expression, or specifies a name which is *assigned* the type of the associated context type name and the value *empty*[13].

A context may be specified to reside on a particular node in a machine, and optionally at a specific address. The value of the node and address expressions are implementation dependent.

Abbreviation of variable and context declarations

$$
\begin{array}{lcl}
\textit{allocation} & = & \langle_1 \boxed{,}\ \textit{name}\rangle\ \boxed{\texttt{:=}}\ \textit{type} \\
 & | & \langle_1 \boxed{,}\ \textit{name}\rangle\ \boxed{\texttt{:=}}\ \textit{expression}
\end{array}
$$

$$
\texttt{LET}\ v^0,\ ..\ ,\ v^n\ :=\ \tau\ :\ \stackrel{def}{=}\ \texttt{LET}\ v^0 := \tau\ ..\ v^n := \tau\ :
$$

where τ is a type or expression.

[13] An empty context.

Enumeration

$$\begin{array}{lcl} \textit{enumeration} & = & \boxed{\texttt{ENUM}} \langle_1 \boxed{,}\ \textit{name} \rangle \\ & | & \boxed{\texttt{ENUM}}\ \boxed{\texttt{FROM}}\ \textit{base} \langle_1 \boxed{,}\ \textit{name} \rangle \\ \textit{base} & = & \textit{expression} \\ & & \\ \textit{declaration} & = & \textit{enumeration} \end{array}$$

An enumeration specifies an enumerated set of constants, and is defined by

$$\texttt{ENUM}\ c^0 \mathrel{..} c^n \quad \stackrel{def}{=} \quad \texttt{LET}\ c^0 = 0 \mathrel{..} c^n = n - 1$$

and

$$\texttt{ENUM FROM}\ s\ c^0 \mathrel{..} c^n \quad \stackrel{def}{=} \quad \texttt{LET}\ c^0 = s \mathrel{..} c^n = s + n - 1$$

where the type of c^n is the type of s.

8.3.10 Definitions

Type definition

$$\begin{array}{lcl} \textit{definition} & = & \boxed{\texttt{TYPE}}\ \textit{type_name}\ \boxed{\texttt{IS}}\ \textit{type} \\ \textit{type_name} & = & \textit{name} \\ \textit{type} & = & \textit{type_name} \end{array}$$

A type definition defines a name for the specified type. A variable, context, or expression whose type is defined by a type definition are of the same type if their type has been defined in the same definition.

All expression operators remain defined for values of the named type as for values of the base type.

Context

```
definition = TYPE type_name CONTEXT ⟨1 | bag⟩
bag        = bag_type
           | priority bag_type
bag_type   = type
           | singleton
           | stream
           | type REPLY type
singleton  = SINGLE type
           | [ expression ] singleton
           | [ expression ] stream
stream     = STREAM type

priority   = LO
           | HI
           | LO expression
           | HI expression
```

A context definition specifies a name for a *context* type.

The *type* components of a context must be distinct types.

Outputs to a context stream are ordered such that inputs are satisfied by one of the least recently output values[14].

An output to a context singleton adds the value to the context, or replaces the previous value of the singleton if a value is existent.

An output *context*[*i*]!*e* places the value of the expression *e* in the context at subscript *i*, or replaces the previous value of the singleton if a value is existent.

A *priority* `HI` is a higher priority than a *priority* `LO`. An expression associated with a priority is some signed integer value[15]. The greater value signifies a higher priority. Interactions on components with higher priority will be satisfied in preference to those interactions on components with lower priority. Those *bag types* not explicitly given a priority are priority `LO` by default.

[14] If a single process is performing output to a context stream inputs will be satisfied in strict order.
[15] An implementation may choose to reserve positive values for subsystem use.

Procedure definition

Procedures are defined by textual substitution of the process they name[16]. Typing of procedure parameters is polymorphic or type assertive.

$$
\begin{array}{lcl}
\textit{definition} & = & \boxed{\texttt{PROCEDURE}}\ \textit{proc_name}\ \boxed{(}\ \langle_0 \boxed{,}\ \textit{pformal}\ \rangle\ \boxed{)} \\
 & & \textit{body} \\
 & & \\
\textit{body} & = & \textit{process} \\
\textit{pformal} & = & \textit{name} \\
 & | & \textit{compliant name} \\
 & | & \boxed{\texttt{VAL}}\ \textit{name} \\
 & | & \boxed{\texttt{VAL}}\ \textit{compliant name}
\end{array}
$$

A procedure *definition* names a process.

$$
\begin{array}{lcl}
\textit{instance} & = & \textit{proc_name}\boxed{(}\ \langle_0 \boxed{,}\ \textit{actual}\ \rangle \boxed{)} \\
\textit{actual} & = & \textit{element} \\
 & | & \textit{expression} \\
 & | & \textit{context} \\
\textit{process} & = & \textit{instance}
\end{array}
$$

Let B be the body of a procedure then given the definition

$$
\begin{array}{l}
\texttt{PROCEDURE}\ P() \\
B
\end{array}
$$

then an instance is exactly the substitution of the body; i.e.,

$$P() \stackrel{def}{=} B.$$

Let n be a name which appears in the body B, and let x be an actual, then given the definition

$$
\begin{array}{l}
\texttt{PROCEDURE}\ P(n) \\
B
\end{array}
$$

an instance is a substitution of the body with the name specified as a formal renamed; i.e.,

$$P(x) \stackrel{def}{=} \texttt{LET}\ n\ \texttt{RENAME}\ x : B.$$

[16] This definition represents a change to the definition of the July 1990 Yale Report. This version is has a better algebra for recursion and utilizes the introduction of RENAME

Alternatively a formal may be specified to be the value of the actual; i.e., given the definition

$$\begin{array}{l}\texttt{PROCEDURE}\ P(\texttt{VAL}\ n)\\ B\end{array}$$

and instance takes the value of the actual; i.e.,

$$P(x) \stackrel{def}{=} \texttt{LET}\ n = x : B$$

A compliant restricts the type of an actual to that specified by the compliant. Let C be a type compliant, then given the definition

$$\begin{array}{l}\texttt{PROCEDURE}\ P(C\ n)\\ B\end{array}$$

an instance of P is valid only if the actual is of the type specified by the compliant; i.e.,

$$P(x) \stackrel{def}{=} \texttt{LET}\ n\ \texttt{RENAME}\ a\ \ \texttt{=>}\ C:\ B.$$

Similarly for actuals specified to be values; i.e., given the definition

$$\begin{array}{l}\texttt{PROCEDURE}\ P(\texttt{VAL}\ C\ n)\\ B\end{array}$$

a similar type assertion compliance is applied

$$P(x) \stackrel{def}{=} \texttt{LET}\ n = a\ \ \texttt{=>}\ C:\ B.$$

The full range of compliant assertion can be use. Thus let $[]C$ be a type compliant and let x^e be an array type actual with e components, then given the definition

$$\begin{array}{l}\texttt{PROCEDURE}\ P([]C\ n)\\ B\end{array}$$

an instance will accept an array with any number of components of type C; i.e.,

$$P(x^e) \stackrel{def}{=} \texttt{LET}\ n\ \texttt{RENAME}\ A^e\ \ \texttt{=>}\ []C:\ B$$

Recursive sequences

A recursive reference must be the last process in the sequential construction which forms the procedure. A new instance of the procedure is instantiated. This form of recursion is commonly called "tail recursion"; i.e., given the recursive procedure definition

$$\begin{array}{l}\texttt{PROCEDURE}P()\\ \quad\{\ B\\ \quad\quad P()\ \ \}\end{array}$$

an instance of P is

$$P() \stackrel{def}{=} \{\ B \\ P()\ \}$$

and so on. Similarly

$$\texttt{PROCEDURE}P(n) \\ \{\ B \\ P(a)\ \}$$

is

$$P(x) \stackrel{def}{=} \{\ B \\ P(a)\ \}$$

A recursive sequence is a terminal extension of a procedure.

Recursion in parallel

The terminating characteristic of recursive sequences is *hidden* by subordination. This permits the recursive instance to form a self–replication of the procedure in parallel to the first instance; i.e., a procedure

$$\texttt{PROCEDURE}P() \\ \{\ B \\ //P(); \\ B'\ \}$$

is

$$P() \stackrel{def}{=} \{\ B \\ //P(); \\ B'\ \}$$

and so on.

A recursive reference may not occur in a cooperation.

Function definition

Functions are side effect free and polymorphic.

```
definition    = FUNCTION fn_name ( ⟨0 , fformal ⟩ )
                fexpression

fformal       = name
              | compliant name
fexpression   = = expression
              | = expression WHERE process
λ             = VAL ⟨0 , fformal ⟩ : ( fexpression )
λ_expression  = specification_block ( fexpression )
expression    = fn_name ( ⟨0 , expression ⟩ )
              | lambda_expression
```

The type and value of a *function expression*

$$= e$$

is the type and value of the expression e, and is valid iff the names in e are constants.

The type and value of a *function expression*,

$$= e \texttt{ WHERE } P()$$

is the type and value of the expression e after the associated process $P()$ has been performed, and is valid iff the names in e are either constants or names specified in the initial *specification block* of $P()$. $P()$ may not contain references to free variables or references to a free context.

Let $\mathcal{E}$ be a function expression. The *lambda definition*

$$\texttt{VAL}\ n\ :(\mathcal{E})$$

defines a function expression where n defines a constant name which appears free in $\mathcal{E}$ the value and type of which is determined in an instance of the expression (called a *lambda expression*) defined by the definition.

Let $\mathcal{E}$ be a function expression, and e be an expression, then the value of the lambda expression,

$$\texttt{LET}\ n = e : (\mathcal{E})$$

is the value of $\mathcal{E}$ where each instance of the name n in $\mathcal{E}$ is substituted for the value of the expression e. The substitution of e's value is valid iff the type `TYPE OF` e is compatible with each instance of n in $\mathcal{E}$.

Let $\mathcal{E}$ be a function expression. The lambda definition

$$\texttt{VAL}\ Cn\ :(\mathcal{E})$$

defines a function expression where n is a name which appears in $\mathcal{E}$. A lambda expression

$$\texttt{LET}\ n = e \texttt{=>} C : (\mathcal{E})$$

is a valid instant of the definition. The type of e is asserted[17] that of the compliant used in the definition.

A lambda definition is given a name in a function definition. The instance of a function is defined by substitution of the corresponding lambda expression.

Let $\mathcal{E}$ be a function expression, v be a result type compatible variable, e, x and y be expressions, and C a compliant, then given the function definition

$$\texttt{FUNCTION}\ f()\mathcal{E}$$

an instance of f is

$$v := f() \stackrel{def}{=} v := (\mathcal{E})$$

A function definition names a lambda definition; i.e.,

$$\texttt{FUNCTION}\ f(n)\mathcal{E} \stackrel{def}{=} \texttt{VAL}\ n\ : (\mathcal{E}).$$

An instance of this function is

$$v := f(e) \stackrel{def}{=} v := \texttt{LET}\ n = e : (\mathcal{E})$$

and so on for functions with multiple arguments; i.e.,

$$\texttt{FUNCTION}\ f(a,b)\mathcal{E} \stackrel{def}{=} \texttt{VAL}\ a\ \texttt{VAL}\ b\ : (\mathcal{E})$$

then

$$v := f(x,y) \stackrel{def}{=} v := \texttt{LET}\ a = x\texttt{LET}\ b = y : (\mathcal{E})$$

and functions whose arguments are compliant, given

$$\texttt{FUNCTION}\ f(Cn)\ \mathcal{E} \stackrel{def}{=} \texttt{VAL}\ Cn\ : (\mathcal{E})$$

then

$$v := f(e) \stackrel{def}{=} v := \texttt{LET}\ n = e \texttt{=>} C : (\mathcal{E})$$

given

$$\texttt{FUNCTION}\ f(Ca, Cb)\ \mathcal{E} \stackrel{def}{=} \texttt{VAL}\ Ca\ \texttt{VAL}\ Cb\ : (\mathcal{E})$$

then

$$v := f(x,y) \stackrel{def}{=} v := \texttt{LET}\ a = x \texttt{=>} C\ \texttt{LET}\ b = y \texttt{=>} C : (\mathcal{E}).$$

[17]Not coerced; i.e., it is an error if e is not of type C.

Recursive function expressions

$$
\begin{array}{lcl}
\textit{fexpression} & = & \boxed{\texttt{=}}\ \textit{recursive} \\
 & | & \boxed{\texttt{=}}\ \textit{recursive}\ \boxed{\texttt{WHERE}}\ \textit{process} \\
\textit{recursive} & = & \boxed{\texttt{IF}}\ \textit{boolean expression}\ \boxed{\texttt{ELSE}}\ \textit{tail} \\
 & | & \boxed{\texttt{IF}}\ \textit{boolean tail}\ \boxed{\texttt{ELSE}}\ \textit{expression} \\
 & | & \boxed{\texttt{IF}}\ \textit{boolean tail}\ \boxed{\texttt{ELSE}}\ \textit{tail} \\
\textit{tail} & = & \textit{recursive} \\
 & | & \boxed{\texttt{RECURSE}}\ \boxed{\texttt{(}}\ \langle_1 \boxed{\texttt{,}}\ \textit{fformal}\ \rangle\ \boxed{\texttt{)}}
\end{array}
$$

Recursive function expressions allow the construction of tail recursive functions, such that an instance of

$$\texttt{FUNCTION}\ f(x) = \texttt{IF}\ b\ e\ \texttt{ELSE}\ \texttt{RECURSE}(x+1)$$

is e if b is true, $f(x+1)$ otherwise. Similarly an instance of

$$\begin{array}{l}\texttt{FUNCTION}\ f(x) = \texttt{IF}\ b\ e\ \texttt{ELSE}\ \texttt{RECURSE}(x+1) \\ \quad \texttt{WHERE}\ \ P\end{array}$$

is e if b is true after P, $f(x+1)$ otherwise.

8.3.11 Other recursive definitions

Recursive definitions (e.g. mutual recursion) other than those specified for procedure and lambda are invalid — held for later review.

8.3.12 Comments

$$\textit{comment} = \boxed{\texttt{/*}}\ \textit{text}\ \boxed{\texttt{*/}}$$

The text between two comment symbols is ignored by the compiler. Nested comments should be permitted by a compiler.

8.3.13 Modules

definition	=	`MODULE` *name mbody* `END MODULE`
mbody	=	*show hide*
	|	*hide show*
	|	*show*
show	=	`SHOW` *specification_block*
hide	=	`HIDE` *specification_block*

specification	=	*module_instance*
module_instance	=	`USE` *name*

A module is defined by substitution of it's body.

When instanced, all the names specified in the *hide* portion of the module definition are given a canonical form; i.e., they are given distinct unknown names, names which do not exist in the current or subsequent scope.

The *mbody* of a module cannot contain references to free names.

Chapter 9

CSP

9.1 Communicating processes

My review of generalized communication as a programming model forces me to draw a remarkable conclusion given today's art in programming high performance machines; i.e., communication is an unnecessary distinction, simply too mechanistic for practical utilization, and too little facility for expression in the large.

It is assignment, and not communication, that is the fundamental notion. This in itself is not new, theory has often seen it this way (e.g., assignment as a single indivisible event is common in both CCS [Mil89] and CSP [Hoa85]) even if programmers haven't. It expresses the changing environment, and this changing environment characterizes the progress of an algorithmic solution (i.e., a computer program).

Communication is simply a mechanism of assignment; i.e., a variable receives a value output by an expression, but to make this distinction is to revel in the mechanism. An assignment is simply a single indivisible event — a respecification of a name's value.

Let us consider communication then in the light of these remarks.

We must call to mind that in providing effective engineering tools for parallel programming we must be concerned with the classic problem of how to maintain consistency and avoid conflict between an assignment to a variable and it's intended usage.

We must first agree that assignment to local variables occurs only in strict and specified sequence, as it has done these past years in conventional sequential programming. A communication then between concurrent processes (as is defined in CSP) is simply the assignment of a value yielded in one process to a variable existent in another. A single event engaged by two processes simultaneously.

From this we must observe that point–to–point communication imposes two particular constraints on these "non–local" assignments. Communications are both hidden and synchro-

nized; i.e.,

- since the variable to which the assignment is being made is private (local) to the receiving process the event is "visible" only to processes subsequent to it; i.e., hidden from concurrent processes,
- the event requires the synchronization of the two processes involved in the interaction.

This is the very basis of the point–to–point communication mechanism. However, while this is an essential mechanism in the mathematics of process behavior these are undesirable constraints on the programming model for two primary reasons.

1. the hidden nature of a communication means that to share information between several processes many such events must be specified by the programmer allowing information to propagate,
2. the forced wait of outputting processes creates greater opportunities for deadlock since it forbids any subsequent interaction by the process which may release a deadlock state[1].

It might at first seem that the obvious solution to our problem is to remove the hidden nature of variables that would be assigned to by communication. Making their value visible to all processes. Yet we must be careful since this could simply bring us full circle and present us with the greater set of problems communication was conceptually deemed to overcome in the first place.

We have seen that communication as a concept is simply a particular form of assignment. It is a solution that arose in response to problems in concurrent programming and mirrors mechanisms found in hardware. It was a natural first step from an engineering point of view and remains important from a theoretical one. However, it has proven deficient by the reasoning outlined here and in the earlier sections.

What can we learn from our experience with communication that will point us in the right direction?

Assignment to local variables causes us no concern since we live in the certain knowledge that other assignment to the variable either precedes or is subsequent. Let us consider the nature of such a variable for a moment.

We may regard each variable as itself being a process whose behavior is defined by the sequence of assignments to it. If we exchange each assignment for the value it assigns we can,

[1] We must be prepared here to accept a wait by inputting processes since the programmer has designated this instant for the respecification of the associated variable – to continue beyond this instant will leave the environment in a partially specified state and render it's composition with subsequent events meaningless. Lazy evaluation is an issue which belongs in another paradigm or as an implementation detail; I ignore it here.

in turn, consider a variable to be a set of values, totally ordered by the originating sequence. An environment consisting of many local variables can therefore be seen as a family of sets. Each set "interleaved" with the others according to the specified sequence and identified by the variable name; i.e.,

$$\begin{array}{l} x = \{\{1_0\},\{1_0,3_2\},\{1_0,3_2,5_4\}\}, \\ y = \{\{2_1\},\{2_1,3_3\},\{2_1,3_3,3_5\}\} \end{array} \quad . \left\{ \begin{array}{l} x := 1_0 \\ y := 2_1 \\ x := 3_2 \\ y := 3_3 \\ x := 5_4 \\ y := 3_5. \end{array} \right.$$

where the left side of the equation defines the evolved environment specified by the sequence of assignments on the right (each distinquished by the subscript value). Any of the assignments in this example could be replaced by an appropriate point–to–point communication which is an input to the variable.

What is characteristic of this view remains the single indivisible nature of an assignment, its clear association with the value it assigns to the variable, and the notion of interleaving.

9.2 Shared data structures

Let us now, as we did for communication, try to characterize operations on shared data structures. In the following discussion I shall generalize, much as I did when discussing implementation.

A Write is an assignment of a value to an intermediary. This intermediary is, in fact, a set of values identified by an event; each event is distinct though its value may not be. A Get is defined as a deletion of a value from the set; thus its behavior is to synchronize on the presence of a value (i.e., it cannot proceed if the set is empty) and, involves the assignment of a value to a local variable.

Indeed, we can choose to view this intermediary as a process, a process that is eager to satisfy inputs to it, and satisfies its outputs only when data is available to do so, thus we have a mechanistic decomposition of the behavior of such an intermediary in a manner consistent with our earlier deliberation.

Again we find it is the single indivisible assignment that is the fundamental notion. We can choose to regard this intermediary, as we did local variables in the previous section, as a process whose behavior is defined by the operations upon it. In our earlier example we say the sequence in which assignments occurred provides a total order on the values, however here the operations may occur in concurrent processes. What may we say about their order?

Let us take two concurrent processes, one that writes six values and one of that consumes

six values.

$$\begin{array}{ll} write(1_0) \parallel & get(x) \\ write(2_1) & get(y) \\ write(3_2) & get(x) \\ write(3_3) & get(y) \\ write(5_4) & get(x) \\ write(3_5) & get(y) \end{array}$$

such that, the environment is described

$$x = \{\{a\}, \{a, b\}, \{a, b, c\}\},$$
$$y = \{\{d\}, \{d, e\}, \{d, e, f\}\} \ .\{a, b, c, d, e, f\} = \{1_0, 2_1, 3_2, 3_3, 3_4, 5_5\}.$$

In this equation the set $\{1_0, 2_1, 3_2, 3_3, 3_4, 5_5\}$ represents the values held by the intermediary, variables are represented as before.

The operations have a characteristic we have seen before. Each event with the intermediary is single and indivisible. The behavior of the intermediary process is thus defined by an interleaving of the operations upon it. However, this cannot be an arbitrary interleaving, for we have the behavior of the intermediary to take into account.

We shall undertake a systematic interleaving of the operations such that they observe the following relation. *By definition* we shall permit no Get operation to *precede* a concurrent Write operation, and no Write operation to *precede* a Get to which it is subsequent (in a sequence).

The previous paragraph defines a relation in the interleaving between the operations on the intermediary that is a partial order on the whole set of operations acting upon it (figure 9.1).

Thus, the interleaving orders the operations on an intermediary such that Get operations are considered to return a value assigned to the intermediary by one of the preceding Write operations; i.e., we can systematically "walk" along the interleaved form associating Write/Get pairs where the Write defines the value returned by the Get operation.

Any single execution of such a program will, in effect, manifest a total order, which is but one ordering of the possible interleaving the partial order permits[2].

We can describe this interleaving further by extending our partial order relation. First by observing that two Gets in the same sequence maintain their order with respect to each other in an interleaving.

The ordering of Write operations in the interleaving is a little more difficult to observe. Writes may not preserve their order with respect to each other in a sequence since they *by definition* terminate immediately and are thus unordered when associated with the intermediary (as above).

[2]Those familiar with CSP will recognize this interleaved semantic form as CSP *traces*.

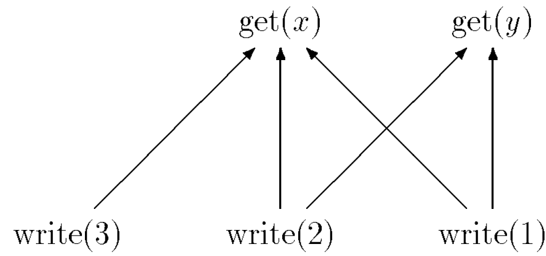


Figure 9.1: Partial order diagram for the interacting sequences:

$$\text{write}(1) \| \begin{array}{l}\text{write}(2)\\ \text{get}(x)\end{array} \| \begin{array}{l}\text{get}(y)\\ \text{write}(3)\end{array}$$

The relation is

- *No Get may precede a concurrent Write, and*
- *no Write may precede a Get to which it is subsequent.*

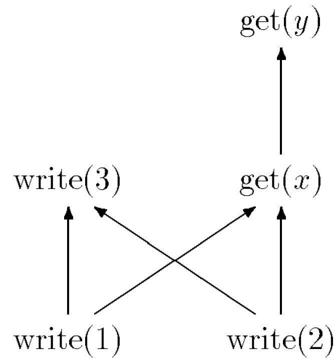


Figure 9.2: Extended partial order diagram to illustrate the interleaving of:

write(1)|| write(2)
get(x)
get(y)
write(3)

The extended relation is

- No Get may precede a concurrent Write,
- no Write may precede a Get to which it is subsequent (in a sequence),
- *Gets preserve their respective order in sequence, and*
- *an initial Write precedes subsequent Writes only if the number of Gets at the start of each sequence is equal to or greater than the number of Writes in the initial set.*

However, a Write in the initial set of Writes (i.e., those first in the sequences that act upon the intermediary) precedes Writes not in the initial set if the number of Get operations *at the start of each sequence* is equal to or greater than the number of Writes in the initial set. This much should be clear since such will require the association of all the Writes in the initial set with Get operations before the subsequent Write regardless of which Get is satisfied. (illustrated in figures 9.2 and 9.3).

In figure 9.3 it is worth noting that a further ordering between Writes is guaranteed by the interleaving in this case; i.e., each Write precedes those subsequent to it *in the same sequence*. However a general relation to express this order is complex, and dependent on the undertaking of all possible interleavings since we must establish that such a Write *must* precede a Get operation which is subsequent to it in the sequence, whatever the result of

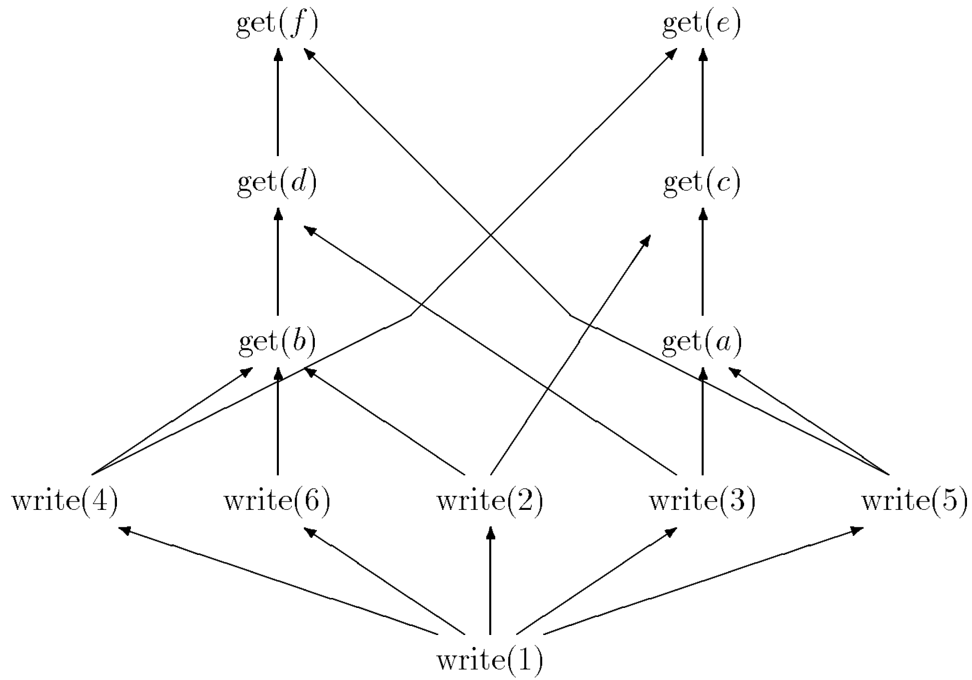


Figure 9.3: A further illustration of the extended partial order relation
The diagram describes the interleaving of

write(1) ‖	get(a)
get(b)	write(2)
write(3)	get(c)
get(d)	write(4)
write(5)	get(e)
get(f)	write(6)

---

the preceding interleaving. Since we cannot determine this *a priori* in any single step of the interleaving (because subsequent interleaving may invalidate the relation) it is a manifestation not a part of the general partial order relation.

I have not mentioned Read operations in the discussion. The behavior of read operations can simply be modeled by the substitution of each Read for a sequence consisting of a Get then Write operation.

What is characteristic of the view presented here is, again, the single indivisible nature of an assignment, its clear association with the value it assigns to the intermediary or variable, and the notion of interleaving which allows us to view the behavior of an intermediary as events with a partial order.

I have laid the foundation for an understanding of a formal semantic description of the *Ease* model. In the definition that follows I use CSP interleaving and traces as the formal basis for the model but first, for those unfamiliar with CSP I present a brief overview.

9.3 A brief overview of CSP

In the first report that presented *Ease* published at Yale University, I deferred all detailed semantic issues to CSP, much as had been done with Occam, indicating how the semantics of the *Ease* model could be expressed in CSP. Hoare's book on CSP is a remarkable piece of work but it took me several years before I began to understand the depth and insight held in it. In considering the problem of how to formalize the semantics of *Ease* and considering the audience for it I balked at the thought that other engineers would have to dig to the depths I had before they could understand the model I was presenting.

I made several attempts to express the semantics in a simpler form; an educational — and chastising — experience. Coming up with a whole new solution to behavioral semantics is a) akin to reinventing the wheel and, b) requires depth of knowledge and experience of the kind manifestly held by Prof.C.A.R.Hoare. Thus it is that I gladly return to the path of righteousness and recognise that my initial intuition was both expedient and correct.

In the following section I introduce CSP and in the subsequent section present the semantics of the *Ease* model in it. This may puzzle the reader a little who has not followed earlier discussion since CSP may be regarded as the bastion of message passing.

Hoare repeatedly claims CSP is a programming language in the conventional sense; if this is so it is several decades ahead of its time in that role. Occam was an attempt to bring CSP into the realm of engineering — early. As a programming tool it is subject to all the foregoing criticism against generalized message passing. In *Ease* I have maintained a dependence on the rigorous mathematical foundations of CSP but provided a simpler model as an engineering solution; usable by the average engineer and efficient to implement.

In the following semantics an *Ease* context is a subordinate process shared by the processes

interacting with it. Concurrency between these processes is thus described using the CSP interleaved form; since *Ease* processes interact only via contexts this form is perfect for our needs.

Thus CSP is viewed here as a mathematical foundation for programming. The concepts of event found within CSP are fundamental to program behavior in the truest sense.

This thesis argues against generalized message passing as a high level programming model and my above salutation to CSP in no way does harm to that argument nor does my argument do harm to CSP; indeed, they are complementary tools.

A complete description of CSP can be found in [Hoa85]. The following is a brief, incomplete, precise based on that work and should provide enough understanding to follow the later semantic description. The omissions avoid unnecessary distractions and are not required to understand the later notation.

9.3.1 Events, alphabets and processes

In CSP we are first asked to invent a set of distinct names that describe the possible events in which a process is prepared to engage. This set of names is called the *alphabet* of a process.

A simple vending machine can be described by two possible events

- *coin* — the insertion of a coin, and
- *choc* — the delivery of a chocolate.

Each name describes an event; there may be many instances of each event in a *trace* of given process. In a conventional sense these names differ from procedural abstractions in that they represent events that concurrently composed processes having the event in their alphabet must perform simultaneously.

The alphabet of a process is denoted by the symbol α; i.e.,

$$\alpha VMS = \{coin, choc\}.$$

A process with the alphabet αVMS that never does anything is called STOP_αVMS. This describes a broken vending machine; even though it is equipped with the ability to instance one of the events in αVMS, it never does so.

A process

$$coin \rightarrow VMS$$

behaves like the event *coin* and then like VMS; said, "*coin* then VMS". This *prefix* describes the evolving behavior of a process with an alphabet αVMS, thus

$$\alpha(coin \rightarrow VMS) = \alpha VMS.$$

A vending machine that consumes a single coin then breaks is described by

$$coin \rightarrow \mathrm{STOP}_{\alpha VMS}.$$

Recursion is used extensively in CSP as a notational convenience; e.g.,

$$VMS = (coin \rightarrow (choc \rightarrow VMS))$$

defines a simple vending machine that serves chocolate endlessly. This is, in fact, an abbreviation of a more formal recursive definition

$$VMS = \mu X : \{coin\}, choc.(coin \rightarrow (choc \rightarrow X)).$$

This definition introduces a label and binds the alphabet to it; said, "VMS is the process X with the alphabet $\{coin, choc\}$ that behaves like *coin* then *choc* then X".

So far we have the tools to describe the sequential behavior of processes, a choice

$$(x \rightarrow P|y \rightarrow Q)$$

is a process whose behavior is dependent on the first event to occur; said, "x then P choice y then Q". Let x and y be distinct events, then if the first event is x the behavior of the process is

$$x \rightarrow P,$$

if the first event is y the behavior of the process is

$$y \rightarrow Q$$

.

The alphabet of a choice is consistent with what we have seen so far:

$$\alpha(x \rightarrow P|y \rightarrow Q) = \alpha P = \alpha Q.$$

At this point we can relate alphabets to conventional programming language experience: the alphabet of a process is analogous to the set of names in a scope, i.e., it is the environment. We see from the above equation that the components of a choice share the same scope, as we would expect.

A vending machine that accepts a coin and then returns either a chocolate or a toffee is defined

$$VMCT = coin \rightarrow (choc \rightarrow VMCT|\mathrm{toffee} \rightarrow VMCT)$$

The type of choice described here is known as *deterministic choice* since the first event is determined either by the environment or is known at the moment it occurs; e.g., a user of the above vending machine inserts a coin and then may take either a chocolate or toffee, and thus the first event in the choice is known.

Non–deterministic choice is a selection between processes in which the choice event is unknown, denoted

$$P \sqcap Q$$

said, "P or Q"; this process behaves either like P or like Q and the environment has no control or knowledge of the choice made.

Generalized choice, denoted

$$P [] Q$$

combines the features of deterministic choice and non–deterministic choice. If the first action is a possible first action of P then P is selected, if the first action is possible for both P and Q then the choice between them is non–deterministic. The *Ease* choice construct has the generalized form; i.e., it is possible to describe all these choice forms.

9.3.2 Traces

The behavior of a process is recorded as a *trace*; a record of the behavior witnessed by an imaginary observer. Even though the witness may see simultaneous events (in parallel processes) they are recorded in sequence; the order the witness chooses to note such events is unimportant.

A trace

$$< coin, choc >$$

denotes the two events *coin* followed by *choc*, and

$$<>$$

denotes an empty trace; the trace of a process that has yet to engage in any of its possible events. A possible trace of the first four events of $VMCT$ is

$$< coin, toffee, coin, choc >$$

Several useful operations are defined on traces. Concatenation is denoted

$$s \hat{} t$$

for example

$$< coin, toffee > \hat{} < coin, choc > = < coin, toffee, coin, choc >$$

concatenates two traces.

Restriction limits the observable trace to a specified set of events; i.e.,

$$< coin, toffee, coin, choc > \lceil \{toffee\} = < toffee > .$$

Generally,

$$t\lceil\mathcal{A}$$

is the trace of events in t specified by the set of events $\mathcal{A}$.

Each event in a nonempty trace is identified by a subscription and in particular the first event in the nonempty trace s is s_0. The remainder of a trace from which the first event is removed is denoted s'; i.e.,

$$< x, y, z >_0 = x$$

and,

$$< x, y, z >' = < y, z > .$$

The set of all finite traces formed by a set of events $\mathcal{A}$ is denoted

$$\mathcal{A}* .$$

The length of a trace is denoted

$$\#t$$

i.e,

$$\# < x, y, x > = 3.$$

A trace s is said to be a prefix of the trace t if there exists a set u such that

$$s^\wedge u = t.$$

This defines an ordering relation $s \leq t$; i.e.,

$$< x, y > \leq < x, y, x, w >,$$

$$< x, y > \leq < z, y, x > \equiv x = z.$$

The $\leq$ relation is a partial ordering with the empty trace as the least element. If s is any subset of t (including a prefix) we say

$$s \textbf{ in } t.$$

If s is a trace of P then

$$P/s$$

said, "P after s", is the behavior of P after the trace s has occurred.

9.3.3 Parallel processes

If two processes composed in parallel

$$P \| Q$$

have intersecting alphabets they interact; i.e., events that occur in both processes require simultaneous participation.

A greedy customer (page 66, [Hoa85]) will take chocolate or toffee from a vending machine without paying; however, the customer pays a coin for chocolate if given no other option; i.e.,

$$\begin{aligned} GRCUST = \; & (toffee \rightarrow GRCUST \\ & |choc \rightarrow GRCUST \\ & |coin \rightarrow choc \rightarrow GRCUST) \end{aligned}$$

The parallel composition of our greedy customer and a vending machine ($VMCT$, defined earlier) is

$$GRCUST \| VMCT.$$

The first event that each process is ready to participate in simultaneously is the event *coin* after which the greedy customer selects chocolate. The vending machine is only prepared to participate in the event *toffee* after payment; hence the customer only gets what is paid for (TANSTAAFL![3]).

In the above case both processes have the same alphabet; i.e.,

$$\{coin, choc, toffee\}.$$

Consider two processes with disjoint alphabets

$$P = (up \rightarrow down \rightarrow P)$$

and,

$$\begin{aligned} Q = \; & (right \rightarrow left \rightarrow Q \\ & |left \rightarrow right \rightarrow Q) \end{aligned}$$

In the parallel composition

$$P \| Q$$

there is no common event; the events of the parallel processes can be considered as an arbitrary interleaving. To consider such processes in this way makes no difference to our reasoning and we do not have to juggle with notions of "true concurrency".

The interleaving operator composes parallel processes with the same alphabet; i.e., in

$$P|||Q. \alpha P = \alpha Q$$

[3] "There Ain't No Such Thing As A Free Lunch" — Robert Heinlein.

the events of P and Q are interleaved. If both processes can instance an event only one of them does so; which one is non–deterministic; e.g., a vending machine that will accept up to two coins before dispensing up to two chocolates (page 120, [Hoa85]) is

$$VMS|||VMS.$$

A consequence of what we have seen so far is

$$VMS||VMS = VMS.$$

The composition of two processes with the same alphabet behaves as one since each process must participate in each event. The alphabets of two processes can be made disjoint by appending a unique label to every event in the alphabet; i.e.,

$$left : VMS||right : VMS$$

are two distinct processes; vending machines standing side by side.

9.3.4 Communication

A communication is an event

$$c.v$$

where c identifies a *channel* on which the communication takes place and v identifies the message passed. A process

$$(c!v \to P) = (c.v \to P)$$

is "output v then P"; the first event is the communication $c.v$. Similarly,

$$(c?x \to P(x)) = (c.v \to P)$$

is "input a value x then $P(x)$"; the first event is the communication $c.v$.

9.3.5 Subordination

A subordinate process is a process whose alphabet is contained in a principal process; i.e.,

$$\alpha P \subseteq \alpha Q$$

In the parallel composition of these processes each action of P can occur only when Q permits it, while Q can independently act on those events not in the alpabet of P. As such, P serves as a slave to Q.

The composition

$$P//Q$$

is the parallel composition of the principal Q with a subordinate process P. The alphabet of P must be some subset of the alphabet of Q, and that subset is hidden in the composition from the view of external processes, so that

$$\alpha(P//Q) = (\alpha Q - \alpha P).$$

It is useful to be able to identify the subordinate explicitly from within a principal so all interactions with it are clearly identified, thus

$$m : P//Q$$

where m identifies actions in P. Each interaction is a triple

$$m.c.v$$

where $\alpha m.c(m : P) = \alpha c(P)$ and $v \in \alpha c(P)$, where $c(P)$ is the set of communications in P. Q communicates with P on channels with the compound name, say, $m.c$ or $m.d$ where P uses the simple names c and d for the corresponding communication.

9.3.6 Conditional

A conditional

$$P \lhd b \rhd Q$$

is P is b is true and Q otherwise.

The reader should now have enough understanding of CSP to understand the following semantics. For complete details of the notation used in this section see [Hoa85].

9.4 A CSP semantics of the *Ease* model

In the following section I provide the behavioral semantics of the *Ease* model; contexts, interaction operations and process constructions. A context is described as a CSP subordinate process.

9.4.1 Singleton semantics

$$\begin{aligned}
&\alpha\text{write} = \alpha\text{read} = \alpha\text{get}\\
&\text{SINGLETON} = \text{write}?x \rightarrow \text{SINGLE}_x\\
&\text{ where SINGLE}_x = \quad (\text{write}?y \rightarrow \text{SINGLE}_y\\
&\qquad\qquad\qquad\qquad |\text{read}!x \rightarrow \text{SINGLE}_x\\
&\qquad\qquad\qquad\qquad |\text{get}!x \rightarrow \text{SINGLETON})
\end{aligned}$$

Initially a singleton is only prepared to accept a write action then either a write, read or get. A write will change the value, a read will return the value, a get returns the value and then causes the singleton to return to an empty state.

9.4.2 Stream semantics

$$\begin{aligned}
&\alpha\text{write} = \alpha\text{read} = \alpha\text{get}\\
&\text{STREAM} = \text{STR}_{<>}\\
&\text{where } \text{STR}_{<>} = \text{write}?x \rightarrow \text{STR}_{<x>}\\
&\ \text{and } \text{STR}_{<x>^\wedge s} = \ (\text{write}?y \rightarrow \text{STR}_{<x>^\wedge s^\wedge <y>}\\
&\qquad\qquad\qquad |\text{read}!x \rightarrow \text{STR}_{<x>^\wedge s}\\
&\qquad\qquad\qquad |\text{get}!x \rightarrow \text{STR}_s)
\end{aligned}$$

where s is a trace. In the above traces are used to keep track of the values in the stream. $<>$ is the empty trace, $< x >^\wedge s$ is the trace whose first component is x and whose remainder is s, $< x >^\wedge s^\wedge < y >$ is the trace whose first component is x, whose last component is y, and whose remainder is again s.

The initial action of a stream will be a write, a read returns the least recent value in the trace but does not change the trace state, a get behaves like a read but deletes the least recent value from a trace. A stream with an empty trace will only accept a write action.

9.4.3 Bag semantics

$$\begin{aligned}
&\alpha\text{write} = \alpha\text{read} = \alpha\text{get}\\
&\text{BAG} = \|_{i\geq 0} i : \text{STORE}\\
&\ \text{where STORE} = \ \text{write}?x \rightarrow\\
&\qquad\qquad \mu X.\ \ (\text{read}!x \rightarrow X\\
&\qquad\qquad\qquad |\text{get}!x \rightarrow \text{SKIP})
\end{aligned}$$

Bags are described as an infinite set of processes each of which will accept a single write and then either a read or get; these return the value given by the write. A read is simply recursive. The process terminates after satisfying exactly one get.

9.4.4 Context usage

A context named c in CSP is defined for a single process P by

$$c : \text{SINGLETON}//P$$

i.e., a context is a subordinate process (in the CSP sense).

A context shared by multiple processes must share its channels. In CSP we define this requirement by interleaving; e.g.,

$$c : \text{SINGLETON}//(P|||Q).$$

Interactions by P and Q with the context will be arbitrarily interleaved.

9.4.5 Interaction operators

The following illustrates the definitions of *Ease* actions (*Ease* on the left with the CSP implementation on the right) for singleton and stream contexts. Functional notation is used to avoid confusion with the actual *Ease* syntax.

$$\begin{aligned}
\texttt{read}(c, v) &= c.\text{read}?v \\
\texttt{write}(c, e) &= c.\text{write}!e \\
\texttt{get}(c, v) &= c.\text{get}?v \\
\texttt{put}(c, v) &= (c.\text{write}!e; v{:=}\text{undefined})
\end{aligned}$$

where c identifies a context and "undefined" is an arbitrary value of the type.

The following again illustrates the *Ease* syntax on the left with the CSP implementation on the right for bag contexts.

$$\begin{aligned}
\texttt{read}(c, v) &= []_{i\geq 0} c.i.\text{read}?v \\
\texttt{write}(c, e) &= []_{i\geq 0} c.i.\text{write}!e \\
\texttt{get}(c, v) &= []_{i\geq 0} c.i.\text{get}?v \\
\texttt{put}(c, v) &= []_{i\geq 0} (c.i.\text{write}!e; v{:=}\text{undefined})
\end{aligned}$$

Interactions on bags make an arbitrary choice from the ready processes which comprise the bag.

9.4.6 Shared resources — combinations

Call reply resources are said to block on the pending input. In fact, whether they block on the pending input or the output makes no difference to the behavior. We will find that by keeping the output data locally until a receiver is ready (as for point to point communication) gives an advantage both in implementation and in semantic description. As a result resources may be simply described and turn out to look just like CSP resources; except that both ends of the exchange are well defined.

An *Ease* resource syntax

$$\texttt{RESOURCE}\ (c, f(x))$$

is an abbreviation for the sequence

$$\texttt{get}(c, x); y := f(x); \texttt{write}(c, y)$$

or

$$\texttt{get}(c, x); y := f(x); \texttt{put}(c, y)$$

where the type of the components of c guarantee that the write will satisfy the input of a corresponding call sequence.

$$y := \texttt{call}(c, x)$$

Which, in turn is an abbreviation for

$$\texttt{write}(c, x); \texttt{get}(c, y)$$

or

$$\texttt{put}(c, x); \texttt{get}(c, y)$$

Given the CSP definition

$$\text{RES} \;=\; \text{resource } ?x!f(x),$$

then

$$\texttt{RESOURCE }(c, f(x)) = \text{RES}$$

where

$$\texttt{RESOURCE }(c, f(x)) = \\ \texttt{get}(c, x); y := f(x); \texttt{write}(c, y)$$

and,

$$\texttt{RESOURCE }(c, f(x)) \;=\; (\text{RES}; y = \text{undefined})$$

where

$$\texttt{RESOURCE }(c, f(x)) = \\ \texttt{get}(c, x); y := f(x); \texttt{put}(c, y).$$

RES takes a little liberty with CSP syntax but the meaning is obvious. Once again a resource is a CSP subordinate,

$$c : \text{RES}//P$$

and call semantics are thus straight forward:

$$x := \texttt{call}(c, e) \;=\; c.\text{resource}!e?x$$

where

$$x := \texttt{call}(c, e) \;=\; \texttt{write}(c, x); \texttt{get}(c, y)$$

and,

$$\begin{aligned} x := \texttt{call}(c, e) \;=\; & c.\text{resource}!x?y; \\ & (x := \text{undefined} \lhd x \neq y \rhd \text{SKIP}). \end{aligned}$$

where

$$x := \texttt{call}(c, e) \;=\; \texttt{put}(c, x); \texttt{get}(c, y)$$

9.4.7 Parallel semantics

In the following, *Ease* parallel composition is defined to be equivalent to the CSP interleaved parallel construction. The notation on the left of the equations is *Ease* the notation on the right is CSP.

Cooperation

$$\{\|P\|Q; R\} \;=\; ((P|||Q); R)$$

is the sequential composition of a parallel construction with subsequent processes (represented by R). Subordination

$$\{//P//Q; R\} \quad = \quad (P|||Q|||R)$$

is the parallel composition of a parallel construction with subsequent processes.

The interleaving form of the parallel construction may be used since *Ease* processes do not interact directly but rather interact indirectly via what are, in CSP, subordinate processes; i.e., Contexts, the *Ease* shared data structures.

It is important to note that in *Ease* all variables are accessible only to the sequential process for which they are specified. As a result the alphabets (the possible events in a process) of *Ease* processes are completely disjoint.

Chapter 10

Ease Implementation

Simple variations on queues provide full support for *Ease* model implementations. In the following discussion the focus is on the implementation of context structures since these are the unique components of the model. The primary issue is one of data distribution and placement.

CSP is used throughout to describe the implementation of *Ease* context structures. In CSP the behavior of a queue is simply

$$\begin{array}{l} \alpha\text{write} = \alpha\text{read} = \alpha\text{get} \\ \text{QUEUE} = \text{QUE}_{<>} \\ \text{where QUE}_{<>} = \text{write?}x \rightarrow \text{QUE}_{<x>} \\ \text{and QUE}_{<x>^\wedge s} \\ \;= \;\; (\text{write?}y \rightarrow \text{QUE}_{<y>^\wedge<x>^\wedge s} \\ \qquad\;\; |\text{read!}x \rightarrow \text{QUE}_{<x>^\wedge s} \\ \qquad\;\; |\text{get!}x \rightarrow \text{QUE}_{s}) \end{array}.$$

Initially, and while the queue is empty, it is prepared only to accept a value and record it. While values remain in the queue it is prepared to continue accepting and recording values, or return the last value recorded, or return the last value and remove it from the record.

10.1 The implementation of Bags and Streams

The only distinction between QUEUE and STREAM (defined in the previous section) is the ordering of the recorded sequence; QUEUE is a last–in–first–out (LIFO) buffer while STREAM is first–in–first–out (FIFO). If y is the input and x is the pending output, then STREAM stores $< x >^\wedge s^\wedge < y >$ and QUEUE stores $< y >^\wedge< x >^\wedge s$ (for similar CSP examples see [Hoa85]).

Since the ordering of values present in Bags is nondeterministic it is perfectly valid for

an implementation to choose to impose any order of its own; thus an implementation may choose to implement BAG either by STREAM or QUEUE for example.

By using BAG the programmer has simply stated that there is no dependency in the algorithm on the order of context values. It would be improper for a programmer to introduce such dependency based upon an understanding of a particular implementation.

STREAM guarantees the order of values and must therefore be implemented as described in the semantics as a FIFO buffer.

How are STREAMs implemented in a distributed memory environment? To maintain order consistency there should be a single process implementing STREAM; i.e., the STREAM data remains on a single node. Each remote process accessing the Stream must therefore be represented in the STREAM implementation. A distributed implementation of Streams where the Stream structure itself is distributed presents consistency problems since the contributed order must be maintained, while strategies to manage such consistency are possible, they are likely to prove so expensive as to invalidate their usefulness.

There are demands in these implementations on the machine architecture and run time system that we should just recap. A *node* in the sense used here is a machine with a common address space; regardless of the number of processors that share that space. Thus a node may execute some number of processes all of which may be accessing a single context. The run time system must provide a scheduling mechanism; usually a “threads” package (UNIX Fork is likely to provide too large an overhead), or hardware support as found on the transputer. The run time system support for communication between nodes, for the implementation described here, enables point–to–point process connection; i.e., a process on one node connected to a process on another node. Such mechanisms are becoming available in hardware and are available today in run time systems such as Express, PVM and P4 [Par88, PVM, P4]. With this support STREAM can be implemented on a distributed machine directly by the description given in the semantics as a FIFO queue.

BAG contexts have no defined order, and so the order consistency problem associated with implementing the distributed structure of a Stream does not exist. It is therefore less difficult to implement a distributed BAG structure. An implementation may choose to avoid data replication[1] to avoid consistency complications, alternatively an implementation may choose to replicate data to increase accessibility. The increased accessibility provided by replication may be particularly desirable in distributed network computing where node interconnection bandwidth is low, and also in process optimization where analysis or compiler directives identify the structure as infrequently changed. Several strategies to manage replication exist and a reasonable discussion of these can be found in [Bal89]. A replication strategy suitable for *Ease* implementations is outline in the following paragraphs.

Before considering the detail of a BAG implementation recall the semantics. We find BAG is an infinite set of processes each of which will accept a single Write and then either a Read or Get, terminating on Get. Actions upon elements in this infinite set make a arbitrary selection

[1] Maintaining multiple copies of a single data item.

between them. This description will, in fact, be difficult and costly to implement directly. Therefore, based on the assertion that an implementation may choose to implement BAG with any actual ordering, either QUEUE or STREAM are perfectly valid implementations. Generally, an implementation can be reasonably expected to select an implementation strategy for BAG that is efficient on the architecture of the machine — taking advantage of the nondeterministic ordering semantics.

How can a Bag context structure then be implemented on a distributed memory machine? Consider the simple example of implementing a distributed BAG on a machine with just two nodes, without replication. The context implementation at each node behaves as

$$
\begin{array}{l}
\mathrm{TWIN}_i = \mathrm{NODE}_{<>} \\
\text{where } \mathrm{NODE}_{<>} \\
\;= \; (\text{i.write?}x \rightarrow \mathrm{NODE}_{<x>} \\
\qquad |\text{i.readrq} \rightarrow \; (\text{i.write?}x \rightarrow \text{i.read!}x \rightarrow \mathrm{NODE}_{<x>} \\
\qquad\qquad\qquad\quad |\text{rget?}x \rightarrow \text{i.read!}x \rightarrow \mathrm{NODE}_{<x>}) \\
\qquad |\text{i.getrq} \rightarrow \; (\text{i.write?}x \rightarrow \text{i.get!}x \rightarrow \mathrm{NODE}_{<>}) \\
\qquad\qquad\qquad\quad |\text{rget?}x \rightarrow \text{i.get!}x \rightarrow \mathrm{NODE}_{<>})) \\
\text{and } \mathrm{NODE}_{<x>\wedge s} \\
\;= \; (\text{i.write?}y \rightarrow \mathrm{NODE}_{<x>\wedge s \wedge <y>} \\
\qquad |\text{i.readrq} \rightarrow \text{i.read!}x \rightarrow \mathrm{NODE}_{<x>\wedge s} \\
\qquad |\text{i.getrq} \rightarrow \text{i.get!}x \rightarrow \mathrm{NODE}_{s} \\
\qquad |\text{rget!}x \rightarrow \mathrm{NODE}_{s})
\end{array}
$$

Initially, and while empty, TWIN is prepared to accept a value and record it, a Read request, or a Get request. Following an input request TWIN either accepts a value from the local processes or accepts a value from the other member of the pair on the neighbouring node. Thus a written value is initially placed on the local node and stays until required to satisfy a non–local input; at which point the value "migrates" to its neighbour. The composition

$$c : (TWIN_0 || TWIN_1)//(P_0 ||| Q_1)$$

illustrates usage of the Bag context c, implemented by the pair $(TWIN_0 || TWIN_1)$. Process P_i on a node identified by $i = 0$ and a process Q_i on a node identified by $i = 1$ denote the processes on the two nodes that use the context. Within P_0 and Q_1 *Ease* interaction operators behave as

$$
\begin{array}{lcl}
\texttt{read}(c, v) & = & (\text{readreq}; c.i.\text{read?}v) \\
\texttt{write}(c, e) & = & c.i.\text{write!}e \\
\texttt{get}(c, v) & = & (\text{getreq}; c.i.\text{get?}v) \\
\texttt{put}(c, v) & = & (c.i.\text{write!}e; v{:=}\text{undefined})
\end{array}
$$

The interaction between the two nodes (i.e., each instance of TWIN) is described by the "rget" event. A read request to an "empty" local Bag will be fulfilled by a value either from a subsequent Write on the local node or a value existing or subsequently written on the neighbour.

Local processes are decoupled by explicit Read or Get requests; i.e., in the interleaving of local processes (implemented perhaps by the scheduling lists mentioned in chapter 3) one process may have a request outstanding — between the request event and the satisfaction of the input local events continue to occur. In particular, allowing the read to be satisfied by a local write or put.

10.1.1 Reference exchange

Recall that, though not explicitly shown here, the definition of Get and Put actions allow their implementation by reference exchange. Reference exchange should however be avoided when the components of a context are types implemented by only a few bytes. The trade offs are system dependent, with cache characteristics likely to have particular effect, however generally it will be inefficient to implement contexts of scalars by reference exchange (since pointers are around the size of the data item this should be obvious). In such cases an implementation may, for efficiency reasons, choose to implement Put as Write; the fact that the variable used in the Put remains the same value subsequently is an incidental fact that the programmer should ignore since other implementations are at perfect liberty to assign the variable some arbitrary value. In all cases implementations should provide an error message or warning indicating usage of an undefined value, be it initial usage of a variable declared undefined or a variable used after a Put action upon it. It is an error to use an undefined value in an expression.

10.1.2 Replication

The implementation described above has a high bandwidth cost in a case where a single value is written and processes on each node proceed to Read repeatedly, possibly resulting in the value "bouncing" between the two nodes. In such cases replication will prove desirable.

Replication presents state and (in the case of Streams) order consistency problems and it becomes necessary to clearly identify the duplicate values and synchronization cross dependencies. The cost of replication is thus greater complexity of implementation and additional overhead for consistency protocol, in whatever form that may be. An implementor using replication should be convinced that the semantics of the model are maintained for, as we shall see, cross dependency between interactions on different contexts present subtle difficulties. However, simple and useful demand driven replication strategies can be conceived.

Consider the simplest case of replication between the nodes of the TWIN machine studied above. Replication can be greatly simplified for Bags. Regardless of the number of values written at a node a copying neighbour need acquire only a single value. Value selection in Bags is nondetermininistic and process interaction is indirect and nonsynchronized, this makes it permissible for the process on one node to continue reading a copy of a value deleted from another provided that value is not used to satisfy a Get; in other words an arbitrary number

of Reads in one process may be interleaved before a Get in another, though, as we shall see, cross dependencies must be accounted for.

Once a copy of a remote value has been acquired we must be careful to ensure local Write and Get operations continue to be satisfied. If a local write occurs the copied value is no longer required, and to maintain consistency a copy must not satisfy a Get. This behavior is implemented by the choice

$$
\begin{aligned}
DUP = \; & (\text{i.write?}x \rightarrow \text{NODE}_{<x>} \\
& |\text{i.getrq} \rightarrow \; (\text{i.write?}x \rightarrow \text{i.get!}x \rightarrow \text{NODE}_{<>} \\
& \qquad\qquad\quad |\text{rget?}x \rightarrow \text{i.get!}x \rightarrow \text{NODE}_{<>}) \\
& |\text{i.readrq} \rightarrow \text{i.read!}y \rightarrow DUP)
\end{aligned}
$$

where y is the copied value. The context

- accepts a value from a local process and subsequently disregards the copy, or
- accepts a Get request and satisfies the Get either with a value written locally or one from the neighbour, the copy is ignored since it may no longer be valid (i.e., it may have satisfied a Get on the neighbour), or
- satisfy any number of Reads with the value of the copy.

Since the processes using the context on each node are entirely disjoint there is no sequentialization of operations in them by definition (except, as we shall see shortly, by cross dependency). Thus even though the node that supplied the copy may be empty the copy continues to satisfy Read actions. It would clearly be an error to permit the empty node to take a copy of the copy in return and we do not permit it. We can now see the complete implementation, TWIN is replaced by

$$
\begin{aligned}
& \text{RTWIN}_i = \text{NODE}_{<>} \\
& \text{where NODE}_{<>} \\
& = \; (\text{i.write?}x \rightarrow \text{NODE}_{<x>} \\
& \qquad |\text{i.readrq} \rightarrow \; (rcopy?y \rightarrow DUP \\
& \qquad\qquad\qquad\quad |i.write?x \rightarrow \text{i.read!}x \rightarrow \text{NODE}_{<x>}) \\
& \qquad |\text{i.getrq} \rightarrow \; (\text{i.write?}x \rightarrow \text{i.get!}x \rightarrow \text{NODE}_{<>}) \\
& \qquad\qquad\qquad\quad |\text{rget?}x \rightarrow \text{i.get!}x \rightarrow \text{NODE}_{<>}) \\
& \qquad) \\
& \text{and NODE}_{<x>\wedge s} \\
& = \; (\text{i.write?}y \rightarrow \text{NODE}_{<x>\wedge s \wedge <y>} \\
& \qquad |\text{i.readrq} \rightarrow \text{i.read!}x \rightarrow \text{NODE}_{<x>\wedge s} \\
& \qquad |\text{i.getrq} \rightarrow \text{i.get!}x \rightarrow \text{NODE}_s \\
& \qquad |\text{rget!}x \rightarrow \text{NODE}_s \\
& \qquad |\text{rcopy!}x \rightarrow \text{NODE}_{<x>\wedge s})
\end{aligned}
$$

Now reconsider the implementation of Streams, this time with replication: while the Stream structure itself is maintained on a single node replication is a valid strategy where high Read frequency exists on a node; recall that a node may have many processes reading the value. The ordering characteristics of Streams guarantee that a process will not input a value output earlier than a previous value input by that process. This permits us to implement Read by copy buffering, so that a sequence of Reads will read the same value continuously unless that process performs a get. This strategy is justified because of the nature of interleaving, since we may choose to interleave all the Read operations before a parallel Get. Read and Get on STREAM are then implemented

$$\begin{aligned} \texttt{read}(c, v) &= (v := y \lhd \text{buffered} \rhd c?y; v := y); \\ &\quad\ \text{buffered} := \text{true} \\ \\ \texttt{get}(c, v) &= \text{buffered} := \text{false}; c?v \end{aligned}$$

where "buffered" is initially false, and y is the buffer. An implementation should be careful not to use replication where the read frequency on a node does not demand it. A node may have many processes that perform a read on a single context, or a single process; a copy penalty will be unnecessarily incurred if low frequency reads are buffered.

Unfortunately, these implementations with replication as described are not valid in all circumstances since they do not allow for a state consistency problem caused by cross dependencies. This problem can be illustrated by considering the following sequences of interactions upon two contexts c and k,

$$(c : (BAG \sqcap STREAM) \| k : (BAG \sqcap STREAM) // D$$

$$\begin{aligned} D \ = (&\quad (c.\text{write}!0; c.\text{get}?v; k!0) ||| \\ &\quad (c.\text{read}?x; k.\text{read}?y; c.\text{read}?z) \quad) \end{aligned}$$

this process contains a certain deadlock that, given our implementation of replication, will not occur. This might be considered a good thing but the semantics are being broken and thus, we assume, the intention of the programmer ignored. The deadlock should certainly occur with $c.\text{read}?z$, though it may occur with $c.\text{read}?x$. The traces are

$$< c.\text{write}!0, c.\text{get}?v >$$

$$< c.\text{write}!0, c.\text{read}?x, c.\text{get}?v, k.\text{write}!0, k.\text{read}?y >$$

in both cases the Get $c.\text{get}?v$ removes the value in c and since c is now empty the subsequent read operations on c cannot occur, our implementation may safely ignore the first trace above since a valid trace remains possible, and the buffering permits this. Unfortunately the buffering will also permit the instance of $c.\text{read}?z$ to complete allowing a trace

$$\begin{aligned} &< c.\text{write}!0, c.\text{read}?x, c.\text{get}?v, \\ &k.\text{write}!0, k.\text{read}?y, c.\text{read}?z > \end{aligned}$$

and this is not a trace of the process. This example contains a cross dependency caused by the interaction upon the context k that causes the two processes to synchronize, illustrating that the interactions on a context cannot be looked at in isolation.

Managing this inconsistency is straight forward but potentially expensive. The problem can be settled by simply invalidating a context buffer when the risk of a cross dependency exists. A simple strategy is to invalidate the buffer before the next Read action in a sequence if that action is upon a different context. This can easily be detected by the compiler which can insert the appropriate invalidation instructions.

This implementation of demand driven replication (in fact, simple cache buffering) applies to all cases where a node directly communicates with the context structure but is not itself a part of that structure; e.g., in the case of actions upon Streams. The implementation of these interactions simply requires that the "buffer" flag is invalidated before a Read if the previous Read was on some other context. An invalidation event must be introduced for Bag implementation in DUP to allow the buffer to be invalidated.

This strategy is a blanket one; buffers are invalidated even if no cross dependency exists, further, it is not useful in processes where Reads occur in sequence alternately on two contexts. Alternate Reads will invalidate the buffer in turn; introducing the overhead of invalidation and buffering with none of the benefits. Solving this problem demands greater complexity in analysis. So far invalidation of buffers can be identified with minimum effort by the compiler, which needs only to determine if the previous input was on a different context. A compiler may thus choose not to replicate on a node where sequences of alternate Reads occur, but to improve the efficiency the compiler must determine the full cross dependency risk and this demands complete deadlock analysis.

Deadlock analysis is useful to the programmer as well as the compiler. A compiler may well (perhaps should) choose to ignore potential deadlock traces of a process behavior in favor of other non–deadlocking traces. If a compiler does aid the programmer in this way it should issue warnings since some other implementations may choose to do otherwise. Automatic deadlock analysis is a continuing subject of research and early work looks promising, *Ease* compilers should provide such analysis as a debugging aid and it is hoped a simple strategy, currently being researched and under development, will prove successful.

10.1.3 Bags on more than two distributed memory nodes

So far I have considered the implementation of Streams and Bags on a two node machine only. What about machines with more than two nodes? Obviously, given the run time system requirements, STREAM can be implemented directly on multiple nodes, with or without replication, the operations are simply distributed in the natural way (recall a run time system, such as Express, provides the internodal connectivity required).

An implementation can choose to implement Bags similarly by extending the direct events in the TWIN pair, or may extend TWIN by enhancing the rget communication event. The

rget communication can be enhanced simply by replication of the communication over the number of nodes implementing the Bag, possibly by local broadcast. In the first case a Bag is implemented by a combination of the two strategies; several closely coupled nodes implement the Bag structure and more distant nodes directly communicate with an implementing node, possibly using the data buffering replication described above.

10.2 The implementation of Singletons

The distinguishing characteristic of Singletons is that they are identities. Each Singleton has only a single value and arrays of Singletons are identified by subscription. While writes replace the previous value, the Singleton possesses an "empty" state that represents the initial state and the state subsequent to a Get.

A simple optimization is immediately observable: if there are no Gets present in the scope of a Singleton then there is no need to track the post–Get empty state after the Singleton has been initialized. This can greatly simplify the Singleton context implementation — especially when dealing with large, potentially distributed, matrices. In such cases the Singleton can simply be represented by memory and conventional load and store operations and in the distributed case the Singleton may be simply read or written to by a remote node without synchronization; the distributed implementation builds a process associated with the context to perform remote reads and writes.

In the case where Singletons are acted upon by Get an implementation using a queue (semaphore) for each is required and the behavior is exactly that defined for SINGLETON in the previous chapter.

The demand driven replication strategy given above may also be used for Singletons.

10.3 Type associative contexts

The context implementation discussed in the previous sections are of contexts of a single type. What about contexts of multiple type and the promised type associativity?

In fact, all type associative contexts are decomposed at compile time into their constituent parts and thus are implemented as in the foregoing sections. This decomposition is simple and straight forward, the compiler in effect decomposes the context into a set of new contexts identified by the name and type. Interactions on the context are then checked to be of a defined type and attributed to the appropriate implementation structure.

10.4 Placement

10.4.1 Manual placement

Ease provides direct placement directives for both process and data to serve optimizers and decomposition tools with differing strategies. In *Ease* the first stage of an *Ease* compiler may transform a program without placement directives into one with such directives, or it may rewrite existing directives.

Whether the placement directives are added explicitly by the programmer or generated automatically, a program with directives is the final form compiled for a machine with more than one node. A program with no directives will be compiled for a single node; recall that such a node may still be a large parallel machine with common memory.

10.4.2 Automatic placement

Strong locality enables effective automatic placement of distributed data structures and processes on a distributed memory machine. Particular placement is very target dependent and optimizing stages for compilers need to be developed for each architecture and application domain to support all existing and new programming models and languages.

However, these automatic strategies are very specialized, generally targeted to a specific machine architecture or application, and it is difficult to see a general purpose placement strategy for large programs and general development. Therefore, current compilers provide manual placement support to allow the programmer to guide the compiler. The strong locality expressed by *Ease* contexts, along with side effect free programming, provides a significant guide to compilers.

10.4.3 Fine grain data parallelism.

Like most other languages with a strong process model *Ease* is well suited to the expression of fine grain data parallel applications as well as the coarser grain MIMD style. Such expressions are very target dependent and are preferably generated by an optimizer from a more general form.

10.5 Program transformation

It is apparent that any parallel processing model that seeks to be general purpose and widely adopted must ensure that it can generate programs that execute efficiently regardless of the architecture and available resources. In particular, the implementation of process interaction

models should not incur a noticeable overhead over native methodologies for a given architecture. These overheads can arise in the implementation of process scheduling and the excessive introduction of copy operations. Only some part of this efficiency can be solved by improving the general efficiency of the model itself and designing the model so as to simplify the task of identifying opportunities for efficient optimization. Ultimately however, a program may need to be "specialized" for the target and not literally translated.

Such program specialization can be achieved by program transformation. Program transformation techniques allow programs to be transformed from a general form into a special form while maintaining the behavioral semantics of the program. The prerequisite for effective transformation is that the language should have a strong mathematical basis. CSP provides that mathematical basis for *Ease*.

Chapter 11

Future work and directions

11.1 Prototype compilers

As of writing, three prototype compilers are in existence. My own is a full implementation of the language with a focus on performance efficiency. Current indications are that *Ease* programs compiled with this compiler will be faster than the equivalent C programs for sequential code; this is especially interesting since both programs use the same C compiler to generate the final code. The research is still in the primary stages but the work looks very promising and statistical experimentation has begun.

For the curious, the gains are simply accounted for. The *Ease* compiler generates code that no human programmer would consider by taking a view that a C compiler architecture represents a universal machine, the gains come from in–line code generation, minimizing the use of automatics, and a general storage strategy that determines run time storage requirements in most cases and allows space reuse. Stack frames usage is limited to standard C library function calls. In addition the reuse strategy greatly improves data temporal and spatial locality and thus improves cache performance. Research and development of these techniques has been constrained by available funding but I hope to continue this research in future work. It is true that with effective analysis the equivalent C program can take advantage of such efficiencies also. However, it is in the nature of C that such analysis is difficult, while *Ease* is defined in such a way that the analysis is simply not required — primarily because effects are well defined. A good compiler strategy enables the direct generation of efficient code.

Work is also in progress on scheduling overhead elimination (static scheduling) and dead lock detection. Scheduling overhead (context switch) can prove significant on uniprocessors and it is important that *Ease* compilers should generate code with competitive performance in such environments[1]. Static scheduling simply "cuts and pastes" the parallel processes in a

[1]Even with scheduling overhead I hope to do well since the compiler trades off the benefits mentioned in the previous paragraph; i.e., I trade procedure context switch for process context switch.

program; i.e., a static scheduler produces a valid interleaving of the program. Static scheduling is not possible where the number of process creations are unknown to the compiler, so such scheduling is restricted to those programs that use constants in replicators and avoid process creation within repetitions, but this is an important class of program, typically embedded system and real time programs.

Two other implementations exist to my knowledge. John Redman, of the University of Western Australia, has also implemented the full language and this implementation is described in his Honours dissertation "An Implementation of the *Ease* Programming Language" [Red91], which I'm pleased to relate won an IBM award as best Honours project. John's work is impressive. He uses the μSystem library, a run time system that provides process (threads) support from the University of Waterloo (USA). John's compiler is derived from a Joyce–Linda compiler originally written by Chris McDonald. This compiler also uses C as a target and runs on a uniprocessor workstation.

One brief aside, John mentions in his report the introduction of a LIFO queue as a first class characteristic for Contexts; this may seem an obvious requirement but is unnecessary. While stacks are an important programming structure I contented that they are less useful for describing data exchange and shared data characteristics, if a stack were used for data exchange then it seems apparent that the ordering characteristic of the data is unimportant and thus the requirement is well met by the Bag characteristic. Shared stacks may, of course, be implemented as resources (an example is given in the appendix).

The second implementation is also in Australia at Monash. Tim MacKenzie has implemented *C–with–Ease*. This compiler also targets C and also uses the Waterloo University μSystem. This implementation is most interesting because it implements a distributed system over a network of workstations, and also produces the first published results from an *Ease* code. This shows close to linear speed up for the restriction mapping problem over 15 DEC 5000 workstations, much as you might expect. Unfortunately, this tells us nothing about the effectiveness of the model since any model would do for the size program run. Much work needs to be done but early results are promising.

On another brief aside, MacKenzie also introduces a primitive "empty" that returns the ready state of a context (and not, as the name suggests, the empty state), this provides the support required to implement Choice like constructions in *C–with–Ease*. I have avoided placing a choice construct in *C–with–Ease* until more experience is gained with implementation; a ready primitive is a reasonable intermediate solution and at low level is essential, however I am sure to prefer a higher level construction in the end. Non–deterministic constructions are notoriously difficult to implement correctly (even the transputer implementation of `ALT` is flawed[BaGoJoKa88]), they are also full of subtle pitfalls in use – whilst I have no hesitation introducing them to the full language based on the strong process model and CSP semantics, I hesitate to do so to *C–with–Ease* until further research is complete.

Choice implementation has not been described in any detail. Briefly, in current implementation a determined choice selects data that is available in the local address space, if none of the inputs is ready the choice is descheduled and other processes are allowed to continue.

A choice does not commit to an input unless the data is available or becomes available. This means that "readiness" implies there must be a concurrent process with access to the local address space prepared to write the data at some point or there must be a default process. Implementation mechanisms to allow the full generality of choice on networked machines are not widely available.

The implementations of *Ease* I have considered so far are all non–preemptive; i.e., processes are not interrupt driven. This approach derives from a general philosophy I adopt that says that in embedded system, real time, applications urgent processing will be provided by the appropriate hardware; i.e., in my design I will provide transducers with processors. This provides a simpler approach to building real time systems, the additional cost of a processor is minimal and to a degree offset by the reduced software costs — I simply must guarantee that the processing latency is shorter than my required response bandwidth and I do that by providing a processor fast enough. For example, a conventional workstation can be built as a multiprocessor machine without interrupts, interacting according to data dependencies. The keyboard would have an appropriate processor, the visual display also, with applications being driven on yet others. In reality such machines are built today — but they are interrupt driven[2]. This issue is beyond the scope of this thesis and I shall say no more than this paragraph on the subject.

11.2 Future directions

Future research must now include applications development. There are several issues here. It will not be interesting to simply demonstrate the effects of parallelism in the new model, though it will be comforting to confirm that they exist in the new model as they do in existing parallel programming models. It will be interesting to place the new model in head to head experiments with the models criticized in this thesis. This is, in fact, a demanding issue to handle fairly. It is simply not simply enough to compare published results, or to run two implementations head to head; there are simply too many implementation variables. To make a fair scientific comparison we must be certain that we are comparing like with like. To meet this demand the models must use the same basic implementation technique and same code generator. To this end the current *Ease* compiler can be extended with message passing primitives and a Linda preprocessor, then some time is required to develop equivalent and fair applications. Only then, can a clear assessment of the performance advantages of the model be made and at this point in time the problem does not seem approachable.

Performance advantage does provide a direct measure of the model but it is, to some degree, the less interesting one. For many applications, depending on granularity and scale, the performance advantage will not be significant. It is far more important in future research to develop an understanding of the semiotic issues here. Does the model live up to its name?

[2]Rethinking of virtual memory and program security requires program validation and thus a closer relationship between compiler and operating system.

Does it really *Ease* the construction of parallel programs? Is it an effective engineering tool? The answers to these later questions lie outside of my research and entirely in the hands of the engineering community, but the semiotic issues, those that address the effect programming languages have on the behavior of engineers, are likely to be a rich source of new understanding. This understanding will enable us to design new and more effective models for programming high performance machines — and ultimately it is this overall effectiveness that is important.

Chapter 12

Conclusion

The goal of this thesis has been to develop a model and language to ease the development of parallel programs, driven by observations of deficiencies in existing models that distract the engineer. These were identified in the introduction to the thesis primarily as the distraction of data distribution in generalized message passing, and the distraction of loose contrivance in value associative matching of Linda.

The solution to such distractions is to adopt a data space abstraction that hides distribution complexity and expresses locality directly, allowing the engineer to focus on algorithm development and data locality. The solution presented, *Ease* contexts, is based on shared data structures with particular characteristics identified from experience to be those commonly used by application developers.

The goal has also been to identify a model that can make a reasonable claim to architecture independence; i.e., a model that provides not only program portability but also uniformity of performance across various machine architectures. Generalized message passing is not uniform since nontrivial message structures become copy operations when moved from a distributed system to a shared memory system. Linda is not uniform since tuple space optimizations have radically different effects and only implied data locality. However, it is worth noting that Linda itself is moving in the direction of *Ease* with the refinement of multiple tuple space concepts.

The solution presented here to these problems is to provide a mechanism that encapsulates exchange by reference for nontrivial data structures but whose semantics are valid for copies between disjoint address spaces; allowing efficient implementation on both shared and distributed memory architectures. This solution is provided by simple, symmetric operations upon Contexts that provide a strong expression of locality. This goal also limits the choice of data structure characteristics given to Contexts and operations upon them; e.g., giving a Context a set type and a "test for presence" operator would considerably complicate implementation (requiring search operations) and uniformity of the language. The implementation of such structures is best left to the engineer or to library support.

Secondary goals have been satisfied in the design of the full language. Occam was based on the sound mathematical principles of CSP but was deficient as an engineering tool in a number of ways over and above the problems associated with generalized message passing. In the design of *Ease* I have attempted to answer some of the frustrations early Occam programmers found; though I have maintained the same mathematical basis. Some of this is simply (though importantly) syntactic but there are other things too: the principal solutions are to the typing system, data structure handling, support for embedded systems, real time, and resources.

In concluding I address why these goals are more generally important and how the solutions presented here help. I answer the published concerns of Hennessy and Patterson [Hen90] that summarize the problems this thesis addresses.

In considering how difficult it is to program parallel machines Hennessy and Patterson put the problem well:

> "Why should it be so much harder to develop MIMD programs than sequential programs? One reason is that it is hard to write MIMD programs that achieve close to linear speed up as the number of processors dedicated to the task increases. ... think of the communication overhead for a task done by a committee .... While n people may have the potential to finish any task n times faster, the communication overhead for the group can prevent it from achieving this .... (Imagine the communication overhead going from 10 people to $1,000$ people to 1,000,000)."
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 575.

It remains to be seen if *Ease* can make a valid contribution to reducing this complexity — only application experience can really tell us. It is the pragmatic judgement of engineers that decide, not reasoning, no matter how well conceived. I have argued that *Ease* reduces complexity in parallel programming by removing the complication of message passing; individual processes focus on the data they share, not on the processes that share that data and thus locality occurs naturally.

I make a contention that has relevance here since it is an underlying assumption in the thesis: programming with parallel composition is simpler than programming with only sequential composition. Sequential programmers are preoccupied with interleaving the activities of a program and object oriented approaches have developed as a result of the need to address this problem. The evolution from the object oriented model to process models is not an unreasonable projection.

Hennessy and Patterson continue:

> "Another reason for the difficulty in writing parallel programs is how much the programmer must know about the hardware. On a uniprocessor, the high level

> language programmer writes his program ignoring the underlying machine organization — that's the job of the compiler. For a multiprocessor today, the programmer had better know the underlying hardware and organization if he is to write fast and scalable programs. This intimacy also makes portable parallel programs rare."
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 575.

This is an important observation, one that has driven a significant effort into the development of automatic translation of conventional programs into efficient forms for parallel machines. But the uniprocessor programmer has had it easy these past years. The nature of the beast has been tolerant of an unreasonable dependence on global structures and tolerance of side effects. New models of programming must evolve that express identifiable locality and are side effect free; *Ease* provides both these characteristics. In addition, the *Ease* model abstracts away from the underlying hardware memory architecture assisting the compiler in its efforts to achieve portable parallel programs with efficient implementation.

It must ultimately become the job of the compiler to provide the efficient placement of data and processes enabling fast scalable programs to be written without regard to the underlying hardware. This task can be significantly aided by the language and interaction model — though neither can provide a direct solution to it since it is dependent on the hardware architecture.

Again, Hennessy and Patterson:

> "The real issues for future machines are these: Do problems and algorithms with sufficient parallelism exist? And can people be trained or compilers be written to exploit such parallelism?"
>
> —J L Hennessy & D A Patterson
> Computer Architecture: A Quantitive Approach, page 579.

The first question is beyond the scope of this thesis, though observation of the natural world and existing applications suggests that even if there is some limit to parallelism in specific problems and algorithms the elaborate parallel composition of these remains useful.

However, it is the second part of this question that this thesis addresses directly. Training the engineering population requires significant investment, as far as possible engineers need familiar tools and, specifically, programmers prefer familiar notations. Occam met resistance as much for its idiosyncratic notation as it did for its introduction of parallelism. It is not parallelism that is complex, rather it has been the complexity of process interaction that has presented a hurdle. In *Ease* this complexity is significantly reduced and a familiar notational style maintained. Compilers are greatly assisted in their task by the side effect free nature of the language and the Context abstraction.

Hennessy and Patterson elaborate:

"Compilers of the future have two challenges on machines for the future:

1. Lay out of data to reduce memory hierarchy and communication overhead, and
2. Exploitation of parallelism."

—J L Hennessy & D A Patterson
Computer Architecture: A Quantitive Approach, page 581.

The first of these is addressed by increasing the degree of data locality in programs, the second is provided for by the nature of Context structures, side effect free expressions and functions (that permit a compiler the easy identification of fine grain parallelism) and explicit parallel constructions.

Finally, a type secure language with a strong mathematical basis that does not intrude upon the engineer is generally desirable, allowing for the application of formal methods when necessary and simplifying the task of program transformation.

Bibliography

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.,
Compilers: Principles, Techniques, and Tools,
Addison-Wesley 1986.

[Anc90] Ancourt, C.,
Code Generation for Data Movements in a Hierarchical Memory Machine ,
in [FeIr90].

[AnTzGr88] Anderson, D.P., Tzou, S., Graham, G.S.,
The DASH Virtual Memory System ,
EECS, University of California, CSD 88/461, 1988.

[And91] Andrews, Gregory R.
Concurrent Programming,
Benjamin/Cummings, 1991.

[Bal89] Bal, Henri E.
The Shared Data-Object Model,
PhD Thesis, VRIJE UNIVERSITEIT TE AMSTERDAM, 1989.

[BaMé92] Banâtre, J.P. & Le Métayer, D. (Eds.),
Research Directions in High–Level Parallel Programming Languages,
Springer–Verlag, 1992.

[BaGoJoKa88] Barrett, G., Goldsmith, M., Jones, G., Kay, A.
The Meaning and Implementation of PRI ALT in Occam ,
IOS, Proc. 9th Occam User Group, 1988.

[P4] Boyle et.al,
Portable Programs for Parallel Processors, Holt, Rinehart and Winston, 1987.

[Han72] Brinch Hansen, P.
A comparison of two synchronizing concepts, Acta Informatica 1, 190–199, 1972.

[Han73a] Brinch Hansen, P.
Operating System Principles, Prentice Hall 1973.

[Han73b] Brinch Hansen, P.
Concurrent Programming Concepts, ACM Computing Surveys 5, 4, 223–245.

[Han75] Brinch Hansen, P.
The Programming Language Concurrent Pascal,
IEEE Trans. on Software Engineering. SE-1, 2 (June), 199-206.

[Han77] Brinch Hansen, P.
The Architecure of Concurrent Programs, Prentice Hall 1977.

[Han87] Brinch Hansen, P.
Joyce–a programming language for distributed systems, Software Practice and Experience 11, 4, 325–361

[Han89] Brinch Hansen, P.
The Joyce Language Report,
A Mulitprocessor Implementation of Joyce, Software Practice and Experience 19, 6, 553–592.

[Car87] Carriero, Nick.J.
Implementation of tuple space,
Yale University Ph.D Thesis, Department of Compuer Science, RR–567, 1987.

[Car89] Carriero, N.J. & Gelernter, David.
How to write parallel programs: a guide to the perplexed,
ACM Computing Surveys, Vol 21, No.3, Sept 1989.

[Con63] Conway, M.,
A Multiprocessor System Design,
Proceedings of the AFIPS Fall Joint Computer Conference, 1963.

[DaHe81] Davis, Philip J. & Hersh, Reuben.
The Mathematical Experience,
Houghton Mifflin, 1981.

[Dij65] Dijkstra, E.W.,
Solution of a Problem in Concurrent Programming Control,
Communications of the ACM, Vol. 8, No.9, page 569, September 1965.

[Dromey] Dromey, R.G.
How to Solve it by Computer,
Prentice-Hall 1982.

[Enc88] ENCORE,
Encore Parallel Threads Manual,
Encore Computer Corporation, 1988.

[Zen90] Ericsson Zenith, Steven.,
Programming with Ease: the semiotic definition of the language,
Yale Research Report 809, July 1990.

[Zen91] Ericsson Zenith, Steven,
Linda Coordination Language; Subsystem Kernel Architecture (on transputers),
in [Per92].

[Zen92] Ericsson Zenith, Steven.,
A Rationale for Programming with Ease,
in [BaMé92].

[Euclid] Eucild, trans. by T.L.Heath,
The Thirteen Books of Euclid's Elements,
Cambridge University Press, 1908.

[FeIr90] Feautrier, P., Irigoin, F., (Eds),
International Workshop on Compilers for Parallel Computers ,
Ecole des Mines de Paris & MASI.

[Gel85] Gelernter, David.
Generative Communication in Linda,
ACM Transactions, 1985.

[Gel89] Gelernter, David,
Multiple Tuple Spaces in Linda ,
Proc. of PARLE 89.

[Gell etal.] Gellert et al.,
The VNR Concise Encyclopedia of Mathematics,
Second Edition, Van Nostrand Reinhold, 1989.

[Gol88] Goldsmith, Michael,
The Oxford Occam Transformation System ,
Oxford University Computing Lab., 1988.

[Hen90] Hennessy, John L. & Patterson, David A.
Computer Architecture A Quantitative Approach,
Morgan Kaufmann Publishers, 1990.

[Hoa85] Hoare, C.A.R.
Communicating Sequential Processes,
Prentice-Hall 1985.

[Hoa74] Hoare, C.A.R.,
Monitors: An Operating System Structuring Concept,
Communications of the ACM, Vol.18, No.2, page 95, February 1975.

[Hoa72] Hoare, C.A.R.,
Towards a Theory of Parallel Programming,
"Operating Systems Techniques", (Hoare & Perrott, Eds.), Academic Press, 1972.

[HoWe89] Hopkins, T.R. & Welch, P.H.,
Transputer Data–Flow Solution for Systems of Linear Equations ,
Computing Lab., University of Kent at Canterbury RR-68, 1989.

[Hup90] Hupfer, S.,
Melinda: Linda with Multiple Tuple Spaces,
Yale University, Department of Computer Science, RR–766, February 1990.

[IEEE754] ANSI/IEEE 754-1985,
IEEE Standard for Floating Point Arithmetic,
IEEE, 1985.

[INM90] INMOS Limited.
The Transputer Data Book, INMOS Limited, 1990.

[INM88] INMOS Ltd (Ericsson Zenith, S. & May, M.D.)
Occam 2 Reference Manual,
Prentice-Hall 1988.

[INM91] INMOS Limited,
Special T9000 ,
La lettre du Transputer et des Calculateurs Distribués, Avril 1991.

[INTEL92] INTEL Corp.,
Paragon XP/S Technical Summary, INTEL Corp. 1992.

[Jag92] Jagannathan, Suresh.,
Expressing Fine–Grained Parallelism Using Concurrent Data Structures,
in [BaMé92].

[JoGo88] Jones, G. & Goldsmith, M.
Programming in Occam 2,
Prentice-Hall 1988.

[KoMe90] Koelbel, C. & Mehrotra, P.,
Compiling Global Name–Space Programs for Distributed Execution ,
ICASE, NASA Langley Research Center, RR 90–70, 1990.

[Lei89] Leichter, J.,
Shared Tuple Memories, Shared Memories, Buses and LAN's — Linda Implementations Across the Spectrum of Connectivity,
Yale University Ph.D Thesis, RR–714, July 1989.

[Lel90] Leler, W.,
Linda Meets UNIX,
IEEE Computer, Vol.23, No.2, pages 52–76, September 1990.

[Li89] Li, Kai and Hudak, Paul,
Memory Coherence in Shared Virtual Memory Systems,
ACM Transactions on Computer Systems 7, 4, 321–359. 1989.

[Mac91] MacKenzie, Tim.
Communication in Ease,
Final Honours Report, CS400 Monash University, Australia, 1991.

[MeVR90] Mehrotra, P. & Van Rosendale, J.,
Programming Distributed Memory Architectures Using KALI ,
ICASE, NASA Langley, RN 90–69 1990.

[May83] May, M.D.
OCCAM, SIGPLAN Notices 18, 4 (April), 69-79.

[May88] May, M.D.
The transputer implementation of Occam, INMOS Limited, 1985.

[Mil78] Milner, A.J.
A theory of type polymorphism in programming,
Journal of Computer and System Sciences. Vol. 17. 1978.

[Mil89] Milner, Robin
Communication and Concurrency,
Prentice Hall, 1989.

[Nau63] Naur, Peter (Ed),
Revised Report on the Algorithmic Language ALGOL 60 ,
CACM, Vol 6, 1963.

[Par88] ParaSoft Corporation,
An Overview of the Express System ,
ParaSoft Corporation, 1988.

[Per92] Perrott, R.H.,
Software for Parallel Computers,
Chapman & Hall, 1992.

[PeSi85] Peterson, J.L. & Silberschatz, A.
Operating Systems Concepts,
Addison-Wesley 1985.

[Pou90] Pountain, D.,
Virtual Channels: The Next Generation of Transputers,
BYTE, Vol.3.No.12, April 1990.

[Prz90] Przybylski, Steven A.,
Cache and Memory Hierarchy Design,
Morgan Kaufmann Publishers, Inc. 1990.

[RaKh88/38] Ramachandran, U. & Khalidi M.Y.A.,
Programming with Distributed Shared Memory ,
School of Info. and Computer Science, Georgia Inst. of Tech. GIT–ICS–88/38 1988.

[RaKh88/50] Ramachandran, U. & Khalidi M.Y.A.,
An Implementation of Distributed Shared Memory ,
School of Info. and Computer Science, Georgia Inst. of Tech. GIT–ICS–88/50 1988.

[RaTe et.al.89] Rashid, R., Tevanian, A., et.al.,
Machine–Independent Virtual Memory Management for Page Uniprocessor and Multiprocessor Architectures ,
Department of Computer Science, Carnegie Mellon University, 1989.

[Red91] Redman, John.
An Implementation of the Ease Programming Language, Final Honours Report, The University of Western Australia, 1991.

[Rey91] Reynolds, John. C.
The Craft of Programming, Prentice-Hall, 1981.

[RoCu90] Roman & Cunningham,
Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency,
IEEE Trans. on Software Engineering, Dec. 1990.

[Ros86a] Roscoe, A.W. & Dathi, N.
The Pursuit of Deadlock Freedom,
Oxford University Monograph, PRG-57. 1986.

[Ros86b] Roscoe,A.W. & Hoare,C.A.R.
The Laws of Occam Programming,
Oxford University Monograph, PRG-53. 1986.

[Sal90] Saltz, Joel,
Run Time Parallelization and Scheduling of Loops ,
in [FeIr90].

[SaBeWu90] Saltz, J. Berryman, H., Wu, J.
Multiprocessors and Runtime Compilation ,
in [FeIr90].

[Schm86] Schmidt, David. A.
Denotational Semantics, Allyn and Bacon, 1986.

[Ski90] Skillicorn, D.B.,
Architecture–Independent Parallel Computation ,
Dept. of Computing and Info. Science, Queen's University, Canada 1990.

[Ski90] Skillicorn, D.B.,
Models for Practical Parallel Computation ,
Internatinal Journal of Parallel Programming, Vol. 20, No. 2, 1991.

[PVM] Sunderam, V.S.,
PVM: A Framework for Parallel Distributed Computing,
Department of Math and Computer Science, Emory University, Atlanta.

[TaMu81] Tanenbaum, A.S. & Mullender, S.J.
An overview of the Amoeba Distributed Operating System,
Operating Systems Review, pp.51-64, July 1981.

[TMC92] Thinking Machines Corp.
CM5 Technical Summary, TMC 1992.

[Wil91] Wilson, Greg (Ed),
Linda–Like Systems and their Implementation ,
EPCC, Edinburgh University 1991.

[Witt69] Wittgenstein, L.
On Certainty, Harper and Row, 1969.

Appendix A

Programs in Ease

A.1 Conway's game of life.

The following example is a reworking of the classic Conway game of life. This version is based on the Occam version found in Jones and Goldsmith "Programming in Occam 2"[JoGo88]. A comparison (though space does not permit the repetition of the Occam version here) is worthwhile since it illustrates just how badly Occam programs are affected by data distribution issues though in fairness it must be observed that the Occam version is a systolic program. The *Ease* version differs in a number of significant ways. The primary difference is in how the main data structure, is conceived.

The game of life is a classic and simple simulation of an evolutionary system based upon a rule set. The rule set in this case applies to a 2D matrix of infinite size where each "cell" of the matrix has one of two states called "alive" and "dead". The cell evolves through a sequence of generations with its state determined by its own state and that of its eight neighbours. The rules are

1. if the cell is alive and less than two neighbours are alive then the cell becomes dead,
2. if the cell is alive and two or three neighbours are alive then the cell remains alive,
3. if the cell is alive and more than three neighbours are alive then the cell becomes dead,
4. if the cell is dead and three neighbours are alive the cell becomes alive.

These rules respectively represent cell states of starvation, stability, overcrowding, and birth.

```
let array'width  = 256           /* cells across the board */
let array'height = 256           /* cells down the board   */

let radius     = 1               /* 'sphere of influence'  */
let diameter   = 2 * radius + 1
let neighbours = diameter * diameter - 1

let alive = true
let dead  = not alive

type BOARD context [array'width][array'height]single BOOL
 let board := BOARD
let pause := INT'SINGLE'CONTEXT

procedure next'state (x, y, nx, ny)

    let living := 0
    let neighbours'state := _ -> BOOL
    :
    {
      { i for neighbours /* count the number living */
         board[nx[i]][ny[i]] ? neighbours'state
         test neighbours'state = alive : living ++
              neighbours'state = dead  : skip ;
      }
      test           /* death from isolation */
           living < 2 : board[x][y] ! dead
                     /* cell is stable */
           living = 2 : skip
                     /* stable if alive, birth if dead */
           living = 3 : board[x][y] ! alive
                     /* death from overcrowding */
           living > 3 : board[x][y] ! dead ;
}
```

```
procedure cell(x, y)

    let left   = (x - 1 + array'width)  % array'width
    let right  = (x + 1)                % array'width
    let up     = (y + 1)                % array'height
    let down   = (y - 1 + array'height) % array'height

    let nx = [right, x, left, left, left, x, right, right]
    let ny = [down, down, down, y, up, up, up, y]

    let running := true
    let any := _ -> INT
    :
    {
      board[x][y] ! dead
      while running
      {
        next'state(x, y, nx, ny)
        screen[x][y] := board[x][y]
        pause ? any
      }
}
```

```
procedure edit'board ()

    /* start cursor in mid screen */
    let x       := array'width  / 2
    let y       := array'height / 2

    let editing := true
    let key     := _ -> char

    function min (a, b) = if a <= b : a else b
    function max (a, b) = if a >= b : a else b

    :
    while editing
    {
      screen ! (move, x, y)

      keyboard ? key

      select key
       /* change state */
      ' '
        {
         board[x][y] ? state
         board[x][y] ! not state
      }
       /* move up if possible    */
      'U', 'u'
         y := max(y-1, 0)
       /* move down if possible  */
      'D', 'd'
         y := min(y+1, array'height-1)
       /* move right if possible */
      'R', 'r'
         x := min(x+1, array'width-1)
       /* move left if possible  */
      'L', 'l'
         x := max(x-1, 0)
       /* quit */
      'Q', 'q'
         editing := false
       /* ignore other input */
      else skip ;
}
```

```
procedure control()

    let key     := _ -> char
    let running := true
    let any     := 0
    :
    while running
    {
      keyboard ? key

      pause ! any

      select key
      'E', 'e'
        {pause ?* any
         edit'board()
         pause ! any
        }
      'Q', 'q'
         running := false
   else skip ;
}

: /* body of program */

{ screen ! clear
  edit'board()

  || control()
  || x for array'width
    || y for array'height
      cell(x, y);;
}
```

The rules of the game of life determine the state of the whole matrix for each generation since the evolution is synchronized by barriers of cooperation and deterministic global structures evolve in the form of identifiable patterns, illustrating how a few simple local rules can control the evolution of global structures. Several patterns are well known, passing through cycles of growth and decay, or even endless growth. In the version of the simulation presented here an non–synchronized evolution can be expressed by replacing the cooperation with subordination, so that the main program becomes

```
{ screen ! clear
  edit'board()

  // x for array'width
    // y for array'height
      cell(x, y);;

  control()
}
```

The implementation represents the matrix as a singleton context of boolean components.

The loop that reads the neighbouring states is described in sequence. All sharing of data must be expressed by context. The Occam version of this program[JoGo88] uses a construct

```
PAR d = 0 FOR neighbours
  link[nx[d]][ny[d]][d] ? state.of.neighbour[d]
```

to express parallel input (the extra subscript is required to identify the input) into an array of booleans. How can we express parallel input in *Ease*? We can rewrite `next'state` to allow parallel input by using a local context to describe the shared data, thus it becomes

```
procedure next'state (x, y, nx, ny)
    /* with parallel input */
    let count  := INT'SINGLE
    let living := _ -> INT
    :
    { count ! 0
      || i for neighbours /* count the number living */
         let neighbours'state := _ -> BOOL :
         { board[nx[i]][ny[i]] ? neighbours'state
           test neighbours'state = alive : inc(count)
                neighbours'state = dead  : skip ;
         }
      count ?* living
      test           /* death from isolation */
           living < 2 : board[x][y] ! dead
                     /* cell is stable */
           living = 2 : skip
                     /* stable if alive, birth if dead */
           living = 3 : board[x][y] ! alive
                     /* death from overcrowding */
           living > 3 : board[x][y] ! dead ;
}
```

where `inc(c)` has the behavior

```
procedure inc(c)
  let count := _ -> INT :
  { c ?* count
    count ++
    c ! count }.
```

An implementation that is able to use read–modify–write instructions will use such library routines to allow this behavior to be implemented as read–modify–write.

A.2 Stream exchange sort

In an application not all algorithms need to be parallel. The elegance of programming with a process model over conventional programming comes from the useful composition of processes.

This example illustrates a single, sequential process, acting upon a Stream context. The program is an implementation of simple sort by exchange, but this example servers as an illustration of resource use and the passing of contexts as first class data structures.

```
type STRING'STREAM context stream string
type FILE'STORE    context STRING'STREAM reply STRING'STREAM

let file'system := FILE'STORE
```

```
procedure exchange'sort (STRING'STREAM strings)
    /* sort a non-empty string context */
    let s := _ -> STRING
    let t := _ -> STRING
    let swaps := true
    let end'marker = "A unique string"
    :
{ while swaps
    { strings ! end'marker

      strings ?* s
      strings ?* t

      swaps := false

      enum s', t'
      let last := t'
      :
      while s <> end'marker and t <> end'marker
          test s < t : { strings !* s
                         strings ?* s
                         test last = s' : swaps := true
                         else             last  := s';
                       }
               t < s : { strings !* t
                         strings ?* t
                         test last = t' : swaps := true
                              last = s' : last  := t';
                       }
               s = t : test last = t' : { strings !* s
                                          strings ?* s
                                        }
                            last = s' : { strings !* t
                                          strings ?* t
                                        } ;
          ;
      /* leave end marker *after* last "sorted" item */
      test s = end'marker : { strings !* s
                              strings !* t }
           t = end'marker : { strings !* t
                              strings !* s }
      ;
    }
  strings ?* s
}
```

```
:
|| users()
||
  let file := STRING'STREAM   :
  while true
    resource file'system ?* file
       exchange'sort (file)
    !* file
;
```

The exchange sort algorithm is more popularly called "bubble sort". A complete description of the algorithm, its weakness and implementation, can be found in Dromey [Dromey].

A.3 A shared stack

I mentioned in the discussion on future directions the question of LIFO characteristics for shared data; i.e., stacks do not have useful characteristics for data exchange or data sharing since use of stacks implies a lack of ordering interest in such circumstances and Bag meets that requirement. None–the–less, stacks are important for many algorithms and can be implemented conventionally or as resources.

We begin by defining the types involved

```
type VALUE is ... /* type of values stacked */
type STATE is BOOL

type STACK'RESOURCE context
   /* push */ VALUE reply STATE  |
   /* pop  */ STATE reply VALUE
```

STACK'RESOURCE is an associative type consisting of a value and state. We define a stack resource as

```
let stack := STACK'RESOURCE
```

We must now defined Push and Pop operation on the stack, but since this is a shared resource we define a few extra desirable features. Firstly, the stack should

- refuse Push if there is no space available, and
- refuse Pop if there are no values present.

In the implementation that follows I provide the Pop operation with the ability to block Push. This will permit a process to Pop several items from the stack while preventing other processes from Pushing values onto the stack.

```
let ok = true

let state := ok
let stack'store   := _ -> [max'size]VALUE
let stack'pointer := 0
let value         := VALUE
let running       := true
:
while running
choice
  /* push */
  state:
  resource stack ?* value
       test stack'pointer < max'size-1
            { stack'pointer ++
              stack'store [stack'pointer] := value
            }
       else { state := NOT ok
              stack'pointer ++
              stack'store [stack'pointer] := value
            } ;
       !* state
  /* pop */
  stack'pointer <> 0:
  resource stack ?* state
       test stack'pointer > 1
            { value := stack'store [stack'pointer]
              stack'pointer --
            }
       else { value := stack'store [stack'pointer]
              stack'pointer --
              state := ok
            } ;
       !* value
  /* shut down command */
  quit ? running : skip
;
```

The resource process is constructed as a loop composed of a choice with three components: a Push resource, a Pop resource, and a quit instruction.

The Push resource is guarded by a boolean `state`, initially true and set to false if the current Push operation fills the stack. This value is returned to the user process and indicates that the resource may not accept subsequent values.

The Pop resource is guarded by a boolean expression that allows a Pop to occur only if a

value exists on the stack. The Pop operation is allowed to specify a state value that may block subsequent Pushes and permits a process to remove several values from a stack without new values being Pushed. This feature is not useful unless a certain discipline is observed between the processes wishing to use it, since several Popping processes will interfere with each other (it is useful where there are many Pushers and a single Popper). Popping processes are not permitted to block out Push operations when the stack is empty.

A user process might call the stack by

```
procedure push (v)
  let stack'condition := _ -> STATE :
  { stack !* v ?* stack'condition
    etc...
  }
procedure pop (v)
  stack ! true ?* v
```

The stack type associativity will select the correct resource. Finally, a controller can shut down the stack by

```
quit ! false.
```

A.4 The sieve of Eratosthenes

This example is an implementation of the well known "Sieve of Eratosthenes". The object of the program is to sum all the primes from the first prime 2 to LIMIT.

I begin by defining a context type stream called "ordered". A stream of integers. Contexts of this type will connect processes in a pipeline.

```
type ORDERED context stream INT
```

In addition a singleton type is required to hold the final summation.

```
type ONE context single INT
```

The algorithm is well known. Briefly, a stream of candidate prime numbers is passed through a pipeline of filters. The first number in the stream is a new prime. Each filter looks for numbers which are not multiples of the prime the filter represents – such numbers are potentially prime and are passed on to the next filter in the pipeline. The implementation for such a filter is

```
procedure filter (pipe, sum, sigma)
  let next := ordered
  let prime, candidate := _ -> INT
  :
{
  pipe ?* prime
  pipe ?* candidate
  test (prime*prime) < LIMIT
    { let new'sum = sum + prime :
      // filter (next, new'sum, sigma);
      while candidate <> NULL
        test (candidate % prime)=0 : pipe ?* candidate
        else {next !* candidate
              pipe ?* candidate};
      next ! NULL
      }
  else let result := sum + prime :
    { while candidate <> NULL
        { result := result + candidate
          pipe   ?* candidate
          }
      sigma ! result
    };
}
```

A filter process begins by allocating a context `next`. This will be used to pass data to the next stage in the pipeline. A new pipeline stage, a copy of the filter process itself, is created as required.

Next a procedure which writes into the head of the pipe a sequence of odd numbers.

```
procedure source(pipe)
{ {i for LIMIT/2-1 from 3 by 2 : pipe ! i}
  pipe ! NULL
}
```

Two is considered a "found" prime. The final program can be expressed by

```
let sum  := _ -> INT
let pipe := ordered
:
{
  //filter(pipe, 2, sigma) //source(pipe) ;
  sigma ? sum
  printf("Sum of primes: %d.\n", sum) }
```

Here there are three processes. The two subordinate processes, the first stage filter process and the source process writing odd numbers. The creating process waits and presents the final result.

A.5 Common cores

Operations on matrices are among the most compute intensive components of High Performance Computing applications. The expression of large shared matrices is straight forward, simply

```
type matrix context single [N][N]FLOAT64

let a, b, c := matrix
```

defines an N by N matrix. The following paragraphs illustrate how the core of a matrix multiply and Gauss/Jordan computation on such a matrix can be simply expressed.

A.5.1 Matrix multiply

The common sequential core of a matrix multiplication is

```
{ i for N
  { j for N
    { k for N
        c[i][j] := c[i][j] + a[i][k] * b[k][j]
}}}
```

The parallel core can be expressed as

```
// i for N
  // j for N
     { k for N
       c[i][j] := c[i][j] + a[i][k] * b[k][j]
};;
```

and this is algebraically equivalent to

```
// i for N
  // j for N
   { k for N
     let tmp0, tmp1, tmp2, tmp3 := _ -> FLOAT64:
     {
     c[i][j] ? tmp0
     a[i][k] ? tmp1
     b[k][j] ? tmp2

     tmp3 := tmp0 + tmp1 * tmp2

     c[i][j] ! tmp3
    }
};;
```

the order of the read operations preceding the computation is unimportant since read operations are side effect free; we cannot write these inputs as being in parallel since parallel processes may not write to free variables, however, an implementation can derive the disjointness of the temporaries from the abbreviated construction and will, in fact, implement parallel inputs where advantageous to do so.

A.5.2 Gauss-Jordan

A similar approach applies to the common Gauss/Jordan core. The sequential core is

```
{ i for N
  { j for N
    { k for N
      test i <> j
        a[j][k] := a[j][k]-(a[j][i]*a[i][k])/a[i][i]
      else skip ;
}}}
```

Similarly the parallel core can be expressed

```
{ i for N
  || j for N
    || k for N+1
    test i <> j
      a[j][k] := a[j][k]-(a[j][i]*a[i][k])/a[i][i]
    else skip ;
;;}.
```

A.6 Farming: master/worker

The farming paradigm is well established now as a method of work scheduling among some number of processes. A Farmer, often called the "master" has some number of similar tasks to perform makes them available to a force of workers able to perform them. Generally this paradigm is suited to applications with the form

```
{ i for n
    answer[i] := worker(task[i])
}
```

where the controlling loop represents the Farmer and `worker`, predictably, represents the worker.

This can be written with *Ease* by first establishing a context to contain a pool of tasks. For the purposes of illustration we shall implement a Farmer worker process that determines the sum and the largest element of a matrix. A task is simply a row of the matrix.

```
type TASK is [N]INT
```

Now we define the Bag context type to hold the tasks as

```
type TASK'BAG context TASK
```

A further context is required to collect the results.

```
type SUM     is INT
type LARGEST is INT

type ANSWERS context
            SUM   |
            LARGEST
```

The results context is an associative context that collects all the answers.

Again for the purposes of illustration, I show the Farmer process acting as a resource receiving a matrix which it then places as rows along with the appointed tasks into the task bag.

```
procedure farmer (environment)

    let matrix  := _ -> [N][N]INT
    let sum     := _ -> SUM
    let total   := 0 -> SUM
    let big     := _ -> LARGEST
    let biggest := _ -> LARGEST

    let work    := TASK'BAG
    let results := ANSWERS
    let quit    := BOOL'SINGLE
    :
    ||
     resource environment ?* matrix
        let m = matrix :
        || i for N  /* output rows */
            work ! m[i]
        ||{{ i for N  /* read in sums and accumulate */
               answers ?* sum
               total := total + sum
               }
           print ! "Matrix sum = %\n"(total)
          }
        ||{answers ?* biggest
           { i for N-1 /* find largest among the large */
              answers ?* big
              biggest := if big > biggest : big else biggest
           }
           print ! "Matrix biggest = %\n"(biggest)
           quit ! false
          } ;
     !* matrix

    || i for a'good'number'of'workers
        worker()
    ;
```

The Farmer itself consists of a number of parallel processes, the two primary processes are the resource and the creation of workers. The resource consists of

- a process to output the rows of the matrix in parallel,
- a process to accumulate the summation by the workers, and
- a process to select the largest of the large values found by the workers.

The constant definition `let m = matrix` allows the matrix to be acted upon in parallel as a read only value; recall that free variables may not appear in parallel processes, however, it is possible to provide read only access in cooperations to variables in this way (though the same constant definition preceding a subordination will, of course, result in a copy of the matrix being made). When the work is complete the matrix is returned to the sender.

The Farmer creates some number of processes determined by the environment constant `a'good'number'of'workers`.

Implementation note: it should be obvious in the above that a compiler does not have to work very hard to see that the output of the matrix in the write `work ! m[i]` can be implemented under some circumstances without copying the matrix since `m` is a constant. If the compiler now finds (as it will) that these components are only read by all concurrent processes in the scope a simple optimization can be made and the matrix itself need never be copied. I mention this to illustrate the simplicity in detecting such optimizations in an environment free of side effects and to point out to implementors that optimization should still be pursued.

The worker process simply acts as

```
procedure worker ()

    let row := _ -> [N]INT
    let sum := 0 -> SUM
    let big := _ -> LARGEST

    let going := true
    :
    while going
    choice
      work ?* row
        let row = row -> [N]SUM :
        { { i for N : sum := sum + row[i]}
          results ! sum
          big := row[0]
          { i for N-1 from 1
              big := if row[i] > big : row[i] else big }
          results ! big
        }
      quite ? going : skip
    ;
```

The worker simply takes up a row and places the sum and largest values in the answer context.

For the optimization mentioned in the previous implementation note the compiler must simply determine that the base type of `row` is the same as `[N]SUM`.

Appendix B

A YACC/Bison grammar

```
%{ /* Yacc/Bison grammar for
   Ease - a language for programming concurrent systems.

   Version Beta.09
   Copyright (C) 1991, 1992 Steven Ericsson Zenith.
   Copyright (C) 1992 Science Frontiers, International.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 1, or (at your option)
   any later version.

   This program is distributed in the hope that it will be
   useful, but WITHOUT ANY WARRANTY; without even the implied
   warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
   PURPOSE. See the GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
   */
%}
/* Terminals */
%union {
  struct symbol *sym;
  struct list   *list;
  char          *value;
}
```

```
/* name */
%token NAME

/* keywords and key symbols */
%token BECOMES READ WRITE GET PUT         /* := ? ! ?* !* */
%token SEQUENCE TEST SELECT RESOURCE CHOICE
%token ELSE AFTER ON STOP
%token COOPERATE SUBORDINATE              /* || // */
%token WHILE DO UNTIL
%token FOR FROM BY
%token INCREMENT DECREMENT                /* ++ -- */
%token STOP SKIP

/* expression symbols */
%token MUL DIV REM EXP                    /* * / % ^ */
%token SUB ADD                                 /* - + */
%token EQ NE GT LT GE LE                  /* = <> > < >= <= */
%token AND OR XOR
%token BITAND BITOR BITXOR                /* /\ \/ >< */
%token RSHIFT LSHIFT                      /* >> << */
%token COMPLEMENT NOT                     /* ~ */
%token TEXT CHARACTER NUMBER
%token TRUE FALSE
%token IF SIZE OF
%token OPENTO CLOSETO TO                  /* (.. ..) .. */
%token COERCE ASSERT CAST                 /* -> => >| */

/* declarators and types */
%token LET VAL RENAME TYPE ENUM IS AT
%token COUNTED                            /* :: */
%token CONTEXT STREAM SINGLE LO HI REPLY
%token PROCEDURE FUNCTION WHERE
%token BOOL
%token INT INT8 INT16 INT32 INT64 INT128
%token FLOAT32 FLOAT64 FLOAT128 CHAR STRING
%token UNSIGNED TRUNC ROUND

/* modules */
%token MODULE END SHOW HIDE USE

/* linebreak points */
%token NEWLINE
```

```
%start program

%%
/* processes */
program: . process .
        ;

process:  action
        | construction
        | instance
        | specification_block ':' ... process
        | error_handler
        ;

error_handler:  /* Handles grammatical errors:
                   disgards process or finds next semicolon */
          error process
        | error ';'
        ;

action:   assignment
        | interaction
        | STOP
        | SKIP
        ;

construction:
          sequence
        | test
        | selection
        | combination
        | choice
        | cooperation
        | subordination
        | repetition
        ;

assignment:
          element BECOMES expression
        | element INCREMENT
        | element DECREMENT
        ;
```

```
interaction:
          input
        | output
        ;

input:    read
        | get
        ;

output:   write
        | put
        ;

read:     element READ element_list
        ;

write:    element WRITE expression_list
        ;

get:      element GET reference_list
        ;

put:      element PUT reference_list
        ;

reference: NAME
        ;

reference_list:
          reference
        | reference ',' . reference_list
        ;

element_list:
          element
        | element ',' . element_list
        ;

sequence: SEQUENCE . '}'
        | SEQUENCE . subsequence '}'
        | SEQUENCE . replicator .. subsequence '}'
        | SEQUENCE . ON STOP . process .. subsequence '}'
        ;
```

```
subsequence:
          process .
        | process ... subsequence
        ;

test:     TEST . default
        | TEST . conditional_body default
        | TEST . replicator .. conditional . default
        ;

default:  ';'
        | ELSE . process . ';'
        ;

conditional_body:
          conditional .
        | conditional ... conditional_body
        ;

conditional:
          expression .. process
        | test
        ;

selection:
          SELECT selector .. option_body default
        ;

selector: expression
        ;

option_body:
          option .
        | option ... option_body
        ;

option:   match_list .. process
        ;

match_list:
          expression
        | expression ',' . match_list
```

```
        ;

combination:
          call
        | reply
        ;

call:     write GET reference
        | put   GET reference
        ;

reply:    RESOURCE . get . WRITE expression
        | RESOURCE . get .. process . WRITE expression
        | RESOURCE . get .. process . PUT  reference
        ;

choice:   CHOICE . default_choice
        | CHOICE . alternative_body default_choice
        | CHOICE . replicator .. alternative . default_choice
        ;

default_choice:
          ';'
        | ELSE . process . ';'
        | ELSE . AFTER time .. process . ';'
        ;

alternative_body:
          alternative .
        | alternative ... alternative_body
        ;

alternative:
          input .. process
        | boolean .. input .. process
        | reply
        | boolean .. reply
        | '|' . process
        | boolean '|' . process
        | choice
        ;

time:     float
```

```
        ;

cooperation:
          cooperate . ';'
        | cooperate . cooperation
        ;

cooperate:
          COOPERATE . process
        | COOPERATE . replicator .. process
        ;

subordination:
          subordinate . ';'
        | subordinate . subordination
        ;

subordinate:
          SUBORDINATE . process
        | SUBORDINATE . replicator .. process
          /* Process placement */
        | SUBORDINATE . ON node .. process
        | SUBORDINATE . ON replicator .. process
        ;

repetition:
          WHILE boolean .. process
        | DO . process . UNTIL boolean
        ;

replicator:
          replicant FOR . count
        | replicant FOR . count FROM . base
        | replicant FOR . count FROM . base BY step
        ;

replicant:
          NAME
        ;

instance:
          procid '(' argument_list ')'
        ;
```

```
/* specifications */
specification_block:
          specification
        | specification specification_block
        ;

specification:
          declaration .
        | definition
        ;

definition:
          procedure
        | function
        | typedef   ...
        | module    ...
        ;

declaration:
          LET specifier
        | ENUM name_list
        | ENUM FROM base . name_list
        | USE modulid
        ;

specifier:
          NAME EQ expression
        | name_list BECOMES allocation
        | NAME RENAME element
        ;

allocation:
          expression
        | expression ON node
        ;

node:     integer
        | integer AT integer
        ;

name_list:
          NAME
        | NAME ',' . name_list
```

```
        ;

procedure:
          PROCEDURE procid '(' pformal_list ')' . process .
        ;

procid:   NAME
        ;

pformal_list:
          /* EMPTY */
        | pformal
        | pformal ',' . pformal_list
        ;

pformal:  NAME
        | compliant NAME
        | VAL NAME
        | VAL compliant NAME
        ;

function:
          FUNCTION funcid '(' fformal_list ')' . function_expression
        ;

funcid:   NAME
        ;

function_expression:
          EQ . expression .
        | EQ . expression . WHERE . process .
        ;

fformal_list:
          /* EMPTY */
        | fformal
        | fformal ',' . fformal_list
        ;

fformal:  NAME
        | compliant NAME
        ;
```

```
compliant:
          type
        | '[' ']'type
        ;

/* modules */
module:   MODULE modulid ... module_body END MODULE
        ;

modulid:  NAME
        ;

module_body:
          show hide
        | hide show
        ;

show:     SHOW . specification_block
        ;

hide:     HIDE . specification_block
        ;

/* types */
type:     primitive_type
        | tuple_type
        | array_type
        | integer_type '[' limit ']' COUNTED type
        | typeid
        | TYPE OF expression
        ;

typedef:  TYPE typeid IS type
        | TYPE typeid CONTEXT . bag_list
        ;

typeid:  NAME
        ;

bag_list: bag
        | bag_list '|' . bag
        ;
```

```
bag:      bag_type
        | priority .. bag_type
        ;

bag_type:
          unordered
        | singleton
        | stream
        | exchange
        ;

unordered:
          type        ;

singleton:
          SINGLE type
        | '[' integer ']' . singleton
        | '[' integer ']' . stream
        ;

stream:   STREAM type
        ;

exchange: type REPLY type
        ;

priority:
          LO
        | HI
        | LO integer
        | HI integer
        ;

primitive_type:
          BOOL
        | integer_type
        | UNSIGNED integer_type
        | float_type
        ;

integer_type:
          INT
        | CHAR
```

```
        | INT8
        | INT16
        | INT32
        | INT64
        | INT128
        ;

float_type:
          FLOAT32
        | FLOAT64
        | FLOAT128
        ;

array_type:
          '[' integer ']' . type
        | STRING
        ;

tuple_type:
          '(' type ',' . type_list ')'
        ;

type_list:
          type
        | type ',' . type_list
        ;

/* elements and expressions */
literal:  CHARACTER
        | string
        | NUMBER
        | TRUE
        | FALSE
        | '_'
        ;

string:   TEXT
        | TEXT '\\' ... string
        ;


tuple:    '(' expression ',' . expression_list ')'
        ;
```

```
table:    '[' expression_list ']'
        ;

element:  NAME
        | tuple
        | table
        | element '[' . subscript ']'
        | element '[' . base CLOSETO
        | element OPENTO limit ']'
        | element '[' . base FOR . count ']'
        | element '[' . base TO  . limit ']'
        ;

lambda:   NAME '(' argument_list ')'
        | lambda_binding . '(' . function_expression ')'
        ;

argument_list:
          /* EMPTY */
        | expression_list
        ;

lambda_binding:
          declaration . ':'
        | declaration . lambda_binding
        ;

merge:    string '(' expression_list ')'
        ;

primary:  literal
        | merge
        | element
        | lambda
        | '(' expression . ')'
        ;

monadic:  primary
        | ADD primary
        | SUB primary
        | NOT primary
        | COMPLEMENT primary
```

```
        | SIZE OF element
        ;

exponent: monadic
        | monadic EXP . exponent
        ;

multiplicative:
          exponent
        | multiplicative MUL . exponent
        | multiplicative DIV . exponent
        | multiplicative REM . exponent
        ;

additive: multiplicative
        | additive ADD . multiplicative
        | additive SUB . multiplicative
        ;

shift:    additive
        | shift LSHIFT . additive
        | shift RSHIFT . additive
        ;

relational:
          shift
        | relational LT . shift
        | relational GT . shift
        | relational LE . shift
        | relational GE . shift
        ;

equality: relational
        | equality EQ . relational
        | equality NE . relational
        ;

bitwise:  equality
        | bitwise BITAND . equality
        | bitwise BITOR  . equality
        | bitwise BITXOR . equality
        ;
```

```
logical:  bitwise
        | logical AND . bitwise
        | logical OR  . bitwise
        | logical XOR . bitwise
        ;

constraint:
          logical
        | logical COERCE type
        | logical COERCE TRUNC type
        | logical COERCE ROUND type
        | logical ASSERT type
        | logical CAST   type
        ;

expression:
          constraint
        | IF boolean .. expression . ELSE . expression
        ;

expression_list:
          expression
        | expression ',' . expression_list
        ;

/* semantic limiters */
subscript:integer
        ;

base:     integer
        ;

count:    integer
        ;

limit:    integer
        ;

step:     integer
        ;

boolean:  expression
        ;
```

```
integer:  expression
        ;

float:    expression
        ;

/* Newlines and colon */
.:        /* EMPTY */
        | ...
        ;

...:          NEWLINE
        | ... NEWLINE
        ;

..:          ...
        | ':' .
        ;

%%
```

Appendix C

A FLEX lexer

```
/* Ease Compiler - Copyright (C) 1991, 1992 Steven Ericsson Zenith

   A flex specified lexer for
   Ease - a language for programming concurrent systems
   Copyright (C) 1991, 1992 Steven Ericsson Zenith.
   Copyright (C) 1992 Science Frontiers, International.
                     #HPC01 Computer Science Project.

   This lexer accompanies the grammar "ease.y".

   Version Beta .09 April 1992.

*/
%{

#include <stdio.h>
#include "ease_tab.h" /* bison generated definitions */
#include "ease.h"
#include "version.h"

int lineno = 1;
%}

Name              [a-zA-Z][0-9a-zA-Z_']*
Whitespace        [ \t]+
Newline           [\n\f\r]
Integer           [0-9]*|"#"[0-9A-Fa-f]*
Float             [0-9]+("."[0-9]+)?[eE][+-]?[0-9]+|[0-9]+"."[0-9]*
```

```
Text            \"(\\.|[^\\"])*\"
Character       '.'|'\\.'
%%

"/*"            comment();

{Whitespace}    ;

{Newline}       {
        ++lineno;
        return(NEWLINE);
}

test            { return(TEST);         }
select          { return(SELECT);       }
resource        { return(RESOURCE);     }
choice          { return(CHOICE);       }
else            { return(ELSE);         }
after           { return(AFTER);        }
on              { return(ON);           }
while           { return(WHILE);        }
do              { return(DO);           }
until           { return(UNTIL);        }
for             { return(FOR);          }
from            { return(FROM);         }
by              { return(BY);           }

and             { return(AND);          }
or              { return(OR);           }
xor             { return(XOR);          }
not             { return(NOT);          }
true            { return(TRUE);         }
false           { return(FALSE);        }
if              { return(IF);           }
size            { return(SIZE);         }
of              { return(OF);           }

let             { return(LET);          }
val             { return(VAL);          }
rename          { return(RENAME);       }
type            { return(TYPE);         }
enum            { return(ENUM);         }
is              { return(IS);           }
```

```
context          { return(CONTEXT);     }
stream           { return(STREAM);      }
single           { return(SINGLE);      }
lo               { return(LO);          }
hi               { return(HI);          }
reply            { return(REPLY);       }

procedure        { return(PROCEDURE);   }
function         { return(FUNCTION);    }
where            { return(WHERE);       }

bool             { return(BOOL);        }

int              { return(INT);         }
int[8]           { return(INT8);        }
int[1][6]        { return(INT16);       }
int[3][2]        { return(INT32);       }
int[6][4]        { return(INT64);       }
int[1][2][8]     { return(INT128);      }

float[3][2]      { return(FLOAT32);     }
float[6][4]      { return(FLOAT64);     }
float[1][2][8]   { return(FLOAT128);    }

string           { return(STRING);      }
char             { return(CHAR);        }

module           { return(MODULE);      }
end              { return(END);         }
show             { return(SHOW);        }
hide             { return(HIDE);        }
use              { return(USE);         }

stop             { return(STOP);        }
skip             { return(SKIP);        }

round            { return(ROUND);       }
trunc            { return(TRUNC);       }

at               { return(AT);          }

{Name}           {
```

```
        yylval.value = (char *)strdup(yytext);
        return(NAME);
}

{Integer}        {
        default_type = "int";
        yylval.value = (char *)strdup(yytext);
        return(NUMBER);
}

{Float}          {
        default_type = "float32";
        yylval.value = (char *)strdup(yytext);
        return(NUMBER);
}

{Text}           {
        default_type = "string";
        yylval.value = (char *)strdup(yytext);
        return(TEXT);
}{Character}      {
        default_type = "char";
        yylval.value = (char *)strdup(yytext);
        return(CHARACTER);
}

"{"              { return(SEQUENCE);    }
"||"             { return(COOPERATE);   }
"//"             { return(SUBORDINATE); }

"++"             { return(INCREMENT);   }
"--"             { return(DECREMENT);   }

":="             { return(BECOMES);     }
"?*"             { return(GET);         }
"!*"             { return(PUT);         }
"?"              { return(READ);        }
"!"              { return(WRITE);       }

"^"              { return(EXP);         }
"*"              { return(MUL);         }
"/"              { return(DIV);         }
"%"              { return(REM);         }
```

```
"-"                 { return(SUB);          }
"+"                 { return(ADD);          }

"="                 { return(EQ);           }
"<>"                { return(NE);           }
">"                 { return(GT);           }
"<"                 { return(LT);           }
">="                { return(GE);           }
"<="                { return(LE);           }

"/\\"               { return(BITAND);       }
"\\/"               { return(BITOR);        }
"><"                { return(BITXOR);       }
">>"                { return(RSHIFT);       }
"<<"                { return(LSHIFT);       }
"~"                 { return(COMPLEMENT);   }

"(.."               { return(OPENTO);       }
"..)"               { return(CLOSETO);      }
".."                { return(TO);           }

"=>"                { return(ASSERT);       }
"->"                { return(COERCE);       }
">|"                { return(CAST);         }

"::"                { return(COUNTED);      }

"}"                 { return('}');          }
":"                 { return(':');          }
";"                 { return(';');          }

"("                 { return('(');          }
")"                 { return(')');          }

"["                 { return('[');          }
"]"                 { return(']');          }

"|"                 { return('|');          }

","                 { return(',');          }

"_"                 { return('_');          }
```

```
"\\"            { return('\\');        }

.               {
        char m[40];
        sprintf(m,"illegal character '\\%o'", (int)yytext[0]);
        yyerror(m);
}

<<EOF>>         {
        if (diagnose) printf("/* Lexer: end of file */\n");
        yyterminate();
}
%%

int comment()
/* For now we won't take care of nested comments or premature EOF
*/
{
        char c ;

        while ((c = input()) != '/') {
           if (c == '\n') ++lineno ;
           while ((c = input()) != '*') if (c == '\n') lineno++ ;
        }
}

yyerror(char *s)
{
        printf("Ease %s: at line %d = %s (%s)\n",
                EASE_VERSION, lineno, s, yytext);
}
```

Appendix D

C–with–Ease

D.1 *C–with–Ease* definition

At the highest level a *C–with–Ease* program consists of a collection of processes which interact via shared data structures.

The C language is enhanced by combining with the *Ease* process and process interaction model.

Ease provides simple and symmetric operators (read and write, get and put), a well defined process model. Constructions for both cooperative and subordinate concurrency and a mechanism (combinations) for building statically reusable and virtual resources on parallel and distributed machines.

The Ease keywords which appear in C programs begin with an *escape* character (generally %) to distinguish them from the names used in the standard C function library.

D.1.1 Process definition

A process differs from a C function since it does not return a value and obeys the *Ease* rules for parallel construction which place restrictions on the use of free variables and pointers.

A process is defined in a similar way to ANSI–C function definitions except where C functions specify the type of the returned value, processes are distinguish from functions by the keyword `%process`.

definition	=	`process` *name* `(` $\{_0$ `,` *formal* `)`
		body
body	=	*compound_statement*
process	=	*name* `(` *actual* `)`

Since the body of a process does not return a value the body is simply terminated by the use of the C keyword `return`.

Names declared in a process definition may only appear in *Ease* process creation statements.

D.1.2 Parallel construction

cooperation	=	`cooperate` `{` $\langle_1$ `;` *process*$\rangle$ `}`
	|	`cooperate` `(` *replicator* `)` *process* `;`

The processes in a cooperation start simultaneously and continue together. A cooperation terminates when all the components of the cooperation have terminated.

A free variable can be assigned to in only one component of a cooperation. If a free variable is assigned to, it can only appear in the assigning component. Otherwise, free variables can appear in the expressions of all components.

A context can be output by only one component in a cooperation. If a context is output, it can only appear in the outputting component. Otherwise, free contexts can appear in the interactions of all components.

subordination	=	`subordinate` *process* `;`
	|	`subordinate` `(` *replicator* `)` *process* `;`

The components of a subordination start simultaneously and continue independently. A subordination terminates when all the components of the subordination have started.

A subordinate process may not contain references to free variables in its scope. Subordinate processes may not output a free context. A subordinate process terminates if it attempts to interact with a context whose scope has terminated or whose value has been output.

D.1.3 Replication

```
replicator = name [for] count
           | name [for] count [from] base
           | name [for] count [from] base [by] step
count      = expression
base       = expression
step       = expression
```

D.1.4 Contexts – shared data structures

A context is a shared data structure, which may be an unordered bag, a stream or a singleton. The type of data in a context is defined by equivalence to those in the reference language.

For C programmers the most significant thing to observe is that pointers may not be used in contexts. C pointers are not meaningful things to exchange between processes since subordinates may not share the same address space. However, cooperating processes may share access to free pointers according to the rules for cooperation.

Indeed, put and get operations are designed to take care of data exchange by reference, since they manipulate pointers where they may and copy data where they may not.

Types

The types used in *Ease* operations exclude pointers and unions. However, no restriction is placed on the use of pointers within a process.

```
type = data_type
     | [stream] type
     | [[] expression []] type
     | data_type [reply] data_type
```

data_type	=	*primitive_type*
	|	*array_type*
	|	*tuple_type*
primitive_type	=	*integer_type*
	|	`unsigned` *integer_type*
	|	`signed` *integer_type*
	|	*float_type*
`signed` *integer_type*	=	*integer_type*

The following equivalences define primitive types and syntax in C-with-Ease in terms of equivalence to those specified in the reference language.

integer_type	=	`int`	≡INT
	|	`char`	≡INT8|UNSIGNED INT8†
	|	`short`	≡INT16|INT32†
	|	`short int`	≡INT16|INT32†
	|	`long`	≡INT32|INT64†
	|	`long int`	≡INT32|INT64†
float_type	=	`float`	≡FLOAT32
	|	`double`	≡FLOAT64

Those types marked by † are machine dependent. An implementation must state the true equivalence.

array_type = *type* `[` *expression* `]`

An array type is a homogeneous sequence of components of some type. The size of the sequence is specified by the associate expression, which must be of integer type.

These arrays are directly equivalent to arrays in the reference language.

array_type	=	*integer_type* `::` `[` *count* `]` *type*
count	=	*expression*

A counted array type is a *count* component of an integer type, followed by a homogeneous sequence of components of some type. The size of the sequence is bounded to be not greater than the value specified by the associated *count* expression.

Let e be an expression, I an integer type and t be some type, then the type $I :: [e]t$ is valid iff $e > 0$.

$$tuple_type = \boxed{\texttt{struct}}\ \boxed{\{}\ \langle_2 \boxed{,}\ type\rangle \boxed{\}}$$

Tuple types are equivalent to structurally equivalent ANSI C standard structures.

Type definitions

A name defined by a C type definition is valid only if the types in the definition are structurally compatible which those mentioned.

D.1.5 Context type definition

$$\begin{aligned} definition &= \boxed{\texttt{context}}\ name\ type\ \boxed{;} \\ &\mid \boxed{\texttt{context}}\ name\ \boxed{\{}\ \langle_1 \boxed{;}\ type\ \rangle \boxed{\}} \\ type &= name \end{aligned}$$

A context type definition defines a name for the specified context type. A context whose type is defined by a type definition is of the same type if their type has been defined in the same definition (i.e. name equivalence).

D.1.6 Context allocation

$$\begin{aligned} allocation &= \boxed{\texttt{share}}\ type\ context\ \boxed{;} \\ &\mid placement \\ context &= name \end{aligned}$$

An allocation specifies a name for a context.

placement = `share` *type context* `on` $\langle_1$*node* $\rangle$
  | `share` *type context* `on` *node* `at` *address*

node = *expression*
address = *expression*

A placement allocates a context (which must be a singleton) on a node or group of nodes.

D.1.7 Interaction

interaction = *input* | *output*

input = *read* | *get*
output = *write* | *put*

read = `read` `(` *context* `,` *variable* `)`
write = `write` `(` *context* `,` *expression* `)`

get = `get` `(` *context* `,` *name* `)`
put = `put` `(` *context* `,` *name* `)`

The interactions specified here are simple syntactic variants of those specified in the reference language.

There are four simple, symmetric, operations on contexts. They are

- write (c, e) – copies the value of the expression e to the context c.
- read (c, v) – copies a value from the context c to a variable v.

- put (c, n) – moves the value associated with the name n to the context c.
- get (c, n) – moves a value from the context c and binds it to the name n.

Write and read are copy operations. Put and get are binding operators.

The synchronization characteristics of the operations are similarly symmetric

- get and read block if data is not existent
- write and put are non-blocking.

Consider how these operations change the state of a program.

Write changes the state of a context, leaving the local state unchanged. Read changes the local state whilst leaving the context state unchanged.

Put changes both the context state and local state, i.e. subsequently the value associated with the variable name used in the operation is undefined. Get also changes both the context state and the local state, i.e. the value bound to the variable name used in the operation is removed from the context.

D.1.8 Resource

The construction of and interaction with resources has special requirements. To enable the simple and uniform view of resources in parallel and distributed environments, Ease provides *combinations*.

combination	=	*call* | *reply*
call	=	`call` `(` *context* `,` *expression* `,` *name* `)`
reply	=	`resource` `(` *context* `,` *name* `)`
		`reply` *function* `;`

A *combination* provides guaranteed *call reply* semantics via some context. Access to system resources is provided by use of combinations.

A *call* behaves like an output and get in sequence.

A *reply* behaves like a get, process and output in sequence.

The behavior of a combination is described as the synchronization of the calling process and the resource process, where the output of the resource process in the call reply context is guaranteed to satisfy the input of the corresponding call.

D.1.9 Scope

The scope of a context allocated in a process is from the point of allocation to the end of that process. æ