Functional Programming and Embedded Systems

Malcolm Wallace, MEng

Thesis submitted for the degree of DPhil in Computer Science

Department of Computer Science

University of York

January 1995

Abstract

Embedded computer systems seem to be the antithesis of functional language systems. Embedded systems are small, stand-alone, and are often forced to accept inelegant design compromises due to hardware cost. They run continuously and are reactive, that is, their primary goal is to monitor sensors and control effectors, using observed external events to trigger state-changing control actions. Yet this thesis describes how functional abstraction can tame the inelegance of embedded systems. Architectural compromises can be made in device drivers, programmed within the functional language, but a function-level interface is presented to the application programmer.

Four modifications are introduced to a test-bed purely-functional language in order to facilitate embedded-systems programming: I/O register access; communicating processes; interrupts; and a real-time incremental garbage collector. Referential transparency is preserved. The conventional model of communicating processes is augmented by the use of type classes and constructor classes to add type security around message-passing.

Two case studies of embedded applications are programmed: a marble-sorter and a liftshaft. Both give encouraging results for the adequacy of the approach. Two important areas are identified for future research. (1) It is not clear to what extent lazy evaluation is a help or a hindrance in programming embedded systems. (2) Real-time expression and guarantees of schedulability would extend the attractiveness of functional languages still further.

Contents

Abstract 2

Contents 3

List of Figures 8

List of Tables 10

Acknowledgements 11

Author's Declaration 12

1. Introduction 13
1.1. Aims 13
1.2. Functional programming languages 15
1.3. Embedded systems programming languages 16
1.4. Motivation: bringing the two together 18
1.5. Roadmap 19

2. I/O, exceptions, and processes 21
2.1. Functional approaches to I/O 21
2.1.1. *Landin streams* 22
2.1.2. *Matched streams* 22
2.1.3. *Continuations* 23
2.1.4. *Monads* 24
2.2. Low-level I/O 25
2.2.1. *Some approaches* 25
2.2.2. *New proposal* 26
2.3. High-level I/O 26
2.4. Interrupt handling and exceptions 27
2.4.1. *Program-generated signals* 28
2.4.2. *Environment-generated signals* 29
2.4.3. *Continuation-based handlers* 29
2.4.4. *A lifted datatype* 30
2.4.5. *Failure as a list of successes* 30
2.5. Non-determinism 31
2.6. Communicating processes 33
2.6.1. *Models* 33
2.6.2. *Previous implementations* 35
2.7. Summary 36

3. Embedded functional I/O 38
3.1. Outline of interrupt scheme 38
3.2. Design considerations 39
3.2.1. *Communicating processes* 39
3.2.2. *I/O devices* 40
3.2.3. *Interrupts* 41
3.2.4. *Main programs* 41
3.2.5. *Synchronisation, laziness, and scheduling* 41
3.3. Programmer's view of process I/O 42
3.3.1. *Types* 43
3.3.2. *Request and Response constructors* 43
3.3.3. *Message parity rules* 44
3.3.4. *Continuations* 44
3.3.5. *CPS combinators* 46
3.4. Some implementation details 47
3.4.1. *Run-time evaluator modifications* 47
3.4.2. *Building and interpreting the process table* 48
3.4.3. *I/O primitives* 49
3.4.4. *Scheduler* 51
3.4.5. *Target embedded board* 53
3.5. Results 53
3.6. Conclusions 55

4. Case study I: a marble sorter 57
4.1. Dividing the program into processes 57
4.2. The hopper mechanism 58
4.3. Detectors 59
4.4. Sorter arm 59
4.5. Terminal-line driver 60
4.6. The clock/timer 62
4.7. Bringing it all together 63
4.8. Performance 64
4.9. Conclusions 64

5.	**Type-checked message passing**	**66**
	5.1. Introduction	66
	5.1.1. *The problem*	66
	5.1.2. *Some previous solutions*	66
	5.1.3. *A new solution*	68
	5.2. Modern functional I/O	68
	5.2.1. *Monads*	68
	5.2.2. *Processes*	69
	5.3. Overloading expresses addressing	70
	5.3.1. *Constraints on addresses*	70
	5.3.2. *Using overloading*	71
	5.3.3. *Examples*	71
	5.4. Specialised classes of process	72
	5.4.1. *Flat specialisation*	72
	5.4.2. *Layered specialisation*	73
	5.5. Conclusions	73
6.	**Case study II: a model liftshaft**	**75**
	6.1. Apparatus	75
	6.2. Method	77
	6.3. Overview of Embedded Gofer's monadic I/O	79
	6.3.1. *Why change to monads?*	79
	6.3.2. *Combinators and loops*	80
	6.4. The timer device	81
	6.4.1. *Periodic interrupts*	81
	6.4.2. *Timeouts*	82
	6.4.3. *Elapsed time*	82
	6.4.4. *Mixed models*	83
	6.4.5. *Discussion*	84
	6.5. Controlling other devices	85
	6.5.1. *Registers are shared*	85
	6.5.2. *Writing shared registers: a server process*	86
	6.5.3. *Reading shared registers: the need to poll*	87
	6.5.4. *An example program*	88
	6.5.5. *A further set of shared registers*	89
	6.6. Interrupts	90
	6.6.1. *Telemetry technique*	90
	6.6.2. *Example*	91
	6.7. A naive scheduling algorithm	92
	6.8. The full solution	94
	6.8.1. *A new intuition*	94
	6.8.2. *Independent lifts or communicating lifts?*	96
	6.8.3. *The process table*	98
	6.9. Conclusions	98

7. Garbage collection **100**

7.1. A review of some previous collectors 100

7.1.1. *Incremental collection* 100

7.1.2. *Queinnec's Mark-During-Sweep method* 101

7.1.3. *Yuasa's Stack-Collect method* 102

7.2. A hybrid algorithm 103

7.2.1. *The marker* 104

7.2.2. *The sweeper* 105

7.2.3. *The collector* 106

7.2.4. *The allocator* 106

7.2.5. *The mutator* 106

7.2.6. *Discussion* 106

7.2.7. *Optimisations* 107

7.3. Sketch of proof 107

7.3.1. *Unbounded stack* 107

7.3.2. *Termination* 108

7.3.3. *Bounded stack* 109

7.4. Results 109

7.5. Conclusions 112

8. Real-time issues **114**

8.1. Introduction 114

8.2. Timing expression 115

8.2.1. *Clock requests* 116

8.2.2. *Timed evaluation of streams* 116

8.2.3. *Timestamped data in streams* 117

8.2.4. *Dataflow model: time windows in LUCID* 119

8.2.5. *Dataflow model: timing nets in LUCID* 120

8.2.6. *Dataflow model: strong synchrony* 120

8.2.7. *Time description in musical languages* 121

8.2.8. *A functional/sequential hybrid* 123

8.2.9. *Erlang* 124

8.2.10. *Summary* 124

8.3. Deadline guarantees 125

8.3.1. *Classifications of real-time systems* 126

8.3.2. *Brief review of scheduling analyses* 126

8.3.3. *Guaranteed scheduling in a functional language?* 127

8.3.4. *Laziness* 128

8.4. Conclusions 129

9. Conclusions and future work **131**
9.1. A functional language for embedded systems 131
9.1.1. *I/O control* 131
9.1.2. *Interrupts* 132
9.1.3. *Memory management* 132
9.1.4. *Distinctives* 132
9.2. Evaluating the effectiveness of Embedded Gofer 133
9.2.1. *Higher-order functions* 133
9.2.2. *Lazy evaluation* 134
9.2.3. *Type system* 135
9.3. Further issues 135
9.4. Future work 136
9.5. Summary 137

Appendix A. Addendum to the Gofer manual **138**
A.1. Processes 138
A.2. Process types and primitives 138
A.3. The process prelude 140
A.4. Hexadecimal constants and bit-wise operations 141

Appendix B. The monadic marble sorter **143**

Appendix C. Detailed device-driving code: the register model **145**
C.1. PIT 68230 145
C.1.1. *Data port initialisation* 145
C.1.2. *Timer initialisation* 148
C.2. DUART 68681 148
C.2.1. *Initialisation* 148
C.2.2. *Serial line use* 151

Appendix D. Shared register server process **152**

References **154**

List of Figures

2.1 I/O: matched stream example. 22
2.2 I/O: Landin stream example. 23
2.3 I/O: continuation example. 24
2.4 I/O: monadic example. 25
2.5 I/O: example of abstraction. 27

3.1 Implementation: process table structure. 48
3.2 Implementation: evaluating `main`. 49
3.3 Implementation: building the process table. 49
3.4 Implementation: sending a message. 50
3.5 Implementation: receiving an interrupt or message. 50
3.6 Implementation: context switching. 51
3.7 Implementation: the scheduler. 52
3.8 Implementation: interrupt handling. 52

4.1 Marble-sorting apparatus. 57
4.2 Marble-sorter: processes. 58
4.3 Marble-sorter: hopper mechanism. 58
4.4 Marble-sorter: detectors. 59
4.5 Marble-sorter: sorter mechanism. 60
4.6 Marble-sorter: terminal-line declarations. 60
4.7 Marble-sorter: transmitting a character. 60
4.8 Marble-sorter: transmitting a string. 60
4.9 Marble-sorter: screen driver. 60
4.10 Marble-sorter: background keyboard driver. 61
4.11 Marble-sorter: polling keyboard driver. 61
4.12 Marble-sorter: inefficient alarm. 62
4.13 Marble-sorter: efficient alarm. 63
4.14 Marble-sorter: `wait` operation. 63
4.15 Marble-sorter: `defer` operation. 63
4.16 Marble-sorter: message datatype. 64
4.17 Marble-sorter: main program. 64

6.1 Liftshaft: car mechanism. 75
6.2 Liftshaft: shaft layout. 76
6.3 Liftshaft: periodic timing. 81
6.4 Liftshaft: timeouts. 82
6.5 Liftshaft: elapsed time server. 83
6.6 Liftshaft: full alarm server. 84
6.7 Liftshaft: PIT effectors and sensors. 86
6.8 Liftshaft: shared registers. 87

6.9	Liftshaft: register update functions.	87
6.10	Liftshaft: extension to shared register server.	88
6.11	Liftshaft: reading shadowed registers.	88
6.12	Liftshaft: first exercise.	89
6.13	Liftshaft: DUART registers.	89
6.14	Liftshaft: basic telemetry.	90
6.15	Liftshaft: virtual interrupts.	91
6.16	Liftshaft: memoised telemetry server.	91
6.17	Liftshaft: second exercise.	92
6.18	Liftshaft: second exercise re-coded.	92
6.19	Liftshaft: naive scheduling algorithm.	93
6.20	Liftshaft: determination of requests.	93
6.21	Liftshaft: the full algorithm.	95
6.22	Liftshaft: new determination of lift requests.	96
6.23	Liftshaft: the process table.	98
7.1	GC: Heap implementation.	103
7.2	GC: The incremental marker.	104
7.3	GC: Stack operations.	104
7.4	GC: Safety-bit colouring.	105
7.5	GC: Safety-bit search operation.	105
7.6	GC: The incremental sweeper and collector.	106
7.7	GC: The allocator and mutator operations.	106
A.1	*proc.prelude*: control structures.	141
A.2	*bit.prelude*.	142
B.1	Monadic marble-sorting.	143
B.2	Monadic marble-sorting (continued).	144
C.1	PIT: register addresses.	145
C.2	PIT: initialisation.	146
C.3	PIT: descriptive initialisation.	147
C.4	PIT: timer register addresses.	148
C.5	PIT: timer initialisation.	148
C.6	DUART: two base addresses.	149
C.7	DUART: register addresses.	149
C.8	DUART: programming information.	150
C.9	DUART: register initialisation.	150
C.10	Operations *putchar* and *getchar*.	151
D.1	Shadowed register access.	152
D.2	Shared register server.	153

List of Tables

3.1	I/O requests and expected responses.	44
3.2	Comparative cost of RTS routines.	54
3.3	Costs in other systems.	54
3.4	Sample of times for primitive applications.	55
7.1	GC: Varying stack size.	110
7.2	GC: Achieving work parity.	111
7.3	GC: Keeping work parity.	111
7.4	GC: Reducing the heap size.	111
8.1	The effect of Arctic operators.	122
9.1	Application code-size comparisons.	133

Acknowledgements

This work was supported by a Research Studentship from the Department of Education for Northern Ireland, and latterly in part by Canon Research Centre Europe Ltd. Thanks are due to many people. First and most important is Colin Runciman, whose supervisorial eye was forever constructive, inspiring and organised. David Wakeling provided a sounding board for several ideas early on in this project. Andy Wellings loaned the marble-sorter apparatus to us, and kept pushing the question "but is it real-time?" which served to tighten up our terminology. Rick Pack patiently configured and re-configured the embedded computer hardware to extend its memory and swap between marbles and lifts again and again. Gary Morgan provided low-level libraries and loading software for the embedded board.

Beyond York, Mark Jones always gave ready advice on the internal workings of Gofer, and his constant improvement of the language's implementation frequently made my own work easier. Several anonymous conference and journal referees contributed helpful suggestions about the material of Chapters 3, 4, and 5. Niklas Röjemo and Thomas Hallgren made useful comments on the Stack-Safety garbage collection algorithm presented in Chapter 7. Christian Queinnec cleared up some questions about his Mark-During-Sweep GC algorithm. Phil Wadler inspired the push to convert to monadic I/O which resulted in the material of Chapter 5.

Finally, there's no telling how I have would have coped with any of this work without a healthy dose of play, so thanks to family and friends for making life fun. Mum and Dad's support was invaluable; Sue's love is great; Tim is a good correspondent; and all in the Warehouse community made relating to God so very refreshing. Thanks.

Author's Declaration

The material of Chapter 7 was presented at the Chalmers Winter Meeting 1993, and appeared in the Proceedings [WallaceRunciman93]. An early version of the content of Chapter 3 was presented to an IFIP WG2.8 workshop in September 1993. A cut-down version of Chapters 3 and 4 has been published as a paper in Software Practice and Experience [WallaceRunciman95], and I have presented it at seminars at the Universities of Hull and Kent, and University College London. I presented Chapter 5 as a paper at the Glasgow Workshop on Functional Programming 1994, and it appears in the Proceedings [WallaceRunciman94].

Unless otherwise stated in the text, the ideas presented in this thesis are solely the work of the author.

Chapter 1. Introduction

The study of computer science is now a very wide field, and it is expanding at an ever increasing rate as new hardware architectures and software techniques are developed. Computers themselves are increasingly ubiquitous in the everyday life of the public at large, both as stand-alone "personal computing" machines and hidden inside almost every household consumer product, vehicle, or communications device.

As hardware technology has advanced and become ever cheaper to manufacture, the software technology which specialises that hardware for specific applications has struggled to keep pace. The speed and correctness with which new applications can be developed and programmed is a limiting factor both as regards their safety and also in commercial terms. People in the Western world are now so dependent on computing systems, for instance in nuclear power plant control, aircraft guidance, and financial accounting, that errors in software can have catastrophic and life-threatening results.

Against this backdrop, we find that although advanced software techniques such as functional programming, formal specification, and refinement calculi not only exist but have existed for many years, very few applications yet make adequate use of them. Looking more closely at some of the reasons for this, it appears that there are still some technical problems in using such techniques more generally than they have been up to now, in terms of their integration with existing systems. The inertia caused by the enormous number of systems already in use precludes the widespread introduction of any technique too radical. For instance, developing specialised hardware to support a functional language's evaluation model is not viable – better implementations of functional languages on *existing* hardware are needed.

One particular area of computing technology that has so far been incompatible with the introduction of functional languages is embedded systems: the computer hidden inside another machine.

1.1. Aims

Therefore it is the aim of this thesis:

(i) to extend the effective applicability of functional languages to encompass embedded systems;

(ii) to demonstrate that functional programming is a help to the embedded systems programmer, in that it provides the advantages of conciseness of expression, modularity, and analysability to a field which requires dependable and maintainable software.

Functional languages are generally used in the environment of workstations with large memory, big disks, fast processors, virtual memory and an underlying operating system. However, functional languages could also be used in the programming of small computer systems, even embedded systems with small memory, no backing storage, and no operating system.

Some previous applications of functional languages to embedded systems do exist [FijmaUdink91] [Armstrong+93], but their implementations tend to have undesirable features specific to the particular application. The alternative, presented in this thesis, is to develop a general scheme for high-level programming which can be applied to a range of low-level controller hardware.

This thesis is based on practical work extending a functional language and its implementation for embedded-systems control, and applying it to case studies in an effort to solve those exercises adequately and elegantly. Issues covered in detail include embedded I/O, interrupt handling, and garbage collection. We also look briefly at the closely related field of real-time systems.

Although imperative languages are used to program most embedded systems, they have several undesirable properties. Too many errors are allowed to go undetected: in parameter passing, in types, in memory management, and so on. Often, a program's task is obscured by the language's structures, rather than being clearly described. Writing programs can take a long time, and it is hard to be sure that they are correct. Maintenance costs are therefore high.

This thesis argues that functional programming techniques have considerable potential to reduce or eliminate the problems just noted. Functional languages can reduce the number of undetected errors: higher-order combining functions, strong polymorphic type systems, and automatic memory management all reduce the possibility of direct programming errors. The equational style allows a more direct and concise expression of programs, amenable to simple reasoning by substitution. Functional programs can be written in a shorter time, with greater confidence that they are correct, and they are easier to maintain.

In this chapter we give an introduction to functional languages and to embedded systems, then examine in more detail the specific requirements of embedded systems. A ''roadmap'' of later chapters is given: each chapter shows how some particular requirements have not been met by functional languages in the past, and how they can now be addressed by particular techniques.

1.2. Functional programming languages

A *functional language* is generally taken to be one in which computation proceeds entirely by the evaluation of expressions without side-effects, and where functions are treated as first-class objects [Hudak89]. This is in contra-distinction to *imperative* languages, in which programs operate almost entirely by side-effects; that is, where functions are a form of static subroutine and they operate by making alterations to an implicit global state. In a pure functional language the assignment statement for destructive update to variables is outlawed. Instead, a name always refers to the same value within its scope. Of course, global and local state may be modelled functionally, but this state is expressed and modified explicitly. For instance, data structures called *lists* can represent a series of values, where an imperative language might use sequential assignment to a single location.

An important property of pure functional languages is *referential transparency*. Because one name always refers to the same value (perhaps computed through an expression), substitution of expression for name, value for name, value for expression, and so on is possible. A referentially transparent language is much easier to reason about mathematically than one which is not [SondergaardSestoft88b]. The style is *equational* rather than imperative. The *lambda calculus* [Church41] is the formal underpinning to all functional languages, dealing with function *abstraction* (i.e. parameterisation of an expression) and its converse, function *application*.

Other advantages claimed over imperative languages include: programs can be written more quickly and accurately; they are more concise, allowing higher levels of abstraction over the basic low-level behaviour; and they are more amenable to parallel execution [Hudak89]. These qualities are needed in almost every application area, including embedded systems.

Modern functional languages tend to have many syntactic and semantic features to aid the programmer which, although possible in more traditional languages, are not often found there: pattern matching in equations; lazy (i.e. demand-driven) evaluation; polymorphic type systems; data abstraction; higher-order functions; automatic storage allocation and reclamation; and so on. Recursion is the usual style of programming, but this is often concealed behind higher-order functions.

The most common accusation levelled against the functional style is that implementations are inherently inefficient in terms of both space and time usage. Having to use a sequence of values to model alteration in the value of a single abstract object is seen as wasteful of memory. In defence, the functional community is researching the whole area of safe *destructive update*, that is, the ability to collapse a structure whose constituent values are time-ordered into a single storage block whose contents may be destructively overwritten by the implementation without destroying the substitutive properties of the program [Swarup+91]. This would make the spatial requirements of a functional program

broadly comparable to an imperative implementation. It is now the case that some functional language compilers can generate code which runs as fast as (for instance) C [Smetsers+91], and for at least one numerically intensive application, a functional program has parity in both speed and space usage compared to hand-written FORTRAN [BoyleHarmer92]. Functional languages more generally are within an order of magnitude of their imperative counterparts. They could be valuable tools for "real" production programming.

Recursion is often prohibited when programming embedded or real-time systems, because the consequent space and time requirements are potentially unbounded. This conflicts with the status of recursion as a primary tool of the functional programmer. In fact a particular form of recursion, *tail recursion*, is equivalent to an imperative loop, having a constant space requirement. With careful analysis it is often possible to provide bounds on the depth of recursion, and hence bounds on time requirements. It may be that a functional language for embedded systems would have to enforce these constraints on the programmer by syntactic means.

1.3. Embedded systems programming languages

Many programming languages have been designed to run on large, general-purpose computers that typically have a screen, keyboard, and file store. Input and output (I/O) to and from these devices is usually provided either in the language itself, or through privileged library routines. Most functional languages fall into this class. However for some applications, notably control systems, this characterisation of the need for I/O is totally inadequate. Examples of such applications include the microprocessor controllers inside vehicles, factory machinery, and household consumer items. The common theme of these applications is that the computer is *embedded* within a system whose *primary* purpose is not computation. In every different system, the computer must control different transducers, and read different sensors. In other words, the specific I/O requirements are unique to each application.

For this reason, the central issue in designing an *embedded systems programming language* is the treatment of I/O. The language must provide facilities that are low-level enough to allow direct access to devices, but that are also general enough to be applicable to many different sorts of device.

In many embedded systems, the cost of the computing component is significant with respect to the cost of the whole unit. The component cost is usually minimised by using off-the-shelf processors, memory, and connecting logic, rather than developing specialised hardware. Essentially, this means that a new language compiler does not need to be written for every new embedded system, merely for each target processor. However, even though the I/O devices connected to the processor are also off-the-shelf, their activity cannot be "wired in" to the compiler in the same way. Identical data and status values from

the same I/O devices in different physical systems should have different interpretations. This is why the programmer must have control of I/O.

There are several secondary objectives in the design of embedded systems languages [BurnsWellings89].

(i) *Dependability.* Embedded applications are often critical systems. In controlling a nuclear power plant, or a vehicle, one must be as certain as possible that the software is correct. Even for non-critical systems, commercial considerations dictate that a product must work correctly. This means that the ability to argue formally, i.e. mathematically, about properties of programs is needed.

(ii) The *size and complexity* of program source must be manageable, because confidence in the dependability of a program can be higher if it is easy to read and understand. Short, readable programs are easier to maintain for the same reasons of confidence.

(iii) *Modularity* is very desirable, especially during program maintenance. If the boundaries between the parts of a program are clear, errors can be located more easily and precisely.

(iv) *Cost.* An efficient language implementation, by using small amounts of *memory* and *time*, is helpful in reducing the cost of the overall system; fewer and less complicated components can be used in the physical construction of the computer system. For mass-produced embedded systems, this end unit-cost is important. Also important in some applications is the *weight* and *size* of the final system, which is of course also affected by the software's efficiency.

(v) *Concurrency* in the sense of multiple distributed processors is sometimes desirable for speed.

(vi) *Communicating processes.* Whether or not parallel processors are available, the parallel communicating-tasks style of programming is very popular. Each unit of program is largely independent, but tasks pass messages between each other.

(vii) *Exception handling* is the term for recovering from exceptional conditions. There is some debate about just what constitutes an exceptional condition – should the base case of a recursion be programmed as an exception, or should exceptions be reserved for things like overflow errors? The latter is more generally accepted.

There are sub-classes of embedded systems in which other issues are to the fore: most notably real-time systems. Real-time systems are often confused with embedded systems; here they are treated as a separate class. However, we adopt a design principle that any features introduced to a language to facilitate embedded I/O must not conflict with the language's future extension to cover real-time too.

1.4. Motivation: bringing the two together

In contrast to imperative languages, which are usually employed in writing embedded software, functional languages permit rapid development of systems, with better confidence in the correctness of the product. Both qualities are very commercially attractive. Functional languages already address several of the requirements set out in section 1.3.

(i) *Dependability.* Whereas imperative languages are notoriously difficult to reason about, functional languages are much more mathematically tractable [Turner81].

(ii) *Size and complexity.* Functional languages are noted for their compactness of expression and elegance [Turner81].

(iii) *Modularity.* The lack of side-effects in functional languages reduces the complexity of interaction between individual components of programs. Both higher-order functions and lazy evaluation add new possibilities for modular programming which are not available in imperative languages [Hughes89].

(iv) *Cost.* Miniaturisation of functional programs is a "new" topic of research. Several space-profiling tools have been developed recently [RuncimanWakeling93] [Sansom94], and there is work in progress on compacting the run-time space usage of programs [Rojemo94].

(v) *Concurrency.* It is often claimed that functional languages could automatically apportion pieces of work to separate processors, due to the absence of side-effects or interference [Burn87]. For instance, given a simple addition of two complex expressions, referential transparency says it is safe for each complex expression to be evaluated separately and in parallel, before the addition is performed. Actually providing automatic parallelisation is still a very active area of research, however. (See [PeytonJones89] for an overview.)

(vi) *Communicating processes.* A lot of previous work, particularly with respect to functional operating systems, has been done in this area. See Chapter 2 for a review.

(vii) *Exception handling.* Various functional models of exceptions have been proposed, and these too are reviewed in Chapter 2.

The remaining unanswered questions which prevent the use of functional languages for embedded systems are those of general I/O capability, especially the ability to handle interrupts, and some cost factors of time and space. This thesis addresses the I/O and interrupt questions and one cost factor: that of garbage collection. Section 1.5 gives an outline of how these are tackled.

This thesis has been motivated by practical examples of embedded systems – the case studies in chapters 4 and 6. These examples were devised independently by others for the purposes of teaching embedded systems and real-time programming. We chose the path of modification and extension to an existing functional language in pursuit of elegant and

workable solutions to these examples. There were two reasons.

(i) By starting from a standard language, the presentation of new elements can be clear to the intended audience.

(ii) The amount of change needed in a functional language to accommodate embedded systems as a target is small compared with the effort of building a functional language from scratch.

The language used throughout is Gofer [Jones94], a dialect of Haskell [Hudak+92], except in reviews of previous research using different languages. Gofer is a portable interpreter and environment for programming, and it also comes with a simple compiler. The system is relatively small and fast by functional interpreter standards and, distributed free with source code, it is ideal for experimentation on small embedded systems. Its main advantages are: that prototyping programs is quicker than with a full compiler; that its sources can easily be modified to implement new features; that it fits on small machines; that its run-time performance is efficient despite the interpretive overhead; and that it is based on an emerging standard for lazy functional languages. Its main disadvantage is that, although it is efficient, in some cases it is not quite fast enough for the embedded applications.

1.5. Roadmap

Chapter 2 reviews common functional approaches to I/O, and the specific I/O needs of embedded systems. It also reviews some previous attempts to apply functional languages to embedded systems, showing that few of these have really addressed the I/O questions satisfactorily. A simple extension to the prevailing I/O model provides a general means to program a large variety of embedded devices. Existing functional models of exception-handling are investigated, to see whether they shed any light on interrupt-handling. An explanation of non-determinism and ways to handle it is followed by a review of the programming technique of ''communicating functional processes''.

Chapter 3 suggests that communicating processes are in fact an elegant means by which interrupts may be handled in a functional setting, and describes the definition and implementation of such a model.

Chapter 4 describes the use of this process model to program a small case study of an embedded system: a marble-sorter.

One deficiency of communicating processes is the lack of type security around messages that pass between processes. Chapter 5 describes some previous means of enforcing a secure type discipline and presents a new scheme, based on the use of constructor classes [Jones93]. The new scheme is seen to provide additional benefit by allowing different

privileges to be accorded to different processes.

Chapter 6, gives an account of a larger case study of an embedded system, programmed using all the techniques introduced thus far.

The remainder of the thesis focuses on issues beyond embedded I/O.

Chapter 7 addresses one cost factor in using functional languages for embedded systems: automatic garbage collection (GC). GC has a pronounced effect on the time and space characteristics of functional programs. The common collector algorithms, both stopping and incremental, are not well suited to embedded systems. A new algorithm is designed specifically to meet some of the special requirements of embedded systems.

Chapter 8 discusses the challenges of using functional languages to program real-time systems as a further specialisation of the needs of embedded systems. Various functional means of modelling time are reviewed. This chapter also discusses the all-important question of guaranteeing timing behaviour, and whether a lazy functional language can ever hope to provide those guarantees. A brief review of some selected formal methods for real-time is added.

Chapter 9 concludes, and points the way forward to questions that still need answers before functional languages can be said to meet all the needs of embedded systems programming.

Chapter 2. I/O, exceptions, and processes

Programs have to interact with the external world, but a functional program is supposed to operate without side-effects. On the face of it, these positions are not easily reconciled. If functional languages are side-effect-free, how can they cause events in the external world to start and stop? In fact, numerous styles of I/O interaction without side-effects have been developed. This chapter reviews those styles, then examines how previous applications of functional languages to embedded systems have dealt with unusual devices. A simple extension to the common I/O model addresses the specific need for embedded systems to use register-level operations. However this provision of low-level I/O does not constrain the programmer to express every I/O operation at the lowest level.

The second part of the chapter reviews the area of interrupt handling and its closely related cousin exception handling. Since the main contribution of this thesis relies on multi-processing, the review also considers the issues associated with introducing processes to the functional I/O model, especially non-determinism, and how previous research has tackled these.

2.1. Functional approaches to I/O

Some early functional languages provided what were called *pseudo-functions* [McCarthy+62] for I/O. Although a call of `print` or `read` looks like a function application and returns a result accordingly, in fact such calls operate by hidden side-effects on the state of some input and output devices. Their disadvantage is that they are *non-deterministic* (see section 2.5), and this is especially apparent in lazy languages, where the order of evaluation of parts of a program is often unexpected. Most modern functional languages, and in particular lazy ones, strive for *purity* by rejecting non-determinism and hidden side-effects.

Two common purely functional approaches to I/O are based on the insight that a program is a function which can be *applied* to its input data, and that any output is then simply the program's *result*. *Stream-processing* models [Landin65] were probably the earliest to abandon side-effecting I/O primitives. *Continuation* I/O [Karlsson81], coming along slightly later, in some ways seems to be a return to those side-effecting I/O primitives – its advantage however is that a program's I/O is forced to be single-threaded. Side-effects are *deterministic* rather than hidden and their sequence is guaranteed, even in lazy languages. *Monadic* I/O [PeytonJonesWadler93] is the newest model of all, and is most closely related to the continuation style.

Gordon's thesis [Gordon92] gives a fuller introduction to the subject, together with proofs of equivalence and a strong advocacy that monads are the most suitable model for the future. Here however, the following subsections describe and illustrate each of the I/O styles mentioned, in preparation for a treatment of embedded I/O.

2.1.1. Landin streams

Under Landin stream I/O [Landin65], a program is attached to one character-based device for input, and another for output. The type of a program is

```
type Program = [Char] -> [Char]
```

The virtual machine must provide the list of input characters from a keyboard or file, and direct the output list of characters to a screen or file. This model is a little restrictive, although it is adequate for many simple terminal-based applications. There are no synchronisation constraints between input and output.

2.1.2. Matched streams

Karlsson [Karlsson81] was the first to report using the technique of *matched* streams, later adopted very widely, and most notably in standard Haskell (version 1.2) [Hudak+92]. It is predicated on having a form of underlying virtual machine which processes messages. A program generates a stream of *requests* as output, receiving a stream of *responses* as input. That is, a program is of type `Dialogue` where

```
type Dialogue = [Response] -> [Request]
```

Each request is matched by exactly one response: the environment surrounding the functional language is the producer of responses and consumer of requests. The virtual machine is therefore of a corresponding type

```
type Machine = [Request] -> [Response]
```

although of course the machine cannot be written in the functional language itself. The requests are like operating system calls, and it is the programmer's responsibility to ensure that a program does not attempt to read the response from a request until the request has actually been sent!

This style of I/O programming is illustrated in Figure 2.1. The first request from `userprog` is to read a file, and subsequent requests are generated either by the program `nextact` or the program `exit`, depending on the value of the first response.

```
userprog :: Dialogue
userprog resps =
        ReadFile "filename" :
            (case (head resps) of
                 Str contents      = nextact contents (tail resps)
                 Failure ReadError = exit (tail resps)
            )

nextact :: String -> Dialogue
nextact str resps = WriteFile "new/name" str : []

exit :: Dialogue
exit resps = AppendChan stderr "Couldn't read file\n" : []
```

Figure 2.1: I/O: matched stream example.

The Haskell set of requests and responses is quite limited. It allows for reading, writing, and appending files, for reading and appending channels, for examining program arguments, and for reading and setting environment values and buffering characteristics. Landin streams can be simulated in the request/response model as shown in Figure 2.2.

```
landin :: ([Char] -> [Char]) -> Dialogue
landin prog resps = [ReadChan stdin, AppendChan stdout output]
    where output    = prog input
          Str input = head resps
```

Figure 2.2: I/O: Landin stream example.

2.1.3. Continuations

Haskell I/O may also be treated in a continuation style. This style has been proved to be equivalent in power to both the matched stream and Landin stream styles [HudakSundaresh88] [Gordon92]. Many implementations layer the continuation definitions over the basic matched streams, much as Karlsson who originated the style did [Karlsson81]. Other languages such as Hope+C [Perry91] have continuations as the basic I/O mechanism.

Looking again at Figure 2.1, it can be seen that if a program reads a series of files at different times, for each file it will duplicate the code pattern in Figure 2.1. The values of *nextact* and *exit* may change in each instance, but otherwise the code will be identical. It therefore makes sense to abstract this pattern and parameterise it. In general, a function is defined for every I/O request, taking at least one extra functional argument: the function to be evaluated next in sequence. Often a second continuation argument will be used to indicate an error recovery action.

Figure 2.3 shows the stream-based implementation of a continuation function *readfile* which issues a request to read a file and consumes the response. *userprog*, being just one instantiation of *readfile* with particular arguments, is consequently more compact and readable, with more obvious sequencing. The effort of constructing the request and pattern-matching the response is abstracted away from the point of call. This is the advantage of the continuation style over the stream style: the ''plumbing'' of responses to requests (ensuring they match) is largely taken out of the programmer's hands, removing one potential source of bugs. Because continuations and streams are so closely related, the standard set of I/O operations is identical in both styles in Haskell.

```
readfile name errorcont successcont resps =
    ReadFile name :
        (case (head resps) of
             Str file             = successcont file (tail resps)
             Failure ReadError = errorcont (tail resps)
        )

userprog = readfile "filename" exit nextact
```

Figure 2.3: I/O: continuation example.

2.1.4. Monads

It is suspected that monadic I/O is also equivalent in power to both streams and continuations [Gordon92]. Monadic I/O probably first arose in the development of KAOS [Cupitt89] [Turner87] as a refinement of I/O combinators suggested by Thompson [Thompson87]. Monads have since been championed more generally as a computational model by Moggi [Moggi89] and Wadler [Wadler90]. Their use for I/O [Peyton-JonesWadler93] has been widely adopted; there is a proposal that monadic I/O should be standard in Haskell version 1.3 [GordonHammond94].

The style relies on a new abstract type: the type of a computation which performs some I/O and produces a value.

```
data IO a
```

Any side-effecting I/O operations are provided as primitives with this new type as result type. There are two combinators that can be used with the new type: `result` lifts an ordinary value into the I/O type; and `bind` connects two I/O computations together in sequence, passing the value produced by the first computation forward into the second.

```
result :: a -> IO a
bind   :: IO a -> (a -> IO b) -> IO b
```

The example of Figures 2.1 and 2.3 is rewritten in monadic style in Figure 2.4. The operation `readfile` is assumed to be primitive. As in the continuation model the sequence is made clear, but here the specific combinator `bind` is required between the I/O actions. The end result type of any program is `IO ()`, because the program performs some I/O but returns nothing, represented by the unit value. In the example, the error case and its result (evaluation of `exit`) appears to have been omitted. In fact, it has merely been hidden by some cunning manipulation of the types. A later section on exception-handling (§2.4.4) reveals the details.

```
readfile :: String -> IO String
nextact  :: String -> IO ()
userprog :: IO ()

userprog = readfile "filename" `bind` nextact
```

Figure 2.4: I/O: monadic example.

2.2. Low-level I/O

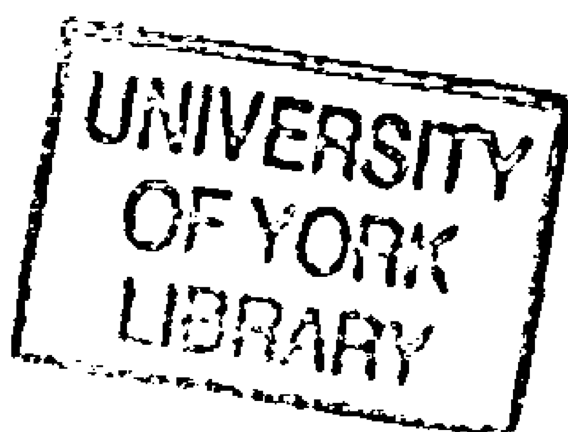
One reason why programming embedded applications in functional languages is difficult is the variety of possible hardware requirements in typical systems. It is essential that the programmer has source-level control over device driving. For this reason, an embedded system usually cannot use the standard set of requests, whether in stream, continuation, or monadic style. Files and channels are concepts at too high a level to have any direct utility.

2.2.1. Some approaches

Descriptions of existing functional embedded applications therefore often use specialised requests designed for specific hardware devices. For example, Cupitt, in programming the operating system KAOS [Cupitt90], uses requests `readDisc` and `writeDisc` which take and return disc IDs, block numbers, and blocks of text. We learn however that in the implementation, disc I/O is simulated in software, and the only actual disc accesses are at the very beginning and end of a KAOS session. Fijma and Udink report a case study of programming a robot arm to carry wooden blocks around an area which simulates a warehouse [FijmaUdink91]. Examples of their requests are `Task x y num` and `OutString cmd`; responses include `Ready`, `Time t` and `KeyStroke c`. The details of how, for instance, the robot arm interprets a `Task` request are not given. The virtual machine is assumed to implement these very application-specific details.

Erlang [Armstrong+93] is a functional language targetted at programming commercial embedded systems, namely telecommunications equipment. Requests may read data from and write data to *ports*, which produce and accept only byte-oriented values. A port may be either: (a) a file descriptor, handled as in Unix; or (b) a device, in which case the virtual machine must already know the name and characteristics of that device; or (c) an external program, written in any language. In this way, standard device drivers are provided built-in, but any unusual device drivers must be written separately in a language other than Erlang.

It should be clear from these examples that although adding new high-level requests for *specific* hardware is certainly possible, it is not very general. Lower level requests would be more general. For instance, a request to read a temperature gauge returning a tagged value is quite high-level and specific, whereas a request to read an input data register returning a simple byte is lower-level but can handle a more general range of hardware. The essence of using a programming language in an embedded system is that *every* part of the embedded system should be written in the language. To have to write parts of the *language*, not just the program, for every new application seems highly undesirable.


UNIVERSITY OF YORK LIBRARY

2.2.2. *New proposal*

This is why a new, but standard, set of requests is required. The most general way to approach embedded I/O is at the register level. No matter what particular hardware configuration is constructed, it will still be seen by the processor as a set of registers. The registers are addressed either in memory space, or in a separate I/O space, and the choice is purely dependent on the main CPU's processor family. The registers contain data which may be read or written, control values which may only be written, or status values which may only be read.

The simple proposal of this chapter is to adopt the following requests and responses, here expressed in matched stream style:

```
data DevRequest  = GetReg Addr
                 | PutReg Addr Word
data DevResponse = Register Word
                 | Success
```

The first request asks for the current contents of the register at the specified address, and receives a response which holds the word-value found there. The second request asks for the register at the specified address to be filled with the specified word-value, and receives an acknowledgement in response†.

2.3. High-level I/O

Of course, it would be tedious for a program to have to use register-level I/O requests throughout. Fortunately, since functional languages are specifically designed to enable abstraction, it is possible to package higher-level requests on top of the low-level ones.

Runciman demonstrates a technique [Runciman91] that uses low-level matched-stream requests similar to those outlined above. A device-driving function can be constructed which provides a high-level request interface to the user, but translates each request to manipulate words and registers.

```
devdriver :: ([HighResponse] -> [HighRequest])
                         -> [DevResponse] -> [DevRequest]
```

It takes as its argument a user program of type

```
userprog  :: [HighResponse] -> [HighRequest]
```

and produces as its result a program using the real, primitive requests.

```
main :: [DevResponse] -> [DevRequest]
main = devdriver userprog
```

† In Chapter 3 we will drop the requirement for each request to receive exactly one response: there, the acknowledgement is superfluous and will no longer be given.

Sinclair builds high level requests on top of Landin streams at the base level, in his work on functional operating systems [JonesSinclair89]. He presents functions which translate requests such as `ReadDisc` into lower-level activity. The proposal is that these functions take and return a number of arguments which are Landin streams, directly attached to hardware I/O registers. However, although registers are stated to be the sources and sinks of the extra data streams, no description is given of *how* specific registers can be attached to specific input and output streams.

Under either continuation or monadic style, the creation of high-level requests is simply accomplished by combination and abstraction. Many low-level I/O actions are sequenced together, and the resultant I/O action can be named and parameterised. For instance, a high-level action to write a string of characters to a terminal could be defined in monadic style as shown in Figure 2.5.

```
writeString :: String -> IO ()
writeString  []     = result ()
writeString (c:cs) = awaitTxReady 'bind' \() ->
                     putReg txbuffer (encode c) 'bind' \() ->
                     writeString cs
        where awaitTxReady :: IO ()
              awaitTxReady = getReg txstatus 'bind' \status ->
                             if status==ready then result ()
                             else awaitTxReady
```

Figure 2.5: I/O: example of abstraction.

2.4. Interrupt handling and exceptions

The remaining facet of I/O which we have not yet tackled is *interrupts*. The other languages mentioned above tend to neglect this area. KAOS [Cupitt90] sees an interrupt as an event which causes a program to terminate, but allows the program to "tidy up" a little before terminating. This is accomplished by *replacing* the evaluation of the program with the evaluation of a handler. Each program in a system is associated with its own unique message-wrapper, which is made available to other programs for the purpose of sending messages to it. However each program also has a unique "kill-wrapper", which sends a special message (an interrupt) to it, invoking its termination routine. The kill-wrapper can be used either by another program in the system or by a true interrupt.

In Fijma and Udink's case study [FijmaUdink91], the virtual machine converts interrupts into high-level responses, but in an *ad hoc* manner. For instance, `Time t` is a response which contains two sorts of information: both that an interrupt has occurred from the clock device, and also the value of the data register holding the current time. Erlang [Armstrong+93] has no explicit treatment of interrupts – a device might generate special data values to indicate an interrupt, but it depends entirely on the particular implementation of the port driver. Sinclair [JonesSinclair89] treats interrupts only in a brief suggestion

that the act of reading an input data register might block until an interrupt happens.

A full model of interrupt-handling is presented in Chapter 3, but the following sections review previous approaches to functional *exception-handling*, which covers a wider range of unpredictable but urgent conditions with which programs must deal, including error conditions. Errors are often signalled in a similar way to interrupts, and so the implementation mechanisms are often broadly similar too. Although error recovery is an important topic in its own right for embedded systems language designers, we review it here primarily for its bearing on the interrupt question.

2.4.1. Program-generated signals

The language ALEX [BretzEbert88] approaches exception handling as follows. The result of an expression can be a *signal*, which propagates outwards to the nearest *handler* within scope. For example, given the following definition of `f`:

```
f x = if x<0 then (signal bad x) + 5 else x
```

in a call of `f(-1)`, the exception `bad` can be handled in three ways:

```
handle bad by \y.0 terminate in f (-1)
```

states that the value of the expression should be 0 if an exception occurs;

```
handle bad by \y.-y retry in f (-1)
```

states that the expression should be re-evaluated again in its entirety, replacing the original argument to `f` by its negative (so the result is 1);

```
handle bad by \y.-y resume in f (-1)
```

states that evaluation should be resumed from the point of error, with the incorrect data value replaced *from now on* by its negative (so the result is 6).

Three possible semantics for exception handling are illustrated here: immediate termination; restarting the computation from the beginning with a corrected input value; and resuming the computation from where it left off, with a corrected input value. These constructs all have problems with referential transparency however. The language GERALD [Reeves+89] cleans up the opacity from ALEX, removing the `retry` semantics (which is particularly problematic), and introducing a syntactic device called *firewalling* which can be used to delimit the scope of `terminate` handlers in heavily-nested blocks of functions. A firewall specifies how much of an expression should be terminated in the event of an exception. GERALD also introduces a priority mechanism for signals – where two signals are at the same syntactic level and could potentially be raised simultaneously (depending otherwise only on the evaluation order of the expressions containing them), the priority distinguishes which handler should be invoked. Both ALEX and GERALD are lazy languages: Standard ML [Harper+86] is a strict language with a similar model of exceptions.

2.4.2. Environment-generated signals

A related interrupt mechanism is illustrated both by KAOS [Cupitt90] and by the version 1.3 I/O proposal for Haskell [GordonHammond94]. In these systems, an interrupt is raised as a signal from the enclosing operating system. The effect is that the currently-evaluating continuation or monad is removed and *replaced* by a pre-defined handler. The handler's job is to tidy up the internal state of the program before it terminates. Only termination semantics are provided – there is no way to retry or resume the program from the point at which the signal occurred.

Clack points out that the desired semantics of an interrupt is often not termination [Clack89]. He puts forward two important criteria for an interrupt-handling mechanism:

(i) response to an incoming signal must be *immediate*;

(ii) after the arrival of a signal, the program should be able to continue with its state augmented by the knowledge of the reception of the signal.

His solution assumes that communicating functional processes are supported, with dynamic process creation and asynchronous message-passing. Each process is divided into two components: the demand-driven component and the interrupt-handling component. Both components share the same process address, but the handler plucks signals out of its parent's incoming message stream leaving everything else behind. To communicate with the main component, it sends a message back into its parent's incoming stream.

Clack's interrupt model is closest in spirit to the one presented in Chapter 3, but is less elegant and less developed. It has not, to our knowledge, ever been implemented or used in a real system.

2.4.3. Continuation-based handlers

A completely different approach to exception handling is through the continuation style of programming, outlined earlier (§2.1.3). Exceptions can either be transmitted into the program from the environment as special input data, or can be generated within the program. Here, a function is given two extra functional arguments: these are the functions to be evaluated "next" in the normal case and in the error case. For example:

```
f x normal error = if cond then normal val
                   else error mesg
                   where (cond,val,mesg) = g x
```

The function `g` both evaluates an expression and tests for an error condition. It returns a tuple indicating success/failure, the value `val`, and an error message `mesg`. Assuming non-strict tupling, only one of `val` and `mesg` need be defined. The appropriate continuation function is applied to whichever was returned.

Chapter 3 uses a choice mechanism similar to this one in order to separate actions

associated with interrupts from actions associated with ordinary incoming data. The condition upon which the choice is made arises from the input source however, rather than on a computed condition.

2.4.4. A lifted datatype

Spivey presents a functional theory of exceptions [Spivey90] which also rejects the introduction of special constructs to the language. Instead, in each situation where an exception could be raised, the resultant type of the application is *lifted* to include a distinguished error value.

```
data Maybe a = Just a | Nothing
```

The price to pay is that all functions using exceptions must declare their types to be `Maybe` types. In return, it is easy for the programmer to test for error conditions and to propagate error values, with the help of higher order functions. Note that the `Maybe` construction is fully polymorphic, so a small library of higher-order handlers is sufficient to provide a comprehensive error facility to any program, as illustrated below.

This exception model is used extensively in the version 1.3 proposal for Haskell I/O, but is hidden inside the I/O monad. A more primitive level of I/O is assumed, called `PrimIO`. The `Maybe` type is augmented to the `Either` type which can carry an arbitrarily more complex error notification.

```
type Either a b = Wrong a | Right b
data IOError    = WriteError String | etc.
type IO a       = PrimIO (Either IOError a)
```

The monadic I/O combinators `result` and `bind` are implemented using more primitive combinators, but add in a little extra check to deal with the error case.

```
result x   = primResult (Right x)
x 'bind' y = x 'primBind' \cond ->
                case cond of
                    Right ans -> y ans
                    Wrong err -> failwith err
failwith :: IOError -> IO a
```

In this way, all error handling is conveniently provided in the standard libraries and the programmer is free to write code without explicit checks for error codes.

2.4.5. Failure as a list of successes

Wadler suggests that functional languages need not have any mechanism (either built-in or library-defined) for dealing with exceptions [Wadler85]. His proposal is that a function which might fail should instead be re-coded to return a list of values representing all possible answers. In the case where at most one answer is possible, the return value is either an empty list (indicating failure) or a singleton list containing the answer. So far this is equivalent to Spivey's scheme. However, Wadler's proposal is more general. A function

returning many possible answers does so lazily. If the consumer of answers decides to reject the first element of the list as being exceptional, the implementation ''backtracks'' and produces another answer (the next element of the list). Under lazy evaluation, only those answers actually consumed are ever evaluated, meaning that this scheme simulates exceptions and backtracking with no extra syntax and with no performance overhead.

The disadvantage is that the program has to be completely rewritten in two places: where the values are produced, and where they are consumed. At the very least, the simple meaning of the program is obscured to a degree because every type is ''lifted'' into a list type. It is also obscured because the test of exceptionality of any value must be coded explicitly into the program: an exception cannot be raised by the run-time system, only by a user-coded function – that is, unless every run-time primitive's implementation is also ''lifted'' to return a list.

2.5. Non-determinism

A further major issue which much ''real-world'' functional programming (including embedded systems programming) must address is *non-determinism.* The commonest source of non-determinism is temporal sequencing on two input sources, that is, when we want to choose whichever value ''came first''. This arises especially when communicating processes are introduced to a language and several message sources must be merged into a single input stream for each process.

Consider a non-deterministic choice `amb a b` [McCarthy63] which returns either `a` or `b` in the expression

```
divide x x   where x = amb a b
```

Operationally, `amb` might be understood as returning whichever of its arguments evaluated first, or whichever value was provided by the external world first. So the value of the expression is `a/a` or `b/b`, both of which reduce to `1`. Now if we substitute for `x` as the equational style permits us to, that is,

```
divide (amb a b) (amb a b)
```

there are the additional possibilities that the result could be `a/b` or `b/a`. Hence, the assumption of the equational style, that equals can be substituted for equals, is invalid for a non-deterministic operator. Sondergaard and Sestoft discuss how this simple construction can have at least 12 distinct semantic interpretations [SondergaardSestoft88a].

Henderson describes a variant of `amb` which interleaves, or *merges*, two lists into one, selecting output elements non-deterministically from the sources [Henderson82]. This construct is commonly used in the description of communicating processes, and `amb` can be written in terms of it.

```
amb a b = head (merge [a] [b])
```

It is generally accepted that a language which permits non-determinism in this manner cannot be regarded as purely functional. There have been several schemes proposed with the goal of regaining referential transparency, yet allowing some form of indeterminacy.

Friedman and Wise propose a new constructor called *frons* (rather than *cons*), which builds lazy multisets (rather than lists) [FriedmanWise80]. This is a cunning semantic side-stepping of the issue of non-determinism, since the difference between multisets and lists is that the former are unordered. The *amb* operator becomes the selection of the "first" element from a multiset of two elements. Because *frons* is lazy, the choice of output order is made at time of need rather than time of construction.

Hughes and O'Donnell also propose that the result of a call to *amb* should be a *set* of values rather than a choice between them [HughesODonnell89]. Their analysis is that referential transparency can be preserved provided that a function is thereafter *not able* to choose one value out of the set, like the delayed evaluation of *frons* does. Once a set of values has been created, that set cannot be deconstructed: every enclosing redex must use the set as a whole, or make it larger. The operation of set union is provided, but not set intersection. Some simple function applications must be replaced by a mapping of the function over a set, in order to keep the types correct. For instance:

```
divide {a,b} {a,b}
== mapset (/) ({a,b} X {a,b})
== {a/a, a/b, b/a, b/b}
```

The result of a program including some non-deterministic element is again a set of possible output values rather than a single value. In actuality however, only one of the possible values is required or indeed computed. It is as if the *choice* operation forbidden within the program has been lifted to a single use outside it. Practically, it means that the language implementation never holds sets of values – only one representative of each set. The set notation is used throughout the program however in order to emphasise the semantic interpretation which allows equational reasoning and referential transparency to hold. One drawback is that semantic proofs can only be partial with this model. That is, programs which definitely terminate cannot be distinguished from those which may not, but otherwise produce equivalent results.

Burton's suggestion is that *amb* should take a third argument of an abstract datatype, an *oracle* which records the sense of the choice [Burton88].

```
amb a b c
```

On the first call to Burton's *amb*, the oracle is empty, indicating that a choice should be made by the run-time system. On every call thereafter, the oracle is set, indicating that the expression should evaluate to the same answer as before. Hence equational reasoning holds because the whole expression always evaluates to the same result. Oracles are obtained from a system-provided infinite stream of oracles. The oracles are of an abstract type whose only operation is *amb*, therefore they cannot be read or defined by user code. However, the programmer retains responsibility for ensuring that a particular oracle is

never re-used in a different call of *amb*, and also that the same oracle is always used where the same choice is to be made. Two examples:

```
divide (amb a b c) (amb a b c)
```

always evaluates to *1*†, whereas

```
divide (amb a b c) (amb a b d)
```

can evaluate to *1*, *a/b*, or *b/a*.

Stoye, like Hughes and O'Donnell, also moves non-determinism to a single instance outside the functional language [Stoye86]. In developing communicating functional processes, Henderson [Henderson82], Jones and Sinclair [JonesSinclair89], and others use *merge* to connect the streams of messages from several processes' outputs to the single input stream of another. Stoye places this merging under the authority of a *sorting office* outside the language. All processes can assume the existence of this sorting office which undertakes to deliver all outgoing messages to the correct destination process. In this way, every process is entirely deterministic, although the behaviour of the set of processes as a whole is non-deterministic. Nevertheless, this has been the most popular solution to the difficulty in succeeding implementations of communicating processes [Turner87] [Cupitt90] [FijmaUdink91]. The earlier example recast under Stoye's scheme would involve one process receiving either *a* or *b*, whichever is delivered first from another process.

```
process P1 where P1 = recv ? \aorb -> divide aorb aorb
process P2 where P2 = send (to P1) a
process P3 where P3 = send (to P1) b
```

2.6. Communicating processes

This section reviews previous work in providing communicating processes within functional languages. Not only will processes be useful in developing an interrupt-handling model (Chapter 3), but they are a generally recognised technique for programming embedded systems. For instance, imperative languages specifically designed for embedded systems programming, such as Modula [Wirth77] and Ada [Ichbiah+83], support the definition of processes.

2.6.1. Models

Most embedded languages provide mechanisms to express concurrency, whether or not multiple processing units are available on the target machine. The reason is for modularity – to separate out different concerns into identifiable sections of the program. Operations to deal with devices are often most easily expressed as independent tasks, one for each device, with a means of communicating information from one task to another. For instance,

† Assuming a and b are not 0 of course.

an interrupt-driven keyboard handler is best written as a tight loop which places incoming characters into a buffer, rather than as a small section of a large loop which also services the screen and other devices. This enhances the re-usability of code, as well as performance and readability [BrinchHansen78].

Some languages, like occam [INMOS84], take a fine-grained parallel approach where every statement is a process in its own right. In the functional world, much research has been focused on achieving arbitrarily fine-grained parallelism by allocation of sub-expressions to processing units, a technique suggested by Kennaway and Sleep [KennawaySleep82] and since investigated by many others [ClackPeytonJones85] [PeytonJones89] [WalinskyBanerjee90]. Communication in this style is simply by reduction – the results of sub-expressions are joined by functional application.

Other languages, like Modula [Wirth77], use a coarse-grained model where a process is a form of procedure. Such a coarse-grained model can be supported in a functional language by making some selection of user-defined functions into processes. In imperative languages communication is served by two mechanisms, *shared memory* and *message-passing*. Shared memory communication takes the form of mutable variables, bounded buffers, semaphores, and so on. One process *changes the value of* the shared object, and another process observes this change. In functional languages shared references cannot serve the purposes of communication in the same way because it is not possible to change the value of a shared referent. Explicit communication between processes is usually achieved instead by stream-based message passing. Here, each process is a function which accepts a stream (lazy list) of in-coming messages as an argument, producing a stream of out-going messages as its result, rather like the Haskell I/O `Dialogue` type. However, often there is no underlying virtual machine: messages are consumed and produced entirely by other processes in the network. In some models, connections between coarse-grained processes are established by functional application [Henderson82] [JonesSinclair89]. In others, the processes are created by spawn operations or other mechanisms, and the connections are implicit [Stoye86] [Turner87] [Cupitt90].

The major difficulty with the schemes which connect processes by functional application is the introduction of non-determinism. A set of input messages from different sources must be merged into a single input stream for a process, although they are produced at different times and with different frequencies. Schemes which avoid the use of `merge` in the program code generally rely on Stoye's sorting office instead [Stoye86].

There are three further choices to be made when designing a process scheme:

- **Static *vs.* dynamic creation of processes.** This issue concerns whether the number of processes is determined at compile time, or whether processes can be created and deleted at run-time. Static process definition is not only easier to implement, but for certain sorts of embedded systems, such as hard real-time applications, it is essential in order to be able to guarantee timing properties. On the other hand, other

applications such as graphical user interfaces and operating systems absolutely require dynamic processes.

- **Static *vs.* dynamic communication channels.** Message-based communication can be by fixed connections between specific processes, like channels in occam. On the other hand, a generalised client/server model of interaction, such as that used by some device-drivers, needs a more flexible mechanism, and in particular the ability to respond to the sender of a message, no matter whom. Dynamically-managed message connections however require a more complicated addressing scheme and a sorting office for correct delivery.
- **Synchronous *vs.* asynchronous transactions.** Where dynamic connections are made between processes, both the sending and receiving process may have to be in an appropriate state for the message transaction to happen. Alternatively, if the sender does not block waiting for the receiver to collect the message, then the underlying system must provide buffered queues of messages.

2.6.2. Previous implementations

In the functional arena until recently, most applications of communicating processes had been to the development of operating systems. Several implementations have used non-deterministic constructions: Nebula [Karlsson81] allows dynamic processes with dynamic asynchronous connections; Henderson's range of operating systems [Henderson82] has static processes with static connections. Jones's operating systems [Jones84], like Henderson's, have static processes and connections; Sinclair extends Jones's work to dynamic processes with dynamic asynchronous connections [JonesSinclair89]. In all these schemes, a process network with two keyboards and two screens communicating with a single database could be defined using `merge`, `tag`, and `untag`:

```
program (kybd1,kybd2) = (scrn1,scrn2)
where scrn1 = untag 1 db
      scrn2 = untag 2 db
      db    = dbf (merge (tag 1 kybd1) (tag 2 kybd2))
```

The effective elimination of non-determinism from functional implementations of operating systems came with the sorting office [Stoye86]. Stoye's system has dynamic processes with dynamic asynchronous connections; Turner's design of KAOS [Turner87] builds on Stoye's model, but enforces synchronous connections; Cupitt's implementation of KAOS [Cupitt90] describes both synchronous and asynchronous dynamic connections with dynamic processes. The same example database program in these models can be defined without `merge`:

```
process Kybd Int where
    kybd n = readCmd ? \cmd -> send (to db) (n,cmd) >> kybd n
process Scrn Int where
    scrn n = recv ? \msg -> print msg >> scrn n
process Db  where
    db s = recv ? \(n,cmd) -> send (to (Scrn n)) string >> db t
         where (string,t) = dbf s cmd
```

Beyond the applications area of operating systems, some functional process implementations have been published more recently:

- Fijma and Udink describe a case study programming an embedded system [FijmaUdink91]. Their solution uses a static process network with dynamic asynchronous connections through a sorting office. The conclusion was that the relationship between the functional aspects and the process aspects of the system was not clear, and therefore that reasoning about the system as a whole had to be quite informal.
- The Erlang language [Armstrong+93] was designed for programming embedded systems. It provides dynamic processes with dynamic asynchronous connections, based on a spawn mechanism. Its message addressing scheme, although based on the sorting office, is not referentially transparent: one process address can refer to different processes at different times during the computation.
- Holyer and Carter propose a design extension to Haskell which allows dynamic processes but static connections [HolyerCarter93]. It guarantees referential transparency by allowing only a single writer on each fixed communication channel. Not only is each individual process deterministic, but the entire system of processes is too, unlike models using a sorting office.
- Jones and Hudak propose a different extension to Haskell which has dynamic processes and static connections [JonesHudak93]. Their model however is non-deterministic – it allows both an `amb`-like `fork` operator *and* multiple writers on channels.
- Concurrent ML [Reppy91] is a strict functional language with events as first-class objects. Processes, called *threads*, are spawned dynamically, and communicate over dynamically-created synchronous channels. This is commonly referred to as a *rendezvous* model. A channel rendezvous is of the *event* type, and because events are first-class, it can be named and passed as an argument to functions. This gives a sophisticated means of controlling the behaviour of processes and channels.

From these examples, it seems that no standard process model has yet arisen which fully satisfies the functional programming community, although there has been a recent resurgence of experimentation. Stoye's sorting office is a common link between many of the proposals, but it has not been universally adopted.

2.7. Summary

In summary, the need for input and output is a very prominent characteristic of embedded systems. There are numerous functional models of I/O, each of which is equivalent in power to the others, although some are more conveniently expressed than others. For embedded systems, it is essential to have I/O access to device registers, and this can be added to a run-time implementation of any of the styles. Having low-level I/O access does not however mean that programs need become swamped by I/O details – the mechanism of functional abstraction can be used to hide detail.

Several functional models for handling interrupts, exceptions, non-determinism, and processes have been proposed. None has gained complete acceptance, and experimentation is still common in each area. Chapter 3 proposes that the use of one model of processes and non-determinism can be a solution also to handling interrupts.

Chapter 3. Embedded functional I/O

Many embedded systems contain sub-systems which have a clear need to perform I/O entirely independently of other sub-systems. They are routinely expressed in many languages as a set of independent, but communicating, processes. The process abstraction adds readability and convenience to the language. This chapter presents a general means of handling interrupts from I/O devices, based on the use of communicating functional processes. It develops the process model within two different but equivalent frameworks – streams and continuations. Some details of the implementation of processes in Gofer are described, together with an evaluation of the performance of the resulting system.

Section 3.1 outlines the requirements of an interrupt scheme. Section 3.2 discusses some preliminary design considerations of the chosen scheme. Section 3.3 fleshes out the specific design in terms of a type model for processes in both stream and continuation styles. Section 3.4 describes the run-time implementation. Section 3.5 reports on Embedded Gofer's performance, and Section 3.6 concludes. A case study in the use of Embedded Gofer is given in Chapter 4.

3.1. Outline of interrupt scheme

Interrupts can be viewed as a form of input, rather than as a non-deterministic control structure. Extending the single-thread stream-based I/O model slightly, interrupts can be added into the *response* type.

```
data DevResponse = Interrupt
                 | Register Word
                 | Success
data DevRequest  = GetReg Addr
                 | PutReg Addr Word
```

However, if the one-response-for-one-request rule is retained, there has to be a new *request* for which the interrupt is the correct response. The effect of this extra request would be one of two things. Either the program blocks until an interrupt arrives, which excludes it from getting on with other possible activities; or (with an additional new response value to indicate the lack of an interrupt) the program polls to determine whether an interrupt has occurred since the last poll. The first option is undesirable because it wastes processing time, and the second misses the essential nature of interrupts: their immediacy.

If the parity rule is abandoned, then no request is needed before an interrupt can be received. A program awaits an interrupt by pattern-matching on its input stream. This means it must block until the interrupt happens. Again, this option wastes processing time.

A further factor to take into account is that a lazy functional program is driven by its need to generate output, and may choose to ignore its input entirely. Thus, interrupts as input data would in no sense *drive* the program.

However, a solution allowing interrupts to be handled more in the spirit of a non-deterministic control structure, yet still as input data, is possible. We introduce a network of processes, each of a type similar to the `Dialogue` just described. In this manner, while some processes are blocked awaiting interrupts other processes are free to proceed, and hence no processing time is lost.

Different interrupts are distinguished by associating one process with each interrupt source. Information is shared between processes by asynchronous message-passing. Each process is evaluated independently, and is written in such a way that an incoming interrupt or message is required before further output can be generated. This covers the issue of immediacy, because when an interrupt occurs, it can be handled immediately by a context switch to the appropriate handler. Hence, interrupts as input data can indeed drive the lazy functional program as a whole, because one interrupt may trigger the transfer of data to other parts of the program.

This signals a change from the usual intuition that lazy evaluation makes a program demand-driven rather than data-driven. The reason is that the program as a whole is not evaluated lazily: each constituent process is evaluated lazily, but it is dependent on receiving input generated by another process or by the external environment. This demand for input does not propagate beyond the demanding process however. Instead, each process is stalled by an unfulfilled demand for input. When the input data arrives, it therefore appears to drive the evaluation.

3.2. Design considerations

This section outlines the preliminary design of a scheme in which programs consist of multiple functional processes communicating by message-passing. For this design, based on the analysis in Chapter 2, processes are created statically, but connections (for asynchronous message-passing) are dynamic. These decisions, in particular the static creation of processes, were made with a view to future work on targetting a functional language at real-time applications. The process scheme is essentially a variant of Haskell I/O, implemented in the Gofer dialect. The modified language is called Embedded Gofer.

3.2.1. Communicating processes

Gofer's basic I/O model is like that of Haskell, where a program can be either of type `Dialogue` or of type `IO ()`. With either type, the program generates requests, receiving responses from the run-time system (RTS). In Embedded Gofer this is extended to allow

multiple processes of types similar to the present I/O types. Each process generates a stream of outgoing messages, and receives a stream of incoming messages, with the run-time system operating a sorting office [Stoye86]. Any process may send a message to any other process. The RTS is responsible for directing outgoing messages to the correct incoming stream. However the parity rule is relaxed to allow messages to be asynchronous.

An extension to the standard sorting office models the system's hardware state. Interaction with this part of the RTS looks like message-passing too. A device-handling process can communicate with hardware device I/O registers (demonstrated already in the `DevRequest/DevResponse` types), whereas a non-device process can communicate only with other processes. *Device processes* and *ordinary processes* are distinct classes†. It is expected (but not enforced) that ordinary processes should hold and maintain most shared program state separate from the hardware state. That is, device processes should manipulate hardware I/O registers without direct access to large amounts of software state, due to the demands of typical embedded systems. A device driver is likely to be a time-critical part of any system and cannot afford to deal with large chunks of state, particularly in a lazy language where unpredictable amounts of time may be needed to deal with it. The device driver should stick as closely to the I/O hardware as possible, doing as little as possible in the computational realm. This notional division is like that in Modula [Wirth77] between Device Modules and ordinary Modules.

3.2.2. I/O devices

A *pseudo-process* models hardware I/O registers. I/O device registers are often memory-mapped, although they sometimes inhabit a separate I/O addressing space distinct from the memory addresses. The RTS can easily ensure that the pseudo-process only accesses I/O registers rather than general registers. The pseudo-process can be seen as the source of interrupt messages sent to device processes. It can also be regarded as accepting messages to read or to write the contents of a register. When asked to read a register, it sends a message back to the requesting process with the contents. When asked to write a register, it fills the given value into the register.

There is an important issue of the *atomicity* of updates here. Suppose the programmer wishes to apply a function to the existing contents of a register, for instance to set a single bit, leaving the remaining bits in their original state. In our model this takes two requests, one to read the initial value of the register, and the second to write the updated value, leaving open the possibility that another process could be scheduled between the read and the write. Would it not be better to have a single request that supplies a function for the run-time system to apply to the register? Unfortunately there are other

† Information is shared between the two classes of process through proper message-passing.

considerations to take into account. An atomic update must be implemented at some level as a read followed by a write. However, I/O registers are frequently read-only or write-only. The action of reading a write-only register is undefined. The only means of knowing what value a write-only register contains at present is to keep a software shadow of it. It is not sufficient for the RTS to shadow all registers as a solution, because registers which are read-only must not be shadowed. Embedded Gofer leaves the job of shadowing write-only registers to the programmer. Under the current scheduling policy, explained in Section 3.4, there is no possibility of a second process being scheduled between adjacent reads and writes. Future extensions may address this problem further (see Section 3.6).

3.2.3. Interrupts

To handle interrupts, one device process is associated with each interrupt source. This association again mirrors Modula. An interrupt is a special message to the appropriate handling process. Since the evaluation of every process is independent, the interrupt handler can be executed immediately without interfering with any other process in the system. Shared sections of the program graph may cause one process to have part of its evaluation performed by another process, but this a performance gain, not a disruptive side effect. Priorities are assigned to the interrupt handlers, for the case when more than one interrupt occurs simultaneously. An interrupt message is clearly more important than ordinary incoming messages to the device driver. This can either be modelled as two queues, one for 'express delivery' interrupts, the other for ordinary messages; or as an ordering on a single queue.

3.2.4. Main programs

To glue everything together in this process scheme, the main program is an association list between process IDs and processes. Device processes are additionally associated with an interrupt vector address. The process table is thus *statically* defined, given that it is evaluated hyperstrictly. Process IDs are needed solely for addressing messages. The programmer must define datatypes for the process IDs and the messages – these two types are variables in the definition of the program type.

3.2.5. Synchronisation, laziness, and scheduling

In a lazy language, computation is demand-driven, where the initiating demand comes from the output driver. In Embedded Gofer, laziness is retained on a per process level – demand originates at the output stream of each process. Processes that are stalled waiting on incoming messages (including interrupts) are not *runnable*: they are *suspended*. All other processes are runnable. As runnable processes execute, they generate outgoing messages. These are delivered back to the appropriate incoming streams, perhaps causing

suspended processes to become runnable again. In synchronisation terms, a request to receive a message is blocking; a request to send a message *may be* blocking, depending on the relative priority of the addressee. In general, a process releases the processor on a message-send, but remains runnable itself. Section 3.4 explains the priority-based scheduling algorithm†.

Messages too are lazy. That is, their content need not be fully evaluated by the sender. Rather, the receiver evaluates the content of the message if it is needed. If two processes receive the same shared data value, only the first of them to need the value performs the evaluation. This choice may not be the best one in view of future plans to apply Embedded Gofer to real-time systems. Laziness in evaluating the content of a message makes the analysis of how much work is done by each process much more difficult. However, the discipline of evaluating every message before it is sent is a straightforward modification to the run-time system and would cause no significant semantic changes.

3.3. Programmer's view of process I/O

This section gives a Gofer source definition of how the process model fits into the language, by the introduction of new types, constructors, and continuations for programs and processes. It first describes processes in terms of stream I/O, as this is perhaps the most direct way of thinking about message-passing, for the reader unfamiliar with functional approaches to I/O. This has a corresponding RTS implementation for interpreted Gofer.

The subsequent re-formulation in continuation I/O terms has an RTS implementation for compiled Gofer. There are many practical reasons to prefer continuations over streams [HudakSundaresh88] but the continuation mechanism is likely to be unclear to the general reader without a prior appreciation of the stream-based model. Chapter 6 gives a further re-formulation of the same process model in terms of monadic I/O.

Both versions of Embedded Gofer's run-time system interpret the new stream constructors or continuations as instructions to take some action, such as delivering a message, updating an I/O register, and so on.

---

† It has been pointed out that the priority scheme described here permits *priority inversion*. For instance, when a high-priority process is awaiting a message from a low-priority process, it is possible that a middle-priority process will prevent the low-priority process from running, therefore indirectly preventing the high-priority process from running. A suggested remedy is *lazy inheritance*. That is, when the high-priority process demands an input message, the demand should propagate to the low-priority process, just as in lazy evaluation. The low-priority process temporarily inherits the priority of the demanding process, just until the demand is satisfied. Hence the high-priority process can continue to run, rather than being usurped by the middle-priority process. Unfortunately it is not clear how this might be implemented under the current design.

3.3.1. Types

Type synonyms for hardware-dependent types are as follows.

```
type Addr = Int
type Word = Char
```

A program is a list of process specifications. The program type is parameterised on two type variables: the programmer must define the process identifier type, and the message type.

```
data Main pid msg = Define [ProcSpec pid msg]
```

There are two kinds of process – ordinary tasks and interrupt-handling device processes. Both sorts of process are associated with unique identifiers. A device handling process is additionally associated with a particular interrupt vector address.

```
data ProcSpec pid msg = Process      pid (OrdProc pid msg)
                      | Handler Addr pid (DevProc pid msg)
```

The dialogue model of interaction applies to processes, although without the one-for-one parity rule.

```
type OrdProc pid msg = [OrdResp pid msg] -> [OrdReq pid msg]
type DevProc pid msg = [DevResp pid msg] -> [DevReq pid msg]
```

3.3.2. Request and Response constructors

For ordinary processes, user-defined messages (`Send`, `Recv`) carry a process identifier in addition to the message content. The identifier on an incoming message indicates the sender, and on an outgoing message indicates the intended receiver.

```
data OrdReq  pid msg = Send pid msg
data OrdResp pid msg = Recv pid msg
```

Device processes communicate not only with other processes (`DSend` and `DRecv`), but also with hardware I/O registers, to read (`GetReg`, returning `Register`) and write (`PutReg`) them.

```
data DevReq  pid msg = GetReg Addr
                     | PutReg Addr Word
                     | DSend  pid msg
data DevResp pid msg = Register Word
                     | Interrupt
                     | DRecv  pid msg
```

3.3.3. *Message parity rules*

The standard Haskell Dialogue model requires parity in I/O messages: for every request issued there must be one response given. This rule is relaxed in the process model of Embedded Gofer. In particular, a process need not issue a request before it receives an interrupt or message. Also, many incoming messages may be precipitated by sending one outgoing message. The rules shown in Table 3.1 govern the matching of incoming to outgoing messages.

Request	Number of responses	Specific responses
`PutReg`	0	–
`GetReg`	1 (immediate)	`Register`
`DSend`	any number ≥ 0	`DRecv/Interrupt`
`Send`	any number ≥ 0	`Recv`

Table 3.1: I/O requests and expected responses.

- `PutReg` does not expect any matching incoming message: the request cannot fail, so any response would be redundant.
- `DSend` and `Send` may cause any number of messages (including zero) to be returned, under the programmer's control. An `Interrupt` message may arrive at any time.
- `GetReg` expects exactly one message by immediate return, namely `Register`.

The notion of *express delivery* applies to interrupt and register messages. These messages are delivered ahead of all others which may be waiting on an incoming queue. The justification is that a process cannot search its incoming queue for particular messages, and it should not have to wait indefinitely for replies to `GetReg` requests.

- An `Interrupt` message overtakes all `DRecv` messages.
- A `Register` message overtakes all `DRecv` messages.

There is no contention between `Interrupt` and `Register` messages being pushed to the front of the queue for a device process, because once a device handler is running, it cannot be interrupted by that device again due to interrupt priority masking at the hardware level.

3.3.4. *Continuations*

An alternative, but semantically equivalent [HudakSundaresh88] [Gordon92], expression of I/O uses *continuation passing style* (CPS). A function is provided for each possible I/O action, which takes "the rest of the program" as one of its arguments. It performs the

action and then "continues" by evaluating the continuation argument. The advantage of this style is that responses are matched to requests automatically, neatly taking care of the parity rules set out above for streams. Indeed, the parity rules are made apparent in the types of continuations. In addition, the inherently sequential nature of I/O becomes more obvious.

The typical form of a CPS function for I/O is as follows. First, it generates the appropriate I/O request, then it reads some information out of the I/O response. Finally, the continuation is evaluated, using some or all of the information gathered from the I/O response.

Continuation functions for message-passing processes can be defined in terms of the stream constructors already given, as follows. In some of these CPS functions, either the generation of a request or the reading of a response is unnecessary, as indicated in the parity rules. First, the I/O operation in which a message is sent by an ordinary process:

```
send :: a -> b -> OrdProc a b -> OrdProc a b
send to msg cont inmsgs = Send to msg: cont inmsgs
```

Now consider operations in which a device process sends a message:

```
getReg :: Addr -> (Word -> DevProc a b) -> DevProc a b
getReg addr cont msgs = GetReg addr: cont word (tail msgs)
    where Register word = head msgs

putReg :: Addr -> Word -> DevProc a b -> DevProc a b
putReg addr val cont msgs = PutReg addr val: cont msgs

dsend :: a -> b -> DevProc a b -> DevProc a b
dsend to msg cont ms = DSend to msg: cont ms
```

Finally, there are I/O operations involving the receipt of messages. For ordinary processes, there is only one possible type of incoming message, but device processes must select between interrupt inputs and ordinary message inputs. Hence, `dselect` takes *two* continuation arguments.

```
getMsg :: ((a,b)->OrdProc a b) -> OrdProc a b
getMsg cont (Recv f m: msgs) = cont (f,m) msgs

dselect :: DevProc a b -> ((a,b)->DevProc a b) -> DevProc a b
dselect intcont msgcont (Interrupt: ms) = intcont ms
dselect intcont msgcont (DRecv f m: ms) = msgcont (f,m) ms
```

Often a device driver will respond purely to interrupts, or purely to messages. In these cases the `dselect` continuation can be more specific:

```
getInterrupt :: DevProc a b -> DevProc a b
getInterrupt intcont =
        dselect intcont (\(_,_) -> getInterrupt intcont)

getDMsg :: ((a,b)->DevProc a b) -> DevProc a b
getDMsg msgcont = dselect (getDMsg msgcont) msgcont
```

As with the constructor definitions given above, these CPS source definitions are intended for elucidation only. In practice, the continuation-style functions are built into Embedded Gofer as primitives. The examples in the Chapter 4 use CPS notation.

3.3.5. CPS combinators

In Embedded Gofer the symbols `($)` and `(?)` are often used to connect continuations together: `f $ a` is equivalent to the simple application `f a`, except that the former is right associative whilst the latter is left associative. Similarly, `f ? \v -> a` is a right associative combination, where the continuation on the right incorporates a lambda abstraction. The value passed forward from `f` is bound to the name `v` in `a`. The definitions of `($)` and `(?)` are as follows:

```
($) :: (a->b) -> a -> b
f $ a = f a

(?) :: ((a->c)->d) -> (a->b->c) -> b -> d
f ? g = \c -> f (\v -> g v c)
```

The use of both `($)` and `(?)` eliminates excessive bracketing, and they can be read as ''*and then*''. The higher-order function `seq` combines a list of continuations into a single continuation, using `($)`.

```
seq list cont = foldr ($) cont list
```

In embedded programming it is common to regard all processes as either *periodic* or *sporadic* – actions are repeated either at timed intervals or intermittently in response to events. This notion of repetition is encapsulated in the higher-order function `loop`, defined as the regular fixed point function.

```
loop :: (a -> a) -> a
loop action = action $ loop action
```

Unfortunately, `($)` cannot be used inside a loop as if it were a simple sequencing combinator. Its type is that of *application*, whereas the loop just defined requires a higher-order type for its constituent action. There are two ways to express this: either use a lambda abstraction together with `($)`s, or use functional *composition* `(.)` rather than application.

```
wrong === loop (action1 $
               action2)

right === loop (\cont ->
               action1 $
               action2 $
               cont)

right === loop (action1 .
               action2)
```

We choose to use the compositional style inside loops. However, with this choice it must be remembered that whereas `(.)` is used where one might expect `($)` inside a loop, `(.)` cannot be used *outside* a loop. On the other hand, the `(?)` connective may be used both inside and outside loops. The issue of which connective to use where can become rather confusing.

This confusion is a good argument for using monads rather than CPS: monadic actions may only be composed, not applied to each other. The monadic composition operator, *bind*, closely resembling the CPS `(?)` described above, is the sole basic combinator. Chapter 4's case study uses CPS as presented in the current description, but Chapter 5 gives further motivation for adopting monadic I/O. The case study in Chapter 6 proceeds to use this monadic I/O. For reference, the complete formulation of Embedded Gofer's process model in monadic I/O style is shown in Appendix A.

3.4. Some implementation details

Embedded Gofer has been implemented by modifying the Gofer [Jones94] functional language interpreter and compiler. The modified *interpreter* uses *stream* I/O primitives to implement the processes, and runs on a Sun 3/50 workstation with a simulation of I/O device registers, including a simulation of interrupts using the UNIX signal mechanism. The Embedded Gofer *compiler* uses *continuation* I/O primitives and, in addition to running in simulation, supports downloading of programs to a real single-board embedded system based on a 68010 processor. This section describes extensions to the Gofer compiler's RTS to create Embedded Gofer, and the mechanisms for running Embedded Gofer programs on the target embedded board.

3.4.1. Run-time evaluator modifications

The internal workings of Gofer are most fully described in [Jones94]. The G-machine model of graph reduction [Johnsson83] is used. This requires a stack, so to permit multiple processes the Embedded Gofer RTS has one stack per process. One change to the G-machine evaluator is needed: when an evaluation is stalled awaiting incoming messages or interrupts, the evaluator saves its state and returns control to the scheduling instance which called it. This is where Embedded Gofer departs from the usual view of

lazy evaluation: a demand for I/O may block, allowing evaluation to continue on a part of the program which may or may not satisfy that demand. The interaction of the scheduler and the evaluator is described further in Section 3.4.5.

The heap is common between all processes, since there is only a single processor and therefore no memory contention. There are several new RTS structures: most notable are the *process table*, the *interpreter* for `main`, the *I/O primitives* themselves, and the *scheduler.*

3.4.2. Building and interpreting the process table

The process table holds the following information about each process: its process ID; the heap location of the current root of its graph; the base and top of its local stack; booleans indicating the device/ordinary distinction and the runnable/suspended status; and the locations at which its incoming messages should be attached and detached. Device processes also need a boolean indicating whether an interrupt is pending. See Figure 3.1.

```
process :: record of
    pid            :: Cell
    graph          :: Cell
    localStackBase :: ptr to Cell
    localSp        :: StackPtr
    writeMsgs      :: Cell
    readMsgs       :: Cell
    msgCount       :: Int
    device         :: Bool
    runnable       :: Bool
    pending        :: Bool
```

Figure 3.1: Implementation: process table structure.

In a production-quality compiler, `main` would be evaluated at compile-time. This is made possible because in the current design the process table is static. However for ease of implementation, the process table is currently built from `main` at run-time. A mini-interpreter evaluates the list of process associations, filling in the process table as it goes. It also sets the state pointer to the location of the initial state value, and fills in a table of interrupt vectors. Once interpretation of `main` is complete, the evaluator is called on the highest priority process to start the system running. Figure 3.2 gives an outline of the evaluation of `main`, and Figure 3.3 demonstrates how the process table is filled.

```
evaluateMain(prog) =
    reset(processTable)
    case eval(prog) of
        Define processList ->
            repeat
                case eval(processList) of
                    Cons h t -> headspec    := h
                                processList := t
                                case eval(headspec) of
                                    Handler a p d -> newHandler(a,p,d)
                                    Process p d   -> newProcess(p,d)
                    Nil  -> scheduler()
        _ -> type_error("in main")
```

Figure 3.2: Implementation: evaluating `main`.

```
newHandler(intVector,thisPid,definition) =
    with processTable[index] do
        pid            := thisPid
        graph          := definition
        localStackBase := allocate(stackSize)
        localSp        := localStackBase
        writeMsgs      := (suspended :: Cell)
        readMsgs       := writeMsgs
        msgCount       := 0
        device         := True
        runnable       := True
        pending        := False
    setInterrupt(intVector,index)
    inc(index)
```

Figure 3.3: Implementation: building the process table.

3.4.3. I/O primitives

Gofer has a mechanism for declaring foreign language routines at the user-source level [Jones94]. This makes it straightforward to write new primitives in the underlying implementation, written in C, and then make them available to the Embedded Gofer programmer. For example, a prelude file contains a series of declarations similar to:

```
primitive send "primSend" ::
                    a -> b -> OrdProc a b -> OrdProc a b
```

The new primitives are simply supercombinators of the same form as those produced by the Gofer compiler itself. At run-time, they are called in exactly the same manner as any user function. The difference in effect is that they cause I/O actions, either through device registers or through message passing, and may also call the scheduler to enable other processes to run. Such I/O functions are *safe*, in the sense that they can be used at any point in a program where they are type-correct.

Message passing is achieved through a sorting office: the primitives for sending messages do a lookup in the process table for the PID matching the "receiver" field†, and deposit the message into a FIFO queue at the appropriate location indicated in the table. The current implementation of *lookup* searches the process table sequentially for PIDs, but this is quite inefficient. Because the PID datatype is known statically, a perfect hash function could be constructed at compile-time to enable fast lookup.

Figure 3.4 outlines the implementation of the *send* operation. The new message is attached to the receiver's FIFO queue – the attachment pointer, *writeMsgs*, is overwritten with a singleton list, after which the pointer is moved on to the (empty) tail. The receiver field of the message is replaced with the sender's PID.

Figure 3.5 outlines the *dselect* implementation in which a message is received after first checking for interrupts. If no messages or interrupts are available, the scheduler is called to allow other processes to run. Once the scheduler returns, we can be sure that no higher priority process is runnable, but we must check once again for interrupts and messages. The head of the FIFO message queue is held by the pointer *readMsgs*.

```
send(pid,msg,cont) =
    rcvr            := lookup(pid)
    rcvr.writeMsgs  := cons((self.pid,msg),suspended)
    rcvr.writeMsgs  := tail(rcvr.writeMsgs)
    inc(rcvr.msgCount)
    if rcvr<self then scheduler()
    eval(cont)
```

Figure 3.4: Implementation: sending a message.

```
primDSelect(intcont,msgcont) =
    if self.pending then
         self.pending := False
         eval(intcont)
    else if self.msgCount == 0 then
              scheduler()
              if self.pending then
                  self.pending := False
                  eval(intcont)
         if self.msgCount == 0 then
              suspend
         else self.graph     := apply(msgcont,head(self.readMsgs))
              self.readMsgs := tail(self.readMsgs)
              dec(self.msgCount)
              eval(self.graph)
```

Figure 3.5: Implementation: receiving an interrupt or message.

† We require that the PID datatype contains only first-order elements, that is, that two PIDs can be tested structurally for equality. In this simplified presentation, we also assume that a process exists for every possible value of the PID datatype.

Access to I/O registers for both reading and writing is again through new primitives, and includes a check that the given address is properly in I/O space, not in general memory.

3.4.4. Scheduler

There is a large literature on scheduling theory and schedulability analysis: see [Burns91] for a review of the field. Hard real-time systems are reactive and require a static priority algorithm for efficiency, and pre-emption for timeliness. Although some aspects of Embedded Gofer currently preclude its use in hard real-time applications, the process scheduler has the following rules based on a static priority pre-emptive algorithm [LiuLayland73].

- Processes have three possible states: *running*, *runnable*, or *suspended.*
- All processes begin in the runnable state.
- Only one process at a time can be running because there is only one processor.
- The choice of which runnable process gets the processor is by static priority: the ordering is defined by the list order of `main`, highest first.
- A running process becomes suspended when it is awaiting an incoming message.
- A running process can be de-scheduled, but remains runnable, when any higher priority process becomes runnable.
- A suspended process becomes runnable when either (a) a message is delivered to it, or (b) it is a device process and an interrupt is delivered to it.

```
procStack :: array [1..numProcess] of process
procSp    :: 1..numProcess

saveContext =
    self.localSp       := sp
    procStack[procSp] := self
    inc(procSp)

contextSwitch(x) =
    self      := x
    stackBase := self.localStackBase
    sp        := self.localSp
    eval(self.graph)

restoreContext =
    dec(procSp)
    self      := procStack[procSp]
    stackBase := self.localStackBase
    sp        := self.localSp
```

Figure 3.6: Implementation: context switching.

This scheduling algorithm is implemented through calls within the primitive I/O continuations and uses a stack for storing the context of pre-empted processes (see Figure 3.6). At every point when a process could be pre-empted, the run-time system polls the scheduler, which proceeds to check the status of all higher priority processes. If any is runnable, the current process is stacked, otherwise it continues evaluation. When a process suspends, the scheduler first checks the status of all higher priority processes, then descends through the lower priorities until it reaches the process at the top of the pre-empted stack. Figure 3.7 outlines the scheduler code.

```
scheduler() =
    basePriority := index(self)
    saveContext()
    if interrupted then contextSwitch(interrupted)
    for index := 1 to basePriority
        with processTable[index] do
            if pending or msgCount>0 or runnable then
                contextSwitch(processTable[index])
    restoreContext()
```

Figure 3.7: Implementation: the scheduler.

Interrupts must be treated carefully because memory allocation, incorporating incremental garbage collection, is a critical region. A process would be corrupted badly if it were pre-empted during allocation of a heap memory cell. In fact all interrupts are trapped through a single handler which sets a global flag (*lastInterrupt*), checked by the scheduler at the next safe moment when pre-emption can occur. The flag is then used as an index to the appropriate programmed handler. The interrupt pre-handler and code to fill in the interrupt vector table are outlined in Figure 3.8.

```
setInterrupt(vector,index) =
    vectorTable[vector] := index
    setPreHandler(vector,preInterrupt)

preInterrupt(vector) =
    intProc     := vectorTable[vector]
    interrupted := processTable[intProc]
    interrupted.pending := True
```

Figure 3.8: Implementation: interrupt handling.

There are one or two obvious optimisations to this scheduling scheme. For instance, when a message is delivered downwards, the scheduling call can be avoided altogether, as is the case in Figure 3.4 in the line:

```
if rcvr<self then scheduler()
```

Similarly, when a message is being delivered upwards it is safe to jump straight to the higher-priority recipient without polling. However, care must be taken not to introduce priority inversion through this optimisation. Once the recipient has blocked, the scheduler must ensure that any processes with priority *between* the sender and receiver have the

chance to run. To implement this optimisation, the following section of code replaces the line just quoted:

```
if rcvr<self then
    saveContext()
    contextSwitch(rcvr)
    scheduler()
    restoreContext()
```

3.4.5. Target embedded board

The target embedded board is a custom-built single-board microcomputer, based around the Motorola 68010 central processor. Additional plug-in interface cards provide logic to control each specific application's hardware. The target board is connected to the host computer (a workstation) by two serial lines. One of these is used for downloading programs, and either can be used for program input or output.

The board contains 768kb of RAM and 32kb of EPROM. The EPROM contains only a simple monitor program which recognises a control sequence on one of the terminal lines and then receives a program in Motorola S-record format. This is translated to op-codes and stored in RAM. Once the S-records have finished loading, the monitor jumps straight to the start instruction of the program in RAM. Transmitting S-records is slow, so we use a second-stage downloader called "zippy load". This is a short program in S-record format. Once it is loaded and running on the embedded board, we transmit a real test program in standard binary `a.out` format. The zippy loader stores the program directly into a fresh section of RAM and jumps to its start instruction once loading is complete.

3.5. Results

This section reports on the performance of Embedded Gofer. The following figures are based on compiled Gofer, running the marble-sorter program of Chapter 4 in simulation on a Sun 3/50. Simulation has been used purely because it is the most convenient way to gather figures: the real embedded board has no tracing or profiling mechanisms.

Table 3.2 demonstrates the cost of various components of the RTS: the memory allocator, the evaluator, the scheduler, and I/O primitives. The cost of sending messages is higher than that of receiving, because the sorting office does a linear search for the recipient. A re-implementation using a hashing search would be more efficient. Memory allocation is very expensive because each allocation incorporates a call to an incremental garbage collector (see Chapter 7). Other similar functional languages also incur a high GC cost, as table 3.3 shows.

system component	av. exec. time per call (μs)	% of total run-time
mem. allocation/GC	250	38.0
evaluator	110	11.2
scheduler	90	0.6
`send`	140	<0.5
`recv`	40	<0.5
`dsend`	100	<0.5
`dselect`	50	<0.5
`getReg`	40	<0.5
`updReg`	30	<0.5

Table 3.2: Comparative cost of RTS routines.

Table 3.3 gives performance figures for Concurrent ML [Reppy91] (running on a 20MHz Sparc 1) and Erlang [Armstrong+92]. Concurrent ML and Erlang are both languages which are "mostly functional" and also have multiple processes and scheduling. The GC overhead in all three languages is high. In CML a context switch is analogous to a call to the Embedded Gofer scheduler, and the rendezvous is analogous to a message send and receive. It can be seen that Embedded Gofer's system overheads are within an order of magnitude of CML's performance, despite running on a much slower processor.

system component	av. exec. time per call (μs)	% of total run-time
Erlang mem.allocation/GC	-	25
CML mem. allocation/GC	-	14-30
CML context switch	26	-
CML rendezvous	94	-

Table 3.3: Costs in other systems.

Finally, Table 3.4 shows average costs for a sample of predefined functions in Gofer for further comparison. The cost of message-passing is broadly comparable with the cost of simple arithmetic functions and boolean tests.

function	av. exec. time per call (μs)
integer *	60
integer /	110
integer +	80
boolean ==	20
boolean <=	110

Table 3.4: Sample of times for primitive applications.

3.6. Conclusions

Processes give functional languages the ability to handle interrupts cleanly. In an embedded systems context, they also separate device drivers both from each other and from the rest of the program. The present design of Embedded Gofer is flexible enough, and its implementation efficient enough, to be used in programming prototype systems (see Chapter 4).

Some of the design choices in Embedded Gofer are open to experimentation. For instance non-real-time applications, such as operating systems [Cuppitt90] or graphical user interfaces [CarlssonHallgren93], generally require dynamic creation and deletion of processes. The language could be adapted to provide such dynamic facilities (though it is possible to simulate dynamic processes within the existing static process scheme). As a second example, an enforcement of strictness on message contents would allow clearer analysis of the work done by each process. As a third example, to allow one process to be associated with more than one interrupt source could be useful in certain circumstances, although no need for this was found in the case studies in Chapters 4 and 6.

Embedded Gofer could be enhanced in several ways. For instance, register addresses and word values are currently implemented as type synonyms: Abstract Data Types would be better. Perhaps too the register address type could be partitioned by the read/write properties of registers. Real-time programming might be accommodated by introducing timing primitives and/or higher-order operators to express the common ideas of periodicity, delays, deadlines, timeouts and so on: Chapter 8 discusses this issue further.

Embedded processes follow a loop-based pattern. Compilation techniques could exploit this fact to produce better code. Whereas functional language implementations are often criticised for lack of efficiency, the repeating nature of processes should permit the code generator, memory allocator, and garbage collector to make significant assumptions about run-time behaviour, perhaps leading to substantial performance gains. One such gain would be the ability to allocate local memory which can be updated in-place. Another is the guarantee of tail-recursion.

There are two specific deficiencies to be addressed in the present design of Embedded Gofer.

1. The programmer must set out the priorities of processes by explicit ordering. Tools already exist within the real-time community to analyse a set of processes to determine priority automatically, for instance using a deadline-monotonic scheme. In particular, the use of annotations on periodic processes denoting their period, and on sporadic processes denoting their maximum frequency would enable the use of these tools during the compilation of an Embedded Gofer program. Chapter 8 has more to say about this possibility.

2. Message-passing in the present model is insecure: there is one datatype encompassing all messages sent or received by all processes. It is tiresome to write every process with an exhaustive `case` comparison on every message received, so inevitably the programmer makes assumptions about which messages ''should'' be received. If these assumptions are wrong, the program fails at run-time because a process cannot match a particular message. In an ideal model, the message datatype should be subdivided by process, so that each process can send or receive only certain messages. Any other message to or from that process should constitute a static type error. This gives much greater security at run-time, by signalling certain sorts of programmer error at compile time. A scheme to specify and enforce such a discipline, based on the use of constructor classes [Jones93] is given in Chapter 5.

Chapter 4. Case study I: a marble sorter

This chapter describes a case study of a particular embedded system, illustrating the techniques presented in Chapter 3. The case study involves device driver programming for a particular embedded system: a marble-sorting chute controlled from a Motorola 68010-based single-board computer. This apparatus is used at York for undergraduate teaching in real-time and embedded systems programming. In the past, a procedural language such as Modula, C, or Ada has been used for the exercise; now Embedded Gofer can do the same job.

The marble-sorter shown in Figure 4.1 is a chute, at the top of which is a hopper containing a mixture of metal and glass marbles. Two solenoids can be operated in alternation to release marbles singly down the slope. Halfway down the chute are two detectors. One generates a signal when a metal object passes; the other signals when a light beam is broken. At the bottom of the slope a hinged arm can sweep from side to side under the control of two more solenoids, directing marbles into one of two collection bins. The exercise is to release marbles at a constant rate and to sort them correctly into separate bins for metal and glass. The controller board has connections to a terminal screen and keyboard: a full solution to the exercise should control these also, providing a user interface for starting and stopping operation and altering the release rate.

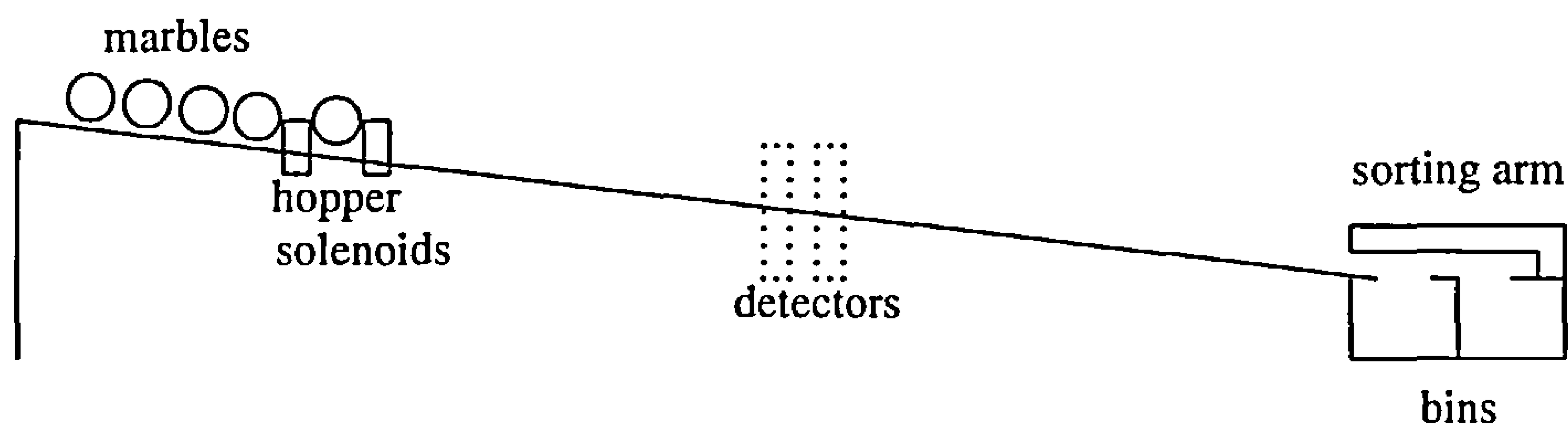


Figure 4.1: Marble-sorting apparatus.

The time constraints on the problem as are follows: marbles can be released at intervals shorter than the time it takes them to travel the slope. Hence, several marbles may pass the detectors before the first of them reaches the sorter arm. The program must implement a precise delay between each detection and its related sorting action, and keep a queue of these delays to model the marbles currently in motion between the detector and the sorter.

4.1. Dividing the program into processes

Embedded Gofer requires a separate process for each interrupt source. There are three here: the metal detector, the light-beam detector, and the on-board clock/timer. Other processes are needed to release marbles, to sort them, to drive the screen, to read the keyboard, and to provide a neat user interface. Hence the datatype for process identifiers in Figure 4.2.

```
data Pid = MetalDetect
         | LightDetect
         | Clock
         | HopperRelease
         | SorterArm
         | Screen
         | Keyboard
         | UserInterface
```

Figure 4.2: Marble-sorter: processes.

Ensuing sections illustrate device drivers for the hopper release mechanism, the marble detection mechanism, the sorter mechanism, and the terminal line. The clock/timer mechanism is presented at the end, as it is the most complex driver. For now, we can assume that it implements two operations: `wait`, and `defer`. The former suspends a process for a precise time; the latter delays the sending of a message for a precise time. Their types are as follows:

```
wait  :: Time -> DevProc Pid Msg -> DevProc Pid Msg
defer :: Time -> Pid -> Msg -> DevProc Pid Msg -> DevProc Pid Msg
```

4.2. The hopper mechanism

The first problem is releasing marbles from the hopper (Figure 4.3). The time it takes for the mechanism to collect a single marble in preparation for release is a constant. The hopper process also needs to know the rate at which to release them: for now let this rate be a program constant; eventually the user interface process should send messages specifying release rates. As with all devices, the hopper registers need some initialisation before periodic activity can start.

```
pacr, padr, paddr :: Addr
hopper_mode, data_dir, open, collect :: Word
min_collect_time, release_rate :: Time

pacr = pit_base_addr + 0x1a      -- example register address value
hopper_mode = 0xf8               -- example control word value

hopper_driver :: DevProc Pid Msg
hopper_driver =
    putReg pacr hopper_mode $
    putReg paddr data_dir $
    loop (putReg padr collect .
          wait min_collect_time .
          putReg padr open .
          wait release_rate)
```

Figure 4.3: Marble-sorter: hopper mechanism.

4.3. Detectors

There are two detectors generating interrupts in the marble-sorting system: the light beam detector triggers when a marble of any sort passes; the metal detector triggers only when a metal marble passes. The program must distinguish glass from metal. The best way to do so is to "ignore" interrupts from the metal detector (Figure 4.4). When the light beam detector is triggered by a metal marble, a status register holds a value indicating that the metal detector device is still awaiting service. At this point both interrupts are dismissed. If the status register indicates that the metal detector is not awaiting service, then it is a glass marble, and there is only a single interrupt to dismiss.

```
metal_mask, dismiss_m_and_l, dismiss_just_l :: Word
psr :: Addr                          -- status register
&   :: Word -> Word -> Bool          -- bit-wise 'and' masking + test
travel_time :: Time                  -- a constant

metal_detect :: DevProc Pid Msg
metal_detect =
    loop getInterrupt    -- i.e. do nothing in response to intrpts

light_detect :: DevProc Pid Msg
light_detect =
    init_int_device $
    loop (getInterrupt .
          getReg psr  ? \status_val ->
          if status_val & metal_mask then
               putReg psr dismiss_m_and_l .
               defer travel_time SorterArm MetalMarbleComing
          else putReg psr dismiss_just_l .
               defer travel_time SorterArm GlassMarbleComing)
```

Figure 4.4: Marble-sorter: detectors.

4.4. Sorter arm

The sorter arm process (Figure 4.5) is straightforward, given that the clock/timer is responsible for ensuring that it receives sorting messages at exactly the right moment.

```
pbcr, pbdr, pbddr :: Addr
sorter_mode, data_dir, move_left, move_right :: Word

sorter :: DevProc Pid Msg
sorter =
    putReg pbcr sorter_mode $
    putReg pbddr data_dir $
    loop (getDMsg ? \(Clock,msg) ->
          case msg of
              MetalMarbleComing -> putReg pbdr move_left
              GlassMarbleComing -> putReg pbdr move_right)
```

Figure 4.5: Marble-sorter: sorter mechanism.

4.5. Terminal-line driver

Terminal line drivers come in many flavours. This section illustrates one very simple style of interaction only. A more complicated terminal line driver is illustrated in Appendix C. First, the usual register and control-word declarations are shown in Figure 4.6.

```
txdr, txsr, rxdr, rxsr, dgmr, dssr :: Addr

tx_enable, rx_enable, baud_rate,
duplex_mode, ready_mask, ready_reset :: Word
char_to_word :: Char -> Word
word_to_char :: Word -> Char
```

Figure 4.6: Marble-sorter: terminal-line declarations.

To transmit a character, poll the status register until it signals ready, then place the character into the data register, and finally reset the ready signal in the status register (Figure 4.7).

```
transmit :: Char -> DevProc a b -> DevProc a b
transmit c cont =
    poll_loop $
    putReg txdr (char_to_word c) $
    putReg txsr ready_reset $
    cont
    where poll_loop cont = getReg txsr ? \val ->
                           if val & ready_mask then cont
                           else poll_loop cont
```

Figure 4.7: Marble-sorter: transmitting a character.

Transmitting a string is as simple as mapping over a list (Figure 4.8). The screen driver process is sporadic, triggered by the arrival of messages from the user interface process (Figure 4.9).

```
transmits :: String -> DevProc a b -> DevProc a b
transmits str = seq (map transmit str)
```

Figure 4.8: Marble-sorter: transmitting a string.

```
screen_driver :: DevProc Pid Msg
screen_driver =
    putReg dgmr duplex_mode $
    putReg dssr baud_rate   $
    putReg txsr tx_enable   $
    loop (getDMsg ? \(UserInterface, Output str) -> transmits str)
```

Figure 4.9: Marble-sorter: screen driver.

Receiving characters from the keyboard could be done under interrupt control, but it

is more interesting to consider a polled model. The keyboard process can be written as if it polls continuously (Figure 4.10). If this process has the lowest priority in the system, it in fact polls only when the processor is otherwise idle: it is a "background" activity. For the moment, we ignore the possibility of losing characters if other processes in the system keep the processor busy. Input characters must be buffered until they are requested by the user interface process. This requires an intermediate process to hold the buffer. Here, a list models the buffer, and it is held in reverse order for efficiency reasons. A queue structure would perhaps be more appropriate: recent research on efficient implementations of functional queues has significantly simplified their coding [Okasaki95].

```
keyboard_driver :: DevProc Pid Msg
keyboard_driver =
    putReg rxsr rx_enable $
    loop (poll_loop   .
          getReg rxdr ? \wval ->
          dsend Buffer (Ch (word_to_char wval)))
    where poll_loop cont = getReg rxsr ? \val ->
                           if val & ready_mask then cont
                           else poll_loop cont

buffer :: [Char] -> OrdProc Pid Msg
buffer str =
    getMsg ? \(from,msg) ->
    case from of
        KeyboardDriver  -> let Ch c = msg in buffer (c:str)
        UserInterface   -> send UserInterface (Input str) $
                           buffer []
```

Figure 4.10: Marble-sorter: background keyboard driver.

To overcome the possibility of losing characters, the driver can be forced to poll at a fixed rate rather than as a background activity (Figure 4.11).

```
poll_rate :: Time          -- a constant

keyboard_driver =
    putReg rxsr rx_enable $
    defer poll_rate Keyboard WakeUp $
    loop (getDMsg ? \(Clock,WakeUp) ->
          getReg rxsr ? \val ->
          (if val & ready_mask then
                getReg rxdr ? \wval ->
                dsend Buffer (Ch (word_to_char wval))
           else id) .
          defer poll_rate Keyboard Wakeup)
```

Figure 4.11: Marble-sorter: polling keyboard driver.

4.6. The clock/timer

Assume that the clock device is set to interrupt at a precise rate, e.g. at 20Hz. The details of timer initialisation are shown (in monadic style) in Appendix C. An *alarm* process can be coded using the client/server idea. When a client process wishes to delay for a specified time, it sends a request to the alarm server, which inserts the request into a queue. Likewise, when a client wishes to defer sending a message for a specified time. On each interrupt from the timer device, the delay component of every request in the queue is decremented. When the delay component of the first request in the queue reaches zero, the alarm sends a message signifying that the delay has expired. The message goes either to the original client or to the deferred receiver of the original client's message.

One formulation of an alarm process (Figure 4.12) keeps a queue of triples containing time-remaining, PID, and message. It decrements the time remaining for every item when an interrupt occurs, or inserts a new triple when a message arrives. There is only one sort of incoming message needed for the alarm, because a process delay can be implemented simply as a special deferred message – one that says "wake up the original client".

```
alarm :: [(Int,Pid,Msg)] -> DevProc Pid Msg
alarm q =
    dselect -- first, how to deal with interrupts
              (let (ready,keep) = span (\(t,_,_) -> t==1) q in
              putReg tcr dismiss $
              seq (map (\(_,p,m) -> dsend p m) ready) $
              alarm (map (\(t,p,m) -> (t-1,p,m)) keep))
            -- and second, how to deal with messages
              (\(_, Delay t p m) ->
              let (ltq,geq) = span (\(t',_,_) -> t'<t) q in
              alarm (ltq++(t,p,m):geq))
```

Figure 4.12: Marble-sorter: inefficient alarm.

This formulation is rather inefficient because it requires two traversals of the queue on either branch. Also, lazy evaluation leads to the work of these traversals building up as large closures. An alternative approach (Figure 4.13) avoids the decrements: instead of recording the delay relative to the moment a request arrives, record its delay relative to its neighbour in the queue. On every timer tick decrement only the first item of the queue. When the first item reaches zero, respond to it and any following it which are also zero.

Defined this way, `alarm` traverses the queue at most once for each message or interrupt. Decrements do not build up as lazy closures, because evaluation of `act_on` forces the decrement. Insertion into the queue is still lazy, but in practice this version of `alarm` performs adequately and the closure build-up does not cause a problem.

```
alarm q =
    dselect (putReg tcr dismiss $ act_on (dec q))
            (\(_, Delay t p m) -> alarm (insert t p m q))
    where dec []                = []
          dec ((t,p,m):qs)      = (t-1,p,m):qs
          insert t p m []       = (t,p,m): []
          insert 0 p m q        = (0,p,m): q
          insert t p m ((t',p',m'):qs)
                | t'>=t         = (t,p,m): (t'-t,p',m'): qs
                | t>t'          = (t',p',m'): insert (t-t') p m qs
          act_on ((0,p,m):qs) = dsend p m $ act_on qs
          act_on q            = alarm q
```

Figure 4.13: Marble-sorter: efficient alarm.

The hopper process in Figure 4.3 assumed a *wait* continuation to express a delay of a specified time. Using the alarm process just described, *wait* can be written as in Figure 4.14.

```
ticks :: Time -> Int

wait :: Time -> DevProc Pid Msg -> DevProc Pid Msg
wait t cont =
    dsend Clock (Delay (ticks t) HopperRelease Wakeup) $
    getDMsg ? \(Clock,WakeUp) ->
    cont
```

Figure 4.14: Marble-sorter: *wait* operation.

The detector process in Figure 4.4 assumed a *defer* continuation to insert a specified time interval between sending a message to the sorter process, and the sorter actually receiving it. Again, the alarm server provides the mechanism for this, illustrated in Figure 4.15.

```
defer :: Time -> Pid -> Msg -> DevProc Pid Msg -> DevProc Pid Msg
defer t p m = dsend Clock (Delay (ticks t) p m)
```

Figure 4.15: Marble-sorter: *defer* operation.

Two forms of synchronisation behaviour are illustrated by *wait* and *defer*. The former forces the calling process to cease executing and await a synchronisation condition. The latter allows the calling process to continue executing, having created a synchronisation condition for a different process.

4.7. Bringing it all together

To complete the marble-sorter program, it only remains to tie up a few loose ends. The *Buffer* PID should be added to the *Pid* datatype. The *Msg* datatype should include all the messages introduced along the way (Figure 4.16).

```
data Msg = Delay Int Pid Msg     -- alarm messages
         | WakeUp
         | MetalMarbleComing     -- sorter messages
         | GlassMarbleComing
         | Output String         -- terminal messages
         | Input  String
         | Ch     Char
```

Figure 4.16: Marble-sorter: message datatype.

The definition of *main* (Figure 4.17) includes the initial local states of those processes which require them:

```
metal_int_vector, light_int_vector, clock_int_vector :: Addr

main :: Main Pid Msg
main = Define
         [Handler clock_int_vector Clock         (alarm []),
          Handler light_int_vector LightDetect   light_detect,
          Handler metal_int_vector MetalDetect   metal_detect,
          Handler 0                HopperRelease hopper_driver,
          Handler 0                SorterArm     sorter_driver,
          Handler 0                Screen        screen_driver,
          Process                  Buffer        (buffer []),
          Process                  UserInterface (ui arg1 arg2),
          Handler 0                Keyboard      keyboard_driver]
```

Figure 4.17: Marble-sorter: main program.

4.8. Performance

The marble-sorter program sorts accurately at release rates of up to one marble every 0.30s – at least equal to solutions in C, Ada, and Modula. To do this, the system has to deal with up to 19 interrupts every second. Interrupts at greater than 20Hz are beyond the capacity of the present program. However, the Gofer compiler is partly to blame – even evaluation of compiled code is partly interpretive, and hence has a poor level of performance. Were the process model to be adopted in a production Haskell compiler, far better performance could be expected. Even in Gofer some optimisations, such as in-lining common functions, are yet possible.

4.9. Conclusions

One criticism easily levelled at the CPS formulation of Embedded Gofer is that parts of programs look very similar to sequential imperative language statements: simply read semicolons instead of dollars. Even the *loop* statement is there. But why expect anything different? After all, I/O is essentially sequential and imperative. The CPS notation (and its monadic cousin) are well suited to expressing sequentiality; Embedded Gofer however

offers all the benefits of modern functional languages, in addition to this convenient I/O mechanism.

That said however, for the simple example presented in this chapter, the functional program does not differ substantially from previous solutions written in imperative languages, either in size, level of abstraction, or algorithmic complexity. Nevertheless, the alarm server process shows a degree of sophistication giving it application well beyond this example. Chapter 6 describes another case study where Embedded Gofer is used to program a much more demanding system. The alarm server used in the marble sorter is re-used in the larger system, without significant modification. Additionally, the indications there are that the complete functional program is about two thirds the size of the corresponding source code in Modula, or three times smaller than the code in C.

On the positive side, in the present study, the functional approach has reduced any memory-management duties associated with the device drivers, making their action clearer. It has also slightly simplified the algorithmic components by allowing their specification to be expressed at a higher level of abstraction.

In summary, this case study has demonstrated that Embedded Gofer offers within a functional language a general means to control specific devices for I/O. Most importantly, the I/O mechanism provides for timely response to interrupts. There is a reasonably efficient prototype implementation, in which an embedded-systems exercise devised independently by others has been programmed.

Chapter 5. Type-checked message passing

5.1. Introduction

It was noted at the end of Chapter 3 that the process model so far presented has no mechanism for type-checking message traffic. This chapter describes the use of type classes [WadlerBlott88] and constructor classes [Jones93] to provide static type-checking in this area, with examples.

5.1.1. The problem

Consider a set of functional processes. In order for the processes to communicate with each other, they each have an address, and each can send messages to other addresses through a sorting office, which is part of the run-time support system. Each process therefore generates a stream of address-message pairs as output, and receives a stream of messages as input.

```
type Process = [Message] -> [(Address,Message)]
```

This is essentially the scheme Stoye presented [Stoye86]. The typing suffers a major deficiency: there is only one message type, which is global to the entire set of processes. In operation, each process is likely to deal with only a subset of all possible messages. This leads to two things. First, processes can detect ''wrong'' message inputs only at run-time. Secondly, a local change to one process's set of ''valid'' messages may necessitate a global change in the `Message` type, and hence recompilation of other processes. For these reasons, it is difficult to write *re-usable* processes, suitable for inclusion in libraries. The global message type tends to constrain processes to rely too much on the particular context of the program in which they were originally written.

5.1.2. Some previous solutions

Some refinements to the type scheme above have been proposed. Stoye himself recognised a possible way forward. The first insight is that all messages received by a particular process should be of the same type, but distinct processes may receive different types of message. In other words, a process can be *parameterised on its input message type.* However, a process can *send* messages containing data items of different types, provided that each one is of an appropriate type for its receiver.

```
type Process imsg = [imsg] -> [(Address,Message)]
```

But now what is the type of a *Message*? It must still encompass all the messages in a particular program. Even if this type could be formulated, perhaps as a tagged union, it would be too general. It should not be possible to send an integer to a character-receiving process, or a character to a boolean-receiving process. The process address ought to constrain the type of message that can be attached to it. Stoye states that the usual Hindley-Milner type system is unable to describe the *Message* type. He posits that an *existential* type system would be of benefit.

Under such a scheme, rather than the type *Address*, there would be a type generator *AddressFor x*, being the type of process address to which items of type *x* can be sent.

```
type Process imsg = [imsg] -> [∃omsg.(AddressFor omsg, omsg)]
```

MacQueen et al. describe a system in which this type is valid [MacQueen+84], but at the time of Stoye's investigation, no automatic checker existed for the system. As a result, Stoye's implementation of functional processes sidestepped type-checking altogether in numerous places. Recent developments in type systems do allow the specification and checking of existential types, but only to a limited degree [Augustsson93].

Turner points out that *Message* could be an *abstract* type, whose representation is known only to the run-time system [Turner87]. He suggests that a series of *wrapper* functions be created, each of which casts a value of some type to a *Message*:

```
type Wrapper omsg = omsg -> Message
type Process imsg = [imsg] -> [Message]
```

Each process in the system is associated with its own individual wrapper function, provided by the run-time system on the process's creation. A process can send a message to another only by using the *receiver's* wrapper: this neatly avoids any need for addresses at all. The wrapper can be viewed as exactly an address function. The scheme relies on dynamic process creation, because the only way for a process to get hold of another's wrapper is if a parent/child relationship holds between them. This is due to Turner's insistence that messages contain only *data* values: a wrapper cannot be sent inside a message, because it is a function.

The actual mechanism for creating and passing wrappers is a *fork* function. This works just like the UNIX mechanism of the same name. It takes two continuation arguments, the first of which is executed as the parent, and the second as the child.

```
fork :: (Wrapper c->Process p) -> (Wrapper c->Process c) -> Process p
```

Both the parent and child are given the new child's wrapper, which can easily be bound to a name for future reference. The child in addition inherits the name bindings of

the parent, and hence has access to the parent's wrapper.

Turner's scheme works, but has two drawbacks. First, dynamic process creation may not be desirable, e.g. hard real-time systems require a static process net. Secondly, the wrapper functions are locally bound to each process, which increases the amount of local state they have to manage. The number of wrapper arguments to a process can get very large if it intends to send messages to many other processes!

5.1.3. A new solution

An addressing function, like Turner's wrapper, ensures that items are sent only to a process which can receive messages of that type. This chapter shows how the overloading permitted by type classes [WadlerBlott88] can give every such addressing function the *same name*. For this reason the addressing functions are defined globally, relying on the resolution of overloading rather than on scope rules to ensure that the correct function is used locally at run-time. One advantage is that program clutter is reduced, because the processes do not have to manage a local namespace for the wrappers. The approach also allows static creation of processes.

Additionally, the use of constructor classes [Jones93] gives the ability to create *specialised* classes of process: either flat or layered sets of I/O privileges can be granted to a process by the judicious use of class contexts.

The following section re-formulates the problem in terms of recent ideas about functional I/O. Section 5.3 describes how overloading can be used to make the provision of addressing functions less ad hoc. Section 5.4 builds specialisation on top of this process mechanism, and Section 5.5 concludes.

5.2. Modern functional I/O

5.2.1. Monads

For the remainder of the chapter we adopt a *monadic* approach to the expression of I/O, rather than the *synchronised stream* approach used by Stoye, or the *continuations* used by Turner. Monadic I/O is semantically equivalent to both stream and continuation I/O, but arguably provides a cleaner syntax [PeytonJonesWadler93]. Additionally, the later use of constructor classes in describing I/O depends on the monadic treatment. It is worthwhile rehearsing the basic ideas of monadic I/O here, before we describe how message traffic may be type-checked.

The definitions below follow Jones's characterisation of monads in Gofer [Jones93].

His treatment of monads relies on constructor classes (provided in Gofer version 2.30 [Jones94]); a simpler type class approach would be sufficient for the part of our addressing mechanism which uses overloading, but the advanced facilities of constructor classes are necessary for the part dealing with process specialisation. For an introduction to standard type classes, see [WadlerBlott88], [HudakFasel92], or [Jones92].

Some parts of a program can perform I/O actions. A monad encapsulates the actions, ensuring that they are sequenced in a single-threaded manner. This is very similar to the sequencing enforced by the continuation passing style of I/O. The monad additionally ensures that exactly one value is bound forward from each action; for instance, in the following program fragment using the usual I/O monad [PeytonJonesWadler93], a value representing the contents of a file is passed forward from the `readFile` action, and the `writeFile` action passes forward `()`, representing success.

```
readFile  :: Filename -> IO String
writeFile :: Filename -> String -> IO ()

main = readFile  "/foo/bar"           `bind` \contents ->
       writeFile "/rab/oof" contents  `bind` \() ->
       etc
```

I/O actions are combined together using the `bind` operator whose result type is an action. Simple values can be lifted into the monad using the `result` operator. The `Functor` class context simply states that a `Monad` constructor must have a `map` function already defined over it. For further details on the `Functor` class see [Jones93].

```
class Functor m => Monad m where
    result :: a -> m a
    bind   :: m a -> (a -> m b) -> m b
```

5.2.2. *Processes*

A process is simply an I/O action, made up of a combination of smaller actions. However, we define a new type for processes: `Action`. It differs from the standard I/O monad because it is parameterised on the type of message it expects to receive, as well as by the type of value it will pass on. Although the message type does not appear in the concrete representation of `Action`s, it plays an important role in type-checking. For instance, there are occasions where the type of value to pass on must be exactly the message type.

```
data Action imsg val = ST (World -> (val,World))
instance Monad (Action imsg) where
    result x         = ST (\w -> (x,w))
    (ST f) `bind` g  = ST (\w -> let (x,w') = f w
                                     (ST h) = g x
                                 in h w')
```

We can use the class system to describe the basic operations which characterise a process: sending and receiving messages. Later, we shall see how to extend this characterisation to include file I/O and other interaction with the real world.

```
class Monad (act imsg) => Process act imsg where
    send :: Address -> omsg -> act imsg Bool
    recv :: act imsg imsg
```

This says that a monadic type, such as `Action imsg`, constitutes a process type if there are operations `send` and `recv` defined, of the appropriate types to send and receive messages. A message *of any type* can be sent to an `Address`, passing forward within the sending process an indication of whether it could be delivered. A message can be delivered only if it has the specific message type allowed in the receiving process's type specification; the input message value is passed forward within the receiving process. `Action imsg` is declared to be an instance of this class by associating the names `send` and `recv` with primitive implementations in the run-time system.

```
instance Process Action imsg where
    send = primsend
    recv = primrecv
```

So far we have simply described the situation as it existed in Stoye's paper, but in a monadic framework. The following section describes how the class system can be used to ensure that the `send` operation only sends a message to the address of a process capable of receiving that message type.

5.3. Overloading expresses addressing

5.3.1. Constraints on addresses

At the implementation level, the sorting office requires both an address and a message in order to be able to deliver that message. (The syntax for introducing a primitive function here is Gofer-specific.)

```
primitive primsend "sendmsg" :: Address -> omsg -> Action imsg Bool
```

At the level of the source program however, we must disallow processes from making the attachment between address and message. As in Turner's scheme, a process sends just the message, and some *other* part of the run-time system works out what the address should be. But here, a class provides the necessary link between messages and the addresses they can be sent to:

```
class AddressFor omsg where
    address :: omsg -> Address
```

An appropriate context constraint on the *send* operation is that the message type must belong to this class. That is, when *send* is invoked, it can send a message of any type *provided that an address exists* for that message type.

```
class Monad (act imsg) => Process act imsg where
    send :: AddressFor omsg => omsg -> act imsg Bool
    recv :: act imsg imsg
```

It is still the case that although an address is guaranteed to exist for every message accepted by this type schema, that address may not be associated with a process. For example, in a dynamic process network, there must always be "spare" addresses available which can be attached to new processes as they are created. It is for this reason that *send* still passes on an indication of whether delivery was possible or not. Where processes are created statically however, successful delivery can be guaranteed through additional static analysis, in which case the boolean report is redundant. The latter is the case in Embedded Gofer.

5.3.2. *Using overloading*

Now, we need to ensure that the appropriate address is actually attached to each message as it is sent. The declaration of *Action imsg* as an instance of the *Process* class is where this happens.

```
instance Process Action imsg where
    send x = primsend (address x) x
    recv   = primrecv
```

This says that when a message is sent, the underlying sorting office mechanism is given both an address and a message. The class system provides function-name overloading which guarantees that the right addressing function is used in every actual call.

5.3.3. *Examples*

In any particular program, an instance of the *address* function must be declared for every message type used. If every process receives a distinct message type, this is simple. For example, with the following definition of *Address*, there are three addressing functions.

```
data Address = Characters | Booleans | Integers
instance AddressFor Char where address = const Characters
instance AddressFor Bool where address = const Booleans
instance AddressFor Int  where address = const Integers
```

When any process calls, say, *send False*, the value *False* will be delivered correctly to the process *Booleans*.

However, there might be more than one process able to receive messages of a particular type. In this case, the content of the message must be sufficient to distinguish the address for delivery. For example, with the following enumerated address type, every process expects to receive characters. The value of each character message is used to determine which process receives it. A call to `send 'M'` would deliver the character `M` to the process `UpperAlpha`, whilst `send '*'` would deliver to the process `Other` if it exists.

```
data Address = UpperAlpha | LowerAlpha | Numeric | Other
instance AddressFor Char where
    address c | 'A' <= c && c <= 'Z'    = UpperAlpha
              | 'a' <= c && c <= 'z'    = LowerAlpha
              | '0' <= c && c <= '9'    = Numeric
              | otherwise               = Other
```

As a final example, consider a functional program to control two liftshafts. The code for each liftcar process is identical, dealing with exactly the same type of messages, but each copy of the process has a different address. Again, the content of the message must distinguish which process receives it. Here, the request to `GoToFloor B 4`, generated by the `B` set of buttons is delivered to `Lift B`. The request having been serviced, `DoneFloor B 4` is delivered back to `Buttons B`. This example is a much simplified fragment from the case study given in Chapter 6.

```
data AB = A | B
data Address = Lift AB | Buttons AB
data Request = GoToFloor AB Int
data Service = DoneFloor AB Int
instance AddressFor Request where
    address (GoToFloor ab n) = Lift ab
instance AddressFor Service where
    address (DoneFloor ab n) = Buttons ab
```

5.4. Specialised classes of process

5.4.1. Flat specialisation

Having provided a class which defines the basic process I/O operations, it is possible to go on to define more specialised classes of process. For instance, although it suffices for most processes in a system to communicate solely with other processes, there may be a need for some processes to communicate in addition with an external file system.

```
data FilingReport = OK | Error String

class Process act imsg => FileProcess act imsg where
    readFile   :: Filename -> act imsg (FilingReport,String)
    writeFile  :: Filename -> String -> act imsg FilingReport
    appendFile :: Filename -> String -> act imsg FilingReport
```

A different class of process may be required to interact with terminal screens:

```
class Process act imsg => TermProcess act imsg where
    putChar :: Char -> act imsg ()
    getChar :: act imsg Char
```

With these *two* specialised *classes* of process, we now have the possibility of *four* different *types* of process: one which handles files; one which handles a terminal; one which handles both; and one which handles neither. The advantage gained is protection from programmer errors. For instance, if a process is specified to interact with a terminal but not with files, then a programming error in which files are accessed by that process is caught as a static type error. This form of process specialisation should however be used with care, because with n different classes, there are 2^n different possible types.

5.4.2. *Layered specialisation*

In addition to flat specialisation, *hierarchies* of subclasses can be layered on top of the basic process class. Each inherits all the operations of its superclasses, whilst adding some new operations. This technique is used in Embedded Gofer, to allow two layers of specialisation in defining device drivers. The first layer permits access to device I/O registers.

```
class Process act imsg => DevProcess act imsg where
    getReg :: RegAddr -> act imsg Word
    putReg :: RegAddr -> Word -> act imsg ()
```

The second layer allows a process to receive interrupts as well as messages, provided that it already has device access.

```
class DevProcess act imsg => IntrptProcess act imsg where
    select :: (Interrupt->act imsg val) ->
                    (imsg->act imsg val) ->
                            act imsg val
```

In effect, this provides a *privilege* mechanism on I/O. The type system can infer the level of privilege for each process by examining what operations are used in its definition. If the programmer does not agree with the inferred class constraint, this signals a static error.

5.5. Conclusions

We have shown that recent developments in the technology of type systems can bring improved type security to communicating functional processes.

- Parameterisation allows different processes to receive messages of different types.

- The overloading afforded by type classes makes it straightforward for a sending process to address outgoing messages and be sure that the message is of the correct type for the receiver.
- Overloading also eliminates any need to store the addresses in the local state of the process, and is therefore suitable for a system in which processes are statically created.
- Elimination both of a global message type and of the local creation of addresses simplifies the re-use of processes. A process and its associated addressing function can be a self-contained unit suitable for separate compilation and storage in a library.
- Constructor classes enable specialised sets of I/O operations and hierarchies of privilege to be expressed. (Thrift is called for here, however, because of the potential for an explosion in the number of possible process types.)

In summary, the class facilities provided by Gofer, especially constructor classes, have been found very useful in characterising certain sorts of I/O behaviour. The scheme described here† has been applied to example programs in Embedded Gofer giving gains in security and error detection. The case study of Chapter 4 is re-expressed in Appendix B, using message type-checking and the monadic I/O style. Chapter 6 describes a further case study using this scheme.

† With slight changes and further parameterisation: see Appendix A for a full description of Embedded Gofer's definition of type-checked monadic I/O.

Chapter 6. Case study II: a model liftshaft

This chapter documents the experience of programming another system, somewhat larger than the marble-sorter of Chapter 4, using Embedded Gofer. Section 6.1 describes the application, a model liftshaft, and the embedded computing resources available for its solution. Section 6.2 outlines the method of attack on the problem. Section 6.3 briefly presents some monadic I/O combinators to be used in the solution. Section 6.4 describes the timer process, going into greater depth of discussion than Chapter 4. Section 6.5 details particular problems encountered in programming the remaining device drivers. Section 6.6 looks at interrupts in the liftshaft system. Section 6.7 proceeds to build a first-cut control algorithm over the device drivers already programmed. Section 6.8 describes and discusses the full control algorithm. Finally, Section 6.9 draws some conclusions on whether the functional approach lends itself to the task of device-driving, and especially on whether the programmer gains anything useful by using a functional language as opposed to the standard device-driving languages; assembler, C, Modula, or Ada.

6.1. Apparatus

The liftshaft apparatus was originally built as a test-bed for students' real-time programming experiments, initially in Modula and later in Ada. It was designed to provide a range of critical and non-critical demands, with a scheduling problem of moderate complexity, and with some scope for ergonomic design [Freeman+82].

The model liftshaft is approximately 4'6" high. There are two cars built of meccano, each of which has a corresponding counterweight. As shown in Figure 6.1, for each car, a string of fixed length is attached to the roof of the construction in two places – above the car and above the counterweight. The car and the counterweight are suspended on this string by pulley wheels. When the counterweight is at the top, the car is at the bottom and vice versa. The middle section of the string passes through a series of pulley wheels above the roof of the unit. One of the wheels is driven directly by a low-voltage bi-directional motor, with a maximum speed of 60 r.p.m.; another is connected to a precisely-milled telemetry wheel and optical detection system; the third wheel provides tensioning.

The liftshaft as a whole is shown in Figure 6.2. Four end-stop switches provide a fail-safe mechanism for the cars. When a car reaches the top or bottom of the shaft, it trips one of these switches which automatically cuts the power to the motor†.

The two shafts are sectioned into six floors, with a platform for each floor. There is

† This not only stops attempts to launch cars across the room, but also reduces wear and tear on the suspending strings and the motors.

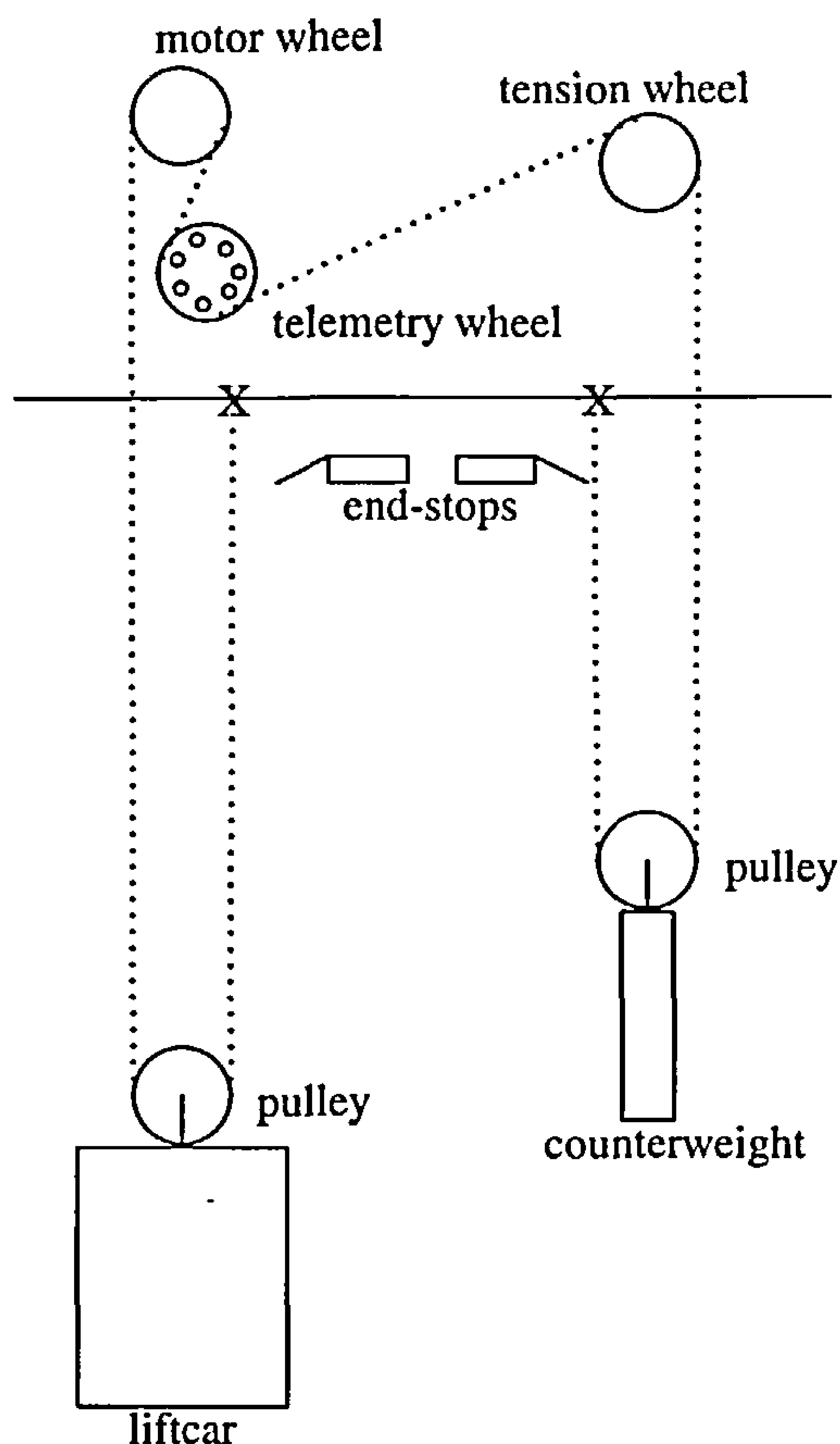


Figure 6.1: Liftshaft: car mechanism.

one call button marked ''Up'' on all floors except the highest, and one marked ''Down'' on all floors except the lowest. The call buttons are located between the two shafts, and each has an internal indicator lamp which can be lit under software control.

Each lift-car has six request buttons, one per floor. These buttons do not have internal lamps. Each car also has a seven-segment LED display and a loudspeaker capable of making a ''bleep'' noise.

The liftshaft is controlled using a custom-built single-board microcomputer, based around the Motorola 68010 central processor. Two linked custom-built interface cards provide some extra control devices and hold logic to translate control signals into motor voltages and so on.

Two Motorola 68230 Parallel Interface/Timer (PIT) chips are connected to the motor, lights, telemetry wheels, end-stops, and shaft buttons. One Motorola 68681 Dual Asynchronous Receiver/Transmitter (DUART) connects to the host computer via two serial lines. These lines are used to download programs and display program output. A second 68681 DUART connects to the liftcar buttons, seven-segment displays, and bleeps.

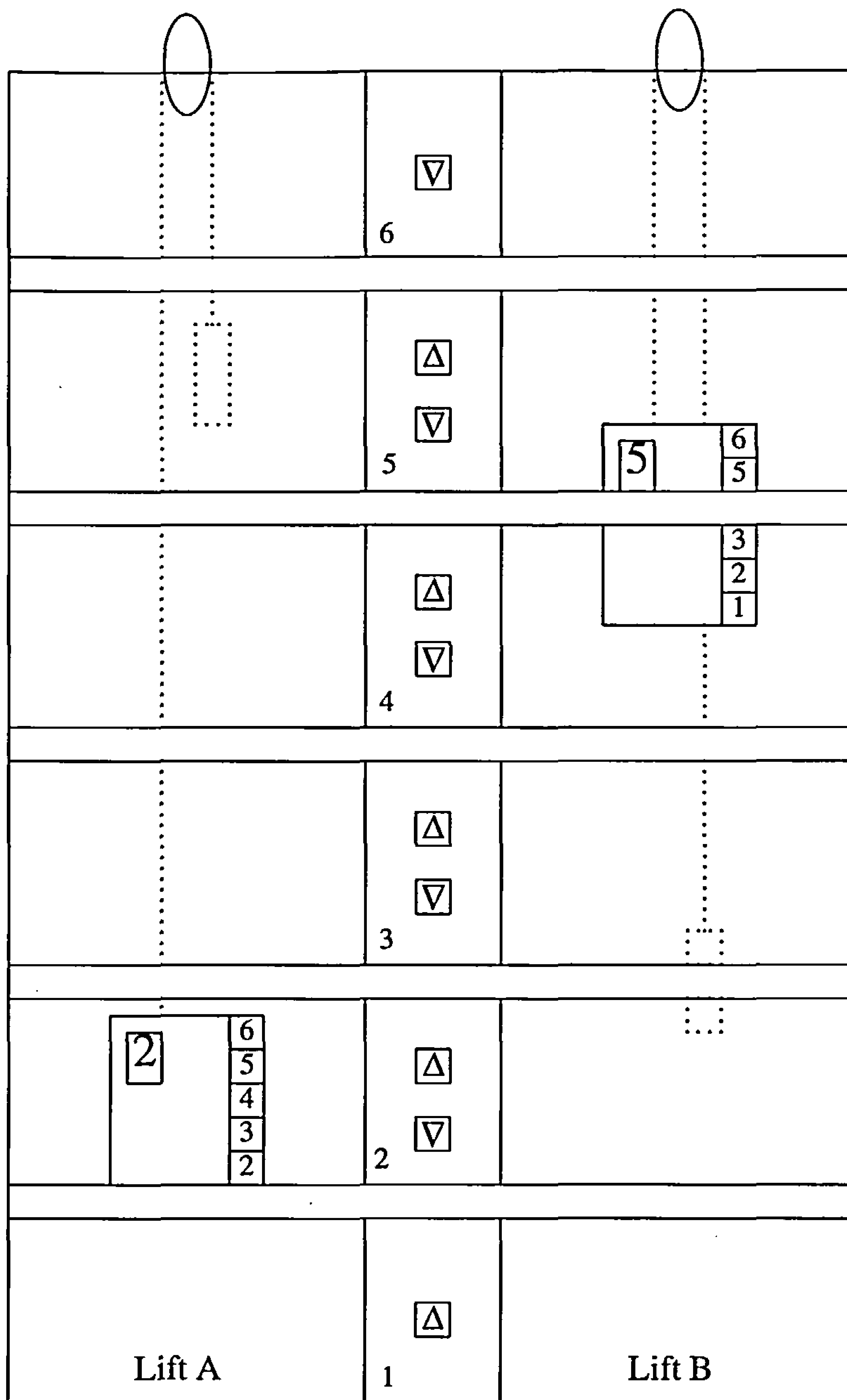


Figure 6.2: Liftshaft: shaft layout.

6.2. Method

The overall requirement for the programming exercise is to control the model hardware to simulate the "expected" behaviour of a liftshaft. The behaviour of the liftshaft as a whole involves many independent strands of I/O activity. One possible model of the system divides the program into at least six processes:

- two interrupt drivers, one for each lift-car, handling positioning by telemetry;
- two software drivers, scheduling each car's movement and visual display;
- one device driver, interacting with all the system's buttons, recording requests;
- and a clock driver handling all timing requirements.

Under this model, the final `main` program defines a process table as in Figure 6.23 (presented towards the end of the chapter, following discussion of each of the component processes).

A number of decision algorithms for liftshaft scheduling have been published, for instance, in the Erlang book [Armstrong+93] Chapter 11, pp.162-168. However, since the algorithm to decide which lifts to send to which floors is not trivial, let us attack the problem from a lower level, starting with the device controllers and some simple scheduling algorithms, then building up in complexity towards a full solution. A series of graded exercises is presented, illustrating successively more involved behaviours.

Section 6.3 briefly outlines how the monadic I/O facilities of Embedded Gofer differ from the stream and CPS facilities given in Chapter 3, and derives some monadic combinators to be used in programming the case study.

Section 6.4 programs a general timing server for the system involving a mixture of basic behaviours. Each basic behaviour is examined in turn before the complete server is described.

Section 6.5 examines the problem of shared control registers. The lift motors and call-buttons are attached to the *same* parallel I/O registers, yet ideally the devices should be programmed in *separate* processes. In addition, some write-only registers must be shadowed in software so that they can be read too. At this point, a program can solve the first exercise, to move the lifts up and down without stopping, detecting when they reach the end-stops and reversing their direction when they do so. In addition, the program can light up a shaft call lamp when the corresponding button is pressed, then extinguish it again after a fixed period of time.

Section 6.6 adds interrupt control. The telemetry wheels cut light beams and so cause interrupts at fixed displacements. The second exercise is to use the telemetry to cause the lifts to stop at every floor on their way up and down. As they stop, the appropriate shaft call lamp is extinguished if it was lit. Each lift stops for a fixed period at each floor. The car LED display indicates the current floor. (The in-car request buttons and LED displays are attached to a serial I/O device. Programming them is very similar to the parallel devices.)

Section 6.7 develops a simplistic scheduling algorithm for the lifts in which cars respond individually to requests from both the in-car buttons and the shaft buttons.

Section 6.8 completes the exercise: the algorithm of Section 6.7 is refined to closer approximate the expected behaviour of real lifts.

6.3. Overview of Embedded Gofer's monadic I/O

This section briefly introduces Embedded Gofer's monadic I/O facilities. Embedded Gofer was described in Chapter 3 in terms of both matched stream and continuation I/O. Chapter 5 presented the basis of a monadic I/O style for processes. Here, further motivation for the change to monads is given, and several I/O combinators which will be used in the liftshaft solution are derived.

6.3.1. Why change to monads?

Chapter 3 noted that in CPS I/O, there are three basic combinators for joining two I/O actions together into a new single action: application `($)`, composition `(.)`, and lambda-binding `(?)`. It is not always clear which combinator should be used in a given situation. By contrast, monadic I/O has a single basic combinator, `bind`, from which all further I/O combinators can be derived.

The `dselect` operation of Chapter 3 is slightly different from the other CPS I/O operations. It is in effect a branch point, from which control passes either to the interrupt continuation or the message continuation. For this reason, it cannot be used inside the CPS `loop` combinator without explicitly marking the two points from which the loop could repeat.

```
wrong === loop (dselect intcont msgcont)
right === loop (\c -> dselect (intcont $ c) (msgcont $ c))
```

Again, monadic I/O helps to tidy away the extra plumbing. The equivalent monadic branch operation, `select`, *returns* a monadic value representing which action should be taken, and the returned value is plumbed into sequence by the `loop` combinator.

```
right === loop (select intaction msgaction)
```

Two further reasons for converting to monadic I/O are that:

(i) type-checked message-passing, as described in Chapter 5, requires it;

(ii) the functional programming community is beginning to standardise on it [Gordon-Hammond94].

Appendix A gives a full source definition and explanation of the classes, types, and monadic I/O primitives used in Embedded Gofer. It differs from the presentation of Chapter 3 primarily only by being in the monadic style, and by incorporating the message type-checking of Chapter 5. However the scheme of Chapter 5 is itself extended slightly. In Embedded Gofer, a process action is parameterised not only on the process's incoming message type, as in Chapter 5, but also on the global PID type. This parameterisation is to allow the programmer to define a PID type specific to the application.

6.3.2. Combinators and loops

The monadic combinator, *bind*, is quite unwieldy both to read and to type in. In the literature it is often abbreviated to two shorter forms, *(>>=)* and *(>>)* [Gordon92] [GordonHammond94]. The former is just *bind*, and the latter is for the special case when the value being bound forward is of no importance.

```
x >> y = x `bind` \_ -> y
```

The operator *(>>)* is a useful short form, but *(>>=)* is still rather unwieldy, especially since it is always followed by a lambda abstraction (`\x -> ...`). Instead, let the symbol *(?)* be defined as a synonym for *bind*, by analogy with the CPS operator in Chapters 3 and 4 which had the same purpose.

Under the monadic style, we can write various sorts of loop construct, just as in CPS:

- A simple non-terminating loop:

```
loop :: Monad m => m () -> m ()
loop body = body >> loop body
```

- A non-terminating loop which keeps a local state – the state is updated every time the loop body is evaluated:

```
loopwith :: Monad m => s -> (s->m s) -> m ()
loopwith initstt body = body initstt ? \newstt ->
                        loopwith newstt body
```

- Loops that terminate on specific boolean conditions:

```
while :: Monad m => Bool -> m Bool -> m ()
while cond body = if cond then
                      body ? \newcond ->
                      while newcond body
                  else result ()

until :: Monad m => m Bool -> m ()
until body = body ? \cond ->
             if cond then result ()
             else until body
```

- A fixed iterative loop with counter:

```
for :: Monad m => Int -> m () -> m ()
for 0     body = result ()
for (n+1) body = body >> for n body
```

The liftshaft controller makes use of *loop*, *loopwith*, and *until*. A further combinator, *forall*, is also used. It turns a list of items into a list of monadic actions using *map*, then combines all the actions together using *foldr*.

```
forall :: Monad m => [a] -> (a->m ()) -> m ()
forall list body =
    foldr (>>) (result ()) (map body list)
```

The behaviour of some of these loops can be seen as special cases of a more general monadic combinator *foldm*, which is itself used in the liftshaft controller.

```
foldm :: Monad m => (a->b->m a) -> a -> [b] -> m a
foldm action accum []      = result accum
foldm action accum (x:xs) =
    action accum x ? \newacc -> foldm action newacc xs

loop body             = foldm (\_ _ -> body)       ()  inflist
loopwith stt body = foldm (\new _ -> body new) stt inflist
for n body            = foldm (\_ _ -> body)       ()  [1..n]
forall list body  = foldm (\_ x -> body x)     ()  list

inflist = (): inflist
```

6.4. The timer device

The 68230 PIT device incorporates a clock/timer which can be used in several modes: to generate interrupts periodically; to generate a square wave output; to interrupt after a timeout period; to measure elapsed time; or to act as a device watchdog. Every software process could program one or more of these timer modes as it had need. Unfortunately there is but one timer, and many potential client processes. A single mode of operation must therefore be used, and a single device process must act as a time-server to all possible client processes.

Given this architecture, the obvious mode of operation is periodic interruption from the timer device. The server can then simulate various of the other modes in software, as illustrated in the following subsections. (Appendix C.1 gives details of initialising the PIT device to operate in this mode.)

6.4.1. Periodic interrupts

To multiplex the periodic interrupt to several clients, the combinator *forall* is used, and the message *Tick* duplicates the interrupt. The list of clients is defined in the call to *multiplex* (Figure 6.3).

```
data TickMsg = Tick Pid
instance AddressFor TickMsg Pid where address (Tick p) = p

multiplex :: [Pid] -> IntAction Pid () ()
multiplex clients =
    loop (getInterrupt >> forall clients (send . Tick))
```

Figure 6.3: Liftshaft: periodic timing.

6.4.2. Timeouts

To generate a signal after a timeout period, the time-server must first accept a message from a client requesting the timeout. Because several processes may request concurrent timeouts it is necessary to keep a queue of client processes awaiting signals. An interrupt causes the queue timeout values to be decremented, and when the head of the queue reaches zero, a signal can be sent to that client. For efficiency, every timeout value is stored relative to its neighbour in the queue. The definition of *timeout* in Figure 6.4 resembles that of *alarm* in Chapter 4, but is restricted to allow only one type of expiry message.

```
data TimeoutMsg = Timeout Time Pid
data ExpiryMsg  = Expired Pid
instance AddressFor TimeoutMsg Pid where address = const Clock
instance AddressFor ExpiryMsg  Pid where address (Expired p) = p

timeout :: IntAction Pid TimeoutMsg ()
timeout =
    loopwith [] (\q ->
        select (\Interrupt -> updReg tcr dismiss >> act_on (dec q))
               (\(Timeout t pid) -> result (insert pid t q)))
    where dec []             = []
          dec ((p,t):qs)     = (p,t-1): qs
          insert pid t  []   = (pid,t): []
          insert pid 0  q    = (pid,0): q
          insert pid t0 ((p,t1):qs)
                | t1>=t0     = (pid,t0): (p,t1-t0): qs
                | otherwise  = (p,t1): insert pid (t0-t1) qs
          act_on ((p,0):qs)  = send (Expired p) >> act_on qs
          act_on q           = result q
```

Figure 6.4: Liftshaft: timeouts.

A failing of this implementation is that the value of *Timeout t* must be a strictly positive integer: zero or a negative number causes the *timeout* process to cease signalling.

6.4.3. Elapsed time

Measuring elapsed time is similar to timeouts, except that the time value at the head of the queue is *incremented* on each interrupt, and the client process must produce *two* messages during the transaction. The definition of *elapsed* in Figure 6.5 is based on the definition of *timeout* (Figure 6.4), but contains an inefficiency because it traverses the queue twice for every *Stop* message. No doubt this inefficiency could be avoided; however, since measuring elapsed time is not required by the liftshaft problem, this is left as an exercise to the reader.

```
data ElapsedMsg   = Start Pid | Stop Pid
data ElapsedReply = Elapsed Time Pid
instance AddressFor ElapsedMsg   Pid where address = const Clock
instance AddressFor ElapsedReply Pid where address (Elapsed _ p) = p

elapsed :: IntAction Pid ElapsedMsg ()
elapsed =
    loopwith [] (\q ->
        select (\Interrupt -> updReg tcr dismiss >> result (inc q))
               (\msg -> case msg of
                   (Start pid) -> result ((pid,0):q)
                   (Stop  pid) -> reply pid 0 q >> result (pluck pid q)))
    where inc []          = []
          inc ((p,t):qs)  = (p,t+1):qs
          reply pid _ []  = result ()
          reply pid n ((p,t):qs)
               | p==pid    = send (Elapsed (n+t) pid)
               | otherwise = reply pid (n+t) qs
          pluck pid []    = []
          pluck pid ((p,t):qs)
               | p==pid    = qs
               | otherwise = (p,t): pluck pid qs
```

Figure 6.5: Liftshaft: elapsed time server.

6.4.4. Mixed models

Any reasonably complex application will require a mixture of the above time-server models. It is straightforward to combine them. For the liftshaft program, both timeouts and multiplexing are required, but not elapsed times. A refinement to the `timeout` server in Figure 6.4 is suggested by the frequent need for a client process to defer sending a specific message to another process for a fixed period of time, but to continue evaluating during the wait (as in the `alarm` server of Chapter 4). Although the `timeout` server above can send a message on to any process (not necessarily the original client), it sends only one sort of message. In the more general coding of a timing server in Figure 6.6, which includes multiplexing, the client can specify a particular message for the server to send following the timeout. This version again closely resembles the `alarm` server in Chapter 4, but has some extra subtleties in the type of `AlarmMsg` in order to satisfy the message type-checking scheme of Chapter 5.

```
data AddressFor msg Pid => AlarmMsg msg = At Time msg
instance AddressFor (AlarmMsg m) Pid where address = const Clock

alarm :: [Pid] -> IntAction Pid (AlarmMsg m) ()
alarm periodclients =
    loopwith [] (\q ->
        select (\Interrupt -> updReg tcr dismiss >>
                               forall periodclients (send . Tick) >>
                               act_on (dec q))
               (\(At t m)  -> result (insert t (send m) q)))
    where dec []              = []
          dec ((t,a):qs)      = ((t-1),a): qs
          insert t a []       = (t,a): []
          insert 0 a q        = (0,a): q
          insert t a ((t',a'):qs)
                | t'>=t       = (t,a): ((t'-t),a'): qs
                | otherwise   = (t',a'): insert (t-t') a qs
          act_on ((0,a):qs)   = a >> act_on qs
          act_on q            = result q
```

Figure 6.6: Liftshaft: full alarm server.

Other processes use the alarm server by means of the actions *defer* and *wait*, whose definition is again similar to that in Chapter 3. Whereas *defer* is fully polymorphic, *wait* is not, because it expects a specific message value by return. Hence, a different version of *wait* must be written for every process which requires the service.

```
defer :: (Process a p m, AddressFor (AlarmMsg m) p) =>
         Time -> m -> a p m ()
defer t m = send (At t m)

wait t = defer t (send TimesUp) >>
         recv ? \TimesUp -> result ()
```

For a periodic client list of three processes and the timeout requirements of the other liftshaft processes, a system including this *alarm* process can cope with timer interrupts at 5Hz. In practice this is adequate resolution, but only just.

6.4.5. Discussion

The timeout server could take a number of different forms. In essence, the differences are in what action it takes when a particular timeout request expires. The alternatives are as follows:

(a) The alarm sends a simple fixed signal to a client: a message with no content except its presence or absence. The receiving client can await only one timeout at a time, and can perform only a fixed action on receipt of the timeout.

(b) The alarm sends a variable message to a receiving client – in fact, the particular message specified by the requesting client. Several timeouts can be requested and awaited simultaneously or disjointly, and one of a set of different actions can be

performed on receipt, selected by the timeout message content. In this way, a client can delay an arbitrary action, not necessarily one it intends to perform itself.

(c) The alarm performs a variable action when a timeout reaches fulfilment. The requesting client specifies the particular action to be taken, but the alarm undertakes the action itself. This subsumes all the previous behaviours, because the action could be merely to send a message to another process. It could however be an action of arbitrary complexity.

The generality of alternative (c) is slightly dangerous. The purpose of a timing device driver is to *trigger* actions at specified times. It is less than ideal for the trigger process to engage in complex actions itself, when it ought to be dealing with interrupts. The complex action should be performed by a process with a lower priority. Run-time scheduling then allows the alarm's proper activity to interleave with the actions it has triggered. For this reason, the second alternative presented above is arguably the best model to adopt for a timing process. Like the first model, it sends a trigger message to another process, but unlike the first, it provides greater flexibility as to how the trigger may work.

6.5. Controlling other devices

It is often the case in embedded systems that although there are many sensors and effectors to be controlled, there are only a few interface devices available to do the controlling. The unit cost of devices, and size and weight considerations, are especially important for systems embedded in mass-produced goods. For this reason, there seem to be two common mismatches between the ideal level of hardware control and that which can actually be provided. Both these problems were encountered in programming the liftshaft exercises.

1. *Overlapping*. Disparate mechanisms may not only share the same I/O device, but also the same registers in that device.
2. *Polling*. Although it may be desirable for certain sensors to generate interrupts, the actual devices available may not be able to provide as many interrupts as there are sensors, and polling has to be used.

6.5.1. Registers are shared

The liftshaft motors, buttons, and call-lamps are controlled through two PITs, one for sensing, the other for effecting. Each device has two data registers which are connected to the apparatus. The different effectors and sensors overlap in these registers. Figure 6.7 shows the register bit-mapping on the devices. PIT device 2 is used for control: up and down refer to the call-lamps behind the buttons on the shaft. A and B are the motors for the two lift cars. PIT device 1 is used for the sensors. Here, the registers hold the instantaneous state of the shaft call buttons and the end-stop sensors for the cars A and B.

register/bit	7	6	5	4	3	2	1	0
pit2_padr	down4	down3	down2	up5	up4	up3	up2	up1
pit2_pbdr	X2	X1	Bon/off	Bdir	Aon/off	Adir	down6	down5
pit1_padr	down4	down3	down2	up5	up4	up3	up2	up1
pit1_pbdr	Q1	P1	Bbot	Btop	Abot	Atop	down6	down5

Figure 6.7: Liftshaft: PIT effectors and sensors.

This overlapping presents us with a problem. Embedded Gofer's I/O mechanism can read and write only whole-register values at a time. It does not provide for single-bit operations. When turning one motor on, a process must "know" the state of the other motor, and also of a couple of call-lamps, otherwise it could inadvertently cause them to change. But it is likely that the program will control the motors and lamps from separate processes, so a means of sharing the device state must be accomplished.

It was originally proposed that the I/O mechanism for writing to a register should in fact be an updating function, i.e.

```
primitive updReg "primUpdReg" :: DevProc a p m =>
                                  Addr -> (Word->Word) -> a p m ()
```

This does give single-bit control because the run-time system applies the given function, which might for instance be a masking operation, to the existing contents of the appropriate register. The mechanism was altered to its present form when it was realised that often I/O registers can be read-only or write-only. It even happens sometimes that two different registers, one of which is read-only and the other write-only, are mapped to the same address. In these situations it would be quite wrong for the run-time system to read a register, apply the function to its value, and write the new value back. Chapter 3, Section 3.2.2, discusses the atomicity of register updates.

6.5.2. Writing shared registers: a server process

Given then that the I/O mechanism does not provide atomic update, perhaps the best way to ensure that the current state of the two effector registers is respected, is for a single process to guard access to them. The process in Figure 6.8 shadows the register values in its local state, and other processes may use an updating function in a message in order to communicate with the registers.

```
data ShRegT = UpdReg AB (Word->Word)
            | GetReg AB AB
instance AddressFor ShRegT Pid where address = const SharedReg

sharedreg :: DevAction Pid ShRegT ()
sharedreg =
    let initval = byte 0 in
    putReg (pit2 padr) initval >>
    putReg (pit2 pbdr) initval >>
    loopwith (initval,initval) (\(a,b) ->
        recv ? \msg ->
        case msg of
            (UpdReg A f) -> let fa = f a in
                            putReg (pit2 padr) fa >> result (fa,b)
            (UpdReg B f) -> let fb = f b in
                            putReg (pit2 pbdr) fb >> result (a,fb)
            (GetReg p A) -> send (Reg p a) >> result (a,b)
            (GetReg p B) -> send (Reg p b) >> result (a,b))
```

Figure 6.8: Liftshaft: shared registers.

A series of updating functions can easily be defined to turn the motors and lamps on or off from within other processes, as shown in Figure 6.9. The functions are defined here in tabular style, partly because the mapping from logical action to control action is unorthogonal. Were the system to have a larger number of lifts, the hardware location of lifts in registers could probably be computed arithmetically from the logical action rather than by pattern-matching.

```
data Dir = Stop | Up | Down

motor :: Process a p m => AB -> Dir -> a p m ()
motor A Up   = send (UpdReg B (setbit 3 . clrbit 2))
motor A Down = send (UpdReg B (setbit 3 . setbit 2))
motor A Stop = send (UpdReg B (clrbit 3))
motor B Up   = send (UpdReg B (setbit 5 . clrbit 4))
motor B Down = send (UpdReg B (setbit 5 . setbit 4))
motor B Stop = send (UpdReg B (clrbit 5))

lampOff :: Process a p m => Floor -> Dir -> a p m ()
lampOff (Floor 1) Up = send (UpdReg A (clrbit 0))
-- etc.
```

Figure 6.9: Liftshaft: register update functions.

6.5.3. Reading shared registers: the need to poll

It is important to realise how the call buttons and lamps are intended to work. When a button is pressed, the corresponding lamp should be lit immediately. The lamp should be extinguished only when the call is serviced by a lift. The set of lamps therefore acts as a persistent record of outstanding lift requests. Now ideally, the interface logic should interrupt when a button is pressed, since such an event is infrequent by comparison with other events. Due to the shortage of interrupt lines on the PIT however, the hardware is not

configured to make this possible. The only way to determine when a button has been pressed is by polling the data registers.

For these reasons, part of the expected behaviour of the buttons and lamps is programmed within the shared register server process. This process periodically reads the state of the buttons and copies it to the lamps *without* extinguishing any lamps already lit. This can be accomplished by a simple bit-wise OR operation. Provided the period is shorter than the average time for which a user holds a button, the visible behaviour should be as expected. Any other process needing to know whether a button has been pressed can simply request to see the current state of the lamps. Lamps are lit *only* by the shared register process; they are extinguished *only* by requests from other processes. This scheme is reflected in Figure 6.10, which extends the process definition of Figure 6.8. To determine whether a lamp is lit or not, a process must first read the register state then apply a bit mask to it, as in Figure 6.11.

```
data ShRegT = as before, but adding
            | ClockTick

sharedreg =
    as before, up to
        case msg of
            as before, but adding
            ClockTick -> getReg (pit1 padr) ? \aval ->
                         getReg (pit1 pbdr) ? \bval ->
                         let fa = (notw aval) `orw` a
                             fb = (notw bval) `orw` b in
                         if fa==a && fb==b then
                              result (a,b)
                         else putReg (pit2 padr) fa >>
                              putReg (pit2 pbdr) fb >>
                              result (fa,fb)
```

Figure 6.10: Liftshaft: extension to shared register server.

```
lamplit :: Process a p m => AB -> Floor -> Dir -> a p m Bool
lamplit p (Floor n) dir =
    let ab = whichreg dir n
        bm = bitmask  dir n in
    send (GetReg p ab) >> recv ? \(Reg _ val) -> result (mask bm val)
```

Figure 6.11: Liftshaft: reading shadowed registers.

6.5.4. An example program

Using the `sharedreg` server process and the `alarm` process of the previous section, the first of the series of graded exercises in liftshaft control can now be coded. The exercise in Figure 6.12 is to move a lift car up and down, detecting when it reaches the top and bottom floors, and reversing direction when it does so. Register B is polled every 10 clock ticks to determine whether the end stop has been activated yet.

```
data Ex1 = Reg Word
instance AddressFor Ex1 Pid where address = const Example1

lift :: AB -> DevAction Pid Ex1 ()
lift car =
    loop (motor car Up >> await car Top >>
          motor car Down >> await car Bot)
    where await car end = until (send (At 10 (GetReg car B)) >>
                                 recv ? \(Reg _ val) ->
                                 result (endstop car end val))
          endstop A Top = bitset 2
          endstop A Bot = bitset 3
          endstop B Top = bitset 4
          endstop B Bot = bitset 5
```

Figure 6.12: Liftshaft: first exercise.

6.5.5. A further set of shared registers

The set of buttons inside each lift car is attached to a DUART, as also is the LED display and audible bleep. In this case, unlike the PITs, different effectors and sensors do not overlap in the data registers. Each car has a separate channel on the device, and the register arrangement shown in Figure 6.13 is identical for both cars.

register/bit	7	6	5	4	3	2	1	0
duart_tx	bleep	7-seg-7	7-seg-6	7-seg-5	7-seg-4	7-seg-3	7-seg-2	7-seg-1
duart_rx	–	–	floor6	floor5	floor4	floor3	floor2	floor1

Figure 6.13: Liftshaft: DUART registers.

The transmit register sends to each segment of the seven-segment display and also to the switch for the bleep. The receive register holds the instantaneous state of the in-car buttons at the moment it was last polled. The DUART can be initialised to poll at a wide range of baud rates. Due to the hardware design, it is not possible for a button press to generate an interrupt. Also, it is necessary to keep a persistent record of which buttons have been pressed but not serviced yet.

This similarity in behaviour to the shaft buttons suggests that the program solution should be similar. Although the in-car buttons do not have a corresponding set of lamps to hold the state, the same technique can be introduced by keeping a simple software shadow of the receive register. This can be added as an extension to the shared register server in Figures 6.9 and 6.11. Appendix D gives the full coding of this extended server.

There are functions corresponding to `lamplit`, and `lampOff` which operate similarly on the shadowed state of the in-car request buttons: `carcall` and `clearcall`.

6.6. Interrupts

In order to determine with some precision the position of each of the two liftshaft cars, a telemetry mechanism is provided. The driving pulleys and strings incorporate a wheel in which a set of holes has been precisely drilled. Two optical detectors straddle each of these wheels and generate values for status registers on the PITs indicating "hole" or "not hole". The PIT devices can be programmed to cause interrupts to the main processor every time this status value changes.

A simple count of the number of telemetry interrupts, together with knowledge of the direction of motion, is enough to determine the exact position of either car to the accuracy of the distance the pulley string moves between holes (approximately 3mm). This assumes that interrupts are dealt with in a timely fashion.

6.6.1. Telemetry technique

A basic telemetry process behaves as follows (see Figure 6.14). Keeping a local record of the motor direction and current position, it awaits interrupts. When an interrupt occurs, it either increments or decrements the current position, depending on direction. At any time the process could receive a message indicating a change of direction, or could receive a request for the value of the current position.

```
data TelemMsg = Change AB Dir | GetPos AB
data Dir      = Down | Up | Stop
instance AddressFor TelemMsg Pid where
    address (Change ab _) = Telem ab
    address (GetPos ab)   = Telem ab

updpos :: Dir -> Int -> Int
updpos Stop n = n
updpos Down n = n-1
updpos Up   n = n+1

telemetry :: AB -> IntAction Pid TelemMsg ()
telemetry ab =
    loopwith (0,Stop) (\(pos,dir) ->
        select (\Interrupt -> result (updpos dir pos, dir))
               (\msg -> case msg of
                   Change _ newdir -> result (pos, newdir)
                   GetPos _        -> send (Pos ab pos) >>
                                      result (pos,dir))
```

Figure 6.14: Liftshaft: basic telemetry.

The disadvantage of this telemetry server process is that it causes whatever process is controlling a lift car to *poll* for the car's position. This is likely to lead to inaccuracy, and runs counter to the hardware design principle which directed that interrupts should be used for this job. A preferable method is *virtual* interrupts. The telemetry server process in

Figure 6.15, while counting real interrupts, sends out a virtual interrupt to the controlling process every time the car's position is exactly at a floor. The virtual interrupt is of course just a message.

```
floorposns :: [Int]
floorposns = [0,23,45,69,93,115]

floor :: Int -> Floor

telemetry ab =
    loopwith (0,Stop) (\(pos,dir) ->
        select (\Interrupt ->
                    let newpos = updpos dir pos in
                    if newpos `elem` floorposns then
                        send (Reached ab (floor newpos)) >>
                        result (newpos, dir)
                    else result (newpos, dir))
               (\(Change _ newdir) -> result (pos, newdir)))
```

Figure 6.15: Liftshaft: virtual interrupts.

What about the efficiency of this server? With `telemetry` defined as in Figure 6.15, every interrupt causes a comparison of `newpos` against some of the elements of `floorposns`. In most cases, it compares with *every* element, and the equality test fails for all six floors. The cost of the six equality tests and boolean *or* operations incurred by `elem` is quite high. For any single increment of motion, we only *need* to test the new position against one possible floor position – the next floor in that direction. This value can be memoised in the local state, rather than evaluated on every increment. See Figure 6.16.

```
telemetry ab =
    loopwith (updFlr 0 Stop) (\(pos,dir,flr) ->
        select (\Interrupt ->
                    let newpos = updpos dir pos in
                    if newpos == flr then
                        send (Reached ab (floor newpos)) >>
                        result (updFlr newpos dir)
                    else result (newpos, dir, flr))
               (\(Change _ newdir) -> result (updFlr pos newdir)))
    where
        updFlr :: Int -> Dir -> (Int,Dir,Int)
        updFlr pos dir = (pos, dir, nextfloor pos dir)
```

Figure 6.16: Liftshaft: memoised telemetry server.

6.6.2. Example

A lift car process can now be programmed (Figure 6.17) to solve the second exercise: to stop the car at every floor, extinguishing at each floor any call-lamps that might be lit. There is no need to *test* whether a lamp is lit – it can be turned off regardless of its prior state.

```
data LiftMsg = Reached AB Floor | Restart AB
instance AddressFor LiftMsg Pid where
    address (Reached ab _) = Lift ab
    address (Restart ab)   = Lift ab

top, gnd :: Floor

lift :: AB -> Action Pid LiftMsg ()
lift ab =
    send (Change ab Up) >> motor ab Up >>
    loopwith Up (\dir ->
        recv ? \msg ->
        case msg of
            (Reached _ flr) ->
                    motor ab Stop >> writeLED ab flr >>
                    defer waitTime (Restart ab) >>
                    if flr==top || flr==gnd then
                        let newdir = rev dir in
                        lampOff flr newdir >>
                        send (Change ab newdir) >> result newdir
                    else lampOff flr dir >> result dir
            (Restart _) -> motor ab dir)
```

Figure 6.17: Liftshaft: second exercise.

The formulation of this exercise in Figure 6.18 is perhaps more elegant, using the intuition that the lift car does the same thing *for all* floors.

```
lift ab =
    send (Change ab Up) >> motor ab Up >>
    loop (forall [(gnd+1)..top] (dofloor Up) >>
          send (Change ab Down) >> motor ab Down >>
          forall [(top-1)..gnd] (dofloor Down) >>
          send (Change ab Up) >> motor ab Up)
    where
    dofloor dir flr =
        recv ? \(Reached _ _) ->
        motor ab Stop >> writeLED ab flr >> lampOff flr dir >>
        wait waitTime >> motor ab dir
```

Figure 6.18: Liftshaft: second exercise re-coded.

6.7. A naive scheduling algorithm

The device drivers are now all in position, so the complete lift scheduling algorithm can be written. A first attempt is shown in Figure 6.19. We treat all lift cars as independent processes. Assume that a car is stationary. There is a monadic function `anyrequests` which determines whether there are any outstanding requests in a particular direction starting from a particular floor. Using this function, the controller process first checks in the direction the lift has most recently been moving, then in the opposite direction. In either case if a request is found the lift is set in motion. When in motion, the car LED display is updated on reaching each new floor. When the car reaches the appropriate floor, it is

stopped and the controller returns to check for new requests. If no request is found, the controller idles for a brief period of time before checking again.

```
lift :: AB -> Action Pid LiftMsg ()
lift ab =
    writeLED ab (Floor 1) >>
    loopwith (Floor 1,Up) (\(curflr,curdir) ->
        anyrequests ab curflr curdir ? \req1 ->
        case req1 of
            Just newflr -> go newflr curdir
            Nothing -> let otherdir = rev curdir in
                       anyrequests ab curflr otherdir ? \req2 ->
                       case req2 of
                           Just newflr -> go newflr otherdir
                           Nothing -> wait briefpause >>
                                      result (curflr,curdir))
    where go flr dir = lampOff flr dir >>
                       send (Change ab dir) >>  motor ab dir >>
                       until (recv ? \(Reached _ n) ->
                              writeLED ab n >>
                              result (n==flr)) >>
                       motor ab Stop >>  clearcall ab flr >>
                       result (flr,dir)
```

Figure 6.19: Liftshaft: naive scheduling algorithm.

The monadic *anyrequests* function, defined in Figure 6.20, is where most of the work is done. For each floor from the current one, either upwards or downwards, it determines whether there is a request for that floor on either the in-car buttons or the shaft buttons. The result is the first floor for which a request is found.

```
anyrequests :: AB -> Floor -> Dir -> Action Pid LiftMsg (Maybe Floor)
anyrequests ab flr dir =
    case dir of
        Down -> foldm check Nothing [flr..ground]
        Up   -> foldm check Nothing [flr..topmost]
    where check :: (Maybe Floor) -> Floor ->
                                   Action Pid LiftMsg (Maybe Floor)
          check Nothing thisflr = if carcall ab thisflr
                                  || lamplit ab thisflr dir then
                                     result (Just thisflr)
                                  else result Nothing
          check (Just n) _      = result (Just n)
```

Figure 6.20: Liftshaft: determination of requests.

Unfortunately, the controller process of Figure 6.19 does not accurately capture the "expected" behaviour of a lift. There are at least three problems.

1. A car does not respond to any requests made between the moment it starts moving and the moment it stops again, whether on the in-car buttons or the shaft buttons. For instance, a car begins at the ground floor and sets off carrying a passenger to the top

floor. Shortly afterwards, a person on a middle floor presses a shaft button to indicate that they would also like to go up. It would be sensible for the car to stop at this middle floor, since it is already travelling in the correct direction and the delay would not much inconvenience the first passenger. However, as it stands, the controller actually takes the lift first to the top floor, then back down to the middle floor, then possibly down even further, before finally taking our weary second traveller upwards.

2. The operation `anyrequests` only reports shaft requests *in the potential direction of motion* of the car. That is, a car currently positioned above a request to go up will not service it, and a car currently positioned below a request to do down will not service that either. In some instances, this is the correct decision. For instance, a car on the ground floor with an in-car request to go to the top floor should not stop at a middle floor to admit passengers who intend to go down. However, in other instances the decision is wrong. A car on the ground floor with *no* in-car requests ought to move to service a shaft request on a middle floor, no matter whether that request is to go up or down.

3. The controller does not deal correctly with requests for the lift to service the floor it is currently at. The controller starts the car moving, then waits forever for the car miraculously to return to its starting position!

6.8. The full solution

6.8.1. A new intuition

In order to provide the correct behaviour, we need a new intuition about how lifts work. Essentially, the controller has to make two different types of decision at two different points. First, when the car is stationary, the decision is whether or not to start it moving, and if so, in what direction. Secondly, when the car is in motion, but shortly before it reaches each new floor, the decision is whether or not to stop at the approaching floor.

Hence, we come to the following algorithm, defined in Figure 6.21. The controller memoises in its local state a record of whether the car is moving; if so, in what direction; what is the floor being approached; and should the car stop there. If the car is not moving, the most recent direction of motion and current floor are kept. There has to be a moment in the algorithm at which the decision is made whether to stop at the approaching floor. For this coding, that moment is when the car passes the previous floor, although it could be later. It is important however that the computation is given enough time to evaluate this decision before the floor is reached. In a real lift, sufficient time must also be set aside for the car to decelerate.

```
data HowFar    = ToNext | Beyond
data Motion    = Moving HowFar | Stopped
data LiftMsg   = Reached AB Floor | Reg AB Word | TimesUp AB
type CarState = (Motion,Floor,Dir)

instance AddressFor LiftMsg Pid where
    address (Reached ab _) = Lift ab
    address (Reg ab _)     = Lift ab
    address (TimesUp ab)   = Lift ab

lift :: AB -> Action Pid LiftMsg ()
lift ab = writeLED ab (Floor 1) >>
          loopwith (Stopped, Floor 1, Up) lift'
  where
      lift' :: CarState -> Action Pid LiftMsg CarState
      lift' (Moving ToNext, _, dir) =
              recv ? \(Reached _ flr) ->
              motor ab Stop >> writeLED ab flr >>
              wait briefpause >> result (Stopped, flr, dir)
      lift' (Moving Beyond, _, dir) =
              recv ? \(Reached _ flr) ->
              writeLED ab flr >>
              let nf = next flr dir in
              stopAt nf dir ? \howfar ->
              result (Moving howfar, nf, dir)
      lift' (Stopped, flr, dir) =
              anyrequests ab flr dir ? \ndir ->
              case ndir of
                  Stop -> wait briefpause >> lampOff flr dir >>
                          result (Stopped, flr, dir)
                  _    -> clearcall ab flr >> lampOff flr ndir >>
                          send (Change ab ndir) >> motor ab ndir >>
                          let nflr = next flr dir in
                          stopAt nflr ndir ? \howfar ->
                          result (Moving howfar, nflr, ndir)
      stopAt :: Floor -> Dir -> Action Pid LiftMsg HowFar
      stopAt flr dir = carcall ab flr  ? \x ->
                       lamplit ab flr dir  ? \y ->
                       result (bool2howfar (x||y))
      next :: Floor -> Dir -> Floor
      next (Floor n) Up   = Floor (max (n+1) top)
      next (Floor n) Down = Floor (min (n-1) gnd)
```

Figure 6.21: Liftshaft: the full algorithm.

The monadic operation *anyrequests* is now simpler, as shown in Figure 6.22. Its answer is now just the direction to move instead of a floor number, so it can be coded more efficiently. Rather than checking floors in sequence for a request, it checks them all at once, using a mask on the bitmap representation. The proper priority of requests can now also be respected. First priority goes to in-car requests in the current direction; second priority goes to in-car requests in the opposite direction; then shaft requests in the current direction are checked; and finally shaft requests in the opposite direction are checked. If no request is found, the lift should not move.

```
anyRequests :: AB -> Floor -> Dir -> Action Pid LiftMsg Dir
anyRequests ab flr dir =
        readInCarButtons ab ? \incar ->
        case whichDir (inCarMask flr) dir incar of
            Up   -> result Up
            Down -> result Down
            Stop -> readShaftButtons ab ? \shaft ->
                    result (whichDir (shaftMask flr) dir shaft)

whichDir :: (Dir -> Word) -> Dir -> Word -> Dir
whichDir mask dir word =
    let otherdir = rev dir in
    if (mask dir) 'orw' word then result dir
    else if (mask otherdir) 'orw' word then result otherdir
    else result Stop
```

Figure 6.22: Liftshaft: new determination of lift requests.

6.8.2. Independent lifts or communicating lifts?

In this solution, neither lift car communicates with the other – they are totally independent. This may seem counter-intuitive, but consider the following situation.

1. *Both cars are idle*, immediately before the morning or evening "rush-hour". Suddenly, a *large* number of people arrive at the shaft, all on the same floor, all wanting to go in the same direction. *Without* independence, one car signals that it is closest to the demand, and the second car remains idle until the first car is full. Only then can the second car respond to the extra demand, because it has until that moment been "locked out" by the nearer car. The extra passengers have to wait even longer for the second car to arrive than the first batch of passengers had to. However, *with* independence, both lifts can move towards the demand simultaneously, thus collecting as many passengers as possible, as quickly as possible.

Now consider independence against communication in the only three other possible situations.

2. *Both cars are idle and a* small *number of passengers arrives on one floor.* Without independence, only one car moves to service the request. With independence, both cars move to service this request, but only one car is actually required. After the request has been serviced, the position of the cars is different to what it might have been with communication, but it is impossible to say that the positioning is either worse or better, because the lift cannot predict future demand. Independence causes only expenditure of energy, which may or may not turn out to have been unnecessary.
3. *One car is idle, the other is in motion, and a new shaft request is made.* Either the idle car is closer to the request, or the moving car is. In the first case, whether or not the cars communicate, the idle car moves to service the request. In the second case, with communication, the idle car stays idle; with independence, the idle car moves. This reduces to the argument of situation 2.

4. *Both cars are in motion and a new shaft request is made.* Whether or not the lifts communicate, the better car to service the request does so.

Avoiding communication between cars leads to a simple controlling algorithm. Figures 6.21 and 6.22 demonstrate that the algorithm for independent cars is a straightforward sequential case analysis, executed periodically. An alternative algorithm based on communication between cars would add another layer of complexity (see for example [Armstrong+93], pp.162-168). First, each car ''lays claim'' to each request. Secondly, a complex calculation is performed to determine which claim should win, based not only on the car's proximity and direction, but also on the number of intervening requests that car must service. Thirdly, the car must insert each granted claim into a local schedule, and follow that schedule.

The algorithm based on communication must have a *fourth* element, if it is to be optimal. Consider the situation where both cars are idle at the same floor. A request is made at a far distant floor, *f*. Only one of the cars is granted the claim to service the request, but it does not matter which. Now, once that car is in motion, a series of requests is made for every floor between the starting point and *f*. The car already in motion is the natural winner in every claim to these requests, because it is both closer and already moving in the right direction. However, by so doing, the new requests are inserted *before* the original request in its schedule, and hence the first request is delayed significantly due to each intermediate stop. One solution is to completely re-allocate *both* cars' schedules on every new request. Eventually, the idle car would be kick-started to service at least some of the new requests.

However, even this complicated solution does not provide the optimal behaviour displayed by independent cars. In the same situation, both cars are started simultaneously on the journey to floor *f*. Each car decides independently whether or not to stop at each intervening floor, but only as it approaches it. For various reasons, ranging from differing output rate of the two motors to the behaviour of new passengers, the cars very quickly cease to move in synchrony, and start to display a leap-frog action: they dynamically partition the set of floors between the starting position and *f*. One of the cars eventually services the first request, but almost certainly in a shorter time than a communicating car would.

It could be argued that it was not functional programming techniques that led to the formulation of this new lift scheduling algorithm. The discussion above focuses rather on the nature of the *state* held by each component in the system, and the volume of communication that is required to share this state. However, it is a consequence of using a functional language that the programmer *must* think carefully about state issues, because the state has to be coded explicitly. It becomes natural to try to minimise the amount of state information where possible.

For this particular implementation, the volume of communication between the lifts was a fairly important issue, due to the performance limitations of Embedded Gofer. The solution described requires no communication between the lifts, whereas the alternative with communicating lifts would require at least two extra message deliveries. Although the cost of message passing is not critical, it is still significant here.

6.8.3. The process table

Finally, we give the `main` definition of the process table for the complete liftshaft program.

```
main :: Main Pid
main = Define
         [defIntProc (telem_vector A) (Telem A) (telemetry A),
          defIntProc (telem_vector B) (Telem B) (telemetry B),
          defIntProc clock_int_vector Clock     (alarm [SharedReg]),
          defDevProc                  SharedReg sharedreg,
          defOrdProc                  (Lift A)  (lift A),
          defOrdProc                  (Lift B)  (lift B)]
```

Figure 6.23: Liftshaft: the process table.

6.9. Conclusions

The liftshaft program is fast enough to run a single lift comfortably, but there are signs of stress when running both lifts. For instance, when both lifts are in motion, one of them is inclined to overshoot its destination floor. This suggests that the implementation has exceeded its maximum throughput of interrupts; compared with a single lift system, twice as many interrupts arrive per unit time, but the same amount of evaluation is required for each interrupt. Nevertheless, the system is close to meeting its targets; a small amount of optimisation in the run-time system, perhaps especially the garbage collector (Chapter 7), would probably give a sufficient speed increase to complete the exercise satisfactorily.

The space performance is encouraging however. The Embedded Gofer program for the liftshaft uses a heap of 10,000 cells, about 80K of data space. The run-time system (RTS) for Embedded Gofer is about 48K. The compiled program (including the RTS) is below 160K in binary (text) size. In total therefore, the program uses less than 250K of memory, well within the limits supplied (768K). This is small for a functional program, although an embedded systems programmer might still regard it as rather large – the C version of the liftshaft program totals under 40K.

Several points of interest came out of the programming exercise.

- With the monadic I/O framework, new forms of control structure can be defined as needed, for example the various loop constructs in §6.3.2. Of these, the

straightforward `loop`, the `loopwith` local state, and the `until` loop were used frequently throughout the exercise. All of these have direct analogues "for free" in the imperative world, so perhaps there is no gain from the ability to define new control structures? Indeed, might not complete freedom to define new structures lead to errors and confusion? The standard structures should of course be provided in a standard prelude, and novel structures used sparingly. Nevertheless, novel control structures such as `forall` and `foldm`, which are not standard structures in any imperative language, were found useful in this case study.

- Run-time timing requirements can be served to the main body of a program by a single device-driver; this driver was largely re-used from a previous application. The re-use of components is made easier in a functional language by the knowledge that a component can have no side-effects on other components.
- The mismatch between an ideal hardware model and its actual implementation often has to be patched up in software. For instance, data registers are often shared between disparate transducers. In Embedded Gofer, not only can the patch be made, but abstractions can be defined to hide the sharing from the rest of the program.
- Another hardware mismatch occurs when true interrupts cannot be provided and polling is required in their place. Again, Embedded Gofer can abstract from polling and generate messages as simulations of the intended interrupts.
- On the other hand, interrupts sometimes signal low-level information that is too detailed for some parts of the program. An Embedded Gofer process can make an appropriate abstraction by transforming a sequence of interrupts into a single virtual interrupt (again, a message).
- The functional approach tries to avoid using too much software state, because this must be coded explicitly. Thinking carefully about the software state needed by the exercise led to a new insight into the application and hence a new solution.

Chapter 7. Garbage collection

So far, this Thesis has addressed the problem of expressing I/O for embedded computations. However, a lack of suitable I/O facilities is not the only obstacle to the use of functional languages in programming embedded systems. All functional languages incorporate automatic memory management; allocation and de-allocation of memory cells is implicit, hidden from the programmer. Memory allocation does not present a problem unless there is not enough free memory available. To ensure that sufficient memory is available, cells which have been used but are no longer needed are de-allocated by *garbage collection*. It is this de-allocation algorithm which can often obstruct the proper functioning of an embedded system.

This chapter reviews some previous garbage collection algorithms (Section 7.1) and presents a new hybrid algorithm (Section 7.2), which is designed to minimise memory overheads at the same time as providing the guarantees needed by embedded systems. An outline proof of correctness is sketched (Section 7.3), and performance results are given (Section 7.4). Section 7.5 summarises and concludes.

7.1. A review of some previous collectors

Traditional garbage collectors (GCs) for functional languages run on a *stop-and-collect* basis. When there is no free heap left, the computation is interrupted, a garbage collection is invoked to reclaim dead space, and then the computation resumes. However embedded systems, and in particular real-time systems, require some guaranteed throughput and cannot allow pauses for arbitrary lengths of time at unspecified intervals. For example, in the model liftshaft case study, if a stop-and-copy garbage collection were to occur shortly before a car reached a floor, the telemetry process (which reports the lift's position) would be delayed, causing the lift to stop between floors, rather than exactly at the floor.

7.1.1. Incremental collection

The alternative is to take an *incremental* approach to GC. By doing some small, known, amount of GC work at frequent intervals, the collector ensures that the supply of free heap cells never dries up. Overall the amount of work expended on GC might be higher, but the performance of the system is more evenly predictable. Hence, incremental GC is often termed ''real-time'' GC.

For more information on general uniprocessor GCs, see Wilson's review paper [Wilson92] which includes a number of so-called ''real-time'' algorithms. Many such algorithms for garbage collection have been suggested, employing incremental techniques.

Most, however, assume that

- an *additional processor* will be available to handle GC in parallel "on-the-fly", for instance [Ben-Ari84, Appel+88, Queinnec+89]; see also the review paper [Abdullahi+92] on general distributed garbage collection,

and/or that

- *additional memory* can be used to offset the extra time costs of the incremental algorithm, for instance [Baker78, LiebermanHewitt83, Brooks84, Yuasa90, Baker92]

Neither of these assumptions is necessarily valid for embedded systems. Although many applications of functional languages can safely assume large memory spaces, in embedded systems the amount of memory available is severely limited by cost: when designing electronic instruments, for example, the microprocessor element is only one part of the overall device, and one wishes to keep the cost of each part as low as possible if the device is destined for mass production. For many embedded systems, a single processor with small memory is the ideal.

Most incremental garbage collectors require memory much larger than the size of the usable heap. Baker in his seminal 1978 paper (and later variants) relies on *semi-spacing*, which requires twice the usable heap space; *generational* collectors [LiebermanHewitt83] work on a similar principle; and the *treadmill* [Baker92] requires the layering of a doubly-linked circular list on top of the heap. In addition, many of these algorithms are designed to optimise virtual memory performance (the treadmill is a notable exception); an embedded system has no virtual memory, and often cannot afford the expense of much non-active-heap memory. In this respect, a *mark-sweep* [McCarthy60] based collector is more attractive for an embedded system, since its overhead is a single bit per heap cell.

A traditional stop-and-collect mark-sweep algorithm is generally invoked when the *freelist* becomes empty. The *marker* recursively traces the graph reachable from the computational roots, marking cells as it goes, after which the *sweeper* scans linearly through the heap, linking garbage (unmarked) cells into a new freelist. To convert mark-sweep to an incremental scheme, it is necessary to take a snapshot of the roots at a point in time before the freelist is exhausted, and interleave the operation of the garbage collector with the operation of the *evaluator/mutator*. Some extra functionality is needed in the mutator to ensure that any new sections of graph that are created will also be traced and marked. However, no special action is needed to cater for sections of graph which become garbage after the marker has started: they will either be picked up in the current cycle (if not yet marked), or the next (if already marked).

7.1.2. Queinnec's Mark-During-Sweep method

Queinnec et al. [Queinnec+89] describe such a real-time GC algorithm that performs incremental work in both the mark and sweep phases simultaneously, with each phase operating on distinct sets of mark bits. When both phases terminate, the set of mark bits

just created by the marker is passed to the sweeper; meanwhile, the marker re-starts with a fresh set of mark bits. Hence, the sweeper is always working on an older set of mark bits than the marker, while both are interleaving their activity with the mutator. In this way, *Mark-During-Sweep* is a conservative algorithm: some cells which are not part of the live heap may be marked, with the implication that some garbage may not be recycled immediately—there may be up to two GC cycles after a cell becomes garbage but before it is added to the free list.

The state of the on-going marking operation must be recorded explicitly in any incremental algorithm, since the marker must be interleaved with the mutator. Mark-During-Sweep uses *grey colour* bits (originally due to Dijkstra [Dijkstra76]) for this purpose. Marked (black) cells can only point to unmarked (white) cells indirectly, through half-marked (grey) cells. The set of grey cells is the "wavefront" dividing the set of black cells from the set of white cells. As a grey cell is examined by the marker, the cells it points to are shaded grey (if they are not grey or black already), and the cell itself is darkened to black. Marking commences with only the computational roots shaded grey, and all else white. As marking continues, the wavefront gradually moves into the white region until the entire reachable graph has been marked black and there are no grey cells left, at which point marking is complete and the sweeping phase may begin.

The important aspect of Mark-During-Sweep is that the marker can take a new snapshot of the roots and continue working (with a fresh set of mark bits and grey bits) whilst the sweeper reclaims as garbage those cells unmarked in the previous cycle.

Mark-During-Sweep has a low memory overhead, requiring just 3 bits per heap cell—two denoting black/white and grey for the marker, and one denoting black/white for the sweeper. The inefficiency of the algorithm is that every invocation of the incremental marker must search the heap for grey cells. This has a worst-case behaviour where the time for a full mark phase is quadratic in the heap size. Queinnec et.al. propose that efficient implementation of the marker should be possible with an additional parallel processor.

7.1.3. Yuasa's Stack-Collect method

Yuasa [Yuasa90] on the other hand describes an incremental scheme where the state of the marking phase is recorded in an explicit stack (rather than using grey bits). In colour terms, the grey cells are those currently on the stack—a cell on the stack has been marked, but its child pointers have not yet been examined. As cells are popped off the stack, their pointers are followed and the cells pointed at are pushed onto the stack, unless they have already been marked.

This *Stack-Collect* algorithm avoids the time inefficiency of Mark-During-Sweep—worst-case behaviour of the marker takes time proportional to the size of the live heap—

but introduces a space inefficiency: the additional memory needed for the stack in the worst case grows linearly with the size of the heap. Also, Yuasa's algorithm does *not* take advantage of the possibility of running both the marker and sweeper together incrementally.

7.2. A hybrid algorithm

A new algorithm combines the best time and space properties of Mark-During-Sweep and Stack Collect. Marking and sweeping are performed simultaneously, using an explicit stack to record the progress of the mark phase. However, a *very small* upper bound is set on the depth of the stack (for instance, one ten thousandth of the heap size). When the stack overflows, the grey bits are brought into action as "safety" bits. Once the stack is empty again, the collector searches for any grey cells and places them back on the stack. The algorithm is called *Stack-Safety.*

The idea is that for most graph structures, marking can be accomplished entirely through the stack. The *occasional* expense of an (incremental) search through the heap for a grey cell will be tolerable. Worst-case time behaviour is still quadratic in principle, though in practice most marking can be achieved in time proportional to the size of the live heap. As for memory costs, this hybrid algorithm requires three bits per heap cell (as in Mark-During-Sweep) plus a small constant for the bounded stack.

Assume now that the heap is a collection of `H` binary cells holding a `car` and `cdr`, which can either be pointers to other cells in the heap, or some other sort of value (Figure 7.1). For each cell there are also two mark bits (one for the marker, one for the sweeper) and a grey bit (for the marker). Global variables, `m` and `s`, record which bits each of the marker and sweeper is currently using. The stack has depth `D`.

```
value    :: tagged item
            | ptr to cell
cell     :: record of
               car  :: value
               cdr  :: value
               mark :: array [A,B]
                       of bit
               grey :: bit
heap     :: array [1..H] of cell
gcStack  :: array [1..D] of cell
gcSp     :: integer 0..D
m, s     :: enumeration A | B
freelist :: ptr to cell
```

Figure 7.1: GC: Heap implementation.

Note that a `ptr to cell` is effectively an index `1..H` into the `heap` array. That is, `hptr->car` is synonymous with `heap[hptr].car`.

7.2.1. The marker

The marker performs its work in small bursts of activity: using the stack as a scratchpad which persists between the bursts, it recursively traces the graph from the roots (Figure 7.2). When a cell is marked, it is placed on the stack. Having been marked, its sub-graph is traced by removing the cell from the stack and placing its *car* and *cdr* on the stack (and marking them), unless they are non-cell-pointers, or have already been marked. If the pop operation (Figure 7.3) reveals an empty stack, the marker looks for a grey cell to push onto the stack.

```
Km :: integer constant

marker =
    repeat Km times
        c := gcPop
        if c =/= EMPTY then
            gcPush(c->car)
            gcPush(c->cdr)
        else c := searchForGreyCell
            gcPush(c)
```

Figure 7.2: GC: The incremental marker.

The stack is bounded, so overflow is handled by the push operation (Figure 7.3): specifically, the grey bit records an item that cannot fit on the stack due to overflow. Such cells will be pushed onto the stack again at some later time when there is room for them, either when the stack is empty, or through being reachable from another cell on the stack. The push operation checks that its argument is a cell pointer, and that the cell in question is unmarked.

```
gcPush(c) =
    if not isCell(c) then
        return
    if c->mark[m] then
        return
    if gcSp < D then
        markCell(c)
        inc(gcSp)
        gcStack[gcSp] := c
    else shadeCell(c)
```

```
gcPop =
    if gcSp > 0 then
        c := gcStack[gcSp]
        dec(gcSp)
    else c := EMPTY
    return c
```

Figure 7.3: GC: Stack operations.

The mark bit operations are straightforward (Figure 7.4). It is an invariant that a cell cannot be both black and grey—the operations *markCell* and *shadeCell* enforce this. Maintaining a count of the number of grey cells eases testing for termination of the marker.

```
greyCount :: integer 0..H

markCell(c) =
    if c->grey then
        c->grey := FALSE
        dec(greyCount)
    c->mark[m] := TRUE

shadeCell(c) =
    if not c->mark[m]
    and not c->grey then
        c->grey := TRUE
        inc(greyCount)
```

Figure 7.4: GC: Safety-bit colouring.

The operation to search for a grey cell is shown in Figure 7.5. The search is incremental, performing a fixed small amount of work on every invocation (determined by the constant *Kg*). A persistent pointer records the current search position.

```
search :: ptr to cell
Kg     :: integer constant

searchForGreyCell =
    if greyCount > 0 then
        repeat Kg times
            inc(search)
            if search > H then
                search := 1
            if search->grey then
                return search
    return EMPTY
```

Figure 7.5: GC: Safety-bit search operation.

7.2.2. The sweeper

The sweeper also performs a burst of work on every call, using a pointer into the heap as its persistent state between calls (Figure 7.6). Mark bits which were set by the marker in the previous cycle are reset again for the next pass of the marker; unmarked cells are joined onto the freelist. Cells already on the freelist must not be swept to the freelist again—they are identified by their tag.

```
scan :: ptr to cell
Ks   :: integer constant

sweeper =
    repeat Ks times
        if scan > H then
            return
        if scan->mark[s] then
            scan->mark[s] := FALSE
        else
        if scan->car =/= FREE then
            scan->car := FREE
            scan->cdr := freelist
            freelist  := scan
        inc(scan)

incrementalGC =
    if scan > H
    and gcSp == 0
    and greyCount == 0 then
        swap (m,s)
        scan := 1
        for c in rootset
            gcPush(c)
    else
        marker
        sweeper
```

Figure 7.6: GC: The incremental sweeper and collector.

7.2.3. The collector

A routine is needed to co-ordinate the marker and sweeper, ensuring that they do not interfere (Figure 7.6). When the marker and sweeper have both completed all their work, the marker's bits are swapped with the sweeper's. The roots of the computation are marked to start the new ''wavefront''.

7.2.4. The allocator

A burst of GC is invoked on each allocation of heap storage (Figure 7.7). Every newly allocated cell is marked immediately.

7.2.5. The mutator

For every destructive alteration to a cell pointer performed by the mutator, the mutator must guarantee to mark the destination cell of the new reference. See Figure 7.7. In addition, any destructive update to the rootset must lead to a marking operation.

```
cons(l,r) =
    incrementalGC
    c        := freelist
    freelist := freelist->cdr
    c->car   := l
    c->cdr   := r
    gcPush(c)
    return c

rplaca(c,l) =
    c->car := l
    gcPush(l)

rplacd(c,r) =
    c->cdr := r
    gcPush(r)
```

Figure 7.7: GC: The allocator and mutator operations.

7.2.6. Discussion

When a cell is shaded grey, it does not necessarily follow that a search through the heap will be needed to find that cell again later. Even if one path to that cell from a root overflows the stack, another shorter path may exist from the same root or from a different root altogether. The cell, though grey, may still be marked through the stack, avoiding the need to search for it.

Ideally, the marker and sweeper should reach the end of their cycles together. The constants K_m and K_s, determining how much work will be done by the marker and sweeper in each call, must therefore be chosen carefully to give acceptable performance. The most appropriate values may depend on the program under evaluation. One possibility is to make them variables, and extend the garbage collector to adjust their values dynamically to redress any imbalance at the end of each phase. This approach may, however, have an unreasonable effect on the predictability of the algorithm.

The stack depth is another constant which may need to be tuned to suit the computation. If a single perfectly balanced tree fills the entire heap, a stack of $\log_2 H$ obviates any need for grey bits. In the worst case, a tree with a backbone stem and a single leaf at each node requires a stack of `H/2` cells to obviate grey bits. As the stack depth approaches this upper limit, we approximate Yuasa's Stack Collect. As it approaches zero, we approximate Queinnec et.al.'s Mark-During-Sweep. A compromise must be found which keeps both the stack and the grey count small.

The constant `Kg` will not depend on the particular program being evaluated - its value should be chosen solely to balance the two branches of the `if` statement in `marker`. A balanced conditional ensures minimum variation in the cost of a call to `marker`.

7.2.7. Optimisations

Possible optimisations of the algorithm (besides compiler techniques such as in-lining function calls) include:

(a) The mark bits associated with cells can be compacted into separate bitmaps, making bit operations more efficient, especially since virtual memory performance is not an issue. In particular, searching for a grey cell will be faster.

(b) The range in which grey cells lie might be marked by maximum and minimum pointers. If the grey count indicates a single grey cell, the max and min will be equal and point to that cell. If there are two grey cells, max and min will point to them both. Otherwise, searching need only be attempted within the indicated range. However this technique might interact badly with *(a)*.

7.3. Sketch of proof

This section sketches a correctness proof for the Stack-Safety algorithm. It argues: first, that the marker has marked (at least) every cell which is reachable from the root at the end of the marker cycle, and therefore that the sweeper collects no live cells; and second, that every unreachable cell is eventually reclaimed as garbage.

7.3.1. Unbounded stack

Assuming an unbounded stack, a static examination of the code justifies the following statements about the marker, within a single GC cycle:

- Once a cell has been marked, it cannot be unmarked again. (The only operation to clear a mark bit is in the sweeper.)
- For a cell to be marked it must be placed on the stack. Being on the stack implies a cell has been marked. (The only operation to set a mark bit is in `markCell`, which is called only from `gcPush`.)

- The roots are initially on the stack. (See the code for `incrementalGC`.)
- If the cell at the top of the stack points to another cell, the latter cell will be placed on the stack or already be marked. (`marker` calls `gcPush` on the first cell's `car` and `cdr`.)

Now, given termination, which says that the stack will eventually empty (see argument below), take the inductive step: for every cell which is on the stack at some time, the cells it points to will also be on the stack at some time. Further:

- If a cell's `car/cdr` is altered before it appears on the stack, the cell previously referenced through the `car/cdr` is no longer reachable by that path, and need not be marked.
- Hence, a subset of the cells reachable from the roots at the *start* of the cycle appears on the stack during the course of the cycle, each element of which is marked. Those cells which were reachable from the roots at the start of the cycle but have not been marked by the end of the cycle are unreachable at the end of the cycle.

Safety requires that all cells reachable from the roots at the *end* of the cycle are marked. These cells will be a subset of the cells reachable from the roots at the *start* of the cycle, plus any cells made reachable by the mutator *during* the cycle. But all cells allocated by, and all pointers destructively updated by, the mutator are placed on the stack. Hence, at the end of a GC cycle, the set of marked cells is a superset of the reachable cells at that moment, and an unmarked cell is indeed garbage.

Progress requires that any unreachable cell not on the freelist will at some time be identified as garbage and be added to the freelist. Reachable cells may be made unreachable by the mutator during a GC cycle (by destructive update of pointers). If at the moment of being made unreachable in cycle n, a cell has not yet been marked, it will remain unmarked to the end of cycle n and hence be reclaimed by the sweeper in cycle $n+1$. If it was already marked, it will remain marked to the end of cycle n. It will be unreachable from the roots at the start of cycle $n+1$, therefore it will be unmarked throughout cycle $n+1$, thereafter being reclaimed by the sweeper in cycle $n+2$.

7.3.2. Termination

As regards proof of termination of one cycle of the marker:

- Each iteration removes one cell from the stack (via `gcPop`).
- Any cell can be placed on the stack once only. (`gcPush` ensures that if a cell is already marked, it will not be placed on the stack again.)
- There is an upper bound on the number of reachable cells, namely, the heap size.
- Therefore the stack will eventually empty.

7.3.3. Bounded stack

To extend the argument to allow for a bounded stack with safety bits, the key is to show that a grey cell can be treated for the purposes of proof as if it were on the stack.

- Once a cell is grey, it can only become black, not white. (The only operation to reset a grey bit is in `markCell`.)
- Once a cell is black, it can become neither grey nor white. (The only operation to set a grey bit is in `shadeCell` which checks that the cell is not already black.)
- If a cell is on the stack, it is black. (`gcPush` ensures this.)
- The roots are initially either grey or on the stack. (`incrementalGC` calls `gcPush` on the roots.)
- If the cell at the top of the stack points to another cell, that cell will be placed on the stack or become grey. (`marker` calls `gcPush`.)
- When the stack is empty, any remaining grey cell is found and placed on the stack (by the `marker`.)
- Hence (given termination) every grey cell eventually blackens by transferral to the stack, and by induction, every cell reachable from the roots at the start of the cycle appears on the stack, and therefore is blackened.

The *safety*, *progress*, and *termination* arguments continue to hold when every grey cell can be treated as if on the stack.

7.4. Results

The Embedded Gofer interpreter, compiler, and example applications have been described in previous chapters. Using an early version of this system, before development of the alternative GC, the marble-sorter application was pushed to its maximum speed of marble-release, whilst still correctly meeting its deadlines. It ran at this speed only until the freelist was exhausted, at which point it collapsed due to the pause for mark-sweep GC.

The Stack-Safety algorithm was then implemented as a replacement GC for Embedded Gofer†. The application can now run continuously through any number of GC cycles without failure. However the Stack-Safety routines account for 30-45% of run-time! Since the system can now run at about only 55-70% of its previous speed, it is necessary to relax the maximum marble-release speed accordingly so the application can meet its deadlines again. The percentage overhead of Mark-Sweep is only 7-10%, but the comparison is not on equal terms, since Mark-Sweep causes the application to run incorrectly.

† Gofer is supplied with two stopping garbage collectors—mark-sweep and semi-space—and a straightforward interface allowing selection between them or a custom GC.

Experiments with the new algorithm varied the constant values for the number of increments in each burst Km and Ks, the stack depth D, and the heap size H. The value of the search constant Kg was fixed at 3 to balance the branches of the marker. A record was kept of: the minimum length f of the freelist during a cycle; the number of cells marked and reclaimed in a cycle (giving the approximate live heap size); the number of stack overflows; the maximum number of grey cells; the number of grey cells which are marked by sharing rather than searching, and so on. The following tables show measurements of such values during a typical GC cycle in one test program (the marble sorter of Chapter 4) which has a live heap of fairly constant size—about 2,250 cells (±40). In practice we would expect most embedded programs to have a live heap size that does not vary greatly.

Unfortunately, the particular version of Gofer used to obtain these figures (v2.21) stored a character cache, CAFs and other information in the heap, leading to a large rootset (about 650 root cells), so in fact about 1,500 cells of the live heap of this test program was static data. These static roots have been excluded from the stack overflow counts given below. In brief tests on a more recent version of Gofer (v2.28), this static heap data could be bypassed by the collector: in performance the static overhead was reduced, but there was little difference in the other measures.

The *work ratio* is a measure of parity between the work being done by the marker and the sweeper. *Work* is defined for the marker as the number of times it was called when there still remained grey or stack cells; for the sweeper, work is defined as the number of calls before it reached the end of the heap. Values above unity show the marker did more work; values below unity show the sweeper did more work. Ideally (for cost prediction), the marker and sweeper should complete their work in the same number of bursts. GC cycle time—and hence the minimum freelist size—is also affected by any disparity.

Table 7.1 shows that (for this test program) only a very small stack is needed to eliminate stack overflows completely when traversing the live graph.

H=40000, Km=4, Ks=20, Kg=3			
D	overflow	f	work ratio
1	396	16120	5.361
2	75	22175	3.669
3	1	28756	2.002
4	0	28343	1.999

Table 7.1: GC: Varying stack size.

Table 7.2 illustrates that varying the number of iterations in each burst of marking brings the work ratio closer to parity—achieved for this program when Km/Ks is 8/20.

H=40000, D=3, Ks=20, Kg=3			
Km	grey	*f*	work ratio
4	650	28756	2.002
5	650	30871	1.601
6	648	31649	1.334
7	650	32244	1.143
8	652	32699	1.001
9	652	32685	0.890

Table 7.2: GC: Achieving work parity.

Table 7.3 indicates that it is the ratio Km/Ks which determines parity. The absolute values of Km and Ks determine the time taken for a complete cycle, and hence the minimum heap space required.

H=40000, D=3, Kg=3			
Km	Ks	work ratio	*f*
2	5	1.001	21759
4	10	1.000	29648
6	15	1.001	31419
8	20	1.001	32699
10	25	1.001	33627

Table 7.3: GC: Keeping work parity.

Keeping the ratio Km/Ks constant, in Table 7.4 the heap size is varied downwards. Because the live heap size (and the marking task) stays constant whilst the sweeper has less work to do, the work ratio is affected.

D=3, Kg=3			
H	*f*	Km/Ks	work ratio
40000	32699	8/20	1.001
30000	23822	8/20	1.016
20000	15039	8/20	1.116
15000	10645	8/20	1.277
15000	10598	8/16	1.021

Table 7.4: GC: Reducing the heap size.

The amount of sharing (which enables safety-bit grey cells to be marked from the stack without needing a search) varies with the depth of the stack: $D=1$ gives an

occasional single sharing, whereas $D=2$ gives 18-30 sharings per cycle amongst an overflow count of 60-80 cells. This is a significant result because searching is very expensive in comparison to stack operations: the less searching is needed, the sooner a cycle can complete, and hence the smaller the heap can be.

For the Stack-Safety algorithm to deserve the tag of "real-time" it should be possible to predict within small margins exactly how much work is to be charged to GC, and when. It is clear that calls to Stack-Safety GC occur only on every cell allocation. The cost of a call is bounded by

$$\max\ ((KmM + KsS),\ R)$$

where M and S are the time taken for one iteration of the marker and sweeper respectively, and R is the time taken to mark the roots at the beginning of a new cycle. By aiming for parity in the work ratio, variation in the cost of a GC call is reduced, leading to greater stability. In real-time parlance, there is less jitter. For the cost of R to be roughly equivalent to the cost of marking/sweeping, techniques such as defining a "super-root", from which the entire rootset is accessible, may be required for particular machines†.

7.5. Conclusions

This chapter has demonstrated a memory management scheme for functional languages that is suitable in particular for embedded systems with small memory and a single processor.

The design choice to invoke an increment of GC at every memory allocation is a *scheduling* decision. The intuition is that when there is a high allocation rate, there should also be high collection activity to match, and when the allocation rate drops, so should the GC rate. A different choice might treat GC as a concurrent process‡, with a certain priority amongst the other processes in the system. Such a choice would have the advantage that every process goes faster when it runs, and that the total amount of time spent in GC is more directly observable by the embedded systems programmer. The disadvantage is that great care is needed in assigning a priority to the collector process to ensure that the freelist never runs dry.

Results show that a very small stack is often sufficient for incremental GC. Even when the stack overflows, a good proportion of the overflowed cells do not incur the extra time-cost of a safety-bit search. Aiming for parity in the work ratio (which must be determined for each application), increases the average length of the freelist: hence the program can run with a smaller overall heap. However, altering the heap size has a knock-on effect

† The implementation in Embedded Gofer uses such a super-root.
‡ The former scheduling choice was made in Embedded Gofer because the GC algorithm was developed before the process extensions.

on the work ratio, so the three variables K_m, K_s, and H must be adjusted together.

Tests were carried out on a program with a roughly constant live heap. This is an ideal situation for an embedded system. Under these conditions, Stack-Safety consistently marked about twice the actual live heap, and the freelist was maintained at about four times the actual live heap. This means that the total heap size could have been reduced even further.

On the other hand, programs with a widely varying live heap demand would require more headroom: a bigger average freelist and therefore a larger heap. However, future analysis of monadic combinators and their use in embedded systems programming may find ways of constraining embedded programs to a near-constant live heap demand.

Stack-Safety's space performance is very good, but its time performance is disappointing. Although it demonstrated acceptable speed for the case study in Chapter 4, the case study of Chapter 6 showed that when many events happen in rapid succession, the garbage collector can be a hindering factor in dealing with them in a timely fashion. It is possible that scheduling GC as a process, rather than at every memory allocation, could help solve this problem. Also, the implementation is pretty much un-optimised, and it should therefore be possible in future work to reduce the absolute time overhead. It seems unlikely however that the percentage of time chargeable to Stack-Safety could be reduced below, say, 15–20% (roughly half its current cost).

Chapter 8. Real-time issues

8.1. Introduction

This chapter considers one important class of applications that is closely related to embedded systems – *real-time* systems. There are several competing views of what exactly "real-time" means.

1. *Any reactive system is a real-time system.* A reactive system is one that involves interaction with real world entities, and it can be classed as real-time because those external entities place time-related demands on the system. A graphical user interface, for instance, would fall under this definition because the user can get confused or impatient if no response to some action is evidenced within a second or two. Almost every embedded system satisfies this criterion too. This view is sometimes characterised as: *real-time* equals *real fast.* It will not be considered further in this chapter.

2. *A real-time program expresses timing information directly or indirectly* in its algorithm, in monitoring inputs, or in controlling outputs. For instance, Embedded Gofer allows a timer device to be programmed that can serve timing information to other processes. There are numerous other possible means of expressing time in programming languages. Some functional approaches are reviewed in Section 8.2.

3. *The computational correctness of a real-time system depends on the timely production of its results.* This is the most rigorous view of real-time. Program correctness is measured both by the logical correctness of the results and the ability to meet deadlines [PanzieriDavoli93]. This third view is not necessarily incompatible with the second, but it is certainly very different in style and in the linguistic approach needed. For instance, in the liftshaft case study in Chapter 6, the controlling process for a car has to react to an interrupt from the telemetry wheel within 0.2 seconds, which is the minimum time before the next interrupt. This constraint is not expressed anywhere in the program: indeed, there is no way of expressing this sort of timing information in Embedded Gofer. This third view is treated in Section 8.3.

There are numerous calculi which attempt to formalise notions of time; some are timed extensions of process algebras such as CSP and CCS (e.g. [Moller&Tofts89] [Schneider90]), others are based on timing extensions of specification methods (e.g. [DukeSmith89]). At least one formalism [ScholefieldZedan92,93] appears to bridge the gap between the second and third views of real-time: the calculus can be seen either as a specialised programming language allowing time-expression, or as a formal analysis technique amenable to transformation and proof. However, such formalisms are outside the scope of the present work and are not discussed further.

The chapter concludes with a brief summary of the issues and the state of the art in functional languages. This chapter does not seek to add any new treatments or techniques to a very diverse field; rather, it intends to highlight the immense scope for future work in the real-time area.

8.2. Timing expression

This section examines the specific linguistic requirements of time expression and reviews several techniques and languages which have sought to address those requirements in a functional setting.

Given that programs should be able to base control or computational decisions on timing information gleaned at run-time, the programming language should allow the explicit expression of time. Timing specifications might be defined in terms of precise instants or bounded intervals (''windows of opportunity'')—the latter gives the underlying system greater flexibility in coming up with a schedule which meets all the demands. There are four possible sorts of expression:

(i) *Scheduling.* A computation or action should begin after a stated time.

(ii) *Deadlines.* A computation or action should be completed before a stated time. Deadlines may be hard or soft, that is, it may or may not be critical that the deadline is met precisely.

(iii) *Monitoring.* Given actual timings of events, timing errors can be detected and corrected.

(iv) *Timeouts.* These express the passing of time in the absence of other events. They provide information about the non-occurrence of an event, allowing detection and correction of a different class of errors.

It is common for languages to provide program constructs that allow scheduling, but uncommon to see deadline constructions. It has been said, in the context of some real-time experiments using Modula [Wirth77]:

> "Clearly the ability to specify that an action should occur *after* a given period is not enough; it is the essence of real-time performance that actions occur *before* a period elapses" [Runciman81].

Or, more succinctly, "I don't want to `delay`; I want to hurry up!". Burns and Wellings [BurnsWellings89] state that

> ''although deadline scheduling is in many ways the essence of the real-time domain, it is dealt with very inadequately in most (so-called) real-time languages''.

It could be argued that timing specifications might fit more naturally and appropriately into a declarative language than a sequential language, since timing information is essentially declarative. Subsections 8.2.1–8.2.9 examine several functional models: a clock-reading approach; stream approaches; timestamping; some dataflow languages; new time-varying types; a functional/imperative hybrid; and the commercial "real-time" language Erlang. Each model is evaluated against the four requirements described above, and Subsection 8.2.10 summarises.

8.2.1. Clock requests

Embedded Gofer provides the ability to program a server process to interact with a timer device, as illustrated in the case studies in Chapters 4 and 6 (especially §6.4). This model gives the same functionality as a set of clock requests added to the standard Request/Response I/O model would.

The ability to *schedule* events is clearly present, but although the programmed `delay` request causes the delayed actions to occur after the delay expires, *how long* after is not precisely known. In Embedded Gofer, it depends on the number and priorities of other processes in the runnable state. There is no way to impose *deadlines*. *Monitoring* time is possible if the server is written to measure elapsed time, as shown in §6.4.3. This measurement ability is not very precise however, for the same reason as `delay` is imprecise. *Timeouts* can be programmed by simultaneously awaiting an event and a message from the timer, as shown:

```
timeoutOnInterrupt t intrptAction timeoutAction =
    defer t Self Timeout >>
    select (\Interrupt -> intrptAction)
           (\(Alarm,Timeout) -> timeoutAction)
```

Greater timing precision could be achieved in Embedded Gofer by treating the alarm server specially. Messages to and from the alarm server could override the normal priority scheduling policy, and cause the server or client (as appropriate) to execute immediately. However, analyses which guarantee real-time properties (such as those in Section 8.3) may prove to be a better route to precision.

8.2.2. Timed evaluation of streams

A different method of dealing with time in a functional language is through the choice of a particular evaluation strategy. *Eager* evaluation requires sub-expressions to be computed as they are encountered, whether or not they are actually needed. Under *lazy* evaluation, values are not computed until and unless they are needed. A compromise strategy is *time-driven* evaluation. The "future" is a lazily constructed part of a list, but the "past" is the fully evaluated part of the list.

Change to an individual data item over time can be represented as an infinite stream of different items. Time is the index to the stream, and there is a constant time interval between items. For example:

data:	15	3	-27	9	4	128	76	75	...
time:	1	2	3	4	5	6	7	8	...

This is similar to the model of time in the dataflow languages LUSTRE and SIGNAL (cf. §8.2.6). *Scheduling* the start of a computation is not immediately obvious, but can be accomplished by a producer/consumer pair. The producer must deadline the production of a ''start'' flag, and the consumer must perform the computation in question only after reading the flag value. *Deadlines* are implicit in the fact of producing list items in a particular order – at every time instant, some output value is definitely fully evaluated. *Monitoring* too is implicit in the ordering of input values. *Timeouts* are possible with *hiatons.* If no actual value is available on a input or output stream, a special object called a *hiaton* can be returned. For example:

data:	15	hiaton	3	-27	hiaton	hiaton	9	4	...
time:	1	2	3	4	5	6	7	8	...

Hiatons are normally discarded by the reading process, unless it was expecting a genuine value in which case a timeout-recovery action can be taken. Unfortunately, a state of *busy-waiting* can occur, where much work is wasted in processing nothing but hiatons. Also, because every function must contain code to enable hiatons to propagate through the entire system, the source definitions tend to look rather cluttered.

The major disadvantage of the timed stream model however is that time is not mentioned *explicitly*, it is a matter of interpretation. While the full power of semantic reasoning can be applied to the functional aspects of the computation, a separate model must be built for its timing characteristics.

8.2.3. Timestamped data in streams

One way to improve upon implicitly-clocked streams is to place explicit timestamps as data items within the streams. Hence a stream consists of a list of time/data pairs: an input stream pairs values with the time they became available from a producer; and an output stream pairs values with a specification of when they are to be made available to a consumer. For example:

(time,data): (1,15) (3,3) (4,-27) (7,9) (8,4) (10,128) ...

The functional languages ART [Broy83] and Ruth [Harrison87,88] take this approach, which has similarities to the real-time LUCID dataflow method (§8.2.4). Timestamps can be manipulated in the language by isomorphism with the natural numbers. *Scheduling* the start of a computation is performed in the same producer/consumer manner as in Section 8.2.2. *Deadlines* and *monitoring* are clearly possible. Note however that timestamps on input streams only indicate when that particular value became available, they do not necessarily indicate the current time. This makes precise scheduling and monitoring more difficult.

For *timeouts*, the language Ruth provides a primitive `Ready(chan, time)`. It returns `True` immediately data is available on the channel, or `False` when it times out at, and no later than, the given time. `Ready` is necessarily implemented as a primitive because programs do not have direct access to the "real" clock—only to timestamps. The primitive can essentially be viewed as checking for hiatons. In the following example, the consumer requests a data value which is not available before a timeout occurs at $t=6$.

(time,data): (1,15) (3,3) (4,-27) (6,hiaton) (7,9) (8,4) ...

The advantage of timestamped streams over implicitly-clocked streams is that timing information is just another type of data in the program, and can be reasoned about in the usual manner. The `Ready` function however has a more esoteric semantics, which deserves brief discussion.

Ruth's semantic interpretation is based on a *herring-bone domain*. This is a product-domain: a lattice built from a cartesian product of two sets, and a partial ordering. The ordering is based on the time domain. An example herring-bone domain for Boolean values is shown in Figure 8.1.

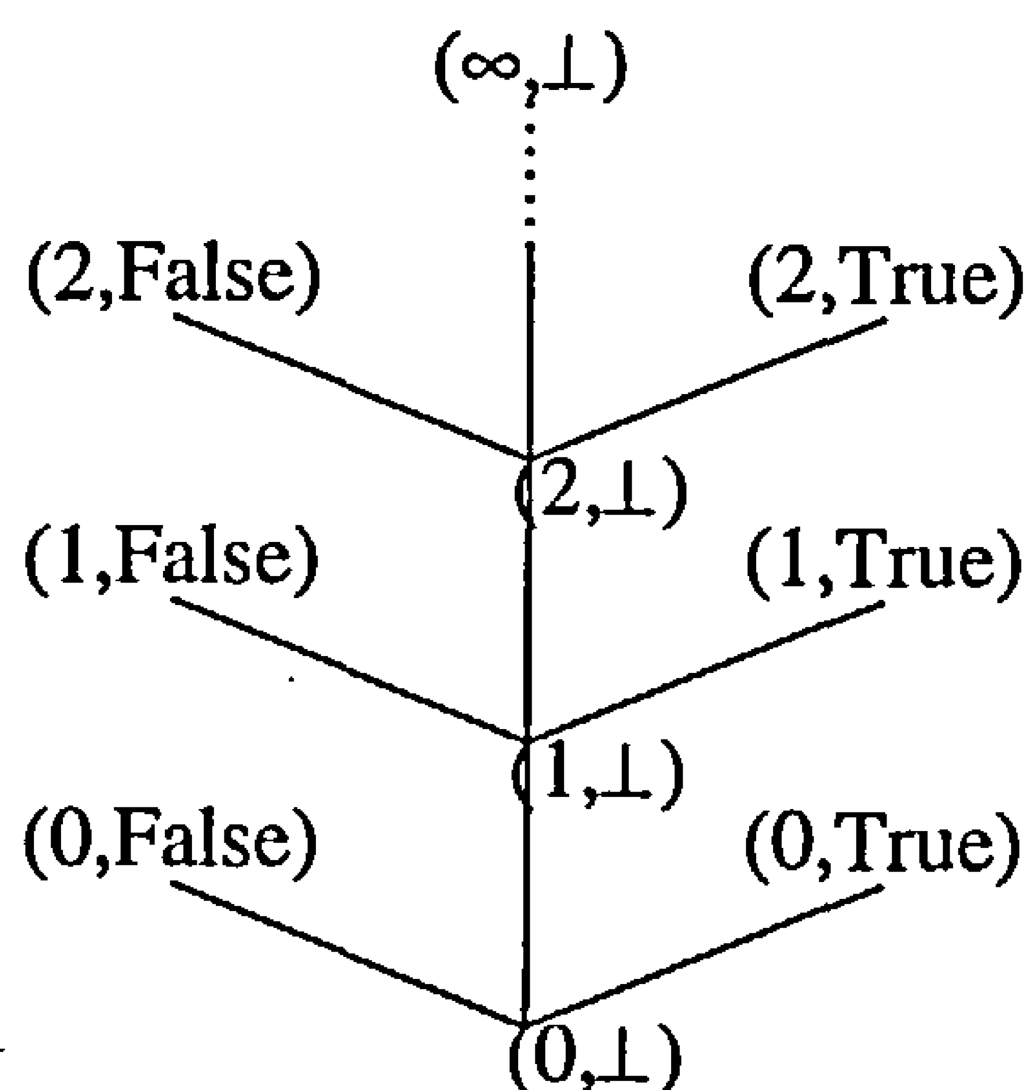


Figure 8.1: A Boolean herring-bone domain

Here, the cartesian product is `Time x Lift(Boolean)`. *Lifting* the Boolean set introduces the $\perp$ (bottom) element to indicate an undefined value. Harrison's ordering to build the domain is:

$$\forall\, t_1, t_2 \in \mathit{Time},\ b_1, b_2 \in \mathit{Lift(Boolean)},$$
$$(t_1, b_1) \sqsubseteq (t_2, b_2) \Leftrightarrow (t_1 = t_2 \wedge b_1 \sqsubseteq b_2) \vee (t_1 \leq t_2 \wedge b_1 = \perp)$$
$$\text{and } (t_1, \perp) \sqsubseteq (\infty, \perp)$$

In operational terms, the computation progresses up the spine until it completes, when it takes one of the branches. The non-terminating computation has the semantic value $(\infty,\perp)$. Broy [Broy83] finds this pleasing, since it is the topmost element of the product-domain. It contrasts with semantic models which exclude time, where the non-terminating computation has the value $\perp$, the bottom element of the domain. It seems appropriate to the operational understanding that a non-terminating computation should move through the lattice, passing potential terminating values though not stopping at any of them, rather than remain at the foot of the lattice.

A herring-bone domain can be built for each the standard programming language types. Harrison presents a full denotational model for constructing these product-domains and mapping Ruth programs onto them [Harrison88]. However, they become ever more complex as the types become more structured. Fijma and Udink [FijmaUdink91] dismiss this semantic model as being too complex to be useful. Harrison himself admits that the semantics of the `Ready` function are far from clear: the semantic equation for `Ready(c, t)` defines that it should be able to match a value from the herring-bone channel `c` against the pattern $(t,\perp)$. Testing for $\perp$, though operationally straightforward in this case, is rather suspect semantically.

8.2.4. Dataflow model: time windows in LUCID

In a dataflow graph [WadgeAshcroft85] each node is a function from its set of inputs to its set of outputs. Each sub-graph can therefore also be viewed abstractly as a node. A stream of input data items is pumped through the whole graph on each of its input arcs, yielding a stream of output data items on each of the graph's output arcs. A dataflow language is a means of describing the graphs textually. The only way in which a *state* can be stored is by feedback arcs in the graph, much like recursion in functional languages.

The dataflow language LUCID [AshcroftWadge77] [WadgeAshcroft85] has been extended to permit time expression, by associating a stream of *time-windows* with each input and output stream [FaustiniLewis86]. There are no time-windows attached to internal streams, since these should be as unconstrained as possible in trying to meet the external deadlines. This model is very close in spirit to the timestamped model in §8.2.2.

A time window is a pair `[a,b]`, specifying the earliest and latest time at which its associated value will be available. Either (or both) elements of the pair can indicate an open-ended time specification with a star, e.g. `[a, *]`. During execution, it may occur that the machine cannot generate a value within the time window associated with the expression, or it cannot read an input value within the given time window. In this case, a special object, `tf` indicating a time fault, is emitted instead of the value. Time faults, rather like hiatons, propagate appropriately through the dataflow network. A unary predicate `istf` can be used in the network to check for such faults.

Explicit *scheduling* is not possible at all in this model because time specifications are not permitted on internal graph arcs. *Deadlines*, *monitoring*, and *timeouts* however are explicit.

8.2.5. Dataflow model: timing nets in LUCID

Skillicorn and Glasgow take an alternative *transformational* view of real-time specification using LUCID [SkillicornGlasgow89]. A LUCID program describes an operator net. Simple transformations on the net can produce two more nets, called EARLY and LATE. Together with a specification of the earliest and latest times at which inputs will be available, and the earliest and latest times at which outputs must be produced, these nets solve equations for the pointwise execution time of the individual components of the net. Or, given pointwise execution times, and either of the input or output constraints, the nets solve for the unknown input/output characteristics. The EARLY and LATE nets are, of course, LUCID programs, and can hence be executed. One advantage of this approach is that timing problems can be located precisely within the net. The real-time specification can be dealt with separately from the program (statically), or integrated with it (dynamically).

This model allows *scheduling* and *deadlines* very explicitly. Although timing faults can be located, *monitoring* of actual timings is not possible and *timeouts* are not permitted, because the model is predictive, not reactive. In this way, it has some similarities to the more rigorous view of real-time discussed in section 8.3.

8.2.6. Dataflow model: strong synchrony

The language LUSTRE [CaspiHalbwachs86] [Caspi+87] [Halbwachs+91] is a descendant of LUCID, but uses a different mechanism again to express time, based on synchronous streams but with some novel features. Essentially, the input data and most operators in the dataflow network are clocked at a known rate (the basic clock), but a stream of booleans can be used as a slower, erratic clock inside the dataflow network. The construct `x when b` describes this: the value of `x when b` is only defined when `b` is `true`, that is, the boolean stream `b` clocks the data stream `x`. This is known as *undersampling*.

The reverse construct, known as *oversampling* or *interpolation*, is `current(y)`, and it restores the basic clock. If *y* is undersampled based on a boolean stream *b*, then when *b* is `False`, *y* is undefined. However, the value of the expression `current(y)` *is* defined at instants when *b* is `False`: its value is the most recent value of *y*.

The SIGNAL language [LeGuernic+86,91] [Gautier+87] [LeGuernicGautier91] is very similar in conception to LUSTRE, and is directed at signal processing applications. Both languages bear similarities to the timed evaluation of functional streams, outlined in §8.2.2 above. They add some extra timing apparatus however, and SIGNAL is provided with both a comprehensive algebraic model of timing properties [Benveniste+91], and a timing model based on abstract interpretation [Jensen94]. These formalisms are a very important addition to the field.

However, to *schedule* a specific computation to start at a specific time, the producer/consumer trick of §8.2.2 is needed. Also, whilst this dataflow model is good at specifying *deadlines*, LUSTRE is not capable of *monitoring* whether those deadlines have been met. In neither language is it possible to recover from missed deadlines: there is no way to express *timeouts*. Indeed, languages like LUSTRE and SIGNAL invoke the *strong synchrony hypothesis* [Harrison88], which says that machine operations take no observable time, and hence, each clock tick will definitely cause a value to be transmitted along each graph arc. This assumption seems unrealistic.

8.2.7. Time description in musical languages

Arctic is a functional language designed for musical applications such as sound synthesis and signal processing [Dannenberg84,90a] [Dannenberg+86] [RubineDannenberg87] [DannenbergFraley89]. Dannenberg points out that for centuries, musicians have been using a real-time declarative language – the musical score – which embodies the concepts of loops, counters, conditionals, and other features familiar to programmers.

Arctic introduces a fresh way to make time explicit in a functional language: *time-varying types*. It provides two new types: time-varying Booleans and time-varying Reals. Time is conceptually continuous – values of the time-varying types can be thought of as square waves or analogue signal waves. So for instance, in the expression

```
a = b + c
```

the value of the signal `a` at time `t` is the sum of `b(t)` and `c(t)`. Signals can be built together into sequences:

```
[ a | b | c | d ]
```

(every value carries with it a start-time and a duration, so for instance the start-time of the sequence above is the start-time of `a`, and its duration is the sum of the durations of `a`, `b`, `c` and `d`), or parallel collections:

`[ a ; b ; c ; d ]`

(the parallel collection's start-time is the earliest of the individual start-times, and its duration is the maximum of the individual durations when added to their respective start-times).

The operators `@` (shift) and `~` (stretch) alter the time characteristics of these types of values. *Shift* alters the beginning-time of a value, while *stretch* alters the duration of the value. Table 8.1 illustrates their effects where `a` has start-time `0` and duration `1`.

expression	start-time	duration
`a`	0	1
`a ~ 2`	0	2
`a ~ 2 ~ 3`	0	6
`a @ 5`	5	1
`a @ 5 @ 3`	8	1
`(a ~ 2) @ 5`	5	2
`(a @ 5) ~ 2`	10	2

Table 8.1: The effect of Arctic operators.

Primitive waves such as `ramp`, a smooth gradient from `0` to `1` over the time interval `(0,1)`, are used to build up more complex waveforms. There is notation for conditionals and loops, and a process can (for instance) block waiting for an interrupt on a time-varying Boolean value supplied by the external world.

The language permits *scheduling* of events by shifting a function into the future. *Deadlines* can be expressed in as much as output values are functions of time, and hence if an actual output value does not match its programmed specification at any particular time, the compiler is wrong. *Monitoring* of events is provided because input values are time-varying. *Timeouts* can be programmed by shifting a Boolean value into the future, on which a process can then wait.

Reasoning about Arctic programs is expected to be much like reasoning about continuous functions in mathematics, although how for instance the differential calculus might be used is unclear. Dannenberg gives a small number of transformations for understanding the temporal characteristics of the language constructs in terms of stop times and durations [Dannenberg90a].

A non-real-time interpreter for Arctic [RubineDannenberg87], and a real-time run-time system [Dannenberg90b] exist, although there is no compiler – example programs are hand-compiled. Dannenberg explores variations on the Arctic theme in the languages Canon [Dannenberg89] and Fugue [Dannenberg+91], which are explicitly designed for musical description and manipulation, although at different structural levels from Arctic

itself. They are basically extensions to an interpreted LISP, but while they express time well, real-time implementations do not exist.

Haskore [Hudak+94] is a newer functional language allowing time expression for musical description – it is a Haskell library of abstract data structures. It is composition-oriented rather than reactive, that is, a musical ''program'' must be pre-executed to produce a ''performance''. The performance can be replayed in real-time, but it does not interact in any way with the external world during performance. Thus its potential use in more general real-time systems is blocked.

8.2.8. A functional/sequential hybrid

A hybrid approach to introducing time-expression to a functional language is taken by Harrison's language STRuth [HarrisonNelson89] [Harrison91]. A functional language is used for computation, but to it is added a sequential behavioural language, based on the occam/CSP model. In STRuth, assignment statements are permitted, as are loops, conditionals, and I/O statements. I/O is over fixed channels, communicating between fixed processes. No explicit *scheduling* is provided, except by sequence. *Monitoring* is not provided either. The only timing construct apart from sequence is the `timeout` block, which provides both *deadlines* and *timeouts.*

```
x := E1 timeout T1 continue with
     E2 timeout T2 continue with
     E3 timeout T3;
```

The `timeout` block has a fixed limit (the sum of the timeouts), and the final value of the variable in the assignment will be the first of the constituent (functional) expressions to complete. The idea is that the different expressions are successively poorer approximations to the value the programmer really wants. So for instance, *E1* is a very precise but complicated expression. If it fails to complete within *T1*, then *E1* is abandoned and *E2*, a less complicated expression, is begun. If this too fails to complete within the given time *T2*, then *E3*, a very simple calculation, is invoked.

Although this timeout facility is very precise, giving precise and guaranteed deadlining as well, timeouts can only apply to functional computations and not to the arrival of messages on channels, so the model is very restricted. The rest of the language does not seem to allow expression of time at all.

8.2.9. Erlang

The "almost functional" language Erlang claims to be suitable for programming large-scale fault-tolerant real-time applications [Armstrong+93]. It has two facilities for expressing time: a non-deterministic pseudo-function, `time()`, which returns a tuple representing the current time to a one second resolution; and a timeout inside the message-receiving conditional.

Scheduling is implemented as a special case of a timeout. No facility for attaching *deadlines* to actions is available in Erlang. Calling `time()` allows a certain degree of *monitoring* despite its poor resolution, but its non-determinism is a distinct hindrance (see Chapter 2.5). The *timeout* facility is however very flexible. When awaiting an incoming message, there is a conditional match on the content of any messages received. The final clause of the `receive` construct specifies a timeout precise to the millisecond, after which message-waiting is abandoned.

```
receive
    message1 -> do_action1
    message2 -> do_action2
after t -> do_action3
end.
```

Scheduling an action can only be implemented by a special case of awaiting a message, where there is no content-matching. Hence, the timeout clause is necessarily followed, and this contains the action to be scheduled.

```
schedule(action,t) =
    receive
    after t -> action
    end.
```

8.2.10. Summary

In summary, there are four major ways of expressing time in a purely functional manner (discounting hybrid languages and non-deterministic methods).

(i) Clock requests can be added into a program's I/O facilities, returning a time value on a process's input stream. This model is easily introduced to most existing functional languages, whether in matched stream, continuation, or monadic style.

(ii) A particular data evaluation strategy can be adopted in which there is a constant time interval between data items on input streams. Other streams derived from the input streams can have non-constant time intervals between the data items – work is still continuing at INRIA in France on SIGNAL's dataflow time model.

(iii) Streams of input time values can complement input data streams. It is interesting to note that Harrison abandoned the prospect of programming real-time in a purely

functional language following his experience with time-stamped streams in Ruth, and took up the hybrid STRuth model [HarrisonNelson89]. This was primarily because of difficulties with the semantic interpretation of timeouts.

(iv) Lastly, time-variance can be modelled as a basic feature of individual data items. This continuous model of time was pioneered by Dannenberg. Rather than constructing elaborate herring-bone domains for semantic analysis, traditional, well-understood, mathematical tools such as the differential calculus could be sufficient for reasoning about programs.

Table 8.2 illustrates how these language models match up to the time-expression requirements outlined at the beginning of this section.

Language model	scheduling	deadlines	monitoring	timeouts
clock requests	✓	✗	✓	✓
clocked streams	?	✓	?	✗
time-stamped streams	?	✓	✓	✓
time-varying types	✓	✓	✓	✓

Table 8.2: Summary of time expression characteristics

Most of the languages covered in this review have simulators, but few have yet reached the stage of linking *program*-time to *real*-time (Erlang is a notable exception). Clearly there is still much work to be done in implementing the technology, both to achieve and to guarantee the desired timing properties. Those models allowing both deadline specification and lazy evaluation (e.g. Ruth) are probably the hardest to implement.

Another important question to ask of each linguistic model is whether there exists a formal semantic model for understanding the timing aspects of programs. We will not address this question here, as semantic modelling of time is a huge field still under active research.

8.3. Deadline guarantees

In many applications, timing of a program's execution is critical. A ''correct'' output value produced too early or too late is not correct at all. *Predictability* of the timing properties of a given program is essential. That is, some guarantee should be given, before the system runs, that the correct value will be produced at the correct time. This section discusses general approaches to guaranteeing timing properties of systems (*deadlines*), and asks whether it is possible to achieve such guarantees in a lazy functional language. Both analysis and implementation techniques are needed.

This section describes the common terminology of critical real-time systems (§8.3.1), and briefly reviews some scheduling analyses that aim to guarantee the timing properties

of programs (§8.3.2). We then discuss how such analyses might be integrated with functional languages (§8.3.3), and how they might interact with lazy evaluation (§8.3.4).

8.3.1. Classifications of real-time systems

Real-time systems (under the third definition of *real-time* in Section 8.1) fall into two broad categories: *hard* and *soft*. Hard real-time systems are those where failure to meet a deadline can result in a catastrophic system failure. Soft real-time systems are those where failure to meet a deadline does not result in a system failure – although missing the deadline is an error, the error can be tolerated to some degree. In most real-time systems, there is a mixture of hard, soft, and non-real-time components.

Real-time applications are generally modelled as a set of communicating sequential processes [Burns91]. Each process, or *task*, can be classified as hard, soft, or non real-time. In addition, each task can be classified as *periodic*, or *aperiodic*. Periodic tasks execute at regular intervals of time. Aperiodic tasks execute in reaction to internal or external events. If an aperiodic task is to be assigned hard deadlines, it is required that the maximum frequency of occurrence of the triggering event be known. A hard aperiodic task is often called *sporadic*. Each task in a system therefore has four attributes: period, offset†, deadline, and execution time.

8.3.2. Brief review of scheduling analyses

There is a wide range of scheduling analyses for periodic tasks, and run-time scheduling policies corresponding to them [LiuLayland73]. All operate by assigning priorities to tasks. *Static* policies fix the priority of each task before run-time; *dynamic* policies compute changing priority values for each task at run-time. It is generally accepted that hard real-time systems require a static timing analysis [Shaw89].

Pre-emptive scheduling policies allow a task to be displaced from execution by another with a higher priority. Such policies include *Rate Monotonic*, *Earliest Deadline First*, and *Least Slack Time First*. Rate monotonic scheduling assigns fixed priorities according to task period: the shorter the period, the higher the priority. Earliest deadline scheduling assigns dynamic priorities: the sooner the task deadline, the higher the priority. Least slack time scheduling assigns dynamic priorities: the smaller the difference between the available time for a task to run and its required execution time, the higher the priority. Rate monotonic scheduling has been shown [LiuLayland73] to be *optimal*, that is, if a set of tasks can be scheduled feasibly under any policy, it can be scheduled under the rate monotonic policy.

† If period corresponds to frequency, then offset corresponds to phase.

Scheduling sporadic tasks is more complicated [Lehoczky+88]. *Background* scheduling allows sporadic tasks to run only when there are no periodic tasks executing. *Polling* scheduling turns a sporadic task into a periodic task with a fixed priority based on the rate monotonic analysis: its period is taken to be the maximum frequency of the triggering event. Inevitably this policy often leaves the actual processor utilisation rather lower than an analysis of periods and execution times would suggest.

Both *Priority Exchange* and *Deferrable Server* scheduling aim to address this problem, by having one high-priority periodic task which encapsulates all the intended sporadic tasks. The server task responds to all the pending triggers within its period provided its total execution time does not exceed its allocation. If its execution time is exceeded, the pending events are held over until the next period. While no trigger events occur, lower priority tasks are free to to run. A priority exchange server exchanges its dynamic priority with the next-highest task at the beginning of its period, provided no triggers are pending at that moment. A deferrable server keeps its static priority throughout, enabling events which occur after the beginning of its period to be serviced immediately.

All these analyses require program annotations for period, deadline, and offset, and some means of determining task execution time. In practice the analysis can be automated: a tool takes a task set, analyses it, and returns a statement of whether a feasible schedule could be found, i.e. whether every task can be allocated its complete execution time within its period, after its offset and before its deadline, within the context of all the other tasks.

Determining execution times of tasks is difficult. Often exact execution times cannot be found, and a *worst case execution time* (WCET) must be assumed, although finding a realistic WCET that is not overly pessimistic can be difficult too.

8.3.3. Guaranteed scheduling in a functional language?

In an ideal world, the tools for analysing a program with respect to its required real-time characteristics would be fully integrated with the language compiler. Parts of the compiler could then concentrate optimisation effort on certain sections of the program in order to reduce their execution time and hence satisfy the analysis if at all possible.

It does not in principle seem to be a large extension to a functional language to add timing annotations. The structure of Embedded Gofer already divides a program into tasks, and the various sorts of `loop` construct express both periodic and aperiodic tasks. There are two possible ways to implement scheduling based on annotations.

(i) Use the timing server programmed in Embedded Gofer to construct a dynamic run-time schedule.

(ii) Subsume the timing server into the run-time system, and construct a static schedule for the processes, to be implemented at run-time by a simple priority dispatcher.

Using the run-time alarm process.

The first option could be implemented by higher-order definitions extending the `loop` constructs. For instance, a typical periodic process behaviour is encapsulated in the `periodic` combinator.

```
periodic :: Process a p m => Time -> a p m () -> a p m ()
periodic t body =
    loop (send Alarm (Timeout t) >>
          body >>
          recv ? \(Alarm,Expired) -> result ())
```

An aperiodic process behaviour, triggered by interrupts, is also easily coded.

```
intAperiodic :: IntProcess a p m => a p m () -> a p m ()
intAperiodic body = loop (getInterrupt >> body)
```

This suggested mechanism places the burden of scheduling into the program itself, and hence is very run-time oriented. In addition to a compile-time analysis of tasks for schedulability, it requires a reasonably sophisticated programmed run-time alarm server, which introduces the inevitable overhead of its implementation in a very high-level language.

Static scheduling.

The alternative is a compile-time analysis complete enough that only an extremely simple run-time scheduler, linked to the embedded clock, need be included in the run-time system. The program still uses higher-order functions like `periodic` and `intAperiodic`. However, these are not implemented by message-passing to an alarm server: they are instead used at compile-time as annotations to guide the analysis, and at run-time are implemented as a simple context switch to the dispatcher. Under a rate monotonic analysis for instance, the dispatcher contains a fixed priority pre-emptive algorithm, much like the current run-time scheduler in Embedded Gofer, and the alarm server process is eliminated altogether.

8.3.4. Laziness

The big question is whether a timing analysis of a lazy functional program is possible at all. A seemingly simple value can be held internally as a huge graph requiring much reduction, and hence much execution time. Static determination of the space and time needs of lazy functional programs is still an open problem.

Especially in a process architecture it is all too easy for one process to construct a lazy closure and pass it unevaluated to another process. Since all the analyses described in §8.3.2 are explicitly based on the individual execution times of separate processes, this situation would be intolerable. There are however several observations that can be made about Embedded Gofer programs.

(i) Message-passing between processes could be made hyperstrict. That is, a message's content could be fully evaluated before it is delivered. This simple change to the run-time system guarantees that the evaluation work for a message can always be charged to the sender. It may also be necessary to prohibit sending functions as values inside messages – in KAOS this was explicitly disallowed [Turner87].

(ii) For a process which follows a `loop` pattern without any local state, any evaluation work depends solely on the values of received messages. The messages are pre-evaluated by the sender. A WCET analysis can therefore assume that all such values are in normal form when determining the execution time of the loop body.

(iii) For a process which follows a `loopwith` state pattern, the local state could be evaluated hyperstrictly at the end of each loop body. Thus, WCET analysis of the loop body can again assume that all initial values are in normal form. (A space optimisation here is that the local state could be updated in place, since it is freshly bound at the beginning of each iteration. However, if in-place update is used and any part of the local state is sent in messages, the message must contain a copy rather than a shared reference.)

These measures would eliminate closure build-up from two of the commonest sources in process-based programs: messages and local state. Real-time analyses insist that processes be either periodic or aperiodic: the two loop constructs `loop` and `loopwith` cover both of these cases, and hence the measures just described would have a far-reaching effect on the analysability of real-time Embedded Gofer programs. The remaining question now is whether a WCET analysis is possible inside the bodies of the loops. Here, recursion is the main obstacle. In imperative real-time languages, conditional iteration must often be annotated with numeric bounds. Annotations could similarly be added to functional recursion too. Type-inference technology may one day be able automatically to determine bounds for many instances of recursion [Hughes94].

8.4. Conclusions

For many embedded applications, it is not enough that the software should merely run "fast enough". It is often important to express timing information as an integral part of the program. This entails scheduling computations, setting deadlines, monitoring the clock time, and specifying timeouts on certain events. However, this too is often not enough. Real-time software must be *guaranteed* to meet the schedules and deadlines specified in the program text.

In a functional setting, several linguistic means of expressing time have been proposed, meeting the requirements with varying degrees of success. Deriving *guaranteed* time and space bounds for lazy functional programs is an open problem ripe for further research.

Chapter 9. Conclusions and future work

In concluding a thesis, it is appropriate to examine to what extent the initial *aims* were met (sections 9.1 and 9.2). Section 9.3 discusses some further issues that arose in the course of this work. Section 9.4 looks ahead to future work. Section 9.5 summarises the overall importance of the current work in the wider context of computer science and the computing industry.

9.1. A functional language for embedded systems

The first aim identified in Chapter 1 was to extend the effective applicability of functional languages to encompass embedded systems. There are several aspects of functional languages and their implementations which had to be addressed in meeting this goal, and the chosen base language system, Gofer, has correspondingly been modified in several ways.

Embedded computing systems are by definition part of larger machines whose purpose is not computing, so an embedded system needs a specialised computer-machine interface. Input and output between the computing system and the larger machine is different for every new application, not just in the values being transferred, but at the very detailed structural level of *how* the values are transferred. This leads to a division of activities within the software: *computational* activity, and *control* activity. Functional languages already provide excellent facilities for expressing computational activity. What they have lacked is effective and expressive facilities for control.

9.1.1. I/O control

The first control requirement of embedded systems is the ability to describe in detail *how* I/O is performed. Embedded Gofer, in contrast to most functional languages, allows direct access to I/O device registers. This gives the program exactly the detailed device control needed by embedded systems. However, the program need not become swamped by this detail, for two reasons.

1. The functional approach is extremely well suited to abstraction. Common patterns of computation or control can be defined as higher-order functions.
2. The provision of processes in Embedded Gofer allows a convenient separation of independent sorts of I/O activity. Each abstract device can be implemented by one driver process that handles the physical devices but displays a value-level interface to the computational part of the program.

9.1.2. Interrupts

The second control requirement of embedded systems is that they are *asynchronous*. Each of a collection of different I/O devices can represent a different, essentially independent, thread of control and/or computation. Their asynchronous nature is expressed by interrupts, by which the computing system is informed of events in the wider machine. In Embedded Gofer we have chosen communicating sequential processes to model threaded behaviour, and modelled interrupts as input data. However, interrupt-handling processes are written to be explicitly input-data-driven, although the underlying lazy evaluation mechanism is output-demand-driven. It is precisely the dependence of communicating processes on incoming data in messages which permits interrupts to be expressed as input data without suffering the loss of their immediacy.

9.1.3. Memory management

A third issue when targetting a functional language at embedded systems is memory management. Embedded systems are typically uniprocessor, small-memory machines, whereas functional languages are famed for their extravagant memory usage; even those memory management schemes which claim real-time suitability either use much memory or an auxiliary processor. A new garbage collection algorithm, Stack-Safety, has been added to Embedded Gofer to manage heap memory *without* incurring further high memory overheads; it also avoids any need for auxiliary hardware support. Stack-Safety has predictable timing properties – both the moment of invocation, and the amount of time it takes per invocation, are known†. In addition, its incremental nature allows the overall size of the heap to be small from the outset.

9.1.4. Distinctives

Embedded Gofer differs from previous functional languages for embedded systems in several respects. Unlike Erlang [Armstrong+93], it has a strong polymorphic type system, it can handle interrupts, and most importantly, it allows device drivers to be written within the functional framework. Embedded Gofer's process model improves upon other models by being both type secure (unlike e.g. [Stoye86]) and referentially transparent (unlike e.g. [JonesHudak93]). Previous functional treatments of interrupts have either followed a narrow termination semantics (e.g. [GordonHammond94]), or else have been largely undeveloped (e.g. [Clack89]), whereas Embedded Gofer's treatment is both more realistic and has been tested in practice. Finally, Embedded Gofer has been designed explicitly for use on machines with small memory, a stated aim which is absent from every other language design referred to here.

---

† At present it may be difficult to predict the pattern of invocation, closely tied to the pattern of memory allocation. With a different scheduling policy (see §7.6), the moment of invocation would indeed be accurately known.

9.2. Evaluating the effectiveness of Embedded Gofer

The second aim at the outset of this thesis was to demonstrate that functional programming is a *help* to the embedded systems programmer. There are several quantitative, though crude, metrics by which one could attempt to gauge this claim. For instance, a comparison of source code size of the same application in different languages gives a rough estimate of the relative programming effort required. Comparing the average size of function definitions between languages is also a quick measure of source code readability. By either of these measures (see Table 9.1), Embedded Gofer comes out well, although the advantage is slighter than some functional language advocates might hope. One reason why the advantage is not greater is that device register definitions and initialisations essentially take the same amount of code regardless of language.

Application/Language	Embedded Gofer	Modula	Assembler	C
marble-sorter (total lines)	112	n/a	120	n/a
marble-sorter (av. function size)	3.6	n/a	5.3	n/a
liftshaft (total lines)	770	1082	n/a	2235
liftshaft (av. function size)	10.5	16.6	n/a	15.3

Table 9.1: Application code-size comparisons.

However, the real test is whether the "new kinds of glue" [Hughes89] that functional languages provide have any use in programming embedded systems. Higher-order functions and lazy evaluation are the two main techniques identified by Hughes as the important distinguishing features of a purely functional approach to programming.

9.2.1. Higher-order functions

In the two case studies of Chapters 4 and 6, there are several examples of higher-order functions which model particular patterns of *control*, such as while-loops and loops-with-state, quite aside from their use in modelling patterns of *computation*. The functional approach has provided an ability to "extend the language" by writing new control structures. Against this, it could be argued that the new monadic structures simply re-construct what imperative languages give "for free"; moreover, the ability to add *new* control structures *ad hoc* is positively dangerous, because their non-standard behaviour can confuse other programmers. In reply, there is no doubt that control structures should be defined sparingly. However it does seem that there may be new insights to be gained from developing control structures in the monadic framework. For instance, the monadic `foldm` has no analogue in the imperative world, yet was found useful (§6.7).

On a more general question of whether functional abstractions (higher-order or not) are useful in embedded systems programming, the answer is clearly positive. The liftshaft

case study required some subtle device programming due to the constraints imposed by pragmatic hardware interfaces that did not match the ideal abstract behaviour of the available devices. The fact is that, in many embedded systems, the unit-cost issue forces compromises in hardware design that must be rectified in software. The abstraction mechanisms of the functional language were effective in hiding the low-level situation from the controlling processes—it was straightforward to present a value-level interface disguising whether the underlying mechanism was polling or interrupts, and also hiding the fact that some data registers were shared and some shadowed.

9.2.2. Lazy evaluation

Lazy evaluation, on the other hand, might actually obstruct the programming of embedded systems, due to the loss of predictability of evaluation times. Even in an embedded system which cannot be classed as hard real-time (such as the liftshaft case study), it is important that large lazy closures do not build up, only to be reduced at some critical moment inside an interrupt-handler. In Chapter 6, care was taken in the coding of certain drivers, especially the alarm and shared-register servers, to ensure that parts of the local state were evaluated at the end of each loop. Also, in the lift driver processes, some important values were memoised, such as the boolean indicating whether the lift should stop at the approaching floor. Not only was it memoised, but it was specifically evaluated well before it was needed (one floor's-worth of loop iterations beforehand!).

This thesis has looked almost exclusively at mechanisms for I/O. Sequence and control are the essence of I/O, and so laziness is undeniably a difficulty here. It is however true that the *computational* side of any embedded system is left free to use laziness to its advantage. It is possible that, in this sphere, its use is still appropriate. Lazy evaluation allows a general problem-solving technique in which the universal set of possible "answers" is defined as a huge (but mostly un-evaluated) ordered data structure, and solving the problem becomes a simple selection out of this universal set. The selected answer would no doubt be transmitted as a message, perhaps to a critical interrupt driver, but by enforcing evaluation of messages before they are sent, the work would occur at the correct priority level rather than in the interrupt driver. Unfortunately, neither case study presented here uses the technique, and the question remains as to whether embedded systems more generally contain a sufficient computational element for this aspect of laziness to be beneficial.

Dannenberg, referring to the real-time language Arctic [Dannenberg90a], concludes that

> "The [functional] approach seems particularly well-suited to systems that synthesise rather complex output. These are applications where there is a need to assemble and schedule complex behaviours. In contrast, systems that test for conditions and respond immediately, or that perform fixed arithmetic operations on a stream of samples, are easily implemented in almost any language."

It would be fair to say that both case studies in this thesis fall into Dannenberg's second class of applications, as their computational behaviour is not overly complex. In the musical systems studied by Dannenberg, both laziness and higher-order abstractions were found useful [Dannenberg+91].

9.2.3. Type system

To Hughes's two new kinds of glue, we might add strong polymorphic type systems as a third valuable feature of purely functional languages. Although such type systems could in principle be used in non-declarative languages, the lack of side-effects in pure functional programs means that type annotations are a very rich specification of what program components do. In addition, type inference provides a heightened level of security, both syntactic and semantic, around program text, allowing fewer errors to go undetected before run-time. In Embedded Gofer, types are used not only as computational specifications, but also to create layers of I/O privileges for processes, and to ensure the correct routing of messages between processes.

9.3. Further issues

One of the first things to become clear during the case studies was that stream-based I/O is far from adequate for any task which needs frequent or complicated input and output. Continuations were found to be much more manageable, although the choice of operator used to build combined actions can be confusing at times. Monadic I/O [PeytonJonesWadler93] seems to subsume all the benefits of continuation I/O and, in addition, to clear up the confusion over combinators. Monads do have a drawback however: using more than one in the same program (for instance, an array monad together with the I/O monad) is not easy [JonesDuponcheel93]. Our experience here with I/O models mirrors the development of Haskell's I/O system from version 1.1 through to the current proposals for version 1.3 [GordonHammond94], across roughly the same timescale.

Recent developments in type system technology were found particularly useful in adding type security to processes and message-passing. Both type classes [WadlerBlott88] and constructor classes [Jones93] provide rich overloading mechanisms which increase the ability to detect programming errors at compile-time, without significantly increasing source code size.

It is possible that the functional programming "mind-set" can lead to better solutions to imperative control problems than the more obvious codings. In the liftshaft case study, for instance, the two lift cars are naturally programmed as two processes. It seems obvious that some information about which car intends to service which floor should be shared between them. However, sharing state is more difficult in a functional setting than in an imperative one: examining the precise need for this state led to a new insight into how lifts

might be expected to operate. The shared state between the lifts was kept to an absolute minimum, being in fact simply a record of outstanding shaft requests. As a result, code size was kept small, and performance sufficiently fast.

The generally loop-based nature of embedded processes became clear through programming the two case studies. In the introduction (§1.2) it was mentioned that a functional language may need to constrain all recursion to be *tail recursion*. All the looping combinators we have used are indeed tail recursive, so although the language has not enforced the constraint, the applications have. The question of whether to formally require such a constraint remains open however.

9.4. Future work

The current work has demonstrated the viability of several techniques by which a functional language can be extended for programming embedded applications. However, Embedded Gofer is essentially a collection of modifications to a system intended for teaching and experimentation, not for serious applications development. The same principles could be applied to the development of a production-quality language implementation, for use in serious embedded applications.

Future work to improve the *efficiency* of Embedded Gofer could include:

1. The current implementation of the incremental Stack-Safety garbage collection algorithm has a high time cost. Several optimisation strategies could be applied to it to reduce this overhead. Changing the scheduling policy to make it a periodic process (rather than running it at every memory allocation) would be one such improvement. Analysing every instance of `rplaca` and `rplacd` in the run-time system to determine whether a `gcPush` is necessary there would be another.
2. Under monadic I/O, the loop combinators which maintain a local state could be implemented efficiently to use destructive updates on the state values, saving both space and time.

Examples of more speculative future work are as follows:

1. Timing expression, at present specific to any particular program, could become part of the language, perhaps implemented through the monadic combinators. The language's type system, compiler, and related analysis tools, would decide before run-time whether the program as a whole could be scheduled feasibly, and devise that schedule at compile-time. The run-time system would contain code for the timer device and a simple process despatcher which follows the static schedule.
2. Formal reasoning about functional programs is relatively straightforward, but it is still not clear how formal reasoning about programs expressed as communicating functional processes should proceed. Fijma and Udink pointed out the same conclusion in their case study [FijmaUdink91]. There are certainly a number of process

calculi, such as CSP [Hoare85] and CCS [Milner89], but their use for communicating functional processes is an area open to greater research. In the future, formalisms based on co-induction and bisimilarity (e.g. [Gordon95]) may have a part to play.

9.5. Summary

Functional language advocates claim that their use can reduce by at least an order of magnitude the work required to implement medium-to-large software systems. Evidence to back up this claim is now appearing in the literature:

- In a recent prototyping exercise, a team of Haskell programmers was in competition with several teams of imperative language programmers. The results of the experiment ably demonstrated superior rapidity and correctness of development [HudakJones95].
- The FLARE project [RuncimanWakeling95] demonstrated similar results in several substantial application areas.
- Users of the language Erlang report a similar payoff in programming large embedded systems [Armstrong+92], despite the language's lack of a type system. Experience of writing one application in Erlang led to the conclusion that

  > "These technologies, when they have matured sufficiently for use in products, will make it possible to move the volume of our work away from implementation problems and, in the future, allow us to concentrate on customer need and the development of new services." [Persson+92]

It is likely that embedded systems will form an ever more important part of the computing industry's range of products in the coming years. Silicon technology has now reached such levels of sophistication and affordability that embedding computing units inside other products is standard practice. The more quickly that software can be written for such products, the better competitive advantage a company will derive, provided that software is correct. The more confidence that can be placed in the correctness of embedded software, the better will end-users of those products be protected from frustration, danger, and perhaps fatality.

This thesis has demonstrated that the significant advantages conferred by developing a system in a strongly-typed functional language—rapidity of development, conciseness, understandability, and correctness—can be made available to a large range of applications that was previously thought the exclusive preserve of imperative "hacking". Embedded Gofer's design offers all the benefits of modern functional languages—a polymorphic type system; higher-order functions; referential transparency; code re-usability; automatic memory allocation and reclamation; pattern-matching equation syntax—and in addition, a convenient and type-secure I/O system, suitable for embedded systems. This can only be an aid in producing more reliable, concise, and well-understood embedded programs.

Appendix A. Addendum to the Gofer manual

This appendix describes Embedded Gofer's additions to basic Gofer, and in particular its process and I/O facilities.

A.1. Processes

Embedded Gofer provides a new top-level program entity – the process. A program consists of multiple processes, running concurrently. All processes are created before any of them runs. Some processes can be used for device-driving, being able to read and write device registers. Some of these device processes can in addition receive interrupts. All processes can communicate with other processes by means of messages. In order to deliver messages, processes must have addresses, known as PIDs. Each program is free to define an appropriate type for PIDs. A type must be defined for each sort of message too, along with the relationship between particular PIDs and the type of messages they should receive.

Device processes are the only means of effecting I/O with the outside world from Embedded Gofer. There is no operating system to provide terminal or file I/O. Section A.2 describes the types of processes and the primitives used to deliver messages between them. Section A.3 describes some of the combinators provided for monadic I/O in addition to the basic processes. Section A.4 describes some bit-level notations and functions that may be useful in programming devices.

A.2. Process types and primitives

Embedded Gofer provides several monadic types corresponding to different sorts of process. The basic type is `Action`, parameterised on the global PID type, the process's incoming message type, and the value to be passed forward following this action.

```
data Action pid msg val = ST (World -> (val,World))
instance Monad (Action pid msg) where
    result v          = ST (\w -> (v,w))
    (ST f) `bind` g = ST (\w -> let (v,w') = f w
                                    (ST h) = g v
                                in h w')
```

Neither messages nor PIDs appear in the *representation* of values of the `Action` type, but the types are nevertheless used by other parts of the system to ensure certain correctness properties. Parameterisation on PIDs is to allow the programmer to define a PID type specific to the application.

The operations basic to every type of process are `send` and `recv`, for the transmission of messages. They are implemented as primitives.

```
class Monad (a p m) => Process a p m where
    send   :: AddressFor x p => x -> a p m ()
    send x = sendmsg (address x) x
    recv   :: a p m m
    recv    = recvmsg

primitive sendmsg "sendmsg" :: (Process a p m, AddressFor x p) =>
                               p -> x -> a p m ()
primitive recvmsg "recvmsg" :: Process a p m =>
                               a p m m
```

The receive operation returns the first message waiting in the process's incoming queue. If no message is available, the process is blocked until a message arrives. The stated context for sending a message, `AddressFor x p`, guarantees that there is an addressing function for this type of message. The addressing function returns a PID which the runtime system uses to the deliver the message correctly.

```
class AddressFor msg pid where
    address :: msg -> pid
```

Both `send` and `recv` have primitive implementations which are polymorphic in the process type. The *program* however must define the `address` function for every message type it uses, that is, it must provide all instances of the `AddressFor` class.

Processes are partitioned into types which can engage in different sorts of action. In addition to `Action`, we have `DevAction` and `IntAction`, whose definitions as monads exactly follow the scheme above.

```
instance Process Action p m
instance Process DevAction p m
instance Process IntAction p m
```

These different types of action are mirrored by subclasses of `Process` which define device operations and interrupt-handlers, in two layers of privilege. A device-handling process has the operations `getReg`, which returns the word stored in the given device register, and `putReg`, which writes the given word to the given device register. An interrupt-handling process has these two operations and in addition the branching operation `select`, which awaits both interrupts and incoming messages – whichever arrives first decides the sense of the branch, but interrupts are preferred. The default method `getInterrupt` is for the special case when no ordinary messages are expected.

```
class Process a p m => DevProc a p m where
    getReg :: RegAddr -> a p m Word
    putReg :: RegAddr -> Word -> a p m ()

class DevProc a p m => IntProc a p m where
    select :: (Interrupt->a p m val) ->
                        (m->a p m val) ->
                            a p m val
    getInterrupt :: a p m ()
    getInterrupt = select (\Interrupt -> result ())
                          (\_ -> getInterrupt)
```

The process types which fall under these classes are pre-defined as instances of them. The instances associate *primitive* operations with the overloaded operation names.

A static process table is defined for a program in `main`, of type `Main`. It is simply a list of process definitions. A process definition is an abstract type† with three primitive constructor functions, one for each specialised class of process. Each construction associates a process identifier with a process (i.e. a monadic `Action`). Interrupt handlers are additionally associated with an interrupt vector address.

```
type ProcDef pid
data Main pid = Define [ProcDef pid]

defOrdProc :: Process a p m =>  p -> a p m () -> ProcDef p
defDevProc :: DevProc a p m =>  p -> a p m () -> ProcDef p
defIntProc :: IntProc a p m =>
                        Vector -> p -> a p m () -> ProcDef p
```

A.3. The process prelude

All the above definitions are provided to the Embedded Gofer programmer as a prelude file called `proc.prelude`. This prelude is additional to the Gofer standard prelude (`cc.prelude`, because constructor classes are used), rather than replacing it. The process prelude contains other useful monadic functions:

The monadic combinator, `bind`, is given the synonym `(?)`, and a specialisation, `(>>)` defined as follows:

```
x >> y = x 'bind' \_ -> y
```

Some basic monadic loop constructs are provided, shown in Figure A.1.

† This treatment is different from the definition in Chapter 3, where `Main` was not abstract, and had visible data constructors rather than constructor functions. The change is necessary because each constructor now has a class context, which cannot currently be expressed on data constructors.

```
loop :: Monad m => m () -> m ()
loop body = body >> loop body

loopwith :: Monad m => s -> (s->m s) -> m ()
loopwith initstt body = body initstt ? \newstt ->
                        loopwith newstt body

forall :: Monad m => [a] -> (a->m ()) -> m ()
forall list body =
    foldr (>>) (result ()) (map body list)

foldm :: Monad m => (a->b->m a) -> a -> [b] -> m a
foldm _    accum []      = result accum
foldm body accum (x:xs) =
    body accum x ? \newacc -> foldm body newacc xs
```

Figure A.1: *proc.prelude*: **control structures.**

Embedded Gofer's run-time system is implemented for the Gofer compiler only – it does not make sense to use device register operations in *interpreted* Gofer on a workstation. Both the run-time system and the C program generated by the Gofer compiler must be cross-compiled for the target embedded architecture, and the resulting executable then loaded to the target machine.

A.4. Hexadecimal constants and bit-wise operations

Hexadecimal notation is accepted practice, and often convenient, for specifying machine-word constants and machine addresses. It is supported by Embedded Gofer. The character sequence *0x* introduces a hexadecimal constant, which is a sequence of any of the characters *0-9*, *a-f*, or *A-F*.

Embedded Gofer supports bit-wise operations on machine-words, because twiddling bits is very common as a part of controlling embedded devices. Another additional prelude file (*bit.prelude*) gives access to the bit-wise operations. Three operations, *andw*, *orw*, and *notw* are implemented by new primitives; the rest are defined in the prelude.

```
byte :: Int -> Word                      -- type cast
byte = chr

primitive andw "primAndWord" :: Word -> Word -> Word
primitive orw  "primOrWord"  :: Word -> Word -> Word
primitive notw "primNotWord" :: Word -> Word

bit, notbit :: Int -> Word
bit n     | n==0      = byte 0x01
          | n==1      = byte 0x02
          | n==2      = byte 0x04
          | n==3      = byte 0x08
          | n==4      = byte 0x10
          | n==5      = byte 0x20
          | n==6      = byte 0x40
          | n==7      = byte 0x80
          | otherwise = byte 0
notbit n = notw (bit n)

setbit, clrbit :: Int -> Word -> Word
setbit n w = orw  (bit n) w
clrbit n w = andw (notbit n) w

bitset, bitclr :: Int -> Word -> Bool
bitset n w = andw (bit n) w /= (byte 0)
bitclr n w = andw (bit n) w == (byte 0)

maskset, maskclr :: Word -> Word -> Bool
maskset x y = andw x y /= (byte 0)
maskclr x y = andw x y == (byte 0)
```

Figure A.2: *bit.prelude.*

Appendix B. The monadic marble sorter

This appendix contains a monadic version of the marblesorter case study described in Chapter 4†. This version employs the message type-checking scheme of Chapter 5, extended slightly as described in Appendix A.

```
data Pid = MetalDetect
         | LightDetect
         | Alarm
         | HopperRelease
         | SorterArm

type Time     = Int
data AlarmMsg = At Time (IntAction Pid AlarmMsg ())
instance AddressFor AlarmMsg Pid where address = const Alarm

alarm :: IntAction Pid AlarmMsg ()
alarm =
    deviceInit >>
    loopwith [] (\q ->          -- q :: [AlarmMsg]
        select (\Interrupt -> actOn (dec q))
               (\(At t a)  -> result (insert t a q)))
    where dec    []          = []
          dec (At t a: q)    = At (t-1) a: q
          actOn (At 0 a: q)  = a >> actOn q
          actOn  q           = result q
          insert t a []      = At t a: []
          insert 0 a q       = At 0 a: q
          insert t a (At t' a': q)
                | t'>t       = At t  a : At (t'-t) a': q
                | t>=t'      = At t' a': insert (t-t') a q

defer :: (Process a p m, AddressFor AlarmMsg p) =>
        Time -> IntAction Pid AlarmMsg () -> a p m ()
defer t a = send (At t a)
```

Figure B.1: Monadic marble-sorting.

† Constant definitions for device addresses and initialisation sections are excluded for clarity, but can be seen in Appendix C. Also, no provision is made here for terminal interaction—again see Appendix C.

```
data HopperMsg = TimesUp
instance AddressFor HopperMsg Pid where address = const HopperRelease

wait :: AddressFor AlarmMsg Pid => Time -> DevAction Pid HopperMsg ()
wait t =
    defer t (send TimesUp) >>  recv  ? \TimesUp ->  result ()

releaseRate    = 20
minCollectTime = 4

hopper :: DevAction Pid HopperMsg ()
hopper =
    loop (putReg padr close   >>
          wait minCollectTime >>
          putReg padr open    >>
          wait (releaseRate - minCollectTime))


instance AddressFor () Pid where address = const MetalDetect

metalDetect, lightDetect :: IntAction Pid () ()
metalDetect = loop getInterrupt

metalMask    = byte 0x10
dismissMandL = byte 0x05
dismissJustL = byte 0x01
travelTime   = 5

lightDetect =
    loop (getInterrupt >>
          getReg psr   ? \statusVal ->
          if (statusVal 'andw' metalMask) /= byte 0  then
              putReg psr dismissMandL >>
              defer travelTime (send MetalMarbleComing)
          else putReg psr dismissJustL >>
              defer travelTime (send GlassMarbleComing))


data SorterMsg = MetalMarbleComing
               | GlassMarbleComing
instance AddressFor SorterMsg Pid where address = const SorterArm

sorter :: DevAction Pid SorterMsg ()
sorter =
    loop (recv ? \msg ->
          case msg of
             MetalMarbleComing -> putReg pbdr left
             GlassMarbleComing -> putReg pbdr right)


main :: Main Pid
main = Define [defIntProc 0x44 Alarm         alarm,
               defIntProc 0x42 LightDetect   lightDetect,
               defIntProc 0x40 MetalDetect   metalDetect,
               defDevProc      HopperRelease hopper,
               defDevProc      SorterArm     sorter]
```

Figure B.2: Monadic marble-sorting (continued).

Appendix C. Detailed device-driving code: the register model

This appendix illustrates the manner in which devices are programmed at the most detailed level. Definitions of device addresses, status, control, and data values are shown, together with device initialisation code and some driver processes. This material collects together device code common to the marblesorter and liftshaft case studies, but excluded from Appendix B and Chapter 6 respectively.

The demonstration single-board computer has two main types of interface devices: the Parallel Interface Timer (PIT) and the Dual Asynchronous Receiver/Transmitter (DUART). The PIT is a parallel device and is covered in section C.1, whilst the DUART is a serial device and is covered in section C.2: there are two of each device on the board.

C.1. PIT 68230

C.1.1. *Data port initialisation*

The two PITs [Motorola83] are memory-mapped 8-bit devices, and are aligned at the high and low bytes of addresses from *0x900000* upwards. In the marblesorter case study only one of the PITs is used, and in the liftshaft this same one PIT is used primarily. So in order to address this PIT as concisely as possible, we assume that all references to "the" PIT refer to the high-byte PIT. An explicit modifier (*pit1* or *pit2*) can be used to make a reference more precise.

```
pit :: Addr
pit = 0x900001

pit1, pit2 :: Addr -> Addr
pit1 = id
pit2 = ((-)1)
```

Each register is offset from the PIT base address, as defined in the device manual [Motorola83] (Figure C.1).

```
pgcr    = pit + 0           -- port general control reg
psrr    = pit + 2           -- port service request reg
paddr   = pit + 4           -- port A data direction reg
pbddr   = pit + 6           -- port B data direction reg
pcddr   = pit + 8           -- port C data direction reg
pivr    = pit + 10          -- port interrupt vector reg
pacr    = pit + 12          -- port A control reg
pbcr    = pit + 14          -- port B control reg
padr    = pit + 16          -- port A data reg
pbdr    = pit + 18          -- port B data reg
paar    = pit + 20          -- port A address reg
pbar    = pit + 22          -- port B address reg
pcdr    = pit + 24          -- port C data reg
psr     = pit + 26          -- port status reg
```

Figure C.1: PIT: register addresses.

There are three parallel ports on the PIT: we do not use port C, but ports A and B can be configured for independent bit I/O, either unidirectional or bidirectional. In addition to eight data pins, ports A and B each have two handshake pins that can be configured to generate interrupts on certain conditions. The monadic code in Figure C.2 initialises the PIT register contents for the purposes of the marblesorter.

```
pitInit :: DevAction p m ()
pitInit =
    putReg pgcr  (byte 0x3a) >>    -- unidir, 8-bit, intrpts-enabled
    putReg psrr  (byte 0x18) >>    -- vectored intrpts, prio H 1,2,3,4 L
    putReg paddr (byte 0xff) >>    -- data dir: all pins out
    putReg pbddr (byte 0xff) >>    -- data dir: all pins out
    putReg pivr  (byte 0x44) >>    -- store interrupt vector
    putReg pacr  (byte 0x82) >>    -- submode 1X, H2/input/dis, H1/en
    putReg pbcr  (byte 0x82) >>    -- submode 1X, H4/input/dis, H3/en
    putReg padr  (byte 0x02) >>    -- hopper open
    putReg pbdr  (byte 0x01)       -- sorter swept left
```

Figure C.2: PIT: initialisation.

This initialisation code, while adequate, is far from descriptive. It need not be. Figure C.3 shows an alternative description of the same initialisation action. Each action is a composition of bit-wise updates.

This coding is rather less efficient than the previous one, but then efficiency may not be a general concern for initialisation code. Almost by definition, it will only be executed once, before anything else; performance is unlikely to be critical at this stage in the embedded application.

Figure C.3 is somewhat more descriptive of the semantics of the intended operations than Figure C.2. However, there is very little security associated with the updating functions. For instance, the function *unidir*, intended to be used only with the general control register, could be inadvertently used on the interrupt vector register. This would make no logical sense, but is the sort of error one would hope that the language system might detect automatically. As currently coded, such error detection is not possible. For that, a model of registers would be needed that used *types* to identify sets of operations that could be performed on each register. This proposal is inevitably much more complicated than we have scope to discuss here. It does seem that it would form a good topic for future research however.

On the whole, the gain of using descriptive initialisation is so insubstantial that we revert to the simple hexadecimal notation in what follows.

```
-- pgcr modes
unidir     = clrbit 7
bidir      = setbit 7
eightbit   = clrbit 6
sixteenbit = setbit 6
H34disable = clrbit 5
H34enable  = setbit 5
H12disable = clrbit 4
H12enable  = setbit 4
H4senselow = clrbit 3
H4sensehi  = setbit 3
H3senselow = clrbit 2
H3sensehi  = setbit 2
H2senselow = clrbit 1
H2sensehi  = setbit 1
H1senselow = clrbit 0
H1sensehi  = setbit 0

-- psrr modes
noDMA    = clrbit 6
dmaH1    = setbit 6 . clrbit 5
dmaH3    = setbit 6 . setbit 5
noints   = clrbit 4 . clrbit 3
autovect = clrbit 4 . setbit 3
pc5piack = setbit 4 . clrbit 3
vectored = setbit 4 . setbit 3
prio 1 2 3 4 = clrbit 2 . clrbit 1 . clrbit 0
prio 2 1 3 4 = clrbit 2 . clrbit 1 . setbit 0
prio 1 2 4 3 = clrbit 2 . setbit 1 . clrbit 0
prio 2 1 4 3 = clrbit 2 . setbit 1 . setbit 0
prio 3 4 1 2 = setbit 2 . clrbit 1 . clrbit 0
prio 3 4 2 1 = setbit 2 . clrbit 1 . setbit 0
prio 4 3 1 2 = setbit 2 . setbit 1 . clrbit 0
prio 4 3 2 1 = setbit 2 . setbit 1 . setbit 0

-- etc. for all registers

fillreg a f = putReg a (f (byte 0))

pitInit =
    fillreg pgcr  (unidir . eightbit . H34enable . H12 enable .
                   H4sensehi . H3sensehi . H2sensehi . H1sensehi) >>
    fillreg psrr  (noDMA . vectored . prio 1 2 3 4) >>
    fillreg paddr (compose setbit [0..7]) >>
    fillreg pbddr (compose setbit [0..7]) >>
    fillreg pivr  (const (byte intloc)) >>
    fillreg pacr  (bitIO . H2input . H2disable . H1enable) >>
    fillreg pbcr  (bitIO . H4input . H4disable . H3enable) >>
    fillreg padr  open >>
    fillreg pbdr  left
```

Figure C.3: PIT: descriptive initialisation.

C.1.2. *Timer initialisation*

The timer part of the PIT device has the following registers (Figure C.4).

```
tcr      = pit + 32              -- timer control reg
tivr     = pit + 34              -- timer interrupt vector reg
cprh     = pit + 38              -- timer counter pre-load reg high
cprm     = pit + 40              -- timer counter pre-load reg mid
cprl     = pit + 42              -- timer counter pre-load reg low
tsr      = pit + 52              -- timer status reg
```

Figure C.4: PIT: timer register addresses.

The following control values are written to the control register to enable and disable the timer, and to set its interrupt mode as a periodic interrupt. In this mode, the internal counter registers count downwards with the system clock strobe. When the internal count is zero the timer interrupts. The counter is automatically re-initialised to the pre-loaded 24-bit value and continues decrementing.

```
tstart = byte 0xa1
tstop  = byte 0xa0
```

The action of Figure C.5 describes how to initialise the timer to interrupt at precisely 5Hz. The internal clock resolution is 4 microseconds per decrement, so a counter preload value of `0x00c350` (decimal 50,000) is given.

```
timerIntLoc = 0x48

timerInit =
    putReg tcr   tstop >>                -- Disable timer.
    putReg tivr  (byte timerIntLoc) >>   -- Set interrupt vector.
    putReg cprh  (byte 0x00) >>          -- Counter preload is
    putReg cprm  (byte 0xc3) >>          --   set to 0.2s
    putReg cprl  (byte 0x50) >>          --   per interrupt.
    putReg tcr   tstart >>               -- Set intrpt mode/enable.
    putReg tsr   (byte 1)                -- Reset count to zero.

alarmServer = timerInit >> alarm
```

Figure C.5: PIT: timer initialisation.

C.2. DUART 68681

C.2.1. *Initialisation*

The demonstration board has two DUARTs [Motorola85]: one for controlling the terminal lines, and one linked to the in-car buttons on the lifts. Neither device is primary, so the approach taken with the PITs is not appropriate here. Rather, the DUARTs are distinguished by their function:

```
data DuartType = Term | DDev

base68681, duart_int_vector :: DuartType -> Addr
base68681 Term = 0x800000
base68681 DDev = 0xa00000

duart_int_vector Term = 0x44
duart_int_vector DDev = 0x48
```

Figure C.6: DUART: two base addresses.

Registers are again at offsets from the device base address. There are two channels on each DUART, conventionally labelled *A* and *B*. See Figure C.7.

```
offset :: Int -> DuartType -> AB -> Addr
offset n t A = (base68681 t) + n
offset n t B = (base68681 t) + 0x10 + n

duart_mode, duart_stat, duart_clk_sel, duart_cmnd, duart_tx_buf,
duart_rx_buf :: DuartType -> AB -> Addr

duart_mode    = offset 1
    -- mode registers channel A/B (read/write)
duart_stat    = offset 3
    -- status register channel A/B (read only)
duart_clk_sel = offset 3
    -- clock select register channel A/B (write only)

duart_cmnd    = offset 5
    -- command register channel A/B (write only)
duart_tx_buf  = offset 7
    -- transmit buffer channel A/B (write)
duart_rx_buf  = offset 7
    -- receive buffer channel A/B (read)

duart_isr t = (base68681 t) + 11
    -- interrupt status register (read)
duart_imr t = (base68681 t) + 11
    -- interrupt mask register (write)
duart_ivr t = (base68681 t) + 25
    -- interrupt vector register (read/write)
```

Figure C.7: DUART: register addresses.

Some deviousness is needed when initialising the *mode* register for a channel, because there are actually *two* *mode* registers for each channel. One of them is only accessible following a particular command to the *cmnd* register. After this *mode* register has been written to, the address reverts to the usual *mode* register.

```
program_mr1 :: DuartType -> AB -> Word -> DevAction p m ()
program_mr1 t ab value = putReg (duart_cmnd t ab) (byte 0x1a) >>
                         putReg (duart_mode t ab) value
```

The other registers have various meanings, defined in the device manual [Motorola85] (Figure C.8):

```
reset_rx t ab = putReg (duart_cmnd t ab) (byte 0x2a)
reset_tx t ab = putReg (duart_cmnd t ab) (byte 0x3a)
reset_errs  t = putReg (duart_cmnd t A) (byte 0x4a) >>
                putReg (duart_cmnd t B) (byte 0x4a)
reset_ints  t = putReg (duart_cmnd t A) (byte 0x5a) >>
                putReg (duart_cmnd t B) (byte 0x5a)
stop_break  t = putReg (duart_cmnd t A) (byte 0x7a) >>
                putReg (duart_cmnd t B) (byte 0x7a)

enable_rx t ab = putReg (duart_cmnd t ab) (byte 0x01)
enable_tx t ab = putReg (duart_cmnd t ab) (byte 0x04)
enable_xx t ab = putReg (duart_cmnd t ab) (byte 0x05)

disable_rx t ab = putReg (duart_cmnd t ab) (byte 0x02)
disable_tx t ab = putReg (duart_cmnd t ab) (byte 0x08)
disable_xx t ab = putReg (duart_cmnd t ab) (byte 0x00)

set_rx_ints t = putReg (duart_imr t) (byte 0x22)

duart_reset :: DuartType -> DevAction p m ()
duart_reset t =
    reset_rx t A >> reset_rx t B >>
    reset_tx t A >> reset_tx t B >>
    reset_errs t >> reset_ints t >>
    stop_break
```

Figure C.8: DUART: programming information.

Finally, each DUART can be initialised, with slightly different baud rates and interrupt settings (Figure C.9):

```
duart_init :: DuartType -> DevAction p m ()
duart_init dt =
    duart_reset dt >>
    program_mr1 dt A (byte 0x13) >>   -- irq on rxrdy, 8bits/char
    program_mr1 dt B (byte 0x13) >>
    putReg (duart_mode dt A) (byte 0x07) >> -- normal mode
    putReg (duart_mode dt B) (byte 0x07) >> -- 1 stop bit
    putReg (duart_clk_sel dt A) (baud_rate dt) >>
    putReg (duart_clk_sel dt B) (baud_rate dt) >>
    enable_rx dt A >> enable_tx dt A >>
    enable_rx dt B >> enable_tx dt B >>
    putReg (duart_ivr dt) (byte (duart_int_vector dt)) >>
    case dt of
        DDev -> result ()
        Term -> set_rx_ints Term
    where baud_rate DDev = byte 0xee
          baud_rate Term = byte 0x99
```

Figure C.9: DUART: register initialisation.

C.2.2. *Serial line use*

It is relatively straightforward to use the DUART channels for communication: the operations `putChar` and `getChar` are monadic actions that can be used inside any device process (Figure C.10).

```
putChar :: DuartType -> AB -> Char -> DevAction p m ()
getChar :: DuartType -> AB -> DevAction p m Char

putChar t ab c =
    until (getReg (duart_stat t ab) ? \val ->
           result (bitset tx_rdy val)) >>
    putReg (duart_tx_buf t ab) c
    where tx_rdy = 2

getChar t ab =
    until (getReg (duart_stat t ab) ? \val ->
           result (bitset rx_rdy val)) >>
    getReg (duart_rx_buf t ab)
    where rx_rdy = 0
```

Figure C.10: Operations `putchar` **and** `getchar`**.**

Appendix D. Shared register server process

Figure D.2 contains the full coding of the shared register server outlined in section 6.5 The server's local state keeps a shadow of each of four registers: two on a PIT, two on a DUART. Figure D.1 defines some auxiliary functions which sugar access to these four shadowed registers.

```
data Dev    = Pit | Duart
data Regs   = Regs Word Word Word Word

addr   :: Dev -> AB -> Addr
read   :: Dev -> AB -> Regs -> Word
update :: Dev -> AB -> Word -> Regs -> Regs

addr Pit   A  = pit1 padr
addr Pit   B  = pit1 pbdr
addr Duart ab = duart_rx_buf DDev ab

read Pit   A (Regs x _ _ _) = x
read Pit   B (Regs _ x _ _) = x
read Duart A (Regs _ _ x _) = x
read Duart B (Regs _ _ _ x) = x

update Pit   A w (Regs _ x y z) = (Regs w x y z)
update Pit   B x (Regs w _ y z) = (Regs w x y z)
update Duart A y (Regs w x _ z) = (Regs w x y z)
update Duart B z (Regs w x y _) = (Regs w x y z)
```

Figure D.1: Shadowed register access.

When the lamps and motors are updated, note that we have to store the change in the PIT shadow registers as well, but when an LED display or bleep is updated, we don't store the change in the DUART shadow registers, because these are shadows of the *received* values, not of the *transmitted* values. Also, on every clock tick, although we read the values of all four registers, we only update the corresponding write-registers on the second PIT, not on the DUART. This is because the PIT controls lamps which keep a visible record of the shadowed state of the shaft buttons, but there is no corresponding visible record kept of the state of the in-car buttons.

The reason we test the new values of the registers against the old values is twofold: if there has been no change, then we can save the expense of two updates; but more importantly, the equality check is actually a strictifying operation, ensuring that large closures do not build up in the local state, only to be evaluated when an update is actually required. This is an inelegant means of achieving strictness, and deserves further thought.

```
data ShRegT = UpdReg Dev AB (Word->Word)
            | GetReg Dev AB

sharedreg =
    let pinit = byte 0
        dinit = byte 0xff in
    putReg (pit2 padr) pinit >>
    putReg (pit2 pbdr) pinit >>
    loopwith (Regs pinit pinit dinit dinit) (\regs ->
        recv ? \msg ->
        case msg of
            (UpdReg Pit ab f) ->
                    let newval = f (read Pit ab regs) in
                    putReg (addr Pit ab) newval >>
                    result (update Pit ab newval regs)
            (UpdReg Duart ab f) ->
                    let newval = f (read Duart ab regs) in
                    putReg (addr Duart ab) newval >>
                    result regs
            (GetReg dev ab) ->
                    send (RegVal (read dev ab regs)) >>
                    result regs
            ClockTick ->
                    getReg (addr Pit A) ? \pa ->
                    getReg (addr Pit B) ? \pb ->
                    getReg (addr Duart A) ? \da ->
                    getReg (rddr Duart B) ? \db ->
                    let fa = orw (notw pa) (read Pit A regs)
                        fb = orw (notw pb) (read Pit B regs)
                        ga = orw (notw da) (read Duart A regs)
                        gb = orw (notw db) (read Duart B regs)
                    in if fa==pa && fb==pb
                       && ga==da && gb==db then
                            result regs
                       else putReg (pit2 padr) fa >>
                            putReg (pit2 pbdr) fb >>
                            result (Regs fa fb ga gb))
```

Figure D.2: Shared register server.

References

[Abdullahi+92] **Collection schemes for distributed garbage**, Saleh E Abdullahi, Eliot E Miranda, Graem A Ringwood, pp.43-81, in *Proceedings International Workshop on Memory Management*, St. Malo, France, Springer-Verlag, LNCS **637** (September 1992)

[Appel+88] **Real-time concurrent garbage collection on stock multiprocessors**, Andrew A Appel, John R Ellis, Kai Li, pp.11-20, in *Proceedings ACM Sigplan Symposium on Programming Language Design and Implementation*, Atlanta, Georgia (June 1988)

[Armstrong+92] **Implementing a functional language for highly parallel real time applications**, J L Armstrong, B O Dacker, S R Virding, M C Williams, in *Proceedings of SETSS 92*, Florence (April 1992)

[Armstrong+93] **Concurrent programming in Erlang**, Joe Armstrong, Robert Virding, Mike Williams, Prentice-Hall (1993)

[AshcroftWadge77] **LUCID, A Nonprocedural Language with Iteration**, E A Ashcroft, W W Wadge, Communications of the ACM **20***(7)*, pp.519- (July 1977)

[Augustsson93] **Haskell B user's manual**, Lennart Augustsson, Chalmers University, Sweden (October 1993)

[Baker78] **List processing in real time on a serial computer**, Henry G Baker, Communications of the ACM **21***(4)*, pp.280-294 (April 1978)

[Baker92] **The treadmill: real-time garbage collection without motion sickness**, Henry G Baker, ACM SIGPLAN Notices **27***(3)*, pp.66-70 (March 1992)

[Ben-Ari84] **Algorithms for on-the-fly garbage collection**, Mordechai Ben-Ari, ACM Transactions on Programming Languages and Systems **6***(3)*, pp.333-344 (July 1984)

[Benveniste+91] **Synchronous programming with events and relations: the SIGNAL language and its semantics**, Albert Benveniste, Paul Le Guernic, Christian Jacquemot, Science of Computer Programming **16**, pp.103-149, Elsevier (1991)

[BoyleHarmer92] **A practical functional program for the CRAY X-MP**, James M Boyle, Terence J Harmer, Journal of Functional Programming **2**_(1)_, pp.81-126 (January 1992)

[BretzEbert88] **An exception handling construct for functional languages**, M Bretz, J Ebert, in _Proceedings ESOP88, 2nd European Symposium on Programming_, Nancy, France, Springer-Verlag, LNCS **300** (March 1988)

[BrinchHansen78] **Distributed processes: a concurrent programming concept**, Per Brinch Hansen, Communications of the ACM **21**_(11)_, pp.934-941 (November 1978)

[Brooks84] **Trading data space for reduced time and code space in real-time garbage collection on stock hardware**, Rodney A Brooks, pp.256-262, in _Proceedings ACM Symposium on Lisp and Functional Programming_, Austin, Texas (August 1984)

[Broy83] **Applicative Real Time Programming**, M Broy, IFIP Congress, Paris, pp.259-264 (1983)

[Burn87] **Evaluation transformers: a model for the parallel evaluation of functional languages (extended abstract)**, G L Burn, pp.446-470, in _Functional Programming Languages and Computer Architecture_, Springer-Verlag, LNCS **274** (September 1987)

[BurnsWellings89] **Real-time systems and their programming languages**, A Burns, A J Wellings, Addison Wesley (1989)

[Burns91] **Scheduling hard real-time systems: a review**, Alan Burns, Software Engineering Journal **6**_(3)_, pp.116-128 (May 1991)

[Burton88] **Non-determinism with Referential Transparency in Functional Programming Languages**, F W Burton, Computer Journal **31**_(3)_, pp.243-247 (1988)

[CarlssonHallgren93] **"Fudgets - A Graphical User Interface in a Lazy Functional Language**, M Carlsson, T Hallgren, pp.321-330, in _Proceedings of Functional Programming Languages and Computer Architecture_, Copenhagen, ACM Press (June 1993)

[Caspi+87] **LUSTRE: A declarative language for programming synchronous systems**, P Caspi, D Pilaud, N Halbwachs, J A Plaice, pp.178-188, in _Proceedings of 14th ACM Symposium on Principles of Programming Languages_ (January 1987)

[CaspiHalbwachs86] **A functional model for describing and reasoning about time behaviour of computing systems**, Paul Caspi, Nicolas Halbwachs, Acta Informatica **22**, pp.595-627 (1986)

[Church41] **The calculi of lambda conversion**, A Church, Princeton University Press, Princeton, New Jersey (1941)

[ClackPeytonJones85] **Generating parallelism from strictness analysis**, Chris Clack, Simon Peyton Jones, pp.92-131, in *Proceedings of the Workshop on Implementation of Functional Languages*, Chalmers University of Technology, Goteborg, Sweden, **PMG 17** (February 1985)

[Clack89] **Signal handling for functional programs**, Chris Clack, University College London (May 1989)

[Cupitt89] **A Brief Walk Through KAOS**, J Cupitt, Computing Laboratory, University of Kent at Canterbury, **TR 58** (Feb 89)

[Cupitt90] **The design and implementation of an operating system in a functional language**, John Cupitt, PhD Thesis, Computing Laboratory, University of Kent at Canterbury (October 1990)

[Dannenberg84] **Arctic: A Functional Language for Real-Time Control**, Roger B Dannenberg, in *ACM Symposium on LISP and Functional Programming*, Austin, Texas, pp.96-103 (1984)

[Dannenberg+86] **Arctic: a functional language for real-time systems**, Roger B Dannenberg, Paul McAvinney, Dean Rubine, Computer Music Journal **10***(4)*, pp.67-77 (Winter 1986)

[DannenbergFraley89] **Fugue: composition and sound synthesis with lazy evaluation and behavioural abstraction**, Roger B Dannenberg, Christopher Lee Fraley, pp.76-79, in *Proceedings International Computer Music Conference*, San Francisco (1989)

[Dannenberg89] **The Canon Score Language**, Roger B Dannenberg, Computer Music Journal **13***(1)*, pp.47-56 (Spring 1989)

[Dannenberg90a] **Expressing temporal behaviour declaratively**, Roger B Dannenberg, **Proceedings Twenty-Fifth Anniversary Symposium**, Dept. of Computer Science, Carnegie Mellon (1990)

[Dannenberg90b] **A run-time system for Arctic**, Roger B Dannenberg, pp.185-187, Proceedings International Computer Music Conference, Glasgow (1990)

[Dannenberg+91] **Fugue: a functional language for sound synthesis**, Roger B Dannenberg, Christopher Lee Fraley, Peter Velikonja, IEEE Computer **24***(7)*, pp.36-42 (July 1991)

[Dijkstra76] **A discipline of programming**, E W Dijkstra, Prentice-Hall (1976)

[DukeSmith89] **Temporal logic and Z specifications**, R Duke, G Smith, Australian Computer Journal **21***(2)*, pp.62-66 (May 1989)

[FaustiniLewis86] **Towards a Real-Time Dataflow Language**, Antony A Faustini, Edgar B Lewis, IEEE Software **3***(1)*, pp.29-35 (January 1986)

[FijmaUdink91] **A case study in functional real-time programming**, Duco H Fijma, R T Udink, Department of Computer Science, University of Twente, Enschede, the Netherlands, Memoranda Informatica, **91-62** (August 1991)

[Freeman+82] **A Model Lift Shaft as a Test-Bed for Student's Real-Time Programming Experiments**, W Freeman, M Baslington, R Pack, Department of Computer Science, University of York, **YCS 52** (May 1982)

[FriedmanWise80] **An Indeterminate Constructor for Applicative Programming**, D P Friedman, D S Wise, ACM Principles of Programming Languages **7***(1)*, pp.245-250 (January 1980)

[Gautier+87] **SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems**, T Gautier, P Le Guernic, L Besnard, pp.257-277, in *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS **274** (September 1987)

[Gordon92] **Functional Programming and Input/Output**, Andrew D Gordon, DPhil Thesis, Computing Laboratory, University of Cambridge (August 1992)

[GordonHammond94] **A proposal for monadic I/O in Haskell 1.3**, Andrew D Gordon, Kevin Hammond, Cambridge University Computer Laboratory (March 1994)

[Gordon95] **A tutorial on co-induction and functional programming**, Andrew D Gordon, in *Proceedings of Glasgow Workshop on Functional Programming*, Springer/BCS Workshops in Computer Science (September 1994)

[Halbwachs+91] **The synchronous dataflow programming language LUSTRE**, N Halbwachs, P Caspi, P Raymond, D Pilaud, Proceedings of the IEEE **79***(9)*, pp.1305-1320 (September 1991)

[Harper+86] **Standard ML**, R Harper, D MacQueen, R Milner, Laboratory for Foundations of Computer Science, University of Edinburgh, **LFCS-86-2** (March 1986)

[Harrison87] **Ruth: a functional language for real-time programming**, D A Harrison, in *Proceedings Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, Netherlands, Springer-Verlag, LNCS **259** (June 1987)

[Harrison88] **Functional Real-Time Programming: The Language Ruth and its Semantics**, D A Harrison, PhD Thesis, University of Stirling, Dept of Computer Science Tech Reports, **TR 59** (September 1988)

[Harrison91] **A real-time language based on explicit timeouts**, D A Harrison, in *1st IFAC Workshop on Algorithms and Architectures for Real-Time Control*, Bangor, N Wales., pp.Pergamon (1991)

[HarrisonNelson89] **A real-time language based on explicit timeouts**, Dave Harrison, Sam Nelson, University of Stirling, Dept of Computer Science Tech Reports, **TR 54** (1989)

[Henderson82] **Purely Functional Operating Systems**, P Henderson, in *Functional Programming and its Applications*, edited by J Darlington, P Henderson, and D A Turner, Cambridge University Press (1982)

[Hoare85] **Communicating sequential processes**, C A R Hoare, Prentice-Hall (1985)

[HolyerCarter93] **Concurrency in a purely declarative style**, Ian Holyer, David Carter, Department of Computer Science, University of Bristol (June 1993)

[HudakSundaresh88] **On The Expressiveness of Purely Functional I/O Systems**, Paul Hudak, Raman S Sundaresh, Yale University, Department of Computer Science, **YALEU/DCS/RR-665** (December 1988)

[Hudak89] **Conception, evolution, and application of functional programming languages**, Paul Hudak, ACM Computing Surveys **21***(3)*, pp.359-411 (Sept 1989)

[Hudak+92] **Report on the Programming Language Haskell Version 1.2**, Paul Hudak, Simon L Peyton Jones, Phil Wadler, ACM SIGPLAN Notices **27***(5)* (May 1992)

[HudakFasel92] **A gentle introduction to Haskell**, Paul Hudak, Joseph H Fasel, ACM SIGPLAN Notices **27***(5)* (May 1992)

[Hudak+94] **Haskore music notation: an algebra of music**, Paul Hudak, Tom Makucevich, Syam Gadde, Bo Whong, Yale University Department of Computer Science (October 1994)

[HudakJones95] **Haskell vs. Ada vs. C++ vs. Awk vs. ..., An experiment in software prototyping productivity**, Paul Hudak, Mark P Jones, Journal of Functional Programming **5***(to appear)* (1995)

[HughesODonnell89] **Expressing and Reasoning About Non-deterministic Functional Programs**, J Hughes, J O'Donnell, in *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, Springer-Verlag (1989)

[Hughes94] **Type inference and real-time programming**, John Hughes, in *Glasgow Functional Programming Workshop*, (talk only) (September 1994)

[Hughes89] **Why functional programming matters**, R J M Hughes, BCS Computer Journal Special Issue on Lazy Functional Programming **32***(2)*, pp.98-107 (April 1989)

[Ichbiah+83] **Reference Manual for the Ada Programming Language**, J D Ichbiah et al., United States Department of Defense, **ANSI/MIL-STD-1815A-1983** (February 1983)

[INMOS84] **occam Programming Manual**, INMOS Ltd., Prentice Hall (1984)

[Jensen94] **Clock Analysis of synchronous dataflow programs**, Jensen, INRIA Rennes (unpublished work)

[Johnsson83] **The G-Machine: An Abstract Machine for Graph Reduction**, T Johnsson, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology, pp.1-19, in *Proceedings of the Declarative Programming Workshop*, University College, London (April 1983)

[Jones92] **Computing with lattices: an application of type classes**, Mark P Jones, Journal of Functional Programming **2***(4)*, pp.475-503 (October 1992)

[JonesHudak93] **Implicit and explicit parallel programming in Haskell**, Mark P Jones, Paul Hudak, Department of Computer Science, Yale University, **YALEU/DCS/RR-982** (August 1993)

[JonesDuponcheel93] **Composing Monads**, Mark P Jones, Luc Duponcheel, Yale University Research Report, **YALEU/DCS/RR-1004** (December 1993)

[Jones93] **A system of constructor classes: overloading and implicit higher-order polymorphism**, Mark P Jones, pp.52-61, in *Proceedings of 6th Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, ACM Press (June 1993)

[Jones94] **The implementation of the Gofer functional programming system**, Mark P Jones, Department of Computer Science, Yale University, Technical Report, **YALEU/DCS/RR-1030** (May 1994)

[JonesSinclair89] **Functional Programming and Operating Systems**, S B Jones, A F Sinclair, BCS Computer Journal Special Issue on Lazy Functional Programming **32***(2)*, pp.162-174 (April 1989)

[Jones84] **A range of operating systems written in a purely functional style**, S B Jones, University of Stirling, Dept. of Computer Science Tech Reports, **TR 16** (Sept 84)

[Karlsson81] **Nebula: A Functional Operating System**, K Karlsson, Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteburg, **LPM11** (1981)

[KennawaySleep82] **Expressions as Processes**, J R Kennaway, M R Sleep, pp.21-28, in *ACM Symposium on LISP and Functional Programming*, Pittsburgh, Pennsylvania (August 1982)

[Landin65] **A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II**, P J Landin, Communications of the ACM **8***(2&3)*, pp.89-101,158-165 (February & March 1965)

[LeGuernic+86] **SIGNAL - a data-flow oriented language for signal processing**, Paul Le Guernic, Albert Benveniste, Patricia Bournai, Thierry Gautier, IEEE Transactions on Acoustics, Speech, and Signal Processing **34***(2)*, pp.362-374 (April 1986)

[LeGuernicGautier91] **Dataflow to von Neumann: the SIGNAL approach**, Paul Le Guernic, Thierry Gautier, in *Advanced topics in dataflow computing*, edited by J-L Gaudiot and L Bic, Prentice Hall (1991)

[LeGuernic+91] **Programming real-time applications with SIGNAL**, Paul Le Guernic, Michel Le Borgne, Thierry Gautier, Claude Le Maire, Proceedings of the IEEE **79***(9)* (September 1991)

[Lehoczky+88] **Aperiodic task scheduling for hard real-time systems**, J Lehoczky, B Sprunt, L Sha, in *Proceedings IEEE Real-Time Systems Symposium* (1988)

[LiebermanHewitt83] **A real-time garbage collector based on the lifetimes of objects**, Henry Lieberman, Carl Hewitt, Communications of the ACM **26***(6)*, pp.419-429 (June 1983)

[LiuLayland73] **Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment**, C L Liu, J W Layland, JACM **20***(1)*, pp.46-61 (January 1973)

[MacQueen+84] **An ideal model for recursive polymorphic types**, D B MacQueen, G Plotkin, R Sethi, pp.165-174, in *Proceedings of 11th Symposium on Principles of Programming Languages* (January 1984)

[McCarthy63] **A Basis for a Mathematical Theory of Computation**, J McCarthy, pp.33-70, in *Computer Programming and Formal Systems*, edited by Braffort and Hirschberg, North Holland (1963)

[McCarthy60] **Recursive functions of symbolic expressions and their computation by machine.**, John McCarthy, Communications of the ACM **3***(4)*, pp.184-195 (April 1960)

[McCarthy+62] **LISP 1.5 Programmers Manual**, John McCarthy, Paul W Abrahams, Daniel J Edwards, Timothy P Hart, Michael I Levin, MIT Press, Cambridge, Mass. (1962)

[Milner89] **Communication and concurrency**, Robin Milner, International Series in Computer Science, Prentice-Hall (1989)

[Moggi89] **Computational lambda calculus and monads**, Eugenio Moggi, in *Proceedings of the 4th IEEE Symposium on Logic in Computer Science* (June 1989)

[MollerTofts89] **A temporal calculus of communicating systems**, Faron Moller, Christopher M N Tofts, University of Edinburgh, **ECS-LFCS-89-104** (December 1989)

[Motorola83] **MC68230 Parallel Interface/Timer (PI/T)**, Motorola Inc. (December 1983)

[Motorola85] **MC68681 Dual Asynchronous Receiver/Transmitter (DUART)**, Motorola Inc. (September 1985)

[Okasaki95] **Simple and efficient purely functional queues and deques**, Chris Okasaki, Journal of Functional Programming **5***(to appear)* (1995)

[PanzieriDavoli93] **Real-time systems: a tutorial**, F Panzieri, R Davoli, Laboratory for Computer Science, University of Bologna, Tech Report, **UBLCS-93-22** (October 1993)

[ParkShaw90] **Experiments with a program timing tool based on source-level timing schema**, C Y Park, A C Shaw, pp.72-81, in *Proceedings Real-Time Systems Symposium*, IEEE Computer Society Press (December 1990)

[Perry91] **The implementation of practical functional programming languages**, Nigel Perry, Imperial College of Science & Technology, Dept of Computing, PhD Thesis (second edition) (June 1991)

[Persson+92] **A switching software architecture prototype using real time declarative language**, M Persson, K Odling, D Eriksson, in *Proceedings of International Switching Symposium*, Yokohama (October 1992)

[PeytonJones89] **Parallel implementations of functional programming languages**, Simon L Peyton Jones, The Computer Journal **32***(2)*, pp.175-186, Cambridge University Press (April 1989)

[PeytonJonesWadler93] **Imperative functional programming**, Simon Peyton-Jones, Phil Wadler, in *Proceedings of 20th Symposium on Principles of Programming Languages*, Charleston, South Carolina, ACM Press (January 1993)

[Queinnec+89] **Mark DURING sweep rather than mark THEN sweep**, Christian Queinnec, Barbara Beaudoing, Jean-Pierre Queille, pp.224-237, in *Proceedings PARLE '89*, Springer-Verlag, LNCS **365** (1989)

[Rabhi93a] **Parallel programming paradigms: a functional programming approach,** Fethi A Rabhi, Internal Report, Dept of Computer Science, University of Hull (1993)

[Rabhi93b] **Exploiting parallelism in functional languages: a paradigm-oriented approach,** Fethi A Rabhi, in *Proceedings of Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds (April 1993)

[Reeves+89] **GERALD: an exceptional lazy functional programming language,** A C Reeves, D A Harrison, A F Sinclair, P Williamson, pp.371-390, in *Glasgow Functional Programming Workshop*, Springer-Verlag, BCS Workshops in Computing (1989)

[Reppy91] **CML: A higher-order concurrent language,** John H Reppy, pp.293-305, in *SIGPLAN Conference on Programming Language Design and Implementation* (June 1991)

[Rojemo94] **NHC: Nearly a Haskell Compiler,** Niklas Rojemo, Dept. of Computer Science, Chalmers University, unpublished typescript (1994)

[RubineDannenberg87] **ARCTIC Programmer's Manual and Tutorial,** D Rubine, R Dannenberg, Department of Computer Science, Carnegie-Mellon University, **CMU-CS-87-110** (Jun 87)

[Runciman81] **Modula and a vision laboratory,** Colin Runciman, International Journal of Man-Machine Studies **14**, pp.371-386 (1981)

[Runciman91] **TIP in Haskell: another exercise in functional programming,** Colin Runciman, pp.278-292, in *Fourth Annual Glasgow Workshop on Functional Programming*, Springer-Verlag, Workshops in Computing (August 1991)

[RuncimanWakeling93] **Heap profiling of lazy functional programs,** Colin Runciman, David Wakeling, Journal of Functional Programming **3***(2)*, pp.217-245 (April 1993)

[RuncimanWakeling95] **Applications of Functional Programming,** edited by Colin Runciman, edited by David Wakeling, UCL Press (1995)

[Sansom94] **Execution profiling for non-strict functional languages,** Patrick M Sansom, PhD Thesis*(FP-1994-09)*, Department of Computer Science, University of Glasgow (Sept 1994)

[Schneider90] **Correctness and communication in real-time systems**, S Schneider, DPhil Thesis, Oxford University, **PRG-84** (1990)

[ScholefieldZedan92] **TAM: a formal framework for the development of distributed real-time systems**, David J Scholefield, Hussein S M Zedan, in *Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, Nijmegen, Netherlands (January 1992)

[ScholefieldZedan93] **A standard for finite TAM**, David J Scholefield, Hussein S M Zedan, Department of Computer Science, University of York, **YCS 206** (August 1993)

[Shaw89] **Reasoning about time in higher level language software**, A C Shaw, IEEE Transactions on Software Engineering **15***(7)*, pp.875-889 (July 1989)

[SkillicornGlasgow89] **Real-time specification using Lucid**, D B Skillicorn, J I Glasgow, IEEE Transactions on Software Engineering **15***(2)*, pp.221-229 (Feb. 1989)

[Smetsers+91] **Generating Efficient Code For Lazy Functional Languages**, S. Smetsers, E. Nocker, J. van Groningen, R. Plasmeijer, pp.592-617, in *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS **523** (August 1991)

[SondergaardSestoft88a] **Nondeterminism in Functional Languages**, H Sondergaard, P Sestoft, Department of Computer Science, University of Melbourne, **88/18** (1988)

[SondergaardSestoft88b] **Referential transparency and allied notions**, H Sondergaard, P Sestoft, DIKU Report, **88/7**, DIKU, University of Copenhagen, Denmark (1988)

[Spivey90] **A Functional Theory of Exceptions**, M Spivey, Science of Computer Programming **14**, pp.25-42 (June 1990)

[Stoye86] **Message-based functional operating systems**, William Stoye, Science of Computer Programming **6***(3)*, pp.291-311 (May 1986)

[Swarup+91] **Assignments for Applicative Languages**, V Swarup, U S Reddy, E Ireland, pp.192-214, in *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS **523** (August 1991)

[Thompson87] **Interactive functional programs**, Simon Thompson, Computing Laboratory, University of Kent, **TR 48** (November 1987)

[Turner81] **The Semantic Elegance of Applicative Languages**, David A Turner, in *Proceedings of Functional Programming Languages and Computer Architecture* (1981)

[Turner87] **Functional Programming and Communicating Processes**, David A Turner, pp.54-74, in *PARLE, Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, Springer-Verlag, LNCS **259** (June 1987)

[WadgeAshcroft85] **Lucid, the Dataflow Programming Language**, W W Wadge, E A Ashcroft, APIC Studies in Data Processing Series, Academic Press (1985)

[WadlerBlott88] **How to make ad-hoc polymorphism less ad-hoc**, P Wadler, S Blott, pp.33-52, in *Proceedings of the workshop on Implementation of Lazy Functional Languages*, Programming Methodology Group, Chalmers University, Sweden, **Report 53** (September 1988)

[Wadler85] **How to replace failure by a list of successes**, Phil Wadler, pp.113-128, in *Proceedings of Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS **201** (September 1985)

[Wadler90] **Comprehending Monads**, Philip Wadler, pp.61-78, in *ACM Conference on LISP and Functional Programming*, Nice, France (1990)

[WalinskyBanerjee90] **A Functional Programming Language Compiler for Massively Parallel Computers**, Clifford Walinsky, Deb Banerjee, pp.131-138, in *ACM Conference on LISP and Functional Programming*, Nice, France (1990)

[WallaceRunciman93] **An incremental garbage collector for embedded real-time systems**, Malcolm Wallace, Colin Runciman, in *Proceedings of the Winter Meeting*, Chalmers University of Technology, Gothenburg, Sweden, **PMG-R73** (June 1993)

[WallaceRunciman94] **Type-checked message passing between functional processes**, Malcolm Wallace, Colin Runciman, in *Proceedings of the Glasgow Functional Programming Workshop*, Springer Verlag, BCS Workshops in Computer Science (Sept 1994)

[WallaceRunciman95] **Extending a functional programming system for embedded applications**, Malcolm Wallace, Colin Runciman, Software Practice and Experience **25***(1)* (January 1995)

[Wilson92] **Uniprocessor garbage collection techniques**, Paul R Wilson, pp.1-42, in *Proceedings International Workshop on Memory Management*, St. Malo, France, Springer-Verlag, LNCS **637** (September 1992)

[Wirth77] **Design and Implementation of Modula**, Niklaus Wirth, Software Practice and Experience **7***(1)*, pp.67-84 (January 1977)

[Yuasa90] **Real-time garbage collection on general-purpose machines**, Taiichi Yuasa, Journal of Systems and Software **11**, pp.181-198 (1990)