Programming in Temporal Logic

Roger William Stephen Hale

Trinity College

A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge

October 1988

To Melanie and John

Abstract

The idea of writing computer programs in logic is an attractive one, for such programs may be designed, verified and implemented using a single formal language. This brings a number of practical benefits:

1. There is no room for ambiguity in the relationship between specification and implementation, and no need to learn a different language for each.

2. It is easy to test out specifications from the earliest stages of development, which avoids attempting to implement or verify an inappropriate design.

3. Computerised tools can be applied directly to transform and verify programs, using the established machinery of mathematical logic.

4. Logic supports hierarchical design, so a large project can be divided into smaller tasks which may be designed and verified independently.

Similar benefits are bestowed by any formal programming language, but the idea only works if the language suits the intended application. All too often the application is forced to fit the language.

In this dissertation, I describe an approach that suits the development of parallel and real-time systems. The approach is based on Tempura, a deterministic programming language developed by Moszkowski from his work on hardware specification using Interval Temporal Logic (ITL). I present the formal semantics of ITL in higher-order logic, and show how programs can be transformed and verified using the HOL theorem prover. Then, I show how to represent a number of familiar programming concepts in ITL. First, I show that the language of while-programs can be embedded in temporal logic; and that includes the destructive assignment statement with the usual inertial assumption. An interesting corollary is that a simple sequential program, written in Pascal say, becomes a logic program in Tempura. More advanced concepts include parallel processes and message passing, as well as real-time phenomena such as timeouts, interrupts and traps. Each idea is experimentally tested on a suitable example, using an interpreter for Tempura. The examples range from matrix multiplication and parallel sorting, to a pipelined parser and a real-time lift-controller.

Acknowledgements

This dissertation has been a long time in the making. Throughout that time I have benefitted greatly from the support and advice of Ben Moszkowski and Mike Gordon. It is obviously true to say that I would not have begun this research were it not for Ben's initial work on executing temporal logic programs; but his commitment and single-mindedness have been a great stimulus throughout. I am deeply indebted to Mike Gordon for providing a stable and secure working environment, and for nudging me in the right direction from time to time when I most needed it.

Thanks are also due to colleagues who have been around at various times to ask questions or answer mine. I would particularly like to thank Miriam Leeser for making me think about the frame assumption, Albert Camilleri, Inder Dhingra, John Herbert, Jeff Joyce and Tom Melham for help at various times with theorems and tactics and other HOL things, and Randall Lichota for helping to improve the Tempura interpreter.

This dissertation is the result of my own work and, unless otherwise stated in the text, includes nothing which is the outcome of work done in collaboration. No part of this dissertation has already been, or is currently being, submitted for any degree, diploma or other qualification at any other university.

Contents

1 Prologue **1**
1.1 The Problem . . . 1
1.1.1 Program Design . . . 1
1.1.2 Correctness . . . 2
1.2 Programming in Temporal Logic . . . 3
1.2.1 Tempura . . . 3
1.2.2 Advantages . . . 5
1.3 Other Approaches . . . 5
1.3.1 Imperative Languages . . . 6
1.3.2 Array Processing . . . 7
1.3.3 Functional Languages . . . 8
1.3.4 Dataflow Programming . . . 10
1.3.5 Logic Programming . . . 11
1.3.6 Tokio . . . 12
1.3.7 Communicating Sequential Processes . . . 13
1.3.8 Synchronous Languages . . . 14
1.3.9 Restricted Instruction Set Languages . . . 15
1.4 My Contribution . . . 16

2 A Practical Introduction **18**
2.1 The Interpreter . . . 19
2.2 Predicates, Functions and Macros . . . 20
2.3 Programs . . . 22
2.4 Operators . . . 23
2.4.1 Empty . . . 24
2.4.2 Next . . . 24
2.4.3 Computation Length . . . 25
2.4.4 Always . . . 25
2.4.5 Assignment . . . 26
2.4.6 Sequential Behaviour . . . 27
2.4.7 Unit-Assignment and Initialisation . . . 29
2.4.8 Multiple Assignments . . . 29

2.4.9 Iteration 29
2.4.10 Gets 30
2.4.11 Extended and Prefix Computations 32
2.4.12 Other Operators 34
2.5 A Complete Example 34

3 Interval Temporal Logic **41**
3.1 Syntax 42
3.1.1 Expressions 42
3.1.2 Formulae 43
3.2 Semantics 44
3.2.1 HOL 44
3.2.2 Intervals 45
3.2.3 Expressions 46
3.2.4 Formulae 47
3.3 Some Derived Operators 50
3.3.1 Classical Operators 50
3.3.2 Temporal Operators 51
3.3.3 Assignment Operators 53
3.3.4 Iterative Operators 56
3.3.5 Markers 57
3.3.6 Omitting Parentheses 57
3.4 Discussion 58

4 Tempura **60**
4.1 Syntax 61
4.1.1 Programs 61
4.1.2 Expressions 62
4.2 Semantics 62
4.2.1 Canonical Form 63
4.2.2 Reduction of Programs 63
4.2.3 Local Variables 66
4.2.4 Final Transformation 67
4.2.5 Predicates 68
4.3 Derived Operators 68
4.3.1 Classical Operators 68
4.3.2 Temporal Operators 69
4.3.3 Assignment Operators 69
4.3.4 Iterative Operators 69
4.3.5 Input and Output 70
4.4 Discussion 70

5 Verification and Transformation **72**
5.1 Verification 73
5.1.1 Natural Deduction 73
5.1.2 Properties of Programs 75
5.1.3 Mathematical Induction 76
5.1.4 Proof Rules 77
5.1.5 Hierarchical Decomposition 78
5.2 Transformation 79
5.2.1 Transformation Rules 79
5.2.2 Canonical Form 80
5.2.3 Functional Equivalence 83
5.2.4 Efficiency 85
5.3 Discussion 86
5.3.1 Satisfaction Guaranteed? 86
5.3.2 Mechanical Verification 87
5.3.3 Transformation and Synthesis 88

6 Sequential Programs **89**
6.1 Assignment 90
6.1.1 Semantics 90
6.1.2 Notes on Assignment 94
6.2 Inertia 96
6.2.1 Frame Variables 97
6.2.2 Notes on Frame Variables 101
6.2.3 The Operator `local` 104
6.2.4 Doing Without Frame Variables 104
6.3 Discussion 105
6.3.1 Default Values 106
6.3.2 The Frame Problem in Artificial Intelligence 106

7 Recursion and Iteration **108**
7.1 The Towers of Hanoi 109
7.1.1 Recursive Algorithm 110
7.1.2 Transformation 111
7.1.3 Iterative Algorithm 113
7.2 Parallel Summation 114
7.2.1 Recursive Algorithm 114
7.2.2 Iterative Algorithm 116
7.2.3 General Algorithm 116
7.3 Mergesort 118
7.3.1 Recursive Mergesort 119

7.3.2 The Merge Algorithm 120
7.3.3 Iterative Mergesort Algorithm 122
7.4 Discussion 123

8 Parallel Processing **125**
8.1 Processor Arrays 126
8.1.1 Interconnections 126
8.1.2 Summation 128
8.1.3 Mergesort 130
8.2 Parallel Processes 132
8.2.1 The Parallel Composition Operator 134
8.2.2 Matrix Multiplication 134
8.3 Discussion 137

9 Real-Time Systems **140**
9.1 Additional Operators 141
9.1.1 Projection 141
9.1.2 Interrupts 142
9.1.3 Bar 143
9.1.4 Traps 144
9.1.5 Time Limits 144
9.2 A Lift Control System 146
9.2.1 The Interface 146
9.2.2 The Specification 151
9.2.3 The Lift Controller 158
9.2.4 Some Improvements 165
9.3 Discussion 167

10 Communicating Processes **170**
10.1 Message Passing 171
10.1.1 Transferring a List 171
10.1.2 Termination 172
10.1.3 The Operations Put and Get 173
10.2 The Sieve of Eratosthenes 174
10.2.1 Specification 175
10.2.2 Implementation 175
10.2.3 The Filter Processes 176
10.3 A Simple Parser 178
10.3.1 The Parsing Algorithm 179
10.3.2 Error Handling 180
10.4 The Complete Evaluator 181
10.4.1 The Lexical Analyser 181

10.4.2 The Evaluator 183
10.5 Interleaving 184
10.5.1 Simple Timeslicing 185
10.5.2 Rendezvous 185
10.6 Discussion 186

11 Epilogue **187**
11.1 Scaling Up 187
11.1.1 Compilation 187
11.1.2 Data Types 188
11.1.3 Debugging 188
11.2 Applications 188
11.2.1 Real-Time Systems 188
11.2.2 Parallel Programming 189
11.2.3 Rapid Prototyping 189
11.3 Verification and Transformation 190
11.3.1 Automation 190
11.3.2 Transformation 190
11.4 Semantics 190
11.4.1 Frame 191
11.4.2 Prefix 191
11.4.3 Parallel Processes 191

Chapter 1

Prologue

This dissertation describes a rigorous approach to the design of computer programs, particularly those for parallel and real-time applications. The approach is based on Tempura, a programming language developed by Moszkowski from his work on hardware specification using Interval Temporal Logic [Mos86, Mos85]. Experiments with an interpreter and verification system for Tempura show that this is a realistic way to produce correct and efficient programs.

1.1 The Problem

Advances in computer technology have consistently led to smaller and faster circuitry, yet the law of diminishing returns applies as much here as anywhere else; at any time the cost of enhanced performance greatly exceeds the corresponding performance gains. This pattern of development has had repercussions at both ends of the computing spectrum. On the one hand, more and more use is being made of embedded control systems now that suitable processors can be built at an acceptable price. On the other hand, it is more cost-effective to increase performance by using many ordinary processors in parallel than by making special high speed sequential machines. Each of these developments brings the need for new ways to design programs and new concerns over their correctness.

1.1.1 Program Design

Unlike the von Neumann machines we are used to, the emerging generation of parallel processors do not share a common style of hardware organisation. They range from massively parallel synchronous machines to small collections of powerful autonomous processors, with many shades in between. There is very little understanding of how to program, or even to think about, these machines in a general way. Each new architecture seems to spawn its own set of algorithms and its own programming discipline.

Meanwhile, at the other end of the computing spectrum, an increasing number of control systems depend on digital computers. In these systems hardware and software must work closely together. Indeed, many traditional software functions are being taken over by hardware as, with the help of VLSI, it is becoming easier and cheaper to implement quite sophisticated algorithms in hardware. For example, an embedded control system might nowadays be implemented as a combination of application-specific and off-the-shelf hardware running real-time software. But the initial design should not reflect this choice, as the exact combination might not be fixed until the later stages of design.

Here, then, is a situation that demands an effective way to design algorithms, one in which it is possible to represent the desired behaviour in a number of ways suitable for different implementation environments. Ideally it should be possible to transform between different representations, or at least describe their relationship.

1.1.2 Correctness

The application of rigorous methods to programming is always helpful, even when it falls short of formal verification. Complex programs, especially parallel and real-time programs, are notoriously difficult to get right, and mistakes made during the early stages of design can be very costly to correct later on. A combination of formal specification and prototyping can spot those early mistakes.

But there are times when formal verification is necessary. More and more safety-critical systems depend on the correct operation of computers. Examples include the real-time controllers to be found in aircraft, cars, medical equipment and chemical plants. These systems have in common that their malfunction could have catastrophic consequences, resulting in enormous expense or perhaps even loss of life, yet they (or at least the crucial parts of them) are simple enough to be amenable to formal analysis with the presently available tools.

Such control systems typically have to perform under quite stringent time constraints; timing behaviour is a part of their correctness. For instance, a flight control system cannot be said to work correctly if it takes several seconds to respond to directions from the pilot. Whatever tools are used for system design must therefore be based on a clearly defined model of time.

1.2 Programming in Temporal Logic

Temporal logic is an extension of classical logic especially designed for representing time-dependent behaviour [Pri67]. It has proved to be an effective tool for specifying and reasoning about parallel systems, both hardware [Boc82, HMM83] and software [Pnu81, Lam83, Kro87]. My thesis is that temporal logic, through the programming language Tempura, is also a realistic way to design correct and efficient

programs.

1.2.1 Tempura

Tempura embodies a synthesis of logic with the imperative style of programming used in ordinary sequential languages like Pascal. It includes many familiar control structures from imperative programming, such as assignment and iteration, as well as less familiar ones like parallel composition and delay. These constructs are not, as is so often the case, a pragmatic collection of useful ideas without proper formal foundation; they are all derived from a handful of primitive operators in Interval Temporal Logic (ITL); a Tempura program is just a deterministic formula in ITL.

A computation in this model is a discrete sequence of states; think of it as a series of "snapshots" taken over an interval of time. Three typical computations are illustrated in figure 1.1. Figure 1.1(a) shows the values of two variables Y and N on an interval $\langle \mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6} \rangle$ during which Y is assigned the value $\mathtt{2}^\mathtt{3}$ by repeated multiplication. Figure 1.1(b) shows successive values of a list variable as it is sorted in parallel (using mergesort). Finally, figure 1.1(c) shows an interval $\langle \mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{21}, \mathtt{22} \rangle$ on which the expression "$(\mathtt{1} + \mathtt{2}_\sqcup)$" is evaluated using a pipelined lexer, parser and evaluator. All of these computations are fully described in due course.

In previous work Moszkowski and I have shown something of the versatility of Tempura [Mos86, Hal87, HM87], but in this dissertation it is extended in a number of ways. Most importantly, it is extended to take inertial behaviour into account, and as a consequence the language of while-programs becomes a part of Tempura. This means that a simple sequential program, written in Pascal say, may also be regarded as a formula of ITL. Furthermore, I show how such programs may be combined in parallel, either at the level of individual statements or at the process level, or combined with real-time constructs, such as traps, timeouts and interrupts.

1.2.2 Advantages

The strength of the approach derives from its ability to handle a wide range of applications in a formal and coherent way, as well as its inclusion of the imperative style, which is still used by the overwhelming majority of computer programs in everyday use. Moreover, since Tempura is derived from a hardware specification language, it is naturally an appropriate language to use right down to the lowest levels of implementation, including the hardware [Mos83, Hal85]. This makes it suitable for discussing the details of implementation that are crucial to the design of efficient programs. The scope of Tempura will be amply demonstrated in future chapters with examples ranging from matrix multiplication and parallel sorting, to a pipelined parser and a real-time lift controller, but other benefits derive from its mathematical origins.

time	0	1	2	3	4	5	6
Y	1	2	2	4	4	8	8
N	3	3	2	2	1	1	0

(a) Calculation of 2^3 by repeated multiplication (time increases from left to right).

time	A
0	[1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0]
1	[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0]
2	[0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1]
3	[0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1]
4	[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
5	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

(b) Sorting a list *in situ* using a parallel mergesort (time increases down the page).

time	Characters	Tokens	Expression
0	–	–	–
1	"("	–	–
2	–	–	–
3	"1"	"("	–
4	–	–	–
5	"+"	–	–
6	–	–	–
7	"2"	1	–
8	"2"	–	–
9	"2"	"+"	1
10	–	–	–
11	"␣"	–	–
12	–	–	–
13	")"	2	–
14	–	–	–
15	#	")"	2
16	–	–	–
17	–	#	"+"
18	–	–	–
19	–	–	#
20	–	–	#
21	–	–	#
22	–	–	–

(c) A pipelined system for tokenising, parsing and evaluating simple arithmetic expressions. The state of the communication streams is shown during evaluation of the expression: "(1 + 2␣)".

Figure 1.1: Three discrete-time computations.

First, because programs and their specifications are written in the same language, the relationship between them is unambiguous. Furthermore, only one language need be used from the initial specification right down to the final implementation. This makes it easier to produce prototype implementations from the earliest stages of design, so reducing the likelihood of errors propagating to the later stages of implementation or verification where they are much more expensive to put right.

Second, because Tempura programs are formulae of temporal logic they may be transformed and verified using the established rules of mathematical logic, and existing computerised theorem provers, such as the HOL system [Gor87], may be used to assist the processes of transformation and verification. Another advantage of the logical approach is that specifications and programs are compositional, so a large project can be hierarchically decomposed into a number of smaller tasks that may be designed and verified independently.

1.3 Other Approaches

My thesis demonstrates that there is no need to throw out the whole imperative style in order to get a language with straightforward semantics, a rich syntax for dealing with parallel and temporal behaviour, and the ability to mix a wide range of programming techniques, uncommitted to any particular architectural paradigm.

There are a number of other approaches to program development for which some of these benefits might be claimed. This section reviews some of them, and in so doing it gives an informal introduction to some of the strengths and weaknesses of Tempura. In the course of this review, several new operators are presented without proper explanation. They will be properly defined later on; the aim here is just to give you a feel for the language and how it compares with other approaches.

1.3.1 Imperative Languages

Conventional imperative languages, such as Fortran and Pascal, are designed around the von Neumann architecture. Their control structures are just abstract representations of what can easily be done with this architecture, so it is not surprising that they can be implemented very efficiently, if only on conventional processors.

Natural formal semantics, such as Hoare's logic [Hoa69], can be constructed for the simpler sequential languages, but they quickly get out of hand as new constructs are added. Time-dependent behaviour is usually expressed with language extensions for which no precise meaning is given, and parallel constructs, when present, are generally best suited to some particular style of concurrency. For instance, there are a number of parallel dialects of Fortran with control structures to exploit the fine-grained concurrency of array processors, but not for concurrency at the process level. Similarly, there are a number of Pascal-like languages, such as Ada, which

include features to assist the construction of co-operating sequential processes, but not for parallelism at the level of individual statements.

In chapter 6 I show that the basic sequential language constructs, including the usual destructive assignment statement, may be represented in Tempura. For example, the following Tempura program uses inertial variables Y and N to calculate the value of x^n by repeated multiplication:

```
Y, N ⇐ 1, n;
while N ≠ 0 do {
  Y := Y × x;
  N := N − 1
}.
```

Figure 1.1(a) shows how this program behaves for particular inputs; it is exactly as you would expect by association with other imperative languages. The *chop* operator (;) denotes sequential composition, and the operators ⇐ and := denote assignment; the former denotes initialisation, the latter unit-assignment.

The only major imperative construct that does not seem to fit easily into Tempura is the *goto* statement.[1] However, all the most common uses of the *goto*, for conditional branches, exception handling and so on, can be represented in Tempura. It is a great advantage of the language that a large body of existing software can be run with only cosmetic changes. But do not be deceived by this similarity; the Tempura program is also a formula in ITL, and may be transformed and verified accordingly.

1.3.2 Array Processing

Scientific programs often perform operations on large arrays, and such calculations are often suitable for implementation on parallel computers. For this reason, a number of parallel dialects of Fortran have been designed to enable whole vectors to be processed in parallel. In Tempura, parallel operations on vectors and arrays may be expressed with the iterated parallel operator **forall**. This is like the sequential for-loop except that the iterations take place in parallel rather than one after another. For example, the assignment $Y \leftarrow X \times A$ of the product of an n-element vector X and an $n \times n$ matrix A to Y may be implemented as follows:

```
forall i < n : Y_i = 0 ∧
for j < n do
  forall i < n : Y_i := Y_i + X_j × A_ji.
```

Other formulations of this program are discussed in chapter 8.

An alternative way to speed up repetitive array processing problems is through the use of systolic or pipelined algorithms. These algorithms have become popular in

[1] The *goto* statement can, however, be represented in the standard way using continuations.

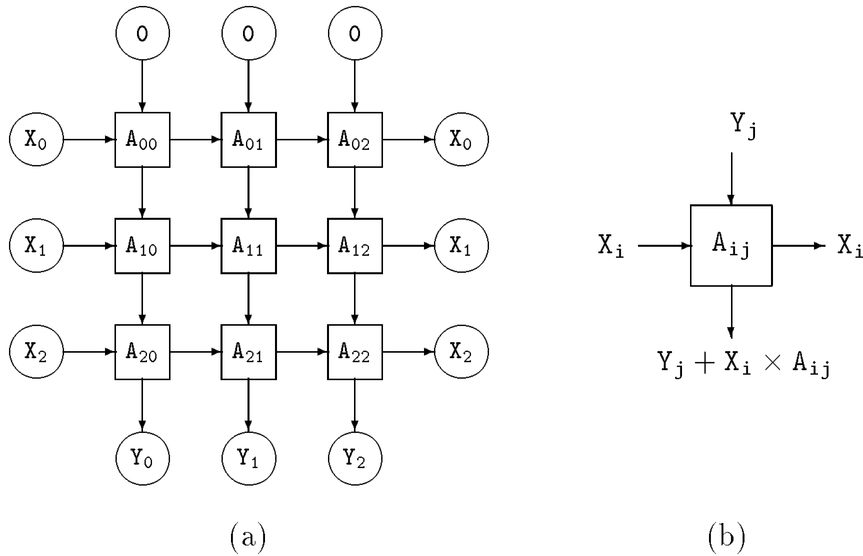


Figure 1.2: (a) A systolic array for multiplying a stream of vectors `X` by a constant 3×3 matrix `A` giving a stream of product vectors `Y`; and (b) a single cell, showing inputs and outputs.

---

recent years because their regularity makes them easy to analyse and to implement, especially as VLSI arrays.

Figure 1.2 represents a systolic array to multiply a stream `X` of vectors by a constant matrix `A`, producing a stream `Y` of product vectors. This kind of array might be used to transform a large number of vector co-ordinates according to a single transformation matrix. Such operations are commonplace in graphical display terminals where transformation speed is of the utmost importance.

The systolic algorithm in figure 1.2 is represented in Tempura as an array of nodes operating in parallel. Every node is connected to its neighbours in each direction (`N`, `S`, `E` and `W`), and has an additional connection `Aij` for loading an element of the matrix `A`. The node at position $(\mathtt{i},\mathtt{j})$ holds $\mathtt{A}_{\mathtt{ij}}$. Streams of elements of `X` pass from `W` to `E` across the array, and streams of elements of the product vectors, `Y`, pass from `N` to `S` accumulating terms as they go. At each step the product of $\mathtt{A}_{\mathtt{ij}}$ with $\mathtt{X}_{\mathtt{i}}$ (from the `W`) is added into $\mathtt{Y}_{\mathtt{j}}$ (from the `N`). The sum is passed on to the `S`, and $\mathtt{X}_{\mathtt{i}}$ is passed on to the `E`.

The nodes are all identical; an individual node may be expressed in Tempura as follows, using the operator **gets**, which denotes a repeated assignment from each state to the next:

$$\mathtt{node(Aij,N,S,E,W)} \stackrel{\text{def}}{=} \mathtt{S}\ \mathbf{gets}\ (\mathtt{N} + \mathtt{W} \times \mathtt{Aij}) \ \wedge\ \mathtt{E}\ \mathbf{gets}\ \mathtt{W}.$$

This algorithm, which is typical of many systolic algorithms, is described in greater

detail in chapter 2.

1.3.3 Functional Languages

Functional languages provide an abstract way to escape dependency on conventional architectures [Bac78]. These languages have simple and elegant semantics; a functional program, written in "pure" Lisp [MAJ62] or ML [Mil84] say, is just a mathematical function whose execution is an application of this function to particular arguments.

Functional languages are basically static, for they exclude the notions of state and change; dynamic variables can only be represented indirectly by means of lists of values or as higher-order functions. Whilst this is mathematically acceptable, it obscures the intended meaning. One can write an expression such as $\mathtt{A} + \mathtt{B}$ in Tempura to represent the changing sum of two values, but in a functional language one is forcibly reminded that `A` and `B` are actually lists or functions.

Functional languages are certainly not committed to any particular architecture because they do not contain any mechanisms to govern the flow of control. They may take advantage of parallel execution, but can equally well be implemented sequentially. However, this generality means that functional programs do not always make the best use of the special features of conventional processors, and there are particular inefficiencies associated with maintaining large data structures by purely functional means. The naive strategy for updating large data structures entails copying the whole structure, with the consequent garbage collection overhead; more sophisticated strategies introduce a higher access overhead. These problems have led to the introduction of imperative constructs into functional languages, examples being *prog*, *setq* and *replaca* in Lisp.

Recursive functions can be defined in Tempura, but they are evaluated on a single state and are to be viewed as static expressions, not dynamic programs. For example, the following ITL specification asserts that the list `A` ends up with a sorted version of its initial value,

$$\mathtt{A} \leftarrow \mathtt{sort(A)},$$

but this is seen as a property which a sorting program ought to have, rather than a complete program. It uses the function `sort`, which works by recursively merging the left and right halves of `A` (denoted by `lt(A)` and `rt(A)`).

$$\begin{aligned}\mathtt{sort(A)} \overset{\text{def}}{=} \ & \mathtt{if}\ |\mathtt{A}| \leq 1\ \mathtt{then\ A} \\ & \mathtt{else\ merge(sort(lt(A)), sort(rt(A)))}.\end{aligned}$$

A mergesort program may be derived from this function by adding operational details which have no analogue in a functional language. In particular, the flow of control must be defined using the parallel and sequential composition operators. A suitable program to sort a list `A` of length $2^{\mathtt{n}}$ is given below. The two recursive

sorts are combined in parallel with logical "and" ($\wedge$), then in sequence (;) the two halves of the list are merged. An efficient way to perform the merge is explained in chapter 7.

$$
\begin{array}{lll}
\mathtt{mergesort(n,A)} & \stackrel{\mathrm{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\
& & \mathtt{else}\ \{ \\
& & \quad \mathtt{mergesort(n-1, A_{0..2^{n-1}}) \wedge mergesort(n-1, A_{2^{n-1}..2^n});} \\
& & \quad \mathtt{A := merge(A_{0..2^{n-1}}, A_{2^{n-1}..2^n})} \\
& & \}.
\end{array}
$$

The behaviour of this algorithm on a particular list is illustrated in figure 1.1(b). Observe that the list is sorted *in situ*, something which cannot be expressed in a functional programming language.

1.3.4 Dataflow Programming

The dataflow approach, embodied in the programming language Lucid [WA85], is similar to the functional approach, but represents dynamic behaviour by taking the arguments and values of functions to be (infinite) streams which represent sequences of values over time. As a program executes, successive values of these sequences are generated, and special functions are provided to refer to values at other points in time, such as `next` to access the value at the next time instant. Like functional programs, dataflow programs have elegant mathematical properties, but they do not make efficient use of conventional processors, and this has led several researchers to investigate the potential of special-purpose dataflow computers.

However, there seem to be several obstacles that must be overcome if such machines are to succeed. One of these is the inefficiency of maintaining large data structures, which is as much a problem for the dataflow approach as for the functional approach. Other problems concern the long instruction execution cycle of dataflow computers and the lack of instruction pipelining. Perhaps these problems will be overcome, but after more than a decade of research into dataflow, not one commercial machine has been built.

Some aspects of Tempura are reminiscent of the dataflow approach. For example, the Tempura predicate $\mathtt{tot(X,S)}$ below maintains a running total of the values of $\mathtt{X}$ in the variable $\mathtt{S}$. The sum $\mathtt{S}$ is initially $\mathtt{0}$ and thereafter is incremented by $\mathtt{X}$ on each state.

$$\mathtt{tot(X,S)} \stackrel{\mathrm{def}}{=} \mathtt{S = 0 \wedge S\ gets\ S + X.}$$

However, it is possible to write "non-causal" programs in Lucid,[2] and these will not execute in Tempura, though they are still acceptable formulae of ITL. For example,

[2]By this I mean non-causal in the synchronous interpretation taken in Tempura.

the predicate `diff(X,D)` below specifies that `D` holds the difference between the next and the current value of `X`,

$$\texttt{diff(X,D)} \stackrel{\text{def}}{=} \texttt{D is (next(X)} - \texttt{X)}.$$

This form of expression is not acceptable in Tempura because the value of `D` must be determined before the next value of `X` is known. On the other hand, ordinary imperative constructs, such as assignment and iteration, are present in Tempura but have no analogue in Lucid.

1.3.5 Logic Programming

Logic programs[3] written in languages such as "pure" Prolog are collections of assertions in first-order logic. The assertions, which are cast in the form of Horn clauses, represent the specification of the problem to be solved. The solution is deduced from this specification using techniques designed for automated theorem-proving.

The goal of logic programming, as expressed by Kowalski [Kow79], is that a programmer should only need to supply the specification of a problem, leaving the computer to determine how to actually solve it. If achieved, this goal would lead to a particularly straightforward semantics, but in practice logic programs use control operators, such as *cut*, and an operational version of negation that is not the normal logical one.

Logic programs can be very efficient in certain kinds of application, especially those which involve deduction from a set of rules, such as expert systems, and this has led to enormous interest in special-purpose logic programming machines, notably in the Japanese fifth-generation project. Nevertheless, Prolog is not a good language in which to express algorithmic or real-time programs, which make no use at all of its theorem-proving features.

Prolog programs, like functional programs, are essentially static, but can represent dynamic systems implicitly in similar ways. Recent extensions to Prolog, such as Parlog [Gre87] and Concurrent Prolog [Sha86], allow the programmer to take advantage of concurrent execution, though still within a static framework. To represent dynamic behaviour directly requires a more powerful logic, such as temporal logic, and logic programming languages based on temporal logic have indeed been proposed. Examples include Gabbay's Temporal Prolog [Gab87], and Abadi's Templog [AM87].

Although there are significant differences between these two languages, they are united in their general approach. Both are based on linear-time (not interval)

[3] The term "logic programming" has come to be synonymous with programming in the Horn clause or Prolog style, usually in first-order logic. The work described below is also, in a sense, logic programming since it uses temporal logic as a programming language, but the style is more conventional and the use of problematic features (like negation) is avoided.

temporal logic and therefore do not have the sequential composition operator, and both follow the mainstream of logic programming. It may therefore be appreciated that the aims of these languages differ greatly from those of Tempura; they are certainly not intended for writing efficient deterministic programs.

Unlike Tempura, it is the predicates that are temporal in these languages; variables are static. For instance, Abadi gives an example program to calculate the Fibonacci sequence, which is expressed by the ITL formula below. The predicate $\texttt{Fib(x)}$ holds for different values of $\texttt{x}$ at different times; the first Fibonacci number is $\texttt{0}$, the next one is $\texttt{1}$, and thereafter the one after next is the sum of the current and the next.

$$\begin{array}{l} \texttt{Fib(0)} \wedge \texttt{next Fib(1)} \wedge \\ \texttt{always} \\ \quad \forall \texttt{x}, \texttt{y}, \texttt{z} : \{\{\texttt{Fib(y)} \wedge (\texttt{next Fib(z)}) \wedge (\texttt{x} = \texttt{y} + \texttt{z})\} \supset \{\texttt{next next Fib(x)}\}\}. \end{array}$$

This program cannot be expressed directly in Tempura, but the following program uses dynamic variables to do the same job:

$$\begin{array}{l} \texttt{F} = \texttt{0} \wedge \texttt{next} \, (\texttt{F} = \texttt{1}) \wedge \\ \exists \texttt{F}', \texttt{F}'' : \{\texttt{F}' \texttt{ gets F} \wedge \texttt{next} \, (\texttt{F}'' \texttt{ gets F}' \wedge \texttt{next} \, (\texttt{F is F}' + \texttt{F}''))\}. \end{array}$$

Successive Fibonacci numbers are assigned to $\texttt{F}$ on each step. The local variable $\texttt{F}'$ holds the value of $\texttt{F}$ on the previous state, and $\texttt{F}''$ holds its value on the state before that.

1.3.6 Tokio

The language Tokio, designed by Fujita and his colleagues at Tokyo University [FKTM86], is an alternative executable subset of ITL. It is similar in many respects to Tempura, particularly in the use of temporal constructs to describe control mechanisms, but incorporates the unification and backtracking features of Prolog, and can therefore execute a wider class of programs. It seems to be an appropriate language for prototyping more abstract specifications than Tempura can handle, but complexity issues make the execution of truly abstract specifications impractical. Tokio does not seem capable of efficient implementation on conventional architectures.

A Tokio program which uses backtracking is the parallel quicksort program below. The predicate $\texttt{qs(L)}$ sorts the list $\texttt{L}$ by first partitioning $\texttt{L}$ into two sublists, $\texttt{L}_{\texttt{0..pivot}}$ and $\texttt{L}_{\texttt{pivot..|L|}}$, so that every element of the first is less than every element of the second, and then sorting the two sublists in parallel. If $\texttt{L}$ contains fewer than two elements there is nothing to do, and $\texttt{L}$ is kept stable.

$$\begin{array}{lcl} \texttt{qs(L)} & \stackrel{\text{def}}{=} & \texttt{if } |\texttt{L}| \leq \texttt{1 then stable (L)} \\ & & \texttt{else } \exists \texttt{pivot} : \{ \\ & & \quad \texttt{partition(L, pivot);} \\ & & \quad \texttt{qs}(\texttt{L}_{\texttt{0..pivot}}) \wedge \texttt{qs}(\texttt{L}_{\texttt{pivot..|L|}}) \\ & & \}. \end{array}$$

The parallel recursive calls to qs may take different amounts of time to complete, but by backtracking the Tokio system is able to choose a computation length which allows both to finish. This is possible since no computation length is specified for the base case, when $|L| \leq 1$. However, if a particular computation is chosen in this way one cannot identify the behaviour of the program with its logical interpretation because an infinity of other computations of different lengths would also satisfy qs(L), but they would not be discovered by the Tokio interpreter.

In Tempura the quicksort program can be written in much the same way, either by introducing explicit flags to decide termination, or by using the parallel composition operator (see chapter 8) as follows:

$$
\begin{aligned}
\mathtt{qs(L)} \;\stackrel{\text{def}}{=}\; & \mathtt{if}\ |\mathtt{L}| \leq \mathtt{1}\ \mathtt{then\ empty} \\
& \mathtt{else\ local\ pivot : \{} \\
& \quad \mathtt{partition(L, pivot);} \\
& \quad \mathtt{qs(L_{0..pivot})} \parallel \mathtt{qs(L_{pivot+1..|L|})} \\
& \mathtt{\}.}
\end{aligned}
$$

The Tempura program is deterministic; it maintains the equivalence between program and logical interpretation; and it can be executed much more efficiently than the Tokio version. The two programs produce exactly the same result; indeed, the Tokio program is a specification of the Tempura one. Nevertheless, there are examples that can be executed in Tokio but not at all in Tempura.

1.3.7 Communicating Sequential Processes

The language CSP [Hoa85] is a development of the traditional sequential approach for describing networks of communicating sequential processes, each with its own thread of control. Related approaches include the programming language Occam [Inm84], which incorporates most of the ideas of CSP, and object-oriented languages. All of these languages are founded on a coarse-grained view of parallel processing together with a particular discipline for interprocess communication. They can be implemented efficiently on any architecture that supports concurrent processes and message passing.

These languages have well-defined semantics, and have the advantage that they encode a number of decisions about message passing and information sharing that must be made explicit in Tempura, where they are not a part of the language. However, they seem less suitable for the description of fine-grained concurrency. Furthermore, since they also abstract from timing details, including the mechanism by which interprocess communication is actually achieved, they do not seem to offer the same potential as Tempura for general real-time applications.

In chapter 10, I show how to define a stream-based communication mechanism in Tempura. Operations put(A, X) and get(A, X) may be used to send and receive data X on the stream A; they are similar to the operations A!X and A?X in CSP. With

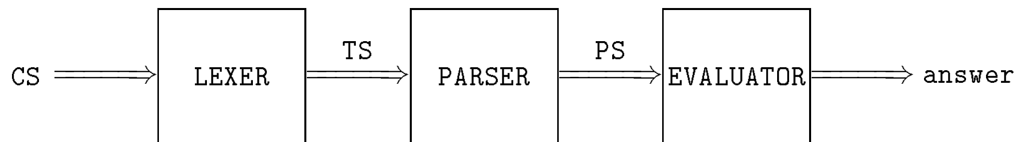


Figure 1.3: A pipelined evaluator for arithmetic expressions

these operators it is possible to construct networks of communicating processes, just as in CSP. Figure 1.3 shows a pipelined system, comprising a lexical analyser, parser and evaluator, for evaluating simple arithmetic expressions which are input in the form of character arrays. It is expressed as the parallel composition of the three processes communicating via the streams `CS`, `TS` and `PS`,

`lexer(CS, TS)` || `parser(TS, PS)` || `evaluator(PS, answer)`.

The characters of the initial expression are fed into the lexical analyser on the stream `CS`. Tokens output from the lexical analyser on the stream `TS` are parsed into reverse polish form, output on the stream `PS`, and evaluated to give the final result in `answer`.

1.3.8 Synchronous Languages

Real-time programs require features for timing, exception handling, and so on that are ill-defined in most conventional languages, and synchronous languages have been proposed as a way to handle these concepts in a rigorous way. Two examples, Esterel [BC84] and Lustre [CPHP87], cover much the same ground, but Esterel is imperative whereas Lustre is declarative. Both languages may be compiled into the form of finite automata, which execute efficiently, and both languages have well-defined semantics. Lustre is closely related to the dataflow language Lucid, which was described in section 1.3.4, and suffers from similar problems in the efficient representation of data structures.

Each of these languages has a formally defined notion of time, just as Tempura does. For instance, Caspi *et al.* define a simple process that counts in steps of `incr` from the initial value `init`; but when `Reset` is signalled, the counter is reset to the value `init`. They then show how this process can be clocked according to a boolean signal `Ck`, so that the counter is only incremented when `Ck` is true. The same thing can be done in Tempura using temporal projection, which will be described in chapter 9. The clocked counter might then be expressed in Tempura as follows:

`count(T, init, incr, Reset) when Ck`.

The variable `T` holds the value of the counter.

Caspi *et al.* go on to describe how to build a simple stopwatch program using their counter, and the stopwatch can be implemented in the same way in Tempura. However, let us take a different line and present a simple stopwatch in the imperative style, closer to the way it might be done in Esterel.

This stopwatch is controlled by a single boolean signal `Run`. When `Run` becomes true the stopwatch starts running and the time `T` is initialised to zero. The time `T` is then incremented by one on every `hsec` time steps (corresponding to one hundredth of a second) until `Run` becomes false. The operator `trap` is used terminate the timing as soon as `Run` becomes false.

```
while true do {
  halt (Run) ∧ T ← 0;
  trap ¬Run : {
    while Run do {
      len (hsec) ∧ T ← T + 1
    }
  }
}.
```

The operators `halt` and `len` are used to indicate when to terminate; the former takes a boolean condition, the latter an integer length.

An important difference between Esterel and Tempura has to do with the duration of actions. In Esterel it is assumed that actions take no time to perform, so for example, an assignment is supposed to happen instantaneously, and Lustre is based on a similar assumption. But in Tempura every action has an associated duration, which may be zero but usually is not.

1.3.9 Restricted Instruction Set Languages

Instead of attempting to formulate ever more complex proof rules to handle the intricate and sometimes badly thought out constructs which appear in many programming languages, it has been suggested by Tang [Tan83], Chandy and Misra [CM88] and Pnueli [Pnu83] that we adopt a simpler low-level programming notation with a few elegant proof rules.

Tang proposes a language (called XYZ/E) based on linear-time temporal logic in which programs are essentially of the form

$$\texttt{always}\,\{\texttt{if}\ c_0\ \texttt{then}\ a_0 \parallel \ldots \parallel \texttt{if}\ c_n\ \texttt{then}\ a_n\},$$

where each c_i is a boolean expression, and the a_i are multiple assignments of the form

$$v_0, \ldots, v_n := e_0, \ldots, e_n,$$

and the operator $\parallel$ denotes parallel composition. Initialisation and termination are handled by special constructs.

Tang does not propose that his language be used directly for high-level programming, but suggests that it may be used as a common base language into which other languages are compiled. To assist the compilation a special program counter is used. This may appear in conditions and in assignments. For example, assigning the value `n` to the program counter is rather like the directive `goto n` in an ordinary sequential language. Tang describes a mapping from Algol-like notation into his language.

The languages proposed by Chandy and Pnueli are similarly based on parallel multiple assignment statements, but provide a more delicate treatment of variables, and handle termination differently. Also, their parallel composition operator has an interleaving semantics, since several assignments may be enabled at once and have contradictory effects. Pnueli describes the semantics in linear-time temporal logic.

Each of these languages offers a possible way to unify various styles of programming in a single framework by providing a mapping from a higher-level language into the parallel assignment form. I believe that Tempura could be used in the same way; at least it is certainly possible to represent repeated multiple assignment programs in Tempura. However, the approach taken in Tempura has been to define the higher-level constructs directly in temporal logic.

The restricted instruction set languages have the advantage of the less complex semantics of linear-time temporal logic (without the chop operator), but they sacrifice the power and clarity of expression that can be achieved with sequential composition. Moreover, Moszkowski has given a translation from ITL into linear-time temporal logic [Mos83], which means that one can, in principle, eliminate sequential composition from Tempura programs if desired.

1.4 My Contribution

Much of this research was done jointly with Ben Moszkowski, and as with nearly all joint research, it is difficult to pinpoint who was responsible for each and every idea. Clearly all the early ideas were due to Moszkowski. The use of ITL for specifying hardware, and the idea of using it as a programming language were entirely his. In this area I have only formalised ITL in a different way and clarified some of the ideas (chapters 3 and 4). Moszkowski also wrote the first interpreter for Tempura in Lisp, but I have since written a much faster interpreter in C, and have made a number of improvements to the original design.

The main contribution of this thesis is the introduction of inertial variables into the language, using the technique described in chapter 6. Not only does this make the description of sequential systems very much easier, it also makes possible the use of a new operator for coarse-grained parallel composition, as described in chapter 8.

Apart from one example, which was originally done by Moszkowski and later improved by me, all of the experimental work described in this dissertation is entirely

my own. In particular, much of the treatment of real-time time systems in chapter 9 is completely new.

My preliminary attempts at using the HOL theorem prover to verify Tempura programs were at the suggestion of Mike Gordon, who also helped me to get going. Other than that, all the work on verification and transformation is my own. This topic is introduced in chapter 5. In later chapters I state several theorems about programs, but in order to avoid misleading the reader I should point out here that most of them have not been rigorously proved.

Chapter 2

A Practical Introduction

This chapter provides an informal introduction to the temporal logic programming language, Tempura. The introduction is based on an interpreter for Tempura. All the main language constructs are described, and each is illustrated with a small example. The chapter ends with a complete description of two parallel matrix multiplication programs. The purpose of this chapter is to give you an intuitive understanding of Tempura in preparation for the theoretical work that follows.

Tempura is a computer programming language that is especially good at expressing temporal behaviour. Suitable applications for Tempura include the design of parallel algorithms and real-time systems. But Tempura is doubly useful when program correctness is important, because it is a language with strong mathematical foundations.

Later on I will give more substance to this claim, but first I want to give a feeling of how Tempura looks in practice. I will show how Tempura handles a number of different styles of programming. I shall also try to convey how its mathematical origins manifest themselves. This will be seen, for example, in the way that all Tempura operators are defined in terms of just a handful of primitive ones; that there is a precise notion of program equivalence; and that strict checks can be made to ensure a variable is uniquely defined before it is used.

This introduction is based on an existing Tempura interpreter which runs on a uniprocessor; and the examples described below form a single continuous session with the interpreter. Used in this way Tempura is best regarded as a simulation or prototyping language, but its potential as an honest-to-goodness parallel programming language ought not to be in doubt. This will anyway be enlarged upon in subsequent chapters.

2.1 The Interpreter

The interpreter prompts the user for a command with a message indicating the current global state number, which is initially `0`.

```
Tempura 0>
```

Possible responses include commands for defining and executing programs, as well as various assertions about the current state. An example of the latter kind would be to assert that the variable `A` has value `99`.

```
Tempura 0> A=99.
```

(user input is always followed by a full stop, "."). This assertion can be checked by asking for the value of `A`:

```
Tempura 0> output(A).
A=99
```

Consistency is enforced. An attempt to assert that `A` is also a five-element list elicits the following error message

```
Tempura 0> list(A,5).
***Tempura error: found expression of type integer when expecting list
   Evaluating: list(A,5)
```

since a variable cannot have two values at once. But it is possible to imagine a different variable, also called `A`, which is a list. In other words there does exist an `A` which designates a five-element list.

```
Tempura 0> exists A: list(A,5).
```

Of course, this version of A only exists for the duration of the command. It is, in other words, a local variable.

The following command introduces a list `A`, sets each of its elements, and prints it out. The logical conjunction operator `and` (also written "$\wedge$") combines the operations in parallel.

```
Tempura 0> exists A:{
         >     list(A,5) and
         >     forall i<5:(A[i] = i mod 2) and
         >     output(A)
         > }.
A=[0,1,0,1,0]
```

But the top-level version of `A` retains its original value.

```
Tempura 0> output(A).
A=99
```

The operator `exists` denotes existential quantification (also written "$\exists$"), which is the logical way to declare local variables. The other quantification operator, `forall` asserts its argument for all values of the index variable in the given range, in this case $0 \leq i < 5$. It is a bounded form of the usual universal quantifier, "$\forall$".

2.2 Predicates, Functions and Macros

Predicates in Tempura take the role of procedures in ordinary imperative languages. Logically, they are just boolean-valued functions, but they are distinguished from ordinary functions by their manner of use; the interesting thing about a function is the value it returns, whereas the interesting thing about a predicate is the pattern of behaviour it defines. Functions and predicates are defined in exactly the same way.

New definitions may be introduced by the command `define`, whose syntax is: `define` *object* = *definition*. For instance, the command below defines a function to test whether a list is sorted in ascending order (the notation |`l`| denotes the length of the list `l`).

```
Tempura 0> define ordered(l) = forall i<|l|-1 : l[i] <= l[i+1].
```

That is, a list `l` is ordered if each element is less than or equal to the next.

An alternative form of definition uses the command `defmacro`. For instance, the following predicate asserts its second argument (itself a predicate) for all elements of a list `l`:

```
Tempura 0> defmacro forallin(l,p) = forall i<|l| : p(l[i]).
```

This predicate is just an alternative form of universal quantification, and is equivalent to a conjunction of terms. For instance,

$$\texttt{forallin}([1,3,5],\texttt{p}) \quad = \quad \texttt{p}(1) \land \texttt{p}(3) \land \texttt{p}(5).$$

It is predefined in the more general form $\texttt{forall x} \in \texttt{l} : \texttt{p}$.

A function or predicate defined in the first way, using `define`, is like a normal procedure and calls its arguments by reference in much the same way as a Fortran subroutine would; when called, it is supplied with a list of pointers to the actual arguments. On the other hand, a definition made with `defmacro` is expanded textually where it is used, in a similar way to macros in other languages. Definitions made with `define` are generally more economical of resources than those made with `defmacro`, but do not always work as required.

For instance, suppose that you want to define a predicate $\texttt{tri}(\texttt{b},\texttt{p},\texttt{p}',\texttt{p}'')$ which is to execute `p` if `b` is `1`, $\texttt{p}'$ if `b` is `0` or $\texttt{p}''$ if `b` has some other value. This predicate may be defined using `defmacro` as follows:

```
Tempura 0> defmacro tri(b,p,p′,p″) =
         >     if b=1 then p else if b=0 then p′ else p″.
```

where the conditional `if` b `then` p `else` p' has its usual meaning; that is, if b is true do p, otherwise do p'. When `tri` is called it expands in-place. Thus, in the following test,

```
Tempura 0> tri(1,output("hi"),output("lo"),output("error")).
Output="hi"
```

it expands to the formula

```
if 1 = 1 then output("hi")
else if 1 = 0 then output("lo")
else output("error").
```

If it had been defined with `define` the system would have attempted to evaluate all arguments before executing the body of `tri`, and this would not have worked, of course. However, this example does not really justify the need for macros; for that we must wait until section 2.4.4.

Definitions of either form may be recursive. A mergesort, for example, is defined below. It sorts a list by recursively merging the left and right halves of the list.

```
Tempura 0> define sort(l) = {
         >    if |l|<=1 then l
         >    else merge(sort(lt(l)),sort(rt(l)))
         > }.
Tempura 0> define merge(l,l') = {
         >    if |l|=0 or |l'|=0 then l^l'
         >    else if hd(l) <= hd(l')
         >         then cons(hd(l),merge(tl(l),l'))
         >         else cons(hd(l'),merge(l,tl(l')))
         > }.
```

The functions $\mathtt{lt}(\mathtt{l})$ and $\mathtt{rt}(\mathtt{l})$ denote the left and right halves of `l`; that is, $\mathtt{lt}(\mathtt{l}) = \mathtt{l}_{0..|\mathtt{l}|/2}$ and $\mathtt{rt}(\mathtt{l}) = \mathtt{l}_{|\mathtt{l}|/2..|\mathtt{l}|}$. The function $\mathtt{hd}(\mathtt{l}) = \mathtt{l}_0$, the first element of the list `l`, and $\mathtt{tl}(\mathtt{l}) = \mathtt{l}_{1..|\mathtt{l}|}$, the list comprising every element of `l` except the first. The other function, $\mathtt{cons}(\mathtt{x}, \mathtt{l})$, denotes the list $[\mathtt{x}]\hat{}\,\mathtt{l}$, whose head is `x` and whose tail is `l`. The mergesort is exercised below.

```
Tempura 0> exists l,l': {
         >     input(l) and l'=sort(l) and output(l',ordered(l'))
         > }.
l=[6,3,2,0,7,1,4,5].
l'=[0,1,2,3,4,5,6,7] ordered(l')=true
```

It can be tested on a few lists in this way, but to really be sure that `sort` will sort any list of integers one must formally prove it. Chapter 5 explains how.

Functions and predicates are stored internally as anonymous functions, known as lambda expressions (as in LISP, for example). Here is the definition of `sort`:

```
Tempura 0> output(sort).
sort=lambda(l): if |l| <= 1 then l else merge(sort(lt(l)),sort(rt(l)))
```

Lambda expressions are sometimes useful in their own right. Consider the predicate $\mathtt{fsum}(\mathtt{n}, \mathtt{f})$ below, which sums the first `n` values of the function `f` (starting from `0`).

```
Tempura 0> define fsum(n,f) = if n=0 then 0 else f(n-1)+fsum(n-1,f).
Tempura 0> output(fsum(4,lambda(i): i)).
fsum(4,lambda (i):{i})=6
```

time	Y	N
0	1	n
1	x	$n - 1$
2	x^2	$n - 2$
⋮	⋮	⋮
n	x^n	0

Figure 2.1: Calculation of x^n.

Use of a lambda expression in such situations avoids having to define a new named function.

Both `define` and `defmacro` are system (meta-level) commands. A new definition simply replaces an old definition of the same name without checking whether the name was previously defined. They should not be used to simulate dynamic behaviour, and as we shall see they need not be.

2.3 Programs

So far, all assertions have referred to a single point in time, whereas interesting computations generally involve quantities that change with time. One of the strengths of Tempura is its ability to talk about evolving computations in a precise way. Mostly, it does this with just three new primitive operators: `empty`, `next` and *chop* (written ";"). These will be described shortly, but first some general comments about variables and programs.

Tempura variables come in two flavours: *state variables* which change with time, and *static variables* which do not; once assigned, the value of a static variable is fixed. By convention, names beginning with an upper case letter are state variables, all others are static. It is very important to remember this convention.

A program describes how the state of a particular system changes over an interval of time. Time is measured on a virtual timescale whose units correspond to computational steps; they could relate directly to real time units, but need not. The length of a computation is simply the number of steps it takes (one less than the number of states).

Take the computation shown in figure 2.1, for example. This shows how the values of two variables, Y and N, change during a calculation of x^n. Initially $Y = 1$ and $N = n$, and on each unit of time Y is multiplied by x and N is decremented until finally $N = 0$ and $Y = x^n$. The whole computation takes n steps.

There are two ways of viewing this computation which give rise to two corresponding ways to program the calculation in Tempura.

1. The computation is composed of `n` unit steps, one after the other. On each step `Y` is multiplied by `x` and `N` is decremented.

2. The value of `Y` on the next state is always `x` times its value on the current state, and the value of `N` is one less than its current value. The computation terminates when `N` = `0`.

Naturally, these two views are logically equivalent, and in Tempura this can be formally proved. But before getting too far ahead, I need to introduce some operators which will be needed later on.

2.4 Operators

Tempura provides several built-in operators for constructing programs. Some of them have been encountered already, others are described below. Only the operators =, `and`, `if`...`then`...`else`..., `exists`, `empty`, `next`, chop and `prefix` are primitive, the rest can be defined in terms of these.

2.4.1 Empty

The predicate `empty` is used to mark termination, in other words it is true on (and only on) the last state of a computation. It can be tested to see if a computation has entered its final state, or it can be asserted in which case it forces termination. The program

```
empty and output("hello world")
```

outputs the string "`hello world`" and terminates straight away. It can be executed with the command

```
Tempura 0> run {empty and output("hello world")}.
State 0: Output="hello world"

Done!  Computation length:  0.
```

The command `run` is another system command. The effect of running a program is to generate the output that it defines. Notice I said *the* output because Tempura programs must be deterministic. Any two runs of the same program will produce the same output. The following examples will not execute even though they contain

no errors of syntax.

```
Tempura 0> run {empty and if X then output("hello world")}.
***Tempura error: state #0 is not completely defined.
   Evaluating: if X then {output("hello world")} else {true}
   Undefined: { X }

Tempura 0> run {output("hello world")}.
***Tempura error: the interval length is undefined.
   Evaluating: run output("hello world")
```

The first example makes an attempt to use an undefined variable; the second contains no indication of when to terminate. A termination statement, or an indication of computation length, is an essential part of every Tempura program.

2.4.2 Next

The operator `next` is used to describe what happens next, where "next" means "after one unit of virtual time". For example, the program

```
A=0 and next{A=1 and next{A=2 and empty}}
```

gives `A` the values `0`, `1` and `2` on successive states of a computation of length two. Thankfully, there are more concise ways to write programs such as this.

2.4.3 Computation Length

Part of the effect of `next` is to ensure that something does happen next, and consequently that `empty` is false. This condition means that the length of any computation can be defined in terms of `next` and `empty`. For example the unit-length computation is given by `next empty`,

```
Tempura 0> run{next empty}.

Done!  Computation length:  1.
```

but there is a built-in operator, `skip`, with this definition. A more general predicate `len(n)`, to specify a computation length of `n`, can be defined as `next` applied to empty `n` times,

```
Tempura 1> define len(n) = if n=0 then empty else next{len(n-1)}.
```

though again it is predefined.

2.4.4 Always

Something is considered to happen always if it happens immediately and then again after each successive step. As a first attempt at defining this, one might try:

```
Tempura 1> defmacro always(p) = {p and next always(p)}.
```

but this is wrong because at the end of an interval, when `empty` is true, there is no next step. This version of `always` will therefore overrun the end of the interval.

```
Tempura 1> run {skip and always(A=0 and output(A))}.
State   0: A=0
State   1: A=0

***Tempura error: attempt to re-assign interval length.
   Evaluating: next always((A = 0) and output A)
```

An error is signalled when the `next` operator tries to make `empty` false.

A correct definition checks for termination before asserting `always(p)` on the next state.

```
Tempura 2> defmacro always(p) = {p and if ~empty then next always(p)}.
Tempura 2> run {skip and always(A=0 and output(A))}.
State   0: A=0
State   1: A=0

Done!  Computation length:  1.
```

The operator `always` is one of the most important temporal operators, and is therefore built into the interpreter.

A number of new operators can be defined in terms of `always`. For instance temporal equality, which asserts that `A` and `B` are equal throughout the computation,

```
Tempura 3> defmacro is(A,B) = always(A=B).
```

and the predicate `halt(A)`, which defines a termination condition, `A`.

```
Tempura 3> defmacro halt(A) = always(if A then empty else ~empty).
```

This definition says that `empty` is true exactly when the condition `A` holds, which is another way of saying that the computation terminates as soon as `A` becomes true. For instance, the following program inputs successive values of the variable `X`, and terminates when `X` $<$ `0`. The values of `X` are input by the user (terminated by a full stop ".").

```
Tempura 3> run exists X:{always(input(X)) and halt(X < 0)}.
State   0: X=1.
State   1: X=2.
State   2: X=3.
State   3: X=-1.

Done!  Computation length:  3.
```

Both `halt` and `is` are predefined, the latter being one of many binary infix operators; that is, it may be written: `A is B`.

2.4.5 Assignment

Another useful binary infix operator is temporal assignment. The temporal assignment $A \leftarrow B$ is another way of writing `assign(A, B)`, where the predicate `assign(A, B)` is defined below. It asserts that the final value of `A` is equal to the initial value of `B`.

```
Tempura 6> define assign(A,B) =
         >     exists x:{x = B and always(if empty then A = x)}.
```

This definition works by storing the initial value of `B` in a local variable `x` and then at the end of the interval, when `empty` is true, assigning the stored value to `A`. Note that `x` is a static variable, and therefore retains its value.

Consider, again, the computation in figure 2.1. This forms the `n`th power of `x` in the variable `Y` by repeated multiplication, using an auxiliary variable `N` to count the steps. The overall effect is that `Y` is assigned x^n in `n` steps and `N` finishes up with the value `0`. This is specified by the predicate `exp_spec(x, n, Y, N)`, where

```
Tempura 6> define exp_spec(x,n,Y,N) = {
         >     Y = 1 and N = n and
         >     len(n) and
         >     Y <- x**n and N <- 0
         > }.
```

The assignments take place in parallel. For example,

```
Tempura 6> run exists Y,N: {exp_spec(2,3,Y,N) and always output(Y,N)}.
State 0: Y=1 N=3
State 1: Y=? N=?
State 2: Y=? N=?
State 3: Y=8 N=0
```

A question mark, "?", indicates that the corresponding variable is undefined.

2.4.6 Sequential Behaviour

It is possible to build quite sophisticated programs using only `next` and `empty`, and some examples are presented later on, but the sequencing operator, chop, gives even greater capabilities. Chop, which is written ";", breaks a single computation into two subcomputations and describes what happens on each.

The two subcomputations intersect at a single state, so the length of the combined computation is equal to the sum of their individual lengths.

```
Tempura 9> run {len(2);skip}.

Done!  Computation length:  3.
```

Each of the subprograms references its own local version of `empty` so that it knows when to finish, as may be seen in the following program:

```
Tempura 12> run exists E,E′: {show_empty(E,E′) and always output(E,E′)}
          > where
          > define show_empty(E,E′) = {
          >     len(2) and E is empty;
          >     skip and E′ is empty
          > }.
State   0: E=false E′=?
State   1: E=false E′=?
State   2: E=true  E′=false
State   3: E=?     E′=true

Done!  Computation length:  3.
```

On the second state the first subprogram ends and the second begins. The operator `where` in this program means the same as `and`, but limits the scope of the second definition to the body of the first.

Superficially, chop behaves like a statement terminator in a conventional imperative language (the semicolon in Pascal, for example). A simple example is the following program which increments `I` twice.

```
Tempura 15> run exists I: {I=0 and inc2(I) and always output(I)}
          > where
          > define inc2(I) = {
          >     skip and I <- I+1;
          >     skip and I <- I+1
          > }.
State   0: I=0
State   1: I=1
State   2: I=2

Done!  Computation length:  2.
```

Assignment on the unit interval, such as $\texttt{skip} \wedge \texttt{I} \leftarrow \texttt{I} + \texttt{1}$ in the definition of `inc2`, turns out to be a very common operation. So much so that it is worth defining a special version of assignment, called unit-assignment, to handle this case.

2.4.7 Unit-Assignment and Initialisation

Unit-assignment is an assignment from one state to the next. It is another binary infix operator, with the syntax $\texttt{A} := \texttt{B}$, and defined simply as:

```
Tempura 17> defmacro uassign(A,B) = {skip and A <- B}.
```

Of course, an implementation of unit-assignment can be much more efficient than the above definition might suggest (which is the main reason for giving it special syntax).

Another kind of assignment that occurs frequently is initialisation; that is, the assignment of an initial value to a variable. This is easily achieved in zero-time using equality,

```
Tempura 17> defmacro iassign(A,B) = {empty and A = B}.
```

but, as with unit-assignment, it is convenient to have a special operator, `A` $\Leftarrow$ `B`, with the execution time built in. This may also be written `A == B` in programs.

2.4.8 Multiple Assignments

Multiple parallel assignments can be written in an abbreviated syntax. For instance, the assignments

```
A := 0 and B := 1 and C := 2
```

may be written

```
A,B,C := 0,1,2
```

and similarly in the general case for an arbitrary number of assignments. In fact, all the assignment operators, $=$, $:=$, $\Leftarrow$ and $\leftarrow$ may be abbreviated in this way (as may the operators `gets` and `stable`, which will be encountered shortly).

2.4.9 Iteration

Iteration can easily be expressed as a recursive application of chop. For example, the familiar while-loop behaves as follows:

```
Tempura 17> defmacro whiledo(b,p) =
          >     if b then {p;whiledo(b,p)} else empty.
```

That is, if `b` is false the loop reduces to `empty`, otherwise `p` is executed and then `b` is tested again. However, this definition must be used with care, for if `p` is a zero-length computation `b` is repeatedly tested on the same state, which means that it cannot possibly change between successive tests. Thus, if `b` is initially true the program will loop forever. A correct logical definition of the while-loop is given in chapter 3.

The loop `whiledo(b,p)` is, of course, predefined with the time-honoured syntax, `while` b `do` p, and a number of other iterative operators are also predefined:

1. `repeat` p `until` b, where b is a boolean expression.

2. `for` $i < n$ `do` p, where i is a variable and n an integer expression.

3. `for` $i \in l$ `do` p, where i is a variable and l a list expression.

4. `for` n `times do` p, where n is an integer expression.

Hopefully, the intended meanings of these loops are clear, but they will anyway be discussed again later. They can all be expressed in much the same way as `whiledo`.

The following little iterative program computes $\mathtt{x}^{\mathtt{n}}$ in `n` unit steps using the predicate `whiledo`.

```
Tempura 17> define exp_pgm(x,n,Y,N) = {
          >    Y,N == 1,n; whiledo(N~=0, Y,N := Y*x,N-1)
          > }.
```

A test run shows that this program meets its specification, `exp_spec(x, n, Y, N)`, for `x` = 2 and `n` = 3,

```
Tempura 17> run exists Y,N: {exp_pgm(2,3,Y,N) and always output(Y,N)}.
State   0: Y=1 N=3
State   1: Y=2 N=2
State   2: Y=4 N=1
State   3: Y=8 N=0

Done!  Computation length:  3.
```

and it can be proved that the specification is met in the general case, in other words, if the program `exp_pgm(x, n, Y, N)` executes on a given interval, then the specification `exp_spec(x, n, Y, N)` is also true on the interval. This is discussed in more detail in chapter 5.

2.4.10 Gets

An alternative approach to the computation of $\mathtt{x}^{\mathtt{n}}$ is to treat each variable separately, observing how its value changes from each state to the next, so that `N` is decremented from state to state and `Y` is always multiplied by `x`. The predicate `gets` expresses this kind of repeated assignment.

```
Tempura 20> define gets(A,B) =
          >     always
          >         if ~empty then
          >             exists x: {x=B and next(A=x)}.
```

It is another binary infix operator, `A gets B`, and may take lists of arguments in the same way as the other assignment operators.

If an expression is repeatedly assigned to itself, `A gets A` for example, its value does not change, and it is said to be stable.

```
Tempura 20> defmacro stable(A) = A gets A.
```

Similarly, `stable(A,B,C)` denotes that variables `A`, `B` and `C` are all stable, and likewise in the general case.

You might not think that this operator is much use, but as things stand one must actively ensure that a variable remains stable when one doesn't want its value to change. For instance, in section 2.4.5 we saw that temporal assignment has no built-in assumption about stability. However, in many situations storage is assumed to remain stable unless explicitly changed, and stability can then be implemented at no cost.

In chapter 6 I discuss a way to eliminate the need for `stable` altogether by introducing a new operator which automatically maintains stability. The construct

`local` $v : p$

introduces a new variable which behaves just like an ordinary program variable. However, the additional checking enforced by `stable` is sometimes useful.

Using `gets` it is possible to come up with another program, $\mathtt{exp_pgm}'(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N})$, to calculate $\mathtt{x}^{\mathtt{n}}$.

```
Tempura 20> define exp_pgm'(x,n,Y,N) = {
          >     Y,N = 1,n and halt(N=0) and Y,N gets Y*x,N-1
          > }.
```

This program has exactly the same behaviour as the previous version, which is hardly surprising because the two programs are logically equivalent. That is,

$$\mathtt{exp_pgm}(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N}) \;=\; \mathtt{exp_pgm}'(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N}).$$

You can see that this is true for $\mathtt{x} = 2$ and $\mathtt{n} = 3$ by simply executing the programs.

```
Tempura 20> run exists Y,N: {exp_pgm'(2,3,Y,N) and always output(Y,N)}.
State   0: Y=1 N=3
State   1: Y=2 N=2
State   2: Y=4 N=1
State   3: Y=8 N=0

Done!  Computation length:  3.
```

More generally, the equivalence can be formally proved. In fact, any occurrence of `gets` can be replaced by an equivalent while-loop.

2.4.11 Extended and Prefix Computations

When two processes are combined in parallel with the logical "and" operator, there must be a single interval on which they both execute without error. In particular, this means that they must run for exactly the same number of steps. For instance, the conjunction $\mathtt{len(3)} \wedge \mathtt{len(5)}$ cannot be executed satisfactorily. Often, however,

one wants to run a number of processes in parallel and simply wait until they all have finished. This is achieved in Tempura by "extending" the shorter processes using a new operator `extend`. A process is extended by composing it in sequence with `true` (which executes on any interval).

```
Tempura 23> defmacro extend(p) = {p; true}.
```

For example, a process that takes three steps may be extended to wait for one that takes five, so the program

```
extend(len(3)) and len(5)
```

would run for five units of time. A more relaxed form of parallel composition can be defined in a similar way to `extend`. The construction $p \parallel p'$ combines two processes p and p' in parallel so that the shorter is extended until both have finished. It may be defined in Tempura by introducing flags `E` and `E′` to mark when each process finishes. The parallel composition finishes when both flags are true.

```
Tempura 23> defmacro par(p,p′) = exists E,E′: {
          >     {p and E is empty; stable(E)} and
          >     {p′ and E′ is empty; stable(E′)} and
          >     halt(E and E′)
          > }.
```

There is a corresponding iterative construct `forpar` $i < n : p$ for combining a number of processes in parallel.

In some circumstances, however, one wants the parallel composition to terminate as soon as the *first* of the individual processes finishes. This might be the case when one of the processes has the task of monitoring for exceptions and terminating the others when something is amiss. For instance, a timeout can be effected by running a timer in parallel with another process in this way.

Thus, there is an alternative kind of parallel composition that meets this requirement. Instead of extending the shorter process, it takes the prefix of the longer one, using the operator `prefix`. The prefix of a program, `prefix` p, runs successfully on any prefix of any interval that satisfies p. For instance, a five-unit computation may be prefixed to stop after three steps, so the program

```
len(3) and prefix(len(5))
```

finishes after three steps. However, `prefix` cannot be defined in terms of the operators we have so far. It is another primitive operator of ITL.

With `prefix`, any two processes, p and p', may be combined so that both are terminated as soon as the first of them finishes. This is defined in much the same way as `par` above, but takes the prefix of the longer process rather than extending

the shorter.

```
Tempura 23> defmacro bar(p,p') = exists E,E': {
          >     prefix {p and E is empty} and
          >     prefix {p' and E' is empty} and
          >     halt(E or E')
          > }.
```

The symbolic form of bar(p, p') is $p \,\rule{0.3em}{1em}\, p'$.

Suppose, for example, that one wants to search two strings `s` and `s'` in parallel for occurrences of the word `w`. The search may be implemented as a for-loop.

```
Tempura 23> define search(w,s,P) =
          >     for i<|s|-|w| do
          >         if s[i..i+|w|] = w then P := cons(i,P) else P := P.
```

It returns in `P` a list of the positions of occurrences of `w`. Suppose also that the most important thing is to locate one occurrence of `w` as quickly as possible, so the search may terminate when `w` is first encountered. The parallel search of two strings `s` and `s'` may then be done as follows, using both forms of composition:

```
Tempura 23> define par_search(w,s,s',P,P') =
          >     bar(halt(|P| > 0 or |P'| > 0),
          >         par(search(w,s,P), search(w,s',P'))).
```

Shown below is a test run to search the strings

```
s = "but thought's the slave of life, and life's time's fool"
s' = "and time, that makes survey of all the world, must have a stop"
```

for the first occurrence of the word "`time`".

```
Tempura 23> run local P,P': {
          >     P,P' = [],[] and par_search("time",s,s',P,P') and
          >     always output(P,P')
          > }.
State   0: P=[] P'=[]
State   1: P=[] P'=[]
State   2: P=[] P'=[]
State   3: P=[] P'=[]
State   4: P=[] P'=[]
State   5: P=[] P'=[4]

Done!  Computation length:  5.
```

On the final state `P'` holds the position of the first occurrence of the string "`time`" in `s'`.

2.4.12 Other Operators

There is, of course, no limit to the number of useful operations that can be defined in terms of the primitive ones, and a few more will be introduced in due course. But some concepts cannot be expressed in this way. One example is the idea of separate processes having different rates of progress. This gives rise to the concept of *temporal projection*, which will be discussed in chapter 9.

2.5 A Complete Example

Finally, let us consider a complete example. The example to be tackled is matrix multiplication, a problem that crops up in numerous applications, and one that gives considerable scope for parallelism. The problem is specified as follows: Given an $\mathtt{n} \times \mathtt{n}$ matrix A and an n element vector X, calculate the vector Y such that $\mathtt{Y} \leftarrow \mathtt{X} \times \mathtt{A}$, that is

$$\mathtt{forall}\ \mathtt{i} < \mathtt{n} : \mathtt{Y}_\mathtt{i} \leftarrow \sum_{\mathtt{j}=0}^{\mathtt{n}-1} \mathtt{X}_\mathtt{j}\mathtt{A}_{\mathtt{ji}}.$$

Here is a Tempura representation of the specification, using the usual programming notation, $\mathtt{A}[\mathtt{i},\mathtt{j}]$ to denote the list element $\mathtt{A}_{\mathtt{ij}}$:

```
define mult_spec(n,A,X,Y) =
    forall i<n:
        Y[i] <- fsum(n,lambda(j): X[j]*A[j,i]).
```

where the summation function fsum(n, f) is defined as above

```
defmacro fsum(n,f) = if n=0 then 0 else f(n-1)+fsum(n-1,f).
```

The specification can be implemented in a straightforward way by forming all the inner products in parallel, and computing each one by sequential addition, as below.

```
define mult(n,A,X,Y) =
    forall i<n: {
        Y[i] == 0;
        for j<n do {
            Y[i] := Y[i]+X[j]*A[j,i]
        }
    }.
```

The following program tests the multiplication predicate, `mult`, and combines the specification in parallel to check the result. The variables are all local, and so

automatically remain stable between assignments.

```
define mult_test(n) =
    local A,X,Y,Y′: {
        input(A,X) and
        mult(n,A,X,Y) and
        mult_spec(n,A,X,Y′) and
        always output(Y,Y′)
    }
    where array(A,n,n)
    and list(X,n)
    and list(Y,n)
    and list(Y′,n).
```

Here array(A, m, n) defines A to be a fixed $m \times n$ array, which is actually represented as a list of lists. For instance, the matrix a below is a 3×3 array.

```
define a = [[ 1, 2, 3],
            [ 4, 5, 6],
            [ 7, 8, 9]].
```

If all of the above definitions are stored in the file "matrix-mult.t", then the following sequence of commands cause the test program to be run.

```
Tempura 28> load "matrix-mult".
[Reading file matrix-mult.t]
Tempura 28> run mult_test(3).
State   0: A=a.
State   0: X=[1,2,3].
State   0: Y=[0,0,0]     Y′=[?,?,?]
State   1: Y=[1,2,3]     Y′=[?,?,?]
State   2: Y=[9,12,15]   Y′=[?,?,?]
State   3: Y=[30,36,42]  Y′=[30,36,42]

Done!  Computation length:  3.
```

It can be seen that the program correctly calculates the product of X with A, since $Y = Y'$ in the final state.

This algorithm takes n steps. However, the inner product calculation is also open to parallel attack. It can be performed in log(n) steps by doing the additions pairwise in parallel using the "divide and conquer" approach. This method will be further discussed in chapter 7.

Let us look instead at a completely different approach to computing the product. In our first stab at the problem the data remained static throughout the calculation, witness the assertion stable(A, X). But for a real application this may not be the best way to proceed. For instance, if each inner product calculation were to be

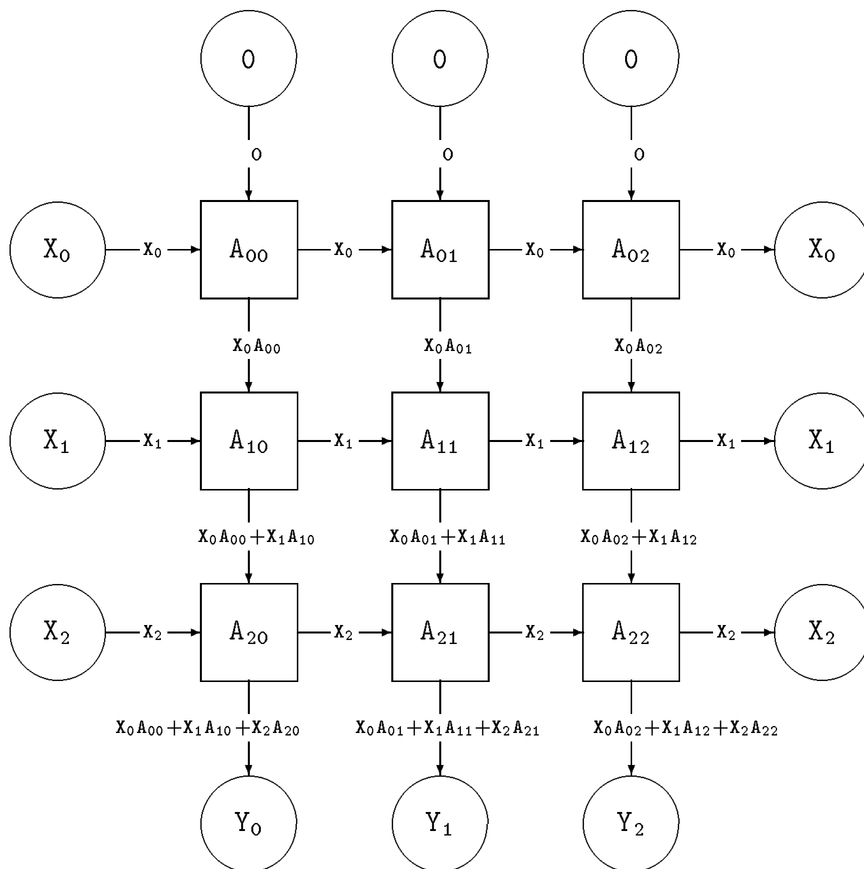


Figure 2.2: A systolic array for calculating a stream of product vectors `Y` from a stream of vectors `X` and a fixed matrix `A`.

---

performed on its own processor, then each processor in the preceding algorithm would need the same element of `X` at each step; and if a stream of vectors were to be multiplied in turn by the same matrix, then the output would be produced at a rate `n` times less than the rate of input. The algorithm above does not perform well under either of these conditions. An application in which both conditions apply is image transformation, where a large number of vector co-ordinates must be transformed in the same way.

If execution speed is important a number of parallel processors may be used to boost performance, and a better way to perform the multiplication is to keep the transformation matrix fixed in an array of cells and have the streams of input and output vectors flow through this array accumulating product terms as they go. Eventually a new output vector will appear on every step. The method is illustrated in figure 2.2.

Really, of course, the problem specification has been changed slightly. What is

asked for now is an algorithm that always produces a new output vector, Y, some fixed time after being supplied with a new input, X. The new specification uses the predicate delay(n, A, B) which specifies that the value of A appears at B after fixed delay, n. A list is used locally to hold the delayed values.

```
defmacro delay(n,A,B) =
    exists C: {
        C[0] is A and
        forall i<n: {
            C[i+1] = B and C[i+1] gets C[i]
        } and
        B is C[n]
    }
    where list(C,n+1).
```

Notice that B retains its initial value over the first n states whilst values of A are percolating through.

The stream multiplication algorithm is then specified in much the same way as the one-shot multiplication algorithm above, with the assignment replaced by a delay.

```
define stream_mult_spec(n,A,X,Y) =
    forall i<n: delay(d(n),fsum(n,lambda(j): X[j]*A[j,i]),Y[i]).
```

The required values of Y begin to emerge after some fixed delay, d(n).

The final program takes the form of an $n \times n$ array of nodes interconnected in both the north-to-south and west-to-east directions to form a square grid, as shown in figure 2.2. Elements of the matrix A are stored one per node, and elements of the input vector X are supplied, appropriately delayed, at the western edge. On each step node (i, j) adds the term $X_i \times A_{ij}$ into Y_j and passes Y_j on to the south and X_i on to the west. However, a node does not need to know its position in the array, so all nodes are identical.

```
define node(Aij,N,S,E,W) = S gets N+W*Aij and E gets W
```

After a delay of $2 \times n + 1$ steps the corresponding output vector Y appears at the southern edge. Thus, for this algorithm $d(n) = 2 \times n + 1$.

The whole array is the parallel composition of the $n \times n$ nodes, together with delays at the eastern and southern edges to synchronise the elements of X and Y, and a source of zeroes at the northern edge to initialise the elements of Y. Nothing

is connected to the western edge; the elements of X are simply discarded.

```
define stream_mult(n,A,X,Y) = {
    exists NS,WE: {
        forall i<n: {
            NS[0,i] is 0 and
            delay(i+1,X[i],WE[i,0]) and
            delay(n-i,NS[n,i],Y[i]) and
            forall j<n:
                node(A[i,j],NS[i,j],NS[i+1,j],WE[i,j+1],WE[i,j])
        }
    }
    where zero_array(NS,n+1,n) and zero_array(WE,n,n+1)
}.
```

The arrays NS and WE denote north-to-south and west-to-east connections. There are n sets of $n+1$ node-to-node connections in each direction, the end connections in each set being used for input and output. All of these connections are initialised to zero by the predicate zero_array (there is a corresponding initialisation predicate zero_list for lists).

The test harness takes successive values of X from an input list xlist, and again executes the specification in parallel with the program to check results.

```
define stream_mult_test(n) =
    local A,X,Y,Y′: {
        generate_inputs(n,A,X) and
        stream_mult(n,A,X,Y) and
        stream_mult_spec(n,A,X,Y′) and
        always output(Y,Y′)
    }
    where
    define generate_inputs(n,A,X) = {
        exists xlist: {
            input(A,xlist) and
            for x in xlist do {skip and X = x};
            len(d(n)-1) and X is xlist[0]
        }
    }
    and array(A,n,n)
    and list(X,n)
    and zero_list(Y,n)
    and zero_list(Y′,n).
```

It is necessary to wait for $2 \times n$ steps after the final input before the final output appears.

A test run using the input list

```
define xl =
    [[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]].
```

goes as follows:

```
Tempura 31> run stream_mult_test(3).
State   0: A=a.
State   0: xlist=xl.
State   0: Y=[0,0,0]     Y'=[0,0,0]
State   1: Y=[0,0,0]     Y'=[0,0,0]
State   2: Y=[0,0,0]     Y'=[0,0,0]
State   3: Y=[0,0,0]     Y'=[0,0,0]
State   4: Y=[0,0,0]     Y'=[0,0,0]
State   5: Y=[0,0,0]     Y'=[0,0,0]
State   6: Y=[0,0,0]     Y'=[0,0,0]
State   7: Y=[0,0,0]     Y'=[0,0,0]
State   8: Y=[7,8,9]     Y'=[7,8,9]
State   9: Y=[4,5,6]     Y'=[4,5,6]
State  10: Y=[11,13,15] Y'=[11,13,15]
State  11: Y=[1,2,3]     Y'=[1,2,3]
State  12: Y=[8,10,12]  Y'=[8,10,12]
State  13: Y=[5,7,9]     Y'=[5,7,9]
State  14: Y=[12,15,18] Y'=[12,15,18]

Done!  Computation length:  14.
```

This confirms that the algorithm works on the given inputs with a throughput delay of $2 \times \mathtt{n} + 1$.

Chapter 3

Interval Temporal Logic

This chapter describes the syntax and semantics of ITL as an embedded theory in higher-order logic. Section 3.1 describes the syntax, section 3.2 describes the semantics of expressions and primitive formulae, and section 3.3 introduces a number of useful derived operators. This chapter is an essential *reference* for the remainder of the dissertation, but it is short on motivation so you might not want to read it right through at one go.

Let us now turn our attention to the mathematical theory underlying Tempura. The theory in question is Interval Temporal Logic (ITL), a system of temporal logic used by Moszkowski in his work on hardware specification [Mos83]. Unlike ordinary first-order logic where the truth of a statement is decided once and for all, a statement of ITL depends on time. But ITL also differs from classical linear-time temporal logic [Pri67] because the truth of every ITL formula is decided relative to an interval of time, rather than just to a point in time; that is, the starting and ending points are both considered. In order to speak about whole computations, rather than about points within computations, it seems that the interval is a more natural starting point than the individual state. In particular, ITL includes the so-called *chop* operator[1] from process logic [HKP82] for composing formulae in sequence. ITL extends first-order logic with a small number of temporal operators, and for most purposes just two of these suffice. One is `next`, a unary operator which intuitively means "after one unit of time ...", the other is the chop operator, written ";", which corresponds to the notion "... and then ...". The other primitive operators, `prefix` and *projection* will be described in subsequent chapters.

The model of behaviour used in ITL is quite natural. The idea is to describe the computation of interest by taking a number of "snapshots" at various points in time, t_i for $i \leq n$ say, and linking these snapshots together to form a "motion picture" which shows the whole behaviour on an interval $\langle t_0, \ldots, t_n \rangle$. Except, of course, it doesn't quite show the whole behaviour. No matter how close together are the sample points $t_0, \ldots, t_n$ a continuous picture will never emerge. It is a basic

[1] It chops an interval into two pieces.

assumption of ITL that an adequate picture can be formed by sampling often enough (but see the discussion of temporal projection in chapter 9).

Shown below are some simple formulae of ITL together with their intended meanings.

Formula	**Meaning**
`N = 5`	`N` has the value `5`
`N := N − 1`	`N` is decremented in one unit of time
`next (empty)`	Terminate in one unit of time
`always (M < N)`	`M` is always less than `N`
`halt (N = 0) ; stable (N)`	Terminate when `N = 0`, then keep `N` constant
`(B := A ; C := B) ⊃ (C ← A)`	Assigning `A` to `B` then `B` to `C` results in `A` being assigned to `C`

As you will see, all of these formulae are defined in terms of the two primitive temporal operators, `next` and chop, together with the usual classical ones.

The remainder of this chapter presents the syntax and semantics of ITL. This can be done in a number of ways. A conventional treatment is given in previous work by Moszkowski [Mos86], but I have chosen here to present ITL as an embedded theory within higher-order logic (HOL). The principal reason for this choice is a practical one, for it provides a direct route to machine-assisted verification using Gordon's HOL proof system [Gor87]. But there are a number of other advantages. Higher-order logic is a mathematical system of equivalent power to set theory, and like set theory can be used to formalise mathematical reasoning. HOL thus provides a framework for unifying ITL with other mathematical theories, such as numbers and lists, which are so necessary for proving properties of real programs. It is also an advantage because it avoids the wasted effort of proving a number of theorems specific to ITL which are in fact just particular instances of more general theorems (temporal induction, for example). The general theorems can just be subsumed into the theory of ITL.

3.1 Syntax

3.1.1 Expressions

Expressions (sometimes called terms) are built inductively from variables, constants and functions.

Variables These are sequences of letters, digits, underscores and primes, beginning with a letter. Some examples are: `X`, `a'`, `In1` and `A_long_name`. Subscripts may be used to denote elements of list-valued variables.

A naming convention separates variables into two classes. Those beginning with a lower case letter, such as `x` and `in1`, are static, whereas those beginning with a capital letter, such as `X` and `In1`, are state variables. These classes will be defined below, but the idea is that state variables change over time and static variables do not. Notwithstanding this convention, the letter v is used in definitions as a meta-variable to range over all variables.

Constants These are fixed values, for example the truth values `true` and `false`, the numbers `0`, `1`, `2`, ..., lists such as $[5, 4, 3, 2, 1]$, and functions such as $+$. Many constants have special syntax; others conform to the syntax of variables.

Function applications These have the form $e(e_1, \ldots, e_n)$, where $0 \leq n$ and e, e_1, ..., e_n are expressions (e should be a function of arity n). Numerous special syntactic forms are permitted, and in particular many binary functions may be used as infix operators. For example, the addition of two numeric expressions may be written $e_1 + e_2$.

Lambda-expressions These have the form $\lambda(v_1, \ldots, v_n) : e$, where $0 \leq n$, $v_1, \ldots, v_n$ are variables and e, the function body, is an expression. Lambda-expressions denote functions. The definition $f(v_1, \ldots, v_n) \stackrel{\text{def}}{=} e$ is another way of writing $f \stackrel{\text{def}}{=} \lambda(v_1, \ldots, v_n) : e$.

3.1.2 Formulae

Formulae are built inductively from predicates and logical connectives. Two temporal connectives, `next` and chop, are defined in addition to the familiar classical ones.

Predicates "the property p is true of expressions $e_1, \ldots, e_n$": $p(e_1, \ldots, e_n)$, where $0 \leq n$ and $e_1, \ldots, e_n$ are expressions. Many familiar predicate symbols, such as the relation $\leq$, have special syntactic forms.

Equality "e and e' are equal": $e = e'$, where e and e' are expressions.

Negation "not p": $\neg p$, where p is a formula.

Conjunction "p and p'": $p \land p'$, where p and p' are formulae.

Existential quantification "there exists a v such that p": $\exists v : p$, where v is a variable (either state or static) and p is a formula.

Next "after one unit of time p holds": `next` p, where p is a formula.

Chop "p followed by p'": $p \,;\, p'$, where p and p' are formulae.

Note that although many interval operators have the same intuitive meanings as their classical counterparts, they are not the same objects. To distinguish the two, I shall use emboldened forms of operators and roman typeface constants, such as $\boldsymbol{\leq}$, $\boldsymbol{+}$, $\boldsymbol{\wedge}$ and 0, to denote the classical versions, whereas the forms, $\leq$, $+$, $\wedge$ and $\mathtt{0}$, will be used to denote the interval representations. This distinction should not interfere with a natural reading of the following material. In fact, you may just ignore the distinction and suppose that operators are overloaded.

3.2 Semantics

The semantics of ITL will be described by providing a mapping from expressions and formulae of ITL to terms of HOL. The HOL translation, denoted by $[\![p]\!]$, of an ITL formula p gives its meaning explicitly as a function from intervals of time to truth values. Similarly, the meaning, $[\![e]\!]$, of an ITL expression e is given as a function from temporal intervals to values of the appropriate type. First, therefore, a brief description of the HOL logic seems in order; a fuller account may be found in [Gor87].

3.2.1 HOL

HOL is based on the typed lambda-calculus. Every term in HOL has an associated type which denotes a set of values that the term itself may denote. I shall usually write $t \in ty$ to mean that the term t is of type ty, but may omit type information when it can be deduced from the situation.[2] For example, $t \in \mathbb{N}$ means that the term t denotes a member of the set of natural numbers, $\{0, 1, 2, \ldots\}$.

HOL terms are built inductively from variables (ranged over by v, possibly subscripted), constants (such as 0, T and the addition function $+$), lambda-terms and function applications.

A lambda-term of the form $\lambda(v_1, \ldots, v_n) : t$, where $v_i \in ty_i$ for each $i \boldsymbol{\leq} n$ are variables and $t \in ty$ is a term, denotes a function of type $ty_1 \times \cdots \times ty_n \to ty$. For example, the term $\lambda(x, y) : x + y$ denotes the addition function.

A function application of the form $t(t_1, \ldots, t_n)$, where t, $t_1, \ldots, t_n$ are terms, denotes the meaning of t applied to the meanings of its arguments. Therefore, if $t_1 \in ty_1, \ldots, t_n \in ty_n$ then t must denote a function of the form $ty_1 \times \cdots \times ty_n \to ty$, and the whole term is of type ty. Only well-typed terms are considered meaningful. For example: $(\lambda(x, y) : x + y)(1, 2)$ denotes the constant 3.

All the usual connectives of first-order logic are predefined in HOL. In other words, if $t, t' \in \mathbb{B}$ are formulae (boolean terms) of HOL and v is a variable, then the

[2] I am being imprecise here in order (hopefully) to make terms easier to read. In the HOL logic types are part of the *syntax* of terms, so $t : ty$ is used to denote a term t of type ty.

following are also formulae (recall that the emboldened forms of the connectives are used to distinguish the HOL connectives from those of ITL):

Equality : $t \mathbf{=} t'$

Negation : $\mathbf{\neg}\, t$.

Conjunction : $t \mathbf{\wedge} t'$.

Disjunction : $t \mathbf{\vee} t'$.

Implication : $t \mathbf{\supset} t'$.

Existential quantification : $\mathbf{\exists}\, v : t$.

Universal quantification : $\mathbf{\forall}\, v : t$.

Many standard predicates, such as $\mathbf{\leq}$, will also be assumed.

3.2.2 Intervals

The meaning of an expression or formula of ITL is only defined relative to an interval of time. Consequently, the semantics of expressions and formulae may be represented in HOL as functions of intervals. For instance, a formula is represented by a function from intervals to booleans.

Let us introduce a new type $\mathbb{I}$ to denote the set of all intervals. An interval τ is a non-empty sequence of time points, such as $\langle \tau_0, \ldots, \tau_n \rangle$, where $\tau_i \in \mathbb{N}$ for each $i \mathbf{\leq} n$, and $\mathbb{I}$ therefore denotes the set of all such sequences. The length of an interval is the number of *time steps*, not the number of *time points*,

$$|\langle \tau_0, \ldots, \tau_n \rangle| \mathbf{=} n,$$

so a zero length interval contains one time point. Note that an interval may be of infinite length, denoted by an infinite sequence beginning at time τ_0.

The temporal operators `next` and chop are concerned with the structure of intervals, and they may be defined in terms of two functions: *prefix* and *suffix*. The function $\mathit{prefix}(i, \tau)$ denotes the i-th prefix of the interval τ, which is the subsequence from element 0 up to and including element i; and the function $\mathit{suffix}(i, \tau)$ denotes the i-th suffix of τ which is the subsequence from element i onwards. If $\tau \in \mathbb{I}$ and $i \leq |\tau|$, then

$$\begin{aligned} \mathit{prefix}(i, \tau) &= \langle \tau_0, \ldots, \tau_i \rangle, \\ \mathit{suffix}(i, \tau) &= \langle \tau_i, \ldots, \tau_{|\tau|} \rangle. \end{aligned}$$

For example:

$$\begin{aligned} \mathit{prefix}(3, \langle 0, 1, 2, 3, 4, 5 \rangle) &= \langle 0, 1, 2, 3 \rangle \\ \mathit{suffix}(3, \langle 0, 1, 2, 3, 4, 5 \rangle) &= \langle 3, 4, 5 \rangle. \end{aligned}$$

For most purposes you may assume that the time points in an interval form a contiguous increasing sequence of values, so that $\tau_{i+1} = \tau_i + 1$. However, this need not be so when prefix and projection are used (see chapters 6 and 9).

We are now in a position to describe the semantic function *sem* which gives the semantic representation in HOL of each expression and formula in ITL.

3.2.3 Expressions

Expressions involve variables, constants and functions. The HOL translation, $[\![e]\!]$, of an ITL expression e is a function of the form $\mathbb{I} \to ty$ for some type ty. For example, the translation of a numeric expression has type $\mathbb{I} \to \mathbb{N}$, that of a string expression has type $\mathbb{I} \to \mathbb{S}$ (where $\mathbb{S}$ denotes the set of strings) and that of a list expression has type $\mathbb{I} \to ty\ list$ (where $ty\ list$ denotes the set of lists of elements taken from the set denoted by ty).

Variables : There are two kinds of temporal variables: *state* and *static*. State variables are conventionally represented by names starting with an upper case letter, whereas names starting with lower case letters represent static variables.

State variables depend on time (but not on intervals). The meaning of a state variable v on an interval is given by taking its value on the *first* state of the interval; that is,

$$[\![v]\!] \quad = \quad mk_state(\hat{v}),$$

where v is a state variable and

$$mk_state(\hat{v}) \quad \stackrel{\text{def}}{=} \quad \lambda\tau : \hat{v}(\tau_0) \quad \text{where} \quad \tau \in \mathbb{I} \text{ and } \hat{v} \in \mathbb{N} \to ty.^3$$

For example, $[\![\mathtt{X}]\!] = \lambda\tau : \hat{\mathtt{X}}(\tau_0)$.

Static variables stand for constant values. The meaning of a static variable v is simply a function from intervals to its value, $\hat{v}$; that is,

$$[\![v]\!] \quad = \quad mk_static(\hat{v}),$$

where v is a static variable and

$$mk_static(\hat{v}) \quad \stackrel{\text{def}}{=} \quad \lambda\tau : \hat{v} \quad \text{where} \quad \tau \in \mathbb{I} \text{ and } \hat{v} \in ty.$$

For example, $[\![\mathtt{x}]\!] = \lambda\tau : \hat{\mathtt{x}}$.

Constants : Constants are treated in the same way as static variables. For each constant c there is a corresponding constant $\hat{c}$ in HOL such that

$$[\![c]\!] \quad = \quad mk_static(\hat{c}).$$

For instance, $[\![\mathtt{0}]\!] = mk_static(0)$, the function that evaluates to 0 on any interval.

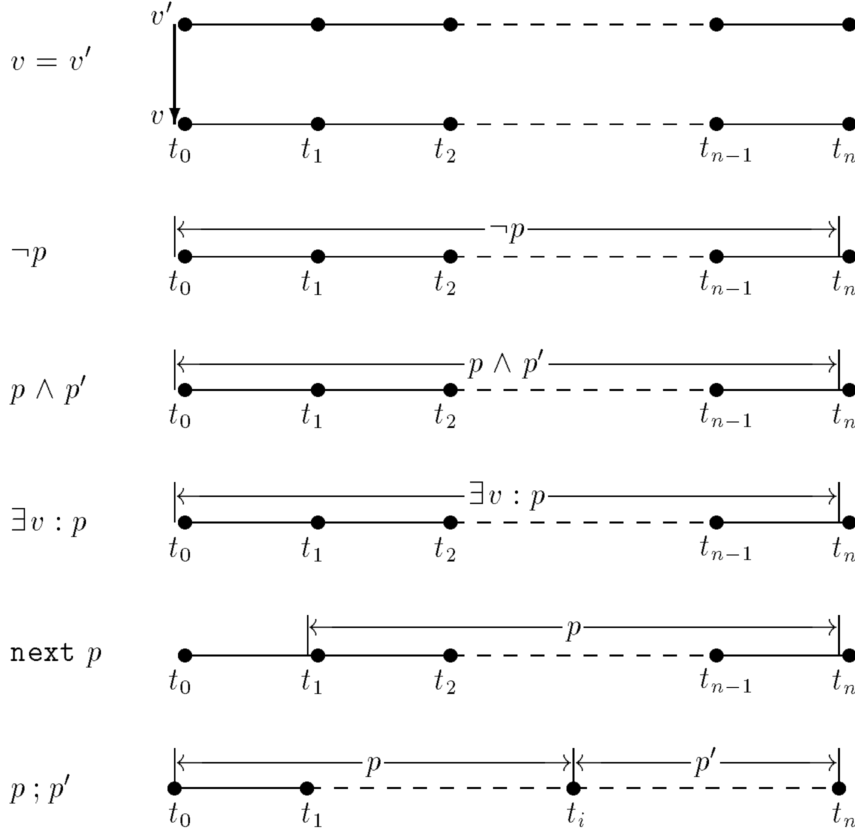


Figure 3.1: The primitive operators.

Function application : The meaning of a function application is the meaning of the function applied to the meanings of its arguments; that is, if $e_1, \ldots, e_n$ are expressions

$$[\![f(e_1, \ldots, e_n)]\!] \quad = \quad \lambda\tau : [\![f]\!](\tau)([\![e_1]\!](\tau), \ldots, [\![e_n]\!](\tau)) \quad \text{where} \quad \tau \in \mathbb{I}.$$

Only static functions appear in the following, so $[\![f]\!](\tau) = \hat{f}$ for some function $\hat{f}$ of type $ty_1 \times \cdots \times ty_n \rightarrow ty$. For example, $[\![\mathtt{X} + \mathtt{Y}]\!] = \lambda\tau : \hat{\mathtt{X}}(\tau_0) \mathbin{\boldsymbol{+}} \hat{\mathtt{Y}}(\tau_0)$.

3.2.4 Formulae

The meaning, $[\![p]\!]$, of an ITL formula p is given by a HOL term of type $\mathbb{I} \rightarrow \mathbb{B}$, where $\mathbb{B}$ denotes the set of (classical) truth values $\{\mathsf{T}, \mathsf{F}\}$. There are six primitive formulae out of which more complex formulae and predicates may be built inductively.

The semantics of the elementary ITL formulae are defined below. You may find that the pictures in figure 3.1 help to explain the meanings of the operators.

Predicates The meaning of a predicate is given by the meaning of the predicate symbol applied to the meanings of its arguments. If $e_1, \ldots, e_n$ are expressions, then

$$[\![p(e_1, \ldots, e_n)]\!] \quad = \quad \lambda\tau : [\![p]\!](\tau)([\![e_1]\!](\tau), \ldots, [\![e_n]\!](\tau)) \quad \text{where} \quad \tau \in \mathbb{I}.$$

Predicate symbols, like function symbols, are generally static, so $[\![p]\!](\tau) = \hat{p}$ for some function $\hat{p}$ of type $ty_1 \times \cdots \times ty_n \to \mathbb{B}$. For example, $[\![\mathtt{X} \leq \mathtt{Y}]\!] = \lambda\tau : \hat{\mathtt{X}}(\tau_0) \leq \hat{\mathtt{Y}}(\tau_0)$.

Equality Two expressions e and e' are equal if their values are equal.

$$[\![e = e']\!] \quad = \quad \lambda\tau : [\![e]\!](\tau) = [\![e']\!](\tau) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example, two state variables are equal on an interval τ if they have the same value on the initial state,

$$[\![\mathtt{X} = \mathtt{Y}]\!] \quad = \quad \lambda\tau : \hat{\mathtt{X}}(\tau_0) = \hat{\mathtt{Y}}(\tau_0).$$

Negation The negation of a formula p is true on an interval if p evaluates to false.

$$[\![\neg p]\!] \quad = \quad \lambda\tau : \neg\, [\![p]\!](\tau) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example:

$$[\![\neg(\mathtt{X} = \mathtt{1})]\!] \quad = \quad \lambda\tau : \hat{\mathtt{X}}(\tau_0) \neq 1.$$

Conjunction The conjunction of two formulae p and p' is true on an interval if both of them evaluate to true.

$$[\![p \wedge p']\!] \quad = \quad \lambda\tau : ([\![p]\!](\tau) \wedge [\![p']\!](\tau)) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example:

$$[\![(\mathtt{X} = \mathtt{1}) \wedge (\mathtt{Y} = \mathtt{1})]\!] \quad = \quad \lambda\tau : (\hat{\mathtt{X}}(\tau_0) = 1) \wedge (\hat{\mathtt{Y}}(\tau_0) = 1).$$

Existential quantification The formula $\exists v : p$ is true on an interval if there is a v that makes p evaluate to true. For a variable v and formula p

$$[\![\exists v : p]\!] \quad = \quad \lambda\tau : \exists\, \hat{v} : [\![p]\!](\tau) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example:

$$[\![\exists \mathtt{X} : \mathtt{X} = \mathtt{1}]\!] \quad = \quad \lambda\tau : \exists\, \hat{\mathtt{X}} : \hat{\mathtt{X}}(\tau_0) = 1.$$

Next The formula $\texttt{next}\ p$ is true on an interval if p evaluates to true on the tail of the interval (which must have length greater than zero).

$$[\![\texttt{next}\ p]\!] \quad = \quad \lambda\tau : (|\tau| > 0 \wedge [\![p]\!](\mathit{suffix}(1, \tau))) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example, the formula $\texttt{next}\,(\texttt{X} = \texttt{1})$ asserts that $\texttt{X}$ has the value $\texttt{1}$ after one step,

$$[\![\texttt{next}\,(\texttt{X} = \texttt{1})]\!] \quad = \quad \lambda\tau : (|\tau| > 0 \wedge \hat{\texttt{X}}(\tau_1) = 1).$$

Chop The sequential composition of two formulae p and p' is true on an interval if there is a way to divide the interval into two pieces such that p is true on the first and p' is true on the second.

$$[\![p\,;p']\!] \quad = \quad \lambda\tau : \exists\, i : (i \leq |\tau| \wedge [\![p]\!](\mathit{prefix}(i, \tau)) \wedge [\![p']\!](\mathit{suffix}(i, \tau))) \quad \text{where} \quad \tau \in \mathbb{I}.$$

For example, the formula $(\texttt{X} = \texttt{0})\,;(\texttt{X} = \texttt{1})$ asserts that $\texttt{X} = \texttt{0}$ and sometime later $\texttt{X} = \texttt{1}$,

$$[\![(\texttt{X} = \texttt{0})\,;(\texttt{X} = \texttt{1})]\!] \quad = \quad \lambda\tau : \exists\, i : (i \leq |\tau| \wedge \hat{\texttt{X}}(\tau_0) = 0 \wedge \hat{\texttt{X}}(\tau_i) = 1).$$

Note that the $\texttt{next}$ operator requires the interval to be of non-zero length (so that it has a tail), and that the two parts of a chopped interval have a state in common.

3.3 Some Derived Operators

Many useful operations can be defined in terms of the basic ones. They can be defined as predicates, but the most useful ones are predefined, often with special syntax.

3.3.1 Classical Operators

All the usual logical connectives (disjunction, implication, *etc.*), quantifiers and truth values can be defined in interval form. Their meanings are the same as if they had been lifted from the standard theory of booleans. Some examples are:

Truth values The constants $\texttt{true}$ and $\texttt{false}$ are the interval forms of the classical truth values T and F.

$$\texttt{false} \stackrel{\text{def}}{=} p \wedge \neg p \quad \text{for any formula } p,$$
$$\texttt{true} \stackrel{\text{def}}{=} \neg\texttt{false}.$$

Disjunction The disjunction of two formulae is true if they are not both false.

$$p \vee p' \stackrel{\text{def}}{=} \neg(\neg p \wedge \neg p').$$

Implication The implication $p \supset p'$ is true if p' is a consequence of p. An alternative notation is the familiar programming construct `if` b `then` p, which is used to test the value of a boolean expression b. Two formulae are logically equivalent, $p \equiv p'$, if each implies the other.

$$\begin{aligned} p \supset p' &\stackrel{\text{def}}{=} \neg p \vee p' \\ p \equiv p' &\stackrel{\text{def}}{=} (p \supset p') \wedge (p' \supset p) \\ \texttt{if}\ b\ \texttt{then}\ p &\stackrel{\text{def}}{=} b \supset p \\ \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p' &\stackrel{\text{def}}{=} (b \supset p) \wedge (\neg b \supset p') \end{aligned}$$

Universal quantification The universal quantification $\forall v : p$ is true if there is no v that makes p false.

$$\forall v : p \stackrel{\text{def}}{=} \neg \exists v : \neg p$$

Bounded forms of both existential and universal quantifiers are also used as abbreviations:

$$\begin{aligned} \exists i < n : p &\Rightarrow \exists i : (i < n \wedge p) \\ \forall i < n : p &\Rightarrow \forall i : (i < n \supset p). \end{aligned}$$

Bounded universal quantification will usually be written `forall` $i < n : p$. For example, the formula $\texttt{forall i} < \texttt{n} : \texttt{A}_\texttt{i} = \texttt{0}$ denotes that every element of the list `A` is initially zero.

Cascades of quantifiers are abbreviated in the usual way: $\exists v_1, v_2, \ldots, v_n : p$ is shorthand for $\exists v_1 : \exists v_2 : \ldots \exists v_n : p$, and similarly for universal quantifiers.

3.3.2 Temporal Operators

A number of useful operators can be defined in terms of the operators `next` and chop. Their definitions are illustrated in figures 3.2–3.4.

Modal operators The operators `always` and `sometime` are characteristic of temporal logics, and are often written $\Box$ and $\Diamond$, respectively. The formula `sometime` p holds if the interval can be divided into two parts so that p holds on the second; that is, if p holds on some suffix subinterval. The dual construct, `always` p, holds if the formula p holds on all suffix subintervals; in other words there is no such subinterval on which p does not hold. These operators are illustrated in figure 3.2.

$$\begin{aligned} \texttt{sometime}\ p &\stackrel{\text{def}}{=} \texttt{true}\,;p, \\ \texttt{always}\ p &\stackrel{\text{def}}{=} \neg\, \texttt{sometime}\ \neg p. \end{aligned}$$

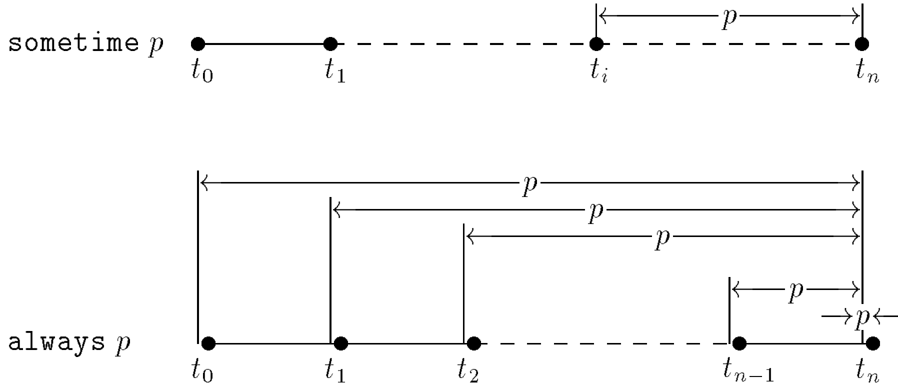


Figure 3.2: The modal operators `sometime` and `always`.

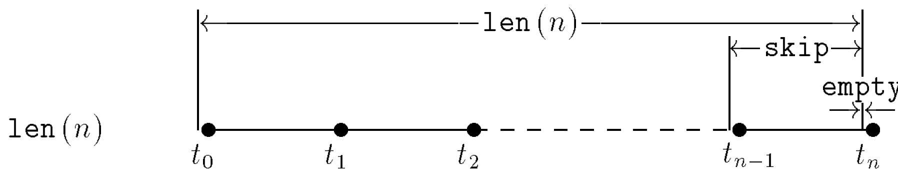


Figure 3.3: The length operators `len`, `skip` and `empty`.

For example, the formula `always` $(\mathtt{X} + \mathtt{Y} = \mathtt{1})$ tests whether the sum of `X` and `Y` is `1` on every state of the interval. It is also useful to divide the interval in other ways. For instance, the construct `extend` p holds if p is true on some prefix subinterval. It is defined as p ; `true` in the same way as `sometime`.

Computation length The formula `len` (n) is true on an interval of length n. It can be used to define any interval length, but intervals of length zero and one occur so frequently that special operators `empty` and `skip` are used to denote these values. An interval is empty if there is no next subinterval, and is of length one if the next subinterval is empty. Figure 3.3 illustrates these definitions.

$$\begin{aligned} \mathtt{empty} &\stackrel{\text{def}}{=} \neg\,\mathtt{next}\ \mathtt{true}, \\ \mathtt{skip} &\stackrel{\text{def}}{=} \mathtt{next}\ \mathtt{empty}. \end{aligned}$$

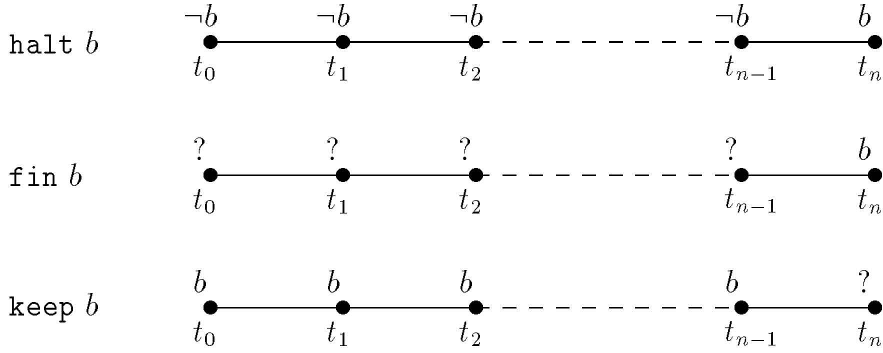


Figure 3.4: The operators `halt`, `fin` and `keep` for a boolean *expression* b. The expression b is true on the states marked b, false on those marked $\neg b$, and may be either true or false elsewhere.

The operator `len` itself is defined using primitive recursion.

$$\begin{aligned} \texttt{len}(0) &= \texttt{empty}, \\ \texttt{len}(n+1) &= \texttt{next len}(n), \end{aligned}$$

where $n \in \mathbb{I} \to \mathbb{N}$. This definition is not in classical primitive recursive form because the argument of `len` is actually an interval function, not a natural number. However, the fault is easily rectified by defining a primitive recursive predicate $len \in \mathbb{N} \to (\mathbb{I} \to \mathbb{B})$ (in just the same way as above), and then taking $\texttt{len}(n) = \lambda\tau : len(n(\tau))$. Since this can be done automatically, let us just accept ITL definitions in the form above as primitive recursive in this sense.

Termination and the final state The formula `halt` p is true if p is true on, and only on, the final state. In other words the computation terminates as soon as the formula p becomes true. Related operators are `fin` and `keep`. The formula `fin` p is true if p is true on the final state, and `keep` p is true if p is true on every state except the last. Thus $\texttt{halt}\ p \equiv (\texttt{keep}\ \neg p) \wedge (\texttt{fin}\ p)$.

$$\begin{aligned} \texttt{halt}\ p &\stackrel{\text{def}}{=} \texttt{always}(\texttt{empty} \equiv p), \\ \texttt{fin}\ p &\stackrel{\text{def}}{=} \texttt{always}(\texttt{empty} \supset p), \\ \texttt{keep}\ p &\stackrel{\text{def}}{=} \texttt{always}(\neg\texttt{empty} \supset p). \end{aligned}$$

For example, $\texttt{halt}(\texttt{A} < 1)$ asserts that the computation terminates as soon as $\texttt{A} < 1$. Figure 3.4 shows how these definitions work out for a boolean expression b, though the definitions apply more generally to arbitrary formulae.

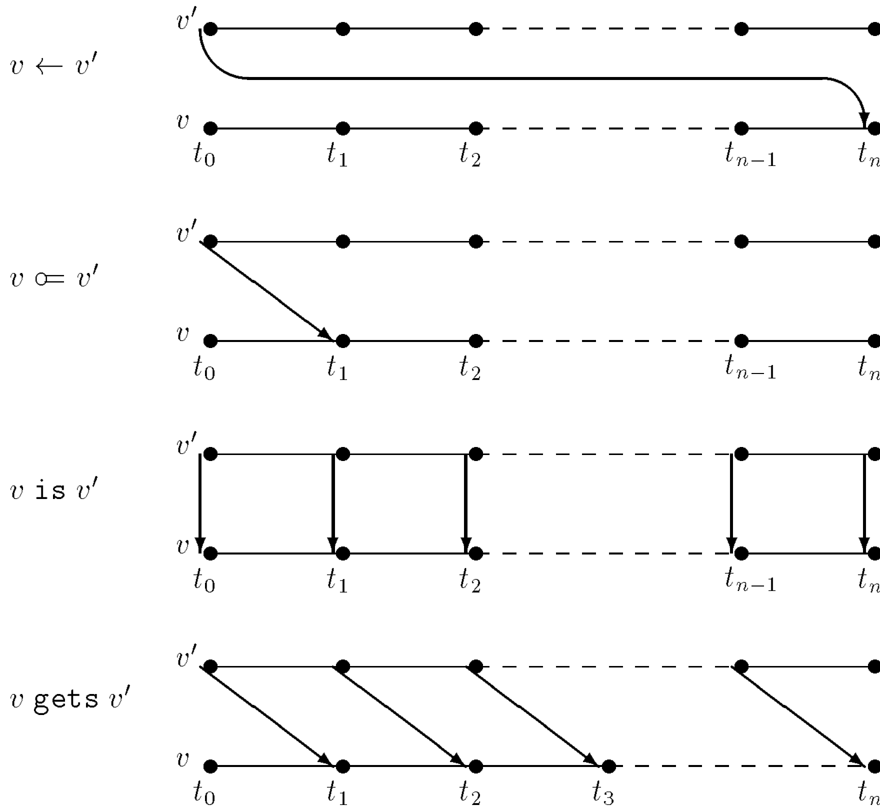


Figure 3.5: The assignment operations $v \leftarrow v'$, $v \mathrel{\circ\!\!=} v'$, v `is` v' and v `gets` v' for state variables v and v'.

It is also possible to define an operator to look at the initial state

$$\texttt{init}\ p \stackrel{\text{def}}{=} (\texttt{empty} \wedge p)\,;\texttt{true},$$

but because of the conventional interpretation of state variables, it is hardly ever needed unless one really needs to ensure that p only refers to the initial state.

3.3.3 Assignment Operators

A number of assignment operators can be defined in ITL. In these definitions e and e' stand for expressions of arbitrary types: $e \in \mathbb{I} \rightarrow ty$, $e' \in \mathbb{I} \rightarrow ty'$. Figure 3.5 shows what these operators mean for state variables v and v'.

Temporal Assignment Three forms of assignment in time have been found useful. The simplest, equality, is true if two values are equal at the same point in time. It has already been defined. The next-assignment $\texttt{A}' \mathrel{\circ\!\!=} \texttt{A}$ is true if the next

value of $\mathtt{A}'$ equals the current value of $\mathtt{A}$; and the temporal assignment $\mathtt{A}' \leftarrow \mathtt{A}$ holds if the final value of $\mathtt{A}'$ equals the initial value of $\mathtt{A}$.

$$\begin{aligned} e \leftarrow e' &\stackrel{\text{def}}{=} \exists v : (v = e' \wedge \mathtt{fin}\,(e = v)), \\ e \coloneq e' &\stackrel{\text{def}}{=} \exists v : (v = e' \wedge \mathtt{next}\,(e = v)), \end{aligned}$$

where v is a static variable which is not free in either e or e'. Temporal assignment is most useful for specifying the functional behaviour of a program. For instance, $\mathtt{A} \leftarrow \mathtt{sort(A)}$ might be the specification of a program to sort the list $\mathtt{A}$.

In practice, assignment is most commonly used on a unit- or zero-length interval, and two special operators have been defined to incorporate the length into the assignment. The assignment $\mathtt{A}' \Leftarrow \mathtt{A}$ ensures that $\mathtt{A}$ and $\mathtt{A}'$ are equal on a zero-length interval, whereas the unit-assignment $\mathtt{A}' := \mathtt{A}$ asserts that the next value of $\mathtt{A}'$ is the same as the initial value of $\mathtt{A}$ on a unit-interval.[4] Unit-assignment is similar to ordinary assignment in an imperative programming language.

$$\begin{aligned} e \Leftarrow e' &\stackrel{\text{def}}{=} \mathtt{empty} \wedge e = e', \\ e := e' &\stackrel{\text{def}}{=} \mathtt{skip} \wedge e \coloneq e'. \end{aligned}$$

For example, if a variable is initially zero and is incremented by one then its final value is one, $(\mathtt{A} \Leftarrow \mathtt{0}\ ;\ \mathtt{A} := \mathtt{A} + \mathtt{1}) \supset (\mathtt{A} \leftarrow \mathtt{1})$. Note that the formula $\mathtt{skip} \wedge \mathtt{A}' \leftarrow \mathtt{A}$ is equivalent to $\mathtt{A}' := \mathtt{A}$, and that $\mathtt{empty} \wedge \mathtt{A}' \leftarrow \mathtt{A}$ is the same as $\mathtt{A}' \Leftarrow \mathtt{A}$.

Repeated assignment The formula $\mathtt{A}'\ \mathtt{is}\ \mathtt{A}$ denotes that the variables $\mathtt{A}$ and $\mathtt{A}'$ are equal throughout time; and the formula $\mathtt{A}'\ \mathtt{gets}\ \mathtt{A}$ is true if $\mathtt{A}$ is always assigned to $\mathtt{A}'$ from one state to the next. If something is always assigned to itself from one state to the next it remains stable, denoted by $\mathtt{stable}\ \mathtt{A}$.

$$\begin{aligned} e\ \mathtt{is}\ e' &\stackrel{\text{def}}{=} \mathtt{always}\,(e = e'), \\ e\ \mathtt{gets}\ e' &\stackrel{\text{def}}{=} \mathtt{keep}\,(e \coloneq e'), \\ \mathtt{stable}\ e &\stackrel{\text{def}}{=} e\ \mathtt{gets}\ e. \end{aligned}$$

For example, the formula $\mathtt{A} = \mathtt{n} \wedge \mathtt{A}\ \mathtt{gets}\ \mathtt{A} - \mathtt{1}$ asserts that $\mathtt{A}$ is initially $\mathtt{n}$ and is decremented from state to state. Note that one cannot use next-assignment on an empty subinterval, hence the use of `keep` rather than `always` in the definition of `gets`.

---

[4]Note that Moszkowski defines the unit-assignment operator := to be what I have called next-assignment $\coloneq$, *i.e.* without the associated `skip` [Mos86].

Multiple parallel assignments, including equality, may be abbreviated in the form:

$$e_0, \ldots, e_n \; op \; e'_0, \ldots, e'_n,$$

where e_i and e'_i are arbitrary expressions and op is an assignment operator. This form is equivalent to

$$e_0 \; op \; e'_0 \wedge \ldots \wedge e_n \; op \; e'_n.$$

The operator `stable` with multiple arguments keeps each of its arguments stable.

3.3.4 Iterative Operators

A number of ITL operators resemble those of ordinary imperative programming languages. Some, such as assignment and chop, have already been encountered, but another important class is that containing the iterative operators, such as the for- and while-loops.

For-loops A particularly simple form of loop is $\texttt{for}\; n \;\texttt{times do}\; p$ which just denotes n iterations of p; that is, $p ; \ldots ; p$ (n times). It is, of course, defined recursively,

$$\begin{aligned} \texttt{for 0 times do}\; p &= \texttt{empty}, \\ \texttt{for}\; n+1 \;\texttt{times do}\; p &= \texttt{for}\; n \;\texttt{times do}\; p \,;\, p, \end{aligned}$$

where $n \in \mathbb{I} \rightarrow \mathbb{N}$. Loops with control variables can also be defined.

$$\begin{aligned} \texttt{for}\; i < 0 \;\texttt{do}\; p &= \texttt{empty}, \\ \texttt{for}\; i < n+1 \;\texttt{do}\; p &= \texttt{for}\; n \;\texttt{times do}\; p \,;\, p[n/i]. \end{aligned}$$

where $n \in \mathbb{I} \rightarrow \mathbb{N}$ and i is static, and the formula $p[n/i]$ denotes p with all free occurrences of i replaced by n. If n is bound at any point in p where there is a free i then the variables of p are "systematically renamed" to remove the conflict.

While-loops The loop $\texttt{while}\; b \;\texttt{do}\; p$ denotes a number of iterations of p on each of which the boolean expression b is initially true. At the end of the last iteration b is false.

$$\texttt{while}\; b \;\texttt{do}\; p \;\stackrel{\text{def}}{=}\; \exists n : (\texttt{fin}\,(\neg b) \wedge \texttt{for}\; n \;\texttt{times do}\; (b \wedge p)),$$

where $n \in \mathbb{I} \rightarrow \mathbb{N}$ and $b \in \mathbb{I} \rightarrow \mathbb{B}$. For example, $\texttt{while A} \neq \texttt{0 do A} := \texttt{A} - \texttt{1}$ asserts that `A` is decremented until it reaches zero.

Finally, a repeat-loop is defined in terms of the while-loop in the usual way, and the straightforward $\texttt{loop}\; p$ denotes an arbitrary number of repetitions of p.

$$\begin{aligned} \texttt{repeat}\; p \;\texttt{until}\; b &\stackrel{\text{def}}{=} p \,;\, \texttt{while}\; \neg b \;\texttt{do}\; p, \\ \texttt{loop}\; p &\stackrel{\text{def}}{=} \texttt{while}\; \neg\texttt{empty do}\; p. \end{aligned}$$

Other loops can be defined in a similar way. For example, in the loop $\texttt{for}\ v \in l\ \texttt{do}\ p$ the control variable v takes successive values from the list l.

3.3.5 Markers

A number of ITL operators are defined using *marker* variables to mark the occurrence of some event. For instance, the definition of the `until` operator below uses a marker μ to watch for p' becoming true. The formula $p\,\texttt{until}\ p'$ is true if p holds until p' is true.

$$p\,\texttt{until}\ p' \stackrel{\text{def}}{=} \exists\mu : \{\mu \wedge \texttt{always}\{\mu \supset (p' \vee (p \wedge \texttt{next}\,(\mu)))\}\}.$$

Marker variables are really just ordinary state variables, of course, but they must not occur freely in the program they are "marking". For instance, μ must not be free in either p or p' in the definition of `until`. Conventionally, I shall use greek letters for markers.

3.3.6 Omitting Parentheses

With just the above definitions, parentheses are needed to delimit the arguments of each operator. Even a relatively modest formula can need a large number of parentheses, and this detracts from its readability. Therefore, it seems a good idea to adopt some conventions to eliminate parentheses. The following precedence hierarchy is used; the operators at level i apply to as little as possible without violating the constraints placed on those at levels less than i.

1. The negation operator, $\neg$.
2. The assignment operators $=$, `is`, $\leftarrow$, $\Leftarrow$, $:=$, $\Leftarrow$ and `gets`.
3. The unary operators `next`, `always`, `sometime`, `keep`, `halt` and `stable`, the quantifiers $\exists$ and $\forall$, the final branch of the conditional (`then` or `else`), and the body of a loop.
4. The conjunction and disjunction operators, $\wedge$ and $\vee$.
5. The implication operator, $\supset$.
6. The chop operator, ;.

For example, the formula

$$\exists v : \texttt{next}\ p \wedge \neg p' \,;\, p''$$

should be read

$$((\exists v : (\texttt{next}\ p)) \wedge (\neg p'))\,;\,(p'').$$

The fact that conjunction, disjunction and chop are all associative operators may also be used to avoid writing parentheses, since the form $p\ op\ p'\ op\ p''$ is unambiguous for these operators.

The usual precedence relations apply to expressions, for instance $\mathtt{A}+\mathtt{B}\times\mathtt{C}$ denotes $\mathtt{A}+(\mathtt{B}\times\mathtt{C})$. Parentheses may also be omitted when a formula only makes sense with a particular grouping, for example `A gets B + C` is only well typed when read as `A gets (B + C)`.

3.4 Discussion

Given that ITL is easily expressed as an embedded theory within HOL, the question naturally arises: Why is ITL needed at all? Why not, for instance, simply represent every variable as a function of time? Aside from the obvious but important point that ITL avoids the consequent proliferation of time variables, I think that the answer to this question is much the same as the answer to the question: Why bother with high-level programming languages when all problems can be tackled in machine code? Both questions have the same answer. The higher-level language is useful precisely because it addresses a restricted, but useful, class of problems. It is sufficient to express any problem in its domain, whilst keeping the complexity of expression to a minimum. Hopefully, this leads to a richer set of laws for manipulating specifications.

One is also entitled to question the assumed model of time. Is it reasonable to suppose that real computing systems can be described within a discrete-time framework using shared variables? Although it may be a fairly good approximation to the way synchronous parallel hardware works, many concurrent systems do not operate in step with some global clock. Nevertheless, so long as the behaviour of interest can accurately be reflected by a discrete sequence of states (possibly with repetitions), time points may simply be identified with these states. Techniques for describing explicit synchronisation will be introduced in chapters 10.

If the model is adequate, is it abstract enough? Use of the operator `next` has been criticised for forcing too much irrelevant detail into specifications [Lam83]. The main reason for this criticism, it seems, is that this operator permits (not compels) one to distinguish between repetitions of the same state or between computations of different lengths. For example, one might say that the behaviours

```
X := 0 ; X := 1
```

and

```
stable(X) ; X := 0 ; stable(X) ; X := 1
```

should be indistinguishable in a specification language. But, of course, the second formula is simply a specification of the first (*i.e.* the first logically implies the second),

and the very purpose of this work is to express specifications and programs in the same framework.

Other researchers use a dense model of time (one in which any two points are separated by another) to get abstract specifications [BKP86]. Temporal projection, which is discussed in chapter 9, produces a similar effect in ITL. For example, projecting the formula `stable(X);skip` onto the first of the formulae above results in the second.

Chapter 4

Tempura

This chapter clarifies the relationship between Tempura and ITL. Section 4.1 defines the syntax of the primitive language; section 4.2 sketches its computational semantics; and section 4.3 discusses which derived operators of ITL are executable. The main point of this chapter is the reduction mechanism presented in section 4.2.2.

The programming language Tempura was introduced informally in chapter 2. Tempura is not the same as ITL, but is an executable sublanguage of it. The principal restriction is that Tempura programs must be deterministic. This means that no arbitrary choices (either of computation length or variable assignment) can be made during execution, so it will be appreciated that many specifications are not executable. For example, neither the formula $\neg\texttt{skip}$ nor the formula $(\texttt{I} = \texttt{0}) \vee (\texttt{I} = \texttt{1})$ is executable, as both are non-deterministic. The former describes any interval of length other than one, the latter gives a choice of values for the variable `I`.

In order to exclude formulae such as the two above, the syntax of Tempura is restricted. The negation of a program is not syntactically permitted, and in its place the conditional and the termination statement, `empty`, are taken as primitive. This means that some of the derived operators of ITL cannot be defined at all in Tempura, and that some others can only be defined in a restricted form.

This chapter does not describe in detail how to execute Tempura programs; Moszkowski does that admirably in his book on the subject [Mos86]. Its purpose is rather to clarify the relationship between Tempura and ITL, and to outline the computational semantics in a way that is not constrained by the details of a particular implementation.

Only the basic language is described. This language contains seven elementary operators — equality, conjunction, conditional, existential quantification, termination, next and chop — and the operators that can be defined in terms of these, The prefix and temporal projection operators are not considered here (see [Mos86]).

4.1 Syntax

This section describes the syntax of the basic language: first the primitive operators, then the expressions.

4.1.1 Programs

Programs are formed from seven elementary operators, five of those used to define formulae (all except negation) and two which are derived operators of ITL. As usual, the symbol e stands for an arbitrary expression, b stands for a boolean expression and p and p' stand for programs. The symbol l stands for an "L-expression", which may be a simple variable (v), a suscripted variable (v_i or $v_{i..j}$) or the size of a list-valued variable ($|v|$). These values correspond to regions of computer memory when a program is executed.

Equality: $l = e$.

Parallel composition: $p \wedge p'$.

Conditional: `if` b `then` p `else` p'.

Local variables: $\exists v : p$.

Termination: `empty`.

Next: `next` p.

Sequential composition: $p\ ;\ p'$.

The absence of the negation operator is explained by the fact that a negated program would, in general, be non-deterministic. This has a number of repercussions. It means that the conditional and `empty` must be taken as primitive, since they are defined (logically) in terms of negation. It also means that certain other operators cannot be derived in the way they were before. For example, universal quantification ($\forall$) and arbitrary choice ($\vee$) cannot be defined in full generality, though a restricted form of universal quantification can be defined, as will be shown.

4.1.2 Expressions

Variables in Tempura have the same syntax as in ITL (characters, underscores and primes) with the capitalised names for state variables and lower case for static variables.

As before, the choice of permissible expressions is largely arbitrary, but only boolean, integer, string and list expressions are used in the following.

Booleans are just the truth values `true` and `false`. They may be combined with all the usual logical connectives.

Integers are the numbers $\ldots, -2, -1, 0, 1, 2, \ldots$. They may be combined with all the usual arithmetic operators.

Lists are sequences of elements separated by commas and enclosed in square brackets, such as $[1, 3, 5, 7]$. The notation $[\mathtt{i}..\mathtt{j}]$ denotes the list of numbers from `i` to $\mathtt{j} - 1$ inclusive, so $[0..4] = [0, 1, 2, 3]$. The number of elements in a list `A` is denoted by $|\mathtt{A}|$, the `i`th element by $\mathtt{A}_\mathtt{i}$, the sublist from element `i` to element $\mathtt{j} - 1$ by $\mathtt{A}_{\mathtt{i}..\mathtt{j}}$, and the concatenation of two lists `A` and $\mathtt{A}'$ by $\mathtt{A}\hat{}\mathtt{A}'$. No distinction is made between lists and arrays in Tempura; arrays, or rather vectors, are just special kinds of lists, and their elements are directly accessible.

Strings are surrounded by double quotation marks, like "`a string`". They may be indexed and manipulated in the same way as lists.

Tempura could be extended to include real numbers, bit vectors, and so on.

4.2 Semantics

The aim of executing a Tempura program is to discover an interval on which the program viewed as an ITL formula is true. In other words, for a program p with free variables $v_1, \ldots, v_n$, the goal is to show that the formula

$$\exists\, \tau : (\exists v_1, \ldots, v_n : p)(\tau)$$

is true by finding a particular interval τ on which the program holds. This section outlines a technique for generating such an interval from a program. The idea is to reduce the program to an equivalent canonical form from which the desired interval is immediately apparent.

4.2.1 Canonical Form

The canonical form of a program p may be represented as a conjunction of "state formulae", p_i, as follows

$$p \quad \equiv \quad \bigwedge_{i=0}^{n} \mathtt{next}^{\,i}\, p_i,$$

where p_i specifies the ith state and $\mathtt{next}^{\,i}\, p_i$ denotes i applications of the operator `next` to the formula p_i. The whole formula therefore specifies a sequence of n states.

The state formula p_i, which specifies the ith state, is a conjunction of terms of the form $l_j = e_j$ in which l_j is an L-expression and e_j is an arbitrary expression; that is,

$$p_i \quad \equiv \quad \bigwedge_{j=0}^{m_i} l_j = e_j.$$

Thus, p_i specifies the values of the expressions l_j (for all $j < m_i$) on the ith state.

A boolean variable ε, corresponding to the predicate `empty`, is used to mark where the sequence terminates; it is true on the last state and false elsewhere. The value of ε is set by the operators `next` and `empty`. For instance, the program $\texttt{skip} \wedge \texttt{A} = \texttt{0}$ generates an interval of two states. On the first, `A` is `0` and the termination flag ε is `false`; on the second, ε is `true`. Thus, if p is the program $\texttt{skip} \wedge \texttt{A} = \texttt{0}$, then

$$\begin{aligned} p_0 \quad &\equiv \quad (\varepsilon = \texttt{false}) \wedge (\texttt{A} = \texttt{0}) \\ p_1 \quad &\equiv \quad (\varepsilon = \texttt{true}). \end{aligned}$$

The interval length is 1 since $val(\varepsilon) = \texttt{true}$ on the second state.

4.2.2 Reduction of Programs

Let us now introduce a technique for reducing simple programs to canonical form. The technique involves unfolding the program step-by-step until the canonical form is discovered. This technique suggests a way to execute Tempura programs on a computer. Each equation of the form $l = e$ might cause the value of the expression e to be stored in a particular memory location designated by l, and each state formula would then define a state of the computer's memory. Successive states would be generated as the program unfolds.

This section describes the main part of the transformation to canonical form. Section 4.2.3 describes how to eliminate existential quantifiers, and predicates are briefly discussed in section 4.2.5.

The main part of the transformation of program p with respect to termination flag ε is given by $reduce(p, \varepsilon)$. The idea is that the program is equivalent to its transformation with termination flag replaced by `empty`,

$$p \quad \equiv \quad reduce(p, \texttt{empty}).$$

This transformation does not yield the canonical form directly, but something from which it is easily derived. The transformation is defined inductively in terms of its effect on the elementary operators.

Equality The statement $l = e$ is left alone by the transformation; that is,

$$reduce(l = e, \varepsilon) \quad = \quad l = e.$$

However, the statement may be further simplified (see below) by replacing the expression e and any subscript expression in l with its (constant) value.

Parallel composition The statement $p \wedge p'$ is reduced by reducing p and p' together.

$$\mathit{reduce}(p \wedge p', \varepsilon) \quad = \quad \mathit{reduce}(p, \varepsilon) \wedge \mathit{reduce}(p', \varepsilon).$$

For example,

$$\mathit{reduce}(\mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{A}, \varepsilon) \quad = \quad \mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{A}.$$

This may be further simplified to the equivalent form $\mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{0}$.

Conditional The statement $\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ p'$ is executed by transforming p or p' according to the value of b (which must be defined).

$$\mathit{reduce}(\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ p', \varepsilon) \quad = \quad \begin{cases} \mathit{reduce}(p, \varepsilon) & \text{if } b = \mathtt{true}, \\ \mathit{reduce}(p', \varepsilon) & \text{if } b = \mathtt{false}. \end{cases}$$

For example,

$$\mathit{reduce}(\mathtt{A} = \mathtt{0} \wedge \mathtt{if}\ \mathtt{A} = \mathtt{1}\ \mathtt{then}\ \mathtt{B} = \mathtt{0}\ \mathtt{else}\ \mathtt{B} = \mathtt{1}, \varepsilon) \quad = \quad \mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{1}.$$

Note that the value of the boolean expression b must be known before the transformation can take place.

Existential quantification The statement $\exists v : p$ is reduced by reducing p within the scope of the local variable v.

$$\mathit{reduce}(\exists v : p, \varepsilon) \quad = \quad \exists v : \mathit{reduce}(p, \varepsilon).$$

For example,

$$\mathit{reduce}(\exists \mathtt{A} : \mathtt{A} = \mathtt{0}, \varepsilon) \quad = \quad \exists \mathtt{A} : \mathtt{A} = \mathtt{0}.$$

Existential quantification is further discussed in section 4.2.3, where it is shown that the scope of the local variable can be increased to the outermost level.

Termination The statement `empty` simply sets the termination flag ε.

$$\mathit{reduce}(\mathtt{empty}, \varepsilon) \quad = \quad \varepsilon = \mathtt{true}.$$

Note that `empty` may also be used as a boolean expression, in which case it may be replaced by ε.

Next The statement $\mathtt{next}\ p$ does two things. It asserts that the execution has not yet terminated and also that p happens next.

$$\mathit{reduce}(\mathtt{next}\ p, \varepsilon) \quad = \quad \varepsilon = \mathtt{false} \wedge \mathtt{next}\ \mathit{reduce}(p, \varepsilon).$$

For example,

$$\mathit{reduce}(\mathtt{next}\ (\mathtt{A} = \mathtt{0}), \varepsilon) \quad = \quad \varepsilon = \mathtt{false} \wedge \mathtt{next}\ (\mathtt{A} = \mathtt{0}).$$

Sequential composition The statement $p\,;p'$ is executed by first executing p and then executing p'. A new flag ε' must be introduced to mark the termination of p (see also section 4.2.3).

$$
\begin{aligned}
\mathit{reduce}(p\,;p',\varepsilon) \;=\; \exists \varepsilon' : \{ & \\
& \mathit{reduce}(p,\varepsilon') \wedge \\
& (\varepsilon = \mathtt{false})\,\mathtt{until}\;\varepsilon' \wedge \\
& \mathit{reduce}(p',\varepsilon')\,\mathtt{atnext}\;\varepsilon' \\
\}, &
\end{aligned}
$$

where the operators `until` and `atnext` satisfy the following expansion properties:

$$
\begin{aligned}
p\,\mathtt{until}\;b &= \begin{cases} \mathtt{true} & \text{if } b = \mathtt{true}, \\ p \wedge \mathtt{next}\,(p\,\mathtt{until}\;b) & \text{if } b = \mathtt{false}, \end{cases} \\
p\,\mathtt{atnext}\;b &= \begin{cases} p & \text{if } b = \mathtt{true}, \\ \mathtt{next}\,(p\,\mathtt{atnext}\;b) & \text{if } b = \mathtt{false}. \end{cases}
\end{aligned}
$$

For example,

$$
\begin{aligned}
\mathit{reduce}(\mathtt{empty}\,;\mathtt{next\ empty},\varepsilon) \;=\; \exists \varepsilon' : \{ & \\
& \varepsilon' = \mathtt{true} \wedge \\
& \varepsilon = \mathtt{false} \wedge \\
& \mathtt{next}\,(\varepsilon = \mathtt{true}) \\
\}. &
\end{aligned}
$$

Notice that when the above technique is used to execute a program the order in which reductions are carried out may be important. For example, if the program $\mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{0}$ is reduced in left-to-right order the value of `B` will not be defined when it is required for the assignment $\mathtt{A} = \mathtt{B}$. There are various ways to deal with this situation. The simplest is to regard the program as erroneous. More sophisticated approaches involve either transforming the program into the equivalent form $\mathtt{B} = \mathtt{0} \wedge \mathtt{A} = \mathtt{B}$ prior to execution, or postponing reduction of the troublesome statement until the value of `B` is defined. The latter approach is used in the Tempura interpreter. This may, of course, result in deadlock if the program really is erroneous and the required value is not defined at all, but there are straightforward ways to detect the deadlock and take appropriate action.

As already observed, the transformation $\mathit{reduce}(p,\varepsilon)$ does not generally reduce the program p to canonical form, but a number of further transformations may be used to remedy this. First, all existential quantifiers must be moved to the outermost level, and then the conjuncts of the transformed program must be rearranged into the desired final form. The next two sections describe how to do this.

4.2.3 Local Variables

The transformation above leaves existential quantifiers alone. This section describes how programs containing local variables may be transformed so that all variables are declared at the outermost level. Hence, all internal declarations may be removed from a program.

Local variables are introduced either by means of existential quantification or as formal parameters of a function or predicate. They are statically bound; that is, the scope of a particular instance of a variable is determined by where it occurs in the program text rather than by where it is used, and in the usual way, each occurrence of a variable is bound to the smallest enclosing declaration of the same name. For instance, the following program fragment contains two quite separate instances of the variable X. The scope of one is confined to the inner existential quantification, the other's scope covers everywhere that is within the outer but outside the inner quantification.

$$\begin{array}{l}\exists \texttt{X},\texttt{Y} : \{\texttt{X} = \texttt{1} \wedge \texttt{f} = (\lambda() : \texttt{X}) \wedge \exists \texttt{X} : \{\texttt{X} = \texttt{2} \wedge \texttt{Y} = \texttt{f}()\}\}.\\ \qquad\quad\uparrow \qquad\qquad\qquad\qquad\quad \uparrow\end{array}$$

The reference to X within the function f is bound to the outer instance of X.[1] Thus, Y is set to 1.

The scope of a local variable may be increased provided that it is renamed as necessary to avoid name clashes. If v' does not occur freely in either p or p' then

$$\begin{array}{rcl} p \wedge (\exists v : p') & \equiv & \exists v' : \{p \wedge p'[v'/v]\} \\ (\exists v : p) \wedge p' & \equiv & \exists v' : \{p[v'/v] \wedge p'\} \end{array}$$

and similarly for the other primitive operators. For instance, the program fragment above is equivalent to the following fragment in which the inner instance of X has been renamed and moved outwards.

$$\exists \texttt{X},\texttt{Y},\texttt{X}' : \{\texttt{X} = \texttt{1} \wedge \texttt{f} = (\lambda() : \texttt{X}) \wedge \texttt{X}' = \texttt{2} \wedge \texttt{Y} = \texttt{f}()\}.$$

In this way all existential quantifiers may be moved to the outermost level of a program. In practice, of course, explicit renaming is not needed to execute a program, it is sufficient to ensure that each variable corresponds to a unique storage location.

4.2.4 Final Transformation

Having moved all existential quantifiers outwards, the conjuncts of the transformed program must be regrouped into the state formulae described in section 4.2.1 above. This is easily achieved since conjunction is both commutative and associative, and

[1] This, incidentally, shows how lambda expressions may be used to define pointers in Tempura.

since the operator `next` distributes through conjunction. In other words, the program may be rewritten according to the following equivalences:

$$\begin{aligned} p \wedge p' &\equiv p' \wedge p \\ (p \wedge p') \wedge p'' &\equiv p \wedge (p' \wedge p'') \\ (\texttt{next}\ p) \wedge (\texttt{next}\ p') &\equiv \texttt{next}\,(p \wedge p'). \end{aligned}$$

Thus, the canonical form is obtained.

4.2.5 Predicates

Finally, consider the semantics of a predicate invocation. In ITL the application of a predicate defined by

$$p(v_1, \ldots, v_n) \stackrel{\text{def}}{=} \mathit{body}$$

to arguments $\mathit{arg}_1, \ldots, \mathit{arg}_n$ denotes the formula or expression gained by substituting arg_i for all free occurrences of v_i in the body of the predicate,

$$\mathit{body}[\mathit{arg}_1/v_1, \ldots, \mathit{arg}_n/v_n]$$

(renaming bound variables as necessary). This expansion takes place each time the predicate is encountered, so a recursively defined predicate is unfolded one step at a time.

The argument-passing semantics described above are assumed in all the following programs; they are the semantics of call-by-name. However, an implementation of Tempura would probably need to use a more efficient argument passing mechanism than this. Indeed, I indicated in chapter 2 that the Tempura interpreter uses a call-by-reference mechanism by default. In most common cases the semantics of call-by-name and call-by-reference are identical (when the arguments are simple references or static expressions). When they are not, the call-by-name semantics may be simulated using either the macro facility or by introducing an extra variable and using call-by-reference, which may be done automatically.

4.3 Derived Operators

Most of the derived operators that were defined in section 3.3 can also be defined in Tempura, but some of those which were defined in terms of negation are not so general as in ITL.

4.3.1 Classical Operators

Of the classical operators, conjunction and the conditional (itself in restricted form) are taken as primitive in Tempura, and

$$\texttt{if}\ b\ \texttt{then}\ p \quad \equiv \quad \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ \texttt{true}.$$

The remaining boolean connectives may only be used in boolean expressions.

Universal quantification ($\forall$) cannot be defined as it was in full ITL, but bounded universal quantification can be expressed as a finite conjunction of terms. It may be expressed inductively:

$$\begin{aligned} \texttt{forall}\ i < \texttt{0} : p \quad &\equiv \quad \texttt{true} \\ \texttt{forall}\ i < n + \texttt{1} : p \quad &\equiv \quad (\texttt{forall}\ i < n : p) \wedge p[n/i]. \end{aligned}$$

Other forms, such as $\texttt{forall}\ v \in l : p$ for a list l, are defined similarly.

4.3.2 Temporal Operators

The operator `empty` has been taken as primitive, but the unit-length construct, `skip`, can be defined as before, as can the length operator, `len`. The operator `always`, defined in terms of negation, is expanded as follows in Tempura,

$$\texttt{always}(p) \quad \equiv \quad p \wedge \texttt{if}\ \neg\texttt{empty}\ \texttt{then}\ \texttt{next}\ \texttt{always}(p).$$

The operator `sometime` is of little use operationally except perhaps for checking a specification. It can be executed by expanding in a similar way to `always`.

The operators `halt`, which defines a termination condition, `keep`, which is like `always` on all but the last state, and `fin` which asserts its argument on the last state, are all defined as before.

4.3.3 Assignment Operators

Unit-assignment can be defined in Tempura as it was in ITL, but that definition is rather inefficient in practice, and such assignments are so common that it is desirable to build in a more efficient form which by-passes the elementary operations. Indeed, I show in chapter 6 that it is sometimes more natural to regard unit-assignment as primitive.

Stability can also be defined in a much more efficient form than its original definition. On conventional processors it can just be ignored if no checking is required. In chapter 6 I show that the need for `stable` can be eliminated by using *frame variables*.

The other assignment operators, `is`, $\leftarrow$, $\Leftarrow$, and `gets`, are defined as before.

4.3.4 Iterative Operators

The iterative constructs $\texttt{for}\ n\,\texttt{times do}\ p$ and $\texttt{for}\ i < n\ \texttt{do}\ p$ may be defined as before, but the following expansion property of the while loop is used operationally:

$$\texttt{while}\ b\ \texttt{do}\ p \quad = \quad \texttt{if}\ b\ \texttt{then}\ (p\,;\texttt{while}\ b\ \texttt{do}\ p)\ \texttt{else empty},$$

but this does not behave in quite the way desired because if p is empty when b is true the computation never terminates. Further iterative constructs, such as $\texttt{repeat}\ p\,\texttt{until}\ b$, are expanded in a similar way.

4.3.5 Input and Output

A real Tempura program must be given the means to communicate with the outside world through input and output. Useful input and output facilities may need to be quite sophisticated, but for the purposes of this discussion two naive functions will suffice. The function `input` reads input from some device, such as a keyboard, and the function `output` produces output on another device, maybe a terminal. Both of these functions take a variable number of (zero or more) arguments whose values are read from the input device or written to the output device. Absence of input or output is denoted by a call of the corresponding function with no arguments: `input()` or `output()`.

Informally, input and output devices may be thought of as list-valued state variables, `Input` and `Output` say. Then a call of `input(X,Y,Z)` sets `X`, `Y` and `Z` to the appropriate elements of the input list

$$\texttt{input(X,Y,Z)} \quad \equiv \quad \texttt{X} = \texttt{Input[0]} \wedge \texttt{Y} = \texttt{Input[1]} \wedge \texttt{Z} = \texttt{Input[2]},$$

and a corresponding call of the output function constructs an output list from its arguments

$$\texttt{output(X,Y,Z)} \quad \equiv \quad \texttt{Output} = \texttt{[X,Y,Z]}$$

(with some syntactic sweetening). By convention, if no output assertion is made at a particular point in time, then the assertion `output()` is assumed. This prevents any output from appearing by outputting the empty list of values, `Output` = [ ]. Later, in section 6.3.1, I discuss how the same effect might be achieved semantically, without the need for this convention.

Although these input and output operations need to be extended to handle all the options required in a usable system, they can be explained within the semantics of Tempura. They are not “side-effects”.

4.4 Discussion

Although the decision to restrict Tempura to a deterministic subset of ITL was made for sound engineering reasons (such as efficiency of execution), there is every reason to expect that other executable subsets of temporal logic might be equally useful in different problem domains. For executing abstract specifications it may be that a language with conventional "logic programming" features is more appropriate. For instance, language Tokio [FKTM86] makes use of the unification and resolution mechanisms of Prolog (see section 1.3.6). On the other hand, it is possible to define a "stripped down" language for (say) real-time programming. Such a language would not perform run-time checks, and statements would be executed strictly in order of occurrence. The existing interpreter is something of a compromise between the two.

Chapter 5

Verification and Transformation

This chapter discusses how to verify that a Tempura program meets its specification, and how to transform it into another program with the same essential properties. Verification and transformation can both be done using a computerised theorem prover such as HOL. Programs may be verified either directly by natural deduction, or by using proof rules. A program may be transformed using either logical equivalence to get a new program with identical behaviour, or using functional equivalence to get a different program that computes the same function.

Even in the early days of computing it was considered that complex programs sometimes need to be formally checked. In 1949 Turing introduced the idea by showing how to demonstrate the correctness of a small program for computing factorials [Tur49]. However, systematic attempts at program verification really began with Floyd's technique for verifying flowchart programs [Flo67], and Hoare's well-known axiomatic method for proving the correctness of simple sequential programs [Hoa69]. Since then Hoare's logic has been extended to handle a wider class of programs, including parallel programs [Lam80], and has remained one of the most widely accepted methods for verifying imperative programs.

Mathematically-inspired programming languages, such as Tempura and functional languages, raise the possibility of more direct methods of verification, since in these languages the programs are themselves mathematical formulae. This means, for example, that the notion of equivalence between two programs is just the usual logical equivalence, and that the idea of a program implementing a specification may be reduced to logical implication.

However, formal verification remains a very expensive activity and it will not be used on a large scale unless the cost of proving real programs is reduced by several orders of magnitude. Despite this, there are already some areas of design where the potential cost of mistakes exceeds the cost of verification. Examples include the safety-critical control systems used in areas such as medical monitoring, fly-by-wire aircraft systems, railway signalling and nuclear reactor control, and the crucial parts of larger systems such as operating system kernels, communications protocols and security systems. A great deal depends on systems such as these, so it is essential

that they do function properly. Furthermore, these systems are often small and relatively simple compared to most commercial computing systems, so they present an easier target for the verifier.

There are basically two ways to guarantee that a program meets its specification. One is to verify the program after it has been written and tested, to give a final seal of approval. The other is to design the program by transformation, either from another program which is known to be correct, or directly from its specification. Each approach has its place, and Tempura adapts well to either. Not only does Tempura have a direct mathematical interpretation, but the same formalism (ITL) is also used for specification.

5.1 Verification

A promising approach to verification, and one which seems to offer the most hope when faced with real programs, uses the embedding of ITL in higher-order logic described in chapter 3. It is a promising approach because higher-order logic provides a foundation for integrating ITL with other mathematical theories, and also because a number of powerful theorem proving systems for higher-order logic already exist. One of these, Gordon's HOL system [Gor87], forms the basis of this discussion.

5.1.1 Natural Deduction

ITL, as presented in chapter 3, is a conservative extension of the primitive HOL logic together with whatever theories are needed to construct expressions (numbers, lists and so on). The only axioms are the definitions of the operators; no new inference rules are axiomatised. Theorems of ITL can be proved by natural deduction from the definitions, using the usual inference rules of logic, such as substitution of equality, *modus ponens* and induction.

For example, the meaning of `empty` is stated in the following theorem, which asserts that `empty` is true on any interval τ of length zero. The turnstile symbol ($\vdash$) denotes that what follows is a theorem.

$$\vdash \texttt{ empty } \mathbf{=} \lambda\tau : |\tau| \mathbf{=} 0 \qquad [\texttt{empty}\text{ semantics}]$$

This theorem is easily proved by expanding the definition of `empty`, substituting the definitions of negation, `next` and `true`, simplifying the result, and then using the fact that zero is the least natural number.[1]

[1] Recall that the emboldened forms denote classical operators.

$$\boxed{\text{TO PROVE} \quad \texttt{empty} = \lambda\tau : |\tau| = 0}$$

$$\begin{aligned}
\texttt{empty} &= \neg\,\texttt{next true} && \text{[definition of } \texttt{empty}\text{]} \\
&= \lambda\tau :\neg\,(|\tau| > 0 \wedge \mathsf{T}) && \text{[expanding definitions]} \\
&= \lambda\tau :\neg\,(|\tau| > 0) && \text{[since } p \wedge \mathsf{T} = p\text{]} \\
&= \lambda\tau : |\tau| \leq 0 && \text{[since } \neg\,(n > m) = n \leq m\text{]} \\
&= \lambda\tau : |\tau| = 0 && \text{[since } \neg\,(n < 0) \text{ for } n \in \mathbb{N}\text{]}
\end{aligned}$$

A semantic theorem of this kind can be obtained for each of the derived operators of ITL. Here are some examples in which v and v' are *state* variables, which, as you will recall from section 3.2.3, denote the functions $mk_state(\hat{v})$ and $mk_state(\hat{v}')$.

$$\begin{aligned}
\vdash\quad p \supset p' &= \lambda\tau : p(\tau) \supset p'(\tau) && [\supset \text{ semantics}] \\
\vdash\quad \texttt{skip} &= \lambda\tau : |\tau| = 1 && [\texttt{skip} \text{ semantics}] \\
\vdash\quad v := v' &= \lambda\tau : |\tau| = 1 \wedge \hat{v}(\tau_1) = \hat{v}'(\tau_0) && [:= \text{ semantics}] \\
\vdash\quad v \leftarrow v' &= \lambda\tau : \hat{v}(\tau_{|\tau|}) = \hat{v}'(\tau_0) && [\leftarrow \text{ semantics}]
\end{aligned}$$

The first example states that implication is just a lifted form of classical implication, the second that `skip` is true on any unit-length interval, the third that the unit-assignment $v := v'$ is true on an interval of unit-length if the next value of v equals the initial value of v', and the last one states that the temporal assignment $v \leftarrow v'$ holds whenever the final value of v equals the initial value of v'. Theorems such as these greatly simplify the task of proving properties of programs.

5.1.2 Properties of Programs

A program p is said to satisfy the specification s if any execution of p results in a behaviour for which the specification is also true; that is, if it can be proved that p implies s on every interval:

$$\vdash\quad \forall\,\tau \in \mathbb{I} : (p \supset s)(\tau).$$

Formulae such as this, which are true on all intervals, are said to be *valid* in ITL; and the double turnstile symbol ($\models$) is used to denote a valid formula. Thus,

$$\models\quad p \quad\supset\quad s$$

is another way of writing the theorem above.

A trivial example of a valid property is the following theorem, which asserts that if the variable `Y` is initially one, and it is multiplied by `x` on a single step, then the effect is to set the final value of `Y` to `x`.

$$\models\quad (\texttt{Y} = \texttt{1} \wedge \texttt{Y} := \texttt{Y} \times \texttt{x}) \quad\supset\quad (\texttt{Y} \leftarrow \texttt{x}).$$

The proof of this theorem in HOL comes in a straightforward way from substituting the definitions of the operators and simplifying the result. Each step in the proof follows from the previous one by elementary rules of logic.

$\boxed{\text{TO PROVE} \quad \forall\,\tau : ((\mathtt{Y} = \mathtt{1} \wedge \mathtt{Y} := \mathtt{Y} \times \mathtt{x}) \supset (\mathtt{Y} \leftarrow \mathtt{x}))(\tau)}$

1. $(\mathtt{Y} = \mathtt{1} \wedge \mathtt{Y} := \mathtt{Y} \times \mathtt{x})(\tau)$ [program]
2. $(\mathtt{Y} = \mathtt{1})(\tau) \wedge (\mathtt{Y} := \mathtt{Y} \times \mathtt{x})(\tau)$ [$\wedge$ semantics]
3. $(\hat{\mathtt{Y}}(\tau_0) = 1) \wedge (|\tau| = 1) \wedge (\hat{\mathtt{Y}}(\tau_1) = \hat{\mathtt{Y}}(\tau_0) \times \hat{\mathtt{x}})$ [$=$, $:=$ semantics]
4. $(\hat{\mathtt{Y}}(\tau_0) = 1) \wedge (|\tau| = 1) \wedge (\hat{\mathtt{Y}}(\tau_1) = 1 \times \hat{\mathtt{x}})$ [substituting $\hat{\mathtt{Y}}(\tau_0)$]
5. $(\hat{\mathtt{Y}}(\tau_0) = 1) \wedge (|\tau| = 1) \wedge (\hat{\mathtt{Y}}(\tau_1) = \hat{\mathtt{x}})$ [since $1 \times \hat{\mathtt{x}} = \hat{\mathtt{x}}$]
6. $(\hat{\mathtt{Y}}(\tau_0) = 1) \wedge (|\tau| = 1) \wedge (\hat{\mathtt{Y}}(\tau_{|\tau|}) = \hat{\mathtt{x}})$ [substituting $|\tau|$]
7. $(\hat{\mathtt{Y}}(\tau_{|\tau|}) = \hat{\mathtt{x}})$ [since $(p \wedge p') \supset p'$]
8. $(\mathtt{Y} \leftarrow \mathtt{x})(\tau)$ [$\leftarrow$ semantics]

This sort of proof can be extremely tedious; it is neither mathematically interesting (because it is shallow), nor computationally interesting (because you already "knew" the result to be true). Large examples are also error-prone because of the huge numbers of logical inferences required to prove even the simplest results. It is mainly for these reasons that proofs are better done with a computerised tool, such as HOL, to take care of all the trivial steps.

5.1.3 Mathematical Induction

An important technique in program verification is mathematical induction. For instance, the following valid property states that if `Y` is initially one, and is multiplied `n` times by `x`, then it ends up with the value $\mathtt{x}^{\mathtt{n}}$:

$$\models \quad (\mathtt{Y} = \mathtt{1} \wedge \mathtt{for\ n\ times\ do\ Y} := \mathtt{Y} \times \mathtt{x}) \quad \supset \quad (\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n}}).$$

To prove this result for general `n` requires induction. The proof is in two parts, the first of which is a proof that the result holds in the base case, when $\mathtt{n} = \mathtt{0}$, and the second that if it holds for some `n` then it must hold for $\mathtt{n} + \mathtt{1}$. From these two proofs one may conclude that the result holds for any value of `n`.

$\boxed{\text{BASE CASE} \quad \forall\,\tau : ((\mathtt{Y} = \mathtt{1} \wedge \mathtt{for\ 0\ times\ do\ Y} := \mathtt{Y} \times \mathtt{x}) \supset (\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{0}}))(\tau)}$

1. $(\mathtt{Y} = \mathtt{1} \wedge \mathtt{for\ 0\ times\ do\ Y} := \mathtt{Y} \times \mathtt{x})(\tau)$ [base case]
2. $(\mathtt{Y} = \mathtt{1} \wedge \mathtt{empty})(\tau)$ [`for` definition]
3. $(\hat{\mathtt{Y}}(\tau_0) = 1) \wedge (|\tau| = 0)$ [`empty`, $=$ semantics]
4. $(\hat{\mathtt{Y}}(\tau_{|\tau|}) = 1)$ [substituting $|\tau|$]
5. $(\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{0}})(\tau)$ [since $\mathtt{x}^{\mathtt{0}} = \mathtt{1}$]

INDUCTIVE STEP	$\forall\,\tau : ((\mathtt{Y} = 1 \wedge \mathtt{for\ n+1\ times\ do\ Y := Y \times x}) \supset (\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n+1}}))(\tau)$
ASSUMING	$\forall\,\tau : ((\mathtt{Y} = 1 \wedge \mathtt{for\ n\ times\ do\ Y := Y \times x}) \supset (\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n}}))(\tau)$

1. $(\mathtt{Y} = 1 \wedge \mathtt{for\ n+1\ times\ do\ Y := Y \times x})(\tau)$ [inductive step]
2. $(\mathtt{Y} = 1 \wedge \mathtt{for\ n\ times\ do\ Y := Y \times x\,;\,Y := Y \times x})(\tau)$ [for definition]
3. $(\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n}}\,;\,\mathtt{Y := Y \times x})(\tau)$ [by assumption]
4. $\exists\, i \leq \hat{\mathtt{n}} + 1 : (\hat{\mathtt{Y}}(\tau_i) = \hat{\mathtt{x}}^{\hat{\mathtt{n}}}) \wedge (\hat{\mathtt{Y}}(\tau_{i+1}) = \hat{\mathtt{Y}}(\tau_i) \times \hat{\mathtt{x}}) \wedge (|\tau| = i + 1)$ [$\leftarrow$, chop, := sem.]
5. $(\hat{\mathtt{Y}}(\tau_{|\tau|}) = \hat{\mathtt{x}}^{\hat{\mathtt{n}}} \times \hat{\mathtt{x}})$ [substituting $|\tau|$ and $\hat{\mathtt{Y}}(\tau_i)$]
6. $(\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n+1}})(\tau)$ [since $\mathtt{x}^{\mathtt{n}} \times \mathtt{x} = \mathtt{x}^{\mathtt{n+1}}$]

Thus, the iterative program has the desired functional behaviour for every value of n.

Functional properties of the kind described so far are perhaps the most common forms of specification, as very often one is only concerned with the final result of a calculation. However, particularly when dealing with interactive programs, it is sometimes necessary to verify what happens during the execution of a program. For instance, a control program might repeatedly increment a certain variable, X say, whilst checking that some other quantity does not exceed $\mathtt{X} \times \mathtt{n}$. Rather than repeatedly multiply X by n, the program maintains another variable Y which it increments in steps of n. This program depends on the property that incrementing Y in steps of n results in the value of Y always being n times that of X. In other words,

$$\models \quad (\mathtt{X} = 0 \wedge \mathtt{Y} = 0 \wedge \mathtt{X\ gets\ X} + 1 \wedge \mathtt{Y\ gets\ Y} + \mathtt{n}) \quad \supset \quad (\mathtt{Y\ is\ X} \times \mathtt{n}).$$

The proof of this theorem also requires induction, but the details are not given.

5.1.4 Proof Rules

Verification is a time-consuming business. One way to speed it up is to develop higher-level proof rules, so that a proof may be done in a smaller number of higher-level steps. For instance, in chapter 6 I show that the rules of Hoare's logic can also be used as proof rules in ITL, though now they become derived theorems rather than accepted laws. This has the advantage that it is possible to derive rules for new program operators as needed, and without the risk of introducing unsoundness into the logic.[2]

5.1.5 Hierarchical Decomposition

Another way to make the verification task more manageable is to divide a large proof into smaller parts, prove the parts independently, and then recombine them

[2] An unsound system is one that allows you to prove contradictory results.

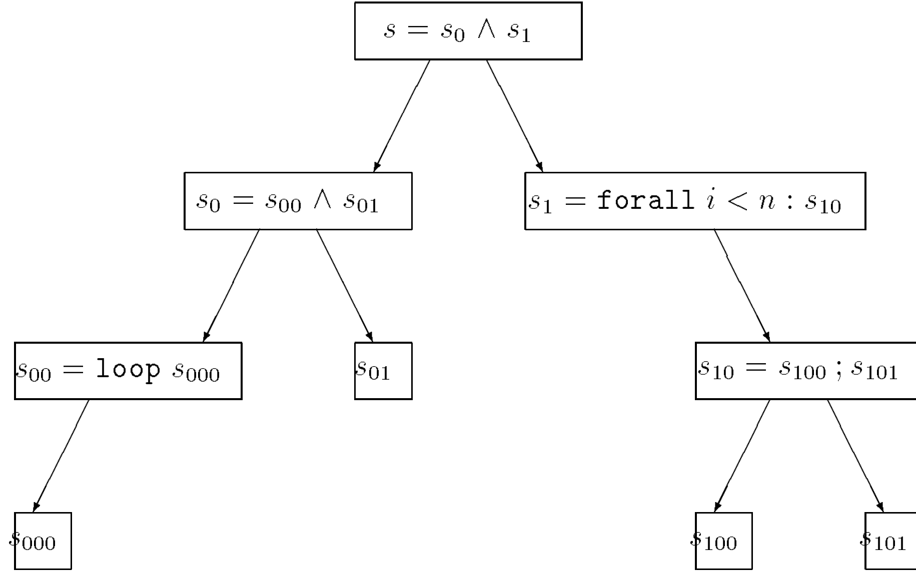


Figure 5.1: Hierarchical decomposition of a specification s into a simpler specifications, each of which may be implemented and verified separately.

---

to get the desired result. This is possible because programs and specifications are compositional; in other words, when two programs are composed together their properties are also.

In general, if p and p' are two programs with specifications s and s', then the conjunction of p and p' satisfies the conjunction of their specifications. This is expressed in the following rule:

$$\frac{\models p \supset s, \qquad \models p' \supset s'}{\models (p \wedge p') \supset (s \wedge s')} \qquad \text{[and-composition]}$$

(the conclusion below the line may be deduced from the hypotheses above). Likewise, the sequential composition of p and p' satisfies the sequential composition of their specifications,

$$\frac{\models p \supset s, \qquad \models p' \supset s'}{\models (p\, ;\, p') \supset (s\, ;\, s')} \qquad \text{[chop-composition]}$$

and similar relationships hold for other Tempura operators. Thus, a large problem may be hierarchically decomposed into its component parts.

Figure 5.1 shows how a typical specification might be decomposed into a number of simpler specifications, each of which may be implemented separately. In a large software project such a decomposition might be used to assign subproblems to different teams of programmers.

5.2 Transformation

Program transformation is just another aspect of verification, but one which is concerned with proving a relationship (such as equivalence) between two different programs. The purpose of transforming a program is normally to change its operational behaviour in some way, perhaps to make it execute more efficiently. But transformation may be used simply to show that two programs are equivalent by transforming them both, step by step, into the same form.

5.2.1 Transformation Rules

The most straightforward method of transformation is to replace one formula or term with a logically equivalent one. This results in a new program whose behaviour is identical in every respect to the original. For example, because conjunction is commutative and associative the order and grouping of parallel operations is immaterial. Thus, the program

$$\mathtt{Y,N}\ \mathtt{gets}\ \mathtt{Y} \times \mathtt{x}, \mathtt{N} - \mathtt{1} \wedge \mathtt{Y,N} = \mathtt{1,n} \wedge \mathtt{halt}\,(\mathtt{N} = \mathtt{0})$$

may be transformed into the equivalent program

$$\mathtt{Y,N} = \mathtt{1,n} \wedge \mathtt{halt}\,(\mathtt{N} = \mathtt{0}) \wedge \mathtt{Y,N}\ \mathtt{gets}\ \mathtt{Y} \times \mathtt{x}, \mathtt{N} - \mathtt{1},$$

and on a uniprocessor the transformed program may be executed more efficiently than the original because each step can be performed in a single left-to-right pass (see section 4.2.2).

Transformations such as these may be applied mechanically by rewriting the original program. Any equivalence may be used for transformation, and equivalent programs or sub-programs may be substituted for one another under any circumstances. In other words,

$$\text{if} \quad \models p \equiv p' \quad \text{then} \quad \begin{cases} \models (p \wedge p'') \equiv (p' \wedge p'') \\ \models (p'' \wedge p) \equiv (p'' \wedge p') \\ \models (\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ p'') \equiv (\mathtt{if}\ b\ \mathtt{then}\ p'\ \mathtt{else}\ p'') \\ \models (\mathtt{if}\ b\ \mathtt{then}\ p''\ \mathtt{else}\ p) \equiv (\mathtt{if}\ b\ \mathtt{then}\ p''\ \mathtt{else}\ p') \\ \models (\exists v : p) \equiv (\exists v : p') \\ \models (\mathtt{next}\ p) \equiv (\mathtt{next}\ p') \\ \models (p\ ;\ p'') \equiv (p'\ ;\ p'') \\ \models (p''\ ;\ p) \equiv (p''\ ;\ p') \end{cases}$$

Some useful equivalences are given here for later use. These examples follow the usual naming convention; that is, p, p' and p'' denote arbitrary formulae and e, e'

denote *state* expressions.

$$\begin{array}{llcll} \models & p \wedge p' & \equiv & p' \wedge p & \text{[and-commut]} \\ \models & (p \wedge p') \wedge p'' & \equiv & p \wedge (p' \wedge p'') & \text{[and-assoc]} \\ \models & (\texttt{next}\ p) \wedge (\texttt{next}\ p') & \equiv & \texttt{next}\,(p \wedge p') & \text{[next-and]} \\ \models & (\texttt{next}\ p)\,;p' & \equiv & \texttt{next}\,(p\,;p') & \text{[next-chop]} \\ \models & (p\,;p')\,;p'' & \equiv & p\,;(p'\,;p'') & \text{[chop-assoc]} \\ \models & (e = e' \wedge p)\,;p' & \equiv & e = e' \wedge (p\,;p') & \text{[equals-chop]} \\ \models & \texttt{empty}\,;p & \equiv & p & \text{[empty-chop]} \end{array}$$

The first two of these transformations were used above; the others will be used in later sections.

5.2.2 Canonical Form

Each of the above transformations produces a program that is logically equivalent to the original. As well as being used to derive a new program from an existing one, they may therefore be used to verify the equivalence of two existing programs. In principle, two programs are equivalent if one can be transformed into the other, but in practice it is often simpler to transform both into the same intermediate form. If this can be done both programs, p and p' say, are equivalent to the intermediate form, p'', and hence to one another because equivalence is transitive; that is,

$$\frac{\models p \equiv p'', \qquad \models p'' \equiv p'}{\models p \equiv p'} \qquad \text{[equiv-transitive]}$$

The technique can be made more methodical by fixing on a canonical intermediate form, and a natural choice is to use a reduction process similar to the one for executing programs, as described in section 4.2.2. The general idea can be understood from the examples below.

As a first example, consider reducing the program below, which sets the variable `Y` to `1` and then multiplies it by `x` in one step.

$$\texttt{Y} = \texttt{1} \wedge \texttt{Y} := \texttt{Y} \times \texttt{x}$$

It is not hard to see that this program just sets `Y` to `x` on the next step and then halts. The transformation follows the sequence of steps below. Each line follows from its predecessor by rewriting with the equivalences above in addition to the usual rules for substitution and quantifier elimination, but some elementary steps are omitted in order to simplify the description.

$$\boxed{\text{TO PROVE}\quad \mathtt{Y} = \mathtt{1} \wedge \mathtt{Y} := \mathtt{Y} \times \mathtt{x} \;\equiv\; \mathtt{Y} = \mathtt{1} \wedge \mathtt{next}\,(\mathtt{empty} \wedge \mathtt{Y} = \mathtt{x})}$$

1. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{Y} := \mathtt{Y} \times \mathtt{x}$ [initial program]
2. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{skip} \wedge \exists \mathtt{x}' : \{\mathtt{x}' = \mathtt{Y} \times \mathtt{x} \wedge \mathtt{next}\,(\mathtt{Y} = \mathtt{x}')\}$ [:= definition]
3. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{next}\,(\mathtt{empty}) \wedge \exists \mathtt{x}' : \{\mathtt{x}' = \mathtt{Y} \times \mathtt{x} \wedge \mathtt{next}\,(\mathtt{Y} = \mathtt{x}')\}$ [`skip` definition]
4. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{next}\,(\mathtt{empty}) \wedge \exists \mathtt{x}' : \{\mathtt{x}' = \mathtt{x} \wedge \mathtt{next}\,(\mathtt{Y} = \mathtt{x})\}$ [substitution of Y and x′]
5. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{next}\,(\mathtt{empty}) \wedge \mathtt{next}\,(\mathtt{Y} = \mathtt{x})\}$ [quantifier elimination]
6. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{next}\,(\mathtt{empty} \wedge \mathtt{Y} = \mathtt{x})\}$ [next-and]

This is really no more than a restatement of the semantics of unit-assignment.

A more substantial example is to show the equivalence of the two programs, $\mathtt{exp_pgm(x,n,Y,N)}$ and $\mathtt{exp_pgm'(x,n,Y,N)}$, which were introduced in chapter 2. Both calculate the value of $\mathtt{x^n}$ in Y, but they are defined in very different ways. The first one uses a while-loop in the usual imperative manner.

$$\begin{aligned}\mathtt{exp_pgm(x,n,Y,N)} &\stackrel{\text{def}}{=} \mathtt{Y,N} \Leftarrow \mathtt{1,n}\,;\mathtt{exp_while(Y,N)}\\ \mathtt{exp_while(Y,N)} &\stackrel{\text{def}}{=} \mathtt{while\ N} \neq \mathtt{0\ do\ Y,N} := \mathtt{Y} \times \mathtt{x}, \mathtt{N} - \mathtt{1},\end{aligned}$$

and for general n this program may be reduced to the recursive form:

$$\models \quad \mathtt{exp_pgm(x,n,Y,N)} \;\equiv\; \begin{array}{l}\mathtt{Y,N} = \mathtt{1,n} \wedge \\ \mathtt{if\ n} = \mathtt{0\ then\ empty}\\ \mathtt{else}\ \{\mathtt{next\ exp_pgm(x,n-1,Y,N)}\}.\end{array}$$

The reduction is in two parts; the first is to transform the chop into a conjunction, and the second is to transform the while-loop. The first part proceeds as follows,

$$\boxed{\text{TO PROVE}\quad \mathtt{Y,N} \Leftarrow \mathtt{1,n}\,;\mathtt{exp_while(Y,N)} \;\equiv\; \mathtt{Y,N} = \mathtt{1,n} \wedge \mathtt{exp_while(Y,N)}}$$

1. $\mathtt{Y,N} \Leftarrow \mathtt{1,n}\,;\mathtt{exp_while(Y,N)}$ [initial program]
2. $(\mathtt{empty} \wedge \mathtt{Y,N} = \mathtt{1,n})\,;\mathtt{exp_while(Y,N)}$ [⇐ definition]
3. $(\mathtt{Y,N} = \mathtt{1,n} \wedge \mathtt{empty})\,;\mathtt{exp_while(Y,N)}$ [and-commut]
4. $\mathtt{Y,N} = \mathtt{1,n} \wedge (\mathtt{empty}\,;\mathtt{exp_while(Y,N)})$ [equals-chop]
5. $\mathtt{Y,N} = \mathtt{1,n} \wedge \mathtt{exp_while(Y,N)}$ [empty-chop]

Transformation of the while-loop depends on the recursive "unfolding" property below, which reflects the way that the loop `while` b `do` p is executed; if b is true do p and then test b again, otherwise halt.

$$\models \quad \mathtt{while}\ b\ \mathtt{do}\ p \;\equiv\; \mathtt{if}\ \neg b\ \mathtt{then\ empty\ else}\ (p\,;\mathtt{while}\ b\ \mathtt{do}\ p) \qquad \text{[unfold-while]}$$

After unfolding the loop, the transformation is much like the one above; the unit-assignment is reduced first, and then the result is simplified.

The second program, $\texttt{exp_pgm}'(\texttt{x},\texttt{n},\texttt{Y},\texttt{N})$, uses the operators **gets** and **halt** in place of the while-loop in $\texttt{exp_pgm}$.

$$\begin{aligned}\texttt{exp_pgm}'(\texttt{x},\texttt{n},\texttt{Y},\texttt{N}) &\stackrel{\text{def}}{=} \texttt{Y} = \texttt{1} \wedge \texttt{N} = \texttt{n} \wedge \texttt{exp_gets}(\texttt{x},\texttt{n},\texttt{Y},\texttt{N}) \\ \texttt{exp_gets}(\texttt{x},\texttt{n},\texttt{Y},\texttt{N}) &\stackrel{\text{def}}{=} \texttt{halt}\,(\texttt{N} = \texttt{0}) \wedge \texttt{Y},\texttt{N}\ \texttt{gets}\ \texttt{Y} \times \texttt{x}, \texttt{N} - \texttt{1}.\end{aligned}$$

Since **gets** and **halt** are defined in terms of **always**, the transformation this time depends on an unfolding theorem for the operator **always**.

$$\models \ \texttt{always p} \ \equiv \ \texttt{p} \wedge \texttt{if}\ \neg\texttt{empty then}\ (\texttt{next always p}) \qquad \text{[unfold-always]}$$

In other respects the transformation proceeds in much the same way as those above to give the equivalent intermediate form:

$$\begin{aligned}\texttt{exp_pgm}'(\texttt{x},\texttt{n},\texttt{Y},\texttt{N}) \equiv\ & \texttt{Y},\texttt{N} = \texttt{1},\texttt{n} \wedge \\ & \texttt{if n} = \texttt{0 then empty} \\ & \texttt{else}\ \{\texttt{next exp_pgm}'(\texttt{x},\texttt{n} - \texttt{1},\texttt{Y},\texttt{N})\}.\end{aligned}$$

The two programs therefore reduce to identical (primitive recursive) form, from which it may be concluded that they are equivalent.

5.2.3 Functional Equivalence

All the transformations encountered up to now have been logical equivalences; in other words they transform one program into another that has identical behaviour. Very often, however, equivalence is too strong a relation. It means, for example, that the original and transformed programs take exactly the same number of steps to complete, whereas it may only be important that they produce the same final result from the same initial data. In fact, one of the principal uses of program transformation is to come up with better and faster ways of achieving equivalent results. In this section I introduce a weaker relation, called functional equivalence, to capture the idea of programs producing equivalent final results.

The method is straightforward. Informally, one first defines the function, $\texttt{function}\,(p)$, of a program p to be its behaviour on the first and last states of an interval. So, for example,

$$\begin{aligned}\models \quad \texttt{function}\,(\texttt{X} := \texttt{0}) &\equiv \texttt{X} \leftarrow \texttt{0} \\ \models \quad \texttt{function}\,(\texttt{X} := \texttt{0}\,;\texttt{X} := \texttt{X} + \texttt{1}) &\equiv \texttt{X} \leftarrow \texttt{1}.\end{aligned}$$

Then one defines two programs to be functionally equivalent if they have the same function. In this view a program is a “black box” that accepts initial data, grinds away for a while, and finally produces a result of some kind; the intermediate behaviour is not considered. An important consequence is that if two programs are functionally equivalent, then replacing one by the other in a sequence of steps results in a new program with the same function.

Let us define functional behaviour semantically.[3] The formula $\texttt{function}\, p$ is true on an interval τ if there is another interval τ' on which p is true, and which has the same first and last states as τ; that is,

$$\texttt{function}\, p \stackrel{\text{def}}{=} \lambda\tau : \exists\, \tau' : (p(\tau') \wedge (\tau'_0 = \tau_0) \wedge (\tau'_{|\tau'|} = \tau_{|\tau|})).$$

For instance, the function of a functional specification is itself,

$$\models \texttt{function}(v \leftarrow e) \equiv v \leftarrow e.$$

and the function of `empty` is to ensure that the initial and final states of the interval are identical, which means that every variable must be assigned to itself.

Functional equivalence is simply defined to be equivalence between functional behaviours. Thus, programs p and p' are functionally equivalent, written $p \sim p'$, if $\texttt{function}\, p$ and $\texttt{function}\, p'$ are equivalent:

$$p \sim p' \stackrel{\text{def}}{=} \texttt{function}\, p \equiv \texttt{function}\, p'.$$

For example,

$$\texttt{Y} = \texttt{1} \wedge \texttt{Y} := \texttt{x}^{\texttt{n}} \sim \texttt{Y} = \texttt{1} \wedge \texttt{for n times do Y} := \texttt{Y} \times \texttt{x}.$$

Note that functional equivalence *is* an equivalence relation. Furthermore, a program must satisfy its own function

$$\models p \supset \texttt{function}\, p,$$

so if two programs p and p' are functionally equivalent, they satisfy the same functional specification.

Functional equivalence has some useful properties. The most important is that all operators, *except* conjunction, preserve functional equivalence. Thus,

$$\text{if} \quad \models p \sim p' \quad \text{then} \quad \begin{cases} \models (\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p'') \sim (\texttt{if}\ b\ \texttt{then}\ p'\ \texttt{else}\ p'') \\ \models (\texttt{if}\ b\ \texttt{then}\ p''\ \texttt{else}\ p) \sim (\texttt{if}\ b\ \texttt{then}\ p''\ \texttt{else}\ p') \\ \models (\exists v : p) \sim (\exists v : p') \\ \models (\texttt{next}\ p) \sim (\texttt{next}\ p') \\ \models (p\, ; p'') \sim (p'\, ; p'') \\ \models (p''\, ; p) \sim (p''\, ; p') \end{cases}$$

Conjunction does not in general preserve functional equivalence because both computation length and intermediate behaviour are important in this case. However, under severe restrictions, functionally equivalent programs can be substituted in conjunctions. Sufficient restrictions are that the computation lengths of the two sub-programs should be equal, and neither conjunct should access a variable that

[3]This can also be defined using *temporal abstraction*, which is the "inverse" of temporal projection (see chapter 9). I have discussed temporal abstraction previous work [Hal87].

is modified by the other; that is, the parallel sub-programs operate independently.[4] Under these restrictions,

$$\text{if} \quad \models p \sim p' \quad \text{then} \quad \begin{cases} \models (p \wedge p'') \sim (p' \wedge p'') \\ \models (p'' \wedge p) \sim (p'' \wedge p') \end{cases}$$

Note that the computation lengths of two programs can be made equal by extending the shorter one to the length of the longer, whilst keeping all program variables stable. The parallel composition operator ($\|$), which is introduced in chapter 8, does this automatically.

5.2.4 Efficiency

Application of these rules lets us systematically improve the operational characteristics of programs whilst preserving their functional behaviour. Consider, yet again, the iterative program for calculating $\mathtt{x}^{\mathtt{n}}$,

$$\mathtt{Y} = \mathtt{1} \wedge \mathtt{for\ n\ times\ do\ Y} := \mathtt{Y} \times \mathtt{x},$$

which was shown in section 5.1.3 to have the function $\mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n}}$. This time the aim is to derive a "faster" version.

The first step is to transform this program into an equivalent one by splitting the for-loop into a sequence of smaller loops. In general, if $\mathtt{n}$ and $\mathtt{n}'$ are non-negative,

$$\models \quad \mathtt{for\ n + n'times\ do\ p} \quad \equiv \quad \mathtt{for\ n\ times\ do\ p\ ;for\ n'times\ do\ p}.$$

In this case, an obvious way to divide the loop is according to the binary expansion of $\mathtt{n}$; that is, the expansion $\mathtt{n} = \sum_{\mathtt{i}=0}^{\mathtt{log(n)}} \mathtt{bit(i,n)} \times 2^{\mathtt{i}}$, where $\mathtt{bit(i,n)}$ denotes the $\mathtt{i}$th bit in the binary representation of $\mathtt{n}$, and $\mathtt{log(n)}$ here denotes the number of bits needed to represent $\mathtt{n}$ (so $\mathtt{log(0)} = \mathtt{1}$). This leads to the logically equivalent program:

$$\begin{array}{l} \mathtt{Y} = \mathtt{1} \wedge \\ \mathtt{for\ i} < \mathtt{log(n)} + \mathtt{1\ do} \\ \quad \mathtt{for\ bit(i,n)} \times 2^{\mathtt{i}}\mathtt{times\ do\ Y} := \mathtt{Y} \times \mathtt{x}. \end{array}$$

The inner loops have the function $\mathtt{Y} \leftarrow (\mathtt{if\ bit(i,n)} = \mathtt{1\ then\ Y} \times \mathtt{x}^{2^{\mathtt{i}}}\ \mathtt{else\ Y}$, so, supposing for the moment that we already have a way to calculate $\mathtt{x}^{2^{\mathtt{i}}}$, they may be rewritten to get the functionally equivalent program:

$$\begin{array}{l} \mathtt{Y} = \mathtt{1} \wedge \\ \mathtt{for\ i} < \mathtt{log(n)} + \mathtt{1\ do} \\ \quad \mathtt{Y} := (\mathtt{if\ bit(i,n)} = \mathtt{1\ then\ Y} \times \mathtt{x}^{2^{\mathtt{i}}}\ \mathtt{else\ Y}), \end{array}$$

---

[4] Naturally, functional equivalence is not a useful concept for reasoning about parallel co-operating processes. General laws for co-operating processes may arise from the message-passing mechanisms described in chapter 10.

and, of course, $\mathtt{x}^{2^{\mathtt{i}}}$ can be calculated very quickly by repeatedly squaring `x`. Thus, this program may be transformed to the logically equivalent form below, using a local variable to hold the values of $\mathtt{x}^{2^{\mathtt{i}}}$.

$$
\begin{array}{l}
\exists \mathtt{X} : \{ \\
\quad \mathtt{X}, \mathtt{Y} = \mathtt{x}, 1 \land \\
\quad \mathtt{for}\ \mathtt{i} < \mathtt{log}(\mathtt{n}) + 1\ \mathtt{do} \\
\quad\quad \mathtt{X}, \mathtt{Y} := \mathtt{X} \times \mathtt{X}, (\mathtt{if}\ \mathtt{bit}(\mathtt{i}, \mathtt{n}) = 1\ \mathtt{then}\ \mathtt{Y} \times \mathtt{X}\ \mathtt{else}\ \mathtt{Y}) \\
\}.
\end{array}
$$

This is functionally equivalent to the original. The new program outperforms the original when $\mathtt{log}(\mathtt{n}) + 1 < \mathtt{n}$; that is, for values of `n` in excess of `2`.

5.3 Discussion

I do not pretend in this chapter to have presented a fully-fledged proof theory for Tempura programs; much more work is needed just to come up a system that is useable on realistic problems. Nevertheless, this is a start. I have shown that a range of proof and transformation techniques can be applied to Tempura programs, and there are many more strategies that might be used. For instance, parts of the verification process might be automated, and certain kinds of programs might be synthesised from their specifications. However, there are limits to what verification can achieve.

5.3.1 Satisfaction Guaranteed?

Taking satisfaction to be logical implication as above gives rise to an unfortunate loophole. If a program is ever inconsistent, and so logically false, it then satisfies any specification whatsoever, because the formula `false` implies anything. Consider the following valid implication:

$$\models \quad (\mathtt{len}(\mathtt{n}) \land \mathtt{I} = 0 \land \mathtt{I} \leftarrow \mathtt{I} + 1) \quad \supset \quad (\mathtt{I} \leftarrow 1).$$

The program $\mathtt{len}(\mathtt{n}) \land \mathtt{I} = 0 \land \mathtt{I} \leftarrow \mathtt{I} + 1$ does indeed satisfy the specification, $\mathtt{I} \leftarrow 1$, provided that `n` is not zero. But if `n` is zero the program asserts that $\mathtt{I} = 0$ and $\mathtt{I} = 1$ at the same time; it is logically inconsistent, and its behaviour is unpredictable. In the extreme case, the program `false` implies every specification, but achieves nothing at all.

Camilleri *et al.* discuss a number of ways around this problem [CGM86] but none of them seems entirely satisfactory. The inconsistency in the example above could be spotted by attempting to prove that the program can be executed successfully for any value of `n`; in other words for all values of `n` there is an interval on which the program is true. Failure to prove the case $\mathtt{n} = 0$ might suggest that something

was amiss, and would certainly prevent one from concluding that the program was correct.

There are, however, some drawbacks to this technique. Firstly, it cannot be directly incorporated into the definition of satisfaction without sacrificing compositionality. Secondly, it makes it harder to write programs that have genuine "don't care" inputs. For example, one might not care what the program above does when `n` is zero if it is never to be used in such a situation. Nevertheless, a consistency check is always required before concluding that a program satisfies its specification.

Of course, it is also possible to draw unintented conclusions if one's specification contains errors; in the worst case, the specification `true` is satisfied by any program whatsoever. This extreme is most unlikely to occur in practice, but it is nevertheless true that one needs to be confident that one has specified the intended behaviour. However, specification errors cannot readily be detected by formal verification; and that is why simulation and prototyping are indispensable.

5.3.2 Mechanical Verification

Numerous computer-aided verification systems have been designed and built, but mostly they fall into two broad categories reflecting the philosophy of their designers. In the first category are the completely automatic verifiers which make few demands of the user. He or she just enters a description of what is to be proved, and some time later the system outputs a verdict: "true" or "false" (or possibly "don't know"). In the second category are the user-guided systems which require the user to know something about proof techniques. As well as providing a description of what is to be proved, he or she must now give directions on how to do the proof. Both approaches have their merits.

There are completely automatic verification systems for temporal logic. Examples include Clarke's Model Checker [CES86] and Abadi's resolution system R [AM86]. The principal advantage of these verifiers is simply that they are automatic. The user need have no knowledge of the underlying mathematics, nor of proof techniques in general. The disadvantage of automatic systems is that they must be based on tractable decision procedures, and this limits the power of the specification language. Clarke's system can only handle finite-state programs, and Abadi's is based on a first-order linear-time temporal logic (having no chop operator and only static variables). Clearly, only certain aspects of program behaviour can be investigated with these systems.

The principal advantage of user-guided systems, such as the HOL system, is that they are based on richer mathematics in which ITL and other useful theories can be embedded. The disadvantage of user-guided systems is that the user must know how to do the proof, at least in outline. The system just checks the proof and fills in the low-level details; though what is meant by low-level depends on the system

and on the skill of the user. The HOL system, for example, can be "programmed" to perform complex proof strategies automatically, thus combining the advantages of human insight and mechanised drudgery.

5.3.3 Transformation and Synthesis

There has been low-key interest in program transformation for many years. A seminal paper by Darlington and Burstall [DB73] describes a system for transforming functional programs. However, they do not give a formal measure of performance, so their guide to program improvement is empirically based. More recently, Hoare and Roscoe have devised systems of algebraic laws which may be used to transform CSP and Occam programs [Hoa85, RH88]. Again, no formal measure of improvement can be given, although parallel and sequential behaviours can be differentiated. Tempura, however, makes possible a delicate measure of improvement, namely the number of computational steps, though there must, of course, be restrictions governing what can be done on each step. These will depend upon the implementation environment.

Another kind of transformation, and one which offers an alternative to verification, is the direct synthesis of programs from their specifications. Other researchers have devised procedures for synthesizing certain types of programs. For example, Burstall and Darlington [BD77] and Manna and Waldinger [MW80] describe ways to synthesize recursive programs; and Emerson and Clarke [EC82] and Manna and Wolper [MW84] describe how to synthesize the synchronisation parts of concurrent programs from temporal logic specifications. Some of these techniques might well be adapted to synthesise Tempura programs, and could be mechanised in HOL.

Chapter 6

Sequential Programs

This chapter shows how to represent the semantics of a simple sequential programming language in ITL. Most importantly, it shows that the usual destructive assignment statement of imperative programming, with its associated inertial assumption, can be represented in ITL without the need for new axioms. This is achieved by defining a new operator, called **`frame`**, which asserts that a variable remains stable unless it is explicitly changed.

This chapter concerns the relationship between Tempura and ordinary sequential programming in a language such as Pascal. It focusses on two crucial issues: assignment and inertia. Both concern change, and it might be thought that they are just two different ways of looking at the same problem. Assignments tell us what does change, inertia tells us what does not. But they are treated in quite different ways. Assignment is an active operation which causes a value to change, whereas inertia is a passive property which allows us to assume that variables do not change between assignments. I think it is fair to say that in the logical analysis of sequential processes these two issues have caused more problems than any other.

Two principal ways have been used to deal at a formal level with the problems of destructive assignment and inertia. One way, exemplified by Hoare Logic [Hoa69], is to axiomatise the problem away. In this approach, one simply takes assignment and stability to be primitive by building their behaviour into a special logic. The trouble is that the elegant semantics of classical logic are lost in the process. The other way, exemplified by the functional approach to programming [Bac78], is to program in abstract languages that entirely reject the concept of change. In this way the simpler mathematical semantics are retained, but something of the range and clarity of expression of imperative languages is lost. Besides, there is no getting away from the fact that the vast majority of processors in use today depend on something like destructive assignment at the primitive instruction level.

In this chapter I hope to convince you that ITL can handle both assignment and inertia in a purely logical framework. First, I show that the semantics of Tempura are already quite close to the accepted view of sequential programming, the differ-

ence being that Tempura contains no inertial assumption. Then, to overcome this problem, I show how to define the property of inertia within ITL. This is done by defining a new operator, `frame`, which chooses the inertial behaviour for a particular variable. At the end of the chapter, I discuss some of the pitfalls of the frame operator and a way to achieve almost the same effect without it.

6.1 Assignment

Simple sequential programs are really much the same whether they are written in Tempura or Pascal. But there is one major difference. In a Pascal program, when an assignment is made to one variable it is assumed that all the other variables stay the same. This is not true in Tempura, at least not until the next section.

6.1.1 Semantics

To see the similarity, let us compare a standard treatment of program semantics with the temporal logic view. Specifically, let us take Hoare's axiomatic logic [Hoa69] as the basis for discussion, and then show that the same behaviour is represented in ITL, without additional axioms.

Hoare's Logic

In Hoare's method two assertions, a pre-condition p and a post-condition q, are associated with each statement s of the program. In his original notation

$$p\{s\}q$$

is used to denote that if p holds immediately before s is executed, and if s terminates, then q will hold on termination.

Assignment is taken to be a primitive operation. Its effect is captured by an axiom scheme which says that if some post-condition p is to hold following the assignment $v \leftarrow e$, then before the assignment the same condition must have held with v replaced by e.

$$\vdash p[e/v]\{v \leftarrow e\}p \qquad \text{[assignment axioms]}$$

In addition, there are a number of inference rules for deriving properties of compound statements from the properties of their constituent parts. Three examples are given below. The first of these allows one to strengthen the pre-condition or weaken the post-condition, the second expresses the semantics of sequential composition, and the last one expresses the semantics of the while-loop.

$$\frac{\vdash p' \supset p, \qquad \vdash p\{s\}q, \qquad \vdash q \supset q'}{\vdash p'\{s\}q'} \qquad \text{[consequence]}$$

$$\frac{\vdash p\{s\}q, \qquad \vdash q\{s'\}r}{\vdash p\{s\,;\,s'\}r} \qquad \text{[sequence]}$$

$$\frac{\vdash p \wedge b\{s\}p}{\vdash p\{\texttt{while}\ b\ \texttt{do}\ s\}\neg b \wedge p} \qquad \text{[iteration]}$$

These rules can be applied, statement by statement, to derive pre- and post-conditions for simple while programs.

Temporal Logic

Now let us look at how this works in temporal logic. The first thing to notice is that assignment in Tempura is not quite as axiomatised by Hoare. Hoare's axiom says two things. It asserts that the variable v is updated in the right way, but it also asserts that no other variable changes. In other words, it assumes an inertial system. For example, the variable $\texttt{Y}$ is assumed to remain stable when the assignment $\texttt{N} \leftarrow \texttt{N} - \texttt{1}$ is made. No such *a priori* assumption is made in Tempura, so the assignment $\texttt{N} \leftarrow \texttt{N} - \texttt{1}$ says nothing whatsoever about $\texttt{Y}$.

In spite of this inertia *can* be represented in ITL, and how to do it is the subject of section 6.2. But for now let us see how far we can get without assuming inertia.

Let us continue with the notation $p\{s\}q$, but now it stands for the equivalent expression in temporal logic,

$$p\{s\}q \quad = \quad \texttt{init}(p) \supset s \supset \texttt{fin}(q),$$

which should be read: "If p holds initially then if the statement s executes successfully then q will hold finally". The meaning is therefore just the same as before, but now the the program is itself a logical formula, so the axioms and rules are derived theorems.

The effect of a single conventional assignment $v_i \leftarrow e_i$ in a program which uses the variables $v_0, \ldots, v_n$ can be represented in Tempura by the multiple assignment

$$v_0, \ldots, v_i, \ldots, v_n \leftarrow v_0, \ldots, e_i, \ldots, v_n.$$

A multiple assignment of this form has essentially the same semantics as before. In general, provided that only the program variables $v_0, \ldots, v_n$, are free in the post-condition p, a multiple assignment satisfies:

$$\models p[e_0/v_0, \ldots, e_n/v_n]\{v_0, \ldots, v_n \leftarrow e_0, \ldots, e_n\}p \qquad \text{[assignment]}$$

It could be taken as a convention that all the program variables not explicitly mentioned in the assignment are assigned to themselves, but I shall not do this.

The others of Hoare's rules are also formally derived from the semantics of ITL, and not surprisingly they turn out to be just as before.

$$\frac{\models p' \supset p, \qquad \models p\{s\}q, \qquad \models q \supset q'}{\models p'\{s\}q'} \qquad \text{[consequence]}$$

$$\frac{\models p\{s\}q, \qquad \models q\{s'\}r}{\models p\{s\,;\,s'\}r} \qquad \text{[sequence]}$$

$$\frac{\models p \wedge b\{s\}p}{\models p\{\texttt{while}\ b\ \texttt{do}\ s\}\neg b \wedge p} \qquad \text{[iteration]}$$

Furthermore, it is easy to derive more rules for other program operators. For instance, there is one for composing programs in parallel:

$$\frac{\models p\{s\}q, \qquad \models q\{s'\}r}{\models p \wedge p'\{s \wedge s'\}q \wedge q'} \qquad \text{[parallel]}$$

Though when using this rule one must be careful to ensure that the parallel programs are not contradictory (see section 5.3.1).

Special Cases

The two most frequently used forms of assignment in Tempura programs are equality and next-assignment. These are both special cases of temporal assignment,

$$\models \quad \texttt{empty} \wedge v \leftarrow e \quad \supset \quad v = e$$
$$\models \quad \texttt{skip} \wedge v \leftarrow e \quad \supset \quad v \circeq e.$$

There are alternative forms of both operators, initialisation and unit-assignment, with the computation length built-in,

$$v \Leftarrow e \quad \stackrel{\text{def}}{=} \quad \texttt{empty} \wedge v = e$$
$$v := e \quad \stackrel{\text{def}}{=} \quad \texttt{skip} \wedge v \circeq e.$$

It follows that these two operators also satisfy the multiple-assignment rule above.

An Example

Using the rules above, one may prove correct the following Tempura program which is supposed to compute $\texttt{x}^{\texttt{n}}$ in $\texttt{Y}$:

$$\begin{aligned} \texttt{exp_pgm}(\texttt{x},\texttt{n},\texttt{Y},\texttt{N}) \quad \stackrel{\text{def}}{=} \quad & \texttt{Y},\texttt{N} \Leftarrow \texttt{1},\texttt{n}; \\ & \texttt{while}\ \texttt{N} \neq \texttt{0}\ \texttt{do}\ \{ \\ & \quad \texttt{Y},\texttt{N} := \texttt{Y} \times \texttt{x}, \texttt{N} - \texttt{1} \\ & \}. \end{aligned}$$

The aim is to show that if the program terminates then $\texttt{Y}$ ends up with the value $\texttt{x}^{\texttt{n}}$. The proof proceeds as follows:

TO PROVE $\mathtt{true}\{\mathtt{Y,N} \Leftarrow \mathtt{1,n}\,;\mathtt{while\ N} \neq \mathtt{0\ do\ Y,N} := \mathtt{Y} \times \mathtt{x,N} - \mathtt{1}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n}}$

1. $\mathtt{true}\{\mathtt{Y,N} \Leftarrow \mathtt{1,n}\}\mathtt{Y} = \mathtt{1} \wedge \mathtt{N} = \mathtt{n}$ [assignment]
2. $\mathtt{Y} = \mathtt{1} \wedge \mathtt{N} = \mathtt{n} \quad \supset \quad \mathtt{Y} = \mathtt{x}^{\mathtt{n-N}}$ [arithmetic]
3. $\mathtt{true}\{\mathtt{Y,N} \Leftarrow \mathtt{1,n}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}}$ [consequence]
4. $\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}} \wedge \mathtt{N} \neq \mathtt{0}\{\mathtt{Y,N} := \mathtt{Y} \times \mathtt{x,N} - \mathtt{1}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}}$ [assignment]
5. $\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}}\{\mathtt{while\ N} \neq \mathtt{0\ do\ Y,N} := \mathtt{Y} \times \mathtt{x,N} - \mathtt{1}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}} \wedge \mathtt{N} = \mathtt{0}$ [iteration]
6. $\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}} \wedge \mathtt{N} = \mathtt{0} \quad \supset \quad \mathtt{Y} = \mathtt{x}^{\mathtt{n}}$ [arithmetic]
7. $\mathtt{Y} = \mathtt{x}^{\mathtt{n-N}}\{\mathtt{while\ N} \neq \mathtt{0\ do\ Y,N} := \mathtt{Y} \times \mathtt{x,N} - \mathtt{1}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n}}$ [consequence]
8. $\mathtt{true}\{\mathtt{Y,N} \Leftarrow \mathtt{1,n}\,;\mathtt{while\ N} \neq \mathtt{0\ do\ Y,N} := \mathtt{Y} \times \mathtt{x,N} - \mathtt{1}\}\mathtt{Y} = \mathtt{x}^{\mathtt{n}}$ [sequence]

The proof is exactly as it would have been in Hoare's logic, except that one now arrives at a theorem of ITL,

$$\models \ \mathtt{exp_pgm(x,n,Y,N)} \ \supset \ \mathtt{Y} \leftarrow \mathtt{x}^{\mathtt{n}}.$$

The proof can be done mechanically using the methods suggested in chapter 5.

6.1.2 Notes on Assignment

By and large, the assignment operators in Tempura behave in just the same way as assignment in a conventional imperative language. But some aspects of their behaviour may not be obvious and are therefore worth pointing out. These observations all have to do with timing, something that is not considered in the conventional treatment.

Equality and Causality

Equality, when used as an assignment, asserts that two values are equal *at the same point* in the program. An assignment such as $\mathtt{A} \Leftarrow \mathtt{A} + \mathtt{1}$ is therefore impossible to achieve. Logically, it is simply false,

$$\models \ \mathtt{A} \Leftarrow \mathtt{A} + \mathtt{1} \ \equiv \ \mathtt{false},$$

since there is no choice of $\mathtt{A}$ for which $\mathtt{A}$ and $\mathtt{A} + \mathtt{1}$ are equal.

Equality represents an instantaneous communication. It is not a causal operation, and this is apparent in Tempura. For instance, the following four programs are logically equivalent, since they all initialise $\mathtt{A}$ and $\mathtt{B}$ to zero on an empty interval:

1. $\mathtt{A} \Leftarrow \mathtt{0}\,;\mathtt{B} \Leftarrow \mathtt{A}$
2. $\mathtt{A,B} \Leftarrow \mathtt{0,A}$
3. $\mathtt{B,A} \Leftarrow \mathtt{A,0}$
4. $\mathtt{B} \Leftarrow \mathtt{A}\,;\mathtt{A} \Leftarrow \mathtt{0}.$

But they are not equally easy to execute!

To be executed with maximum efficiency, the assignments must be encountered in the "correct" order. In the first program this is obviously so. The value of A is defined before it is needed for the second assignment. In the second program the assignments are also correctly ordered, for the multiple assignment expands as follows:

$$\mathtt{A}, \mathtt{B} \Leftarrow \mathtt{0}, \mathtt{A} \quad \equiv \quad \mathtt{empty} \wedge \mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{A}.$$

This being so, the assignments in third program must be encountered out of order. However, this can be discovered by dataflow analysis and the program transformed prior to execution, using the fact that conjunction commutes,

$$\models \quad \mathtt{empty} \wedge \mathtt{B} = \mathtt{A} \wedge \mathtt{A} = \mathtt{0} \quad \equiv \quad \mathtt{empty} \wedge \mathtt{A} = \mathtt{0} \wedge \mathtt{B} = \mathtt{A}.$$

Although the last program can be transformed in a similar way, it is not in general so simple because the chop operator is not commutative.

Assignment and Termination

Temporal assignment on an empty interval is just another representation of equality, and it suffers from all the problems mentioned above. But there is an additional hazard in this case. In a statement such as

$$\mathtt{halt}(\mathtt{B}) \wedge \mathtt{A} \leftarrow \mathtt{A} + \mathtt{1}$$

great care must be taken to ensure that B can never be true initially; if it is, the interval is empty and the program false. As a general rule it is better to reserve temporal assignment for specifications and use unit-assignment in programs.

Unit-assignment is the minimum length assignment over which a change in value can be effected, and it bears a close resemblance to conventional assignment. Unlike equality, unit-assignment is not order-sensitive. A unit-assignment, such as the exchange

$$\mathtt{A}, \mathtt{B} := \mathtt{B}, \mathtt{A},$$

may be executed by first evaluating all the expressions on the right-hand side and only then updating the locations on the left-hand side. This directly reflects the definition of unit-assignment given on page 55, where a local copy of the right-hand expression is made on the current state and assigned to the left-hand location on the next state.

Lists

There are some particular problems associated with assigning values to individual elements of a list. First, if just a single element is assigned on a given step, it is necessary to ensure that all the other elements (and the number of elements) remain stable across the assignment. A predicate such as `alter` could be used for this purpose, where `alter(L, N, X)` changes the `N`th element of the list `L` to `X`, keeping all other elements stable,

$$\begin{aligned} \texttt{alter(L,N,X)} \stackrel{\text{def}}{=} \; & |\texttt{L}| := |\texttt{L}| \wedge \\ & \texttt{forall i} < |\texttt{L}| : \\ & \quad \texttt{if i} = \texttt{N then L}_\texttt{i} := \texttt{X else L}_\texttt{i} := \texttt{L}_\texttt{i}. \end{aligned}$$

A more general predicate could be defined to alter a number of elements at a time.

Another rather more subtle difficulty in assigning to an element of a list occurs when the element is referenced by a variable which itself changes on the same step. For instance, in the program below an element of the list `L` is referenced by the state variable `N`.

$$\begin{aligned} & \texttt{N}, \texttt{L}_\texttt{N} \Leftarrow \texttt{0}, \texttt{0}; \\ & \texttt{N}, \texttt{L}_\texttt{N} := \texttt{1}, \texttt{1}. \end{aligned}$$

This program initialises both `N` and $\texttt{L}_\texttt{0}$ to zero, but then it sets $\texttt{L}_\texttt{1}$ to `1`, not $\texttt{L}_\texttt{0}$ as you might expect. This is because the location to be assigned is evaluated when the assignment is made (after one step), rather than when the right-hand side of the assignment is evaluated. Again, the predicate `alter` solves this problem, because it tests the value of the subscript `N` at the start of the assignment and then keeps that value in a static variable (the control variable of the `forall`) until the assignment is complete.

6.2 Inertia

The accepted view of assignment, as described above, is one aspect of a general physical phenomenon. Our perception of the world is that most things are stable for most of the time, so we confine ourselves to saying what changes from one moment to another, and assume that "everything else" remains the same. This phenomenon has been extensively studied in artificial intelligence circles, where it is known as the "frame problem". It is a major obstacle to a purely logical treatment of human reasoning.

The problem is less acute in the world of programming. Indeed, I have already shown that Tempura can be made to work without a frame assumption. Nevertheless, it is more convenient if we don't have to write extra code just to keep variables stable, especially as most computers have an inertial assumption built-in, making the extra code redundant.

In this section I propose a way to define inertia in ITL by introducing a new class of "frame" variables, which automatically remain stable between assignments. I also discuss some of the limitations of frame variables, and finally I describe a way to achieve a similar effect without frame variables.

6.2.1 Frame Variables

Consider the sequential program $\mathtt{seq_exp_pgm}(\mathtt{x}, \mathtt{n}, \mathtt{Y}, \mathtt{N})$ below, which is intended to calculate the value of $\mathtt{x}^{\mathtt{n}}$ in Y using an auxiliary variable N.

$$\begin{array}{lll}\mathtt{seq_exp_pgm}(\mathtt{x}, \mathtt{n}, \mathtt{Y}, \mathtt{N}) & \stackrel{\text{def}}{=} & \mathtt{Y}, \mathtt{N} \Leftarrow 1, \mathtt{n}; \\ & & \mathtt{while}\ \mathtt{N} \neq 0\ \mathtt{do}\ \{ \\ & & \quad \mathtt{Y} := \mathtt{Y} \times \mathtt{x}; \\ & & \quad \mathtt{N} := \mathtt{N} - 1 \\ & & \}. \end{array}$$

It is syntactically correct, of course, and the intended behaviour, shown here for $\mathtt{x} = 2$ and $\mathtt{n} = 3$, does indeed satisfy it.

time	0	1	2	3	4	5	6
Y	1	2	2	4	4	8	8
N	3	3	2	2	1	1	0

(6.1)

The trouble is that unintended behaviours do also. For example, if $\mathtt{N}'$ behaves in the same way as N, but $\mathtt{Y}'$ is set to zero on the second step, and remains at zero thereafter,

time	0	1	2	3	4	5	6
$\mathtt{Y}'$	1	2	0	0	0	0	0
$\mathtt{N}'$	3	3	2	2	1	1	0

(6.2)

then $\mathtt{seq_exp_pgm}(\mathtt{x}, \mathtt{n}, \mathtt{Y}', \mathtt{N}')$ holds. So too does $\mathtt{seq_exp_pgm}(\mathtt{x}, \mathtt{n}, \mathtt{Y}'', \mathtt{N}'')$, where $\mathtt{Y}''$ behaves like Y, but $\mathtt{N}''$ is set to one on the first step and zero on the second, causing immediate termination.

time	0	1	2
$\mathtt{Y}''$	1	2	2
$\mathtt{N}''$	3	1	0

(6.3)

What must be done is to select just those behaviours which also satisfy the frame assumption.

The idea of this proposal is to define a new operator `frame`, so that the formula $\mathtt{frame}\ v : p$ is true if v satisfies the frame assumption as well as the property p. So the intention is that if all variables are framed the usual semantics of assignment obtain; that is, frame variables retain their values unless explicitly changed. For

instance, framing Y and N in the program seq_exp_pgm above ought to have the same effect as inserting the appropriate assignments; that is,

$$\texttt{frame N : frame Y : seq_exp_pgm(x, n, Y, N)} \equiv \texttt{stb_exp_pgm(x, n, Y, N)},$$

where stb_exp_pgm(x, n, Y, N) is defined as follows:

```
stb_exp_pgm(x, n, Y, N)  ≝  Y, N ⇐ 1, n;
                             while N ≠ 0 do {
                               Y, N := Y × x, N;
                               Y, N := Y, N − 1
                             }.
```

The distinguishing feature of the framed behaviour is that in some sense it "minimises the change" in variables Y and N. Let us therefore try to formalise this idea of minimising change.

Minimising Change

First, we need a way to mark the steps on which a variable changes. This is easily achieved by introducing a predicate $\delta(v)$ which is false whenever the next value of v is the same as its current value; otherwise it is true. For instance, this is how $\delta(\mathtt{Y})$ and $\delta(\mathtt{N})$ behave on the inertial interval (6.1) above:

time	0	1	2	3	4	5	6
Y	1	2	2	4	4	8	8
δ(Y)	true	false	true	false	true	false	true
N	3	2	2	1	1	0	0
δ(N)	false	true	false	true	false	true	true

The predicate $\delta(v)$ is simply true whenever v is not stable from one state to the next,

$$\delta(v) \stackrel{\text{def}}{=} \neg(v \circeq v),$$

but observe that $\delta(v)$ is necessarily true on the last state of an interval.

When presented with two behaviours that satisfy a given formula, represented by the variables v and v' say, then the one which changes less may be chosen by comparing $\delta(v)$ and $\delta(v')$. This is not a straightforward comparison, since it must take causality into account. For example, the behaviour of Y above is preferred to Y$'$ even though the total number of changes in Y on the inertial interval (6.1) is greater than the total number of changes to Y$'$ on the other (6.2). To allow for this, the comparison should only proceed up to the *first* time that $\delta(v)$ and $\delta(v')$ differ.

If the notation $p \lhd p'$ is used to denote this comparison for two formulae p and p' (p is causally less than p' say), then $p \lhd p'$ if at some time p is false and p' is true, and at all times before that p and p' are equal. That is,

$$p \lhd p' \stackrel{\text{def}}{=} (p \equiv p')\,\mathtt{until}\,(\neg p \wedge p'),$$

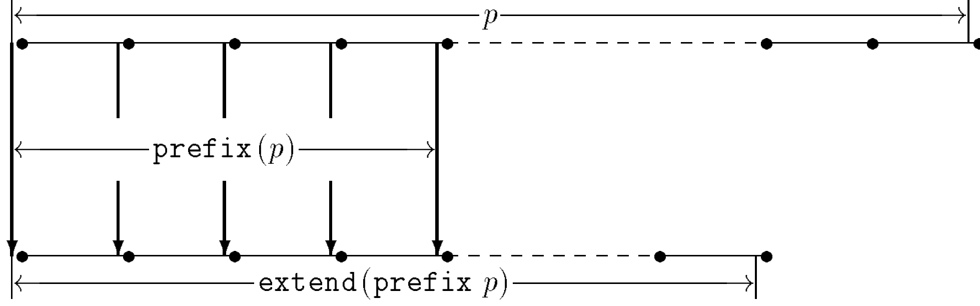


Figure 6.1: Construction of the formula `extend` `prefix` p.

where the property p `until` p' holds if p is true until p' becomes true. It was defined on page 57.

The Operator `frame`

Now the operator `frame` can be defined so that `frame` $v : p$ accepts only those intervals which satisfy the frame assumption for v. It is necessary to look, not just at all possible variables on one particular interval, but also at all the intervals that initially overlap the one being considered. It is essential to do this because otherwise behaviours such as the short interval above on which `N`$''$ was prematurely set to zero (6.3) would not be eliminated.

Two operators, `extend` and `prefix`, may be combined to examine all intervals which are the same up to a certain point, and then diverge. The operator `extend` looks at shorter intervals, whereas `prefix` examines longer ones. Thus, the formula `extend` p is true on an interval if the property p holds on some prefix subinterval, whereas `prefix` p is true on any prefix subinterval of a longer interval on which p holds. Their interval semantics are as follows:

$$\begin{aligned} \texttt{extend}\, p &= \lambda\tau : \exists\, \tau' : |\tau'| \leq |\tau| \wedge \tau' = \mathit{prefix}(|\tau'|, \tau) \wedge p(\tau') \\ \texttt{prefix}\, p &= \lambda\tau : \exists\, \tau' : |\tau| \leq |\tau'| \wedge \tau = \mathit{prefix}(|\tau|, \tau') \wedge p(\tau'), \end{aligned}$$

where $\tau, \tau' \in \mathbb{I}$. A definition of `extend` in terms of chop was given on page 52.

Finally, the inertial property `frame` $v : p$ holds on an interval, τ say, provided that property p holds and there is no more stable behaviour v' that makes p true on another interval, τ' say, which shares its first few states with τ.

$$\texttt{frame}\ v : p \stackrel{\text{def}}{=} p \wedge \nexists v' : \texttt{extend}\{v' = v \wedge \texttt{prefix}(p[v'/v]) \wedge \delta(v') \lhd \delta(v)\},$$

where the variable v' is not free in p. The meaning of `extend` `prefix` p is illustrated in figure 6.1.

An Example

Consider what happens when $\mathtt{Y}$ is framed over the first two steps of the program $\mathtt{seq_exp_pgm}(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N})$, assuming that $\mathtt{n} \neq \mathtt{0}$.

$$\mathtt{frame}\ \mathtt{Y} : \{\mathtt{Y} \Leftarrow \mathtt{1}\,;\mathtt{N} \Leftarrow \mathtt{n}\,;\mathtt{Y} := \mathtt{Y} \times \mathtt{x}\,;\mathtt{N} := \mathtt{N} - \mathtt{1}\}.$$

Framing $\mathtt{Y}$ does not affect the value of $\mathtt{N}$ in this example, so the only choice to be made is the value of $\mathtt{Y}$ on the second state, everything else is predetermined. Denoting this value by $\hat{\mathtt{Y}}(\tau_2)$, the behaviour of $\mathtt{Y}$ is as follows:

time	0	1	2
$\mathtt{Y}$	$\mathtt{1}$	$\mathtt{x}$	$\hat{\mathtt{Y}}(\tau_2)$
$\delta(\mathtt{Y})$	true	$\hat{\mathtt{Y}}(\tau_2) \neq \mathtt{x}$	true

The value of $\mathtt{Y}$ on the first and second states is fixed. To minimise change $\delta(\mathtt{Y})$ must be `false` on the second state, if this is possible, for if it is not then there does exist an $\mathtt{Y}'$ for which the formula holds and $\delta(\mathtt{Y}') \lhd \delta(\mathtt{Y})$, namely the one for which $\delta(\mathtt{Y}')$ *is* false on the second state! In this example it is possible, and only $\hat{\mathtt{Y}}(\tau_2) = \mathtt{2}$ makes $\mathtt{next}\,(\delta(\mathtt{Y}) = \mathtt{false})$. Hence the inertial behaviour of $\mathtt{Y}$ is derived.

The inertial behaviour of $\mathtt{N}$ may be derived in a similar way by framing $\mathtt{N}$, and observing that the only free choice is the value of $\mathtt{N}$ on the second state. Taking this value to be $\mathtt{n} - \mathtt{1}$ minimises $\delta(\mathtt{N})$.

By extending these derivations to take in subsequent steps, the correct inertial behaviour is found for the whole program. It is, as promised, exactly as if explicit assignments had been inserted,

$$\models\ \mathtt{frame}\ \mathtt{N} : \mathtt{frame}\ \mathtt{Y} : \mathtt{seq_exp_pgm}(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N}) \ \equiv\ \mathtt{stb_exp_pgm}(\mathtt{x},\mathtt{n},\mathtt{Y},\mathtt{N}).$$

But things are not quite so simple as they might appear.

6.2.2 Notes on Frame Variables

Does the frame operator always capture the intended behaviour? The answer is that it seems to do so for Tempura programs that are causal, but it has to be admitted that the definition is rather tricky, and has not yet been proved to work in all such programs. On the other hand, it is known *not* to work well with non-causal or non-deterministic formulae. This is quite a reasonable restriction because inertia is fundamentally causal; without the concept of a directional flow of time it has no meaning. The examples below illustrate this.

Equality and Causality

First, note that a frame variable must be initialised, because if it is not then its initial value must be determined by looking ahead in time. For instance, if the

variable A is assigned the value 0 in one step but has no initial value, then framing A forces its initial value to also be 0.

$$\models \texttt{frame A} : \{\texttt{A} := \texttt{0}\} \quad \equiv \quad \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{A} := \texttt{A}\}.$$

This defies causality. It cannot be implemented without backtracking over time.

In practice, there are two ways to deal with this problem. One is to insist that frame variables are explicitly initialised. The other way is to assume a default initial value if none is specified. This could be either a special value, "undefined", or some particular value of an appropriate type. Most other programming languages assume default values, which are usually supposed to be "undefined".

As observed already, equality acts as a zero-delay assignment, which means that it is not causal and so cannot be expected to mix well with the frame operator. For example, in the following formula framing A has no effect:

$$\models \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{skip} \,;\, \texttt{B} \Leftarrow \texttt{A}\} \quad \equiv \quad \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{skip} \,;\, \texttt{B} \Leftarrow \texttt{A}\}.$$

Perhaps you might expect that framing A would force A to always be zero, and hence also force B to be zero on the next step, but this does not happen. Of course, it is still possible that A is zero on the next step, but framing A gives no further information about its next value.

Assignment and Termination

As intended, unit-assignment generally fits well into an inertial environment. For example, when the second initialisation in the previous program is changed into unit-assignment the situation is completely different. Now framing A does force it to always be zero,

$$\models \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{skip} \,;\, \texttt{B} := \texttt{A}\} \quad \equiv \quad \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{A} := \texttt{A} \,;\, \texttt{B} := \texttt{A}\}.$$

This is because the next value of A is chosen by looking at prefix subintervals on which its possible values are unconstrained (whereas before they were constrained to equal B).

However, there are still anomalies with assignment. For example, every assignment must be associated with a particular computation length or termination flag. Unit-assignment therefore works correctly (if it didn't there would be little point to the frame operator), but in general assignment only behaves in the desired way if it is "known in advance" when to complete the assignment.

To see this, consider an interval on which the variable A is initially 0 and is assigned the value 1 in two steps. One might expect that framing A would cause it to retain its initial value until the penultimate state, and then to assume the value 1 on the final state. Indeed, if the interval length is known, this is just what happens,

$$\models \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{len}(2) \wedge \texttt{A} \leftarrow \texttt{1}\} \quad \equiv \quad \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{A} := \texttt{A} \,;\, \texttt{A} := \texttt{1}\}.$$

But if the interval length is not within the scope of the frame operator, the formula is false,

$$\models \texttt{len}(2) \wedge \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{A} \leftarrow \texttt{1}\} \;\equiv\; \texttt{false}.$$

This happens because it is always possible to find another variable $\texttt{A}'$, such that $\texttt{A}'$ is assigned the value `1` on an extended interval, of length `3` say, over which $\texttt{A}'$ is stable for the first `2` steps. Such an $\texttt{A}'$ changes less than `A`.

Non-Determinism

The frame operator is not well-behaved for non-deterministic formulae; but then you would not expect it to be. Typically, it forces a particular choice that makes the formula deterministic. For example, in the following case a unique choice is forced by framing `A`:

$$\models \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, (\texttt{A} := \texttt{1} \vee \texttt{A} := \texttt{0})\} \;\equiv\; \texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{A} := \texttt{0}.$$

But this result is determined by looking ahead in time, rather than by inertia.

It is of no consequence to Tempura that disjunction and the frame operator do not mix well, since disjunction is not a program operator. However, non-determinism can be introduced in a limited way by means of the conditional. For instance, in the program below the frame variable `A` is forced to change because that is the only way to satisfy the program.

$$\models \texttt{frame A} : \{\texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{skip} \,;\, \texttt{if A} \neq \texttt{1 then false}\} \;\equiv\; \texttt{A} \Leftarrow \texttt{0} \,;\, \texttt{skip} \,;\, \texttt{A is 1}.$$

There are, of course, much more subtle representations of `false`.

If an implementation is to generate the "correct" behaviour in these circumstances it is required to look ahead and then backtrack. Otherwise it will generate an error, which actually seems the most appropriate course of action since the programmer did not (or should not) deliberately write a such a program. Programs like this cause more serious problems for verification than for execution, since it is possible to deduce that a program is correct when in fact it is not, in much the same way as it is possible to deduce that the program `false` satisfies any specification you like.

Lists

Although it was not explicitly stated above, the definition of `frame` does not work for lists. When just one element of a list is changed one would expect the rest of the list to remain stable, whereas the frame operator says nothing about individual elements. If one element has been changed nothing further can be said about the values of the others.

It is easy to solve this problem for fixed-length lists (*i.e.* vectors) by defining a special operator `framevect`. This operator minimises changes to each element of the vector.

$$\begin{aligned}\texttt{framevect}\ v : p \ \stackrel{\text{def}}{=}\ & p \land \\ & \nexists v' : \texttt{extend}\{ \\ & \quad \texttt{prefix}(p[v'/v]) \land \exists i < |v| : \delta(v'_i) \lhd \delta(v_i) \\ & \},\end{aligned}$$

where v' is not free in p. How to handle arbitrary data structures is an open question. However, I shall only need framed vectors in the following (and shall assume the above semantics).

6.2.3 The Operator `local`

A frame variable ought to be framed throughout its scope. The semantics of `frame` do not demand this, but it is hard to think of a situation in which it could usefully be otherwise. Furthermore, implementing frame variables is made unnecessarily difficult if this rule is not observed. For example, an attempt to assign a new value to a frame variable from outside the scope of its frame should most probably result in an error, but this is difficult to check for. Such a situation occurs in the program below, which is logically false:

$$\models \ \texttt{A} := \texttt{1} \land \texttt{frame A} : \{\texttt{A} = \texttt{0}\} \ \equiv \ \texttt{false}.$$

Having to detect errors such as this would defeat the whole purpose of frame variables, which is to alleviate unnecessary work.

The scope rule can be enforced by defining a new operator `local` which existentially introduces a new frame variable.

$$\texttt{local}\ v : p \ \stackrel{\text{def}}{=}\ \exists v : \texttt{frame}\ v : p.$$

In an implementation of Tempura, frame variables may only be introduced by means of `local`. The operator `frame` on its own is prohibited. However, `frame` on its own is useful for discussing properties of programs.

6.2.4 Doing Without Frame Variables

Although frame variables are practically useful, their semantic characterisation is quite complex, so it is fortunate that for a restricted class of programs there is a way to keep the inertial semantics without having to deal explicitly with the frame operator. This is achieved by translating inertial programs into a form without the frame operator. The translation process introduces explicit assignments where they are needed to maintain inertia.

The translation function ϕ takes three arguments. The first is the variable v to be framed; the second is a marker variable Δ; and the third is the program p to be translated. The variable Δ is just an ordinary state variable, but it must not be free in p. It has been given special syntax to denote its special role.

The idea of the translation is to make Δ mark the steps on which the variable v changes; that is,

$$\Delta \quad = \quad \delta(v).$$

Once this is done v should be kept stable wherever it is not explicitly changed. So the construct $\texttt{frame}\ v : p$ may be rewritten as

$$\texttt{frame}\ v : p \quad \longrightarrow \quad \exists \Delta : \{\phi(v, \Delta, p) \wedge \texttt{keep if}\ \neg\Delta\ \texttt{then}\ v \multimap v\}.$$

The translation ϕ is defined inductively on the primitive operators, taking unit-assignment, :=, to be primitive since equality cannot in general be handled in an inertial framework.

$$\begin{array}{l}
\phi(v, \Delta, \texttt{empty}) = \texttt{empty} \\
\phi(v, \Delta, v := e) = \Delta \wedge v := e \\
\phi(v, \Delta, v' := e) = \neg\Delta \wedge v' := e \\
\phi(v, \Delta, p \wedge p') = \exists \Delta', \Delta'' : \{\texttt{keep}\,(\Delta = \Delta' \vee \Delta'') \wedge \phi(v, \Delta', p) \wedge \phi(v, \Delta'', p')\} \\
\phi(v, \Delta, \texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ p') = \texttt{if}\ e\ \texttt{then}\ \phi(v, \Delta, p)\ \texttt{else}\ \phi(v, \Delta, p') \\
\phi(v, \Delta, \exists v : p) = \texttt{keep}\,(\neg\Delta) \wedge \exists v : p \\
\phi(v, \Delta, \exists v' : p) = \exists v' : \phi(v, \Delta, p) \\
\phi(v, \Delta, \texttt{next}\ p) = \neg\Delta \wedge \texttt{next}\ \phi(v, \Delta, p) \\
\phi(v, \Delta, p\, ; p') = \phi(v, \Delta, p)\, ; \phi(v, \Delta, p'),
\end{array}$$

where the variable v' is different from v.

The translation does not handle the exceptional cases mentioned above. In particular, it does not work if equality is used for assigning values to frame variables, but I have already given several reasons for regarding such assignments with suspicion. As a result, one cannot claim, as one would like to, that in all cases the rewritten program is equivalent to the original. The best one can hope for is that if the rewritten program "works" then it produces the correct result. In other words, if an interval satisfies the translated program, then it also satisfies the original, but not *vice versa*.

$$\models \quad \exists \Delta : \{\phi(v, \Delta, p) \wedge \texttt{keep if}\ \neg\Delta\ \texttt{then}\ v \multimap v\} \quad \supset \quad \texttt{frame}\ v : p.$$

This has not yet been formally verified.

6.3 Discussion

The frame operator is still experimental, and its properties need to be formally established. Nevertheless, it does seem to work in all the cases where it is supposed to work; that is, for all deterministic Tempura programs. Moreover, a similar technique might be used for other purposes. For example, it might be possible to define the concept of a "default value" in this way.

The frame operator also needs to be put in context. A considerable amount of work has been done on the frame problem in the study of artificial intelligence, and the relationship of the frame operator to this work must be investigated.

6.3.1 Default Values

The technique I used for preferring frame variables to the other variables that satisfy the same formula might equally well be used to choose variables with properties other than stability. For instance, default values can be handled in the same way. Instead of minimising changes in the value of a variable v, one now tries to minimise the times when v does not equal its default value, d say. This might be achieved by using the predicate $\gamma(v,d)$ in place of $\delta(v)$, where

$$\gamma(v,d) \quad \stackrel{\text{def}}{=} \quad \neg(v \mathrel{\circ\!\!=} d).$$

Observe that inertia is a special case of this.

Default values can be used, for example, to formalise the idea that a program produces no output unless the output predicate is used explicitly. One would simply choose the empty list of outputs as the default value for the signal `Output`. Perhaps a similar idea can be made to work for values that decay gradually over time, such as the electrical charge on a capacitor.

6.3.2 The Frame Problem in Artificial Intelligence

For many years the so-called frame problem has been a subject of considerable interest in AI. It was first described by McCarthy and Hayes [MH81] in the context of their *situation calculus*. In the situation calculus one describes the world in terms of *situations*, which are "snapshots" of its state at various points in time,[1] and *actions* which cause a change of situation. Roughly speaking the problem is that most things are left unchanged by most actions, yet to represent this in a naive way one needs to include a number of *frame axioms* each one asserting that some action does not affect some aspect of the situation. For example, drinking a cup of coffee does not (usually) affect the colour of your eyes. Clearly, for any problem of

[1] Situations are like the frames in an animated picture, hence the *frame problem*.

significant complexity the number of such frame axioms would be huge. It seems that their ought to be a better way to represent this information.

Over the years a number of partial solutions to the frame problem have been proposed. McCarthy's idea of *circumscription* was one of the earliest attempts to solve the problem. McCarthy's idea was to introduce a special circumscription axiom which effectively plays the same role as my frame operator. In McCarthy's original version of the circumscription axiom,[2] the effect of circumscribing a formula P with free variables x in a theory A was as follows:

$$A(P) \wedge \nexists p : (A(p) \wedge p \lhd P),$$

where p is a predicate variable with free variables x, and $p \lhd P$ here stands for

$$\forall x : ((px) \supset (Px)) \wedge \neg \forall x : ((Px) \supset (px)).$$

The circumscription axiom encodes a preference for particular models, namely those which are minimal in the above sense. For example, if the formula `can_fly(x)` is circumscribed in the theory $\{\forall \mathtt{b} : (\mathtt{bird(b)} \supset \mathtt{can_fly(b)}), \mathtt{bird(polly)}\}$ one may conclude that the *only* object that can fly is `polly`. Shoham [Sho88] reviews some of the other approaches that have been tried, and also make his own proposal based on what he calls the logic of *chronological ignorance*, a kind of non-monotonic logic.

Clearly, the frame problem is substantially similar to the problem of representing inertial variables, though the latter is more sharply focussed, and some of the proposed solutions seem to be similar in spirit to my frame operator. However, the exact relationship of one to the other remains to be established.

[2] I am borrowing from Shoham's account here [Sho88].

Chapter 7

Recursion and Iteration

In this chapter I discuss three problems which are naturally solved by recursion and show that each may be transformed into a provably equivalent iterative form. Section 7.1 presents that classic example of recursion, The Towers of Hanoi; section 7.2 discusses a parallel summation algorithm; and section 7.3 describes a parallel mergesort algorithm that uses a variant of Batcher's odd-even merge [Bat68]. It is shown that the summation and the merge algorithms have essentially the same structure, and may be transformed to iterative form in the same way.

Many computational problems are most naturally solved recursively by dividing them up into similar subproblems. Often it is much easier to show that a recursive program correctly solves the problem in hand than it is to verify the corresponding iterative solution. Nevertheless, it is a fact of life that on conventional sequential computers an iterative program can frequently be made to run much more efficiently than its recursive counterpart. Some programming languages for these machines do not even permit recursion. It is therefore desirable to have ways of transforming recursive programs into equivalent iterative ones.[1]

In this chapter I present three problems which are best solved by recursion. The first is to solve the Towers of Hanoi puzzle, the second is to sum a list of numbers in parallel, and the third is to sort a list using a variant of Batcher's odd-even merge [Bat68]. All the recursive solutions are easily shown to be correct, and can be transformed into provably equivalent iterative forms. What is more, such transformations require only the rules of logic, there is no need to introduce explicit stacks or similar operational devices.

7.1 The Towers of Hanoi

The Towers of Hanoi is well-known both as a mathematical diversion and as a children's toy. The toy comprises an arrangement of three pegs and a set of discs.

[1] Since iteration is defined recursively, such transformations do not formally eliminate recursion. What they do is to transform one program into another that is better suited to a conventional machine architecture.

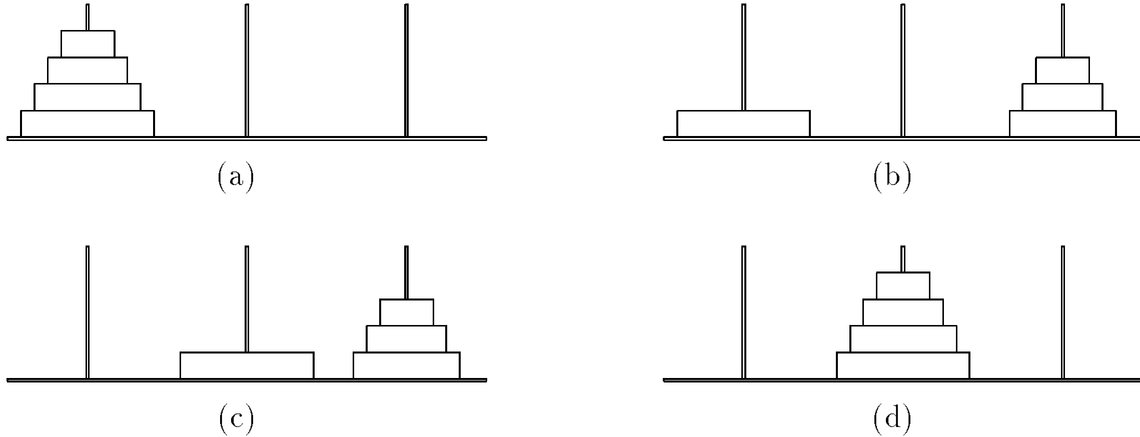


Figure 7.1: The Towers of Hanoi. The steps in the solution for a four-disc tower are shown: (a) the initial configuration; (b) after moving the three-disc tower to the auxiliary peg; (c) after moving the bottom disc to the destination peg; and (d) after moving the three-disc tower back to the destination peg.

The discs, no two of which have the same diameter, are drilled through their centres so that they may be stacked onto the pegs to form towers, as shown in figure 7.1. Initially, the discs are all on one peg and arranged so that they form a "tower" decreasing in size towards the top. From this position the discs are to be moved one at a time from peg to peg with the aim of forming a tower identical to the initial one but on another of the pegs. The only rule to be observed is that a disc may never be placed on top of one smaller than itself; in other words, at any point in time the stack of discs on each peg must decrease in diameter towards the top. Let us represent the whole contrivance as a list of pegs, P say, and each peg as a list of numbers with each number standing for one of the discs on that peg. The head of each list is the top disc on that peg, and a smaller number denotes a smaller disc.

Suppose that a tower of n discs is to be moved from peg P_0 to peg P_1 using P_2 as an auxiliary peg, then a solution, hanoi(n), must satisfy three properties. First, it must produce the desired final result from the given initial conditions;

$$\models \ \mathtt{hanoi(n)} \ \supset \ \mathtt{P_0}, \mathtt{P_1}, \mathtt{P_2} = [\mathtt{0..n}], [\,], [\,] \wedge \mathtt{P_0}, \mathtt{P_1}, \mathtt{P_2} \leftarrow [\,], [\mathtt{0..n}], [\,] \tag{7.1}$$

Second, it must ensure that all the pegs remain in order of increasing size,

$$\models \ \mathtt{hanoi(n)} \ \supset \ \mathtt{forall\ i} < 3 : \mathtt{always\ ordered(P_i)}, \tag{7.2}$$

where the function ordered(L) tests whether or not the elements of L are in ascending order (it was defined on page 20). Finally, it must ensure that exactly one disc

is moved at each step. This means that on each step the top disc from one of the pegs, $\mathtt{P_i}$ say, must be transferred to another of the pegs, $\mathtt{P_j}$ say, whilst the remaining peg, $\mathtt{P_k}$, is kept stable; that is,

$$\models \ \mathtt{hanoi(n)} \ \supset \ \mathtt{loop} \ \exists \mathtt{i,j,k}: \{ \ \mathtt{i} < 3 \wedge \mathtt{j} < 3 \wedge \mathtt{k} < 3 \wedge \ \mathtt{i} \neq \mathtt{j} \wedge \mathtt{j} \neq \mathtt{k} \wedge \mathtt{k} \neq \mathtt{i} \wedge \ \mathtt{P_i, P_j, P_k} := \mathtt{tl(P_i), cons(hd(P_i), P_j), P_k} \ \}, \tag{7.3}$$

where the list functions have their usual meanings: $\mathtt{hd(L)}$ and $\mathtt{tl(L)}$ denote the head and tail of the list $\mathtt{L}$, and $\mathtt{cons(x, L)}$ returns a list whose head is $\mathtt{x}$ and whose tail is $\mathtt{L}$. These functions were defined on page 22.

7.1.1 Recursive Algorithm

This problem has an elegant recursive solution but is rather harder to analyse as an iterative procedure. In the earliest published discussion of which I am aware, Rouse-Ball [RB92] gives a mathematical analysis of the problem for a fixed number of discs. A recent, and very extensive, consideration of the problem from a computational point of view is given by Rohl [Roh87].

The recursive solution is discovered by hierarchical decomposition of the specification (7.1). First, the initialisation part may be separated from the transfer:

$$\mathtt{P_0, P_1, P_2} = [0..\mathtt{n}], [\,], [\,] \wedge \mathtt{move_tower(n, 0, 1, 2)} \ \supset \ \mathtt{hanoi(n)},$$

where the predicate $\mathtt{move_tower(n, f, t, a)}$ solves the problem of moving a tower of $\mathtt{n}$ discs from $\mathtt{P_f}$ to $\mathtt{P_t}$ using the auxiliary peg $\mathtt{P_a}$. Its function is to remove the top $\mathtt{n}$ discs from $\mathtt{P_f}$ and append them to $\mathtt{P_t}$, leaving $\mathtt{P_a}$ unchanged. Assuming that there are at least $\mathtt{n}$ discs on $\mathtt{P_f}$ initially, $\mathtt{move_tower}$ must satisfy

$$\models \ \mathtt{move_tower(n, f, t, a)} \ \supset \ \mathtt{P_f, P_t, P_a} \leftarrow \mathtt{drop(n, P_f), take(n, P_f)\hat{\ }P_t, P_a}, \tag{7.4}$$

where $\mathtt{take(n, L)} \stackrel{\text{def}}{=} \mathtt{L_{0..n}}$ and $\mathtt{drop(n, L)} \stackrel{\text{def}}{=} \mathtt{L_{n..|L|}}$. Now the property (7.4) may be decomposed recursively.

If there are no discs there is nothing to do. Otherwise, suppose that the problem is solved for $\mathtt{n}$ discs, then it is easy to solve the $\mathtt{n} + 1$ disc case by simply moving the top $\mathtt{n}$ discs from the initial peg $\mathtt{P_f}$ to the auxiliary peg $\mathtt{P_a}$ using the $\mathtt{n}$ disc solution, then moving the remaining disc to the destination peg $\mathtt{P_t}$, and then repeating the $\mathtt{n}$ disc solution to move the discs back from the auxiliary peg to the destination peg. The steps in the solution for three discs are illustrated in figure 7.1. In this way the solution for $\mathtt{n} + 1$ discs is decomposed into an $\mathtt{n}$-disc solution followed by a single step followed by another $\mathtt{n}$-disc solution:

$$\models \ \left\{ \begin{array}{l} \mathtt{move_tower(n, f, a, t)}; \\ \mathtt{move_disc(f, t, a)}; \\ \mathtt{move_tower(n, a, t, f)} \end{array} \right\} \ \supset \ \mathtt{move_tower(n+1, f, t, a)}, \tag{7.5}$$

time	P_0	P_1	P_2
0	[0, 1, 2, 3]	[]	[]
1	[1, 2, 3]	[]	[0]
2	[2, 3]	[1]	[0]
3	[2, 3]	[0, 1]	[]
4	[3]	[0, 1]	[2]
5	[0, 3]	[1]	[2]
6	[0, 3]	[]	[1, 2]
7	[3]	[]	[0, 1, 2]
8	[]	[3]	[0, 1, 2]
9	[]	[0, 3]	[1, 2]
10	[1]	[0, 3]	[2]
11	[0, 1]	[3]	[2]
12	[0, 1]	[2, 3]	[]
13	[1]	[2, 3]	[0]
14	[]	[1, 2, 3]	[0]
15	[]	[0, 1, 2, 3]	[]

Figure 7.2: The solution to the Towers of Hanoi problem for 4 discs.

where the function of move_disc(f, t, a) is to transfer the top disc from P_f to P_t.

$$\texttt{move_disc}(f, t, a) \stackrel{\text{def}}{=} P_f, P_t, P_a := \texttt{tl}(P_f), \texttt{cons}(\texttt{hd}(P_f), P_t), P_a.$$

It satisfies property (7.3) and also preserves the ordering of the pegs, as required by property (7.2). Thus, solutions can be constructed for all values of n, and move_tower is defined as follows:

```
move_tower(n, f, t, a)  ≝  if n = 0 then empty
                           else {
                             move_tower(n − 1, f, a, t);
                             move_disc(f, t, a);
                             move_tower(n − 1, a, t, f)
                           }.
```

A call of hanoi(4) generates the complete sequence of moves to transfer a tower of 4 discs from peg 0 to peg 1, as shown in figure 7.2.

7.1.2 Transformation

The program can be transformed in a number of ways. First, notice that there is no need to specify the peg from which a disc is to be moved; the identity of the

disc is known, so its position can be deduced by examining the pegs. A little more thought reveals that there is no need to specify the destination peg either, because the direction of the move suffices. The direction of each move is either positive ($+1$) or negative (-1) modulo 3, and disc $\mathtt{n-1}$ moves in the same direction as the $\mathtt{n}$-disc tower of which it is the base. The resulting program has the form:

$$
\begin{array}{rcl}
\mathtt{hanoi'(n)} & \stackrel{\text{def}}{=} & \mathtt{P_0, P_1, P_2 = [0..n], [\,], [\,] \wedge move_tower'(n, +1)} \\
\mathtt{move_tower'(n, d)} & \stackrel{\text{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\
& & \mathtt{else\ \{} \\
& & \quad \mathtt{move_tower'(n-1, -d);} \\
& & \quad \mathtt{move_disc'(n-1, d);} \\
& & \quad \mathtt{move_tower'(n-1, -d)} \\
& & \mathtt{\}} \\
\mathtt{move_disc'(i, d)} & \stackrel{\text{def}}{=} & \exists \mathtt{f, t, a : \{} \\
& & \quad \mathtt{forall\ p \in P : \{if\ i \in p\ then\ f \Leftarrow p\};} \\
& & \quad \mathtt{t \Leftarrow (f + d)\ mod\ 3;} \\
& & \quad \mathtt{a \Leftarrow (t + d)\ mod\ 3;} \\
& & \quad \mathtt{move_disc(f, t, a)} \\
& & \mathtt{\}.}
\end{array}
$$

The predicate $\mathtt{hanoi'}$ can be further transformed by observing that disc number $\mathtt{n-1-i}$ is always moved in direction $(-1)^{\mathtt{i}} \times \mathtt{d(n)}$, where $\mathtt{d(n)}$ is the direction of the whole transfer (the direction in which the largest disc is moved), and that the direction of each move can therefore be deduced. This results in the program:

$$
\begin{array}{rcl}
\mathtt{hanoi''(n)} & \stackrel{\text{def}}{=} & \mathtt{P_0, P_1, P_2 = [0..n], [\,], [\,] \wedge move_tower''(n)} \\
\mathtt{move_tower''(n)} & \stackrel{\text{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\
& & \mathtt{else\ \{} \\
& & \quad \mathtt{move_tower''(n-1);} \\
& & \quad \mathtt{move_disc''(n-1);} \\
& & \quad \mathtt{move_tower''(n-1)} \\
& & \mathtt{\}} \\
\mathtt{move_disc''(i)} & \stackrel{\text{def}}{=} & \mathtt{move_disc'(i, dir(i))} \\
\mathtt{dir(i)} & \stackrel{\text{def}}{=} & \mathtt{if\ (n-1-i)\ mod\ 2 = 0\ then\ +1\ else\ -1.}
\end{array}
$$

The three programs $\mathtt{hanoi}$, $\mathtt{hanoi'}$ and $\mathtt{hanoi''}$ define identical behaviour, and they are provably equivalent; that is,

$$
\begin{array}{lcl}
\models \ \mathtt{hanoi(n)} & \equiv & \mathtt{hanoi'(n)} \\
\models \ \mathtt{hanoi'(n)} & \equiv & \mathtt{hanoi''(n).}
\end{array}
$$

However, it is easier to derive an iterative solution from the last form, $\mathtt{hanoi''(n)}$.

7.1.3 Iterative Algorithm

To obtain an iterative solution, one should first notice that the solution is nothing but a sequence of individual moves! In other words, the predicate $\mathtt{move_tower''(n)}$ is equivalent to an iterative program of the form

$$\models \ \mathtt{move_tower''(n)} \ \equiv \ \mathtt{for\ i < t(n)\ do\ move_disc''(s(i))} \tag{7.6}$$

for some functions $\mathtt{s}$ and $\mathtt{t}$. The function $\mathtt{t(n)}$ determines the number of steps taken to move $\mathtt{n}$ discs, and $\mathtt{s(i)}$ determines which disc is to be moved on step $\mathtt{i}$. Substitution of the iterative form (7.6) into the definition of $\mathtt{move_tower''(n)}$ imposes the following conditions on $\mathtt{t}$ and $\mathtt{s}$:

$$\begin{array}{rcl} \mathtt{t(0) = 0} & \text{and} & \mathtt{t(n+1) = 2 \times t(n) + 1}, \\ \mathtt{s(t(n)) = n} & \text{and} & \mathtt{forall\ i < t(n) : s(i + t(n) + 1) = s(i)}, \end{array}$$

from which it may be deduced that $\mathtt{t(n)} = 2^{\mathtt{n}-1}$ and $\mathtt{s(i)}$ determines the position of the least significant zero in the binary expansion of $\mathtt{i}$. Let us call this function $\mathtt{ls0}$, then an iterative solution to the Towers of Hanoi problem has the form

$$\mathtt{hanoi_i(n)} \ \stackrel{\text{def}}{=} \ \mathtt{P_0, P_1, P_2 = [0..n], [\,], [\,] \wedge for\ i < 2^{n-1}\ do\ move_disc''(ls0(i))},$$

with $\mathtt{move_disc''}$ defined as above, and $\mathtt{ls0}$ defined as follows:[2]

$$\mathtt{ls0(i)} \ \stackrel{\text{def}}{=} \ \mathtt{if\ i\ mod\ 2 = 0\ then\ 0\ else\ 1 + ls0(i/2)}.$$

The predicate $\mathtt{hanoi_i(n)}$ has been tested in Tempura and proved equivalent to $\mathtt{hanoi''(n)}$ using the HOL theorem prover.

7.2 Parallel Summation

This section describes an algorithm to sum a list of numbers. In order to simplify the discussion it is assumed that the list, $\mathtt{A}$ say, has $2^{\mathtt{n}}$ elements, though it would not be hard to generalise the algorithm. The algorithm, $\mathtt{lr_sum(n, A)}$, works *in situ* by summing the left and right halves of the list $\mathtt{A}$ in parallel, and assigns the final sum to the first element of the list, $\mathtt{A_0}$. Thus, the program must satisfy

$$\models \ \mathtt{lr_sum(n, A)} \ \supset \ \mathtt{A_0 \leftarrow sum(A)},$$

where the function $\mathtt{sum(A)}$ forms the sum of the elements of $\mathtt{A}$,

$$\mathtt{sum(A)} \ \stackrel{\text{def}}{=} \ \mathtt{if\ |A| = 0\ then\ 0\ else\ hd(A) + sum(tl(A))}.$$

The resulting parallel summation algorithm is quite well known. It is derived using the so-called "divide-and-conquer" strategy.

[2] The function ls0 can also be defined without explicit recursion.

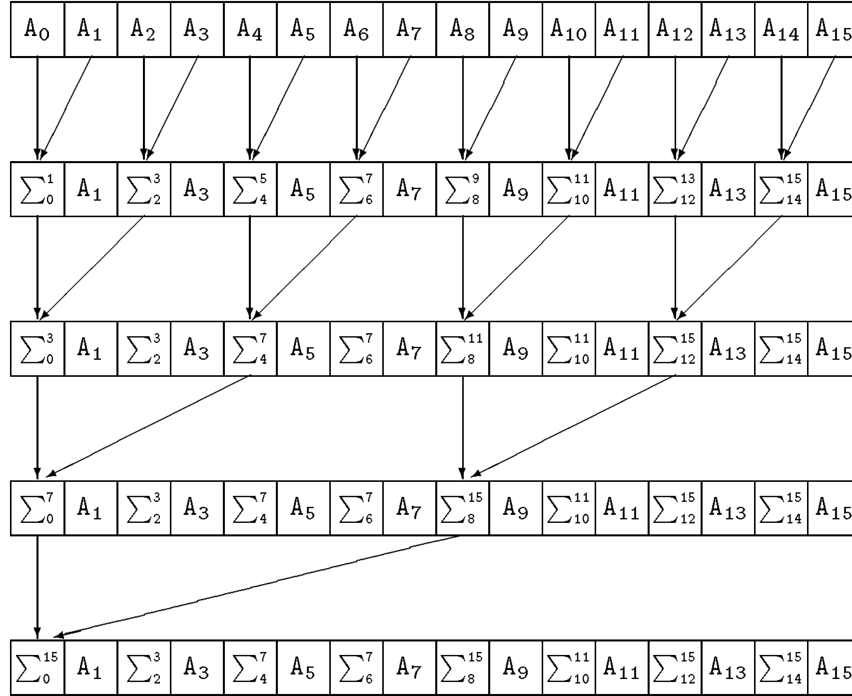


Figure 7.3: Parallel summation of a list A by recursive division into left and right sublists.

7.2.1 Recursive Algorithm

The summation may be divided into subtasks recursively. An algorithm of this type follows from the decomposition theorem below, which holds for all state expressions e, e_0, e'_0, e_1, e'_1 and binary functions f.

$$\models \quad e_0, e_1 \leftarrow e'_0, e'_1 \,;\, e \leftarrow f(e_0, e_1) \quad \supset \quad e \leftarrow f(e'_0, e'_1). \tag{7.7}$$

Taking f to be the addition function and using the commutativity and associativity of addition,

$$\models \quad \mathtt{sum}(A\hat{\ }B) \quad = \quad \mathtt{sum}(A) + \mathtt{sum}(B),$$

permits the summation to be split apart. For instance, instantiating the terms in (7.7) in the following way:

$$\models \quad A_0, A_1 \leftarrow A_0, \mathtt{sum}(\mathtt{tl}(A)) \,;\, A_0 \leftarrow A_0 + A_1 \quad \supset \quad A_0 \leftarrow A_0 + \mathtt{sum}(\mathtt{tl}(A))$$

leads to a sequential algorithm. On the other hand, the substitutions below lead to a parallel algorithm which sums the left and right halves of A, lt(A) and rt(A), simultaneously.

$$\models \quad \left\{ \begin{matrix} A_0, A_{2^n} \leftarrow \mathtt{sum}(\mathtt{lt}(A)), \mathtt{sum}(\mathtt{rt}(A)); \\ A_0 \leftarrow A_0 + A_{2^n} \end{matrix} \right\} \quad \supset \quad A_0 \leftarrow \mathtt{sum}(\mathtt{lt}(A)) + \mathtt{sum}(\mathtt{rt}(A)),$$

time	A
0	[1, 1, 1, 1, 1, 1, 1, 1]
1	[2, 1, 2, 1, 2, 1, 2, 1]
2	[4, 1, 2, 1, 4, 1, 2, 1]
3	[8, 1, 2, 1, 4, 1, 2, 1]

(a)

time	A
0	[1, 1, 1, 1, 1, 1, 1, 1]
1	[2, 2, 2, 2, 1, 1, 1, 1]
2	[4, 4, 2, 2, 1, 1, 1, 1]
3	[8, 4, 2, 2, 1, 1, 1, 1]

(b)

Figure 7.4: Parallel summation using (a) left and right partitions and (b) even and odd partitions

---

where $\mathtt{lt(A)} \stackrel{\text{def}}{=} \mathtt{A}_{0..|\mathtt{A}|/2}$ and $\mathtt{rt(A)} \stackrel{\text{def}}{=} \mathtt{A}_{|\mathtt{A}|/2..|\mathtt{A}|}$. This gives a refined specification for `lr_sum` as follows:

$$
\begin{array}{lll}
\models \ \mathtt{lr_sum(n,A)} & \supset & \mathtt{if\ n = 0\ then\ A_0 \leftarrow A_0} \\
& & \mathtt{else}\ \{ \\
& & \quad \mathtt{A_0, A_{2^{n-1}} \leftarrow sum(lt(A)), sum(rt(A));} \\
& & \quad \mathtt{A_0 \leftarrow A_0 + A_{2^{n-1}}} \\
& & \}.
\end{array}
$$

The case $\mathtt{n} = 0$ must be treated separately since the sum cannot then be split apart.

Replacing the partial sums by recursive calls, and assigning lengths to the various subintervals results in a suitable definition for `lr_sum`:

$$
\begin{array}{lll}
\mathtt{lr_sum(n,A)} & \stackrel{\text{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\
& & \mathtt{else}\ \{ \\
& & \quad \mathtt{lr_sum(n-1, lt(A)) \land lr_sum(n-1, rt(A));} \\
& & \quad \mathtt{A_0 := A_0 + A_{2^{n-1}}} \\
& & \}.
\end{array}
$$

The general strategy is shown in figure 7.3 and figure 7.4(a) shows how this program works for the list [1, 1, 1, 1, 1, 1, 1, 1].

7.2.2 Iterative Algorithm

An iterative form of the summation algorithm can be discovered in much the same way as the iterative algorithm for the Towers of Hanoi problem. The iterative summation algorithm is just a sequence of parallel sum steps; that is,

$$
\begin{array}{lll}
\models \ \mathtt{lr_sum(n,A)} & \equiv & \mathtt{for\ i < t(n)\ do} \\
& & \quad \mathtt{forall\ j < p(n,i) :} \\
& & \qquad \mathtt{A_{a(i,j)} := A_{a(i,j)} + A_{b(i,j)}}
\end{array}
$$

for some functions $\mathtt{t}$, $\mathtt{p}$, $\mathtt{a}$ and $\mathtt{b}$. As before, these functions are determined by substituting the iterative form into the recursive definition of `lr_sum` to get:

$$\mathtt{t(n)} = \mathtt{n}, \quad \mathtt{p(n,i)} = 2^{\mathtt{n}-1-\mathtt{i}}, \quad \mathtt{a(i,j)} = \mathtt{j} \times 2^{\mathtt{i}+1} \quad \text{and} \quad \mathtt{b(i,j)} = 2^{\mathtt{i}} + \mathtt{j} \times 2^{\mathtt{i}+1}.$$

The result is an iterative algorithm that is logically equivalent to the recursive version. However, let us take a slightly different course and derive a general form of the summation algorithm that will be useful later on.

7.2.3 General Algorithm

Because summation is insensitive to the order and grouping of terms, there is no compelling reason to sum the elements of $\mathtt{A}$ in any particular order. The summation can be divided by partitioning $\mathtt{A}$ into any two equally sized disjoint lists at each step and summing each in parallel; that is,

$$\models \quad \mathtt{sum(A)} \quad = \quad \mathtt{sum(lptn(A))} + \mathtt{sum(rptn(A))}$$

for any two functions `lptn` and `rptn` with the property that `lptn(A)^rptn(A)` is a permutation of $\mathtt{A}$. In the derivation of `lr_sum` above, `lptn` and `rptn` were taken to be the functions `lt` and `rt`, which partition a list into left and right halves.

However, different partitions suit different implementation environments. For instance, $\mathtt{A}$ may be partitioned into even and odd elements by taking $\mathtt{lptn} = \mathtt{evns}$ and $\mathtt{rptn} = \mathtt{odds}$, where

$$\begin{array}{lcl} \mathtt{evns(A)} & \stackrel{\text{def}}{=} & \mathtt{if}\ |\mathtt{A}| < 2\ \mathtt{then\ A\ else\ cons}(\mathtt{A}_0, \mathtt{evns}(\mathtt{A}_{2..|\mathtt{A}|})) \\ \mathtt{odds(A)} & \stackrel{\text{def}}{=} & \mathtt{if}\ |\mathtt{A}| < 2\ \mathtt{then}\ [\,]\ \mathtt{else\ cons}(\mathtt{A}_1, \mathtt{odds}(\mathtt{A}_{2..|\mathtt{A}|})). \end{array}$$

This leads to an algorithm that is suited to certain types of processor array, as will be shown in chapter 8.

In general, the parallel summation algorithm is defined on a selection $\mathtt{s} = [\mathtt{s}_0, \ldots, \mathtt{s}_{2^\mathtt{n}}]$ of $2^\mathtt{n}$ elements from $\mathtt{A}$ as follows:

$$\begin{array}{lcl} \mathtt{par_sum(n,s,A)} & \stackrel{\text{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\ & & \mathtt{else}\ \{ \\ & & \quad \mathtt{par_sum(n-1, lptn(s), A)} \wedge \mathtt{par_sum(n-1, rptn(s), A)}; \\ & & \quad \mathtt{sum_step(s, A)} \\ & & \}, \end{array}$$

where `sum_step` is the single step operation

$$\mathtt{sum_step(s,A)} \stackrel{\text{def}}{=} \mathtt{A}_{\mathtt{hd(lptn(s))}} := \mathtt{A}_{\mathtt{hd(lptn(s))}} + \mathtt{A}_{\mathtt{hd(rptn(s))}}.$$

You may check that the algorithm `lr_sum` above is a special case of `par_sum` with $\mathtt{lptn} = \mathtt{lt}$ and $\mathtt{rptn} = \mathtt{rt}$. As you might expect, the iterative form of this general algorithm is similar to the iterative form of `lr_sum`, and is derived in a similar way.

Indeed, it may be proved that *any* recursive algorithm of the same form as `par_sum` is equivalent to an iterative algorithm of the following form:

$$\mathtt{par_sum_i(n,s,A)} \stackrel{\text{def}}{=} \begin{array}{l} \mathtt{for\ i < n\ do} \\ \quad \mathtt{forall\ j < 2^{n-1-i}:} \\ \quad\quad \mathtt{sum_step(i, apply(n-1-i, j, lptn, rptn, s), A),} \end{array}$$

and the function `apply(i, j, lptn, rptn, s)` produces a sequence of applications of the functions `lptn` and `rptn` to `s` based on the `i`-bit binary representation of `j`.

Shown below are the partitions produced by `apply` for the first few values of `i` and `j` (the binary representation of `j` is shown). The general pattern is easily seen.

i	j	apply(i, j, lptn, rptn, s)
0	0	s
1	0	lptn(s)
1	1	rptn(s)
2	00	lptn(lptn(s))
2	01	rptn(lptn(s))
2	10	lptn(rptn(s))
2	11	rptn(rptn(s))

A recursive definition of `apply(i, j, lptn, rptn, s)` is given below. It tests each bit of `j` in turn and applies the appropriate partition function.

$$\mathtt{apply(i,j,lptn,rptn,s)} \stackrel{\text{def}}{=} \begin{array}{l} \mathtt{if\ i = 0\ then\ s} \\ \mathtt{else} \\ \quad \mathtt{if\ even(j)} \\ \quad \mathtt{then\ lptn(apply(i-1, j/2, lptn, rptn, s))} \\ \quad \mathtt{else\ rptn(apply(i-1, j/2, lptn, rptn, s)).} \end{array}$$

For example, $\mathtt{apply(2, 0, lt, rt, [0..2^n])} = [0..2^{n-2}]$, and for arbitrary `i` and `j` the left-right and even-odd partitions denote the following lists:

$$\begin{array}{rcl} \mathtt{apply(i, j, lt, rt, [0..2^n])} & = & [\mathtt{j} \times 2^{n-i} \mathrel{..} (\mathtt{j}+1) \times 2^{n-i}] \\ \mathtt{apply(i, j, evns, odds, [0..2^n])} & = & [\mathtt{j}, \mathtt{j}+2^{i}, \ldots, \mathtt{j} + (2^{n-i}-1) \times 2^{i}]. \end{array}$$

The two iterative summation algorithms corresponding to these partitions are `lr_sum_i`, which is equivalent to `lr_sum`, and `eo_sum_i`, which sums even and odd elements separately.

$$\begin{array}{rcl} \mathtt{lr_sum_i(n,A)} & \stackrel{\text{def}}{=} & \begin{array}{l} \mathtt{for\ i < n\ do} \\ \quad \mathtt{forall\ j < 2^{n-1-i}:} \\ \quad\quad A_{j\times 2^{i+1}} := A_{j\times 2^{i+1}} + A_{2^i + j\times 2^{i+1}}, \end{array} \\ \mathtt{eo_sum_i(n,A)} & \stackrel{\text{def}}{=} & \begin{array}{l} \mathtt{for\ i < n\ do} \\ \quad \mathtt{forall\ j < 2^{n-1-i}:} \\ \quad\quad A_j := A_j + A_{j+2^{n-1-i}}. \end{array} \end{array}$$

An implementation of the latter algorithm, `eo_sum_i`, is described in chapter 8, and how it works is illustrated in figure 7.4(b).

7.3 Mergesort

This section concerns mergesort, another recursive algorithm with the same structure as the parallel summation algorithm. The basic mergesort algorithm is quite straightforward. Its specification is to sort a list A *in situ*, where A is assumed to be of length 2^n, as before; that is,

$$\models \ \mathtt{mergesort(n,A)} \ \supset \ \mathtt{A} \leftarrow \mathtt{sort(A)}.$$

The function sort(A) returns a sorted version of A; it could be the simple mergesort function defined in chapter 2.

7.3.1 Recursive Mergesort

The specification may be decomposed as in the previous example, using theorem (7.7) with appropriate substitutions, to give

$$\models \ \left\{\begin{array}{l}\mathtt{lt(A)} \leftarrow \mathtt{sort(lt(A))} \\ \ \wedge \mathtt{rt(A)} \leftarrow \mathtt{sort(rt(A))}; \\ \mathtt{A} \leftarrow \mathtt{merge(lt(A),rt(A))}\end{array}\right\} \ \supset \ \mathtt{A} \leftarrow \mathtt{merge(sort(lt(A)),sort(rt(A)))},$$

where the function merge is assumed to have the property that

$$\models \ \mathtt{merge(sort(A),sort(B))} \ = \ \mathtt{sort(A\hat{\ }B)}$$

for lists A and B. A recursive definition was given on page 21.

The above decomposition leads directly to a prototype mergesort in just the same way as the parallel summation example.

$$\begin{array}{lcl}\mathtt{mergesort(n,A)} & \stackrel{\mathrm{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\ & & \mathtt{else}\ \{ \\ & & \quad \mathtt{mergesort(n-1,lt(A))} \wedge \mathtt{mergesort(n-1,rt(A))}; \\ & & \quad \mathtt{A := merge(lt(A),rt(A))} \\ & & \}.\end{array}$$

The prototype mergesort is of the same form as the parallel summation algorithm, par_sum and may be transformed into an equivalent iterative form in just the same way. Thus,

$$\begin{array}{rcl}\mathtt{mergesort_i(n,A)} & \stackrel{\mathrm{def}}{=} & \mathtt{for\ i < n\ do} \\ & & \quad \mathtt{forall\ j < 2^{n-1-i}:} \\ & & \qquad \mathtt{A := merge(seg(A,2\times j,2^{i}),seg(A,2\times j+1,2^{i}))} \\ \mathtt{seg(A,i,d)} & \stackrel{\mathrm{def}}{=} & \mathtt{A_{i\times d..(i+1)\times d}}.\end{array}$$

Figure 7.5 shows graphically how this algorithm works on a list of 128 elements, initially in decreasing order. But the heart of the algorithm is really in the way the merge is performed.

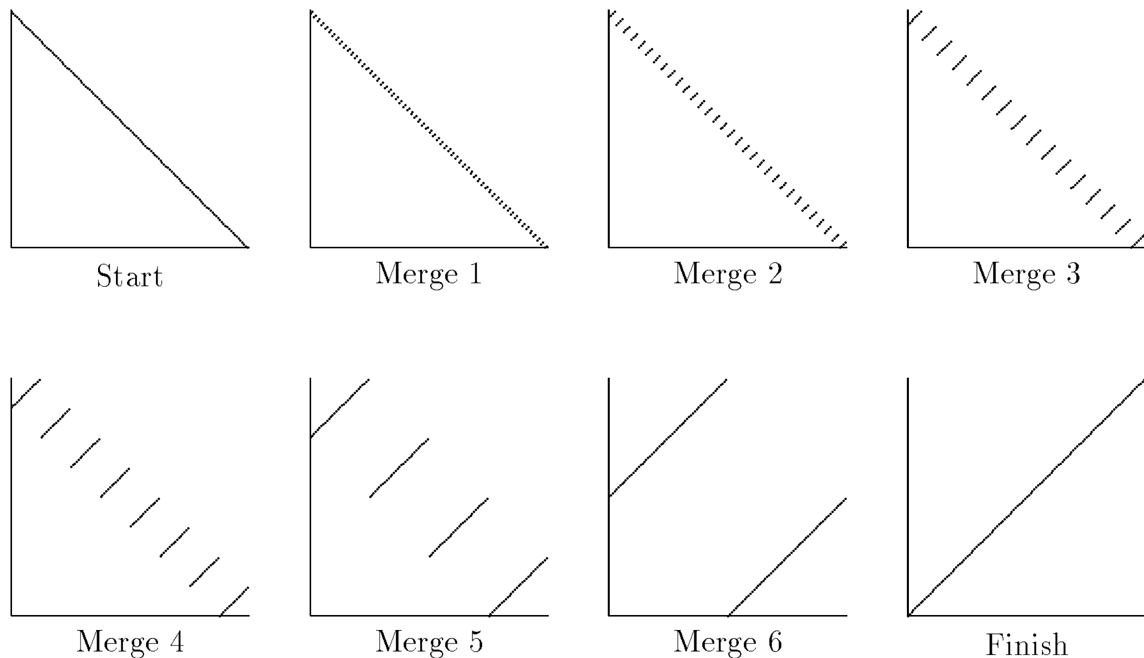


Figure 7.5: Operation of the mergesort on a list `A` of `128` elements initially in decreasing order. The points $(\mathtt{i}, \mathtt{A_i})$ are marked to show the value held in each list element, so the line of unit slope in the final graph indicates that the list is sorted.

7.3.2 The Merge Algorithm

A number of parallel merging algorithms are based on the divide-and-conquer approach; the one presented here is a variation of Batcher's odd-even merge algorithm [Bat68]. This algorithm is chosen because it is suitable for implementation on a SIMD processor array, as will be shown in chapter 8. Batcher's original algorithm requires fewer comparisons, but does not transfer so readily to the most common types of processor array. The two algorithms take the same number of steps to complete.

The merge algorithm, `eo_merge` say, is to take as input a list `A` of $\mathtt{2^n}$ elements whose left and right halves are in order, and to merge the two halves *in situ* to produce a sorted list. The central idea of this algorithm is that the merger can be achieved by first merging the odd elements of the left half of `A` with the even elements of the right and in parallel merging the evens of the left with the odds of the right, and then comparing each even element of the resultant list with the next odd one, exchanging them if they are out of order.

To see why this works, consider a specific example. Suppose that the list `A` has

eight elements and that each half is initially in order; that is, both $\mathtt{ordered(lt(A))}$ and $\mathtt{ordered(rt(A))}$ are initially true. The partition described above separates $\mathtt{A}$ into two sublists,

$$\begin{array}{lcllllllllll} \mathtt{oe(A)} & = & \mathtt{odds(lt(A))\hat{\ }evns(rt(A))} & \to & & \mathtt{A_1} & & \mathtt{A_3} & \mathtt{A_4} & & \mathtt{A_6} & \\ \mathtt{eo(A)} & = & \mathtt{evns(lt(A))\hat{\ }odds(rt(A))} & \to & \mathtt{A_0} & & \mathtt{A_2} & & & \mathtt{A_5} & & \mathtt{A_7} \end{array}$$

The left and right halves of each sublist are already ordered because the left and right halves of the whole list are; therefore the sublists can themselves be sorted by recursively merging.

Now consider what happens when the left and right halves of $\mathtt{A}$ are merged to form a single ordered list, $\mathtt{A'}$ say. The first element of $\mathtt{A'}$ must be the lesser of $\mathtt{A_0}$ and $\mathtt{A_4}$, because the left and right halves are already sorted. Suppose that it is $\mathtt{A_0}$, then the next element of the sorted list, $\mathtt{A'_1}$, is either $\mathtt{A_1}$ or $\mathtt{A_4}$. If it is $\mathtt{A_4}$ then the next one must be $\mathtt{A_1}$ or $\mathtt{A_5}$, and if it is $\mathtt{A_1}$ then the next must be $\mathtt{A_2}$ or $\mathtt{A_4}$, and so on. Observe that every even-odd pair in $\mathtt{A'}$ contains one element from each of the two sublists, $[\mathtt{A_1}, \mathtt{A_3}, \mathtt{A_4}, \mathtt{A_6}]$ and $[\mathtt{A_0}, \mathtt{A_2}, \mathtt{A_5}, \mathtt{A_7}]$.

If these sublists are themselves sorted into order the elements of $\mathtt{A'}$ are found by alternately "peeling off" one element from one sublist followed by one element from the other. In other words, each even element of the sorted list, $\mathtt{A'_{2\times i}}$, is simply the lesser of $\mathtt{A_{2\times i}}$ and $\mathtt{A_{2\times i+1}}$, and the corresponding odd element, $\mathtt{A'_{2\times i+1}}$, is the greater of the two. This result generalises to other values of $\mathtt{n}$.

Since the odd and even elements in each half of $\mathtt{A}$ are themselves ordered initially (because each half is ordered), the two sublists of odd and even elements may be ordered by merging. Once again, therefore, theorem (7.7) applies, and

$$\models \left\{ \begin{array}{l} \mathtt{oe(A)} \leftarrow \mathtt{merge_oe(A)} \\ \ \wedge\ \mathtt{eo(A)} \leftarrow \mathtt{merge_eo(A)}; \\ \mathtt{A} \leftarrow \mathtt{merge2(oe(A), eo(A))} \end{array} \right\} \supset \mathtt{A} \leftarrow \mathtt{merge2(merge_oe(A), merge_eo(A))},$$

where the functions $\mathtt{merge_oe(A)}$ and $\mathtt{merge_eo(A)}$ represent the mergers of the left and right halves of these two sublists, and $\mathtt{merge2(A, B)}$ denotes the pairwise comparison of elements from $\mathtt{A}$ and $\mathtt{B}$.

$$\begin{array}{rcl} \mathtt{merge_oe(A)} & \stackrel{\mathrm{def}}{=} & \mathtt{merge(odds(lt(A)), evns(rt(A)))} \\ \mathtt{merge_eo(A)} & \stackrel{\mathrm{def}}{=} & \mathtt{merge(evns(lt(A)), odds(rt(A)))} \\ \mathtt{merge2(A, B)} & \stackrel{\mathrm{def}}{=} & \mathtt{if}\ |\mathtt{A}| = 0 \vee |\mathtt{B}| = 0\ \mathtt{then}\ \mathtt{A\hat{\ }B} \\ & & \mathtt{else\ if\ hd(A)} \leq \mathtt{hd(B)} \\ & & \quad \mathtt{then\ cons(hd(A), cons(hd(B), merge2(tl(A), tl(B))))} \\ & & \quad \mathtt{then\ cons(hd(B), cons(hd(A), merge2(tl(A), tl(B))))}. \end{array}$$

The property above leads straight to a recursive merge algorithm. The derivation

is just as for the parallel summation algorithm in section 7.2.1.

$$
\begin{array}{rcl}
\mathtt{eo_merge(n,s,A)} & \stackrel{\text{def}}{=} & \mathtt{if\ n = 0\ then\ empty} \\
& & \mathtt{else}\ \{ \\
& & \quad \mathtt{eo_merge(n-1, oe(s), A) \wedge eo_merge(n-1, eo(s), A);} \\
& & \quad \mathtt{merge_step(s, A)} \\
& & \} \\
\mathtt{merge_step(s,A)} & \stackrel{\text{def}}{=} & \mathtt{forall\ j < |s|/2 : comparex(A_{s_{2\times j}}, A_{s_{2\times j+1}})} \\
\mathtt{comparex(A,A')} & \stackrel{\text{def}}{=} & \mathtt{if\ A < A'\ then\ A, A' := A, A'\ else\ A, A' := A', A.}
\end{array}
$$

Furthermore, an equivalent iterative form of this algorithm can be constructed by analogy with the iterative version of `par_sum` in section 7.2.2.

$$
\begin{array}{rcl}
\mathtt{eo_merge_i(n,A)} & \stackrel{\text{def}}{=} & \mathtt{for\ i < n\ do} \\
& & \quad \mathtt{forall\ j < 2^{n-1-i} :} \\
& & \qquad \mathtt{merge_step(i, apply(n-1-i, j, oe, eo, [0..2^n]), A).}
\end{array}
$$

The derivation guarantees that this is equivalent to the recursive version.

7.3.3 Iterative Mergesort Algorithm

Finally, the mergesort algorithm `mergesort_i` may be rewritten with this definition, and the function `apply` rewritten with an equivalent but more direct method of selecting the appropriate part of the list, to give the algorithm:

$$
\begin{array}{rcl}
\mathtt{eo_mergesort_i(n,A)} & \stackrel{\text{def}}{=} & \mathtt{for\ i < n\ do} \\
& & \quad \mathtt{forall\ j < 2^{n-1-i} :} \\
& & \qquad \mathtt{for\ k < i+1\ do} \\
& & \qquad\quad \mathtt{forall\ l < 2^{i-k} :} \\
& & \qquad\qquad \mathtt{forall\ m < 2^k :} \\
& & \qquad\qquad\quad \mathtt{eo_step(i+1, j, k, l, 2\times m, 2^{i-k}, A)} \\
\mathtt{eo_step(i,j,k,l,m,d)} & \stackrel{\text{def}}{=} & \mathtt{comparex(A_{j\times 2^i + a(m)}, A_{j\times 2^i + a(m+1)})}
\end{array}
$$

in which $\mathtt{a(m)}$ is shorthand for

$$\mathtt{if\ m < 2^k\ then\ l + m \times d\ else\ 2^i - 1 - l - (2^{k+1} - 1 - m) \times d.}$$

Mercifully, this algorithm is guaranteed by derivation to be correct,

$$\models \quad \mathtt{eo_mergesort_i(n,A)} \quad \supset \quad \mathtt{A \leftarrow sort(A)},$$

and it is functionally equivalent to the original mergesort algorithm. It is likely to be much more efficient than the original algorithm in many circumstances, indeed it leads directly to a hardware realisation, but it is also much more obscure. Figure 7.6 shows how this algorithm performs the last merge of the mergesort depicted in figure 7.5 above.

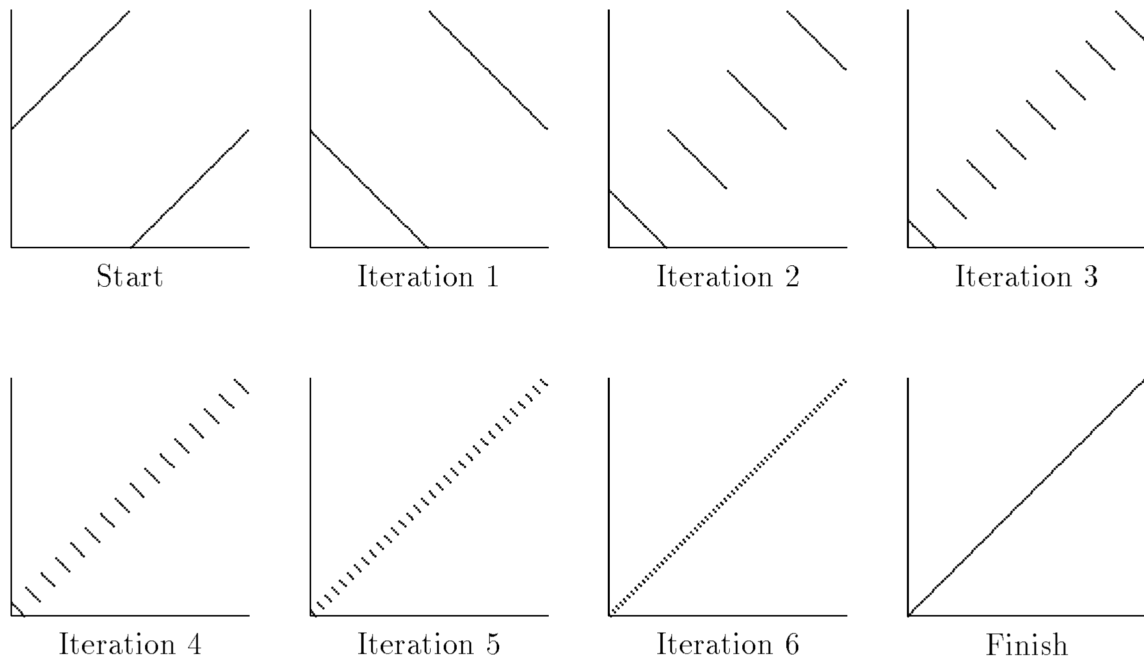


Figure 7.6: Operation of the merge algorithm on a list `A` of `128` elements, each half of which is initially sorted.

7.4 Discussion

The three examples of this chapter have, I hope, made it clear that Tempura is a suitable vehicle for discussing algorithm development and program equivalence. It should also be obvious that in examples like these a recursive algorithm is very much easier to construct and to reason about than its iterative counterpart. Nevertheless, the iterative version often executes more efficiently than the original recursive formulation. It is therefore particularly valuable to be able to transform a recursive program into a logically equivalent iterative one. This can be done by rewriting with theorems of logic, mechanically in a system like HOL, without the need of a stack (or any other operational device) to assist the transformation.

Other solutions to the Towers of Hanoi problem are discussed in a previous paper [Hal87], and Rohl [Roh87] also offers an extensive informal analysis of the problem using Pascal. He derives several iterative versions, but does so by effectively simulating the execution of the recursive program (by explicitly introducing a stack). As observed, a more direct transformation is possible in Tempura, and the two forms are provably equivalent.

Parallel summation is an example of what Hillis and Steele [HS86] call a data

parallel algorithm. Algorithms of this form are particularly suitable for implementation on SIMD processor arrays, and on the Connection Machine in particular. The odd-even mergesort is another algorithm of this form, and both algorithms will be considered again in chapter 8 where practical implementations will be discussed. It will be shown that both algorithms can be implemented efficiently on a shuffle-exchange network.

Chapter 8

Parallel Processing

This chapter discusses a few of the issues involved in adapting parallel programs to run on real parallel computers. The first part of the chapter considers fine-grained parallel systems such as array processors; the second part considers implementation on coarse-grained systems such as multiprocessors and multicomputers. A new parallel composition operator, $\|$, is introduced in section 8.2.1 for describing coarse-grained concurrency.

Up to now I have ignored the limitations of parallel processing, but parallel algorithms have to be run on real computers constructed by connecting together real processors and real storage devices, and this compromises in many ways the ideal model assumed so far. The principal restrictions concern the number of processors and the interconnections between them.

If there are relatively few processors sharing a common memory, as in a typical multiprocessor, the major problem is how to partition the work to make the best use of available resources. If, on the other hand, there are a large number of processors, as in a processor array, connecting them together is a major problem. In this case, it is not feasible for all processors to share a common memory because the processor speed would be limited by contention for that memory, and it is not feasible to connect each processor directly to every other because the interconnection network would be too complex. For these reasons, considerable effort has gone into the design of useful interconnection networks with small numbers of interconnections, and our algorithms have to be adapted to their peculiarities.

This chapter presents some of these issues from the Tempura point of view. It is not intended to be an exhaustive discussion, the point is only to show that it is very easy in Tempura to represent a problem in many different ways, and at different levels of concern. The examples are all simple algorithms which were introduced earlier in different ways. In the first part of the chapter, the summation and merge-sort algorithms of chapter 7 are adapted for implementation on suitable processor arrays. The second half concerns parallel processes. The idea of the parallel composition of two processes, $p \| p'$, is introduced, and for illustration the familiar matrix

multiplication algorithm is adapted for multiprocessor implementation.

8.1 Processor Arrays

A processor array contains a number of identical processors which operate synchronously. Each parallel process proceeds at the same rate and performs the same instruction on each step, though each operates on different data, of course. Usually there are a large number of processors, so it is infeasible to connect each one directly to every other because the number and complexity of the connections would be overwhelming. Nevertheless, there are a number of interconnection patterns that have been found to be versatile enough for a variety of applications.

8.1.1 Interconnections

The simplest scheme is to connect each processor to its nearest neighbour in the form of a linear array, or two-dimensional mesh, and the systolic matrix multiplication algorithm on page 38 uses just this sort of interconnection pattern. More exotic interconnection patterns include the *hypercube* and the *shuffle-exchange*.

Hypercube

A hypercube network consists of $\mathtt{p} = 2^{\mathtt{n}}$ processors connected in the form of an $\mathtt{n}$-dimensional hypercube, as shown in figure 8.1. Two processors, $\mathtt{i}, \mathtt{j} < \mathtt{p}$, are adjacent if the binary representations of $\mathtt{i}$ and $\mathtt{j}$ differ in exactly one bit. Thus, processor $\mathtt{i}$ is connected to processors $\mathtt{i} + \mathtt{c}(\mathtt{d})$, where

$$\mathtt{c}(\mathtt{d}) \stackrel{\text{def}}{=} 2^{\mathtt{n}-1-\mathtt{d}},$$

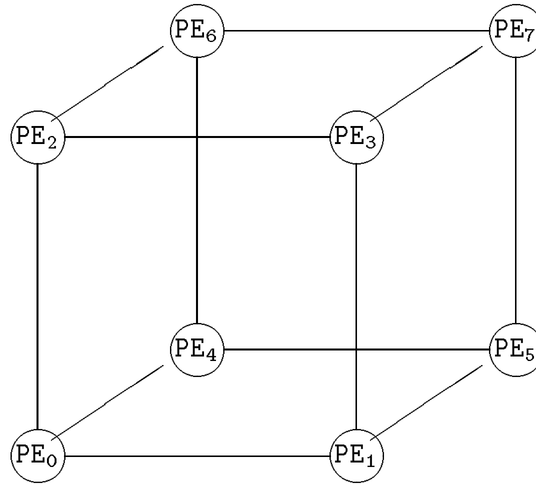
for each dimension $\mathtt{d} < \mathtt{n}$.

Shuffle-Exchange

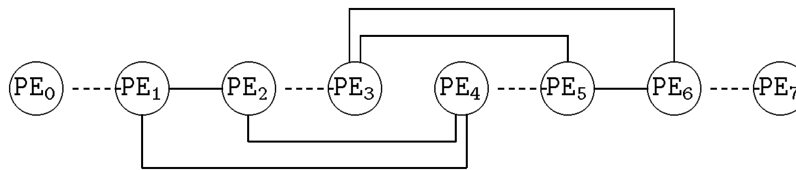
The shuffle-exchange network consists of $\mathtt{p} = 2^{\mathtt{n}}$ processors connected as shown in figure 8.1. There are two kinds of connections. An exchange connection links two processors, $\mathtt{i}, \mathtt{j} < \mathtt{p}$, if the binary representations of $\mathtt{i}$ and $\mathtt{j}$ differ in their least significant bits. A shuffle connection links processor $\mathtt{i}$ to processor $2 \times \mathtt{i}\,\mathtt{mod}\,(\mathtt{p} - 1)$, except that processor $\mathtt{p} - 1$ is connected to itself.[1] Thus, processor $\mathtt{i}$ is connected to processors $\mathtt{e}(\mathtt{i})$ and $\mathtt{s}(\mathtt{i})$, where

$$\begin{aligned} \mathtt{s}(\mathtt{i}) &\stackrel{\text{def}}{=} \mathtt{if}\ \mathtt{i} < \mathtt{p}/2\ \mathtt{then}\ 2 \times \mathtt{i}\ \mathtt{else}\ 2 \times \mathtt{i} + 1 - \mathtt{p} \\ \mathtt{e}(\mathtt{i}) &\stackrel{\text{def}}{=} \mathtt{if}\ \mathtt{even}(\mathtt{i})\ \mathtt{then}\ \mathtt{i} + 1\ \mathtt{else}\ \mathtt{i} - 1. \end{aligned}$$

[1] The term "shuffle" derives from the fact that after a shuffle operation the elements are re-ordered as if they were a perfectly shuffled deck of cards.



(a)



(b)

Figure 8.1: Interconnection networks: (a) the hypercube with eight processors, (b) the shuffle-exchange network with eight processors (shuffle connections are solid, exchange connections dashed).

The function `e` represents an exchange link, and `s` represents a shuffle link. Corresponding to these connections there are three principal data movement operations, `shuffle`(`A`), which moves the data on processor `i` to processor `s`(`i`), `unshuffle`(`A`) which is the inverse of `shuffle`(`A`), and `exchange`(`A`), which exchanges data between each even-odd pair of processors.

$$\begin{aligned}
\texttt{shuffle}(\texttt{A}) &\stackrel{\text{def}}{=} \texttt{forall i} < \texttt{p} : \texttt{A}_{\texttt{s(i)}} := \texttt{A}_{\texttt{i}} \\
\texttt{unshuffle}(\texttt{A}) &\stackrel{\text{def}}{=} \texttt{forall i} < \texttt{p} : \texttt{A}_{\texttt{i}} := \texttt{A}_{\texttt{s(i)}} \\
\texttt{exchange}(\texttt{A}) &\stackrel{\text{def}}{=} \texttt{forall i} < \texttt{p} : \texttt{A}_{\texttt{e(i)}} := \texttt{A}_{\texttt{i}}.
\end{aligned}$$

Note, however, that the unshuffle operation can be implemented in terms of `shuffle`. The elements of `A` may be unshuffled by $\texttt{n} - \texttt{1}$ successive shuffle operations.

8.1.2 Summation

Recall the iterative parallel summation algorithm eo_sum_i given on page 118. It was defined as follows:

$$\texttt{eo_sum_i}(\texttt{n},\texttt{A}) \stackrel{\text{def}}{=} \texttt{for } \texttt{i} < \texttt{n} \texttt{ do forall } \texttt{j} < 2^{\texttt{n}-1-\texttt{i}} : \texttt{A}_\texttt{j} := \texttt{A}_\texttt{j} + \texttt{A}_{\texttt{j}+2^{\texttt{n}-1-\texttt{i}}}.$$

Suppose that one element of the array A is stored on each of $\texttt{p} = 2^{\texttt{n}}$ processors, then on step i processor j adds its own value to that held on processor $\texttt{j} + 2^{\texttt{n}-1-\texttt{i}}$. Thus, processor j should ideally be connected to processors $\texttt{j} + 2^{\texttt{n}-1-\texttt{i}}$ for each i less than n, which is exactly how the n-dimensional hypercube is arranged.

Hypercube

On the hypercube, a single step of the algorithm can be replaced by a transfer followed by an addition, since

$$\models \quad \texttt{A}_\texttt{j} := \texttt{A}_\texttt{j} + \texttt{A}_{\texttt{j}+2^{\texttt{n}-1-\texttt{i}}} \quad \sim \quad \exists \texttt{L} : \{\texttt{L} := \texttt{A}_{\texttt{j}+\texttt{c}(\texttt{i})} \,;\, \texttt{A}_\texttt{j} := \texttt{A}_\texttt{j} + \texttt{L}\}$$

if A is a frame variable. Thus, using the local array B to hold intermediate results, the summation algorithm can be rewritten for the hypercube.

```
eo_sum_c(n, A)  ≝  ∃B :
                     for i < n do
                       forall j < c(i) : {
                         B_j := A_{j+c(i)};
                         A_j := A_j + B_j
                       },
```

where A is assumed to be a frame variable and B is a $2^{\texttt{n}}$ element list. However, it is not essential that processor j be physically connected to each of the processors $\texttt{j} + \texttt{c}(\texttt{i})$. The same effect can be achieved on a shuffle-exchange network.

Shuffle-Exchange

A shuffle operation brings elements j and $(\texttt{j} + \texttt{p}/2) \bmod \texttt{p}$ onto adjacent processors, because

$$\models \quad \texttt{shuffle(A)} \quad \equiv \quad \texttt{forall } \texttt{j} < \texttt{p}/2 : \texttt{A}_{2\times\texttt{j}}, \texttt{A}_{2\times\texttt{j}+1} := \texttt{A}_\texttt{j}, \texttt{A}_{\texttt{j}+\texttt{p}/2}.$$

They can then be moved onto the same processor by an exchange operation.

$$\models \quad \texttt{exchange(B)} \quad \equiv \quad \texttt{forall } \texttt{i} < \texttt{p}/2 : \texttt{B}_{2\times\texttt{j}}, \texttt{B}_{2\times\texttt{j}+1} := \texttt{B}_{2\times\texttt{j}+1}, \texttt{B}_{2\times\texttt{j}}.$$

Repeating the shuffle-exchange brings together elements j and $(\texttt{j} + \texttt{p}/4) \bmod \texttt{p}$, and so on. The ith iteration brings together elements j and $(\texttt{j} + \texttt{p}/2^{\texttt{i}}) \bmod \texttt{p}$.

time	A
0	[0, 1, 2, 3, 4, 5, 6, 7]
1	[0, 1, 2, 3, 4, 5, 6, 7]
2	[4, 6, 8, 10, 4, 5, 6, 7]
3	[4, 6, 8, 10, 4, 5, 6, 7]
4	[12, 16, 8, 10, 4, 5, 6, 7]
5	[12, 16, 8, 10, 4, 5, 6, 7]
6	[28, 16, 8, 10, 4, 5, 6, 7]

(a)

time	A
0	[0, 1, 2, 3, 4, 5, 6, 7]
1	[0, 4, 1, 5, 2, 6, 3, 7]
4	[4, 4, 6, 6, 8, 8, 10, 10]
5	[4, 8, 4, 8, 6, 10, 6, 10]
8	[12, 12, 12, 12, 16, 16, 16, 16]
9	[12, 16, 12, 16, 12, 16, 12, 16]
12	[28, 28, 28, 28, 28, 28, 28, 28]

(b)

Figure 8.2: Summation of an array of numbers on (a) the hypercube, and (b) the shuffle-exchange network.

Thus, at a cost of two communications for each step of the algorithm, the summation can be performed on a shuffle-exchange network.

$$
\begin{aligned}
\mathtt{eo_sum_s(n, A)} \stackrel{\text{def}}{=} \ & \exists \mathtt{B}: \\
& \quad \mathtt{for\ i < n\ do}\ \{ \\
& \quad\quad \mathtt{shuffle(A);} \\
& \quad\quad \mathtt{forall\ j < p : B_j := A_j;} \\
& \quad\quad \mathtt{exchange(B);} \\
& \quad\quad \mathtt{forall\ j < p : A_j := A_j + B_j} \\
& \quad \}.
\end{aligned}
$$

This algorithm results in the sum being assigned to *every* element of A,

$$\models \ \mathtt{eo_sum_s(n, A)} \ \supset \ \mathtt{forall\ i < n : A_i} \leftarrow \textstyle\sum_{\mathtt{j=0}}^{2^{\mathtt{n}}} \mathtt{A_j},$$

as shown in figure 8.2.

8.1.3 Mergesort

The even-odd variant of Batcher's merge algorithm can, like the parallel summation algorithm, be implemented on a shuffle-exchange network. On page 122 the algorithm was defined for a $2^{\mathtt{n}}$-element array A as follows:

$$
\begin{aligned}
\mathtt{eo_merge(n, s, A)} \stackrel{\text{def}}{=} \ & \mathtt{if\ n = 0\ then\ empty} \\
& \mathtt{else}\ \{ \\
& \quad \mathtt{eo_merge(n-1, oe(s), A)} \land \mathtt{eo_merge(n-1, eo(s), A);} \\
& \quad \mathtt{forall\ i < 2^{n-1} : comparex(A_{s_{2\times i}}, A_{s_{2\times i+1}})} \\
& \},
\end{aligned}
$$

where oe(A) = odds(lt(A))^evns(rt(A)) and eo(A) = evns(lt(A))^odds(rt(A)). By unwinding the recursion so that the base case becomes n = 1 rather than n = 0, and

permuting the array A rather than its access list s, the body of eo_merge is easily transformed into the functionally equivalent form:

```
if n = 1 then comparex(A_0, A_1)
else {
  lt(A), rt(A) := odds(lt(A))^evns(rt(A)), evns(lt(A))^odds(rt(A));
  eo_merge(n − 1, lt(s), A) ∧ eo_merge(n − 1, rt(s), A);
odds(lt(A))^evns(rt(A)), evns(lt(A))^odds(rt(A)) := lt(A), rt(A);
  forall i < 2^(n−1) : comparex(A_(2×i), A_(2×i+1))
}.
```

Now suppose that A is stored on a p-element processor array, one element at each processor ($p = 2^n$), and that the processors are connected up in the shuffle-exchange pattern.

Shuffle-Exchange

First, observe that an "unshuffle" operation moves all the even-indexed elements into the left half of the array and all the odd-indexed elements into the right half; that is,

$$\models \quad \texttt{unshuffle(A)} \quad \equiv \quad \texttt{lt(A)}, \texttt{rt(A)} := \texttt{evns(A)}, \texttt{odds(A)},$$

This is almost the permutation required for the merge. The correct permutation is obtained by exchanging the even and odd elements of the left half of A before unshuffling. An exchange operation on the left half of the array does just that. If $p = 2^n$,

$$\models \quad \texttt{lexchng}(n, A) \quad \supset \quad \texttt{evns(lt(A))}, \texttt{odds(lt(A))} := \texttt{odds(lt(A))}, \texttt{evns(lt(A))},$$

where the operation lexchng(n, A) exchanges the left half of A. In fact, it exchanges the left halves of all subarrays of size 2^n, so that the same predicate can be used in recursive mergers.

$$\texttt{lexchng}(n, A) \stackrel{\text{def}}{=} \texttt{forall } i < p/2^n : \texttt{forall } j < 2^{n-1} : A_{e(i\times 2^n+j)} := A_{i\times 2^n+j}.$$

When $p = 2^n$, combining these two operations in sequence gives the required permutation,

$$\models \quad \texttt{lexchng}(n, A) \,;\, \texttt{unshuffle(A)} \quad \sim \quad \texttt{lt(A)} \leftarrow \texttt{odds(lt(A))\^{}evns(rt(A))}$$
$$\models \quad \texttt{lexchng}(n, A) \,;\, \texttt{unshuffle(A)} \quad \sim \quad \texttt{rt(A)} \leftarrow \texttt{evns(lt(A))\^{}odds(rt(A))}.$$

The inverse operation is simply a shuffle followed by another exchange of the left half. However, there is no need to perform the exchange in the merge algorithm because it is immediately followed by a comparison of each even-indexed element with the following odd one, and this comparison requires an exchange on the whole array.

The comparison may be implemented as follows, storing the exchanged element of $\mathtt{A}$ in the array $\mathtt{B}$:

$$
\begin{aligned}
\mathtt{eo_compare(A,B)} \stackrel{\text{def}}{=} \; & \mathtt{forall\ i < p : \{} \\
& \quad \mathtt{B_i := A_i;} \\
& \quad \mathtt{exchange(B);} \\
& \quad \mathtt{A_i := if\ even(i)\ then\ min(A_i, B_i)\ else\ max(A_i, B_i)} \\
& \mathtt{\}.}
\end{aligned}
$$

This is functionally equivalent to the direct comparison used in the original algorithm; that is,

$$\models \; \exists \mathtt{B} : \mathtt{eo_compare(A,B)} \;\sim\; \mathtt{forall\ i < p/2 : comparex(A_{2\times i}, A_{2\times i+1})},$$

provided that $\mathtt{A}$ is a frame variable.

The Mergesort Algorithm

A shuffle-exchange algorithm, $\mathtt{eo_merge_s(n,A,B)}$, may be obtained by using the functional equivalences above to rewrite the appropriate parts of the original even-odd merge algorithm, $\mathtt{eo_merge(n,A)}$. The list $\mathtt{B}$ is used to hold temporary results.

$$
\begin{aligned}
\mathtt{eo_merge_s(n,A,B)} \stackrel{\text{def}}{=} \; & \mathtt{if\ n = 1\ then\ eo_compare(A,B)} \\
& \mathtt{else\ \{} \\
& \quad \mathtt{lexchng(n,A);} \\
& \quad \mathtt{unshuffle(A);} \\
& \quad \mathtt{eo_merge_s(n-1,A,B);} \\
& \quad \mathtt{shuffle(A);} \\
& \quad \mathtt{eo_compare(A,B)} \\
& \mathtt{\}.}
\end{aligned}
$$

Note that the two recursive calls to $\mathtt{eo_merge}$ in the original algorithm have been replaced by a single recursive call to $\mathtt{eo_merge_s}$. This is possible because a call of $\mathtt{eo_merge_s(m,A,B)}$ merges all sublists of size $2^{\mathtt{m}}$; that is, if $\mathtt{m} < \mathtt{n}$ and the arrays $\mathtt{A}$ and $\mathtt{B}$ are of size $\mathtt{p} = 2^{\mathtt{n}}$, then

$$\models \; \exists \mathtt{B} : \mathtt{eo_merge_s(m,A,B)} \;\sim\; \forall \mathtt{i} < 2^{\mathtt{n-m}} : \mathtt{eo_merge(m, A_{i\times 2^m..(i+1)\times 2^m})}.$$

This property makes it particularly simple to do the mergesort.

The parallel mergesort of a list $\mathtt{A}$ is just a sequence of parallel mergers of $\mathtt{i}$-element sublists of $\mathtt{A}$ for $\mathtt{i} < \mathtt{n}$. It was defined iteratively on page 122 as follows:

$$
\begin{aligned}
\mathtt{eo_mergesort(n,A)} \stackrel{\text{def}}{=} \; & \mathtt{for\ i < n\ do} \\
& \quad \mathtt{forall\ j < 2^{n-1-i} :} \\
& \qquad \mathtt{eo_merge(i+1, A_{j\times 2^{i+1}..(j+1)\times 2^{i+1}})}.
\end{aligned}
$$

The corresponding algorithm for the shuffle-exchange network may be obtained by rewriting this algorithm with the property above.

$$\mathtt{eo_mergesort_s(n,A,B)} \stackrel{\text{def}}{=} \mathtt{for\ i < n\ do\ eo_merge_s(i+1,A,B)}.$$

An illustration of how the algorithm works can be found in figure 8.3.

time	A	B
0	[7, 6, 5, 4, 3, 2, 1, 0]	
1	[7, 6, 5, 4, 3, 2, 1, 0]	[7, 6, 5, 4, 3, 2, 1, 0]
2	[7, 6, 5, 4, 3, 2, 1, 0]	[6, 7, 4, 5, 2, 3, 0, 1]
3	[6, 7, 4, 5, 2, 3, 0, 1]	
4	[7, 6, 4, 5, 3, 2, 0, 1]	
5	[7, 4, 3, 0, 6, 5, 2, 1]	
6	[7, 4, 3, 0, 6, 5, 2, 1]	[7, 4, 3, 0, 6, 5, 2, 1]
7	[7, 4, 3, 0, 6, 5, 2, 1]	[4, 7, 0, 3, 5, 6, 1, 2]
8	[4, 7, 0, 3, 5, 6, 1, 2]	
9	[4, 5, 7, 6, 0, 1, 3, 2]	
10	[4, 5, 7, 6, 0, 1, 3, 2]	[4, 5, 7, 6, 0, 1, 3, 2]
11	[4, 5, 7, 6, 0, 1, 3, 2]	[5, 4, 6, 7, 1, 0, 2, 3]
12	[4, 5, 6, 7, 0, 1, 2, 3]	
13	[5, 4, 7, 6, 0, 1, 2, 3]	
14	[5, 7, 0, 2, 4, 6, 1, 3]	
15	[7, 5, 0, 2, 6, 4, 1, 3]	
16	[7, 0, 6, 1, 5, 2, 4, 3]	
17	[7, 0, 6, 1, 5, 2, 4, 3]	[7, 0, 6, 1, 5, 2, 4, 3]
18	[7, 0, 6, 1, 5, 2, 4, 3]	[0, 7, 1, 6, 2, 5, 3, 4]
19	[0, 7, 1, 6, 2, 5, 3, 4]	
20	[0, 2, 7, 5, 1, 3, 6, 4]	
21	[0, 2, 7, 5, 1, 3, 6, 4]	[0, 2, 7, 5, 1, 3, 6, 4]
22	[0, 2, 7, 5, 1, 3, 6, 4]	[2, 0, 5, 7, 3, 1, 4, 6]
23	[0, 2, 5, 7, 1, 3, 4, 6]	
24	[0, 1, 2, 3, 5, 4, 7, 6]	
25	[0, 1, 2, 3, 5, 4, 7, 6]	[0, 1, 2, 3, 5, 4, 7, 6]
26	[0, 1, 2, 3, 5, 4, 7, 6]	[1, 0, 3, 2, 4, 5, 6, 7]
27	[0, 1, 2, 3, 4, 5, 6, 7]	

Figure 8.3: The mergesort algorithm on an 8-processor shuffle-exchange network. The list `B` is used to hold temporary results during comparisons.

8.2 Parallel Processes

Many parallel computations are not nearly so regular as those described in the previous section, and in particular, it may not be the case that each process takes the same number of steps to complete. This section describes a general way to combine parallel processes having different or unknown computation lengths. The familiar example of matrix multiplication is used to illustrate the idea.

8.2.1 The Parallel Composition Operator

The formula $p \parallel p'$ denotes the parallel composition of two processes p and p', which may have different computation lengths. It is defined in terms of the operator `extend`, which was introduced on page 52. The formula $\texttt{extend}\, p$ is true on an interval if p is true on some initial subinterval,

$$\texttt{extend}\, p \quad \stackrel{\text{def}}{=} \quad p\,;\texttt{true}.$$

The parallel composition of two processes, $p \parallel p'$, is true on an interval if p and p' do not disagree up to the point where one of them finishes, and the interval continues until both have finished.

$$p \parallel p' \quad \stackrel{\text{def}}{=} \quad (p \wedge \texttt{extend}\,(p')) \vee (\texttt{extend}\,(p) \wedge p').$$

One of the processes may need to be extended until the other finishes.

In practice, the composition $p \parallel p'$ is executed by introducing markers to determine when each subprocess finishes. If the marker ε is true when process p is done and ε' is true when process p' is done, then $p \parallel p'$ terminates as soon as both of the markers are true,

$$\begin{aligned} p \parallel p' \quad \equiv \quad & \exists \varepsilon, \varepsilon' : \{ \\ & \quad \texttt{halt}\,(\varepsilon \wedge \varepsilon') \wedge \\ & \quad (p \wedge \varepsilon \;\texttt{is empty}\,; \texttt{stable}\,(\varepsilon)) \wedge \\ & \quad (p' \wedge \varepsilon' \;\texttt{is empty}\,; \texttt{stable}\,(\varepsilon')) \\ & \}. \end{aligned}$$

For combining a number of processes in parallel, there is an iterated constructor, $\texttt{forpar}\; i < n : p$, analogous to the universal quantifier `forall`. For example,

$$\texttt{forpar i} < 3 : p \quad \equiv \quad p[\texttt{0/i}] \parallel p[\texttt{1/i}] \parallel p[\texttt{2/i}].$$

It may be defined recursively in much the same way as the for-loop. Note that parallel composition is commutative and associative, so neither the order nor the grouping of the processes is important.

8.2.2 Matrix Multiplication

Let us look, once again, at the problem of matrix multiplication. But now the aim is to develop an algorithm for calculating the product, $\mathtt{C}$, of two $\mathtt{n}\times\mathtt{n}$ matrices $\mathtt{A}$ and $\mathtt{B}$ on a tightly coupled multiprocessor with a small number of processing elements.

Row Decomposition

Suppose, for the moment, that there are just two processors. One way to proceed is to partition the matrix $\mathtt{A}$ into two parts, one comprising rows $\mathtt{0}$ to $\mathtt{n/2-1}$, the other rows $\mathtt{n/2}$ to $\mathtt{n-1}$, and to assign to each processor the task of multiplying $\mathtt{B}$ by one half of $\mathtt{A}$. Thus, each processor does one of the following multiplications:

$$\begin{array}{lcl} \mathtt{C}_{(0..n/2)(0..n)} & \leftarrow & \mathtt{A}_{(0..n/2)(0..n)} \times \mathtt{B}_{(0..n)(0..n)} \\ \mathtt{C}_{(n/2..n)(0..n)} & \leftarrow & \mathtt{A}_{(n/2..n)(0..n)} \times \mathtt{B}_{(0..n)(0..n)}, \end{array}$$

where the notation $\mathtt{A}_{(r..s)(t..u)}$ denotes the submatrix of $\mathtt{A}$ made up of rows $\mathtt{r}$ to $\mathtt{s-1}$ and columns $\mathtt{t}$ to $\mathtt{u-1}$.

Let us denote by $\mathtt{rows(ptn)}$ the multiplication of rows $\mathtt{r}$ in the partition $\mathtt{ptn}$ of $\mathtt{A}$ with $\mathtt{B}$, producing rows $\mathtt{r}\in\mathtt{ptn}$ of $\mathtt{C}$. The multiplication can be defined as below, using a local variable $\mathtt{L}$ to form the inner products.

```
rows(ptn)  ≝  for i ∈ ptn do
                for j ∈ [0..n] do
                  ∃L : {
                    L ⇐ 0;
                    for k ∈ [0..n] do L := L + A_ik × B_kj;
                    C_ij := L
                  }.
```

One processor is therefore to execute $\mathtt{rows([0..n/2])}$ and the other $\mathtt{rows([n/2..n])}$.

Notice that if $\mathtt{n}$ is odd the two tasks, $\mathtt{rows([0..n/2])}$ and $\mathtt{rows([n/2..n])}$, are not equal and will take different numbers of steps to complete, but the multiplication is not complete until both have finished. Therefore, the multiplication is simply the parallel composition of the two subprocesses,

$$\models \quad \mathtt{rows([0..n/2])} \parallel \mathtt{rows([n/2..n])} \quad \supset \quad \mathtt{forall}\ \mathtt{i},\mathtt{j} < \mathtt{n} : \mathtt{C}_{ij} \leftarrow \sum_{k=0}^{n-1} \mathtt{A}_{ik} \times \mathtt{B}_{kj},$$

provided that $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ are frame variables.

The effect of the parallel composition is just as if explicit boolean flags had been introduced to mark where each process terminates, as in the following code:

```
∃Done, Done′ : {
  halt (Done ∧ Done′) ∧
  {rows([0..n/2]) ∧ Done is empty ; stable (Done)} ∧
  {rows([n/2..n]) ∧ Done′ is empty ; stable (Done′)}
}.
```

The two representations are logically equivalent.

If more processors are available, the multiplication can be further subdivided in the same way, because

$$\models \ \mathtt{rows}([\mathtt{r}..\mathtt{s}]) \ \sim \ \mathtt{rows}([\mathtt{r}..\mathtt{i}]) \parallel \mathtt{rows}([\mathtt{i}..\mathtt{s}])$$

for any $\mathtt{i} \in [\mathtt{r}..\mathtt{s}]$, and a general algorithm for $\mathtt{p}$ processes is simply the iterated parallel composition of a number of rows,

$$\mathtt{rowmul}(\mathtt{p},\mathtt{n},\mathtt{A},\mathtt{B},\mathtt{C}) \ \stackrel{\text{def}}{=} \ \mathtt{forpar}\ \mathtt{i} < \mathtt{p} : \mathtt{rows}(\mathtt{part}(\mathtt{i})),$$

where the function $\mathtt{part(i)}$ returns the $\mathtt{i}$th partition,

$$\mathtt{part(i)} \ \stackrel{\text{def}}{=} \ [(\mathtt{i} \times \mathtt{n})/\mathtt{p}..((\mathtt{i}+1) \times \mathtt{n})/\mathtt{p}].$$

However, this way of partitioning is not particularly good if access to shared memory is slow, as it might be on a loosely coupled multiprocessor. This is because each processor must access every element of $\mathtt{B}$ in addition to one quarter of $\mathtt{A}$.

Block Decomposition

A better approach in this case is to partition each matrix into blocks, and use block multiplication to form the product. For instance, the multiplication may be partitioned into four processes as follows:

$$\begin{array}{lcl}
\mathtt{C}_{(0..n/2)(0..n/2)} & \leftarrow & \mathtt{A}_{(0..n/2)(0..n/2)} \times \mathtt{B}_{(0..n/2)(0..n/2)} + \mathtt{A}_{(0..n/2)(n/2..n)} \times \mathtt{B}_{(n/2..n)(0..n/2)} \\
\mathtt{C}_{(0..n/2)(n/2..n)} & \leftarrow & \mathtt{A}_{(0..n/2)(0..n/2)} \times \mathtt{B}_{(0..n/2)(n/2..n)} + \mathtt{A}_{(0..n/2)(n/2..n)} \times \mathtt{B}_{(n/2..n)(n/2..n)} \\
\mathtt{C}_{(n/2..n)(0..n/2)} & \leftarrow & \mathtt{A}_{(n/2..n)(0..n/2)} \times \mathtt{B}_{(0..n/2)(0..n/2)} + \mathtt{A}_{(n/2..n)(n/2..n)} \times \mathtt{B}_{(n/2..n)(0..n/2)} \\
\mathtt{C}_{(n/2..n)(n/2..n)} & \leftarrow & \mathtt{A}_{(n/2..n)(0..n/2)} \times \mathtt{B}_{(0..n/2)(n/2..n)} + \mathtt{A}_{(n/2..n)(n/2..n)} \times \mathtt{B}_{(n/2..n)(n/2..n)}
\end{array}$$

Each process forms one block of $\mathtt{C}$. Observe that now each process needs only to fetch a total of $\mathtt{n}^2$ elements of $\mathtt{A}$ and $\mathtt{B}$ from global memory, a significant saving if $\mathtt{n}$ is large and memory accesses take a significant amount of time.

In general, the matrix $\mathtt{C}$ may be divided into $\mathtt{p}^2$ blocks and one process assigned to calculate each block.

$$\begin{array}{lcl}
\mathtt{blkmul}(\mathtt{p},\mathtt{n},\mathtt{A},\mathtt{B},\mathtt{C}) & \stackrel{\text{def}}{=} & \mathtt{forpar}\ \mathtt{i} < \mathtt{p} : \\
 & & \quad \mathtt{forpar}\ \mathtt{j} < \mathtt{p} : \\
 & & \quad\quad \mathtt{for}\ \mathtt{k} < \mathtt{p}\ \mathtt{do} \\
 & & \quad\quad\quad \mathtt{blk}(\mathtt{part(i)},\mathtt{part(j)},\mathtt{part(k)}).
\end{array}$$

Each block is calculated by repeatedly fetching submatrices of $\mathtt{A}$ and $\mathtt{B}$ from shared

memory, using `fetch`, and adding their product into `C`.

$$
\begin{array}{lll}
\mathtt{blk(pi,pj,pk)} & \stackrel{\mathrm{def}}{=} & \mathtt{local\ Ab,Bb}:\{ \\
& & \quad \mathtt{fetch(A,Ab,pi,pk)}; \\
& & \quad \mathtt{fetch(B,Bb,pk,pj)}; \\
& & \quad \mathtt{for\ i} < |\mathtt{pi}|\ \mathtt{do} \\
& & \quad\quad \mathtt{for\ j} < |\mathtt{pj}|\ \mathtt{do} \\
& & \quad\quad\quad \exists \mathtt{L}:\{ \\
& & \quad\quad\quad\quad \mathtt{L} \Leftarrow 0; \\
& & \quad\quad\quad\quad \mathtt{for\ k} < |\mathtt{pk}|\ \mathtt{do\ L} := \mathtt{L} + \mathtt{Ab_{ik}} \times \mathtt{Bb_{kj}}; \\
& & \quad\quad\quad\quad \mathtt{C_{pi_i pj_j}} := \mathtt{C_{pi_i pj_j}} + \mathtt{L} \\
& & \quad\quad\quad \} \\
& & \}
\end{array}
$$

Naturally, this algorithm is functionally equivalent to the row decomposition algorithm above; that is,

$$\models \mathtt{blkmul(p,n,A,B,C)} \sim \mathtt{rowmul(p,n,A,B,C)}.$$

Figure 8.4 shows how the block multiplication algorithm works, using four processes to perform the multiplication of `A` and `B`, where

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix} \quad \text{and} \quad \mathtt{B} = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 1 & 4 & 3 & 2 \\ 2 & 1 & 4 & 3 \\ 3 & 2 & 1 & 4 \end{pmatrix}$$

Each access to global memory is assumed to take one unit of time.

8.3 Discussion

In this chapter I have shown that Tempura forms a sound basis for transforming algorithms into different forms, suitable for different computer architectures. I have also drawn a distinction between the two parallel composition operators, $\wedge$ and $\|$. Conjunction seems appropriate for dealing with fine-grained parallel operations that proceed in lock-step, as on a typical processor array, whereas the process composition operator is better suited to the coarse-grained concurrency of a typical multiprocessor, where each process proceeds at its own rate. However, there remain two problems with the process composition operator, one practical and one philosophical.

The practical problem concerns how to organise co-operation between parallel processes. As things stand, there are no restrictions governing the use of shared storage, so processes may interfere arbitrarily with one another. In order to make it easier to construct reliable programs a disciplined communication mechanism is needed. This is the subject of chapter 10.

time	C	time	C	time	C
0	$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	17	$\begin{pmatrix} 6 & 11 & 8 & 5 \\ 11 & 0 & 13 & 0 \\ 16 & 25 & 18 & 11 \\ 17 & 0 & 11 & 0 \end{pmatrix}$	34	$\begin{pmatrix} 24 & 22 & 24 & 30 \\ 11 & 18 & 13 & 8 \\ 24 & 30 & 24 & 22 \\ 17 & 16 & 11 & 6 \end{pmatrix}$
11	$\begin{pmatrix} 6 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 \\ 16 & 0 & 18 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	20	$\begin{pmatrix} 6 & 11 & 8 & 5 \\ 11 & 18 & 13 & 8 \\ 16 & 25 & 18 & 11 \\ 17 & 16 & 11 & 6 \end{pmatrix}$	37	$\begin{pmatrix} 24 & 22 & 24 & 30 \\ 22 & 18 & 30 & 8 \\ 24 & 30 & 24 & 22 \\ 30 & 16 & 22 & 6 \end{pmatrix}$
14	$\begin{pmatrix} 6 & 11 & 8 & 5 \\ 0 & 0 & 0 & 0 \\ 16 & 25 & 18 & 11 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	31	$\begin{pmatrix} 24 & 11 & 24 & 5 \\ 11 & 18 & 13 & 8 \\ 24 & 25 & 24 & 11 \\ 17 & 16 & 11 & 6 \end{pmatrix}$	40	$\begin{pmatrix} 24 & 22 & 24 & 30 \\ 22 & 24 & 30 & 24 \\ 24 & 30 & 24 & 22 \\ 30 & 24 & 22 & 24 \end{pmatrix}$

Figure 8.4: Parallel matrix multiplication by partitioning into four 2×2 blocks. The matrix C is the product of the two 4×4 matrices A and B shown in the text.

The philosophical problem also has practical implications. It concerns the underlying computational model. Although the process composition operator is intended for combining autonomous processes that execute at their own rates, it is assumed in the computational model that they do in fact proceed in lock-step (until the first one terminates). This is because the processes are true on the same interval; that is, all of their states are shared. A practical consequence is that it may be possible to draw incorrect conclusions about the behaviour of a parallel composition. For example, the composition

$$(\mathtt{N} \Leftarrow \mathtt{n}\,;\mathtt{while}\ \mathtt{N} \neq 0\ \mathtt{do}\ \mathtt{N} := \mathtt{N} - 1) \parallel (\mathtt{X} \Leftarrow 1\,;\mathtt{while}\ \mathtt{N} \neq 0\ \mathtt{do}\ \mathtt{X} := \mathtt{X} \times \mathtt{x})$$

satisfies the specification $\mathtt{X} \leftarrow \mathtt{x}^{\mathtt{n}}$, whereas one would not expect this program to necessarily give that result. There are two possible responses to this problem. One is to restrict the ways in which parallel processes can share data, so that programs like the one above are disallowed. The other approach is to use temporal projection to introduce local state sequences for each process, so that their execution is interleaved and it is no longer possible to draw incorrect conclusions. Chapter 10 addresses this problem.

Chapter 9

Real-Time Systems

This chapter is about real-time systems; it is divided into two parts. The first part introduces some new operators for expressing real-time concepts. These include projection, interrupts, traps and timeouts. The second part presents a model controller for a system of passenger lifts. The controller is specified in ITL, and a prototype based on that specification is constructed in Tempura.

The value of temporal logic for the specification and verification of real-time systems is well appreciated [Pnu83]. The purpose of this chapter is to show how its value to the designer of non-trivial systems is enhanced when an executable logic is used. A particular example, the controller for a system of passenger lifts, is chosen for illustration. The controller is specified in ITL, and from that specification is drawn a prototype implementation in Tempura. The prototype system has been tested in Tempura, and could be formally proved to meet its specification.

Most essential aspects of the system's behaviour are modelled. There may be any number of lifts and floors. Each lift has one button for each floor, and each floor has separate buttons to request ascending and descending lifts. Also, in each lift there is a button to open or re-open the doors, and another to signal an emergency. An earlier version of this example was presented in [Hal88].

Before describing the lift controller, a number of new operators are introduced. These express real-time constructs, such as interrupts, exceptions and time limits, which are needed in the specification of the lift controller. The remainder of the chapter concerns the lift controller. The first part of this fixes upon an external interface to the controller, the second develops a specification of the controller in ITL, and the third describes the Tempura prototype. Finally, some ways of improving the prototype are discussed.

9.1 Additional Operators

This section discusses some ideas and operations that are particularly relevant to the description of real-time systems. The first of these new operators is temporal

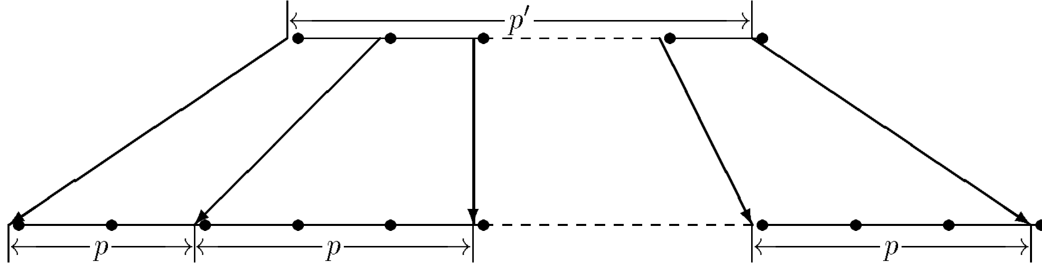


Figure 9.1: The temporal projection p `proj` p'

projection, which may be used for the interruption and later resumption of a process. Another important idea in real-time programming is exception handling, and a general mechanism is defined for this. A new operator, called *bar*, composes two processes in parallel so that both are terminated as soon as the first of them is done. This may be used to define traps and timeouts.

9.1.1 Projection

Temporal projection is one way to model a system on a number of different time-scales. For example, it may be that one needs to monitor the behaviour of some device, `dev(X)` say, but not all the time. Suppose, in fact, that the value of `X` is to be output on every tenth state. This is most easily achieved using the projection operator `proj`.

{`len(10) proj always output(X)`} ∧ `dev(X)`.

Projection of `len(10)` onto `always output(X)` repeatedly interrupts the output process by inserting an interval of length `10` between each pair of states on which `X` is output.

The projection p `proj` p' holds on an interval τ if there is a selection of time points on which p' is true and such that p is true on the subinterval between each pair of adjacent points in the selection. This is illustrated in figure 9.1.

$$
\begin{aligned}
p\ \mathtt{proj}\ p' \ \stackrel{\text{def}}{=}\ & \lambda\tau : \exists\, s, m : \{ \\
& \quad s(0) = 0 \wedge s(m) = |\tau| \wedge \forall\, i < m : s(i) \leq s(i+1) \wedge \\
& \quad p'(\mathit{select}(s, m, \tau)) \wedge \\
& \quad \forall\, i < m : p(\mathit{subint}(s(i), s(i+1), \tau)) \\
& \},
\end{aligned}
$$

where $\tau \in \mathbb{I}$, $s \in \mathbb{N} \rightarrow \mathbb{N}$, $m \in \mathbb{N}$ and

$$\mathit{select}(s, m, \tau) \ = \ \langle \tau_{s(0)}, \ldots, \tau_{s(m)} \rangle$$

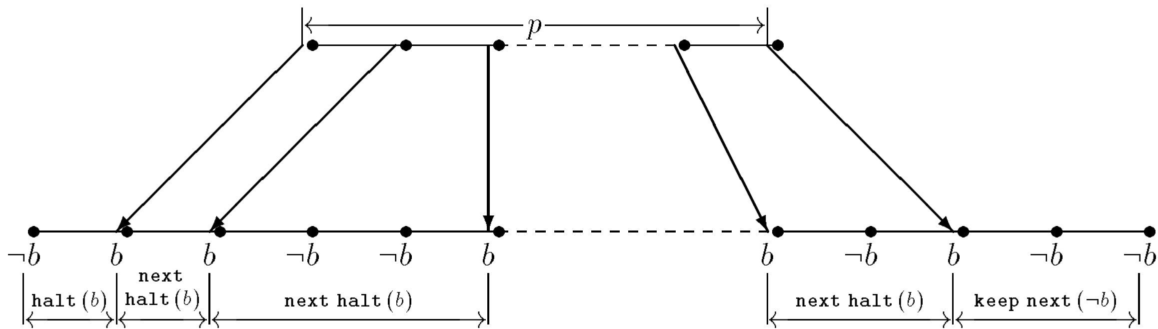


Figure 9.2: Construction of the formula p `when` b.

$$subint(i, j, \tau) \quad = \quad suffix(i, prefix(j, \tau)).$$

Observe that if p `proj` p' holds on an interval then so does `loop` p, and p' is true on the interval made up of the end points of the iterations of p.

A number of related ideas can be defined in terms of projection. For instance, the formula p `when` b is true if p holds on the interval made up of just those states on which the boolean expression b is true. Thus,

`len (7) when (X = 0)`

means that the variable `X` is zero exactly seven times.

The definition of p `when` b goes as follows. Successive subintervals with b true only at the beginning and end are picked out by projecting the formula `next halt` (b) onto p. Additionally, b may be false for some time at the beginning and end of the interval.

p `when` b $\stackrel{\text{def}}{=}$ `halt` (b) ; {`next halt` (b) `proj` p} ; `keep next` $(\neg b)$.

How it works is illustrated in figure 9.2.

9.1.2 Interrupts

Interrupt handling is a familiar problem in real-time programming. When an interrupt occurs, perhaps caused by a device requiring attention, the running program is suspended and execution begins on the appropriate interrupt service routine. When the service routine finishes, execution of the interrupted program resumes. This is just the kind of behaviour produced by temporal projection. The servicing of the interrupt occurs between two consecutive states of the interrupted program.

Let us write p `upon` b `do` p' to mean that whenever the boolean expression b is true, the execution of p is interrupted by p'.

p `upon` b `do` p' $\stackrel{\text{def}}{=}$ {`if` b `then` p' `else skip`} `proj` p.

An interruption is effected by projecting the corresponding formula onto p. For example, the formula

$$pgm \ \texttt{upon Printer do fill_buf(PrintBuf)}$$

might specify that whenever the signal `Printer` is set, the running program is interrupted whilst the print buffer is filled.

Interrupts may be nested by simply projecting those of higher-priority onto those of lower priority. In a specification of the form

$$(pgm \ \texttt{upon LowPri do} \ldots) \ \texttt{upon HighPri do} \ldots,$$

the interrupt `HighPri` has priority over `LowPri`, which in turn has priority over normal processing.

Note that the operator `upon` is useful in other situations besides the description of interrupt behaviour. It is used in the lift controller to add timing details to the specification.

9.1.3 Bar

The *bar* operator is used to compose two processes in parallel so that both are terminated as soon as the first one finishes. The composition $p \,\rule{2pt}{8pt}\, p'$ is defined in a similar way to the parallel composition $p \parallel p'$ (see page 134), but rather than extend the shorter process, it takes the prefix of the longer process. Recall that the prefix of a formula, `prefix` p, holds on the prefix of an interval on which p holds. It is defined as follows:

$$\texttt{prefix}\ p \ \stackrel{\text{def}}{=} \ \lambda\tau : \exists\, \tau' : |\tau| \leq |\tau'| \wedge \tau = \mathit{prefix}(|\tau|, \tau') \wedge p(\tau').$$

where $\tau, \tau' \in \mathbb{I}$. Thus, the composition $p \,\rule{2pt}{8pt}\, p'$ is true on an interval if either p and a prefix of p' holds, or p' and a prefix of p holds; that is,

$$p \,\rule{2pt}{8pt}\, p' \ \stackrel{\text{def}}{=} \ \{p \wedge (\texttt{prefix}\ p')\} \vee \{(\texttt{prefix}\ p) \wedge p'\}.$$

One of the processes runs to completion; the other may be terminated prematurely.

Like ordinary parallel composition, the composition $p \,\rule{2pt}{8pt}\, p'$ is executed by introducing markers to determine when each subprocess finishes, and terminating as soon as one of the markers is true.

$$\begin{array}{rcl} p \,\rule{2pt}{8pt}\, p' & = & \exists \varepsilon, \varepsilon' : \{ \\ & & \quad \texttt{halt}\,(\varepsilon \vee \varepsilon') \wedge \\ & & \quad \texttt{prefix}(p \wedge \varepsilon \ \texttt{is empty}) \wedge \\ & & \quad \texttt{prefix}(p' \wedge \varepsilon' \ \texttt{is empty})) \\ & & \}. \end{array}$$

A number of useful operators may be defined in terms of bar.

9.1.4 Traps

In order to trap exceptional conditions, such as termination signals and errors, it is of course possible to include explicit tests at every appropriate point in a program. But exceptions typically occur infrequently and at unpredictable times, making such a solution somewhat obscure and long-winded. A device that makes exception handling more modular is the trap, which is defined in terms of the bar operator.

If the boolean expression b is always false, the construct `trap` $b : p$ is equivalent to p; otherwise it terminates as soon as b becomes true. Thus,

$$\texttt{trap}\ b : p \ \stackrel{\text{def}}{=}\ \texttt{halt}\,(b) \,\rule{2pt}{1em}\, p,$$

For example, the formula

$$\texttt{trap Control_C} : pgm$$

might indicate that the program pgm exits when the user strikes the special key, `Control_C`.

9.1.5 Time Limits

Sometimes it is necessary to limit the time spent waiting for a particular condition to become true. A familiar real-time programming device which does this is the timeout. For example, when two processes are exchanging messages a failure in one process might cause the other to hang waiting for input. In this situation the working process can recover by giving up after a prearranged time limit has expired.

A timeout is easily defined using bar. For example, to terminate the program pgm after t units if it hasn't already terminated, one simply writes

$$pgm \,\rule{2pt}{1em}\, \texttt{len}\,(t).$$

In general, the timeout process need not be a simple length; it may be a process of arbitrary complexity.

The simplest form of timeout is expressed by the predicate $\texttt{timeout}(t, b)$, which waits at most t units of time for the boolean expression b to become true.

$$\texttt{timeout}(t, b) \ \stackrel{\text{def}}{=}\ \texttt{halt}\,(b) \,\rule{2pt}{1em}\, \texttt{len}\,(t).$$

But a simple timeout such as this can be expressed in a number of other ways without using bar. For instance, it satisfies the inductive property below:

$$\begin{aligned} \texttt{timeout}(0, b) &= \texttt{empty}, \\ \texttt{timeout}(t+1, b) &= \texttt{if}\ b\ \texttt{then empty else next timeout}(t, b), \end{aligned}$$

which is the operational representation used in Tempura.

Another kind of time limit occurs in the specification of real-time systems when one needs to ensure that a condition does become true within a particular interval of time. The construct $b\, \texttt{within}\, p$ takes care of this case. The expression b becomes true within an interval defined by p if there is no initial part of the interval on which p holds but b is not true at some time.

$$b\, \texttt{within}\, p \quad \stackrel{\text{def}}{=} \quad \neg\, \texttt{extend}(p \wedge \neg\, \texttt{sometime}(b)).$$

The formula p defines an interval within which b is sometime true. For instance,

$$\texttt{always if}\ \textit{request}\ \texttt{then}\ \{\textit{service}\ \texttt{within}\ \texttt{len}(t)\}$$

might indicate that a particular request is always serviced within t seconds of being registered.

However, it is often the case that a request is only serviced within a fixed time provided that no other request intervenes. This can be formulated by combining the operators `within` and `when`:

$$\texttt{always if}\ \textit{request}\ \texttt{then}\ \{\textit{service}\ \texttt{within}\ (\texttt{len}(t)\ \texttt{when}\ \neg \textit{request}')\}.$$

The request is serviced within t steps on which the alternative $\textit{request}'$ is not registered.

Just about the simplest usage of `within` is the form $b\, \texttt{within}\, \texttt{len}(t)$, and this can readily be expressed using induction, in a form which is most useful in practice,

$$\begin{aligned} b\, \texttt{within}\, \texttt{len}(0) &= b, \\ b\, \texttt{within}\, \texttt{len}(t+1) &= \texttt{if}\ \neg b\ \texttt{then if}\ \neg\texttt{empty}\ \texttt{then}\ \texttt{next}\,(b\, \texttt{within}\, \texttt{len}(t)). \end{aligned}$$

The more complex construction, $b\, \texttt{within}\, (\texttt{len}(t)\ \texttt{when}\ b')$, can be expressed in a similar way.

9.2 A Lift Control System

Let us now turn our attention to an example: the design of a controller for a system of passenger lifts. This problem has been considered by a number of other authors over the years. For example, in an early treatment Knuth describes an assembler program to simulate the behaviour of a single lift [Knu69], and more recently Barringer has given an abstract specification of a multiple lift system in temporal logic [Bar85]. The approach taken here, to develop a prototype controller in temporal logic, is somewhere between the two, being more operational than Barringer's specification but at a much higher level than Knuth's.

This development is in four parts. The first part describes the external interface to the lift controller, giving first an informal description and then a formal representation. The second part gives a formal specification of the controller in terms of its interface. The third part goes from the specification to an executable prototype of the system, and the last part discusses some ways of improving the prototype.

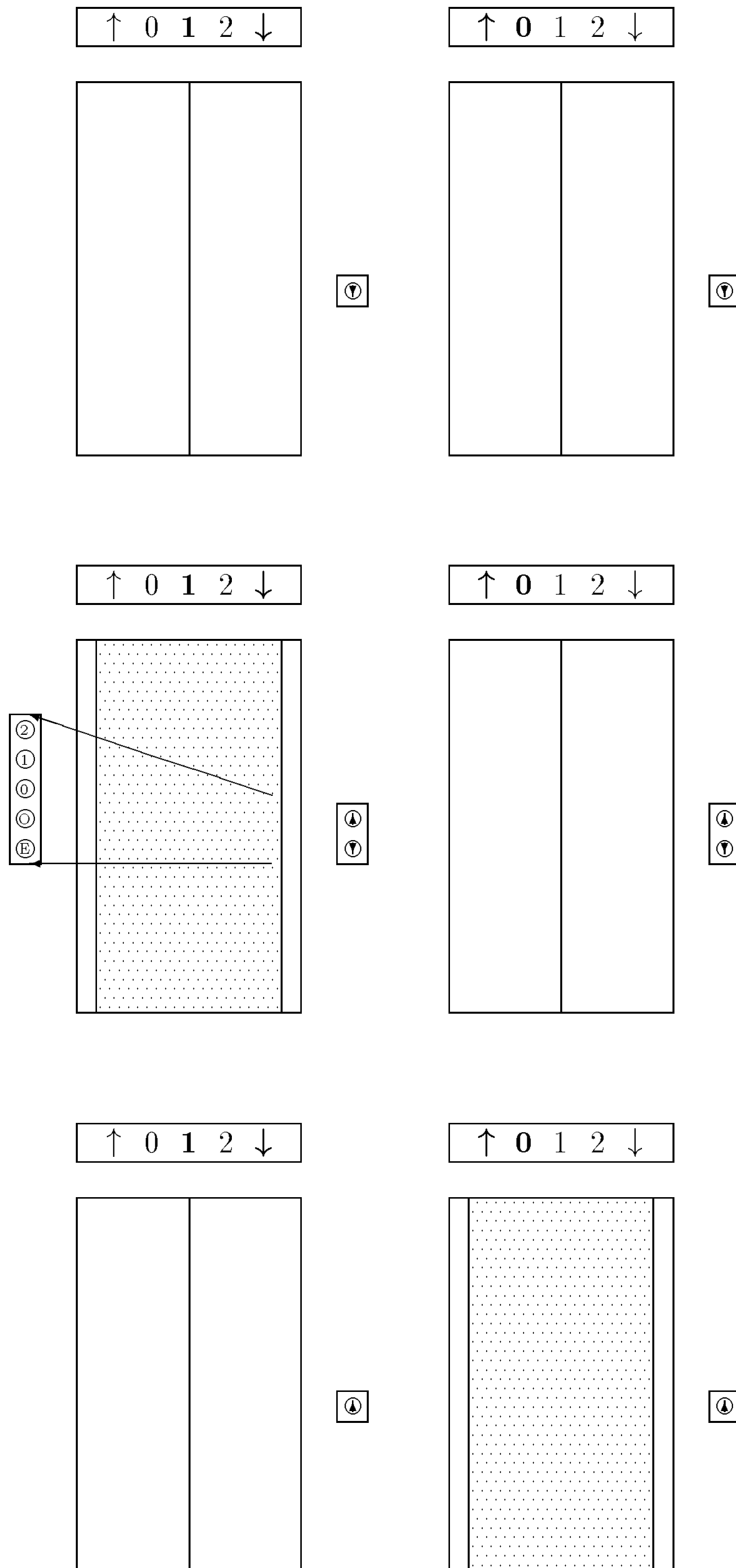


Figure 9.3: The users' view of a lift system.

9.2.1 The Interface

The controller is to govern the operation of n identical lifts servicing an m-storey building. It interacts with the environment through a number of external interfaces.

1. In each lift there is a control panel with one button for each floor. The buttons illuminate when pressed and remain illuminated until the lift visits the appropriate floor.

2. On each floor (except ground and top) there are two buttons, one to request a lift going upwards, the other to request a lift on its way down. Each of these buttons illuminates when pressed and remains on until a lift travelling in the appropriate direction visits the floor. It then goes off.

3. The position of each lift is controlled by interaction with a "motor unit". The motor unit may be instructed to move the lift up or down, or to keep it stationary. The position and direction of each lift is displayed on every floor. (This is a one of the less secretive lift systems).

4. Each lift has doors which open and close on signals from the controller. The doors must open when the lift visits a floor to collect or deposit passengers, and must be closed when the lift is in motion. If the doors encounter an obstruction whilst closing, they should re-open. There is also a button within each lift for signalling the doors to remain open.

5. In each lift there is an emergency button. When this button is pressed an alarm is raised and the lift goes out of service.

Figure 9.3 shows the users' view of a particular system which has two lifts to service three floors. The rest of the section formalises this view.

Position

Three predicates, $\mathtt{below}(\mathtt{l},\mathtt{f})$, $\mathtt{at}(\mathtt{l},\mathtt{f})$ and $\mathtt{above}(\mathtt{l},\mathtt{f})$, are used to fix the position of lift $\mathtt{l}$. They have the obvious meanings; that is, $\mathtt{below}(\mathtt{l},\mathtt{f})$ holds if lift $\mathtt{l}$ is below floor $\mathtt{f}$, $\mathtt{at}(\mathtt{l},\mathtt{f})$ holds if it is at floor $\mathtt{f}$, and $\mathtt{above}(\mathtt{l},\mathtt{f})$ holds if it is above floor $\mathtt{f}$. These three properties are mutually exclusive, of course, because a lift cannot be in two places at once!

$$\mathtt{forall}\ \mathtt{f} < \mathtt{m} : \mathtt{always}\ \mathtt{xor3}(\mathtt{below}(\mathtt{l},\mathtt{f}), \mathtt{at}(\mathtt{l},\mathtt{f}), \mathtt{above}(\mathtt{l},\mathtt{f})), \tag{9.1}$$

where the predicate $\mathtt{xor3}(\mathtt{p},\mathtt{p}',\mathtt{p}'')$ denotes the exclusive disjunction of its three arguments,

$$\mathtt{xor3}(\mathtt{p},\mathtt{p}',\mathtt{p}'') \stackrel{\text{def}}{=} (\mathtt{p} \vee \mathtt{p}' \vee \mathtt{p}'') \wedge \neg((\mathtt{p} \wedge \mathtt{p}') \vee (\mathtt{p}' \wedge \mathtt{p}'') \vee (\mathtt{p}'' \wedge \mathtt{p})).$$

This means that exactly one of $\mathtt{p}$, $\mathtt{p}'$ and $\mathtt{p}''$ is true. Exclusive disjunction of two arguments, $\mathtt{xor2}(\mathtt{p},\mathtt{p}')$, is the same as $\mathtt{xor3}(\mathtt{p},\mathtt{p}',\mathtt{false})$.

Motion

The movement of a lift is controlled by two signals, one to switch its motor on and off, the other to control the direction of travel. Predicates $\mathtt{motor_on(l)}$ and $\mathtt{motor_off(l)}$ determine whether the motor is on or off. They are, of course, mutually exclusive,

$$\mathtt{always}\ \mathtt{xor2}(\mathtt{motor_on(l)}, \mathtt{motor_off(l)}). \tag{9.2}$$

So too are the three directional attributes, $\mathtt{ascending(l)}$, $\mathtt{descending(l)}$ and $\mathtt{neutral(l)}$, which determine whether the lift is on its way up, down, or nowhere.

$$\mathtt{always}\ \mathtt{xor3}(\mathtt{descending(l)}, \mathtt{neutral(l)}, \mathtt{ascending(l)}). \tag{9.3}$$

A lift is said to be going up when it is ascending with its motor on, and going down when descending with its motor on.

$$\begin{aligned} \mathtt{going_up(l)} &\stackrel{\text{def}}{=} \mathtt{motor_on(l)} \wedge \mathtt{ascending(l)}, \\ \mathtt{going_down(l)} &\stackrel{\text{def}}{=} \mathtt{motor_on(l)} \wedge \mathtt{descending(l)}. \end{aligned}$$

Finer degrees of control, allowing speed variation and so on, are not considered.

Service

Lift $\mathtt{l}$ is said to provide a service at floor $\mathtt{f}$ when it is stopped at floor $\mathtt{f}$ ready to take on or discharge passengers.

$$\mathtt{service(l,f)} \stackrel{\text{def}}{=} (\mathtt{at(l,f)} \wedge \mathtt{motor_off(l)}).$$

Strictly speaking, lift $\mathtt{l}$ does not provide a service at floor $\mathtt{f}$ unless its doors are open, but the system will be constrained so that the doors must open when the motor is off and there are serviceable calls.

If an internal request for floor $\mathtt{f}$ is registered in lift $\mathtt{l}$, it is not cancelled until $\mathtt{service(l,f)}$ is true. Similarly an "up-request" demands service from a lift that is ascending, and a "down-request" from a lift that is descending.

$$\begin{aligned} \mathtt{up_service(f)} &\stackrel{\text{def}}{=} \exists \mathtt{l} < \mathtt{n} : (\mathtt{service(l,f)} \wedge \mathtt{ascending(l)}) \\ \mathtt{down_service(f)} &\stackrel{\text{def}}{=} \exists \mathtt{l} < \mathtt{n} : (\mathtt{service(l,f)} \wedge \mathtt{descending(l)}). \end{aligned}$$

These predicates determine when the corresponding calls, $\mathtt{Cu_f}$ and $\mathtt{Cd_f}$, are satisfied.

Buttons and Lights

The button to request floor $\mathtt{f}$ in lift $\mathtt{l}$ is denoted by the Boolean signal $\mathtt{B_{lf}}$, and its associated call light by the flag $\mathtt{C_{lf}}$. When the button is pressed, $\mathtt{B_{lf}}$ is true. This sets the corresponding flag $\mathtt{C_{lf}}$, which then remains true until lift $\mathtt{l}$ visits floor $\mathtt{f}$. At any time the outstanding internal requests in lift $\mathtt{l}$ are therefore to those floors $\mathtt{f}$ for which $\mathtt{C_{lf}}$ is true.

The call-buttons on each floor are represented by the variables $\mathtt{Bu_f}$, to request an "up-lift", and $\mathtt{Bd_f}$, to request a "down-lift". When true, these variables cause the call-flags $\mathtt{Cu_f}$ and $\mathtt{Cd_f}$ to be set, each of which is cancelled when a lift visits the appropriate floor and is travelling in the required direction.

There are two other buttons on the control panel in lift $\mathtt{l}$, one to signal an emergency, the other to open or re-open the lift doors. The emergency button in lift $\mathtt{l}$ sets the boolean flag $\mathtt{E_l}$. Whilst $\mathtt{E_l}$ is true lift $\mathtt{l}$ is deemed to be out of service, and is released from its obligation to service any outstanding internal requests. The emergency status of a lift has to be cancelled by some external agent, probably a service engineer in reality.

The signal $\mathtt{O_l}$ is true whenever the "open-door" button is pressed or the doors cannot close due to some obstruction. If lift $\mathtt{l}$ is stationary and $\mathtt{O_l}$ is true the lift doors open, but when the lift is moving $\mathtt{O_l}$ has no effect.

Calls

At each moment there may be a number of calls waiting to be serviced. If any of these outstanding calls can be serviced by lift $\mathtt{l}$ then $\mathtt{called(l)}$ is true; if any calls require it to move upwards then $\mathtt{called_up(l)}$ is true; and if any require it to move downwards then $\mathtt{called_down(l)}$ is true.

$$\begin{aligned}
\mathtt{called(l)} &\stackrel{\text{def}}{=} \exists \mathtt{f} < \mathtt{m} : \mathtt{request(l,f)}, \\
\mathtt{called_up(l)} &\stackrel{\text{def}}{=} \exists \mathtt{f} < \mathtt{m} : ((\mathtt{below(l,f)} \wedge \mathtt{request(l,f)}) \vee (\mathtt{at(l,f)} \wedge \mathtt{Cu_f})), \\
\mathtt{called_down(l)} &\stackrel{\text{def}}{=} \exists \mathtt{f} < \mathtt{m} : ((\mathtt{above(l,f)} \wedge \mathtt{request(l,f)}) \vee (\mathtt{at(l,f)} \wedge \mathtt{Cd_f})),
\end{aligned}$$

where the predicate $\mathtt{request(l,f)}$ tests whether or not there is a request to go to floor $\mathtt{f}$ that can at some time be serviced by lift $\mathtt{l}$,

$$\mathtt{request(l,f)} \stackrel{\text{def}}{=} (\mathtt{C_{lf}} \vee \mathtt{Cu_f} \vee \mathtt{Cd_f}).$$

These conditions are used to decide when and in which direction lift $\mathtt{l}$ should start moving.

Two other predicates, $\mathtt{stop_up(l)}$ and $\mathtt{stop_down(l)}$, determine when lift $\mathtt{l}$ should stop moving; that is, when it is in a position to service a call. Both predicates are true if lift $\mathtt{l}$ is at floor $\mathtt{f}$ and has an internal request to go to floor $\mathtt{f}$. Additionally, $\mathtt{stop_up(l)}$ is true if lift $\mathtt{l}$ is ascending and can service a call from floor $\mathtt{f}$ (subject to certain constraints on its operation), and $\mathtt{stop_down(l)}$ is similarly true if lift $\mathtt{l}$ is descending.

$$\begin{aligned}
\mathtt{stop_up(l)} &\stackrel{\text{def}}{=} \exists \mathtt{f} < \mathtt{m} : (\mathtt{at(l,f)} \wedge (\neg \mathtt{called_up(l)} \vee \mathtt{C_{lf}} \vee \mathtt{Cu_f})), \\
\mathtt{stop_down(l)} &\stackrel{\text{def}}{=} \exists \mathtt{f} < \mathtt{m} : (\mathtt{at(l,f)} \wedge (\neg \mathtt{called_down(l)} \vee \mathtt{C_{lf}} \vee \mathtt{Cd_f})),
\end{aligned}$$

Note that a lift should stop anyway, whether or not there are outstanding calls, if there are no further calls in its direction of travel.

Once lift l has stopped at some floor f it may remain there for as long as there is a serviceable call on that floor or the open-door button is depressed. These conditions are tested by the predicate stop(l).

$$\begin{aligned}\texttt{stop(l)} \stackrel{\text{def}}{=} \; & (\texttt{ascending(l)} \wedge \texttt{stop_up(l)}) \vee \\ & (\texttt{descending(l)} \wedge \texttt{stop_down(l)}) \vee \\ & (\texttt{O}_\texttt{l} \wedge \texttt{motor_off(l)}).\end{aligned}$$

When stop(l) is true the lift doors should open, and they should remain open until it ceases to hold. This constraint is not really strong enough as far as the open-door button is concerned, because switching the motor on is sufficient excuse to ignore it. In fact, the open-door button is not properly considered until section 9.2.4.

Doors

In this discussion only the lift doors are considered. It is true that there must also be doors on each floor to prevent passengers from falling down the lift shaft, but the mechanism for opening and closing the doors is assumed to be on the lift. The doors on each floor merely open and close with the lift doors when the lift is appropriately positioned.

For the moment let us conveniently forget that the doors take some time to open and close, and constrain them always to be either open or closed,

$$\texttt{always xor2(doors_open(l), doors_closed(l))}. \tag{9.4}$$

A more detailed representation of the door behaviour will be considered in due course.

9.2.2 The Specification

What properties should the lift system have? Obviously it is supposed to move people around the building without taking too long about it. But this section is concerned with trying to constrain the behaviour of the controller so that it does a reasonably good job without being too complex. The first few properties are essential to any reasonable system, the later properties reflect specific design decisions.

The most important property of a lift is that it takes passengers where they want to go; that is, all calls are eventually serviced. For example, if a request for floor f is registered in lift l, then lift l eventually arrives at floor f and stops,

$$\begin{aligned}&\texttt{forall f} < \texttt{m :} \\ &\quad \texttt{always if } \texttt{C}_\texttt{lf} \texttt{ then sometime (service(l,f))}.\end{aligned}$$

Likewise, a call from floor f is eventually rewarded with the appropriate service.

```
forall f < m :
  always {
    if Cu_f then sometime (up_service(f))
    ∧
    if Cd_f then sometime (down_service(f))
  }.
```

The trouble is that for most practical purposes these properties are much too weak. They allow a lift to remain idle for a very long time, so long that no passenger would have the patience to wait.[1]

Maximum Service Time

It would be much more useful to put an upper bound on the time taken to service each call. But unfortunately in a real lift system no absolute maximum can be put on the service time, because it is possible for a passenger to interfere with a lift's behaviour by keeping its doors open. A realistic specification should take account of this possibility, which is most simply done by counting only the time when the motor is on.

```
forall f < m : {
  always {
    if C_lf then
      service(l, f) within {len (s) when motor_on(l)}
    ∧
    if Cu_f then
      up_service(f) within {len (s) when motor_on(l)}          (9.5)
    ∧
    if Cd_f then
      down_service(f) within {len (s) when motor_on(l)}
  }
},
```

where the operator within was defined in section 9.1.5. The time limit s is the maximum time for which a lift can be moving before a call is serviced, which effectively limits how far the lift is able to go out of its way to service other calls. The value of s depends on the speed of the lift and the algorithm used to decide the order in which calls are answered.

Naturally, this constraint only applies when lift l is in service. When it is out of service, that is when E_l is set, nothing at all can be said about how it behaves.

[1] There is also a technical difficulty with termination because sometime, as we have it, is a strong operator which asserts that its argument surely does hold at some time.

In fact, none of the constraints (9.5)–(9.16) apply to broken lifts, but for the sake of brevity this is not explicitly re-stated in each constraint.

The problem has now been transformed into one of ensuring that the lift does not remain stationary forever. This is the purpose of the next constraint.

Maximum Waiting Time

Consider what happens when lift l is stopped with its motor off. What makes it ever start moving? Hopefully, when there are outstanding calls and it is not being held at a floor (by its doors being kept open, for example), then it will begin to move after waiting a maximum of w time steps for passengers to get on and off.

$$\begin{aligned}&\texttt{always}\\&\quad\texttt{if motor_off(l)} \wedge \texttt{called(l)} \wedge \neg\texttt{stop(l) then}\\&\qquad(\texttt{motor_on(l)} \vee \neg\texttt{called(l)} \vee \texttt{stop(l)})\ \texttt{within len (w)}.\end{aligned} \tag{9.6}$$

It is possible for the outstanding calls to be serviced other lifts, or for its doors to be held open before lift l ever gets started.

Maximum Transit Time

In order to calculate an upper bound for the maximum service time s, let us suppose that lift l should take no more than t units of time to travel between floors; that is,

$$\begin{aligned}&\texttt{forall f} < \texttt{m} - \texttt{1} : \{\\&\quad\texttt{always}\ \{\\&\qquad\texttt{if going_up(l)} \wedge \neg\texttt{below(l,f) then}\\&\qquad\quad\neg\texttt{below(l,f+1) within len (t)}\\&\qquad\wedge\\&\qquad\texttt{if going_down(l)} \wedge \neg\texttt{above(l,f+1) then}\\&\qquad\quad\neg\texttt{above(l,f) within len (t)}\\&\quad\}\\&\}.\end{aligned} \tag{9.7}$$

In other words, if lift l is moving upwards and it is at least as high as floor f, then within t units of time it will be at least one floor higher. Similarly, when going downwards it will descend by at least one floor in t steps.

Now observe that the specification could be realised by having each lift continually go up and down stopping at every floor, repeatedly making an entire round trip. In that case the maximum time spent in transit is

$$\texttt{s} = 2 \times (\texttt{m} - 1) \times \texttt{t}.$$

This is an upper bound on the maximum service time for any well designed system; the average service time ought to be much less than s.

Position and Direction

The relationships between the position of a lift and its direction of travel are just the obvious ones: If its motor is off then the lift should remain in the same place; if it is going up then it should be seen to go up, that is, its height must not decrease; and similarly, its height should not increase when it is going down.

```
forall f < m :
  keep
    if motor_off(l) then {
      if above(l, f) then next above(l, f) ∧
      if at(l, f) then next at(l, f) ∧
      if below(l, f) then next below(l, f)
    }
    ∧
    if going_up(l) then {
      if above(l, f) then next above(l, f) ∧
      if at(l, f) then next ¬below(l, f) ∧                    (9.8)
      if below(l, f) then next ¬above(l, f)
    }
    ∧
    if going_down(l) then {
      if above(l, f) then next ¬below(l, f) ∧
      if at(l, f) then next ¬above(l, f) ∧
      if below(l, f) then next below(l, f)
    }
  }
}.
```

It has been assumed here that a lift does not "jump" past floors; that is, it cannot be below a floor one moment and above it the next, and *vice versa*. This simplifies the decision of when to stop at a floor. It is not an essential assumption, but seems reasonable if the grain of time is sufficiently small and the measurement of position not too precise (see section 9.2.3).

Buttons and Lights

The way that calls are set and reset has already been described; that is to say, sometime after a button is pushed the appropriate light switches on and remains on until the call is serviced. But this informal description misses one aspect of the behaviour: What happens if a button is pressed when the corresponding floor is already being serviced? If a call is registered anyway the lift may be held at that floor with its doors open, assuming that the controller is so designed. But if no call is registered there is no possibility of the button having any effect.

It seems reasonable to register a call anyway when an up- or down-button is pressed, but not to register internal requests. This enables a person outside the lift to hold the doors open by pressing a call button, whereas a passenger already in the lift has the open-door button for this purpose. These properties have been observed to hold in a number of real lift systems.

The internal buttons in lift $\mathtt{l}$ therefore behave in the following way. Within some response time, $\mathtt{r}$, of a button being pressed (when the floor is not already being serviced) the call light illuminates, and once illuminated the light stays on until the floor is serviced. The cycle repeats indefinitely.

$$
\begin{array}{l}
\mathtt{forall\ f} < \mathtt{m}: \\
\quad \mathtt{loop}\,\{ \\
\quad\quad \mathtt{halt}\,(\mathtt{B_{lf}} \wedge \neg \mathtt{service(l,f)}) \wedge \mathtt{latch(r,} \neg \mathtt{C_{lf})}; \\
\quad\quad \mathtt{halt}\,(\mathtt{service(l,f)}) \wedge \mathtt{latch(r,C_{lf})} \\
\quad \},
\end{array}
\tag{9.9}
$$

where the predicate $\mathtt{latch(r,b)}$ denotes that the boolean expression $\mathtt{b}$ is set within $\mathtt{r}$ time steps,

$$
\mathtt{latch(r,b)} \stackrel{\text{def}}{=} \{\mathtt{b\ within\ len\,(r)}\} \wedge \mathtt{keep}\,\{\mathtt{if\ b\ then\ next\,(b)}\}.
$$

The call buttons for each floor behave in a similar way, but if a button is pressed continuously the light goes on no matter what. Up- and down-calls are set and reset as follows,

$$
\begin{array}{l}
\mathtt{forall\ f} < \mathtt{m}: \{ \\
\quad \mathtt{loop}\,\{ \\
\quad\quad \mathtt{halt}\,(\mathtt{Bu_f}) \wedge \mathtt{latch(r,} \neg \mathtt{Cu_f)}; \\
\quad\quad \mathtt{halt}\,(\mathtt{up_service(f)} \wedge \neg \mathtt{Bu_f}) \wedge \mathtt{latch(r,Cu_f)} \\
\quad \} \\
\quad \wedge \\
\quad \mathtt{loop}\,\{ \\
\quad\quad \mathtt{halt}\,(\mathtt{Bd_f}) \wedge \mathtt{latch(r,} \neg \mathtt{Cd_f)}; \\
\quad\quad \mathtt{halt}\,(\mathtt{down_service(f)} \wedge \neg \mathtt{Bd_f}) \wedge \mathtt{latch(r,Cd_f)} \\
\quad \} \\
\}.
\end{array}
\tag{9.10}
$$

Note, however, that it is not possible to make an up-request on the top floor, nor a down-request on the bottom. It is assumed that both these signals are always false

$$
\mathtt{always}\,(\neg \mathtt{Bu_{m-1}} \wedge \neg \mathtt{Bd_0}).
\tag{9.11}
$$

In reality one would expect the superfluous buttons to be absent.

Doors

For obvious reasons it is a good idea if, when lift l is moving, its doors are closed, and let us also stipulate that the doors are closed when it is stationary with no outstanding calls (and the open-door button not depressed),

$$\texttt{always if motor_on(l)} \vee (\texttt{neutral(l)} \wedge \neg \texttt{O}_\texttt{l}) \texttt{ then doors_closed(l)}. \tag{9.12}$$

This means that whenever the lift is not providing a service its doors are closed. In particular, pressing the open-door button when the lift is in motion has no effect.

Improving the Performance

Conditions (9.5)–(9.12) do not constrain the controller to be a particularly good one. They are met, as observed above, by a system in which each lift repeatedly makes an entire round trip, stopping at every floor. The next few constraints are aimed at eliminating some undesirable behaviours.

First, suppose that lift l is stationary. Constraint (9.6) determines the maximum time for which it can remain stationary if there are outstanding calls. However, it should not begin to move if there are no calls to service, or if it is held at a floor, and when it does begin to move it must decide whether to go up or down. The only constraint on this choice is that it should not head off upwards if there are no requests to go up, neither should it descend if there are no requests to do so.

```
keep
  if motor_off(l) then
    next {
      if stop(l) ∨ ¬called(l) then motor_off(l) ∧                    (9.13)
      if motor_on(l) ∧ ¬called_down(l) then ascending(l) ∧
      if motor_on(l) ∧ ¬called_up(l) then descending(l)
    }.
```

Note that when its motor is on a lift must be moving in some direction,

$$\texttt{always if motor_on(l) then } \neg\texttt{neutral(l)}. \tag{9.14}$$

It does not seem very useful to expend energy going nowhere!

Next, consider what happens once lift l has started to move. How does its position change, and when should it stop? Constraints (9.7) and (9.8) ensure that within t units of time it arrives at the next floor. It must then decide whether or not to stop. If there is a serviceable request outstanding the lift must stop, but it

should not stop if not required to do so.

```
keep {
  if going_up(l) then
    next
      if stop_up(l) then motor_off(l) else going_up(l)
  ∧                                                        (9.15)
  if going_down(l) then
    next
      if stop_down(l) then motor_off(l) else going_down(l)
}.
```

This simple method of deciding when to stop only works if, as directed by constraint (9.8), the lift cannot whizz past a floor without ever being `at` it. Otherwise it must be decided in advance when to stop.

Lastly, in the interests of fairness, the lift should keep going in the same direction for as long as possible; that is, if it is ascending then requests to continue upwards should be given priority over those that require it to go downwards, and *vice versa.* If it is idle then it should remain so until there are calls to answer.

```
keep {
  if ascending(l) then
    next
      if called_up(l) then ascending(l) ∧
  ∧
  if descending(l) then
    next                                                   (9.16)
      if called_down(l) then descending(l) ∧
  ∧
  if neutral(l) then
    next
      if ¬called(l) then neutral(l)
}.
```

This decision only comes into effect when the lift is stopped at a floor because constraint (9.15) prevents the lift from ever changing direction unless it is stopped.

The Complete Specification

The normal behaviour of lift `l`, when it is in service, is described by a combination of the above constraints. But when the emergency button is pressed those constraints no longer apply and the lift goes out of service. This is readily modelled with the `trap` construct, which was defined in section 9.1.4. The normal behaviour is then

captured in the predicate $\mathtt{in_service_spec(l)}$, where

$$\mathtt{in_service_spec(l)} \stackrel{\text{def}}{=} \mathtt{trap}\ \mathtt{E_l} : \{ \\ (9.1) \wedge (9.2) \wedge (9.3) \wedge (9.4) \wedge \quad \text{[interface]} \\ (9.5) \wedge (9.6) \wedge (9.7) \wedge \quad \text{[time limits]} \\ (9.8) \wedge \quad \text{[motion]} \\ (9.9) \wedge \quad \text{[internal calls]} \\ (9.12) \wedge \quad \text{[doors]} \\ (9.13) \wedge (9.14) \wedge (9.15) \wedge (9.16) \quad \text{[optimisations]} \\ \}.$$

Sometime after the emergency is over, when the flag $\mathtt{E_l}$ has been cancelled, lift `l` should return to normal service. But apart from that, nothing is said of what happens when the lift is out of service.

$$\mathtt{out_of_service_spec(l)} \stackrel{\text{def}}{=} \mathtt{E_l} \wedge \mathtt{fin}\,(\neg \mathtt{E_l}).$$

Lift `l` alternates between being in service and being out of service. Its whole behaviour is captured in the predicate $\mathtt{lift_spec(l)}$, where

$$\mathtt{lift_spec(l)} \stackrel{\text{def}}{=} \mathtt{loop}\,\{\mathtt{in_service_spec(l)}\,;\mathtt{out_of_service_spec(l)}\}.$$

The control panels on each floor, containing the up- and down-call buttons, are subject to the two constraints (9.10) and (9.11),

$$\mathtt{external_calls_spec(m)} \stackrel{\text{def}}{=} (9.10) \wedge (9.11).$$

Finally, the complete system is just the parallel composition of all `n` lifts and `m` floor control panels.

$$\mathtt{lift_sys_spec(m,n)} \stackrel{\text{def}}{=} \mathtt{external_calls_spec(m)} \wedge \mathtt{forall\ l < n : lift_spec(l)}.$$

Let us now construct a prototype implementation.

9.2.3 The Lift Controller

Although the specification determines the most important aspects of behaviour, it is not deterministic and so cannot be simulated in Tempura. In order to construct a prototype, every detail of its behaviour must be fixed.

The Control Logic

Consider first the behaviour of lift `l` as it moves up and down answering calls. There are three possibilities for the direction, and the motor can either be on or off. The lift can therefore be in any one of six major states, but as condition (9.14) affirms, when the motor is on the direction must be either up or down. In other words, one of the states should never occur. This leaves five major states:

1. `idle`: the lift is stationary with no outstanding calls.
2. `going_up`: the lift is moving upwards.
3. `going_down`: the lift is moving downwards.
4. `service_up`: the lift has stopped on the way up to service a call.
5. `service_down`: the lift has stopped on the way down to service a call.

These states, and the transitions between them, are defined below, and a diagram showing the state transitions can be found in figure 9.4.

When there are no calls to be serviced the lift remains on the same floor until a request is registered (9.8 and 9.13). Let us suppose that the lift is in neutral, then condition (9.16) implies that it remains in neutral until a call is registered. Suppose also that it gives priority to upward calls; that is, if `called_up(l)` returns `true`, the lift will next be going up, otherwise it will be going down (9.13 and 9.14). Finally, suppose that the lift moves immediately a call is registered, comfortably within any value of the delay `w` in condition (9.6). This behaviour is captured in the predicate `state0`:

$$\begin{aligned} \texttt{state0(l)} &\stackrel{\text{def}}{=} \texttt{halt}\,(\texttt{called(l)}) \wedge \texttt{keep}\,(\texttt{idle(l)});\\ &\qquad \texttt{if called_up(l) then state1(l) else state2(l)}\\ \texttt{idle(l)} &\stackrel{\text{def}}{=} \texttt{motor_off(l)} \wedge \texttt{neutral(l)}. \end{aligned}$$

The predicates `state1` and `state2` specify the behaviour of the lift when moving up and down, respectively.

Suppose that the lift is going up, then (9.8) and (9.15) force it to continue its ascent until either there is a call to be serviced on the current floor, or there are no outstanding calls to higher floors. In either case `stop_up(l)` = `true`. The predicate `state1` specifies this behaviour.

$$\begin{aligned} \texttt{state1(l)} &\stackrel{\text{def}}{=} \texttt{halt}\,(\texttt{stop_up(l)}) \wedge \texttt{keep}\,(\texttt{going_up(l)});\\ &\qquad \texttt{if called_up(l) then state3(l) else state4(l)}\\ \texttt{going_up(l)} &\stackrel{\text{def}}{=} \texttt{motor_on(l)} \wedge \texttt{ascending(l)}. \end{aligned}$$

If there are calls that require the lift to continue upwards, it then begins `state3` (9.16). In this state it allows passengers to get on and off, and then moves upwards again if required to do so. For the moment it is sufficient to assume that the lift pauses for one unit of time (`skip`), but this assumption will be revised shortly.

$$\begin{aligned} \texttt{state3(l)} &\stackrel{\text{def}}{=} \texttt{skip} \wedge \texttt{keep}\,(\texttt{service_up(l)});\\ &\qquad \texttt{if called_up(l) then state1(l) else state0(l)}\\ \texttt{service_up(l)} &\stackrel{\text{def}}{=} \texttt{motor_off(l)} \wedge \texttt{ascending(l)}. \end{aligned}$$

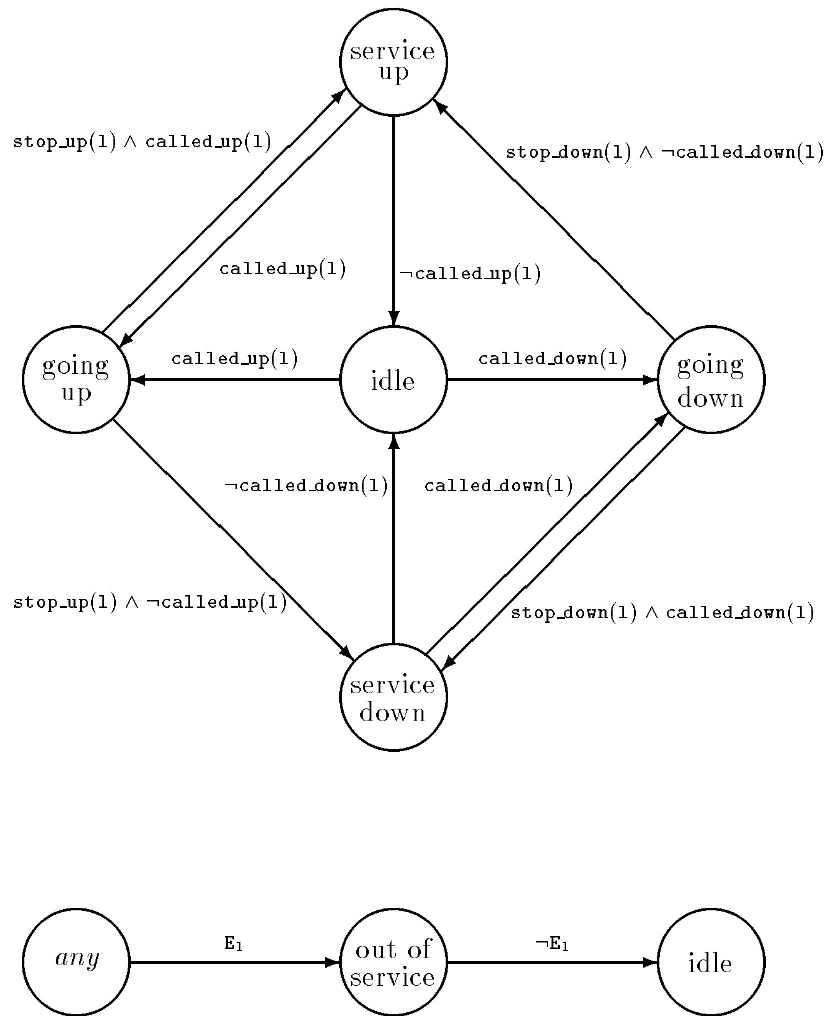


Figure 9.4: A state transition diagram for the lift system.

Predicates state2 and state4 are defined in the same way as state1 and state3, but with the direction of travel reversed. Thus, in state2 the lift is going down

$$\begin{aligned}
\texttt{state2(l)} &\stackrel{\text{def}}{=} \texttt{halt}\,(\texttt{stop_down(l)}) \wedge \texttt{keep}\,(\texttt{going_down(l)}); \\
&\quad\ \texttt{if called_down(l) then state4(l) else state3(l)} \\
\texttt{going_down(l)} &\stackrel{\text{def}}{=} \texttt{motor_on(l)} \wedge \texttt{descending(l)},
\end{aligned}$$

and in state4 it is stationary on the way up,

$$\begin{aligned}
\texttt{state4(l)} &\stackrel{\text{def}}{=} \texttt{skip} \wedge \texttt{keep}\,(\texttt{service_down(l)}); \\
&\quad\ \texttt{if called_down(l) then state2(l) else state0(l)} \\
\texttt{service_down(l)} &\stackrel{\text{def}}{=} \texttt{motor_off(l)} \wedge \texttt{descending(l)}.
\end{aligned}$$

The predicates called_down and stop_down were defined in section 9.2.1.

Buttons and Lights

Conditions (9.9) and (9.10) in section 9.2.1 ensure that within some maximum delay r of a button being pressed, or of a call being serviced, the appropriate light goes on or off. Suppose that the actual mechanism has a fixed delay, which for convenience is taken to be one unit of time, then the setting and resetting of calls in lift l is greatly simplified. It is described by the predicate internal_calls(l), where

$$\texttt{internal_calls(l)} \stackrel{\text{def}}{=} \texttt{forall f} < \texttt{m} : \texttt{C}_{\texttt{lf}}\ \texttt{gets}\ ((\texttt{B}_{\texttt{lf}} \vee \texttt{C}_{\texttt{lf}}) \wedge \neg\texttt{service(l,f)}).$$

The up- and down-calls are set and reset in much the same way, as the described by the predicate external_calls(m)

$$\begin{aligned}
\texttt{external_calls(m)} &\stackrel{\text{def}}{=} \texttt{forall f} < \texttt{m} : \{\texttt{up_calls(f)} \wedge \texttt{down_calls(f)}\} \\
\texttt{up_calls(f)} &\stackrel{\text{def}}{=} \texttt{Cu}_{\texttt{f}}\ \texttt{gets}\ (\texttt{Bu}_{\texttt{f}} \vee (\texttt{Cu}_{\texttt{f}} \wedge \neg\texttt{up_service(f)})) \\
\texttt{down_calls(f)} &\stackrel{\text{def}}{=} \texttt{Cd}_{\texttt{f}}\ \texttt{gets}\ (\texttt{Bd}_{\texttt{f}} \vee (\texttt{Cd}_{\texttt{f}} \wedge \neg\texttt{down_service(f)})).
\end{aligned}$$

Recall that pressing an up- or down-button inevitably sets the corresponding call flag, regardless of whether the lift is at the relevant floor already. This means that a lift may be held at a particular floor for as long as desired by continuously pressing the appropriate button.

Motor

The movement of lift l is controlled by sending signals to its motor unit. The direction of travel is determined by the setting of Dir,

$$\begin{aligned}
\texttt{ascending(l)} &\stackrel{\text{def}}{=} \texttt{Dir}_{\texttt{l}} = +1 \\
\texttt{descending(l)} &\stackrel{\text{def}}{=} \texttt{Dir}_{\texttt{l}} = -1 \\
\texttt{neutral(l)} &\stackrel{\text{def}}{=} \texttt{Dir}_{\texttt{l}} = 0
\end{aligned}$$

and the motor is switched on or off by setting Motor appropriately,

$$\begin{aligned} \texttt{motor_on(l)} &\stackrel{\text{def}}{=} \texttt{Motor}_\texttt{l} \\ \texttt{motor_off(l)} &\stackrel{\text{def}}{=} \neg\texttt{Motor}_\texttt{l}. \end{aligned}$$

Whenever the motor is switched on the lift moves in the direction specified by Dir.

The position of each lift is determined by the setting of two arrays of switches, Above and Below.

$$\begin{aligned} \texttt{below(l, f)} &\stackrel{\text{def}}{=} \texttt{Below}_\texttt{lf} \\ \texttt{above(l, f)} &\stackrel{\text{def}}{=} \texttt{Above}_\texttt{lf} \\ \texttt{at(l, f)} &\stackrel{\text{def}}{=} \neg(\texttt{Above}_\texttt{lf} \vee \texttt{Below}_\texttt{lf}). \end{aligned}$$

The idea is that these switches are positioned at appropriate points in the lift shaft, and toggle as the lift passes by. The switch $\texttt{Below}_\texttt{lf}$ should be just below floor f, and $\texttt{Above}_\texttt{lf}$ should be just above floor f.

When its motor is switched on, lift l travels up or down according to the setting of $\texttt{Dir}_\texttt{l}$ (9.8). If lift l takes one unit of time to travel between floors, and therefore has a maximum transit time, t, of one (9.7), its height goes up or down by one floor on every unit of time that the motor is switched on. A local register H is used to keep track of the current height.

$$\begin{aligned} \texttt{motor(l)} \stackrel{\text{def}}{=} \exists \texttt{H} : \texttt{forall f} < \texttt{m} : \{ \\ &\texttt{always (if at(l, f) then (H = f))} \wedge \\ &\texttt{Above}_\texttt{lf}\ \texttt{gets} \\ &\quad \texttt{(if motor_on(l) then f} < \texttt{H} + \texttt{Dir}_\texttt{l}\ \texttt{else Above}_\texttt{lf}) \wedge \\ &\texttt{Below}_\texttt{lf}\ \texttt{gets} \\ &\quad \texttt{(if motor_on(l) then H} + \texttt{Dir}_\texttt{l} < \texttt{f else Below}_\texttt{lf}) \\ \}. \end{aligned}$$

This is appallingly unrealistic, but it serves to get an initial working model. A more detailed description of the movement is developed in section 9.2.4 below.

Doors

Let us continue for the moment with the two-state doors. Each pair of doors can be controlled by a single boolean signal, $\texttt{Doors}_\texttt{l}$, with

$$\begin{aligned} \texttt{doors_open(l)} &\stackrel{\text{def}}{=} \texttt{Doors}_\texttt{l} \\ \texttt{doors_closed(l)} &\stackrel{\text{def}}{=} \neg\texttt{Doors}_\texttt{l}. \end{aligned}$$

Constraint (9.12) demands that the doors are closed when the lift is moving or is stationary in its idle state (unless the open-door button is depressed), and in this initial version the doors are closed at exactly those times. At all other times they are open.

$$\texttt{doors(l)} \stackrel{\text{def}}{=} \texttt{doors_closed(l) is (motor_on(l)} \vee \texttt{(neutral(l)} \wedge \neg\texttt{O}_\texttt{l})).$$

Section 9.2.4 introduces a more refined model of the doors, which takes account of the opening and closing states.

The Complete System

A first complete prototype can now be constructed by composing all of the parts described above. Each lift contains a call unit and a controller, which comprises some control logic, a motor unit, and a door unit. Under normal normal circumstances it goes up and down visiting floors according to the state transitions detailed above, and its behaviour is just the parallel composition of the various parts, as in the specification in_service_spec above. Assuming that the lift begins life in its idle state,

$$\begin{aligned}\mathtt{in_service(l)} &\stackrel{\text{def}}{=} \mathtt{trap}\ \mathtt{E_1} : \{\mathtt{internal_calls(l)} \wedge \mathtt{controller(l)}\},\\ \mathtt{controller(l)} &\stackrel{\text{def}}{=} \mathtt{state0(l)} \wedge \mathtt{motor(l)} \wedge \mathtt{doors(l)}.\end{aligned}$$

After the emergency button has been pressed, the lift is out of service, and the specification asserts that when it comes back into service the flag $\mathtt{E_1}$ has been reset. Suppose that it is in service again as soon as $\mathtt{E_1}$ is reset.

$$\mathtt{out_of_service(l)} \stackrel{\text{def}}{=} \mathtt{halt}\,(\neg\mathtt{E_1}).$$

Nothing is said about what happens to the lift when it is out of service.

Following the specification lift_spec, each lift is initially in service, but from then on alternates between being in service and being out of service.

$$\mathtt{lift(l)} \stackrel{\text{def}}{=} \mathtt{loop}\,\{\mathtt{in_service(l)}\,;\mathtt{out_of_service(l)}\}.$$

The lift always returns to normal service in its idle state.

Suppose that every lift is initially idle on the lowest floor with its doors closed, and that no calls are set,

$$\begin{aligned}\mathtt{init_sys(m,n)} &\stackrel{\text{def}}{=} \mathtt{forall}\ \mathtt{l} < \mathtt{n} : \mathtt{init_lift(l)} \wedge \mathtt{forall}\ \mathtt{f} < \mathtt{m} : \mathtt{init_floor(f)}\\ \mathtt{init_lift(l)} &\stackrel{\text{def}}{=} \mathtt{idle(l)} \wedge \mathtt{doors_closed(l)} \wedge\\ &\qquad \mathtt{forall}\ \mathtt{f} < \mathtt{m} :\\ &\qquad\quad \{\mathtt{Above_{lf}} = (\mathtt{f} < 0) \wedge \mathtt{Below_{lf}} = (0 < \mathtt{f}) \wedge \neg\mathtt{C_{lf}}\}\\ \mathtt{init_floor(f)} &\stackrel{\text{def}}{=} \neg\mathtt{Cu_f} \wedge \neg\mathtt{Cd_f}.\end{aligned}$$

The entire system is then just the parallel composition of $\mathtt{n}$ lifts and the call units on each floor beginning in the initial configuration above.

$$\mathtt{lift_sys(m,n)} \stackrel{\text{def}}{=} \mathtt{init_sys(m,n)} \wedge \mathtt{external_calls(m)} \wedge \mathtt{forall}\ \mathtt{l} < \mathtt{n} : \mathtt{lift(l)}.$$

It can be simulated directly in Tempura using the program $\mathtt{lift_simulation(m,n)}$, where

$$\begin{aligned}\mathtt{lift_simulation(m,n)} \stackrel{\text{def}}{=} &\ \exists \mathtt{B, Bu, Bd, C, Cu, Cd, E, O,}\\ &\qquad \mathtt{Motor, Doors, Dir, Above, Below} : \{\\ &\quad \mathtt{structure(m,n)} \wedge\\ &\quad \mathtt{init_sys(m,n)} \wedge\\ &\quad \mathtt{generate_inputs(m,n)} \wedge\\ &\quad \mathtt{lift_sys(m,n)} \wedge\\ &\quad \mathtt{always\ display_lifts(m,n)}\\ &\ \}.\end{aligned}$$

The simulation relies on three predicates not described above: `structure` defines the structure of the control signal paths, `generate_inputs` is required to simulate the pressing of the various buttons, and `display_lifts` outputs each state of the simulation in an easy-to-read form. Their definitions will not be given.

Although testing the prototype in this way increases one's confidence that it does behave correctly, it is still better to verify formally that it satisfies the specification. This entails proving that any interval generated by the implementation also meets the specification.

$$\models \mathtt{lift_sys(m,n)} \supset \mathtt{lift_sys_spec(m,n)}.$$

The proof can be broken down into a number of separate parts (see chapter 5). For instance, one might prove

$$\models \mathtt{external_calls(m)} \supset \mathtt{external_calls_spec(m)}$$

and

$$\models \mathtt{lift(l)} \supset \mathtt{lift_spec(l)}$$

separately. The latter property can be further subdivided, in much the same way as it was originally constructed, to make verification more manageable.

9.2.4 Some Improvements

In this prototype system, the relative speeds of different parts of the system are implausible, to say the least. For instance, a lift can travel between floors in the time that it takes to register a call. Furthermore, the description of the doors is much too crude. It assumes that they open and close instantaneously. This section offers improved descriptions of the motion and the doors.

Motor

The motion of a lift can be viewed as a sequence of steps on which it either moves up or down one floor or stays put, according to whether the motor is on or off. This is readily modelled with temporal projection by redefining the controller.

$$\mathtt{controller'(l)} \stackrel{\text{def}}{=} \mathtt{controller(l)}\ \mathtt{upon}\ \mathtt{motor_on(l)}\ \mathtt{do}\ \mathtt{move1(l)},$$

A step on which the lift moves is modelled by the predicate $\mathtt{move1(l)}$,

$$\mathtt{move1(l)} \stackrel{\text{def}}{=} \mathtt{if}\ \mathtt{ascending(l)}\ \mathtt{then}\ \mathtt{up1(l)}\ \mathtt{else}\ \mathtt{down1(l)},$$

and the predicates `up1` and `down1` respectively model the ascent and descent by one floor, so they must have the following properties:

$$\begin{aligned} &\models \quad \mathtt{up1(l)} \supset \forall \mathtt{f} < \mathtt{m}-1 : \mathtt{at(l,f+1)} \leftarrow \mathtt{at(l,f)} \\ &\models \quad \mathtt{down1(l)} \supset \forall \mathtt{f} < \mathtt{m}-1 : \mathtt{at(l,f)} \leftarrow \mathtt{at(l,f+1)}. \end{aligned}$$

If the lift is at floor f and starts going up then it first switches $\texttt{Above}_{\texttt{lf}}$, and then keeps going up until $\texttt{Below}_{\texttt{lf+1}}$ changes (and *vice versa* if the lift is descending). For convenience, it is assumed that this takes exactly t units of time.

$$
\begin{array}{rcl}
\texttt{up1(l)} & \stackrel{\text{def}}{=} & \texttt{forall f} < \texttt{m} : \{ \\
 & & \quad \texttt{if at(l,f) then} \ \{ \\
 & & \quad\quad \texttt{halt} \ (\neg \texttt{Below}_{\texttt{lf+1}}) \wedge \\
 & & \quad\quad \texttt{switch(1,Above}_{\texttt{lf}}) \wedge \\
 & & \quad\quad \texttt{switch(t,Below}_{\texttt{lf+1}}) \\
 & & \quad \} \\
 & & \quad \texttt{else stable} \ (\texttt{Above}_{\texttt{lf}}, \texttt{Below}_{\texttt{l(f+1)mod m}}) \\
 & & \} \\
\texttt{down1(l)} & \stackrel{\text{def}}{=} & \texttt{forall f} < \texttt{m} : \{ \\
 & & \quad \texttt{if at(l,f) then} \ \{ \\
 & & \quad\quad \texttt{halt} \ (\neg \texttt{Below}_{\texttt{lf-1}}) \wedge \\
 & & \quad\quad \texttt{switch(1,Below}_{\texttt{lf}}) \wedge \\
 & & \quad\quad \texttt{switch(t,Above}_{\texttt{lf-1}}) \\
 & & \quad \} \\
 & & \quad \texttt{else stable} \ (\texttt{Above}_{\texttt{lf}}, \texttt{Below}_{\texttt{l(f-1)mod m}}) \\
 & & \}.
\end{array}
$$

The predicate switch(i, B) just asserts that B is inverted at time i,

$$\texttt{switch(i,B)} \stackrel{\text{def}}{=} \texttt{len} \ (\texttt{i} - 1) \wedge \texttt{stable} \ (\texttt{B}) \ ; \texttt{B} := \neg \texttt{B} \ ; \texttt{stable} \ (\texttt{B}),$$

before and after time i it is kept stable.

When its motor is off, lift l is stationary the corresponding switches do not change. They must therefore be represented by frame variables (which in this context correspond to hardware latches).

Doors

Finally, let us define a more realistic model of the lift doors. As promised, this model will take account of the fact that the doors take some time to open and close, which in turn will make it possible for the open-door button to have an effect outside the idle state.

The doors of lift l must open whenever condition stop(l) obtains; that is, whenever the open-door button is pressed and the motor is off, or the lift arrives at a floor where there is a serviceable call outstanding. At all other times the doors must be closed. This is represented, just as the detailed motion was represented, by projecting the fine-grain behaviour onto the controller.

$$\texttt{controller}''\texttt{(l)} \stackrel{\text{def}}{=} \texttt{controller}'\texttt{(l) upon stop(l) do open_close_doors(l)}.$$

Here the predicate $\texttt{open_close_doors(l)}$ represents the opening and closing of the lift doors. The closing doors may be interrupted and caused to re-open if the open-door button or a suitable call button is pressed.

$$\begin{array}{lll}\texttt{open_close_doors(l)} & \stackrel{\text{def}}{=} & \texttt{repeat}\,\{\\ & & \quad\texttt{open_doors(l)};\\ & & \quad\texttt{hold_doors(l)};\\ & & \quad\texttt{trap stop(l)}:\texttt{close_doors(l)}\\ & & \}\,\texttt{until doors_closed(l)}.\end{array}$$

The predicates $\texttt{open_doors(l)}$, $\texttt{hold_doors(l)}$ and $\texttt{close_doors(l)}$ model the opening, open and closing states, respectively.

$$\begin{array}{lrcl}\models & \texttt{open_doors(l)} & \supset & \texttt{fin}\,(\texttt{doors_open(l)}),\\ \models & \texttt{hold_doors(l)} & \supset & \texttt{stable}\,(\texttt{Doors}_\texttt{l}),\\ \models & \texttt{close_doors(l)} & \supset & \texttt{fin}\,(\texttt{doors_closed(l)}).\end{array}$$

To represent the opening and closing states, let us redefine the signal $\texttt{Doors}_\texttt{l}$ to be a numeric value between zero, which represents the closed state, and some maximum value, $\texttt{d}$, representing the open state. When the doors are opening the value of $\texttt{Doors}_\texttt{l}$ increases up to $\texttt{d}$.

$$\begin{array}{lcl}\texttt{open_doors(l)} & \stackrel{\text{def}}{=} & \texttt{halt}\,(\texttt{doors_open(l)}) \wedge \texttt{Doors}_\texttt{l}\ \texttt{gets}\ \texttt{Doors}_\texttt{l} + 1\\ \texttt{doors_open(l)} & \stackrel{\text{def}}{=} & \texttt{Doors}_\texttt{l} = \texttt{d}.\end{array}$$

When the doors have opened, they are held open for a maximum time $\texttt{h}$ before being closed again. But if a request button is pressed while the doors are open, they begin to close immediately.

$$\texttt{hold_doors(l)} \stackrel{\text{def}}{=} \texttt{next}\,\{\texttt{timeout}(\texttt{h}, \exists \texttt{f} < \texttt{m} : \texttt{B}_\texttt{lf})\} \wedge \texttt{stable}\,(\texttt{Doors}_\texttt{l}).$$

When the doors close, $\texttt{Doors}_\texttt{l}$ decreases to zero.

$$\begin{array}{lcl}\texttt{close_doors(l)} & \stackrel{\text{def}}{=} & \texttt{halt}\,(\texttt{doors_closed(l)}) \wedge \texttt{Doors}_\texttt{l}\ \texttt{gets}\ \texttt{Doors}_\texttt{l} - 1\\ \texttt{doors_closed(l)} & \stackrel{\text{def}}{=} & \texttt{Doors}_\texttt{l} = 0.\end{array}$$

Finally, the doors remain static when $\texttt{stop(l)}$ is not true, so $\texttt{Doors}$ must also be a frame variable.

Figure 9.5 shows a small section from the output of a long simulation of a system of three lifts servicing an eight-storey building. The simulation used the improved model, each lift taking three units of time to travel between floors. The doors also take three units of time to open or close; that is, $\texttt{d} = 3$. In figure 9.5 the symbols

⎕, ⎕, ▮ and ■

represent the state of the doors, from fully closed to fully open. The internal requests in each lift are shown at the top of each state picture, and the symbols $\triangle$ and $\triangledown$ denote up- and down-calls.

Figure 9.5: Part of a simulation of the lift system.

9.3 Discussion

There are several ways in which the prototype can be improved. For instance, the controller does not give optimum performance insofar as it does not always minimise the waiting time for calls to be answered. In general, it is not possible to achieve optimum performance without prior knowledge of the pattern of calls, but it is possible to do better than has been done. For example, a lift in its idle state will always answer upward calls in preference to downward. To make the lift fairer the decision must be made on the basis of some additional factor, such as to answer first the call that will take the least (maximum) time to service. This means favouring the downward call if the lift is currently less than half way up, otherwise the upward. Here is a version of `state0` that does this:

```
state0(l)  ≝  halt (called(l)) ∧ keep (idle(l));
              if below(l, m/2) then
                if called_down(l) then state2(l) else state1(l)
              else
                if called_up(l) then state1(l) else state2(l).
```

More sophisticated strategies come readily to mind, but they are not discussed here. Anyway, it should not happen very often in practice that two calls are registered at exactly the same moment, so perhaps the unfairness is not worth worrying about.

A much more serious problem is that the controller may cause several lifts to compete to answer the same call while another goes unattended. This happens because the lift controllers act entirely independently of one another, and such behaviour could be avoided by the introduction of logic to coordinate the lifts. The logic would assign a particular lift to answer each up- or down-call. For instance, one might replace each of the call flags $\mathtt{Cu_f}$ and $\mathtt{Cd_f}$ by `n` separate call flags, one for every lift. But when the appropriate button is pressed only one of the flags should be set. The decision of which flag to set should be based on some attempt to minimise the service time.

Finally, no account has been taken of the fact that in practice the speed of a lift is not constant due, amongst other things, to the need to accelerate and decelerate. One might overcome this by explicitly modelling speed variations, which is not hard, but further complicates the specification.

Chapter 10

Communicating Processes

This chapter presents a stream-based communication mechanism for Tempura. The mechanism is based on two primitive operations, one to put data onto a stream, the other to get data from a stream. These simple operations may be used to construct communicating processes. Three examples are presented. The first, transferring a list, is used to introduce the ideas; the second is the prime sieve; and the last one is a system comprising a lexical analyser, a parser and an evaluator for simple arithmetic expressions. Finally, section 10.5 proposes a way to express interleaved execution in ITL.

At first sight it may seem that ITL and Tempura are only suited to the study of strongly synchronous systems of the kind described so far, but closer investigation reveals that this is not so. Although the logical model deals in states of the whole system, which suggests a close coupling between component parts, it is not hard to embed asynchronous systems within this framework.

There are essentially two ways to do this. One is to use temporal projection to describe different parts of a system according to different timescales, so that their local state sequences are interleaved in the global model. This is the subject of section 10.5. The approach taken in the first part of this chapter is much simpler. Here, one simply describes all parts of the system according to some global timescale, whilst recognising that individual processes do not actually have to run at exactly the same rates.

The global timescale is an abstraction. Depending on the application, it may be seen merely as a particular way of numbering the states of each process, or perhaps as an observation of the system according to some conceptual clock (a "wall clock" perhaps). But it should not be *assumed* to correspond directly to clock cycles on any particular machine.

To demonstrate that Tempura can handle asynchronous processes just as well as synchronous ones, a particular form of synchronised communication is implemented in Tempura. It is used in three examples. The first example illustrates the basic communication mechanism. The second is a standard example of irregular

A	C
[0, 1, 2]	[]
[]	[0, 1, 2]

(a)

A	C
[0, 1, 2]	[]
[1, 2]	[0]
[2]	[0, 1]
[]	[0, 1, 2]

(b)

A	B	C
[0, 1, 2]	–	[]
[1, 2]	0	[]
[1, 2]	–	[0]
[2]	1	[0]
[2]	–	[0, 1]
[]	2	[0, 1]
[]	–	[0, 1, 2]

(c)

Figure 10.1: Transferring a list of data.

communication, the prime sieve; and the last is a pipeline for parsing elementary arithmetical expressions.

There is nothing new about the communication mechanism; in fact, it is just about the most elementary protocol there is. But that is beside the point, because the main purpose of this chapter is to show that communication, like assignment, does not have to be wrapped up in a special theory, but fits comfortably into the logical framework we already have.

10.1 Message Passing

At an abstract level the transfer of a complete message between two processes is represented by a single temporal assignment, but if the communication bandwidth is limited a long message has to be divided into smaller units which are then transferred one at a time. A transfer of this kind typically goes via intermediate storage that is shared by both processes, a buffer or communication channel perhaps. Let us begin with a very simple example to illustrate the technique.

10.1.1 Transferring a List

The task in hand is to transfer a quantity of data from `A` to `C`, so the transfer must implement the temporal assignment

$$\mathtt{C} \leftarrow \mathtt{A}.$$

Suppose that the data is in the form of a list, and to be specific, suppose that `A` has the value [0, 1, 2] initially. Then a satisfactory transfer is shown in figure 10.1(a).

This transfer may be achieved by copying the elements of `A` one at a time, which takes three steps in all. On each step the first element of `A` is removed and appended

to C,

$$\mathtt{for\ 3\,times\ do\ A, C} \leftarrow \mathtt{tl(A), C\hat{}[hd(A)]}.$$

This is the situation in figure 10.1(b).

Finally, all communication can be via a shared buffer, B. In this case the transfer is represented as two parallel processes, one which puts successive elements of A into B, and another which takes them from B into C,

$$\mathtt{for\ 3\,times\ do\ put_elt(B, A)} \parallel \mathtt{for\ 3\,times\ do\ get_elt(B, C)},$$

The predicates put_elt and get_elt define the actions of sending and receiving a single element of data.

Before defining these predicates, a concrete representation must be chosen for the buffer B. The only important thing is that there should be a unique value to distinguish the empty buffer, so let us take this value to be the special token nil.

The send and receive operations can be implemented using a simple stop-and-wait protocol. To send a message, first wait for the buffer to empty, then write the first element of A,

$$\mathtt{put_elt(B, L)} \stackrel{\text{def}}{=} \mathtt{halt\,(B = nil)\,; B, L := hd(L), tl(L)}.$$

The receive operation is the other way around, first waiting for the buffer to fill and then removing the datum in preparation for the next,

$$\mathtt{get_elt(B, L)} \stackrel{\text{def}}{=} \mathtt{halt\,(B \neq nil)\,; B, L := nil, L\hat{}[B]}.$$

Figure 10.1(c) shows how this works on our example.

There are two things to note about this example. The first is that the variables A, B and C are assumed to be frame variables. If they are not, then extra assignments must be put in to maintain their values during idle periods. The second observation is that this example isn't very general. For a start, it has been assumed that both sender and receiver know in advance how much data is going to be transferred. That might be a reasonable assumption for the sender, but surely not for the receiver.

10.1.2 Termination

Generally, some convention is used to indicate termination. One that is quite common in practice is to reserve a special token, # say, to denote the end of the data, and that is the convention adopted here. Moreover, it is convenient when dealing with lists to assume that # is appended to the data prior to transmission.

The predicates list_to_buf(L, B) and buf_to_list(B, L) respectively describe a sender and receiver which use our convention.

$$\mathtt{list_to_buf(L, B)} \stackrel{\text{def}}{=} \mathtt{while\,(|L| \neq 0)\ do\ put_elt(B, L)}$$
$$\mathtt{buf_to_list(B, L)} \stackrel{\text{def}}{=} \mathtt{repeat\ get_elt(B, L)\ until\,(L_{|L|-1} = \#)}.$$

time	A	B	C
0	[0, 1, 2, #]	–	[]
1	[1, 2, #]	0	[]
2	[1, 2, #]	–	[0]
3	[2, #]	1	[0]
4	[2, #]	–	[0, 1]
5	[#]	2	[0, 1]
6	[#]	–	[0, 1, 2]
7	[]	#	[0, 1, 2]
8	[]	–	[0, 1, 2, #]

Figure 10.2: An example of communication via a unit buffer.

Parallel composition of the sender and receiver processes yields a list transfer program, $\texttt{transfer(A, B, C)}$, where

$$\texttt{transfer(A,B,C)} \quad \stackrel{\text{def}}{=} \quad \texttt{list_to_buf(A,B)} \parallel \texttt{buf_to_list(B,C)}.$$

But it only works provided that $\texttt{A}$ contains # as its last element, and that $\texttt{B}$ is initially $\texttt{nil}$ and $\texttt{C}$ is initially empty. Thus, it is claimed that

$$\models \quad \texttt{frame A,B,C} : \{\texttt{init(A,B,C)} \wedge \texttt{transfer(A,B,C)}\} \quad \supset \quad \texttt{C} \leftarrow \texttt{A},$$

where $\texttt{init(A, B, C)}$ checks the initial conditions,

$$\texttt{init(A,B,C)} \quad \stackrel{\text{def}}{=} \quad (\texttt{forall i} < |\texttt{A}| - \texttt{1} : \texttt{A}_\texttt{i} \neq \#) \wedge \texttt{A}_{|\texttt{A}|-\texttt{1}} = \# \wedge \texttt{B} = \texttt{nil} \wedge \texttt{C} = [\,].$$

For the list transfer example above, this program produces the behaviour shown in figure 10.2.

10.1.3 The Operations Put and Get

Future examples will need the more general send and receive operations defined by the predicates $\texttt{put(B, X)}$ and $\texttt{get(B, X)}$. These are similar to $\texttt{put_elt(B, L)}$ and $\texttt{get_elt(B, L)}$ above, but they send and receive single data rather than whole lists.

$$\begin{aligned}\texttt{put(B,X)} \quad &\stackrel{\text{def}}{=} \quad \texttt{halt (B = nil) ; B := X} \\ \texttt{get(B,X)} \quad &\stackrel{\text{def}}{=} \quad \texttt{halt (B} \neq \texttt{nil) ; B, X := nil, B.}\end{aligned}$$

On completion of $\texttt{get}$, the received value is held in $\texttt{X}$. The parallel composition of put and get operations implements an assignment; that is,

$$\models \quad \texttt{local B} : \{\texttt{B} \Leftarrow \texttt{nil} \parallel \texttt{put(X,B)} \parallel \texttt{get(B,X}')\} \quad \supset \quad \texttt{X}' \leftarrow \texttt{X}.$$

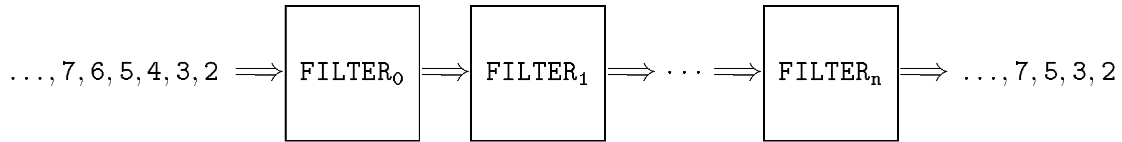


Figure 10.3: The prime sieve.

The two operations need not start simultaneously for this property to hold. For instance, a buffer process is defined below.

$$\texttt{copy(A,B)} \stackrel{\text{def}}{=} \texttt{local X : repeat \{get(A,X) ; put(X,B)\} until (X = \#)}.$$

It can be used to introduce an additional element of storage between sender and receiver, and so decouple them slightly,

$$\texttt{list_to_buf(A,B)} \parallel \texttt{copy(B,B}') \parallel \texttt{buf_to_list(B}',\texttt{C)}.$$

The producer can now run one element ahead of the consumer.

10.2 The Sieve of Eratosthenes

This communication mechanism will now be put to use in a parallel version of the so-called Sieve of Eratosthenes, the prime sieve. The aim is to discover all prime numbers less than a certain limit max(n) using a pipeline of n communicating processes, arranged as shown in figure 10.3.

The pipeline comprises a series of filters, each one removing multiples of a particular prime number from its input. The first filter removes all multiples of two, the second removes all multiples of three, and so on. The ith filter receives a stream of numbers from its left-hand neighbour, and passes on to its right-hand neighbour the same stream with all multiples of the ith prime number removed (the first prime number is 2).

10.2.1 Specification

The input to the pipeline is taken from a list N of successive natural numbers, [2..max(n)], and the output is placed in the corresponding list P which has all composite numbers filtered out. The maximum value in N is one less than max(n), where max(n) is the first non-prime number that would pass through an n-stage sieve. It is well known that max(n) is equal to the square of the (n + 1)st prime.

To see that this is the correct value, first notice that, although max(n) is not prime, it is not divisible by any prime less than the (n + 1)st, and so will not be

eliminated by the sieve. Moreover, any composite number less than max(n) must have a prime factor amongst the first n primes, and such numbers are supposed to have been filtered out by the sieve.

Taking this value for max(n), the output list, P, should finish up with the initial value of primes(N), where the function primes(N) returns a list of the prime elements of N. It may be defined recursively as follows:

$$
\begin{aligned}
\mathtt{primes(N)} &\stackrel{\text{def}}{=} \mathtt{if}\ |\mathtt{N}| = 0\ \mathtt{then}\ [\,] \\
&\quad \mathtt{else}\ \{ \\
&\quad\quad \mathtt{if\ is_prime(hd(N))} \\
&\quad\quad \mathtt{then\ cons(hd(N), primes(tl(N)))} \\
&\quad\quad \mathtt{else\ primes(tl(N))} \\
&\quad \} \\
\mathtt{is_prime(i)} &\stackrel{\text{def}}{=} \neg\exists \mathtt{j} : \{(1 < \mathtt{j} < \mathtt{i}) \wedge (\mathtt{i}\ \mathtt{mod}\ \mathtt{j} = 0)\}.
\end{aligned}
$$

Therefore a sieve having n filter processes, sieve(n, N, P), satisfies the following property:

$$\models \quad \mathtt{N} = [2..\mathtt{max(n)}] \wedge \mathtt{P} = [\,] \wedge \mathtt{sieve(n, N, P)} \quad \supset \quad \mathtt{P} \leftarrow \mathtt{primes(N)},$$

Now let us see how to implement the sieve in Tempura.

10.2.2 Implementation

The sieve is a parallel composition of the n filter processes, filter(i, A, B), together with two other processes, one to feed elements of N into the first filter, the other to gather up elements from the last filter into the list P. These two operations are defined by the predicates list_to_buf(L, B) and buf_to_list(B, L), which have already been described. The processes are connected by $n + 1$ buffers B_i.

$$
\begin{aligned}
\mathtt{sieve(n, N, P)} &\stackrel{\text{def}}{=} \mathtt{local}\ \mathtt{B} : \{ \\
&\quad \mathtt{list_to_buf}(\mathtt{N}, \mathtt{B}_0) \parallel \\
&\quad \mathtt{forpar}\ \mathtt{i} < \mathtt{n} : \mathtt{filter}(\mathtt{i}, \mathtt{B}_\mathtt{i}, \mathtt{B}_{\mathtt{i}+1}) \parallel \\
&\quad \mathtt{buf_to_list}(\mathtt{B}_\mathtt{n}, \mathtt{P}) \\
&\} \\
&\mathtt{where}\ \mathtt{list}(\mathtt{B}, \mathtt{n}+1) \wedge \mathtt{forall}\ \mathtt{i} < \mathtt{n}+1 : \mathtt{B}_\mathtt{i} = \mathtt{nil}.
\end{aligned}
$$

The sieve terminates when all subprocesses have finished.

10.2.3 The Filter Processes

The ith filter, which sifts out all multiples of the ith prime number, is specified by the predicate filter(i, A, B). It receives a stream of numbers from its left-hand neighbour in A, and passes on to its right-hand neighbour every element of this stream that is not divisible by p, as well as p itself.

time	Buf$_0$	Buf$_1$	Buf$_2$	Buf$_3$	Buf$_4$
	filter$_0$	filter$_1$	filter$_2$	filter$_3$	
0	—	—	—	—	—
1	2	—	—	—	—
2	—	—	—	—	—
3	3	2	—	—	—
4	—	—	—	—	—
5	4	3	2	—	—
6	—	—	—	—	—
7	5	—	3	2	—
8	—	—	—	—	—
9	6	5	—	3	2
10	—	—	—	—	—
11	7	—	5	—	3
12	—	—	—	—	—
13	8	7	—	5	—
14	—	—	—	—	—
15	9	—	7	—	5
16	—	—	—	—	—
17	10	9	—	7	—
18	—	—	—	—	—
19	11	—	—	—	7
20	—	—	—	—	—
⋮	⋮	⋮	⋮	⋮	⋮
228	—	—	—	—	—
229	116	115	—	113	—
230	—	—	—	—	—
231	117	—	115	—	113
232	—	—	—	—	—
233	118	117	—	—	—
234	—	—	—	—	—
235	119	—	—	—	—
236	—	—	—	—	—
237	120	119	—	—	—
238	—	—	—	—	—
239	#	—	119	—	—
240	—	—	—	—	—
241	—	#	—	119	—
242	—	—	—	—	—
243	—	—	#	—	—
244	—	—	—	—	—
245	—	—	—	#	—
246	—	—	—	—	—
247	—	—	—	—	#
248	—	—	—	—	—

Figure 10.4: A distributed version of the Sieve of Eratosthenes.

Its first task is to determine its own prime, whose multiples it must strike from subsequent output, and this is easy because the ith prime is simply the ith input to the process. Therefore, the first $i - 1$ numbers from the input stream, A, are copied directly onto the output stream, B, since they are all prime numbers. The predicate skip_to(i, X) takes care of this.

```
skip_to(i,X)  ≝  for i times do {
                   if X = # then empty
                   else {put(B,X) ; get(A,X)}
                 }.
```

If # is encountered before the ith input is reached; that is, if there are less than i inputs, then the predicate simply terminates.

When the predicate skip_to terminates, sift_multiples(X) takes over. The ith input, at this point held in the variable X, is saved in a static variable p. Then the processing enters a loop during which only those inputs that are not multiples of p are output, except for p itself which is output first.

```
sift_multiples(X)  ≝  local p : {
                        p ⇐ X;
                        while X ≠ # do {
                          if X mod p = 0 ∧ X ≠ p then empty
                          else put(B,X);
                          get(A,X)
                        }
                      }.
```

When the end of data is detected on the input stream, the filter process closes its output stream and terminates.

The ith filter process, filter(i, A, B) is formed by composing all these actions in the sequence described above.

```
filter(i,A,B)  ≝  local X : {
                    get(A,X);
                    skip_to(i,X);
                    sift_multiples(X);
                    put(B,#)
                  }.
```

A typical computation generated by the sieve is shown in figure 10.4, where four filters are used to generate all primes less than 121.

10.3 A Simple Parser

The final example is a pipelined parser for simple arithmetic expressions. This parser is designed to be part of a system for evaluating arithmetic expressions, receiving its

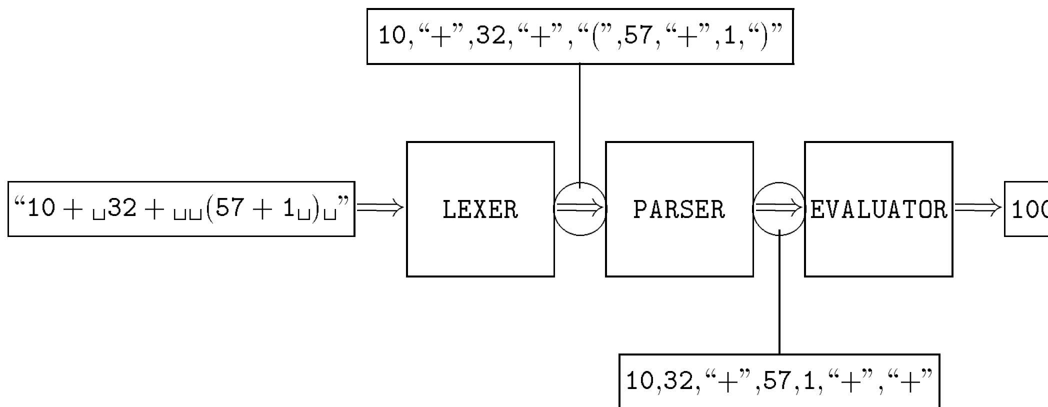


Figure 10.5: The pipelined expression evaluator.

input from a lexical analyser and sending its output to an evaluator. It accepts as input a stream of tokens comprising numbers, parentheses and operators, and on its output produces the reverse polish form of the input expression. The whole system is shown in figure 10.5

The parser operates on a pair of streams. Tokens are read from the input stream, `TS`, parsed into reverse polish form, and written to the output stream `PS`. The current token is always held in the frame variable `Token`. The first action, which gets the ball rolling, is to read a token from the input. When this has been done, the parsing begins, and continues until the whole expression has been read. At this point the input stream should be closed, in which case the output stream is also closed, otherwise a syntax error is flagged.

```
parser(TS,PS)  ≝  local Token : {
                    get_token();
                    parse_expr();
                    if Token = # then empty
                    else syntax_error(“Premature end of expression”);
                    put_polish(#)
                  }.
```

There are three subsidiary predicates here. Two, `get_token()` and `put_polish()`, are just specific uses of `put` and `get`, which were defined above. The former reads a token from the input stream and the latter writes a token to the output stream.

```
      get_token()  ≝  get(TS,Token)
put_polish(token)  ≝  put(PS,token).
```

The other predicate, `parse_expr()`, does the real work.

10.3.1 The Parsing Algorithm

A standard recursive descent algorithm is used for the parsing. The body of the parser is the predicate `parse_expr()`, which parses a complete expression. This may be an integer or an expression in parentheses, or it may be made up of two such primary expressions combined with an operator. To keep things simple, only the addition operator, "+", is permitted.

```
parse_expr()  ≝  parse_primary();
                 while Token = "+" do {
                   get_token();
                   parse_primary();
                   put_polish("+")
                 }.
```

The predicate `parse_primary()` parses a primary expression.

There are two legitimate kinds of primary expression. If the initial token is a number then the whole expression must be a number and there is nothing further to do.

```
parse_num()  ≝  put_polish(Token) ; get_token().
```

If, on the other hand, the initial token is a left parenthesis then the following expression must be recursively parsed and a check made for the terminating right parenthesis.

```
parse_parens()  ≝  get_token();
                   parse_expr();
                   if Token = ")" then get_token()
                   else syntax_error("Missing right parenthesis").
```

Anything else is a syntax error. Thus, `parse_primary()` is defined as follows:

```
parse_primary()  ≝  if Token = #
                    then syntax_error("Premature end of input")
                    else if is_num(Token) then parse_num()
                    else if Token = "(" then parse_parens()
                    else syntax_error("Bad primary").
```

It only remains to describe how errors are handled.

10.3.2 Error Handling

Errors can be handled in a number of ways, but one of the simplest options is to abort the parsing, output the appropriate error message, and then discard all the

remaining input. This is readily achieved by introducing a boolean flag, ErrorSet say, which is trapped in the top-level predicate of the parser, as follows:

$$\begin{array}{lcl}
\texttt{parser}'(\texttt{TS},\texttt{PS}) & \stackrel{\text{def}}{=} & \texttt{local ErrorSet} : \{ \\
 & & \quad \texttt{ErrorSet} \Leftarrow \texttt{false}; \\
 & & \quad \texttt{trap ErrorSet} : \texttt{parser}(\texttt{TS},\texttt{PS}); \\
 & & \quad \texttt{if} \ \neg\texttt{ErrorSet then empty else abort}() \\
 & \} & \\
\texttt{abort}() & \stackrel{\text{def}}{=} & \texttt{while Token} \neq \# \ \texttt{do get_token}(); \\
 & & \texttt{put_polish}(\#).
\end{array}$$

Recall that the construct trap b : p causes the program p to terminate prematurely if b becomes true. It was defined on page 144.

The error flag is set in the predicate syntax_error(text), which may be defined as follows:

$$\texttt{syntax_error}(\texttt{text}) \stackrel{\text{def}}{=} \texttt{output}(\texttt{text}) \wedge \texttt{ErrorSet} := \texttt{true}.$$

It is possible to make the parser even more robust by limiting the time it will wait for a message to be sent or received, thus protecting it against catastrophic errors elsewhere in the pipeline. This is quite simply achieved using the timeout mechanism, which was described in section 9.1.5. In general, the composition

get(B, X) ▌len (t)

waits at most t units of time for a message to arrive on the stream B.

10.4 The Complete Evaluator

For the sake of completeness, this section gives a brief account of the stream-based lexical analyser and the expression evaluator, which were alluded to earlier. These two can be hooked up to the parser as shown in figure 10.5, using the now familiar process list_to_buf(L, B) to feed the input expression into the lexical analyser.

Three streams are needed to connect the four processes: CS for the input to the lexer, TS to carry tokens between the lexer and the parser, and PS for the reverse polish output from the parser. The parallel composition

$$\texttt{list_to_buf}(\texttt{exp},\texttt{CS}) \parallel \texttt{lexer}(\texttt{CS},\texttt{TS}) \parallel \texttt{parser}(\texttt{TS},\texttt{PS}) \parallel \texttt{evaluator}(\texttt{PS},\texttt{answer})$$

then performs the complete evaluation of the expression exp, placing the result in the variable answer. Figure 10.6 shows a typical computation.

There is nothing noteworthy about either the lexical analyser or the expression evaluator. Both use standard algorithms.

time	CS	TS	PS
0	–	–	–
1	"1"	–	–
2	–	–	–
3	"0"	–	–
4	–	–	–
5	"+"	–	–
6	–	–	–
7	"␣"	10	–
8	"␣"	–	–
9	"␣"	"+"	10
10	–	–	–
11	"3"	–	–
12	–	–	–
13	"2"	–	–
14	–	–	–
15	"+"	–	–
16	–	–	–
17	"␣"	32	–
18	"␣"	–	–
19	"␣"	"+"	32
20	–	–	–
21	"␣"	–	"+"
22	–	–	–
⋮	⋮	⋮	⋮
30	–	–	–
31	"1"	57	–
32	"1"	–	–
33	"1"	"+"	57
34	–	–	–
35	"␣"	–	–
36	–	–	–
37	")"	1	–
38	–	–	–
39	"␣"	")"	1
40	–	–	–
41	#	–	"+"
42	–	–	–
43	–	#	–
44	–	–	–
45	–	–	"+"
46	–	–	–
47	–	–	#
48	–	–	#
49	–	–	#
50	–	–	–

Figure 10.6: Evaluating the expression: `exp` = "10 + ␣32 + ␣␣(57 + 1␣)␣" to get the result: `answer` = 100.

10.4.1 The Lexical Analyser

At the top-level, the lexical analyser is much the same as the parser. First, it reads a character from the input stream, then begins the actual analysis, after which the output stream is closed and the process terminates.

$$\texttt{lexer(CS,TS)} \stackrel{\text{def}}{=} \texttt{local Char} : \{\texttt{get_char()} \,;\, \texttt{lex_expr()} \,;\, \texttt{put_token(\#)}\}.$$

The most recently read character is held in the frame variable `Char`.

The body of the analyser is the predicate `lex_expr()`. This separates the input into numbers, which are built from a sequence of digits in the input, spaces, which are ignored, and operators, which are passed straight on to the parser.

```
lex_expr()  ≝  while Char ≠ # do {
                  if is_digit(Char) then lex_number()
                  else if is_space(Char) then lex_spaces()
                  else if is_operator(Char) then lex_operator()
                  else syntax_error("Unrecognised character")
               },
```

where the predicates `lex_number()`, `lex_spaces()` and `lex_operator()` handle the recognition of numbers, spaces and operators.

```
  lex_number()  ≝  local N : {
                      N ⇐ 0;
                      while is_digit(Char) do {
                        N := 10 × N + ascii(Char) − ascii("0");
                        get_char()
                      };
                      put_token(N)
                   }
  lex_spaces()  ≝  while is_space(Char) do get_char()
lex_operator()  ≝  put_token(Char) ; get_char().
```

The predicates `put_token()` and `get_char()` are just particular uses of `put(B,X)` and `get(B,X)`.

10.4.2 The Evaluator

The evaluator produces a single result, `answer`, from the reverse polish input. Its top-level is therefore slightly simpler, since there is no output stream to be closed when the evaluation is done.

$$\texttt{evaluator(PS,answer)} \stackrel{\text{def}}{=} \texttt{local Token} : \{\texttt{get_polish()} \,;\, \texttt{eval_expr()}\}.$$

The most recently read token is held in the frame variable `Token`.

The body of the evaluator, `eval_expr()`, is defined below. It uses a stack to evaluate the input expression. Numeric tokens are simply pushed onto the stack until the operator "+" is encountered, whereupon the top two elements of the stack are added together.

```
eval_expr()  ≝  local Stack : {
                   Stack ⇐ [];
                   while Token ≠ # do {
                     if is_integer(Token) then push(Stack, Token)
                     else if is_plus(Token) then add(Stack)
                     else syntax_error("Unknown operator");
                     get_polish()
                   };
                   if |Stack| = 1 then answer ⇐ Stack_0
                   else syntax_error("Incomplete expression")
                 }.
```

The subsidiary predicate `add(Stack)` pops the top two elements of the stack, adds them together, and pushes the result back onto the stack.

```
add(Stack)  ≝  if |Stack| ≥ 2 then
                  local m, n : {
                    pop(Stack, m);
                    pop(Stack, n);
                    push(Stack, m + n)
                  }
                else syntax_error("Too few arguments to add").
```

When the evaluator finishes there should be just the result left on the stack; otherwise the input expression was not complete.

10.5 Interleaving

Although our communication mechanism works correctly and shows that asynchronous processes can be represented in Tempura, there are two isuues of concern. One is that the states in the underlying model are global, so all the parallel processes share the same state sequence. This does not seem to be a good model of a situation in which the parallel processes are genuinely independent of one another, perhaps running on different machines. The other problem concerns the message passing mechanism. When a process is waiting for a message on a particular stream it is, in effect, busy waiting; that is, it checks the stream on every state.

One possible answer to both of these problems is to use temporal projection. For example, the process

true ; (true proj p) ; true

specifies that p is true on an arbitrary subsequence of the whole interval. Thus, if two such processes are combined in parallel

$$\{\texttt{true}\,;(\texttt{true}\;\texttt{proj}\;p_0)\,;\texttt{true}\}\parallel\{\texttt{true}\,;(\texttt{true}\;\texttt{proj}\;p_1)\,;\texttt{true}\}$$

their state sequences may be arbitrarily interleaved. In general, an interleaving operator, $p\;\texttt{interleave}\;p'$ may be defined such that

$$p\;\texttt{interleave}\;p' \;\stackrel{\text{def}}{=}\; (\texttt{next}\;p)\;\texttt{proj}\;(p\,;p').$$

Roughly, this means that an instance of p precedes every state in the execution of p', including the first. The formula p is used to pick out the states on which p' must be active. The examples below will make this clearer.

The interleaving operator can be used to describe processes that are genuinely asynchronous. Let us look at just two simple applications. The first is to use the above technique to represent a kind of timeslicing; the second is to express a rendezvous mechanism.

10.5.1 Simple Timeslicing

Perhaps the simplest form of interleaving is timeslicing where only one process is active at a time. Each process runs for a certain amount of time, then is suspended whilst another is given its share of the resources. Such a scheme requires a scheduler to "swap" the processes in and out. For instance, suppose that there are n processes, p_i for $i < n$, and that p_i executes during its "turn", when $\texttt{T} = i$. This is readily represented as

$$\texttt{forpar}\;i<n:\texttt{halt}\,(\texttt{T}=i)\;\texttt{interleave}\;p_i.$$

Process i is active on every state on which $\texttt{T} = i$ (until it terminates). The turn must be switched by the scheduler.

10.5.2 Rendezvous

A similar strategy may be used to represent processes that synchronise at certain *rendezvous* points, but otherwise execute independently. Suppose that two processes p_0 and p_1 rendezvous on a shared buffer $\texttt{B}$; that is, they synchronise when the buffer is full. One needs to ensure that the processes have in common the rendezvous states when the buffer is full; at other times their execution may be arbitrarily interleaved. This may be achieved by imposing the constraint that each process is active when the buffer is full, or in other words that the buffer is empty ($\texttt{B} = \texttt{nil}$) whenever a process is suspended. An interleaving with the formula $\texttt{keep}\,(\texttt{B} = \texttt{nil})$ does the trick:

$$\texttt{forpar}\;i<2:\texttt{keep}\,(\texttt{B}=\texttt{nil})\;\texttt{interleave}\;p_i.$$

Note that the processes may be active when the buffer is empty, but may *not* be suspended when it is full.

The rendezvous mechanism is non-deterministic and so cannot be executed in Tempura as things stand. Of course, the buffer could not be a frame variable in this formulation because the rendezvous is a zero-time communication as well as being non-deterministic. However, there does not seem to be any essential problem in extending Tempura to handle this kind of synchronisation. Moreover, it may be possible to express a similar idea within the deterministic framework of Tempura, but this needs to be further investigated.

10.6 Discussion

Moszkowski has previously considered the representation of communicating processes in Tempura [Mos86]. The treatment above improves on his attempt in a number of ways. Most importantly, it is far simpler. The communication mechanism is much more straightforward than the one Moszkowski originally proposed, and the process composition operator ($\|$) removes the need for his rather cumbersome use of signals for starting and stopping parallel processes. The error handling mechanism is also a considerable improvement.

How does our stream mechanism compare to the message-passing facilities in other languages? The most obvious comparison is with Hoare's specification language CSP [Hoa85] and the related programming language Occam [Inm84], both of which take the communication operations to be a primitive. In CSP, the statement $B!X$ puts the data X into the stream[1] B, and $B?X$ is used to get X from B. A difference between the communication primitives in CSP and the predicates *put* and *get* (as I have defined them) is that the put operation in CSP is blocking, in Tempura it is non-blocking. In other words, in CSP a put operation does not complete until the data has been received, but the predicate *put* completes as soon as the data has been written to the stream. It is easy to define in Tempura a version of *put* that is more like the CSP version:

$$\texttt{put}'(B, X) \quad \stackrel{\text{def}}{=} \quad B := [X]\,;\,\texttt{halt}\,(|B| = 0).$$

Likewise, it is possible in CSP to implement a non-blocking send by means of an intermediate buffer process.

A simple message-passing mechanism such as the one described in this chapter might be "packaged", just as in CSP, and the put and get operations made into language primitives. Syntactic rules could then be imposed to ensure, for example, that channels are correctly used or even, if desired, that asynchronous processes never communicate except through channels. However, there are advantages in having the details of the mechanism visible. For one thing it offers greater flexibility

[1] In CSP and Occam they are called channels

because it permits one to understand and control what is really happening. Thus, it is possible to reason about the real-time aspects of message passing in Tempura. For instance, in section 10.3.2 I showed how easy it is to associate a timeout with a particular operation. CSP throws away any chance of dealing formally with such ideas, because the operations (and indeed the whole language) abstracts from time.

Chapter 11

Epilogue

I claimed in chapter 1 that Tempura is a realistic programming language for designing and reasoning about computer programs, especially parallel and real-time programs, and the evidence presented in this dissertation supports that claim. It is particularly pleasing to have discovered that the inertial assumption of ordinary imperative programming may be represented semantically in ITL, and that operators for parallel composition, communication and exception handling appear naturally as a result.

Nevertheless, Tempura is not yet a language for programming realistic systems. For that, much more work is needed, especially on compiler development, on application-specific programming environments, on usable verification and transformation tools, and on outstanding semantic problems which have been neglected here.

11.1 Scaling Up

I have considered a number of small- and medium-sized examples in Tempura, but there is clearly no need to stop there. The approach really needs to be tried on examples of significant complexity. But first the programming environment must be improved.

11.1.1 Compilation

Although simulation is quite efficient using a Tempura interpreter, a compiler is really needed for serious program development. Compilation of Tempura is an interesting problem in its own right. One possible approach is suggested by recent research on compiling the synchronous languages Esterel and Lustre [BC84, CPHP87] into finite-state automata. This has been found to result in very efficient code.

11.1.2 Data Types

Before tackling larger examples, especially when compilation is considered, it is desirable to have a better worked-out type-system. As it stands, the type system is very rudimentary. For example, it excludes record and pointer variables which are are sure to be necessary both for convenience and efficiency. Moreover, type-checking is performed dynamically.

Unlike a functional language (say), where the type of an object is compelled to be static, Tempura variables may change in type as well as value over time, just as they sometimes do in ordinary imperative languages. For instance, a pointer might reference a number of objects of different types at various times during its life. Whether dynamic types are really necessary, or whether a simpler static type system would suffice, remains to be seen.

11.1.3 Debugging

The problems of debugging Tempura programs have not been investigated at all. Many of the problems are quite standard, of course, and by the nature of the language it is possible to provide more run-time checking in an interpreter than for other similar languages. Undefined or overwritten variables may readily be monitored and trapped, for example. Moreover, Tempura programs are deterministic and so any execution is repeatable. Perhaps it is possible to build really good debugging tools for Tempura.

11.2 Applications

The greatest strengths of Tempura lie in its formal treatment of time and concurrency. It might therefore be advantageous to specialise Tempura for applications in which these strengths are most required. Three principal areas of specialisation suggest themselves: hard real-time programming, parallel programming, and rapid prototyping.

11.2.1 Real-Time Systems

Hard real-time systems are those in which timing characteristics are of crucial importance. For example, an aircraft's flight-control system must respond to its inputs within tight real-time bounds. Most real-time languages, such as Ada, provide only a few *ad hoc* delay constructs without formal semantics, whereas in Tempura delay is an integral part of the semantics; programs are executed in virtual time. It is possible, in principle, to design a "stripped down" version of Tempura in which the virtual time units of the program correspond in some direct way to real-time

units of the underlying hardware. The mapping of virtual to real time units would be performed by a compiler. Thus, the statement `len`(t) would complete within t units of real time.

There are, of course, many problems to overcome if this idea is to work. For example, in the assignment $v := e$ there is obviously a limit to the complexity of the expression e that can be evaluated and assigned in unit-time, whatever the time units; so the compiler might need to reject programs with timing errors. It is also necessary to be able to predict accurately the time taken by the underlying hardware to perform each instruction, which suggests that a RISC machine might be a natural host.

Correctness is most important in hard real-time programming, especially when the programs are for controllers in safety-critical applications. Here, of course, Tempura may also claim a large advantage over ordinary real-time languages, since it has well-defined and relatively simple logical semantics. Since ITL is also well-suited to hardware specification, it would be interesting to represent and verify an entire system in ITL, from the hardware right up to the programming language. By verifying a total system in a single formalism, one might eliminate the "grey areas" that usually exist at the interface between verified hardware and verified software. The compiler in such a system would itself be boot-strapped in Tempura.

11.2.2 Parallel Programming

It is not difficult to see a language like Tempura being implemented on real parallel computers. There is a natural correspondence between the "and-parallelism" described in section 8.1 and massively parallel synchronous computers. Likewise, the parallel process constructor of section 8.2 matches the multiprocessor style of architecture, and the communication mechanism of chapter 10 fits well into message passing multicomputer architectures.

11.2.3 Rapid Prototyping

Tempura also has further potential as a rapid-prototyping language for designing parallel and real-time programs. It has been used in this way throughout this dissertation. However, in order to get a more flexible language for experimentation, it would be worth investigating the introduction of features such as unification and backtracking from logic programming. This approach has been tried with some success in the language Tokio [FKTM86], which is another executable sublanguage of ITL.

11.3 Verification and Transformation

I have presented some initial approaches to verification and transformation, but no viable tools for systematic use against large programs. What hope is there for mechanisation?

11.3.1 Automation

A tractable decision procedure for ITL is not a possibility. Moszkowski has shown that, although the properties of Tempura programs are decidable, any general decision procedure must be of non-elementary complexity [Mos83]. Nevertheless, there are subsets of ITL for which tractable decision procedures do exist. One such subset is propositional linear-time temporal logic (without chop), and simple parts of the verification process might therefore be automated. It remains to be seen whether this is of practical value.

11.3.2 Transformation

The transformation of programs to canonical form as described in section 5.2.2 is a form of symbolic execution, following as it does the same strategy as the interpreter. There is therefore a good chance that (at least part of) this process also might be automated and used as a technique for proving equivalence between programs.

Similarly, a number of researchers have devised systematic techniques to transform and synthesise certain types of programs [DB73, MW80, MW84], and it seems there is no reason why similar techniques should not be used on Tempura programs, and again they might be embedded in HOL.

11.4 Semantics

There are a number of outstanding problems concerning the properties and semantics of Tempura operators. Some of them are listed below.

11.4.1 Frame

As I pointed out in chapter 6, the properties of the frame operator have not been formally established. In particular, one would like to be sure that the operator gives the intuitively correct behaviour for all causal Tempura programs. For this it must be proved that if a program containing frame variables executes successfully then the execution generates the same behaviour that is defined by the frame operator; that is,

$$\models\ \exists \Delta : \{\phi(v, \Delta, p) \wedge \texttt{keep if } \neg\Delta \texttt{ then } v \leftleftharpoons v\} \quad \supset \quad \texttt{local } v : p,$$

as claimed on page 105.

11.4.2 Prefix

There is a technical difficulty in the definition of the prefix operator that has so far been glossed over. The difficulty is this. The formula `prefix` p was so defined that it must be possible to find an interval on which p holds, but it may be the case that one wants to use the prefix operator, for error-handling say, precisely as an escape "when something is about to go wrong". A trivial example of the problem is the formula below, which is logically false.

$$\texttt{empty} \land \texttt{prefix}\ \texttt{next}\,(\texttt{A} \neq \texttt{A}).$$

The prefixed formula is unsatisfiable, so there is no possible prefix subinterval on which the formula can hold. However, it may intuitively seem that the formula should be true; at least, one might expect the formula to execute correctly in Tempura.

11.4.3 Parallel Processes

I have already pointed out that a theory of parallel processes in Tempura must prevent the possibility of arbitrary interference between processes. Otherwise, as I showed on page 139, it may be possible to draw incorrect conclusions about a program. Thus, it is necessary to restrict the use of the parallel composition operator so that only certain well-defined interactions are allowed.

One allowed interaction is message passing using the mechanism of chapter 10. In order to reason effectively about parallel processes in Tempura a theory of communicating processes must be developed. It is expected that much of this theory would resemble parts of Milner's Calculus of Communicating Systems [Mil80].

Bibliography

[AM86] M. Abadi and Z. Manna. A timely resolution. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 176–186, Cambridge, Massachusetts, June 1986.

[AM87] M. Abadi and Z. Manna. Temporal logic programming. In *Proc. IEEE Symposium on Logic Programming*, 1987.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar85] H. Barringer. *Up and Down the Temporal Way*. Technical Report UMCS-85-9-3, University of Manchester, Manchester, England, September 1985.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314, AFIPS Press, Montvale, NJ, 1968.

[BC84] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448, Lecture Notes in Computer Science, number 197, Springer-Verlag, Berlin, 1984.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the thirteenth ACM symposium on the principles of programming languages*, St. Petersberg Beach, Florida, January 1986.

[Boc82] G. v. Bochmann. Hardware specification with temporal logic: an example. *IEEE Transactions on Computers*, 31(3):223–231, March 1982.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGM86] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, September 1986.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison Wesley, Reading, Massachusetts, 1988.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the Fourteenth ACM Symposium on the Principles of Programming Languages*, pages 178–188, Münich, West Germany, January 1987.

[DB73] J. Darlington and R. M. Burstall. A system which automatically improves programs. In *Proceedings of the Third International Conference on Artificial Intelligence*, pages 479–485, Stanford, California, 1973.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesise synchronisation skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[FKTM86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: logic programming language based on temporal logic and its compilation to prolog. In *Proceedings of the Third International Conference on Logic Programming*, London, July 1986.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, 19*, pages 19–32, American Mathematical Society, Providence, RI, 1967.

[Gab87] D. M. Gabbay. Temporal and modal logic programming. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 197–237, Academic Press, London, 1987.

[Gor87] M. J. C. Gordon. *HOL: A Proof Generating System for Higher Order Logic.* Technical Report 103, Computer Laboratory, University of Cambridge, England, 1987.

[Gre87] S. Gregory. *Parallel Logic Programming in Parlog.* Addison-Wesley, London, 1987.

[Hal85] R. W. S. Hale. Modelling a ring network in interval temporal logic. In *Proceedings of EuroMicro 85*, pages 77–84, North- Holland, Amsterdam, September 1985.

[Hal87] R. W. S. Hale. Temporal logic programming. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 91–119, Academic Press, London, 1987.

[Hal88] R. W. S. Hale. Using temporal logic for prototyping: the design of a lift controller. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Colloquium on Temporal Logic and Specification*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1988.

[HKP82] D. Harel, D. Kozen, and R. Parikh. Process logic: expressiveness, decidability and completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.

[HM87] R. W. S. Hale and B. C. Moszkowski. Parallel programming in temporal logic. In *Proceedings of PARLE 87*, Lecture Notes in Computer Science, number 259, Springer-Verlag, Berlin, 1987.

[HMM83] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proceedings of the 10-th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, number 154, Springer-Verlag, Berlin, 1983.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–581, 583, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

[HS86] W. D. Hillis and G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Inm84] Inmos Ltd. *Occam™ Programming Manual.* Prentice Hall International, London, 1984.

[Knu69] D. E. Knuth. *The Art of Computer Programming: Volume 1.* Addison-Wesley, London, 1969.

[Kow79] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

[Kro87] F. Kröger. *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

[Lam80] L. Lamport. The “hoare logic” of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[Lam83] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668, North-Holland, Amsterdam, 1983.

[MAJ62] J. McCarthy, D. J. Abrahams, and Edwards D. J. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.

[MH81] J. M. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Readings in Artificial Intelligence*, pages 431–450, Tioga Publishing Co., Palo Alto, California, 1981.

[Mil80] A. J. R. G. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, number 92, Springer-Verlag, Berlin, 1980.

[Mil84] A. J. R. G. Milner. A proposal for standard ml. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[Mos83] B. C. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, California, 1983.

[Mos85] B. C. Moszkowski. A temporal logic for multi-level reasoning about hardware. *Computer*, 18(2):10–19, February 1985.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[MW84] Z. Manna and P. L. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu83] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, Lecture Notes in Computer Science, number 224, Springer-Verlag, Berlin, 1983.

[Pri67] A. N. Prior. *Past, Present and Future.* Clarendon Press, Oxford, 1967.

[RB92] W. W. Rouse Ball. *Mathematical Recreations and Essays.* Macmillan, London, 1892.

[RH88] A. W. Roscoe and C. A. R. Hoare. Laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, September 1988.

[Roh87] J. S. Rohl. Towers of hanoi: the derivation of some iterative versions. *Computer Journal*, 30(1):70–76, 1987.

[Sha86] E. Y. Shapiro. Concurrent prolog: a progress report. *Computer*, 19(8):44–58, August 1986.

[Sho88] Y. Shoham. *Reasoning About Change.* MIT Press, Cambridge, Massachusetts, 1988.

[Tan83] C. Tang. Toward a unified logic basis for programming languages. In *Proceedings of the IFIP Congress 1983*, North-Holland, Amsterdam, 1983.

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematical Laboratory, Cambridge, England, 1949.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, Orlando, Florida, 1985.