Realtime Signal Processing

Dataflow, Visual, and Functional Programming

Hideki John Reekie

Submitted for the Degree of

Doctor of Philosophy at the

University of Technology at Sydney

in the

School of Electrical Engineering

September 1995

Abstract

This thesis presents and justifies a framework for programming real-time signal processing systems. The framework extends the existing "block-diagram" programming model; it has three components: a very high-level textual language, a visual language, and the dataflow process network model of computation.

The dataflow process network model, although widely-used, lacks a formal description, and I provide a semantics for it. The formal work leads into a new form of actor. Having established the semantics of dataflow processes, the functional language Haskell is layered above this model, providing powerful features—notably polymorphism, higher-order functions, and algebraic program transformation—absent in block-diagram systems. A visual equivalent notation for Haskell, Visual Haskell, ensures that this power does not exclude the "intuitive" appeal of visual interfaces; with some intelligent layout and suggestive icons, a Visual Haskell program can be made to look very like a block diagram program. Finally, the functional language is used to further extend dataflow process networks, by simulating timed and dynamically-varying networks.

The thesis thus draws together a number of previously-separate ideas: a reasonable expectation of efficient execution using established dataflow compilation technology; a powerful and high-level programming notation; and a block-diagram style interface.

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Benefits . . . 5
1.3 Overview of the thesis . . . 6
1.4 Previously published work . . . 8

2 Background Material **11**
2.1 Models of parallelism . . . 11
2.1.1 A "meta-model" of parallelism . . . 12
2.1.2 Implicit parallelism . . . 13
2.1.3 Data parallelism . . . 14
2.1.4 Control parallelism . . . 15
2.1.5 The Linda model . . . 17
2.1.6 Pipeline parallelism . . . 18
2.2 Functional programming in five minutes . . . 19
2.2.1 Objects and functions . . . 20
2.2.2 Bindings . . . 22
2.2.3 Patterns . . . 23
2.2.4 Currying and higher-order functions . . . 23
2.2.5 *let*, *lambda*, and *case* . . . 25
2.2.6 Polymorphism and type classes . . . 25
2.3 Evaluation mechanisms . . . 27
2.3.1 Graph reduction . . . 27
2.3.2 Strictness . . . 29
2.3.3 Parallel graph reduction . . . 30
2.3.4 Parallel data structures . . . 31
2.3.5 Functional operating systems . . . 32
2.3.6 Functional process networks . . . 34
2.3.7 Skeletons . . . 35
2.4 Real-time signal processing . . . 36
2.4.1 Discrete-time signals . . . 37
2.4.2 Streams and channels . . . 38
2.4.3 Functions and systems . . . 40
2.4.4 Digital signal processors . . . 41
2.5 Summary . . . 43

3 Dataflow Process Networks **45**
3.1 Related work 46
3.1.1 Computation graphs 47
3.1.2 Synchronous dataflow (SDF) 48
3.1.3 Kahn's process networks 51
3.1.4 Dataflow processes 53
3.1.5 Firing rules 55
3.2 Standard-form dataflow actors 57
3.2.1 Syntax 57
3.2.2 Desugaring 60
3.2.3 Semantics 61
3.2.4 Consumption and production vectors 66
3.2.5 Canonical SDF actors 69
3.3 Phased-form dataflow actors 71
3.3.1 Syntax 72
3.3.2 Phase graphs 76
3.3.3 Semantics 77
3.3.4 Cyclo-static and multi-phase integer dataflow 81
3.3.5 Execution mechanisms 83
3.3.6 Hierarchy and strictness 84
3.4 Summary 87

4 Visual Haskell **88**
4.1 Related work 89
4.2 An introduction to Visual Haskell 93
4.3 Visual syntax preliminaries 97
4.3.1 Visual elements 98
4.3.2 Specifying the visual syntax 99
4.3.3 A simple visual language 101
4.3.4 De-sugaring 104
4.4 The core syntax 104
4.4.1 Simple expressions 105
4.4.2 Structured expressions 108
4.4.3 Patterns 111
4.4.4 Bindings 113
4.4.5 Match phrases 113
4.5 Improving the visual syntax 116
4.5.1 Visual syntactic sugar 116
4.5.2 Iteration boxes 118
4.5.3 Unfolded higher-order functions 120
4.5.4 Wiring 121
4.6 Summary 121

5 Static Process Networks **123**
5.1 Related work 124
5.2 Vectors 126
5.2.1 The *Vector* datatype 126

5.2.2 Iterators . . . 127
5.2.3 Combiners . . . 131
5.2.4 Selectors . . . 132
5.2.5 Example: the Fast Fourier Transform . . . 135
5.3 Streams . . . 138
5.3.1 The *Stream* datatype . . . 139
5.3.2 Process primitives . . . 139
5.3.3 An example . . . 142
5.3.4 Process constructors . . . 144
5.4 Process network construction . . . 148
5.4.1 Simple combinators . . . 149
5.4.2 Simple linear networks . . . 150
5.4.3 Pipelines . . . 151
5.4.4 Meshes and systolic arrays . . . 153
5.4.5 Network construction in dataflow systems . . . 155
5.5 Process network transformation . . . 156
5.5.1 Type annotations . . . 157
5.5.2 Fusion . . . 158
5.5.3 Parallelisation . . . 160
5.5.4 Pipelining . . . 161
5.5.5 Promotion . . . 163
5.6 Summary . . . 165

6 Dynamic Process Networks **166**
6.1 Related work . . . 167
6.2 Timed signals and streams . . . 169
6.3 Functions on timed streams . . . 171
6.3.1 Basic functions . . . 171
6.3.2 Timed vectors . . . 173
6.3.3 Synchronous and timed streams . . . 175
6.4 Dynamic process networks . . . 177
6.4.1 Finite synchronous streams . . . 177
6.4.2 Dynamic process functions . . . 181
6.5 A digital music synthesiser . . . 182
6.5.1 Notes . . . 182
6.5.2 Envelopes . . . 183
6.5.3 Note generation . . . 185
6.5.4 A formant-wave-function note generator . . . 186
6.6 Summary . . . 189

7 Summary **191**
7.1 Contributions . . . 191
7.2 Further work . . . 193
7.3 Concluding Remarks . . . 194

A Haskell Code **206**

List of Figures

1.1 A conceptual framework for DSP programming 2
1.2 The arguments, summarised 6
1.3 An overview of thesis topics 7

2.1 A hierarchy of parallelism 12
2.2 Carriero and Gelernter's models of parallelism 13
2.3 More models of parallelism 16
2.4 A simple block diagram 40
2.5 A simplified DSP chip architecture 42

3.1 A simple dataflow network 45
3.2 A computation graph 48
3.3 A synchronous dataflow graph 49
3.4 A Kahn process network 52
3.5 Sugared syntax of a standard-form actor 58
3.6 Desugaring a standard-form actor 61
3.7 Instantiating an actor 63
3.8 The canonical SDF actors 69
3.9 Sugared syntax of a phased-form actor 73
3.10 The non-deterministic *sumsqrs* actor 74
3.11 The phased-form *delay* actor 75
3.12 Desugaring a phased-form actor 75
3.13 Phase graphs 77
3.14 Phase graph of *iota* 82
3.15 Deadlock of a hierarchical actor 85
3.16 A phased network example 86
3.17 The example phased network as a phased-form actor 86

4.1 The factorial function in Cardelli's language 90
4.2 The factorial function in ESTL 91
4.3 The factorial function in Visual Haskell 94
4.4 Patterns 95
4.5 The *map* function 95
4.6 Icons for some standard prelude data constructors 96
4.7 Icons for some standard prelude functions 96
4.8 Function composition 97
4.9 Simple visual elements 99

4.10 Haskell's abstract syntax 100
4.11 The visual syntax of a simple language 102
4.12 Sugaring the simple language 103
4.13 De-sugaring rules 104
4.14 Visual syntax: simple expressions 106
4.15 An example translation 108
4.16 Visual syntax: structured expressions 109
4.17 Examples of structured expressions 110
4.18 Visual syntax: patterns 112
4.19 Visual syntax: bindings 114
4.20 Visual syntax: match clauses 115
4.21 Sugaring rules 117
4.22 Illustrating type annotations 118
4.23 Illustrating iteration 119
4.24 Unfolded higher-order functions 120
4.25 Wiring 121
4.26 Mixing text and pictures 122

5.1 Unfolded vector iterators 129
5.2 Iterator type signatures 130
5.3 Combiner type signatures 131
5.4 Selector type signatures 132
5.5 Additional selector functions 133
5.6 Selectors as "wiring" 135
5.7 The 16-point FFT butterfly diagram 136
5.8 The FFT function 137
5.9 Parts of the 16-point FFT in Visual Haskell 138
5.10 Types of stream functions 141
5.11 A first-order recursive filter 143
5.12 Process constructor definitions 145
5.13 Process constructors 146
5.14 The FIR filter function 147
5.15 The n-th order recursive filter 148
5.16 Simple network-forming functions 149
5.17 Linear process networks 151
5.18 A pipeline process network 153
5.19 Mesh process networks 154
5.20 Illustrating network types 157
5.21 Process fusion 158
5.22 The transformed FIR filter 159
5.23 Horizontal parallelisation 162
5.24 Pipelining 163
5.25 Promotion 165

6.1 A simple digital gain control 166
6.2 Types of timed stream functions 171
6.3 Types of finite stream functions 178

6.4 More functions for making finite streams 180
6.5 Types of dynamic process functions 181
6.6 Note events and associated code 183
6.7 Ramp and envelope generators 184
6.8 Note generators 185
6.9 The top-level synthesiser 186
6.10 Sine-wave synthesiser output 187
6.11 Formant-wave-function tone generation 188
6.12 Formant-wave-function output 189

Acknowledgements

I would like to thank my academic supervisors, Professor Warren Yates, of the School of Electrical Engineering, University of Technology at Sydney, and Dr. John Potter, of the Microsoft Institute (formerly of the School of Computing Sciences, University of Technology at Sydney), for their advice, support, and patience. I am indebted to the three thesis assessors, Dr Chris Drane (University of Technology at Sydney), Dr Greg Michaelson (Heriot-Watt University), and Dr Andrew Wendelborn (University of Adelaide), for their time and effort reading and commenting on the submitted version of the thesis; I hope this final version adequately addresses the issues they raised. This work was supported by an Australian Post-graduate Research Award. Fudeko and Colin Reekie made the whole thing possible. Jon Hill read and offered useful comments on an early draft of this thesis. Matthias Meyer wrote the TMS320C40 prototype of SPOOK. Ken Dawson wrote the prototype Visual Haskell editor. Mike Colefax wrote the polyphonic version of the formant-wave-function music synthesiser. John Leaney offered enthusiasm and encouragement. Rick Jelliffe told at least one funny story. Morris the psycho-cat took no interest in any of this.

The two virtues of architecture which we can justly weigh, are, we said, its strength or good construction, and its beauty or good decoration. Consider first, therefore, what you mean when you say a building is well constructed or well built; you do not merely mean that it answers its purpose,—this is much, and many modern buildings fail of this much; but if it be verily well built; it must answer this purpose in the simplest way, and with no over-expenditure of means.

John Ruskin, *The Stones of Venice.*

Writers of books and articles should not use *we* in circumstances where the collective anonymity of the editorial of a newspaper is out of place. An author may, taking the reader with him, say *we have seen how thus and thus* ..., but ought not, meaning *I*, to say *we believe thus and thus*; nor is there any sound reason why, even though anonymous, he should say *the present writer* or *your reviewer*, expressions which betray his individuality no less and no more than the use of the singular pronoun. Modern writers are showing a disposition to be bolder than was formerly fashionable in the uses of *I* and *me*, and the practice deserves encouragement. It might well be imitated by the many scientific writers who, perhaps out of misplaced modesty, are given to describing their experiments in a perpetually passive voice, (*such-and-such a thing was done*), a trick that becomes wearisome by repetition, and makes the reader long for the author to break the monotony by saying boldly *I did such-and-such a thing*.

H. W. Fowler, *A Dictionary of Modern English Usage*

Chapter 1

Introduction

Real-time signal processing computation is a unique field. It has its own special class of applications—applications that demand extremely high computational performance and very fast real-time response. It uses specialised micro-processors—often several—with their own unique set of hardware design and programming tricks. It pursues performance at the expense of generality, maintainability, portability, and other generally-accepted measures of good software practice. And a running real-time signal processing program is the most exciting kind of computer program there is, because you can hear it, see it, and interact with it in ways not possible with other kinds of computer program.

Fortunately, the eagerness to sacrifice good practice in pursuit of maximum performance is tempered by a programming approach that promises both. This approach, going under the guise of "block-diagram development systems," is based on a simple metaphor: software modules can be interconnected by software channels in the same manner in which hardware units can be connected by cables carrying analog or digital signals. The concept dates back at least to the work by Kelly *et al* in 1961 [74]. In modern systems, visual interfaces add to the appeal of the metaphor: complete systems can be constructed by making connections on a computer screen.

The computational model on which these systems are based is called *pipeline dataflow*. The combination of a visual interface with pipeline dataflow is well-established in several fields, including signal processing [88, 12, 85], image processing and visualisation [112, 83], instrumentation [82], and general-purpose visual programming languages [99, 59]. Signal processing systems are based on a special class of pipeline dataflow, *dataflow process net-*

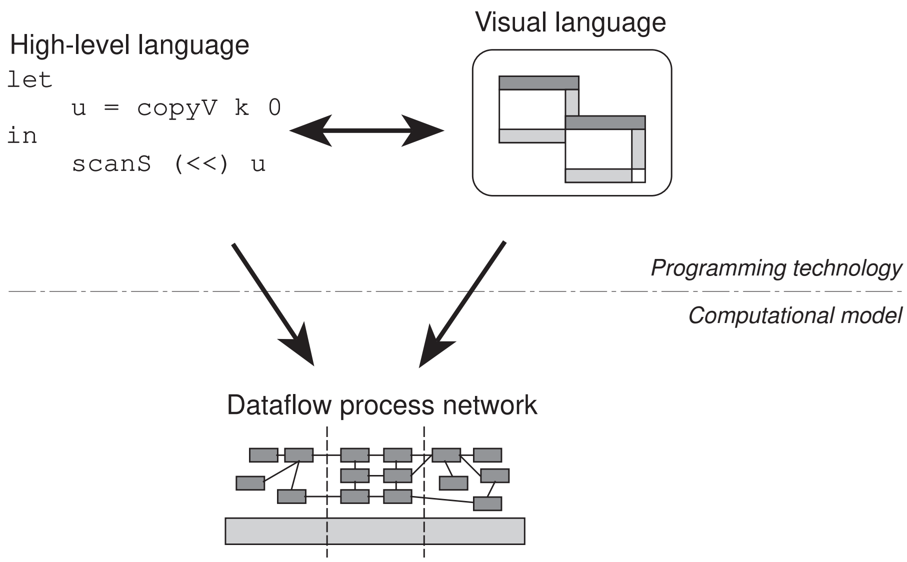


Figure 1.1. A conceptual framework for DSP programming

works (see chapter 3). Block-diagram systems, although well-established, are a practical compromise of usability and implementation technology, and lack some of the features of modern programming languages.

The aim of this thesis is to extend the visual-interface-dataflow-network style of computation. It does so by adding a third dimension as illustrated in figure 1.1. The underlying model of computation is still dataflow process networks; a programmer creates dataflow-network programs using either a visual language or a high-level language. There is more than appears on this diagram, though: the high-level language offers new flexibility and power to both the visual interface and the dataflow process network model. In this thesis, I will use Haskell as the high-level language; Haskell is a modern functional language that provides a common ground for research into functional languages and functional programming [44]. The visual language is Visual Haskell, a visual equivalent for Haskell of my own design.

In the next two sections I summarise the arguments in favour of adding this third dimension. A further two sections provide an outline of following chapters, and review previously-published papers and reports and their relation to this thesis.

1.1 Motivation

There were two main motivations behind the work in this thesis. Firstly, I wanted a way of expressing dataflow network programs that is more powerful and expressive than current block-diagram systems. And secondly, I wanted to show that a more powerful notation need not sacrifice the "intuitive" appeal of a visual interface.

In a block-diagram system, a programmer or engineer places icons representing processing modules, or *blocks*, onto the computer screen. Connections between input and output *ports* represent a flow of data from one block into another. The data flowing over these channels is called a *stream*. Input-output is handled in a uniform way: special blocks with no input ports represent input channels; blocks with no output ports represent output channels. Most systems support hierarchical construction of networks: a network with unconnected input or output ports can be named and then placed into another network. This is the dataflow network equivalent of procedural abstraction.

Block diagrams are conceptually familiar to engineers, and the visual interface is semantically simple and therefore provides little barrier to take-up, as learning a new textual language might. The computational model is, however, *limited* to pipeline dataflow. As a result, block diagram languages cannot express other kinds of computation. For example, although it is easy to express point-wise summation of two streams of numbers, it is not possible to express the addition of two numbers. For this reason, block-diagram systems supply a library of "primitive" blocks—waveform generators, filters, modulators, detectors, and transforms—with which more complex blocks and systems can be constructed. Any algorithm not supplied with the library can be coded as a primitive block by the user, in a conventional language such as C or Fortran.

This approach has two drawbacks. Firstly, algorithms coded in C or Fortran cannot be divided into smaller parts for simultaneous execution on multiple processors. Large primitive blocks—such as the Fast Fourier Transform—may limit throughput or reduce the effectiveness of load-balancing. The second objection is essentially one of elegance: the programmer is *forced* to program in two completely different languages, with two completely different semantics.

Functional programming languages seem a natural choice for coding dataflow network programs. Because pure functional languages have no assignment, we can think of functional programs as dataflow programs: in dataflow, the result of each block flows into the input of the next; in functional programs, the result of each function application is the argument to another function application. If this data is atomic—numbers and so on, or *tokens* in dataflow terminology—then this is like *atomic dataflow*.

In addition, pure functional languages are "lazy," so that evaluation is not performed until needed to produce the result of the program. Laziness is a nice property for a number of reasons. Firstly, it is necessary to guarantee referential transparency: an expression can always be substituted with an equivalent expression without changing program meaning. Secondly, it allows the use of some programming techniques that cannot be used in non-lazy languages—see [19, 66, 147] for examples of programs that can only be constructed in a lazy language. For our purposes, laziness makes it easy to build infinite data structures and therefore to simulate pipeline dataflow systems. Thus, the same language codes the internal computation of blocks as well as networks of blocks—there is no barrier between primitive and constructed blocks as in conventional block-diagram systems.

Functional languages are not, however, visual. Although there has been some work on visual functional languages, none have been widely adopted or implemented. Because the functional language approach to dataflow programming fits the dataflow model so well, I developed a visual notation for Haskell, called Visual Haskell. The aim of Visual Haskell is to provide an unambiguous visual equivalent of textual Haskell, so that programs can be translated back and forth between the two notations—this is represented by the double-ended arrow at the top of figure 1.1.

Thus, all blocks—both constructed *and* primitive—can be created and manipulated in both visual and textual languages. Although typically the visual language would be used at higher levels of the system and the textual language for lower-level algorithms, there is no compulsion to do so; the transition between the two styles of programming is seamless, and the programmer is free to choose whichever is the most appropriate. This "two-view" idea is not new: a multiple-view program development environment has been proposed for procedural languages [124]; and two-view document formatting systems seem a natural synthesis of the compiled and WYSIWYG styles of document production [23].

The choice of a lazy language is not without its drawbacks: there is an inevitable overhead associated with building a run-time representation of unevaluated expressions. This presents serious difficulties with computation *within* blocks, as real-time signal processing simply cannot afford to support any overheads other than those essential to producing the *result* of a computation.

1.2 Benefits

Having decided on the framework of figure 1.1, other interesting benefits become apparent. Because the visual language is general—that is, it is a visual notation for Haskell and it not particularly oriented towards signal processing—it promises to be a powerful tool for functional programming in general. The presentation in chapter 4 and in an earlier paper [117] are written assuming that Visual Haskell will be used as a visual functional language, not as a visual dataflow language.

The connection between functional languages and pipeline dataflow (more specifically, dataflow process networks—see chapter 3) provides a new implementation model for a particular class of functional programs. Although similar to some other approaches to evaluating functional programs (section 2.3.6), it promises efficient execution through the use of well-established dataflow scheduling and compilation techniques.

The high-level language provides the visual language and pipeline dataflow models with some powerful features. Existing block-diagram languages—and even general-purpose visual programming languages—do not support all of the features promised by Visual Haskell. (This is one of the reasons I developed Visual Haskell instead of using an existing visual language and combining it with dataflow networks.) *Polymorphism*, for example, allows functions (or networks) to be written that will accept different datatypes. In Ptolemy and SPW, for example, the programmer must choose an appropriate addition block: integer, floating, or complex. In Visual Haskell, the $(+)$ operator is overloaded on all these types. Typing is still static, so this feature does not require the overheads of run-time type-checking, as would be the case were a Lisp-like language adopted. *Higher-order functions* provide a very powerful mechanism for constructing and parameterising process networks. (A higher-order function takes a function argument, or delivers a function result.) This allows the expression of visual programs with "large thoughts" [139]. I explore this idea in section 5.4. A simple form of higher-order function mechanism has recently been added to Ptolemy [87], inspired by one of my earlier papers [114, 86].

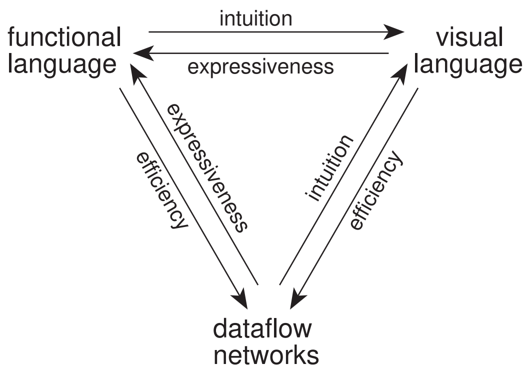


Figure 1.2. The arguments, summarised

An interesting possibility opened up by incorporating a functional language into the programming framework is that of *program transformation*. Program transformation is advocated in functional programming texts as a means of obtaining "efficient" realisation of programs from "inefficient" (but still executable) specifications [18]. In the context of dataflow networks, it offers a means of performing provably-correct alterations to the structure of a network (section 5.5). Since individual modules within the network are often interpreted as units of parallelism (section 2.1.6), the effect is to alter and control the degree and type of parallelism exhibited by a network.

Figure 1.2 summarises the preceding arguments in terms of the three dimensions of the new programming framework: the dataflow network model provides visual and functional programming with an efficient execution mechanism; the visual language adds "intuition" to functional and dataflow programming; and the functional language adds expressiveness to visual and dataflow programming. The result is a powerful framework within which to program and reason about dataflow network programs.

1.3 Overview of the thesis

This thesis covers a broad range of topics. To help explain the material covered and its inter-relationships, figure 1.3 shows dependencies between the main topics. The following outline proceeds chapter-by-chapter.

Chapter 2, *Background Material*, provides background material and surveys relevant to following chapters. Topics covered are models of parallel computation, functional program-

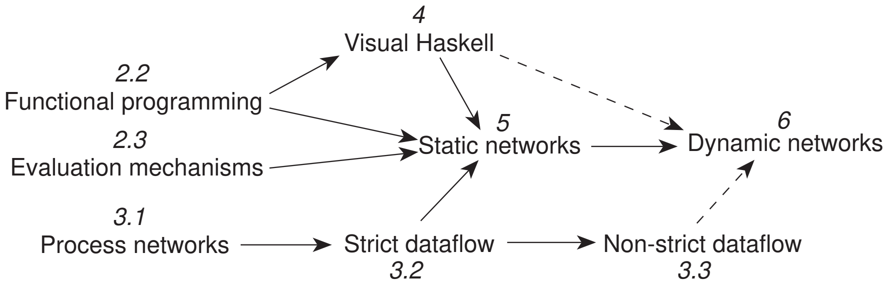


Figure 1.3. An overview of thesis topics

ming, evaluation mechanisms for functional languages, and real-time programs. These sections include much of the basic background material needed for the remaining chapters, although further reviews are given in each chapter.

Chapter 3, *Dataflow Process Networks*, presents a formalisation of dataflow actors and processes. This work builds on seminal work by Kahn on process networks [75], and on more recent work by Lee [87], to whom the term "dataflow process network" is due. The emphasis is on modelling and describing dataflow actors, rather than on the practical issues associated with implementation of dataflow programming environments. The formalisation leads to some new insights into the nature of dataflow actors. These dataflow actors are "strict," because they consume all needed input tokens before producing output tokens. The final section of this chapter introduces non-strict actors, which I call *phased* actors. This idea is quite novel and is still under development.

Chapter 4, *Visual Haskell*, formally describes Visual Haskell. Its syntax is given by a translation from Haskell's abstract syntax directly into a concrete visual syntax. The visual grammar uses some novel techniques to handle differences in the "style" of textual and visual syntaxes. A visual syntax for most of Haskell's expression and definition syntax is given; a notable omission is syntax for type and type class declarations. The final section of this chapter shows that the way I have specified visual syntax is far from perfect: much greater flexibility is needed to support more complex and varied forms of visual representation.

Chapter 5, *Static Process Networks*, builds on the previous two: dataflow processes are programmed in Haskell, and these programs illustrated in Visual Haskell. Firstly the relationship between Haskell functions and dataflow actors is made explicit. The remainder of the chapter demonstrates how Haskell's features increase the expressiveness of dataflow network programs: how higher-order functions can be used as a concise and powerful means of constructing and combining dataflow networks; and how program transformation techniques can be adapted to improve efficiency, or to tailor a program to a particular physical network.

Chapter 6, *Dynamic Process Networks*, takes a different approach: instead of writing functional programs to fit into the dataflow mold, I use this chapter to write Haskell programs in ways that may give insight into extending and improving the dataflow actor/process model. The key issue tackled here is that of representing *time* in dataflow networks. A secondary issue is that of dynamic networks: networks that change with time. In Haskell, we can represent evolving networks, but substantially more work is needed to find a way of translating this type of program into dataflow networks—not least because there has been very little work done on formal characterisations of evolving dataflow networks. There are two dependencies to this chapter shown as dotted lines in figure 1.3; these are areas for future work.

Finally, chapter 7, *Summary*, summarises the work of this thesis, and lists key contributions. As I have already indicated, there is enormous scope for further work, and I outline those areas I consider most promising for future research.

1.4 Previously published work

Some of the material in this thesis has previously been published in technical reports or conference papers. I have not included all of this material in this thesis; the reader interested in particular portions of elided work is directed to the electronic copies referenced in the bibliography, or to the list of papers at the address

`http://www.ee.uts.edu.au/~johnr/papers`

Towards Effective Programming for Parallel Digital Signal Processing [114] contains many of the key ingredients of this thesis. This report proposed that a functional programming language would be a good choice for programming block-diagram systems, and illustrated programs written using vectors and streams with an early version of the visual language of this thesis. It identified the usefulness of program transformation for parallel programming, and pipelined the FFT (Fast Fourier Transform) to illustrate.

Integrating Block-Diagram and Textual Programming for Parallel DSP [121] is essentially a brief summary of the above report, although it focuses more on the notion of the "two-view" development system.

Transforming Process Networks [122] continues the program transformation theme by pipelining an FIR (finite-impulse-response) filter. The transformation proceeds by fusing vector iterators (section 5.2.2) to form a single process, and then by using pipelining and promotion (sections 5.5.4 and 5.5.5) to produce a pipeline with controllable grain size. I suggested in the paper that less difficult means of finding program transformations will be required if program transformation is to become useful to programmers. I have not yet found this means, so have decided against reproducing the paper's key example here.

Process Network Transformation [123] is a condensed version of the above paper, but using a visual notation much closer to the current Visual Haskell. It identifies three classes of process network transformation—this has since become four (section 5.5).

Real-time DSP in C and Assembler [115] is a set of course notes on the TMS320C30 DSP, which may be of interest to the reader seeking an accessible introduction to the features of modern DSP devices.

Generating Efficient Loop Code for Programmable DSPs [119] is somewhat of a leap from the earlier papers, as it describes work aimed at compiling vector functions into efficient code for DSPs. It proposes an abstract DSP machine as a vehicle for DSP compilation, and describes the essential part of a code generation algorithm designed to exploit the high degree of instruction-level parallelism of DSPs. None of this work is included in this thesis.

The Host-Engine Software Architecture for Parallel Digital Signal Processing [118] is, again, totally unrelated to earlier papers: it describes a software architecture called SPOOK (Signal Processing Object-Oriented Kernel) that could best be described as an API-level (Application Programmer Interface) implementation of a dataflow engine for parallel DSP machines. The paper is a combination of architecture description and experience based on two implementations of the architecture: the first written by me for the TMS320C30 processor; the second largely by Matthias Meyer for the TMS320C40. The TMS320C40 implementation was in progress when the paper was written; for a detailed description of the final implementation see Meyer's report [96]. Again, I have decided against including this material in the thesis, as I feel it is tangential to its main theme.

Modelling Asynchronous Streams in Haskell [116] develops Haskell code for modelling timed streams. Two approaches are used: hiatons, which mark "empty" slots, and time-stamps, which mark the times of occurrence of tokens. Chapter 6 of this thesis is a complete revision of this paper; in particular, a new form of timed stream is developed, and the music synthesiser example is extended.

Visual Haskell: A First Attempt [117] is the only paper I have written devoted to explaining Visual Haskell, the final form of the visual language developed and refined over the last few years. Chapter 4 is a revised version of the core of this paper. The visual language is slightly improved, but the way in which the visual syntax is specified is very different, and I think simpler and more elegant. The original paper also contains motivating examples of possible uses of Visual Haskell, and a screen dump of the prototype Visual Haskell editor written by Ken Dawson [41].

Chapter 2

Background Material

Because this thesis covers quite a broad range of topics, I have collected into this chapter some useful background material. Firstly, I give a broad overview of models of parallel computation in terms of a simple "meta-model" of computation. Although parallel programming plays only a minor role in this thesis, one of the motivations for the pipeline dataflow model has always been to harness parallelism, and I think it important to place this model into context with other approaches to parallelism.

The next section is an introduction to functional programming with Haskell. Following that is a section on evaluation mechanisms for functional languages. Again, this serves to place work presented in later chapters into context; in particular, translating a functional program into a dataflow process network is similar to existing approaches to parallel evaluation of functional programs.

The context for this whole thesis is real-time signal processing, so section 2.4 explains some key concepts of real-time programming and of programmable DSP devices. This section is based largely on my own experience with real-time programs.

2.1 Models of parallelism

The pipeline dataflow model is inherently suited for parallel execution. It is, however, only one of several key models of parallelism, and it is important to place it into context. This section reviews key models of parallelism; they are shown as a hierarchy in figure 2.1.

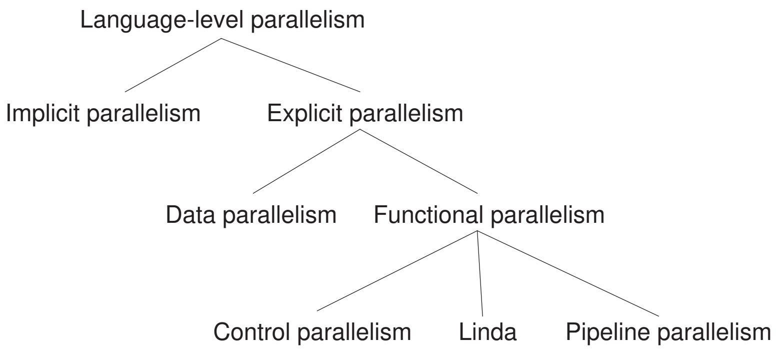


Figure 2.1. A hierarchy of parallelism

2.1.1 A "meta-model" of parallelism

Carriero and Gelernter suggest three "paradigms" of parallelism [32]:

Result parallelism Result parallelism focuses on the structure of the solution. Parallelism arises by simultaneous production of components of the result.

Specialist parallelism Specialist parallelism focuses on the kind of knowledge needed to produce a solution. Parallelism arises because many "specialists," each encapsulating a particular kind of knowledge, work simultaneously.

Agenda parallelism Agenda parallelism focuses on the steps to be taken to arrive at a solution. Parallelism arises by taking many (non-sequential) steps simultaneously.

To translate problems exhibiting these three types of parallelism into operating computer programs in Linda (see section 2.1.5), Carriero and Gelernter offer three corresponding program structures: live data structures, message passing, and distributed data structures respectively. These structures are distinguished by the relationship between *processes* and *data*. In a live data structure program, parallelism is structured around the data: an implicitly-defined process within each datum computes its value. A message-passing program, in contrast, consists of a collection of separate processes, each containing its own private data, and communicating via messages. A distributed data structure program does not have such a tight binding between a process and corresponding data; instead, many processes can share data.

Suppose now that we separate the *knowledge* of how to perform a computation, from the *thing* that performs the computation. These entities I call a *script* and an *agent*

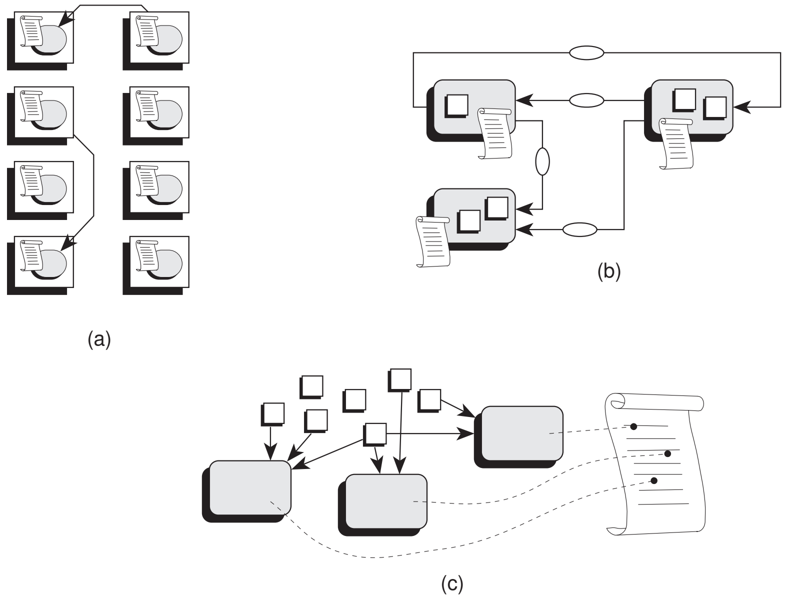


Figure 2.2. Carriero and Gelernter's models of parallelism: a) result parallelism; b) specialist parallelism; c) agenda parallelism

respectively—note that *process* = *script* + *agent*. This *agent-script-data* model serves as a meta-model for describing different models of parallelism. Figure 2.2 shows Carriero and Gelernter's three program structures, modified to show agents, scripts, and data—agents are grey ellipses or rounded rectangles; scripts are the curly glyphs that resemble curled sheets of paper; data are white squares. Figure 2.2a is the live data structure program: each agent has its own script, which it uses to produce the data enclosing it. Figure 2.2b is the message-passing program: again, each agent has its own script; agents contain their own private data and send data to other processes in messages, shown as white ellipses. Figure 2.2c is the distributed data structure program: unlike the other two, this program has only a single script (the "agenda"): all agents read the same script, each performing any available task and operating on shared data.

2.1.2 Implicit parallelism

Implicit parallelism is associated with program expressions with no data- or time-dependence between them. For example, in the expression

```
x = a * b + a * c
```

the two multiplications can be performed in parallel. Greater parallelism is available in loops, if different iterations of the loop can be performed in parallel. For example, in

```
for i = 1 to N do
    x[i] = y[i] * k;
```

all iterations can be performed in parallel. Parallelising compilers for conventional sequential languages analyse programs to find this kind of opportunity for parallel execution. The advantage of implicit parallelism is that (in theory, at least) the programmer need not be concerned with programming for a parallel machine, because the compiler will find and exploit available parallelism.

Implicit parallelism is less a characteristic of a programming language as it is of the compiler and run-time architecture. Still, side-effect-free languages are likely to contain more implicit parallelism than imperative languages because they lack artificial time-dependencies. Two examples of languages and architectures that support implicit parallelism are dataflow languages and architectures [1, 7] and multi-processor graph reduction of pure functional programs [105]. In both cases, the languages are side-effect free. However, Gajski *et al* point out that parallelising compilers can in fact perform better than single-assignment languages [49].

Implicit parallelism is also a key ingredient of modern single-processor compilers. Modern processors (including DSP micro-processors) exhibit significant instruction-level parallelism, which compilers must attempt to exploit.

2.1.3 Data parallelism

Data parallelism is parallelism at the level of elements of a data set. Most of the work on parallel computing for "scientific" applications uses data parallelism. Sipelstein and Blelloch call data-parallel languages "collection-oriented." In [129], they survey this class of languages and the ways in which they support data-parallel computation.

In a data-parallel language, an operation over all elements of a data set is invoked by a single function call or language construct. In the DataParallel C language [55], for example, one would calculate the inner product of two vectors by:

```
domain vpair { float x; float y; float t; } v[N];
```

```
float ip;
...
[domain vpair].{
    t = x * y;
    ip += t;
}
```

v contains N triples, each located on a different processor. x, y, and t thus have instances on each processor. *ip* refers to a single variable located on the system host processor. Execution of the code body first causes each instance of t to be updated with the product of the corresponding instances of x and y. Then all instances of t are summed and the result placed into *ip*.

Figure 2.3a illustrates data-parallelism using the agent-script-data meta-model. All agents read from the same script, and read each others' data when they need it. The greatest advantage of data-parallelism is its descriptive simplicity [132]: the programmer can easily control many thousands of processes because there is only one "thread of control" to manipulate.

Data parallelism is often associated with SIMD machines, while functional parallelism is often associated with MIMD machines. Although this is often the case, it is important not to associate a language model with the physical hardware on which a program might run, since the connection between a language-level model of parallelism and its supposed "obvious" implementation platform is rather tenuous. For example, Hatcher and Quinn [55] describe a data-parallel language compiler for MIMD machines, while Sabot [127] describes how an SIMD computer could be used to simulate an MIMD computer.

2.1.4 Control parallelism

Control parallelism is a form of functional parallelism characterised mainly by explicit communication ("message-passing") and synchronisation between processes. In effect, the programmer writes many separate programs; embedded within each program are commands for communication and synchronisation with other programs.

An example of a control-parallel language implementation for distributed-memory DSP machines is Parallel C [39], based very loosely on Hoare's CSP (Communicating Sequential

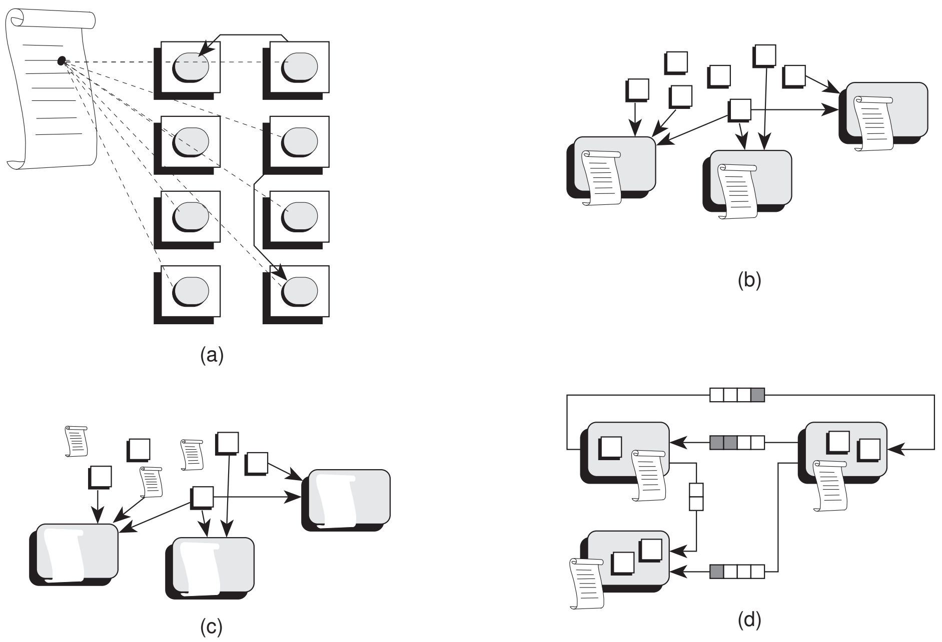


Figure 2.3. More models of parallelism: a) data parallelism; b) shared-memory control parallelism; c) Linda; d) pipeline parallelism

Processes) formalism [60]. The programmer writes a number of tasks—that is, independent C programs. A configuration file specifies the processor topology and the communications links between them, and assigns tasks to processors. Messages are exchanged by calling a message-passing library. For example, a sending task would contain code like this:

```
sometype message;
chan_out_message( sizeof(sometype), &message, outs[0] );
```

A receiving task would contain code like this:

```
chan_in_message( sizeof(sometype), &message, ins[2] );
```

Control parallelism is low-level—that is, the programmer interface is essentially that of the operations that can be performed directly by a target machine: transmission of messages between processors (distributed-memory machines), or synchronisation of memory accesses (shared-memory machines). It is criticised as being low-level and error-prone because the programmer must explicitly manage communication and synchronisation, and keep track of the internal states of many processors [55]. Although control-parallel programs are often machine-specific, there are some projects, such as PVM [136], which use a virtual machine abstraction to achieve architecture-independence.

There is no single agent-script-data model for control-parallel programs. On machines that support message-passing, the message-passing model of figure 2.2b is appropriate. On machines with shared memory, the model of figure 2.3b is more suitable; in this model, each agent has its own script, but all processes can access shared data.

It is important to distinguish between control parallelism (as I have characterised it here) and higher-level forms of functional parallelism. Proponents of the implicit or data-parallel language styles sometimes forget that there are other approaches to functional parallelism that provide better support for managing parallelism.

2.1.5 The Linda model

Linda [3] is a simple, elegant, architecture-independent model for MIMD computation. The basis of the Linda model is a global associative memory, or "tuple space." A task adds a tuple to tuple space by executing an *out* instruction:

```
out( "something", 14, 3.1415 );
```

A task can remove a tuple from tuple space by executing an *in* instruction:

```
in( "something", t, ?x );
```

The arguments to *in*, called an "anti-tuple," are a template for matching against tuples in tuple space. In this case, if the value of t is 14, then the tuple ("*something*", 14, 3.1415) is removed from tuple space, and the variable x in the reading task has the value 3.1415 assigned to it. If the anti-tuple matches no existing tuple, the reading task is suspended until a matching tuple becomes available.

A task can create new tasks with the *exec* instruction. For example, the statement

```
exec( "task", 3, ping() );
```

creates a "live tuple," which actively evaluates all of its fields. In this case, only the third field requires evaluation, so a new task is created to evaluate *ping()*. When the task terminates, the tuple turns back into a "data tuple," replacing the third element of the tuple with the value returned by *ping*.

Linda also has a *rd* instruction, which matches and reads a tuple but does not remove it from tuple space, and predicate versions of *in* and *rd*, called *inp* and *rdp*. *inp* and *rdp* behave as *in* and *rd* if they find a matching tuple, and return the value 1. If they fail to find a match immediately, they do not suspend the reading task, but return with the value 0.

Figure 2.3c is an approximate agent-script-data model of Linda. Each agent has an "slot" for a script: the agent takes a script out of tuple space and performs it; this corresponds to an *eval* operation. Once an agent has a script, it can read, remove, or add tuples. Note that, unlike the shared-memory model of figure 2.3b, an agent cannot *modify* a tuple in tuple space, other than by removing it and putting a new tuple into tuple space. (This is not shown in the diagram.) Linda thus avoids many of the synchronisation problems associated with conventional shared-memory models.

2.1.6 Pipeline parallelism

The functional programs in this thesis are based on the pipeline-parallel model, in which processes communicate only through FIFO-buffered channels. Programs written in this model do not contain explicit communication instructions, but implicitly specify communication by their construction. For example, the Haskell expression

```
mapS abs . scanS (+) 0
```

constructs a pipeline of two processes. The first computes the running sum of its input stream; the second calculates the absolute value of each element in its input stream. Communication between the processes is implicit in the fact that the result of the first is the argument to the second.

Pipeline parallelism is arguably just a special kind of message-passing control parallelism. I think the differences to general control-parallel programs are sufficient to make it a model of its own: processes can only send messages through channels, thus de-coupling senders from receivers; processes are not separate programs, but are just expressions in the program text; and communication between processes is buffered, providing further decoupling between sender and receiver processes. This kind of parallelism is therefore the functionally-parallel counterpart to data parallelism—it focuses on the essential aspects of parallelism without excessive concern for low-level detail.

Figure 2.3d shows the pipeline-parallel model; this figure is the same as the message-passing program of figure 2.2b, but explicitly shows the FIFO-buffered channels.

Pipeline parallelism also includes certain "dataflow" programming languages. Lucid [144] is a (first-order) pipeline-parallel language. In Lucid, the two-process pipeline above could be written

```
absolutes(runner(x))
    where
      runner(x)    = runner where
                     runner = 0 fby (x + runner)
                     end;
      absolutes(x) = abs(x);
    end;
```

In Lucid, all primitive operations are extended point-wise over streams; thus, the + operator sums corresponding elements of two streams. The *fby* operator produces a stream containing the first element of its left argument, followed by its right argument. So, if $x = [1, 2, 3, 4]$, then *runner* produces zero followed by itself summed with x—that is, $[0, 1, 3, 6, 10]$.

The dataflow process network model [87] is lower-level, since communication is explicit. A process is formed by repeatedly firing an "actor"; a complete program consists of a network of actors. Dataflow process networks are examined in chapter 3.

2.2 Functional programming in five minutes

Functional programming languages are "higher-level" than more conventional imperative languages. There have been many persuasive arguments advanced for functional programming languages in general [8], and lazy functional languages in particular [64, 139, 62].

A recent study indicates that at least some of the claimed advantages of functional languages—brevity, rapidity of development, and ease of understanding—can be confirmed [63]. The study compares several languages, including C++ and Ada, in a substantial rapid prototyping exercise. Several metrics were used to compare the solutions given; the Haskell solution was one of the highest-rated.

This section introduces functional programming using Haskell. Haskell is quite a large language, and so I have omitted several of its more complex features: separate modules and data-hiding, array syntax, list comprehensions, and user-defined operators.

Haskell has a library of types and functions contained in its "standard prelude." The standard prelude is a library of code modules that Haskell implementations are expected to support; because of their importance, a compiler is allowed to "understand" the contents of these modules in order to generate more efficient code. I give a cursory overview of standard prelude functions and types.

In later chapters, I will sometimes use a "typeset" version of Haskell for improved readability. The differences to standard Haskell are: the use of Greek characters as type variables; $\lambda x \,.\, e$ instead of `\x -> e`; $\rightarrow$ and $\Rightarrow$ instead of `->` and `=>` in type declarations; and a slanted Roman typeface instead of a constant-width typeface.

2.2.1 Objects and functions

Haskell's standard prelude defines a number of types, operators, and functions. Here are some simple constants with their types, where the notation "`::`" means "has type":

```
7        :: Int
3.1415   :: Float
True     :: Bool
'z'      :: Char
```

More complex types are also defined in the prelude; these include rational and complex numbers, arbitrary-precision integers, lists, and tuples. Complex and rational numbers are built with the `:+` and `:/` *data constructors* respectively. For example:

```
1.0 :+ 2.7      :: Complex Float
4 :/ 7          :: Rational Int
```

Operators defined in the standard prelude include the arithmetic operators `+`, `*`, `-`, and *negate* (unary negation), the relational operators `>`, `>=`, `<`, `<=`, `==`,, and `/=`, and the logical connectives `&&` and `||`. All of these operators are overloaded on appropriate types. Division (`/`) is defined for rational and floating-point numbers; integer types have integer division (*div*) and modulus (*mod*) functions. Other functions on numeric types include transcendental operations such as *sin* and *exp*, and operations specific to complex numbers, such as *magnitude* and *phase*.

Tuples contain a fixed number of fields of possibly different types; fields are separated by commas. For example:

```
(1,'a',6.666)  :: (Int, Char, Float)
```

Two useful functions on tuples are *fst*, which selects the first element of a pair, and *snd*, which selects the second.

Lists contain zero or more elements of the same type. The empty list is denoted "`[]`"; the list constructor "`:`" joins an element onto the front of a list; the syntax `[a,b,c]` is short-hand for `(a:b:c:[])`. Here are some examples of lists:

```
4:7:xs         :: [Int]
['a','b','c']  :: [Char]
"thang"        :: [Char]
```

The standard prelude contains many functions on lists. Two simple ones are *head*, which returns the first element of a list, and *tail*, which returns all elements of a list but the first. Others include: *reverse*, which reverses a list; *length*, which returns the number of elements in a list; *take*, which returns a given number of elements from the front of a list; *last*, which returns the last element of a list; *concat*, which joins a list of lists into a single list; `(++)`, which appends two lists together; and *repeat*, which repeats its argument forever: *repeat* $x \rightarrow [x, x, x, \ldots]$.

Operators are usually written in infix position, as in `x + y`. An operator can be written in prefix position by enclosing it in parentheses, as in `(+) x y`. A binary function can be written in infix position by enclosing it in back-quotes, as in ``x `div` y``.

Function application is denoted by juxtaposition, as in `sin x`, instead of the more usual $sin(x)$. Parentheses are used to disambiguate when necessary; for example, we could take the second element of a list xs with the expression

```
head (tail x)
```

Function application binds tighter than any other operator, so `sin x + 1` is the same as `(sin x) + 1`.

2.2.2 Bindings

A "binding" is the association of an identifier with a value. Consider simple pattern bindings first, in which a variable name is bound to an expression. Here are some examples:

```
pi :: Double
pi = 3.14159265358979

twopi :: Double
twopi = 2 * pi
```

A *function binding* associates an identifier with a function. For example, here is the definition of the factorial function:

```
fact :: Int -> Int
fact n = if n == 0 then 1 else n * fact (n-1)
```

The first line says that *fact* is a function, which has one integer argument and produces an integer result. The *if-then-else* construct is an *expression*, not a control-flow construct as in imperative languages—that is, it returns a value.

Function and pattern bindings can have *guards*, which select from a number of alternative expressions. For example, *fact* could also be defined like this:

```
fact :: Int -> Int
fact n | n == 0    = 1
       | otherwise = n * fact (n - 1)
```

Here, the "`|`" indicates the start of an alternative; it is followed by a predicate, then `=`, then the selected expression. The *otherwise* guard is always true. Guards are tested from top to bottom.

Bindings can contain local bindings, introduced by the *where* keyword. For example, the *butterfly* function used in section 5.2.5 has a local binding for t:

```
butterfly :: Num a => Complex a -> (Complex a, Complex a)
                                -> (Complex a, Complex a)
butterfly w (x0,x1) = (x0 + t, x0 - t)
```

```
where
t = w * x1
```

2.2.3 Patterns

A *pattern* is formed when a data constructor appears on the left-hand side of a binding. The result of evaluating the right-hand side is bound to the corresponding identifiers in the pattern. For example, the binding

```
(x:xs) = [1,2,3,4]
```

will cause the integer 1 to be bound to x, and the list $[2, 3, 4]$ to be bound to xs. Patterns can be nested arbitrarily, as in

```
((x,y) : z : zs) = ...
```

Arguments to function bindings are often patterns; in this case, they also serve to select one of possibly multiple clauses of a function definition. Patterns in this position can also be constants. To illustrate, consider the Haskell definition of *map*, which applies a given function to each element of a list:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Here, the first clause will match its list argument only if it is a null list; otherwise, the second clause will match, with the head of the list bound to x and the tail (possibly null) bound to xs.

Patterns can also be a wild-card, written "_", which matches anything. *map*, for example, would more usually be defined using wild-cards:

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map _ _      = []
```

2.2.4 Currying and higher-order functions

map is an example of a *higher-order function*, or HOF. A higher-order function takes a function argument or produces a function result. In the case of *map*, the first argument is a function, as indicated by its type, `a -> b`. Higher-order functions are one of the more powerful features of functional programming languages, as they can be used to capture *patterns* of computation. *map*, for example, captures "for-all" style iteration across a list. Other functions capture various the "do-across" styles of iteration. *foldl*, for example, applies its function argument to a list element and the result of the previous application: `foldl (+) 0 [1,2,3]` produces 6. *scanl* is similar, but produces a list containing all partial results. For example, `scanl (+) 0 [1,2,3]` produces `[0,1,3,6]`. These functions are defined as follows:

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs

scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f z []      = [z]
scanl f z (x:xs)  = z : scanl f (f z x) xs
```

One of the most useful higher-order functions is function composition, written as the infix operator "`.`" and defined by $(f\ .\ g)\ x = f\ (g\ x)$. A particularly common use is to construct a "pipeline" of functions, as in $f\ .\ g\ .\ h$.

Because Haskell is based on the lambda-calculus, higher-order functions arise naturally when we define functions of more than one argument. For example, consider a function that adds two numbers:

```
add :: Int -> Int -> Int
add x y = x + y
```

Because "`->`" associates from right to left, the type of *add* is really `Int -> (Int -> Int)`; *add* thus takes a single integer argument and produces a function from integers to integers. For example, the expression `(add 3)` is a function that adds three to its argument.

Functions of this type are called *curried*, after the mathematician Haskell Curry. In practice, we refer to a function such as *add* as a function of two arguments, and to application to less than its full number of arguments as *partial application*. For comparison, an uncurried version of *add* could be written:

```
add′ :: (Int, Int) -> Int
add′ (x,y) = x + y
```

2.2.5 *let*, *lambda*, and *case*

A *let*-expression delimits a new scope within which local bindings can be declared. To illustrate, *butterfly* can be defined using a *let*-expression:

```
butterfly w (x0,x1) = let t = w * x1
                      in  (x0 + t, x0 - t)
```

Because *let*-expressions *are* expressions, they can appear anywhere an expression can appear, whereas *where* can appear only in a binding.

A *lambda*-abstraction, or λ-abstraction, is like an anonymous function. For example, we could define *butterfly* as

```
butterfly = \w (x0,x1) -> let t = w * x1
                          in  (x0 + t, x0 - t)
```

The backslash is Haskell's syntax for a lambda-abstraction, and mimics the Greek λ; patterns following the backslash are arguments; the expression following "`->`" is the value returned by the λ-abstraction when applied to arguments.

Note that *where* cannot be used within a λ-abstraction; nor can guards. Functions defined with guards and multiple-clause definitions can be translated into a λ-abstraction with the aid of Haskell's *case* construct, which selects one of several expressions based on pattern matching. For example, *map* can be defined as the following λ-abstraction:

```
map = \f xs -> case xs of
                 []   -> []
                 y:ys -> f y : map f ys
```

2.2.6 Polymorphism and type classes

One of the key innovations of Haskell is its *type classes*, which add *ad-hoc* polymorphism to the Hindley-Milner type system [149]. A type class is a way of grouping types together with functions that operate on those types. Given in the class declaration are the types of the functions of that class, and (optionally) default definitions of some functions. For example, the standard *Eq* class, which groups types that support an equality test, is defined as

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y       = not (x == y)
```

Each type that can support the equality test "instantiates" from the *Eq* class. For example, the integer type *Int* instantiates *Eq* with:

```
instance Eq Int  where (==) = primEqInt
```

where *primEqInt* is a *primitive* function that generates a call to the underlying run-time system. Another example, the *Eq* instance for pairs:

```
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (u,v)  =  x==u && y==v
```

The type constraint `(Eq a, Eq b)` states that the elements of the pair must also be comparable for equality. Type constraints also appear in function definitions. For example, the type of the absolute function indicates that the type of the argument and result are numeric and ordered:

```
abs   :: (Num a, Ord a) => a -> a
abs x | x >= 0    = x
      | otherwise = -x
```

Classes "inherit" from each other to form a hierarchy. For example, the class of ordered types *Ord* inherits from the class of equable types *Eq*:

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    ...
```

The standard prelude implements a full hierarchy of primitive and numeric types—see [44] for details.

2.3 Evaluation mechanisms

Implementations of lazy functional languages are often based on *graph reduction*. This section briefly reviews graph reduction and other topics related to evaluation of functional programs. The aim here is to place into context the suggested implementation (in chapter 5) of stream functions as dataflow actors. A recent survey of issues in parallel functional programming is given by Hammond [54].

2.3.1 Graph reduction

Evaluation of a functional program proceeds by successively reducing expressions. The evaluator chooses a *redex* (for *red*ucible *ex*pression), reduces it, and then repeats. An expression that contains no redexes is in *normal form*. A β-reduction, written $\rightarrow_\beta$, is performed when a λ-abstraction is applied to an argument; the application is reduced by substituting the body of the λ-abstraction and replacing the formal by the actual. A δ-reduction, written $\rightarrow_\delta$, is performed when a primitive function is applied to all of its arguments; the arguments are reduced and then the appropriate operation performed.

There are two key reduction policies. In *normal-order reduction*, the evaluator chooses an outer-most redex. For example,

$$
\begin{aligned}
(\lambda x \,.\, \mathit{fst}\ x + \mathit{fst}\ x)\ (2*4, 6*7) &\rightarrow_\beta \mathit{fst}\ (2*4, 6*7) + \mathit{fst}\ (2*4, 6*7) \\
&\rightarrow_\beta 2*4 + \mathit{fst}\ (2*4, 6*7) \\
&\rightarrow_\beta 2*4 + 2*4 \\
&\rightarrow_\delta 8 + 2*4 \\
&\rightarrow_\delta 8 + 8 \\
&\rightarrow_\delta 16
\end{aligned}
$$

where $\mathit{fst}\ (x, y) = x$. In *applicative-order reduction*, the evaluator chooses an inner-most redex. For example,

$$
\begin{aligned}
(\lambda x \,.\, \mathit{fst}\ x + \mathit{fst}\ x)\ (2*4, 6*7) &\rightarrow_\delta (\lambda x \,.\, \mathit{fst}\ x + \mathit{fst}\ x)\ (8, 6*7) \\
&\rightarrow_\delta (\lambda x \,.\, \mathit{fst}\ x + \mathit{fst}\ x)\ (8, 42)
\end{aligned}
$$

$$
\begin{aligned}
&\rightarrow_\beta \quad \mathit{fst}\ (8,42) + \mathit{fst}\ (8,42) \\
&\rightarrow_\beta \quad 8 + \mathit{fst}\ (8,42) \\
&\rightarrow_\beta \quad 8 + 8 \\
&\rightarrow_\delta \quad 16
\end{aligned}
$$

Operationally, normal-order reduction applies the body of a function to its arguments before any argument is evaluated, while applicative-order reduction evaluates the arguments first. So, normal-order reduction does not evaluate an expression unless its value is needed, as can be seen from the example above: the expression $(6 * 7)$ is never evaluated. Applicative-order reduction, in contrast, evaluates all argument expressions, regardless of whether their values are needed.

The two Church-Rosser theorems are fundamental to considerations of reduction order (see [105, pp. 24–25]). Informally, the first says that any two reduction sequences will end at the same normal form provided that they both terminate. This is of great interest, as it allows the freedom to not only change the reduction order, but to perform reductions in parallel. The second theorem says that normal-order reduction will always terminate if a normal form exists. This is a powerful motivation for choosing normal-order reduction as the evaluation mechanism for a programming language.

Lazy functional languages use normal-order reduction; however, the reducer only reduces until there are no more *top-level* redexes. This is called *weak head-normal form*, or WHNF; any term in normal form is also in WHNF. Data structures are thus evaluated to the top level only—they may still contain unevaluated expressions. For example, $e = (e_1 : e_2)$ is in WHNF, regardless of what form e_1 and e_2 are in, but e is in normal form if and only if both e_1 and e_2 are in normal form. $(+(3*4))$ is in WHNF, since there are insufficient arguments to $(+)$ and it is therefore not a redex.

In the example of normal-order reduction given above, the expression $2*4$ was evaluated twice. To avoid this, functional languages use graphs to share expressions. At each reduction step, the reducer locates the left-most top-level redex, reduces it, and overwrites the redex with its result. If the redex is an application of a λ-abstraction, the reducer *instantiates* (makes a copy of) the body of the λ-abstraction with pointers to the argument expressions substituted for the formals. If the redex is an application of a primitive function, the reducer reduces any strict arguments, and then calls the run-time system to perform the appropriate operation.

2.3.2 Strictness

A function f is said to be strict iff

$$f \perp = \perp$$

That is, the result of applying f is undefined if its argument is undefined. Here, $\perp$ denotes an undefined value—we could also say that $\perp$ represents a term that has no WHNF. Operationally, we take this to mean that f always "needs" its argument; as a result, evaluating the argument of a strict function to WHNF before applying the function will not affect the result. This can be generalised to functions of several arguments; for example, g is strict in its second argument if $g\ x \perp y = \perp$.

Strictness does not imply that an argument can be evaluated *further* than WHNF without endangering termination. Let Ω denote a term without a normal form, just as $\perp$ denotes a term without WHNF. A function f is *hyper-strict* iff

$$f\ \Omega = \perp$$

That is, the result of applying f is *necessarily* undefined if its argument has no normal form. This is also called "exhaustive demand" [91], as opposed to the "simple demand" of evaluation only to WHNF. As a simple example of the difference between strictness and hyper-strictness, consider

$$f\ (x, y) = \mathit{if}\ x > 0\ \mathit{then}\ x\ \mathit{else}\ x + y$$

$$g\ (x, y) = x + y$$

f is strict but not hyper-strict; g, however, is hyper-strict. That is, an argument with no normal form, say $(7, \perp)$, does not necessarily cause the result of applying f to be undefined, while it does necessarily cause the result of applying g to be undefined.

2.3.3 Parallel graph reduction

A program graph typically contains, at any given moment, many redexes. Parallel evaluation of a graph can therefore be achieved by having many processors evaluate these redexes simultaneously. Peyton Jones [105] describes the issues in parallel graph reduction on a shared-memory machine.

There are three choices when performing parallel graph reduction: i) evaluate redexes in parallel only when it cannot alter termination properties; ii) evaluate redexes in parallel even though it may alter termination properties; or iii) allow the programmer to choose when to perform parallel reduction. The first choice—conservative parallelism—requires strictness analysis: any function that is strict in one or more arguments can have the arguments evaluated in parallel with the function body. For example, an application such as

$$e_1 + e_2$$

can have e_1 and e_2 evaluated in parallel, then the addition performed. (In this case, of course, there is no point in evaluating the function body in parallel.)

The second choice is speculative evaluation. In speculative evaluation, an expression such as

$$\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$$

can have e_1, e_2, and e_3 evaluated in parallel. One of e_2 or e_3 will be discarded as soon as the value of e_1 is known.

The third-choice is programmer-controlled parallelism. This can be in the form of annotations [61], or in the form of primitive combinators. Roe [126], for example, points out difficulties with conservative parallel graph reduction, and suggests two new combinators, *par* and *seq*:

$$\begin{aligned} \mathit{par} &:: \alpha \to \beta \to \beta \\ \mathit{par}\ x\ y &= y \end{aligned}$$

$$\mathit{seq} :: \alpha \to \beta \to \beta$$

$$seq\ x\ y \quad = \quad \begin{cases} y & \text{if } x \neq \bot \\ \bot & \text{if } x = \bot \end{cases}$$

par e_1 e_2 creates a new task to evaluate e_1 and returns e_2. If evaluation of e_1 fails to terminate, then e_2 may or may not terminate. *seq* e_1 e_2 evaluates e_1 to WHNF and then returns e_2. Roe gives a wide range of examples and algorithms illustrating how these two combinators can be used to control parallel reduction of a program graph.

2.3.4 Parallel data structures

One of the problems associated with using a pure functional language for parallel programming is the sequential nature of lists. Even if an algorithm has a high degree of intrinsic parallelism, the algorithm as expressed in a functional language may in fact obscure this parallelism, simply because the list data structure can only be accessed sequentially.

If data structures and functions that allow parallel evaluation of multiple elements of the structure can be identified, the compiler can more easily identify parallelism. In addition, the *programmer* is more likely to be better able to understand the behaviour of a parallel implementation, and thus achieve better performance.

Axford and Joy [10], for example, propose a set of list primitives that allow parallel evaluation of list functions, including the following:

– $[x]$ is the list containing one element.

– $s \mathbin{+\!\!+} t$ is the concatenation of lists s and t.

– *split* s is a pair of lists, with the split point chosen non-deterministically.

Because *split* is non-deterministic, the implementation is free to implement lists using any suitable tree structure.

Axford and Joy implement most of the standard Haskell list functions using these primitives; most have a divide-and-conquer recursive implementation. As a result, they have $O(\lg n)$ best-case and $O(n)$ worst-case parallel complexity. *map*, for example, is implemented as

$$map\ f\ [\,] \quad = \quad [\,]$$

$$\begin{aligned} map\ f\ [x] &= [f\ x] \\ map\ f\ (s +\!\!+\ t) &= map\ f\ s +\!\!+\ map\ f\ t \end{aligned}$$

Axford and Joy replace the standard *foldl* and *foldr* functions with a new function, *reduce*, which requires an associative function argument (the obligation is on the programmer to ensure that this is the case). All but three of the 17 Haskell standard prelude functions that use *foldl* or *foldr* meet this criterion.

Maassen proposes a set of three finite data structures—sequences, tables, and sets—and a range of first- and higher-order functions defined on them [92]. He gives the complexities for implementation using lists and AVL trees; the latter has logarithmic complexity for many of the functions. A range of examples illustrate the utility of his chosen functions for programming. Roe, in contrast, describes the use of bags (multi-sets) for expressing parallel computation, and shows how the Bird-Meertens formalism can be applied to parallel computation using bags [126].

2.3.5 Functional operating systems

The expression of operating system functionality in a functional language revolves around the deferred evaluation of streams of messages [71]. In the context of operating systems, a *process* is a function from a list of input messages of type α to a list of output messages of type β; Turner [140], for example, gives an example of a process:

$$\begin{aligned} process &:: [\alpha] \rightarrow [\beta] \\ process &= p\ s0\ where \\ &\quad\ p\ s\ (a:x) = out\ a\ s +\!\!+\ p\ (trans\ a\ s)\ x \end{aligned}$$

where *s0* is the initial state of the process, $trans\ :: \alpha \rightarrow \sigma \rightarrow \sigma$ is the state transition function, which produces the next state from an input message and the current state, and $out\ :: \alpha \rightarrow \sigma \rightarrow [\beta]$ produces a list of output messages for each input message. Networks of processes are formed by applying processes to each other's output messages lists.

Now, because of the corollary to the first Church-Rosser theorem, a network of processes cannot produce a result that depends on anything other than the *values* of its input messages—in particular, the time of arrival of input messages cannot affect the result.

Apparently, an operating system that needs to deal with asynchronous events cannot be written in such a language!

One approach to resolving this dilemma is to add a non-deterministic *merge* operator to the language [56], which merges two lists in order of the "arrival time" of elements. Because, however, *merge* is referentially opaque, reasoning about programs that use it becomes difficult. Jones and Sinclair [70] reduce the difficulties associated with *merge* by restricting its use to "systems programmers." They describe an operating system design and give examples of access to operating system services such as file editing and disk control. To ensure that computation proceeds in an operationally useful way, streams are constructed with *head-strict cons* —that is, elements are fully evaluated before transmission.

A second approach to non-determinism is Stoye's "sorting office" [135]. In this scheme, every output message contains an address of a destination process; the sorting office receives all of these messages and routes each one to the addressed process. Non-determinism is thus contained in one place only—the sorting office—simplifying reasoning about the system and eliminating any need for awkward extensions to the language. New processes can be created by sending a message to the distinguished *process-creation* process. Turner [140] develops Stoye's model, adding "wrappers" to messages to allow type checking, and using synchronous (unbuffered) communication instead of asynchronous communication. Similarly to Jones and Sinclair, wrapper functions are hyper-strict. Apart from eliminating the need for global garbage collection, this allows the operating system to be executed on a loosely-coupled network of processors.

Wallace and Runciman use constructor classes [69] to express type-safe communication between processes [151]. In their scheme, the message type itself serves as the message address. Wallace has implemented an operating system for embedded systems in Gofer; unlike the other systems reviewed here, the stream constructor is not hyper-strict in its head [150]. Nonetheless, Wallace suggests that head-hyper-strictness would be an important ingredient in making evaluation of functional programs predictable enough to meet real-time scheduling constraints.

2.3.6 Functional process networks

Kelly proposed that list-manipulating functions be treated as processes to be mapped onto processors of a distributed-memory MIMD machine [78]. In his Caliban language, programs are written in a pure functional language using standard list functions such as *map* and *filter*. An additional language construct, *moreover*, contains declarations that specify which lists represent communications channels between processes. For example, the expression

$$
\begin{array}{l}
f\ (g\ xs) \\
\quad \mathit{where} \\
\qquad f = \mathit{map}\ \mathit{abs} \\
\qquad g = \mathit{scanl}\ (+)\ 0 \\
\quad \mathit{moreover} \\
\qquad \mathit{arc}\ \Box f\ \Box g
\end{array}
$$

is a two-processor pipeline. The $\Box$ operator converts a function into a process: this indicates that the argument and result lists of the function so indicated are communications channels rather than conventional lazy lists. The *arc* function specifies a connection between two processes.

The power of the host language can be used to define more abstract network-forming operations. For example, the *pipeline* combinator builds a pipeline of processes. In the following, *chain* has been defined to insert *arc* between processes in a list (*fs*):

$$
\begin{array}{rcl}
\mathit{pipeline} & :: & (\alpha \to \alpha) \to \alpha \to \alpha \\
\mathit{pipeline}\ \mathit{fs}\ xs & = & ys \\
& & \mathit{where} \\
& & \quad ys = (\mathit{foldl}\ (\cdot)\ \mathit{id}\ \mathit{fs})\ xs \\
& & \mathit{moreover} \\
& & \quad \mathit{chain}\ \mathit{arc}\ (\mathit{map}\ (\Box)\ \mathit{fs}) \\
& & \quad \wedge\ \mathit{arc}\ \Box(\mathit{last}\ \mathit{fs})\ xs \\
& & \quad \wedge\ \mathit{arc}\ \Box(\mathit{hd}\ \mathit{fs})\ ys
\end{array}
$$

Using *pipeline*, the above network can be written

$$pipeline\ [map\ abs, scanl\ (+)\ 0]\ xs \tag{2.1}$$

The current implementation of Caliban requires that the process network be static. At compile-time, all *moreover* annotations are expanded to extract a single top-level annotation which describes a static network [38].

Kelly suggests that strictness analysis or programmer annotations be used to minimise or eliminate the overheads associated with sending non-normal form expressions over channels. The current implementation supports only head-hyper-strict streams—in other words, only normal-form elements can be sent over channels.

2.3.7 Skeletons

Section 2.3.4 showed that parallelism can be captured in terms of data structures. In contrast, Cole's algorithmic skeletons [35] capture parallel computation in terms of a relatively abstract algorithmic description: a "skeleton" is a particular pattern of parallel computation. Although Cole used an imperative framework, he points out that higher-order functions are an ideal mechanism for expressing skeletons.

The idea of selecting a parallel program structure from a catalogue of such structures is appealing. Darlington *et al* [40] adapt Cole's idea to a functional language framework. They point out that a skeleton has an implementation-independent meaning, as given by its functional definitions, and a behaviour tailored to the target parallel machine—parallelism arises from the behavioural aspect. They propose the following skeletons as higher-order functions:

pipe Given a list of functions, produce a pipeline in which each function is allocated to a different processor.

farm Given a function from a datum and an environment, produce a "farm" of processes which applies this function to each element of a data set and a given environment.

dc (divide-and-conquer) Given functions to a) split a data set, b) test and solve the trivial solution, and c) combine solutions, produce a tree-structured parallel program that implements a divide-and-conquer algorithm.

ramp (reduce-and-map-over-pairs) Given two functions that a) represent the interaction between any two data items, and b) combine results of interactions between data items, produce a parallel program that calculates the interactions between all items in a data set, and combines the interactions to produce a single solution.

dmpa (dynamic-message-passing-architecture) Given a set of processes, each of which accepts a set of messages and produces a set of messages for other processes, generate the parallel program that implements the set of processes.

Darlington *et al* give examples in which skeletons are transformed into other skeletons. The programmer can therefore: i) choose a skeleton that most easily represents the problem to be solved, and then ii) transform the program, based on abstract performance measures, into a skeleton that is more efficient on a given machine architecture.

Bratvold [22] proposes a different set of skeletons: *map*, *filter*, *fold*, and composition. Functionally these functions are the same as Haskell's *map*, *filter*, *foldl*, and (.). The first three are implemented as processor "farms," and composition as pipeline connection of farms. Three additional skeletons, *filtermap*, *mapfilter*, and *foldmap*, are compositions of the first three.

In contrast to Darlington *et al*, Bratvold argues that skeletons should be identified by the compiler, not be a directive from the programmer that parallel execution take place. His compiler uses profiling and performance prediction to decide when to execute a skeleton in parallel; in addition, this approach allows the compiler can detect parallelism in recursive functions [30]. Measurements on compiled programs show execution to be within 20% of the estimates.

2.4 Real-time signal processing

The term "real-time" refers to systems in which "the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced" [133]. Examples of real-time systems include command and control systems, process control systems, flight control systems, and so on.

This thesis focuses on a particular class of real-time system: digital signal processing systems, which analyse, produce, and transform discrete-time signals. Whereas conventional real-time systems respond to or process individual events, which may occur at arbitrary times, signal processing systems process *streams* of data representing discrete-time signals, the elements of which usually occur at known rates. Signal processing systems also tend to have more stringent timing requirements: timing parameters are expressed in tens of micro-seconds, rather than in milliseconds.

This section provides some of the background needed to appreciate some decisions made in later chapters. The section is largely based on my own experience with and observations of real-time programs.

2.4.1 Discrete-time signals

A *discrete-time signal* is a function that is defined only at a particular set of values of time. In the common case called *uniform sampling*, a discrete-time signal $x(n)$ is related to a continuous analog signal $x_a(t)$ by

$$x(n) = x_a(nT), \qquad -\infty < n < \infty \tag{2.2}$$

where T is the *sampling period*, and $f_s = 1/T$ is the *sampling frequency*.

Signals can be internal or external—in the latter case, they constitute an interface between the signal processing system and an external environment. Many signal processing systems are "off-line"—that is, the external signals are not processed and produced simultaneously with their occurrence in the external environment. In real-time signal processing, however, they are.

Although discrete-time signals are defined over all time, computer implementations find it more convenient to consider only non-negative time indices, where time zero is the nominal time at which the real-time program begins operating. Thus:

$$x(n) = x_a(nT), \qquad n \geq 0 \tag{2.3}$$

From here on, quantification of time indexes over positive n is implicit.

Let t_x denote the *clock* of signal x—that is, the sequence of times at which x is defined. Each element of a clock is called a "tick." The clock of a uniformly-sampled signal x with period T is

$$t_x = \{nT \mid n \geq 0\}$$

A non-uniformly-sampled signal is not characterised quite so easily. Let $\widetilde{x}$ be a non-uniformly-sampled signal with clock $\widetilde{t_x}$. Then:

$$\widetilde{x}(n) = x_a(\widetilde{t_x}(n)), \qquad n \geq 0 \tag{2.4}$$

In general, a discrete-time signal does not represent samples of an analog signal. The sequence of values representing changes of the state of my refrigerator door—open, closed, or almost-closed—is an example. A sequence of messages between processes in a multi-tasking operating system is another. I will use the term *uniformly-clocked* to mean a signal with clock $\{nT \mid n \geq 0\}$ for constant T, and *non-uniformly-clocked* for any other signal. Non-uniformly clocked signals will be treated in chapter 6.

As notational short-hand, the logical and arithmetic operators extend point-wise to clocks—that is, for some operator $\oplus$,

$$t_x \oplus t_y \quad \Rightarrow \quad t_x(n) \oplus t_y(n)$$

Two clocks t_x and t_y are thus *equal* if $t_x = t_y$; t_x is *earlier* than t_y if $t_x < t_y$; and t_x is *later* than t_y if $t_x > t_y$.

2.4.2 Streams and channels

A signal is represented in a computer by a register or data structure that updates in time to contain zero, one, or more values of the signal. This register or data structure I will call a *channel*. It could be implemented in software or hardware; in either case, a program (or hardware) can *write to* and *read from* the channel.

Define a *stream* as the sequence of values that passes through a channel. A uniformly-clocked signal x is implemented by the *synchronous* stream $\mathbf{x}$. Streams are thus equivalent to signals:

$$\mathbf{x}(n) = x(n)$$

$$\widetilde{\mathbf{y}}(n) = \widetilde{y}(n)$$

Unlike a signal, a stream has not one but *two* clocks. The *write clock* of $\mathbf{x}$, w_x, is the sequence of times at which stream values become defined—in other words, the times at which they are written to the stream's channel. The *read clock* of $\mathbf{x}$, r_x, is the sequence of times at which stream values are consumed—that is, read from the channel. Read and write clocks are in general non-uniformly spaced: they represent *actual* times at which certain events (reading from or writing to a stream) occur, not idealised times at which signal values occur. So, while signal clocks are understood by the program, read and write clocks are solely for our own use in analysing and describing programs.

Assume that the action of writing a value to or reading a value from a channel consumes an infinitesimally small amount of time. The read clock of a stream x must therefore be later than its write clock:

$$r_x > w_x \tag{2.5}$$

The relation between a signal's clock and its stream's clocks depend on whether the signal is internal or external. Consider an input signal x. Because its stream, $\mathbf{x}$, is produced by the external environment, the signal clock is equal to the stream's *write* clock:

$$w_x = t_x \tag{2.6}$$

For an output signal y, the stream's *read* clock is equal to the signal clock:

$$r_y = t_y \tag{2.7}$$

Correct real-time operation hinges on the program responding appropriately to external signals. From equations 2.5 to 2.7, we have, for input stream x and output stream y,

$$r_x > t_x \tag{2.8}$$

and:

$$w_y < t_y \tag{2.9}$$

In other words, the program cannot attempt to read from an input stream too early, and must write to an output stream sufficiently early. The non-zero time difference between

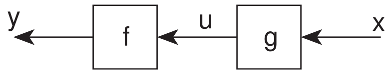


Figure 2.4. A simple block diagram

the read or write clock and the signal clock can be attributed to the time taken by the program to transfer a value between a signal and a stream (for example, the interrupt response time and the time to copy a value from a real-time input-output port into the channel).

2.4.3 Functions and systems

A signal processing system with one input signal x and one output signal y is a function from x to y:

$$y = f(x)$$

A system is composed of functions, each with its own input and output signals. For example, the system consisting of two series-connected functions f and g is:

$$y = f(g(x))$$

Giving a name to the internal stream between g and f, this can also be written as the system of equations

$$u = g(x)$$

$$y = f(u)$$

When implemented in a block diagram system, f and g become blocks, and x, u, and y become streams. This simple system is shown in figure 2.4.

A very important function in signal processing systems is the "delay" operator:

$$y = z^{-k}x \quad \Rightarrow \quad y(n) = \begin{cases} 0, & n < k \\ x(n-k) & n \geq k \end{cases}$$

The delay operator is not usually implemented as a process, but by inserting k initial zero values into a FIFO buffer contained in the relevant channel.

2.4.4 Digital signal processors

Digital signal processors are microprocessors tailored specifically to performing signal processing computations in real time. Most of the differences to other microprocessors–both CISC controllers and more powerful RISC CPUs—centre around two key points: i) the need to support certain types of arithmetic very efficiently; and ii) the need for deterministic execution times. In this section, I give an overview of the characteristics of these devices. I focus only on the modern, floating-point devices; for more detailed information on specific devices, see [115, 119, 34, 5, 6, 98, 138, 137].

Figure 2.5 illustrates a simplified architecture of a typical floating-point DSP core. The device contains six functional units—an ALU, a multiplier, two address units, and two load-store units (not shown here)—all of which can operate simultaneously, a bank of floating-point data registers, and a bank of address registers. A typical instruction thus allows one or two arithmetic operations, two indirect loads or stores, and two address register updates. This combination of operations is ideally suited for vector operations, in which elements are stored in memory at fixed address increments; it is a major factor in the very high performance of DSPs on signal processing algorithms, as compared to CISC or RISC processors fabricated in comparable technology.

Operations for each functional unit are encoded into a single instruction word—32 or 48 bits. DSPs thus resemble horizontally micro-coded devices more than super-scalar (multiple-instruction issue) RISCs. Most instructions execute in one cycle; to maintain this rate, the instruction stream is pipelined, usually to three or four levels. Because of the limited instruction width, some restrictions are placed on possible combinations of operations; for example, many devices allow the multiplier and ALU to be used in parallel if the ALU is performing an addition or subtraction. More complex instructions also place limitations on source and destination registers.

The TMS320C30 and TMS320C40 [138, 137] are essentially register-memory machines: operands can be located in memory or in registers. A typical instruction for these devices is the parallel multiply-and-add instruction:

```
mpyf r0,r1,r0  ||  addf *ar0++,*ar1++(ir0)%,r2
```

In addition to the multiply-and-add operations, two operands are loaded from memory

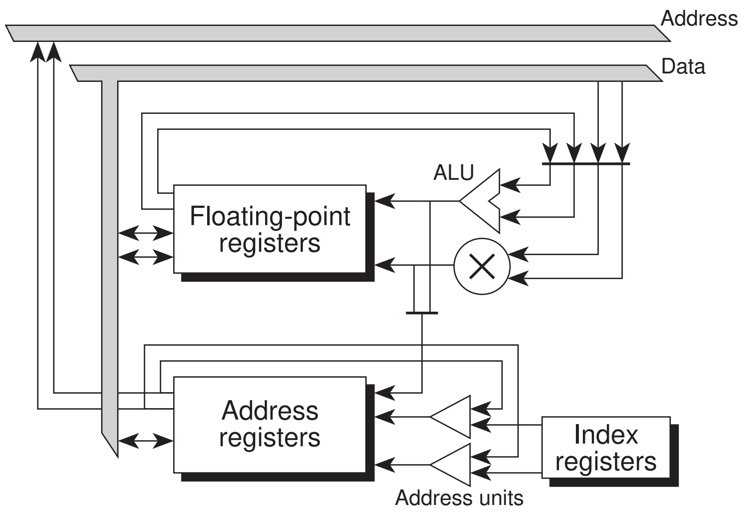


Figure 2.5. A simplified DSP chip architecture

and two address registers incremented. Provided that memory operands are located in zero-wait-state memory,[1] these devices can sustain instructions like this at a rate of one per cycle. In figure 2.5, I have explicitly shown a data path around the data registers and the ALU and multiplier, and around the address registers and address generators. These paths, together with parallel data and address busses, provide the data transfer bandwidth necessary to sustain these instruction rates.

The multiply-add instruction is the core of the FIR (finite-impulse response) filter, a key DSP benchmark. Some devices also have a parallel multiply-add-subtract instruction, which substantially speeds up execution of the FFT (Fast Fourier Transform), another key benchmark.

In contrast to the TMS320C30 and TMS320C40, the ADSP-21020 and DSP96002 [5, 98] are load-store machines: operands are explicitly loaded into data registers prior to operating on them. The programmer writes separate fields of the instruction to control the ALU and multiplier, and the load/store and address units. For example, a typical parallel multiply-add instruction in M96002 code is:

```
fmpy d8,d6,d2  fadd.s d3,d0  x:(r0),d4.s  d2.s,y:(r5)+n5
```

The first two fields are multiplier and ALU operations; the second two are data move and

[1] The placement of memory operands is a little complicated: the device has internal memory which can be accessed twice per instruction cycle, and external memory which can be accessed (at most) only once. The TMS320C40 has two external busses, which can both be active simultaneously.

address update operations.

Apart from the architectural layout and instruction set, DSP devices also feature hardware to perform zero-overhead looping. The mechanism varies with the device, but it essentially allows the device to loop over a block of instructions without flushing the pipeline at the end of the block. Zero-overhead looping is essential, for example, to execute one multiply-add instruction per cycle, as needed by the FIR filter.

The address generators are quite sophisticated, to minimise the need for separate address manipulation instructions. Firstly, they can increment by the contents of an index register, allowing the device to step through memory in arbitrarily sized increments. Secondly, they perform modulo addressing, in which an address register is automatically "wrapped" back to a base address when incremented or decremented past a limit address. (This is descriptive only—real DSPs implement modulo addressing a little differently.) Modulo addressing is essential for efficient implementation of circular buffers and FIR filters. Thirdly, they support reverse-carry addressing, which provides the "bit-reversed" addressing needed by the FFT (see section 5.2.5) at no execution cost.

Early DSPs were fully deterministic in instruction execution time: each instruction took exactly one cycle. The CPUs of modern DSPs are still deterministic, although calculating execution time requires knowledge of the location of memory operands and, for some devices, of the interaction between successive instructions in the instruction pipeline. The presence of prioritised DMA controllers and instruction caches, however, makes exact prediction of execution times impossible. Nonetheless, simpler interrupt-handling architectures, non-hierarchical memory architectures, and predictable instruction execution times still make DSP execution behaviour much more predictable than modern RISC devices, an essential ingredient in the design of time-critical systems.

2.5 Summary

The material presented in this chapter provides the background for following, more detailed, chapters. The pipeline parallelism model underlies the model of computation described in Chapters 3, 5, and 6. Chapters 4, 5, and 6 rely on functional programming concepts and notation; as well as the notation, I presented different evaluation mechanisms for functional programs, to place into context the dataflow style of evaluation of these programs.

Finally, because most of the work in following chapters is influenced by the goal of real-time execution on embedded DSP device, I provided an overview of real-time programming concepts and DSP devices.

Chapter 3

Dataflow Process Networks

Dataflow [1, 7] is a model of computation in which data items, called *tokens*, flow between computing agents, or *actors*. A program is represented by a dataflow graph (DFG), in which vertices correspond to actors and edges to a "flow" of data from one actor to another. The term "dataflow" is used in a number of different contexts; here we are interested in *pipeline dataflow*: actors are long-lived, producing and consuming many items during their lifetimes. This model is identical to Kahn process networks, for which Kahn gave a denotational semantics [75].

A pipeline dataflow network corresponds directly to a signal flow block diagram [74]: for example, a box labelled "+" on a block diagram sums two signals in a point-wise manner. In dataflow terms, the "+" actor is fired repeatedly; each time, it reads a token from both input channels and writes their sum to its output channel. Figure 3.1 illustrates a simple dataflow network. The *summer* actor is the one just described. The *delay* actor is a unit delay: each element of v appears on w in the following time slot. The *scale* actor multiplies each element of its input stream by a constant—in this case, the value $-a$.

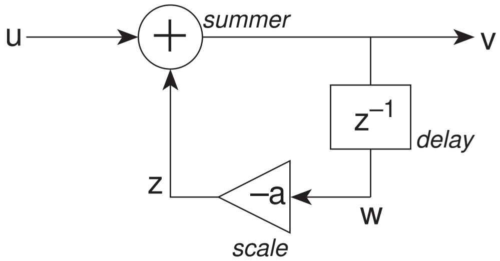


Figure 3.1. A simple dataflow network

Lee coined the term *dataflow process* to describe Kahn processes implemented as a sequence of firings of a dataflow actor [87]. Each firing is governed by a *rule* that states conditions on input values needed to fire the actor. Provided the sequence of actor firings is deterministic, then the actor forms a deterministic Kahn process. We are assured by Kahn that a network of such processes will also be deterministic, and able to produce output data before all input data is received.

This view of a dataflow actor as a Kahn process is relatively new; in this chapter, I develop a formalism for describing dataflow actors, and give the semantics of actors and their corresponding dataflow processes. These actors are *strict*—that is, read all required input tokens before producing any output—as is common practice. The notation extends Lee's firing rules to include state update and token output as well as token matching and reading.

When considering the relationship between dataflow actors and non-strict functional programming languages, non-strict actors are needed to more accurately model the way in which tokens are produced and consumed. I therefore introduce a new form of actor, called *phased* form. Execution of these actors proceed in a series of phases, several of which may be required in place of a single firing of a strict actor. Phased form also permits limited amounts of non-determinism and gives some new insights into the behaviour of dataflow actors.

3.1 Related work

The term "dataflow" is used in a number of different contexts. It is often associated with dataflow machines, in which the dataflow model is the basis of the execution mechanism of a highly-parallel machine [7]. Actors are fine-grained machine operations such as addition, while edges of the graph represent transfer of tokens through the machine. The machine executes an actor when all required tokens are present on its inputs. The languages used to program such machines [1] are often called dataflow languages.

Dataflow analyses (DFAs) are used for code optimisation in compilers [2]. DFAs provide the compiler with information such as variable lifetimes and usage, which enables it to eliminate code that will never be executed, to re-use registers when their contents are no longer needed, and to re-order instructions in order to generate improved machine code sequences. Again, the nodes of the dataflow graph are fine-grained operations corresponding to single machine instructions; the edges of the DFG represent dependencies between instructions.

In pipeline dataflow, actors perform relatively complex tasks—that is, operations that may take dozens, hundreds, or thousands of machine instructions to complete. The edges of the DFG correspond to communications channels between the actors. This is the model in which I am interested in this thesis.

3.1.1 Computation graphs

In 1966, Karp and Miller described a graph-theoretic model of parallel computation that is essentially pipeline dataflow [77]. The computation is represented by a finite directed graph: each node n_j represents a computational operation O_j; each arc d_p represents a queue of data from one node to another. Associated with each d_p from n_i to n_j are four non-negative integers: A_p, the number of tokens initially in the queue; U_p, the number of tokens written to d_p whenever O_i executes; W_p, the number of tokens read from d_p whenever O_j executes; and $T_p \geq W_p$, the number of tokens needed in d_p before O_j can execute. Each arc can be connected to only one producer and one consumer node.

Figure 3.2 illustrates a very simple computation graph, to compute $n!$. Each arc is annotated with its (A_p, U_p, W_p, T_p). n_1 takes a token from its input, and increments and outputs its internal value (with zero initial value). The self-loop around n_1 contains n initial tokens; these are the tokens that "drive" the computation. n_2 multiplies its two inputs together and outputs the result on two arcs. The self-loop around n_2 has one initial token, value assumed to be zero. n_3 writes the final value received to the location that holds the result of the entire computation.

Note that n_1 maintains the value of its counter internally, while n_2 explicitly passes its previous output value back to itself through an arc. Thus, the model does not care whether or not nodes maintain an internal state or perform explicit feedback.

Execution of a computation graph G containing l nodes is represented by a sequence of sets $\xi = S_1, S_2, \ldots, S_N, \ldots$, such that each set $S_N \subseteq \{1, 2, \ldots, l\}$. Let $x(j, N)$ denote the number of sets $S_m, 1 \leq m \leq N$ such that $j \in S_m$, and let $x(j, 0) = 0$. ξ is a *proper execution* of G iff

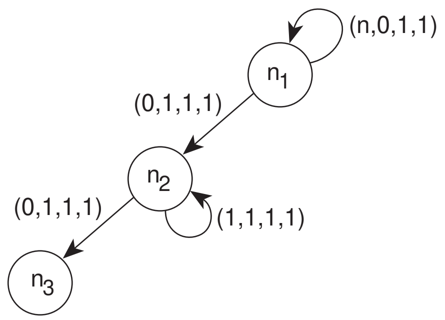


Figure 3.2. A computation graph

1. If $j \in S_{N+1}$ and G has an arc d_p from n_i to n_j, then

$$A_p + U_p x(i, N) - W_p x(j, N) \geq T_p$$

2. If, for all nodes n_i and arcs d_p from n_i to n_j, $A_p + U_p x(i, N) - W_p x(j, N) \geq T_p$, then $\exists R > N : j \in S_R$.

Condition (1) states that, to execute an operation, the initial number of tokens added to the total number of tokens produced to its input channel, must exceed the total number consumed plus the threshold. Condition (ii) states that if an operation *can* be executed then it will be within a finite number of steps. The sequence is thus like a partially-ordered schedule: each S_N contains all nodes eligible for execution at that time. Note that a given graph can have more than one proper execution.

Karp and Miller prove a number of properties of this model. Firstly, the number of performances of an operation is the same in all proper executions of G. Also, if the initial tokens on every arc are the same, then the value of any given token is the same for all proper executions. G is thus *determinate*.

Karp and Miller also develop a number of theories concerning termination of the graph: which nodes terminate, and for those that do, how many times they appear in all possible executions.

3.1.2 Synchronous dataflow (SDF)

Synchronous dataflow (SDF) is a slightly less general model than computation graphs: the input threshold is equal to the number of consumed tokens. It is, however, designed

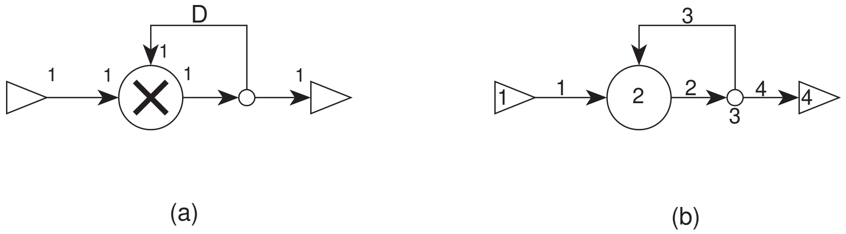


Figure 3.3. A synchronous dataflow graph

specifically for processing infinite streams. SDF scheduling was developed by Lee and Messerschmitt [89, 90], and has since formed the backbone of efforts to implement dataflow networks efficiently for both real-time and non-real-time execution.

Figure 3.3a shows a simple SDF graph: the numbers next to the node inputs and outputs are equivalent to W_p and U_p; a D annotation indicates that an arc contains one zero initial value; the small circle is a "fork" node. This graph computes the running product of its input stream.

Compared to the computation graph, the SDF graph is "driven" by an infinite input stream produced by a node with no input arcs, instead of by a finite number of initial tokens placed on an arc.

A schedule can be calculated for an SDF graph: this is an ordered sequence of node names that, repeated forever, computes the output streams from the input streams. The incidence matrix Γ contains the token consumption and production figures of the graph: $\Gamma(i, j)$ is the number of tokens produced by node j on arc i each time it is fired; if j consumes data, $\Gamma(i, j)$ is negative. Figure 3.3b shows the example graph decorated with node and arc numbers; we then have

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

The vector $\mathbf{b}(n)$ is the number of tokens on each arc at time n. Thus,

$$\mathbf{b}(0) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The vector $\mathbf{v}(n)$ denotes the node scheduled at time n. That is, $\mathbf{v}(n)(i) = 1$ and

$\mathbf{v}(n)(j) = 0$ if node i is scheduled and $i \neq j$. Since executing a node changes the number of tokens on arcs, we have

$$\mathbf{b}(n+1) \;=\; \mathbf{b}(n) + \Gamma\mathbf{v}(n) \tag{3.1}$$

Assuming that sample rates are consistent and the network does not deadlock, the vector $\mathbf{q}$ is the number of times each node is executed in the schedule: if node i is executed k times, then $\mathbf{q}(i) = k$. To find $\mathbf{q}$, assign any node a repetition count of 1. Follow any arc d_p from that node, and assign the connected node the repetition count U_p/W_p, storing this quantity as an exact rational number. When all nodes have been reached, find the least common multiple of the denominators of the repetition counts and multiply to find $\mathbf{q}$.

The example network is the special case called *homogeneous* dataflow, in which all token counts are unity. Calculating $\mathbf{q}$ is trivial:

$$\mathbf{q} \;=\; \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Given q, a schedule is found by simulating execution of the graph at compile-time with equation 3.1. At each step, if the numbers of tokens in an arc is at least equal to the number of tokens consumed, append the consumer node to the schedule. Stop scheduling a node i when it has appeared in the schedule $\mathbf{q}(i)$ times. For the simple example, the only possible schedule is $\langle 1, 2, 3, 4\rangle$.

The sizes of internal FIFO buffers can be determined in the same way. A code generation system can take advantage of this fact to produce extremely efficient code. The efficiency improvement over dynamic scheduling is so great that considerable effort has been expended in dataflow models that are less restricted than SDF, but which are sufficiently constrained that SDF code can be applied to parts of the graph [26, 43].

SDF code generators operate by "firing" code generation actors at compile-time; each time an actor is fired, it emits C or assembler code. For assembler code generation, macros and symbolic names make the programmer's task easier and allow some optimisations. Powell *et al* [111], for example, describe assembler code generation in SPW, and give some examples of the assembler source code. A typical instruction looks like this:

```
ADDL  tA,out   tA,tX      Y:(table.reg)+,tY
```

Here, tA, out, tX, and tY stand for registers, and *table* for a region of memory. Powell *et al* use register allocation heuristics to assign registers, with spill instructions inserted where necessary. For example, the above instruction may become

```
ADDL  A,B       A,X0       Y:(R0)+,Y0
```

As just described, SDF code generation will tend to generate large amounts of code for general multi-rate graphs, since the code for each actor is duplicated each time the actor appears in the schedule. Because embedded DSP processors often have severely limited memory space, more sophisticated scheduling techniques have been developed that produce loops in the generated output code, thus minimising the code memory required [16].

3.1.3 Kahn's process networks

Kahn described the semantics of a language for parallel programming based on process networks [75]. In Kahn's language, a program consists of a network of "computing stations" (that is, processes) connected by FIFO-buffered channels. Each process repeatedly reads data from one or more input channels and writes data on one or more output channels. Figure 3.4 shows the simple pipeline given on page 18 written in Kahn's language.

Kahn processes can only communicate through *channels*. Channels are obliged to transmit data within a finite time. A process can read input channels using blocking reads only; if data is not yet available, the process must suspend until it becomes so. In other words, a process cannot *test* for the presence of data on an input channel; nor can it wait for data on one *or* another of its input channels (unlike other concurrent systems such as CSP [60]).

The semantics of a Kahn process network are expressed in terms of the *histories* of data transmitted on each channel: a Kahn process is a function or set of functions from histories to histories. Given some domain $\mathbf{D}$, let $\mathbf{D}^*$ be the set of all finite sequences containing elements in $\mathbf{D}$; let $\mathbf{D}^\infty$ be the set of infinite sequences containing elements in $\mathbf{D}$; and let $\mathbf{D}^\omega = \mathbf{D}^* \cup \mathbf{D}^\infty$. Sequences are related by the partial ordering $\sqsubseteq$, where $X \sqsubseteq Y$ if and only if X is a prefix of or equal to Y. The minimal element is the empty sequence $[\,]$. A chain of sequences $X = X_1 \sqsubseteq X_2 \sqsubseteq \ldots \sqsubseteq X_n \sqsubseteq \ldots$ has a least upper bound $lub\ X$.

```
begin
    integer channel X, Y, Z;

    process absolutes( integer in I, integer out O );
        begin integer T;
            repeat begin
                T := wait(I);
                send abs(T) on O;
            end;
        end;

    process running( integer in I, integer out O );
        begin integer T,V;
            V := 0;
            repeat begin
                T := wait(I);
                V := V + T;
                send V on O;
            end;
        end;

    comment : begin main program

    running(X,Y) par absolutes(Y,Z);
end;
```

Figure 3.4. A Kahn process network

$\mathbf{D}^\omega$ is thus a complete partial order (cpo).

A Kahn process is built from *continuous* functions—a function f is continuous iff

$$f(lub\ X) = lub\ f(X)$$

Continuity ensures that a process is not able to produce output only after it has received an infinite amount of output.

A continuous process is also monotonic—that is, for any two sequences a and b,

$$a \sqsubseteq b \;\Rightarrow\; f(a) \sqsubseteq f(b)$$

Monotonicity ensures that future output can depend only on future input, thus allowing processes to produce output before receiving all their input.

A process with arity (m, n) has m input channels in $\mathbf{D}_1^\omega, \mathbf{D}_2^\omega, \ldots, \mathbf{D}_m^\omega$ and n output channels in $\mathbf{E}_1^\omega, \mathbf{E}_2^\omega, \ldots, \mathbf{E}_n^\omega$. The process is specified by n continuous functions from $\mathbf{D}_1^\omega \times \ldots \times \mathbf{D}_m^\omega$ into $\mathbf{E}_1^\omega$, $\mathbf{E}_2^\omega$, and so on. A process network Σ_p is a set of fix-point equations over cpos; such a system has a unique minimal solution. More importantly, this minimal solution is a continuous function of the input histories and the processes, and thus is also monotonic. This means that: i) a network of Kahn processes also exhibits the desirable property of producing output before receiving all its input; and ii) networks can be connected in the same manner as single processes, thus allowing hierarchical construction of networks.

Kahn and MacQueen [76] suggest that a Kahn process network be implemented in a *demand-driven* style, using co-routines. When a process attempts to read a token from an empty input channel, it suspends, and the process that writes to that channel is activated. When the producer process writes data to the channel, it is suspended and the consumer process resumed. The producer process itself may be suspended in the course of producing its output data. Ultimately, these "demands" for data propagate back to processes that generate input data (from, say, a file) or read it from some input channel (such as, say, the console). There is a single "driver" process, usually the process responsible for printing program results.

The dual to demand-driven execution is data-driven execution: rather than executing when a successor process requires data, a process executes when it has enough input data. A process suspends whenever it attempts to read from a channel with no data; it is resumed by the operating system some time after data is written to the channel. The implementation of stream-processing functions in SISAL [46] uses this technique [50].

3.1.4 Dataflow processes

Lee defines a *dataflow process* to be the sequence of firings of a dataflow actor [87]. A dataflow process is thus a special kind of Kahn process—one in which execution is broken into a series of "firings." Because the process is a sequence of actor firings, a complete network can be executed by firing actors in an appropriate order. Actors manage their own internal state, so there is no need to provide each process with the illusion that it has exclusive access to the CPU: context-switching is eliminated. This is a very important point for real-time signal processing: context-switching presents a very high overhead to a real-time digital signal processor, as the time needed to save and restore all the registers of a modern DSP is quite high—see [118] for further discussion of this issue.

Demand-driven execution on a single processor can be implemented simply and elegantly. As each actor executes, an attempt to read from an empty channel causes the source actor to fire immediately. Since "firing" an actor involves nothing more than executing a procedure, the propagation of demands for data proceeds recursively down the dataflow graph. The recursion unwinds as data is propagated back through the graph. Real-time execution on multiple processors, however, presents some awkward problems. Firstly, data must be demanded "ahead of time," to guarantee that data is available ahead of the output signal clocks. Secondly, a demand for data across a processor boundary causes the demanding processor to idle while the data is being produced. Although this effect could be overcome by processing a different part of the graph while waiting, it would re-introduce the overhead of multi-tasking.

The difficulties with demand-driven execution of dataflow networks led myself and Matthias Meyer to abandon attempts to incorporate it into the initial design of the SPOOK (Signal Processing Object-Oriented Kernel) parallel DSP kernel [118, 96]. Nonetheless, we still believe that it is important and that an efficient hybrid scheduling mechanism is possible.

With data-driven scheduling, an actor is fired when it has sufficient tokens on its input channels. An external scheduler tests if enough tokens are available, and fires the actor if they are. A naive implementation of data-driven scheduling simply cycles through all actors, firing any which have sufficient input tokens. More sophisticated schedulers attempt to minimise unnecessary testing by tracing the graph in topological order [94], or by "pushing" tokens through the graph [96].

Because of the "eager" nature of data-driven execution, processes may produce *more* data than will ever be needed. Some form of throttling is needed: bounding channel buffers is one solution; pre-emptive scheduling is another. Neither solution is elegant, suggesting that perhaps a hybrid demand- and data-driven solution is really needed. Ashcroft [9] proposes a machine architecture for Lucid based on mixing both methods within the same network. Edges of the network graph are coloured according to whether they are data-driven or demand-driven. Request tokens are propagated *backwards* down the demand-driven arcs—these tokens are called *questons*, while normal data tokens are called *datons*.

Pingali and Arvind [107] take another approach: they give a transformation by which a dataflow graph with demand-driven semantics is transformed into an equivalent graph with data-driven semantics. In essence, demands are made explicit by adding arcs and actors to carry questons. The propagation of questons, and the resulting propagation of datons back up the graph, is performed using data-driven execution. Skillicorn [130] proposes that strictness analysis be applied to Lucid programs to allow questons to be sent up the graph in fewer hops. If it is known that an actor will always demand data from certain inputs, then that actor can be bypassed by the questons, since a demand for its output will simply produce demands for its input anyway. When the up-stream actors to which the questons are sent produce datons, the datons are propagated in a data-driven fashion through the bypassed (by the questons) actor.

Jagannathan surveys the current state of dataflow computing models, including a comparison of demand- and data-driven execution [65]. Dataflow architectures have also been proposed for implementation of functional programming languages [141]. Field and Harrison point out that demand-driven evaluation of functional programs gives normal-order semantics, while data-driven evaluation gives applicative-order semantics [47, chapter 14].

3.1.5 Firing rules

Lee has recently formalised the notion of "firing" an actor [87]. An actor with arity (m, n) has a set of N firing rules

$$\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_N\}$$

where each $\mathcal{F}_i$ is an m-tuple of patterns, one for each input:

$$\mathcal{F}_i = (P_{i,1}, \ldots, P_{i,m})$$

Each pattern is a sequence of tokens, each being either a manifest constant or the "wildcard" symbol '$*$'. Lee defines a modified version of the prefixing predicate, which we can express as:

$$[p_1, \ldots, p_q] \ \sqsubseteq^* \ (x_1 : \ldots : x_r : \perp) \ \Leftrightarrow \ r \geq q \ \wedge \ \forall i \in \{1..q\} : p_i = \text{`}*\text{'} \ \vee \ p_i = x_i$$

where the sequence $(x_1 : \ldots : x_r : \perp)$ is the sequence of "available" tokens on the input channel. There are r available tokens; $\perp$ represents the undefined part of the sequence—in other words, tokens which have not yet been computed.

A firing rule $\mathcal{F}_i$ of an actor with m input channels is *enabled* if and only if

$$P_{i,j} \ \sqsubseteq^* \ A_j, \ \forall j \in \{1..m\}$$

where A_j is the sequence of available tokens on the j'th input channel.

Lee's firing rules capture the essential aspect of scheduling a dataflow actor under a *strict*, *data-driven* semantics. By "strict," I mean that all input tokens must be present before the actor begins to execute. Let us consider some examples. The *summer* actor sums corresponding elements of two input channels; it thus requires a token on each before it can fire:

$$\mathcal{F}_1 = ([*], [*])$$

The *select* actor consumes one token from a boolean control input: if true, it consumes a token from its first data input and passes it to the output; if not, it consumes a token from its other data input and passes it to the output. It has the firing rules

$$\mathcal{F}_1 = ([\mathit{True}], [*], [\,])$$
$$\mathcal{F}_2 = ([\mathit{False}], [\,], [*])$$

A *non-deterministic* actor does not form a Kahn process. The archetypical non-deterministic actor is *merge*, which passes an available token on either input to the output. It has the firing rules

$$\mathcal{F}_1 = ([*], [\,])$$
$$\mathcal{F}_2 = ([\,], [*])$$

Recall that one of Kahn's requirements is that a process perform only blocking reads. The non-deterministic merge does not satisfy this criterion, since it must test its inputs to see whether they have data to pass to the output. Non-deterministic merge can be made deterministic by making time part of the data in the stream—see chapter 6.

Lee gives an algorithm that can determine whether a finite set of firing rules can be tested by a process that performs only non-blocking reads [87]. A set of firing rules that satisfies this condition Lee calls *sequential*. Briefly, his algorithm works as follows: Choose an input j such that all $P_{i,j}$ contain at least one element. If the head elements of the patterns do not unambiguously divide the firing rules into subsets, then fail. If they do, remove the head elements, and repeat the procedure on each subset, with the modified patterns. Fail at any time if any rule in a subset is not either empty or at least one element long. If the algorithm terminates without failing, then the rules are sequential. In effect, the algorithm mimics the run-time behaviour of the actor.

3.2 Standard-form dataflow actors

Lee's firing rules capture the input conditions necessary to fire an actor, although without saying precisely what a "firing" is, or by what mechanism the pattern is matched against available input tokens. Apart from matching and reading input tokens, an actor firing also i) updates the actor's internal state (if any), and ii) calculates and produces output tokens. In this section, I give a syntax for describing actors, followed by a precise semantics of dataflow actors and processes.

3.2.1 Syntax

The term "dataflow actor" is used to mean two things: the description of an actor, and an actual instance of an actor within a network. I will call these an *actor schema* and an *actor instance* where necessary to distinguish between them.

A sugared syntax for actor schemata is shown in figure 3.5. Figure 3.5a is the general form for stateful actors. The actor is called *name*, and has zero or more *parameters* $(v_1, \ldots, v_a)$. Parameters are arguments to the actor other than streams, and must be reducible to a constant at compile-time. The *init* clause specifies the initial value of the

$$
\begin{array}{lllll}
(a) & \textsf{actor } name(v_1, \ldots, v_a) \equiv & & & \\
& \quad \textsf{init} \quad s_0 & & & \\
& \quad \textsf{rule} \quad p & : \textsf{update}(s) & = & e_u \\
& & \quad\ \textsf{output}(s) & = & e_o \\
& & \vdots & & \\
& \quad \textsf{rule} \quad p & : \textsf{update}(s) & = & e_u \\
& & \quad\ \textsf{output}(s) & = & e_o \\
& & & & \\
(b) & \textsf{actor } name(v_1, \ldots, v_a) \equiv & & & \\
& \quad \textsf{rule} \quad p & : e_o & & \\
& & \vdots & & \\
& \quad \textsf{rule} \quad p & : e_o & &
\end{array}
$$

Figure 3.5. Sugared syntax of a standard-form actor: a) stateful actor b) stateless actor

actor's internal state, s_0; this is also assumed to be known at compile-time. It is followed by one or more *rules*, each containing an input pattern p, an update action $\textsf{update}(s) = e_u$, and an output action $\textsf{output}(s) = e_o$.

Each input pattern corresponds to one of Lee's firing rules. A pattern is thus a sequence of *token patterns*, where a token pattern is an identifier, a simple constant, or a structured type containing element patterns. The only difference to Lee's firing rules is that identifiers are used instead of the symbol "*".

Some patterns are *irrefutable*—that is, they cannot fail to match [72, pp. 72–74]. I use a very loose interpretation of this definition, allowing not only identifiers and tuples of identifiers, but other structured patterns if it is known that the matched value is such that the pattern will always match. (For an example of an irrefutable pattern, see the *group* actor of figure 3.8, which uses a vector as an irrefutable pattern.)

The update action is a function from the current value of the actor's internal state, s, to the value of the state after completion of a firing. s cannot be used in rule selection, and must therefore be an irrefutable pattern. The update expression e_u can contain free variables v_1 to v_a and any variable in p. The output action is a function from the actor's internal state to the sequences or sequences of output tokens produced by this firing. The output expression e_o is a *manifest sequence*, or a *let*-expression with a manifest sequence as result. A manifest sequence is an expression in which the length of the sequence can be determined purely syntactically—for example, *let* $x = y^2$ *in* $[x, x, x]$ is a valid output expression, but *let* $x = y^2$ *in copy* 3 x is not. This ensures that the number of tokens produced can be determined by examining the output action, without knowledge of data values.

Patterns and output expressions are sequences if the actor has one input or one output channel respectively. If it has more than one input or output channel, the pattern or output expression is a tuple of sequences.

Some examples will clarify the above description. Consider the *delay* operator, expressed as a standard-form actor schema with one rule:[1]

$$\begin{array}{l}
\textsf{actor}\ \mathit{delay}\,(i)\ \equiv \\
\quad \textsf{init}\quad i \\
\quad \textsf{rule}\quad [x]\ :\ \begin{array}{lcl} \textsf{update}\,(s) & = & x \\ \textsf{output}\,(s) & = & [s] \end{array}
\end{array}$$

The actor has one parameter, i; this value is the initial value of the actor's state. On each firing, the actor produces the current state, and sets the next state equal to the value of a new input token. Output values are thus delayed by one time slot. In signal processing, delays usually have an initial state of zero; the delay actor of figure 3.1 is thus instantiated as *delay*(0).

A slightly more complex actor implements the *running* process of figure 3.4; this actor has no parameters:

$$\begin{array}{l}
\textsf{actor}\ \mathit{running}\ \equiv \\
\quad \textsf{init}\quad 0 \\
\quad \textsf{rule}\quad [x]\ :\ \begin{array}{lcl} \textsf{update}\,(s) & = & x + s \\ \textsf{output}\,(s) & = & [x + s] \end{array}
\end{array}$$

For stateless actors, I use the simplified syntax of figure 3.5b, which omits keywords and expressions related to the internal state. Written in full (as in figure 3.5a), a stateless actor has the *init* value ( ), the update action $\textsf{update}\,(s) = s$, and the output action $\textsf{output}\,(s) = e_o$.

[1] Operationally, this actor is unlikely to be useful, as delays are not strict. A more accurate version of delay is given in section 3.3.1.

Here are some examples of stateless actors. The *summer* actor has one rule:

$$\begin{array}{l} \textsf{actor}\ \mathit{summer} \ \equiv \\ \quad \textsf{rule}\quad ([x],[y]) \ : \ [x+y] \end{array}$$

The *scale* actor also has one rule:

$$\begin{array}{l} \textsf{actor}\ \mathit{scale}\,(v) \ \equiv \\ \quad \textsf{rule}\quad [x] \ : \ [v \times x] \end{array}$$

The *select* actor (section 3.1.4) has two rules:

$$\begin{array}{l} \textsf{actor}\ \mathit{select} \ \equiv \\ \quad \textsf{rule}\quad ([\mathit{True}],[x],[\,]) \ : \ [x] \\ \quad \textsf{rule}\quad ([\mathit{False}],[\,],[y]) \ : \ [y] \end{array}$$

Finally, the non-deterministic *merge* actor also has two rules:

$$\begin{array}{l} \textsf{actor}\ \mathit{merge} \ \equiv \\ \quad \textsf{rule}\quad ([x],[\,]) \ : \ [x] \\ \quad \textsf{rule}\quad ([\,],[y]) \ : \ [y] \end{array}$$

3.2.2 Desugaring

The actor schemata of the previous section are "desugared"—that is, translated into a simpler but more easily-manipulated form—by the translation of figure 3.6. This figure introduces the notation used for syntactic translation functions: double square brackets ($[\![\ ]\!]$) enclose a syntactic element. In desugared form, an actor schema is a triple (a, s_0, R), where a is a (possibly empty) tuple of actor parameter names, s_0 is the actor's initial state, and R is a set of rules. Each rule is itself translated into a triple $(P, \mathit{upd}, \mathit{out})$, where P is a pattern, upd an update action, and out an output action.

$$
\mathit{desugar} \quad \left[\!\!\left[\begin{array}{l} \textsf{actor}\ \mathit{name}(v_1, \ldots, v_a)\ \equiv \\ \qquad \textsf{init}\ s_0 \\ \qquad \mathit{rule}_1; \ldots; \mathit{rule}_k \end{array} \right]\!\!\right]
$$

$$
\equiv\ ((v_1, \ldots, v_a),\ s_0,\ \{\mathit{desugarR}\ [\![\mathit{rule}_1]\!], \ldots, \mathit{desugarR}\ [\![\mathit{rule}_k]\!]\})
$$

$$
\mathit{desugarR} \quad \left[\!\!\left[\begin{array}{llcl} \textsf{rule}\ \ p \ \rightarrow & \textsf{update}(s) & = & e_u \\ & \textsf{output}(s) & = & e_o \end{array} \right]\!\!\right] \equiv\ (p,\ \lambda s \,.\, e_u,\ \lambda s \,.\, e_o)
$$

Figure 3.6. Desugaring a standard-form actor

For example, the *delay* actor, after applying this transformation, becomes the triple

$$
(i,\ i,\ \{([x],\ \lambda s \,.\, x,\ \lambda s \,.\, [s])\})
$$

A stateless actor uses the same translation as above; as a result, the translated actor has $s_0 = (\,)$ and each *upd* equal to the identity function $\lambda s \,.\, s$. For example, the *select* actor becomes

$$
\begin{array}{ll} ((\,),\ (\,),\ \{ & (([\mathit{True}], [x], [\,]),\ \lambda s \,.\, s,\ \lambda s \,.\, [x]), \\ & (([\mathit{False}], [\,], [y]),\ \lambda s \,.\, s,\ \lambda s \,.\, [y])\ \}) \end{array}
$$

3.2.3 Semantics

Lee defines a dataflow process to be a sequence of firings of a dataflow actor. Provided the actor's firing rules are sequential, this process is also a Kahn process—that is, a mapping from input histories and initial state to output histories. In this section, I give a semantics of actors and state precisely the meaning of a dataflow process.

We will need some primitive types to help describe actors more precisely: let the type of the actor's internal state be σ; the type of an input pattern be *Patt*; the type of one or a tuple of patterns be *Pattern*; the type of the (possibly empty) tuple of schema parameters be *Parameters*.

As we saw in section 3.1.3, a Kahn process is a function from sequences to sequences. In this chapter, we consider only infinite sequences. Let the (::=) operator define a type synonym. The inputs and outputs of the actor are infinite sequences or tuples of infinite sequences:

$$\begin{aligned} \mathit{Input} &::= \mathbf{D}_1^{\infty} \times \ldots \times \mathbf{D}_m^{\infty} \\ \mathit{Output} &::= \mathbf{E}_1^{\infty} \times \ldots \times \mathbf{E}_n^{\infty} \end{aligned}$$

When the actor fires, however, it consumes and produces only finite segments of its input and output sequences:

$$\begin{aligned} \mathit{InputSegment} &::= \mathbf{D}_1^{*} \times \ldots \times \mathbf{D}_m^{*} \\ \mathit{OutputSegment} &::= \mathbf{E}_1^{*} \times \ldots \times \mathbf{E}_n^{*} \end{aligned}$$

Update and output actions accept a state value and produce a new state and an output segment respectively:

$$\begin{aligned} \mathit{UpdateAction} &::= \sigma \rightarrow \sigma \\ \mathit{OutputAction} &::= \sigma \rightarrow \mathit{OutputSegment} \end{aligned}$$

A *Rule* is a triple of a pattern and two actions; *Rules* is a set of rules; a *Schema* is a triple of parameters, initial state, and a rule set:

$$\begin{aligned} \mathit{Rule} &::= \mathit{Pattern} \times \mathit{UpdateAction} \times \mathit{OutputAction} \\ \mathit{Rules} &::= \{\mathit{Rule}\} \\ \mathit{Schema} &::= \mathit{Parameters} \times \sigma \times \mathit{Rules} \end{aligned}$$

Execution of an actor proceeds in two distinct phases: i) instantiation of the actor with its parameters; and ii) execution of the actor on its stream arguments. In [87], Lee stresses the difference between parameter arguments and stream arguments in Ptolemy: parameters are evaluated during an initialisation phase; streams are evaluated during the main execution phase. As a result, code generation can take place with the parameters known, but with the stream data unknown. Thus, the separation between parameters and streams—and between compile-time and run-time values—is both clear and compulsory.

An actor is instantiated by supplying its schema with parameter values. To represent

$$
\begin{array}{l}
\mathit{instantiate}\ [\![\mathit{Schema}]\!] :: \mathit{Parameters} \rightarrow \mathit{Actor} \\
\mathit{instantiate}\ [\![(v, s_0, \{R_1,\ \ldots,\ R_k\})]\!]\ e \\
\qquad\qquad\qquad \equiv \quad (s_0[e/v], \{\mathit{inst}\ [\![R_1]\!],\ \ldots,\ \mathit{inst}\ [\![R_k]\!]\}) \\
\qquad\qquad\qquad\quad\ \ \mathit{where} \\
\qquad\qquad\qquad\quad\ \ \mathit{inst}\ [\![(P, \mathit{upd}, \mathit{out})]\!] \equiv (P[e/v],\ \mathit{upd}[e/v],\ \mathit{out}[e/v])
\end{array}
$$

Figure 3.7. Instantiating an actor

instantiation of a schema *name* with parameter values $(e_1,\ \ldots,\ e_a)$, we will write

$$A := \mathit{name}(e_1,\ \ldots,\ e_a)$$

or, if the schema has no parameters,

$$A := \mathit{name}$$

where A is the (unique) identifier of the actor instance. Instantiating a schema creates a new actor instance:

$$\mathit{Actor} \quad ::= \quad \sigma \times \mathit{Rules}$$

To instantiate an actor, each parameter formal occurring free in the initial state and the firing rules is substituted by the corresponding actual. Figure 3.7 gives the instantiation function. The notation $E[e/v]$ means that each occurrence of identifier v occurring free in E is replaced with the expression e. This notation extends point-wise to tuples of substitution variables—that is, if $v = (v_1,\ \ldots,\ v_a)$ and $e = (e_1,\ \ldots,\ e_a)$, then

$$E[e/v] \ \Leftrightarrow\ E[e_1/v_1]\ldots[e_a/v_a]$$

An actor instance is connected to other actors in a network, and can then proceed to the second part of its execution, as a dataflow process. The process is an infinite series of firings, each of which proceeds in three stages: input, state update, and output. The first can in turn be divided into two parts: matching rule patterns against input sequences to select a rule, and reading the input segments from the input sequences.

Pattern-matching and rule selection is performed by the semantic function *match*, which accepts a rule and input sequences, and produces that same rule if it matches the sequences. Thus,

$$
\begin{aligned}
&\mathit{match}\ [\![\mathit{Rule}]\!] \quad :: \quad \mathit{Input} \to \mathit{Rule} \\
&\mathit{match}\ [\![R@((P_1,\ldots,P_m),\_,\_)]\!]\ (X_1,\ldots,X_m) \\
&\qquad\qquad = \quad R, \quad \text{if } \mathit{match}'\ [\![P_i]\!]\ X_i \text{ for all } i,\ 1 \le i \le m
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{match}'\ [\![\mathit{Patt}]\!] \quad :: \quad \mathbf{D}^{\infty} \to \mathit{Bool} \\
&\mathit{match}'\ [\![[p_1,\ldots,p_q]]\!]\ (x_1 : \ldots : x_q : xs) \\
&\qquad\qquad = \quad \mathit{true}, \quad \text{if } \mathit{tmatch}\ p_i\ x_i \text{ for all } i,\ 1 \le i \le q
\end{aligned}
$$

The notation $R@((P_1,\ldots,P_m),\_,\_)$ is from Haskell: R is the name of the rule, which is a triple containing patterns $(P_1,\ldots,P_m)$ and two other elements of insignificant value. The *tmatch* function performs pattern-matching on tokens, as in functional languages [72].

Implicit in the definition of *match* is the concept of failure: if the input patterns do not match the input sequences, then *match* returns $\perp$ (pronounced "bottom"), denoting an undefined value. Note also that because *tmatch* compares each token against a pattern, all required tokens must be available before a rule can be selected and execution proceed. The semantics thus captures the strictness of actors.

Given a selected rule, the *read* function splits the input sequences into the segments to be read by this firing, and the remainder of the sequences:

$$
\begin{aligned}
\mathit{read}\ [\![\mathit{Rule}]\!] \quad &:: \quad \mathit{Input} \to \mathit{InputSeg} \times \mathit{Input} \\
\mathit{read}\ [\![((P_1,\ldots,P_m),\_,\_)]\!]\ (X_1,\ldots,X_m) \quad &= \quad \mathit{let} \quad
\begin{array}{lcl}
(is_1, i_1) & = & \mathit{read}'\ [\![P_1]\!]\ X_1 \\
\ldots & = & \ldots \\
(is_m, i_m) & = & \mathit{read}'\ [\![P_m]\!]\ X_m
\end{array} \\
&\qquad\ \ \mathit{in} \quad ((is_1,\ldots,is_m),(i_1,\ldots,i_m))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{read}'\ [\![\mathit{Patt}]\!] \quad &:: \quad \mathbf{D}^{\infty} \to \mathbf{D}^{*} \times \mathbf{D}^{\infty} \\
\mathit{read}'\ [\![[p_1,\ldots,p_q]]\!]\ (x_1 : \ldots : x_q : xs) \quad &= \quad ([x_1,\ldots,x_q], xs)
\end{aligned}
$$

The *input* function combines matching and reading: it matches a rule set against input sequences and returns the selected rule, the input segments, and the remaining input sequences:

$$\begin{array}{rcl}
\mathit{input}\ [\![\mathit{Rules}]\!] & :: & \mathit{Input} \rightarrow \mathit{Rule} \times \mathit{InputSeg} \times \mathit{Input} \\
\mathit{input}\ [\![\{R_1, \ldots, R_k\}]\!]\ i & = & \begin{array}[t]{lll} \mathit{let} & R & = \mathit{match}\ [\![R_1]\!]\ i \\ & & \ \square\ \ldots \\ & & \ \vdots \\ & & \ \square\ \mathit{match}\ [\![R_k]\!]\ i \\ & (is, i') & = \mathit{read}\ [\![R]\!]\ i \\ \mathit{in} & (R, is, i') & \end{array}
\end{array}$$

input uses the $\square$ operator to select a valid rule from several possible matches; $\square$ is "bottom-avoiding":

$$\begin{array}{rcrcl}
\perp & \square & \perp & = & \perp \\
\perp & \square & r & = & p \\
r & \square & \perp & = & p \\
r_1 & \square & r_2 & = & \perp
\end{array}$$

The last line means that pattern-matching will fail if more than one match succeeds. This behaviour is chosen to emphasise that the semantics works only for deterministic actors.

The *update* and *output* semantic functions update the actor's state and produce output according to a given rule. Their inputs are a segment of the input sequences and the current state. These functions are somewhat simpler to derive than *input*: all that is required is to bind the free variables named by the input patterns by supplying them as arguments:

$$\begin{array}{rcl}
\mathit{update}\ [\![\mathit{Rule}]\!] & :: & \mathit{InputSeg} \rightarrow \sigma \rightarrow \sigma \\
\mathit{update}\ [\![(P, \lambda s \,.\, e_u, \lambda s \,.\, e_o)]\!] & = & \lambda P \,.\, \lambda s \,.\, e_u
\end{array}$$

$$\begin{array}{rcl}
\mathit{output}\ [\![\mathit{Rule}]\!] & :: & \mathit{InputSeg} \rightarrow \sigma \rightarrow \mathit{OutputSeg} \\
\mathit{output}\ [\![(P, \lambda s \,.\, e_u, \lambda s \,.\, e_o)]\!] & = & \lambda P \,.\, \lambda s \,.\, e_o
\end{array}$$

The *fire* function combines input, state update, and output, into a single actor firing. At each firing, it accepts the unread input history and the current state, and produces the remaining unread input history, the next state, and a segment of the output sequences:

$$\begin{aligned}
\mathit{fire}\ [\![\mathit{Rules}]\!] &:: \mathit{Input} \times \sigma \to \mathit{Input} \times \sigma \times \mathit{OutputSeg} \\
\mathit{fire}\ [\![\mathit{Rs}]\!]\ (i, s) &= (i', s', o') \\
&\quad \mathit{where} \\
&\quad \begin{aligned} (R, \mathit{is}, i') &= \mathit{input}\ [\![\mathit{Rs}]\!]\ i \\ s' &= \mathit{update}\ [\![R]\!]\ \mathit{is}\ s \\ o' &= \mathit{output}\ [\![R]\!]\ \mathit{is}\ s \end{aligned}
\end{aligned}$$

The function *process*, given a set of rules, a starting state, and a tuple of input sequences, endlessly chooses and then fires a rule:

$$\begin{aligned}
\mathit{process}\ [\![\mathit{Rules}]\!] &:: \mathit{Input} \to \sigma \to \mathit{Output} \\
\mathit{process}\ [\![\mathit{Rs}]\!]\ i\ s &= o \mathbin{++} \mathit{process}\ i'\ s' \\
&\quad \mathit{where} \\
&\quad (i', s', o) = \mathit{fire}\ [\![\mathit{Rs}]\!]\ i\ s
\end{aligned}$$

The output segments are prepended to the remainder of the output sequences by the `(++)` operator:

$$(x_1 : \ldots : x_q) \mathbin{++} ys = (x_1 : \ldots : x_q : ys)$$

and we assume that `(++)` extends to tuples of sequences.

Finally, we can give the precise meaning of a dataflow process: the dataflow process corresponding to the dataflow actor A is the function $\mathit{io}\ [\![A]\!]$, where *io* supplies the actor's initial state and complete input sequences to *process*:

$$\begin{aligned}
\mathit{io}\ [\![\mathit{Actor}]\!] &:: \mathit{Input} \to \mathit{Output} \\
\mathit{io}\ [\![(s_0, \mathit{Rs})]\!]\ i &= \mathit{process}\ [\![\mathit{Rs}]\!]\ i\ s_0
\end{aligned}$$

3.2.4 Consumption and production vectors

When fired, an actor consumes some number of tokens from its input streams, and produces some number to its output streams. These numbers are manifest in the actor's firing rules—that is, in the input patterns and the output actions.

Let the $\#$ operator return the length of a sequence, and the $\boldsymbol{\#}$ operator return the lengths of sequences in a tuple, as a vector. $\boldsymbol{\#}$ is defined informally by

$$\begin{aligned} \boldsymbol{\#} s &= \langle \# s \rangle \\ \boldsymbol{\#}(s_1, \ldots, s_k) &= \langle \# s_1, \ldots, \# s_k \rangle \end{aligned}$$

The consumption vector $C[\![R]\!]$ and production vector $P[\![R]\!]$ contain the number of tokens consumed and produced by a rule $R = (P, \lambda s \,.\, e_u, \lambda s \,.\, e_o)$:

$$\begin{aligned} C[\![R]\!] &= \boldsymbol{\#} P \\ P[\![R]\!] &= \boldsymbol{\#} e_o \end{aligned}$$

Recall that an SDF actor consumes and produces known and constant numbers of tokens on each firing (section 3.1.2). This property can be formally stated in terms of consumption and production vectors. Define the $\odot$ operator to return a value only if both arguments are equal:

$$\begin{array}{ccccll} x & \odot & x & = & x & \\ x & \odot & y & = & \bot, & x \neq y \\ \bot & \odot & y & = & \bot & \\ x & \odot & \bot & = & \bot & \\ \bot & \odot & \bot & = & \bot & \end{array}$$

Let $\odot$ extend point-wise to vectors:

$$\langle x_1, \ldots, x_k \rangle \odot \langle y_1, \ldots, y_k \rangle = \langle x_1 \odot y_1, \ldots, x_k \odot y_k \rangle$$

The consumption and production vectors of an actor with rules $\{R_1, \ldots, R_k\}$ are defined by

$$\begin{aligned} \mathcal{C}[\![A]\!] &= C[\![R_1]\!] \odot \ldots \odot C[\![R_k]\!] \\ \mathcal{P}[\![A]\!] &= P[\![R_1]\!] \odot \ldots \odot P[\![R_k]\!] \end{aligned}$$

Thus, if, for each input, the input patterns of all clauses are the same length, the consumption vector contains an integer in the appropriate position; otherwise, it contains $\perp$ (pronounced "bottom"), indicating that the number cannot be determined without additional information. If the lengths of all output sequences for a given output are the same length, then the production vector similarly contains an integer.

For example, *summer* has vectors

$$\begin{aligned}\mathcal{C}[\![summer]\!] &= \langle 1, 1\rangle \\ \mathcal{P}[\![summer]\!] &= \langle 1\rangle\end{aligned}$$

The *select* actor has vectors

$$\begin{aligned}\mathcal{C}[\![select]\!] &= \langle 1, \perp, \perp\rangle \\ \mathcal{P}[\![select]\!] &= \langle 1\rangle\end{aligned}$$

The definition of SDF actors, and the sub-class of homogeneous dataflow actors, can now be stated in terms of consumption and production vectors:

Definition 1 (Synchronous dataflow (SDF)) *An actor A with arity $m \times n$ is* synchronous dataflow (SDF) *iff*

$$\mathcal{C}[\![A]\!](i) \neq \perp, \qquad \forall i,\ 1 \leq i \leq m$$

$$\mathcal{P}[\![A]\!](i) \neq \perp, \qquad \forall i,\ 1 \leq i \leq n$$

Definition 2 (Homogeneous dataflow (HDF)) *An actor A with with arity $m \times n$ is* homogeneous dataflow (HDF) *iff*

$$\mathcal{C}[\![A]\!](i) = 1, \qquad \forall i, \quad 1 \leq i \leq m$$

$$\mathcal{P}[\![A]\!](i) = 1, \qquad \forall i, \quad 1 \leq i \leq n$$

SDF actors are usually expressed with a single rule; call an actor with a single rule and irrefutable token patterns a *simple SDF* actor. Such an actor does not need to perform pattern-matching and rule selection—since it only has one rule that, given enough input

$$
\begin{array}{lllll}
\textsf{actor } \mathit{delay}\,(i) \ \equiv & & & & \\
\quad \textsf{init} \ \ i & & & & \\
\quad \textsf{rule} \ \ [x] & : & \textsf{update}\,(s) & = & x \\
 & & \textsf{output}\,(s) & = & [s] \\
 & & & & \\
\textsf{actor } \mathit{group}\,(k) \ \equiv & & & & \\
\quad \textsf{rule} \ \ [x_1, \ldots, x_k] & : & [\langle x_1, \ldots, x_k \rangle] & & \\
 & & & & \\
\textsf{actor } \mathit{concat}\,(k) \ \equiv & & & & \\
\quad \textsf{rule} \ \ [\langle x_1, \ldots, x_k \rangle] & : & [x_1, \ldots, x_k] & & \\
 & & & & \\
\textsf{actor } \mathit{zip}\,(k) \ \equiv & & & & \\
\quad \textsf{rule} \ \ ([x_1], \ \ldots, \ [x_k]) & : & [(x_1, \ \ldots, \ x_k)] & & \\
 & & & & \\
\textsf{actor } \mathit{unzip}\,(k) \ \equiv & & & & \\
\quad \textsf{rule} \ \ [(x_1, \ \ldots, \ x_k)] & : & ([x_1], \ \ldots, \ [x_k]) & & \\
 & & & & \\
\textsf{actor } \mathit{map}\,(f) \ \equiv & & & & \\
\quad \textsf{rule} \ \ [x] & : & [f(x)] & &
\end{array}
$$

Figure 3.8. The canonical SDF actors

tokens, cannot fail to match. It therefore needs only to count input tokens, not read their values.

3.2.5 Canonical SDF actors

Any SDF actor can be implemented as a network containing delays and instances of just five actor schemata. In chapter 5, I will give functions equivalent to these schemata, and give examples of their use; here, I will give the schemata texts and argue that any SDF actor can be translated into a network of these five actors. I will also further discuss the role of actor parameters.

Figure 3.8 lists the five actors, together with *delay*. Each has one parameter. For completeness, *delay* is shown as an actor although it will almost always be implemented as initial values in a FIFO buffer. Note that *delay* is the only actor here that maintains a "state."

group breaks a stream up into a stream of vectors. Unlike *delay*, its parameter must be known in order to produce a formal (that is, executable) actor text. (The notation $x_1, \ldots, x_k$ is not formal.) This is really a limitation of the fact that an actor text must contain a manifest sequence in its output action. As we will see in chapter 5, *group* can, with a more powerful notation, be defined without requiring that k be known at compile-time. *concat* is the inverse of *group*: it concatenates a stream of vectors into a stream. Again, its parameter must be known in order to produce a formal actor text.

zip and *unzip* combine multiple streams into a stream of tuples, and vice versa. They also require that their parameter be known at compile-time; in this case, however, it is not possible to remove this restriction with a more powerful notation, since the parameter determines the number of input or output channels of the actor. In chapter 5, this limitation will force us to use restricted versions of these functions.

Finally, the *map* actor takes a function as its parameter, which it applies to each element of its input channel. If f is known, an efficient implementation of $map(f)$ can be generated; if not, the system must support dynamic creation of functions since it will not have knowledge of f until run-time. An actor of this kind mimics higher-order functions in functional languages, and could therefore be called a *higher-order actor*.

I will now argue informally that any SDF actor can be implemented by delays and these five schemata. As pointed out by Lee [87], a stateful actor can be represented as a stateless actor together with a unit-delay feedback loop carrying the "state" value. The remaining five schemata can implement any stateless SDF actor, as follows. Let the consumption and production vectors of the actor be $\langle c_1, \ldots, c_m \rangle$ and $\langle p_1, \ldots, p_n \rangle$. For any input i such that $c_i > 1$, insert a $group(c_i)$ actor in the connected arc and change the pattern for that input from $[x_1, \ldots, x_{c_i}]$ to $[\langle x_1, \ldots, x_{c_i} \rangle]$. For any output j such that $p_i > 1$, insert a $concat(p_i)$ actor in the connected arc and change the expression sequence for that output from $[y_1, \ldots, y_{p_i}]$ to $[\langle y_1, \ldots, y_{p_i} \rangle]$. The actor is now HDF. If the actor has more than one input—that is, $m > 1$—use a $zip(m)$ actor to gather all input streams into a stream of tuples, and change the input pattern from $([x_1], \ldots, [x_m])$ to $[(x_1, \ldots, x_m)]$. If the actor has more than one output—that is, $n > 1$—change the output expression from $([y_1], \ldots, [y_n])$ to $[(y_1, \ldots, y_n)]$ and use an $unzip(n)$ actor to change the stream of tuples back into a tuple of streams. The actor now has one input channel and one output channel and thus has the form $(s_0, \{([p], \lambda s \,.\, s, \lambda s \,.\, [e])\})$; this actor is implemented by $map\,(\lambda p \,.\, e)$.

3.3 Phased-form dataflow actors

In this section, I introduce the *phased* form of dataflow actor. Execution of a phased-form actor proceeds in a series of "phases"; phased form thus expresses a finer grain of computation than standard form. One of the motivating factors behind its development was a desire to better express non-strict and demand-driven execution. By breaking actor firings up into phases, actors need perform only the minimum amount of input and computation necessary to produce a token on a given output.

Phased form also permits non-deterministic behaviour; the semantics outlined in this section provides a means of characterising non-deterministic behaviour. In particular, an actor can non-deterministically choose to consume tokens from different inputs, yet still form a deterministic process.

To clarify this point, consider the sum-of-squares actor, *sumsqrs*; it is similar to *summer*, but squares each of its two input tokens before summing them:

$$\begin{array}{l} \textsf{actor}\ \mathit{sumsqrs}\ \equiv \\ \qquad \textsf{rule}\quad ([x],[y]) \ : \ [x^2+y^2] \end{array}$$

This definition says nothing about the order in which inputs are read—or outputs produced, although there is no choice in this particular case. As we saw in section 3.2.3, a standard-form actor reads all input tokens before performing any computation or output. But this need not be so: *sumsqrs* could read a token from its first input and square it before attempting to read a token from its second input. It could even wait for a token on *either* input, square it, and then wait for a token on the other input. Although this actor does not satisfy Kahn's blocking-read criterion, it nonetheless implements a Kahn process.

The phased form of an actor is thus a way of precisely expressing the ordering of input, computation, and output, with non-determinism allowed in a limited and manageable way. As we will see, this is useful when considering demand-driven execution and potential deadlock situations. There are a number of other motivations for this work on phased-form actors:

- Phased form can express some actors that cannot be expressed in standard form, such as the *iota* actor described in section 3.3.4.

- Taken as an operational definition, computation can proceed before all input is received. For a single actor, this is unlikely to be important; for a network, however, it may well be, since computation can proceed in one part of the network even though other parts are blocked waiting for data.

- Phased form has the potential to provide a more consistent semantics of hierarchical actors, although I have not yet been able to develop an algorithm that computes the phased form of an actor network. (See section 3.3.6.)

Phased-form actors are a generalisation of Buck's multi-phase integer dataflow actors [27] and the cyclo-static dataflow of Engels *et al* [43]; a comparison is given in section 3.3.4.

3.3.1 Syntax

Phased-form actors are an extension of standard-form actors. There are two key differences:

- At each firing, only a subset of the rule set is considered for firing. This subset is called the *eligible rule set*.

- If more than one rule from the eligible set is satisfied, the actor *non-deterministically* chooses and fires one of them.

The full phased-form syntax is shown in figure 3.9a. The actor has $k \geq 1$ rules. In order to identify the eligible rules, each rule is explicitly numbered. The initial eligible rule set, ϕ_0 where $\phi_0 \subseteq \{1, \ldots, k\}$, is given by the *start* clause. When a rule is fired, the actor uses the *select* action to update the eligible rule set for the next firing.

There are some restrictions not made explicit in the syntax. Firstly, token patterns must be irrefutable (that is, they cannot fail to match). As for standard-form, the state pattern in the actions must also be irrefutable. Secondly, the right-hand side of the select action is either a set of integers, or an expression of the form

$$\textit{if } e \textit{ then } \{\textit{Int}\} \textit{ else } \ldots \textit{ else } \{\textit{Int}\}$$

$$
\begin{array}{llllll}
(a) & \textsf{actor}\ name(v_1,\ldots,v_a)\ \equiv & & & & \\
 & \quad\textsf{init} & s_0 & & & \\
 & \quad\textsf{start} & \phi_0 & & & \\
 & \quad\textsf{rule}\ 1 & p & : & \textsf{update}(s) & = e_u \\
 & & & & \textsf{output}(s) & = e_o \\
 & & & & \textsf{select}(s) & = e_p \\
 & & & \vdots & & \\
 & \quad\textsf{rule}\ k & p & : & \textsf{update}(s) & = e_u \\
 & & & & \textsf{output}(s) & = e_o \\
 & & & & \textsf{select}(s) & = e_p \\
 & & & & & \\
(b) & \textsf{actor}\ name(v_1,\ldots,v_a)\ \equiv & & & & \\
 & \quad\textsf{start} & s_0 & & & \\
 & \quad\textsf{rule}\ 1 & p & : & e_o & \to e_p \\
 & & & \vdots & & \\
 & \quad\textsf{rule}\ k & p & : & e_o & \to e_p
\end{array}
$$

Figure 3.9. Sugared syntax of a phased-form actor: a) stateful actor b) stateless actor

where e is a boolean expression and *Int* stands for a literal integer constant. This syntactic form ensures that the eligible rule sets are manifest in the actor text. ϕ_0 must also be a set of literal integer constants.

Often, some rules do not perform all of the three possible actions. For convenience, I omit the *update* clause if the update action is **update** $s = s$, and the *output* clause if the output action is **output** $s = [\,]$ (or a tuple containing an appropriate number of empty sequences).

Some examples will clarify how phased-form works. The version of *sumsqrs* that non-deterministically reads a token from either input is written with four rules; it is shown in figure 3.10. (An underscore represents a "don't-care" pattern.) The *delay* operator was given as a (strict) standard-form actor on page 59. A strict actor is not, however, a very good way to represent *delay*, since a delay must usually be able to produce an output token before reading an input token. With phased form, we can write a version of *delay* that can produce and consume tokens in arbitrary interleaving, shown in figure 3.11. In this example, the internal state is a list of tokens; in phase 1, a token is removed from this list and written to the output; in phase 2, a token is read from the input and appended to this list.

$$
\begin{array}{l}
\textsf{actor } \mathit{sumsqrs} \equiv \\
\quad \begin{array}{ll}
\textsf{init} & 0 \\
\textsf{start} & \{1,3\}
\end{array} \\
\quad \begin{array}{llllll}
\textsf{rule } 1 & ([x],[\,]) & : & \textsf{update}(\_) & = & x^2 \\
 & & & \textsf{select}(\_) & = & \{2\} \\
\textsf{rule } 2 & ([\,],[y]) & : & \textsf{output}(s) & = & [s+y^2] \\
 & & & \textsf{select}(s) & = & \{1,3\} \\
\textsf{rule } 3 & ([\,],[y]) & : & \textsf{update}(\_) & = & y^2 \\
 & & & \textsf{select}(\_) & = & \{4\} \\
\textsf{rule } 4 & ([x],[\,]) & : & \textsf{output}(s) & = & [x^2+s] \\
 & & & \textsf{select}(s) & = & \{1,3\}
\end{array}
\end{array}
$$

Figure 3.10. The non-deterministic *sumsqrs* actor

For stateless actors, the simpler syntax of figure 3.9b is used instead.[2] For example, *select* is stateless, and has three rules:

$$
\begin{array}{l}
\textsf{actor } \mathit{select} \equiv \\
\quad \begin{array}{llllll}
\textsf{start} & \{1\} & & & & \\
\textsf{rule } 1 & ([b],[\,],[\,]) & : & [\,] & \rightarrow & \mathit{if}\ b\ \mathit{then}\ \{2\}\ \mathit{else}\ \{3\} \\
\textsf{rule } 2 & ([\,],[x],[\,]) & : & [x] & \rightarrow & \{1\} \\
\textsf{rule } 3 & ([\,],[\,],[y]) & : & [y] & \rightarrow & \{1\}
\end{array}
\end{array}
$$

Finally, the non-deterministic *merge* actor has two rules:

$$
\begin{array}{l}
\textsf{actor } \mathit{merge} \equiv \\
\quad \begin{array}{llllll}
\textsf{start} & \{1,2\} & & & & \\
\textsf{rule } 1 & ([x],[\,]) & : & [x] & \rightarrow & \{1,2\} \\
\textsf{rule } 2 & ([\,],[y]) & : & [y] & \rightarrow & \{1,2\}
\end{array}
\end{array}
$$

[2] The eligible rule set is of course a kind of state; since, however, I am treating the eligible rules explicitly, the term "state" will refer to other information maintained between firings.

$$
\begin{array}{llllcl}
\textsf{actor}\ \mathit{delay}\, i\ \equiv & & & & & \\
\quad \textsf{init} & [i] & & & & \\
\quad \textsf{start} & \{1,2\} & & & & \\
& & & & & \\
\quad \textsf{rule}\ 1 & [\,] & : & \textsf{update}(x:xs) & = & xs \\
& & & \textsf{output}(x:xs) & = & [x] \\
& & & \textsf{select}(x:xs) & = & \mathit{if}\ \mathit{null}(xs)\ \mathit{then}\ \{2\}\ \mathit{else}\ \{1,2\} \\
& & & & & \\
\quad \textsf{rule}\ 2 & [x] & : & \textsf{update}(xs) & = & xs +\!\!+ [x] \\
& & & \textsf{select}(xs) & = & \{1,2\}
\end{array}
$$

Figure 3.11. The phased-form *delay* actor

$$
\mathit{desugar}\ \left[\!\!\left[\begin{array}{l}
\textsf{actor}\ \mathit{name}(v_1,\ldots,v_a)\ \equiv \\
\quad \textsf{init}\ s_0 \\
\quad \textsf{phase}\ \phi_0 \\
\quad \mathit{rule}_1;\ldots;\mathit{rule}_k
\end{array}\right]\!\!\right]
$$

$$
\equiv\ ((v_1,\ldots,v_a),\ s_0,\ \phi_0,\ \{\mathit{desugarR}\ [\![\mathit{rule}_1]\!],\ldots,\mathit{desugarR}\ [\![\mathit{rule}_k]\!]\})
$$

$$
\mathit{desugarR}\ \left[\!\!\left[\begin{array}{llllcl}
\textsf{rule}\ i & p & \rightarrow & \textsf{update}(s) & = & e_u \\
& & & \textsf{output}(s) & = & e_o \\
& & & \textsf{select}(s) & = & e_s
\end{array}\right]\!\!\right] \equiv\ (p,\ \lambda s\,.\,e_u,\ \lambda s\,.\,e_o,\ \lambda s\,.\,e_s)
$$

Figure 3.12. Desugaring a phased-form actor

Phased-form actors are desugared in a similar manner to standard-form actors. Figure 3.12 gives the translation: a desugared actor is a four-tuple containing parameters, initial state, initial rule set, and the set of rules; each rule is a four-tuple of input patterns and the update, output, and select actions. Instantiation also proceeds in a similar fashion to standard-form actors, but in this case, an instantiated phased-form actor is a triple of initial state, initial rule set, and the set of rules.

For example, an instantiated *merge* actor is

$$
\begin{array}{ll}
((\,),\ \{1,2\},\ \{ & (([x],[\,]),\quad \lambda s\,.\,s,\quad \lambda s\,.\,[x],\quad \lambda s\,.\,\{1,2\}), \\
& (([\,],[y]),\quad \lambda s\,.\,s,\quad \lambda s\,.\,[y],\quad \lambda s\,.\,\{1,2\}) \\
\quad \}) &
\end{array}
$$

3.3.2 Phase graphs

The possible orders in which rules can be selected are captured by the *phase graph*. A *phase* is execution of a rule, so the phase graph is a useful tool for characterising and analysing the behaviour of phased-form actors. As a notational convenience, rule identifiers k are used interchangeably with the rules themselves. Figure 3.13 shows the phase graphs of the four example actors given previously, annotated with their consumption and production vectors. Note that every vertex of the phase graph is in a cycle; this ensures that the actor will continue to execute while it has input data.[3]

The phase graph has two kinds of vertex. *Rule vertices* correspond to a rule, and are labelled with the rule's identifier; each rule occurs exactly once. *Choice vertices* correspond to non-deterministic choice, and are named $choice_\Phi$, where Φ is the set of its successor vertices. Thus, a choice vertex occurs once for each non-singleton set in the *select* actions. Choice vertices are coloured black; they are not labelled with their names since this is clear from context. The vertex corresponding to the initial eligible rules set is marked by an asterisk.

The edges of the graph correspond to allowable transitions between rules. For each singleton set produced by a rule's select action, there is an edge from that rule vertex to the vertex corresponding to the single rule; for each non-singleton set, there is an edge from the rule vertex to a choice vertex. More precisely,

$$\begin{aligned} succ\ [\![(\_,\_,\_,\lambda s\,.\,e_s)]\!] &= succ'\ [\![e_s]\!] \\ succ\ [\![choice_\Phi]\!] &= \Phi \end{aligned}$$

$$\begin{aligned} succ'\ [\![\{\phi\}]\!] &= \{\phi\} \\ succ'\ [\![\Phi]\!] &= \{choice_\Phi\} \\ succ'\ [\![\mathit{if}\ b\ then\ \Phi\ else\ e_s]\!] &= succ'(\Phi)\ \cup\ succ'(e_s) \end{aligned}$$

A sequence of rule firings corresponds to a path through the phase graph. We will write

[3] It may be useful to have a special phase, called say the initialisation phase, which has no predecessors (and is therefore not in a cycle). This could yield some practical advantages, as it corresponds to computation that can proceed before zero time.

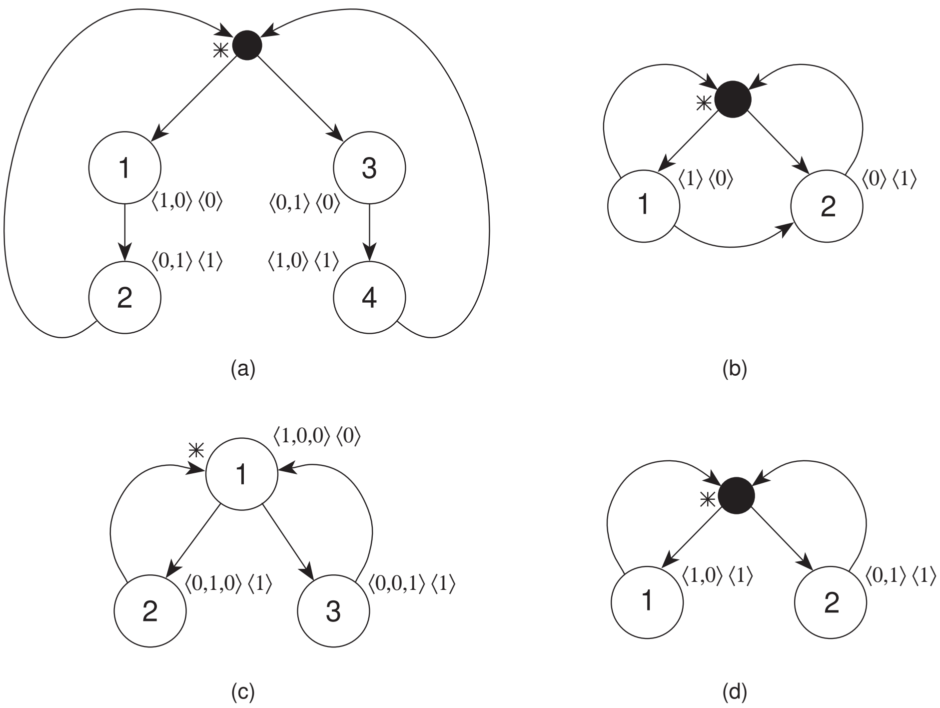


Figure 3.13. Phase graphs: a) *sumsqrs*; b) *delay*; c) *select*; d) *merge*

a k-length path v as a vector $\langle \phi_1, \ldots, \phi_k \rangle$. Let the *paths* function return the set of paths from one vertex to another. If some vertex ϕ_a in the path intersects a cycle, say, $\langle \phi_a, \phi_b \rangle$, the set returned by *paths* is infinite, including the paths $\langle \ldots, \phi_a, \ldots \rangle$, $\langle \ldots, \phi_a, \phi_b, \phi_a, \ldots \rangle$, $\langle \ldots, \phi_a, \phi_b, \phi_a, \phi_b, \phi_a, \ldots \rangle$, and so on.

paths can be used to specify the *cycles* relation, which returns the set of cycles starting at a given node:

$$cycles(\phi) = \bigcup \{paths(\phi, \psi) \mid \phi \in succ(\psi)\}$$

3.3.3 Semantics

Phased form allows a more precise description of the order in which tokens are consumed and produced. As for standard-form, each phase fulfils two conditions:

– All input tokens are consumed before any output tokens are produced.

– A phase must complete before another phase can begin.

Each *phase* is thus "strict"; because, however, an actor firing proceeds in a series of phases, the *actor* is non-strict. I will define the meaning of a firing and other forms of execution in terms of paths through the phase graph; this falls short of a full semantics of phased-form actors, but is sufficient to describe several important characteristics.

A path is represented by a vector of rules:

$$\mathit{Path} ::= \langle \mathit{Rule} \rangle$$

Given a k-length path through the graph $v = \langle \phi_1, \ldots, \phi_k \rangle$, we need to know the consumption and production vectors along that path, and the meaning of "executing" v. Let $C(R)$ and $P(R)$ be the consumption and production vectors of rule R in the same way as for standard-form actors. The consumption and production vectors of path v are then given by

$$\begin{aligned} \mathcal{C}(v) &= C(\phi_1) + \ldots + C(\phi_k) \\ \mathcal{P}(v) &= P(\phi_1) + \ldots + P(\phi_k) \end{aligned}$$

where the addition operator (+) is assumed to extend point-wise to vectors.

Firing a single rule is essentially identical to firing a single simple rule of a standard-form actor. *read*, *update*, and *output* are the same as in section 3.2.3, but slightly modified because a rule is now a four-tuple instead of a triple. Let the *step* semantic function execute a single rule:

$$\begin{array}{lcl} \mathit{step}\ [\![\mathit{Rule}]\!] & :: & \mathit{Input} \to \sigma \to \mathit{Input} \times \sigma \times \mathit{OutputSeg} \\ \mathit{step}\ [\![R]\!]\ i\ s & = & (i', s', o') \\ & & \quad \mathit{where} \\ & & \quad \begin{array}{lcl} (is, i') & = & \mathit{read}\ [\![R]\!]\ i \\ s' & = & \mathit{update}\ [\![R]\!]\ \mathit{is}\ s \\ o' & = & \mathit{output}\ [\![R]\!]\ \mathit{is}\ s \end{array} \end{array}$$

Note that I am not trying to give a full semantics here; in particular, I am examining only the computation performed given a path through the phase graph—that is, with state update and output production, but not with calculation of the next eligible rule set. A full semantics would need to calculate sets of input-to-output sequence mappings. (Sets are required to account for non-deterministic choice.)

Execution along a given path is given by the *execute* function. *execute* takes a path—that is, a vector of rules—and returns the input-output mapping produced by executing those rules in order. I use the (:>) operator on vectors in the same way as the cons (:) operator on sequences.

$$
\begin{array}{rcl}
\mathit{execute}\ [\![\mathit{Path}]\!] & :: & \mathit{Input} \rightarrow \sigma \rightarrow \mathit{OutputSegment} \times \sigma \\
\mathit{execute}\ [\![\langle\,\rangle]\!]\ i\ s & = & ([\,], s) \\
\mathit{execute}\ [\![R :> Rs]\!]\ i\ s & = & (o +\!\!+\ o', s'') \\
 & & \mathit{where} \\
 & & \begin{array}{lcl} (i', s', o) & = & \mathit{step}\ [\![R]\!]\ (i, s) \\ (o', s'') & = & \mathit{execute}\ [\![Rs]\!]\ i'\ s' \end{array}
\end{array}
$$

Now, a *complete firing* of a phased-form actor is execution along any cycle from the start vertex:

Definition 3 (Complete firing) *A* complete firing *of a phased-form actor is execution along any cycle of the phase graph beginning on the start vertex* ϕ_0*:*

$$\mathit{execute}\ [\![v]\!]\ \mathit{where}\ v \in \mathit{cycles}(\phi_0)$$

If there are cycles in the graph that do not pass through ϕ_0, then there are an infinite number of complete firing paths. Section 3.3.4 gives an example of an actor like this. An actor with a infinite number of complete firing paths can be the phased-form equivalent of SDF if it also satisfies a condition on the consumption and production vectors:

Definition 4 (Phased synchronous dataflow) *An actor with initial vertex* ϕ_0 *is* phased synchronous dataflow *iff: i)* $\{v_1, \ldots, v_k\} = \mathit{cycles}(\phi_0)$ *is finite; and ii) the consumption and production vectors along all cycles are equal:*

$$\mathcal{C}(v_i) = \mathcal{C}(v_j)\ \wedge\ \mathcal{P}(v_i) = \mathcal{P}(v_j),\ \forall i, j : 1 \le i, j \le k$$

By summing the vectors around each of the cycles in figure 3.13 and applying this criterion, it can be seen that *sumsqrs* is phased SDF, but the other three are not. Note that a phased SDF actor is not necessarily deterministic. Here is an example of an actor that exhibits this kind of "internal" non-determinism:

$$
\begin{array}{llcll}
\textsf{actor}\ \mathit{silly} & \equiv & & & \\
\quad \textsf{start} & \{1,2\} & & & \\
\quad \textsf{rule}\ 1 & [x] & : & [x] & \rightarrow \ \{1,2\} \\
\quad \textsf{rule}\ 2 & [x] & : & [x+1] & \rightarrow \ \{1,2\}
\end{array}
$$

Note also that the definition of phased SDF does not include all actors that could be considered to be SDF actors—see section 3.3.4 for an example.

In the introduction to this section, I showed by illustration that a non-deterministic actor can produce a deterministic process. For a process to be deterministic, a non-deterministic choice must not affect the computation performed by the actor. A necessary condition for a deterministic process is that, for each choice vertex ϕ, all paths v in *cycles*(ϕ) have equal $\mathcal{C}(v)$ and $\mathcal{P}(v)$. A stronger, necessary and sufficient condition must examine the output state and sequences:

Definition 5 (Kahn actor) *A phased dataflow actor forms a Kahn process iff for each non-deterministic choice ϕ in its phase graph,* cycles(ϕ) *is finite and either*

1. *All paths $v \in$ cycles(ϕ) have equal values of* execute$[\![v]\!]$, *or*
2. (a) *All paths v in* cycles(ϕ) *have equal values of* fst $\cdot$ execute$[\![v]\!]$, *and*

   (b) *All paths v in* cycles(ϕ) *write s before using it (see below).*

In condition (1), all paths behave exactly the same; in condition (2), only the input-output mappings are the same, but the state is ignored in subsequent decisions anyway. Condition 2b requires further explanation: *execute*$[\![v]\!]$ need not return the same state for all v in *cycles*(ϕ), provided that the differing states are ignored. Define *const* $x\ y = x$. A path $v = \langle \phi_1, \ldots, \phi_k \rangle$ writes the state before using it iff there exists an i such that ϕ_i has an update action of the form *const* e_u and $\forall j \leq i$, ϕ_j has output and select actions of the form *const* e_o and *const* e_s.

This proposition is the main insight of this section. A practical means of determining if the conditions are satisfied is to perform a reduction of *execute* on symbolic sequences. Consider *sumsqrs*. Its only choice vertex is $choice_{\{1,3\}}$, and $cycles(choice_{\{1,3\}}) = \{\langle 1,2\rangle, \langle 3,4\rangle\}$. We then have, for the path $\langle 1,2\rangle$,

$$
\begin{array}{ll}
\multicolumn{2}{l}{execute\ [\![\langle 1,2\rangle]\!]\ (([x_1,\ldots],[y_1,\ldots]),\ s)} \\
= & (o +\!\!+\ o', s'') \\
& where \\
& \begin{array}{lcl}(i',s',o) & = & step\ [\![1]\!]\ (([x_1,\ldots],[y_1,\ldots]),s) \\ (o',s'') & = & execute\ [\![\langle 2\rangle]\!]\ (i',s')\end{array} \\
= & (o +\!\!+\ o', s'') \\
& where \\
& \begin{array}{lcl}(i',s',o) & = & (([x_2,\ldots],[y_1,\ldots]), x_1^2, [\,]) \\ (o',s'') & = & execute\ [\![\langle 2\rangle]\!]\ (i',s')\end{array} \\
= & ([\,] +\!\!+\ o', s'') \\
& where \\
& (o',s'')\ =\ execute\ [\![\langle 2\rangle]\!]\ (([x_2,\ldots],[y_1,\ldots]), x_1^2) \\
= & ([\,] +\!\!+\ o', s'') \\
& where \\
& (o',s'')\ =\ step\ [\![2]\!]\ (([x_2,\ldots],[y_1,\ldots]), x_1^2) \\
= & ([\,] +\!\!+\ o', s'') \\
& where \\
& (o',s'')\ =\ ([x_1^2 + y_1^2], x_1^2) \\
= & ([x_1^2 + y_1^2], x_1^2)
\end{array}
$$

Following a similar process for the cycle $\langle 3,4\rangle$ yields the expression $([x_1^2 + y_1^2], y_1^2)$. This satisfies condition 2a above. Since the state is written in rules 1 and 3, but not read until rules 2 and 4, condition 2b is also satisfied, and *sumsqrs* therefore forms a Kahn process. Of the other graphs in figure 3.13, *select* is a Kahn process because it has no choice vertices; *delay* and *merge* are not, because they fail the equal consumption and production vector condition.

3.3.4 Cyclo-static and multi-phase integer dataflow

Buck [27] proposed *multi-phase* integer dataflow actors. An integer dataflow actor is such that the number of tokens consumed or produced on each arc is a function of an integer-valued control token. The REPEAT type of actor consumes an integer control token in its first phase, and then executes another phase the number of times given by that token. For example, Buck's *iota* actor reads a control token with value n, then outputs the sequence of integers from 1 to n inclusive. As a phased actor, we can write *iota* thus:

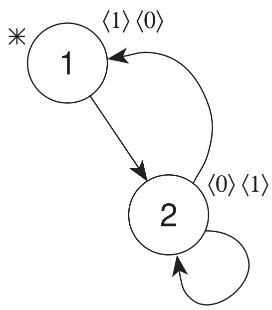


Figure 3.14. Phase graph of *iota*

$$
\begin{array}{llllcl}
\textsf{actor}\ iota\ \equiv & & & & & \\
\quad \textsf{init} & (0,0) & & & & \\
\quad \textsf{start} & \{1\} & & & & \\
& & & & & \\
\quad \textsf{rule}\ 1 & [n] & : & \textsf{update}(\_) & = & (1,n) \\
& & & \textsf{select}(\_) & = & \{2\} \\
& & & & & \\
\quad \textsf{rule}\ 2 & [\,] & : & \textsf{update}(i,n) & = & (i+1,n) \\
& & & \textsf{output}(i,n) & = & [i] \\
& & & \textsf{select}(i,n) & = & \mathit{if}\ i=n\ \mathit{then}\ \{1\}\ \mathit{else}\ \{2\}
\end{array}
$$

Figure 3.14 shows the phase graph of *iota*. Not surprisingly, the actor is not phased SDF (there are infinitely many cycles from the start vertex), although it is deterministic. Except for its consumption and production vectors, this graph is exactly the same as that of another special class of actor, the *cyclo-static synchronous dataflow* (CSSDF) actors proposed by Engels *et al* [43]. The general form of a CSSDF actor is

$$
\begin{array}{llllcl}
\textsf{actor}\ CSSDF(n)\ \equiv & & & & & \\
\quad \textsf{init} & (0,0) & & & & \\
\quad \textsf{start} & \{1\} & & & & \\
& & & & & \\
\quad \textsf{rule}\ 1 & p_1 & : & \textsf{update}(\_) & = & (1,n) \\
& & & \textsf{output}(\_) & = & o_1 \\
& & & \textsf{select}(\_) & = & \{2\} \\
& & & & & \\
\quad \textsf{rule}\ 2 & p_2 & : & \textsf{update}(i,n) & = & (i+1,n) \\
& & & \textsf{output}(i,n) & = & o_2 \\
& & & \textsf{select}(i,n) & = & \mathit{if}\ i=n\ \mathit{then}\ \{1\}\ \mathit{else}\ \{2\}
\end{array}
$$

Compared to *iota*, this actor has its value of n supplied as a parameter, instead of being read from an input. Although this actor is SDF, the definition of phased SDF cannot recognise it as such. Once instantiated and n is known, the actor can be expanded into $n+1$ phases; in this case, the phase graph contains only a single cycle and is recognisable as SDF. Lauwereins *et al* [85] show that a network of CSSDF actors can be scheduled statically, and that the schedule can have (with certain assumptions about execution timing) lower execution time than an SDF static schedule.

Both cyclo-static dataflow and multi-phase integer dataflow actors are used to reduce the memory requirements required by a statically-generated schedule. Buck extends scheduling techniques developed for boolean dataflow (*switch* and *select* are the canonical boolean dataflow actors) to determine if a graph containing integer dataflow actors has bounded schedules, and to "cluster" a graph in order to extract control structures such as *for*-loops and *do-while* loops [26].

3.3.5 Execution mechanisms

Phased-form actors are non-strict. In this section, I examine the implications for dynamic scheduling of phased-form actors; in the next I will look at the implications for networks.

So far, I have not specified any restrictions on the granularity of phases. For example, *sumsqrs* could also have been written

$$\begin{array}{l} \textsf{actor}\ \mathit{sumsqrs}\ \equiv \\ \quad \textsf{start}\quad \{1\} \\ \quad \textsf{rule}\ 1\quad ([x],[y])\ :\ [x^2+y^2]\ \rightarrow\ \{1\} \end{array}$$

which is the same as the standard-form version and is thus strict. Phased-form is more useful if the actor fulfils these (informal) criteria:

1. It consumes the minimum number of input tokens needed to produce a single output token;
2. it outputs that token "as soon as possible" and performs no further computation; and
3. it outputs any other tokens that can be output without performing any further computation.

The criteria, if met, ensure that an actor performs only the minimum amount of input, output, and computation necessary to keep data flowing through the graph. Consider the effect on a data-driven scheduler: since input patterns contain only irrefutable token patterns, a scheduler need only count available tokens, not read their values. The scheduler can thus fire any actor that has sufficient input tokens (and output space, if buffers are bounded) for one of its eligible rules.

Demand-driven execution is also possible. A phased-form actor is likely to perform less work than a strict actor each time it is fired. Let j be the output channel on which a token is demanded. The *target* phase set Φ_t of a q-phase actor is:

$$\Phi_t = \{\phi \mid \phi \leftarrow \{1 \ldots q\}, P[\![\phi]\!](j) > 0\}$$

To meet the demand, the actor must *execute* any path v from an element of the current eligible phase set Φ_c to an element of Φ_t:

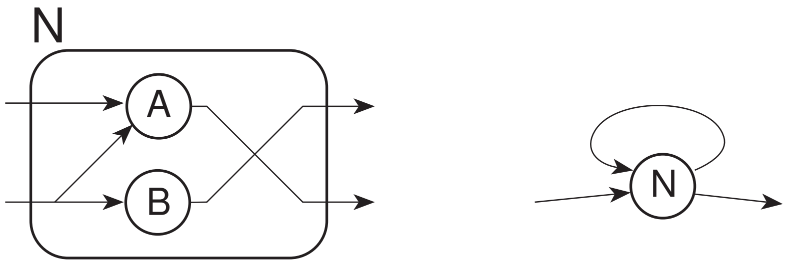
$$v \in \bigcup\{paths(a, b) \mid a \leftarrow \Phi_c, b \leftarrow \Phi_t\}$$

Note that the value of j cannot affect the path taken by the actor—there is no mechanism by which the actor can know on which output data has been demanded. This would be yet another possible source of non-determinism, and is I believe best avoided.

3.3.6 Hierarchy and strictness

One of the issues raised by Lee in his examination of dataflow actors [87] is the dataflow equivalent of procedural abstraction. The natural choice for this role is the dataflow network, and block-diagram systems such as Ptolemy hierarchically compose actors of networks of actors. As Lee points out, however, the semantics of a hierarchical SDF actor are not in fact the same as the network of actors it represents.

Consider the hierarchical actor N of figure 3.15a, where component actors A and B are homogeneous, strict, actors. Considered alone, N is a fine example of a homogeneous, strict actor. When connected in the network of figure 3.15b, however, the network deadlocks: N cannot fire because its top input has no data, and its top output cannot produce data until it fires. Of course, this is no different from how any other strict 2×2 actor would behave. It is, however, different from how N's internal network would behave if it were substituted in place of N. In that case, B could fire, producing a token on the top output;



(a) (b)

Figure 3.15. Deadlock of a hierarchical actor: a) the actor network; b) a deadlocked network

A could then fire, producing a token on the bottom output; the cycle would then repeat. Ptolemy side-steps this problem by expanding all hierarchical actors before scheduling. Nonetheless, it is somewhat unsatisfactory.

A phased version of this actor, say N', allows computation to proceed *before* receiving all of its input. Supposing for simplicity that A and B are stateless and writing, for example, $A(x, y)$ to mean the token produced by firing A with input tokens x and y, we can write N' as

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{actor}\ N' \ \equiv} \\
\quad \mathsf{init} & 0 & & & & \\
\quad \mathsf{start} & \{1\} & & & & \\
& & & & & \\
\quad \mathsf{rule}\ 1 & ([\,], [y]) & : & \mathsf{update}(s) & = & y \\
& & & \mathsf{output}(s) & = & ([B(y)], [\,]) \\
& & & \mathsf{select}(s) & = & \{2\} \\
& & & & & \\
\quad \mathsf{rule}\ 2 & ([x], [\,]) & : & \mathsf{output}(s) & = & ([\,], [A(x, s)]) \\
& & & \mathsf{select}(s) & = & \{1\}
\end{array}
$$

When placed into the network of figure 3.15b, N' does not deadlock—it behaves in the same way as its internal network would if connected the same way. N' can fire rule 1 (B) when it has data on the bottom input; in the next phase, it can fire rule 2 (A), consuming a new input token as well as the token received in the previous phase; it then repeats the cycle. Phased form thus offers the promise of consistent semantics of hierarchical actors and their corresponding networks.

An algorithm to find the phased form of a hierarchical actor is an open research prob-

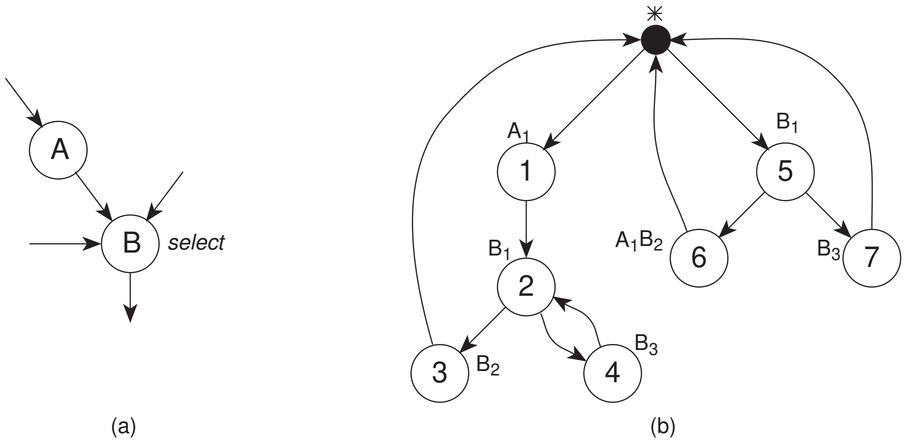


Figure 3.16. A phased network example: a) network graph; b) phase graph

$$
\begin{array}{l}
\textsf{actor}\ NET\ \equiv \\
\quad \begin{array}{ll}
\textsf{init} & 0 \\
\textsf{start} & \{1,5\}
\end{array} \\
\\
\quad \begin{array}{lllrcl}
\textsf{rule}\ 1 & ([\,],[x],[\,]) & : & \textsf{update}(\_) & = & A(x) \\
 & & & \textsf{select}(\_) & = & \{2\} \\
 & & & & & \\
\textsf{rule}\ 2 & ([b],[\,],[\,]) & : & \textsf{output}(\_) & = & [\,] \\
 & & & \textsf{select}(\_) & = & \mathit{if}\ b\ \mathit{then}\ \{3\}\ \mathit{else}\ \{4\} \\
 & & & & & \\
\textsf{rule}\ 3 & ([\,],[\,],[\,]) & : & \textsf{output}(a) & = & [a] \\
 & & & \textsf{select}(a) & = & \{1,5\} \\
 & & & & & \\
\textsf{rule}\ 4 & ([\,],[\,],[y]) & : & \textsf{output}(\_) & = & [y] \\
 & & & \textsf{select}(\_) & = & \{2\} \\
 & & & & & \\
\textsf{rule}\ 5 & ([b],[\,],[\,]) & : & \textsf{output}(\_) & = & [\,] \\
 & & & \textsf{select}(\_) & = & \mathit{if}\ b\ \mathit{then}\ \{6\}\ \mathit{else}\ \{7\} \\
 & & & & & \\
\textsf{rule}\ 6 & ([\,],[x],[\,]) & : & \textsf{output}(\_) & = & [A(x)] \\
 & & & \textsf{select}(\_) & = & \{1,5\} \\
 & & & & & \\
\textsf{rule}\ 7 & ([\,],[\,],[y]) & : & \textsf{output}(\_) & = & [y] \\
 & & & \textsf{select}(\_) & = & \{1,5\}
\end{array}
\end{array}
$$

Figure 3.17. The example phased network as a phased-form actor

lem. The goal is to find an algorithm that approximates at compile-time the set of possible run-time behaviours of the network. To illustrate, figure 3.16a is an example hierarchical actor, called say NET, and figure 3.16b is its phase graph if actor A is homogeneous and stateless. Each phase of NET represents one or more phases of A or B. The phased form description of the network (obtained by tracing execution of the network by hand) is shown in figure 3.17.

3.4 Summary

This chapter has made two contributions to dataflow theory. The first is a semantics of dataflow process networks. Given a description of an actor's firing function, the semantics gives the precise meaning of the dataflow process formed by that actor. The semantic description is not, however, as concise or elegant as I would like; in future work, I would like to re-formulate the semantics to make it more so.

The second contribution is the description of phased actors. Although the semantic description is incomplete, this work nonetheless offers a new characterisation of dataflow actors, with interesting indications for further work. In particular, the task of determining the phased form of a complete network should prove challenging. It remains to be seen whether phased-form actors will lead to more efficient scheduling of certain kinds of dataflow actors and networks, or whether they can only be a descriptive tool.

The formal description of actors has also paved the way for following chapters: in section 3.2.5, I argued that any dataflow actor can be implemented with five schema. If we can implement functions that correspond to these schema, then any function that can be expressed in terms of those functions can be translated into a dataflow network.

Chapter 4

Visual Haskell

Visual Haskell is a visual, dataflow-style language based on Haskell. Its intent is to ultimately provide a complete visual syntax for Haskell, for use as a program visualisation tool, as well as a programming language in its own right. In both of these roles, it is complementary to Haskell's standard textual form, not a replacement or "improved" notation. Presently, only part of Haskell is supported (notable omissions are class and type declarations). Even so, it is sufficiently well-developed to be useful.

Visual Haskell evolved out of a need to explain some of the work in this thesis to non-functional-programmers. It began as an *ad-hoc* notation for "drawing" functional programs (see [114]); later, Ken Dawson of UTS implemented a prototype editor for this language [41], stimulating its development into a more precise and usable language.

The style of Visual Haskell is based on dataflow: programs are described as data and function boxes, connected by arcs representing flow of data. It is in some ways a superset of dataflow, supporting not only "structured dataflow" [82] constructs such as conditionals and *case*-statements, but also pattern-matching, higher-order functions, and scoping. And it has Haskell's polymorphic type system, although no visual notation for it. Because, however, of its lexical scoping, unrestricted Visual Haskell programs do lose one of the key advantages of dataflow: explicit data-dependencies throughout the whole program.

Of particular interest in Visual Haskell is the way it handles higher-order functions: because Visual Haskell is curried, higher-order functions are as easy to construct as first-order functions. There is no need for special "function slots" [99], and no enforced distinction between first-order and higher-order functions as in VisaVis [110]. Any construct—a *case*-expression, say—can be a function, and can be applied in the same way as a named function.

The next two sections briefly survey work on visual languages, and then introduce Visual Haskell by example. Following sections formally give the syntax of Visual Haskell as a translation from Haskell's abstract syntax into a concrete visual syntax.

4.1 Related work

Visual programming languages (VPLs) are relatively new. Although still the subject of disagreement on their worth, there are nonetheless many "real-world" projects that use visual programming [29]. The success of these systems indicates very strongly that visual programming—if applied to appropriate problem domains and supported by appropriate tools—offers a kind of understanding that is absent in purely-textual languages.

Two of the most successful interface paradigms in visual programming languages are *dataflow* and *forms* [80]. In signal processing, dataflow visual languages have been used for some years [17, 88], because of the common use of block diagrams in describing signal processing systems. They have been used successfully in other fields as well, such as image processing and instrumentation [112]; Hils [59] surveys a number of dataflow visual languages. A form is a layout of cells, each containing a formula that computes its value, usually in terms of the values of other cells. Many dataflow languages use forms as well as dataflow: Ptolemy [88], for example, uses a form to enter actor parameters, and dataflow to connect streams; Labview [82] uses forms as "front panels" for "virtual instruments."

Informal diagrammatic notations have been used to describe or explain functional programs for some time. Reade [113], for example, explains function application using box-and-arrow diagrams; Kelly [78] illustrates networks of streams and processes written using an annotated pure functional language; Waugh *et al* [152] illustrate the effect of program transformation on parallelism.

There have also been proposals for formal visual functional languages. Cardelli [31] proposed a visual functional language in which function application is denoted by juxtaposition of a function name with its arguments; bracketing is denoted by containing boxes; pattern-matching can select one of several expressions; bindings are denoted by placing a

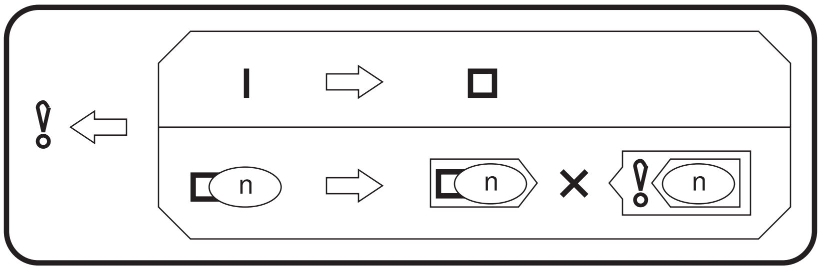


Figure 4.1. The factorial function in Cardelli's language

left-pointing arrow between a name and an expression. Figure 4.1 shows this definition of the factorial function in Cardelli's language:

```
fact 0     = 1
fact (n+1) = (n+1) * fact n
```

There are two frames: one for each clause. In the top frame, the vertical bar represents zero, the box one. In the bottom frame, the heavy rectangle attached to the ellipse represents "+1". Boxes enclosing arguments "point" to the function applied to them. The whole expression is bound to the icon "!".

Cardelli suggests that the visual language is most useful when itself applied to the manipulation of two-dimensional data structures. He gives examples of manipulations on boxes containing characters and other symbols, and of functions which place a frame around a box, and which decompose a compound box into a list of its components.

Although not a functional language, Najork and Golin's visual dataflow language ESTL (Enhanced Show-and-Tell) [99] has many similar features, including higher-order functions and a polymorphic type system. A key concept in ESTL is *inconsistency*: an arc that passes through an inconsistent box cannot carry data. For example, figure 4.2 illustrates the factorial function in ESTL. To the left is its type declaration: factorial is a function from integers to integers. The main box is split into two sub-boxes. In the left one, the input data flows into a box labelled "0"; if the input data is zero, then the box's value, one, is passed to the output. Otherwise, the box is inconsistent and produces no data. On the right, the data flows into the predicate box "> 0"; provided the input datum satisfies the predicate, the containing box is not inconsistent, and data flows to produce an output value. If the datum is zero, then the box is inconsistent and produces no data.

Inconsistency is thus used to perform the role of conditional execution; together with

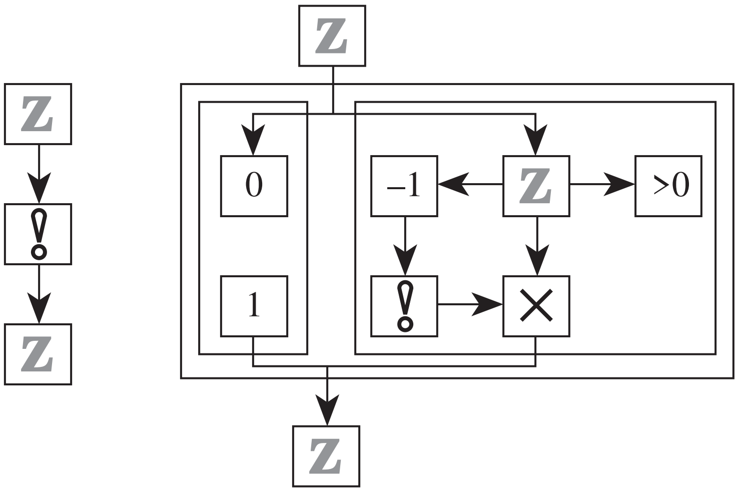


Figure 4.2. The factorial function in ESTL

ESTL's structured type notation, it also performs pattern-matching. Iteration is also governed by inconsistency: the iteration construct repeatedly applies its internal dataflow graph to its previously-produced data for as long as the graph is consistent.

Higher-order functions are supported by function slots, in which a function icon is placed inside the icon of a higher-order function. Visual Haskell's argument slots (section 4.5.1) are a generalisation of ESTL's function slots. ESTL also includes a visual notation for a polymorphic type system; this may be a good starting point for Visual Haskell's missing type notation.

Poswig *et al* describe VisaVis [110], a visual functional language based on Backus' functional programming language FP [11]. The language is essentially a visual dataflow language, with explicit support for higher-order functions: function arguments are placed within icons of higher-order functions. The VisaVis editor, implemented in Smalltalk, supports some interesting concepts to make the act of programming easier. For example, a first-order function icon can be dragged over a higher-order function icon to make the first-order function into an argument. Dragging a box connected to the output of a function over a box connected to the input of a second function cause the functions to become connected by an arc. I do not consider issues such as these at all in this chapter. VisaVis does enforce an artificial (in my view) distinction between first-order and higher-order functions, mainly because it is based on FP.

Kelso also proposes a visual functional language in which programmer interaction influences the design of the language [79]. Expression graphs are similar to other dataflow languages; however, inputs to the expression are connected to terminator nodes containing the argument type; outputs are connected to terminator nodes labelled with the type of the entire expression (that is, a function type). Application of a function to an argument is achieved by dragging an output node onto an input node. Sub-expressions can be collapsed into a single node. Mechanisms such as this are important to reduce the "clutter" of complex visual expressions.

Visual languages cannot be represented by conventional grammars and other means of describing programs, and are often described by example. There is, however, an increasing amount of work on grammars for visual syntax; this is particularly relevant to work on automatic generation of visual language parsers, as in [37]. Golin and Reiss [53], for example, use a "picture layout grammar." The underlying grammar model is based on multisets to eliminate the ordering implicit in textual grammars. A number of primitive operators specify relationships between pictures—for example, the production $A \rightarrow contains(B, C)$ matches a picture with a production C inside a production B. More complex productions can be defined in terms of these operators, together with additional constraints on picture attributes.

A more general formalism, conditional set rewrite systems (CSRS), is proposed by Najork and Kaplan [101]. A CSRS consists of a set of rewrite rules on sets of terms, each governed by a predicate on terms. For example, the rewrite rule

$$box(b), num(n, p) \rightarrow \; fbox(b), \quad \textbf{if}\; inside(p, b)$$

states that b is a "finished box" if a number is located inside it. If, using this rewrite rule together with several others, a picture can be rewritten to a single finished box, then it is valid in the visual language defined by the rewrite rules. Najork and Kaplan show how a CSRS can be used not only to specify visual syntax, but also (with a different set of rewrite rules) to *translate* a picture into a textual language. They also show that a CSRS can specify a three-dimensional visual language (presumably motivated by their work on Cube, a three-dimensional logic programming language [100]).

Following sections of this chapter will touch on differences between Visual Haskell and visual dataflow languages. One of motivations for using a functional language in this thesis is the apparent correspondence between functional languages and dataflow. This is highlighted by the fact that languages for programming dataflow machines are often functional [1]. On closer examination, however, there are some important differences.

Ambler *et al*, in their survey and comparison of programming paradigms [4], highlight some of the differences between the functional and dataflow paradigms. Because pipeline dataflow operates on streams of data, its evaluation mechanism is neither strict nor non-strict (see section 5.3). Functions are not first-class objects in dataflow, whereas they are in Visual Haskell—figure 4.17a, for example, shows a function value (a partially-applied λ-abstraction) flowing along an arc.

Ambler *et al* do not comment on structuring mechanisms for dataflow, such as described by Kodosky *at al* [82]. As I point out in section 4.4.2, Visual Haskell includes some mechanisms similar to those used in structured dataflow languages, but more general. On the down-side, these constructs may interfere with dataflow schedulers and so on, since the "flow" of data into these constructs is not always explicit. Another powerful feature of functional languages that is rare in dataflow languages is pattern-matching, although ESTL [99] and CUBE [100] both support a form of pattern-matching.

Finally, a key difference between functional and dataflow languages is the treatment of I/O. In a functional language, I/O is performed in two key ways [44]: i) by calling system primitives with lazy lists representing the whole history of input or output; or ii) by calling system primitives with a continuation function. Recently, the idea of *monadic I/O* [73] has been adopted: this technique is superficially similar to continuations; however, the external "world" is single-threaded through the I/O code, allowing I/O to be performed immediately as a side-effecting operation. In contrast, dataflow languages perform I/O with special nodes that produce or consume streams.

4.2 An introduction to Visual Haskell

Visual Haskell is essentially a visual dataflow language: functions are represented by named boxes, and function application by arcs between boxes. Figure 4.3 shows the Visual Haskell form of the following version of the factorial function (from section 2.2):

```
fact n | n == 0    = 1
```

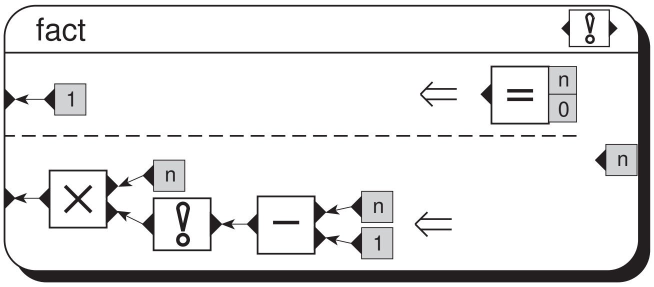


Figure 4.3. The factorial function in Visual Haskell

```
        | otherwise = n * fact (n - 1)
```

A function definition is displayed in its own window, indicated by the shadowed rounded rectangle. The name of the function, *fact*, and its icon, a stylised "!" character, are displayed in the window's title bar.

On the inside right of the window frame are the input patterns; in this case, there is only one, n. On the inside left are the result *ports*: the expressions connected to this port are the result values of the function. The guarded alternatives are separated by a dashed line. At the top right, for example, is the guard `n == 0`. (I use iconic, not textual, representations for binary operators. For example, $=$ instead of `==`, and $\times$ instead of `*`.) The boxes for n and 0 are adjacent to the box for `==`; this is an alternative notation to connecting boxes with arcs. The result expression—in this case, just the constant 1, is connected to a result port. Below the dashed line, the *otherwise* condition is "drawn" as blank space. Note the recursive call to *fact*, denoted by its icon.

Pattern bindings are shown by connecting an arc from an expression to a pattern graphic. The binding

```
twopi = 2 * pi
```

is shown in figure 4.4a. (A variable is the simplest pattern.) Note that the port on the pattern is semi-circular instead of triangular, and that the arc to a pattern port does not have an arrow. Patterns can be nested: they look rather like an expression drawn "backwards." For example, figure 4.4b shows the pattern

```
((x,y) : z : zs) = ...
```

The box divided by the jagged line is the pair constructor, and the tall narrow box is a list constructor. Note that patterns can either be connected by an arc, or juxtaposed in

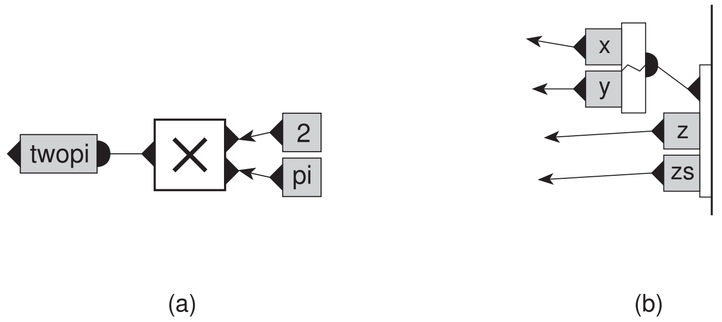


Figure 4.4. Patterns

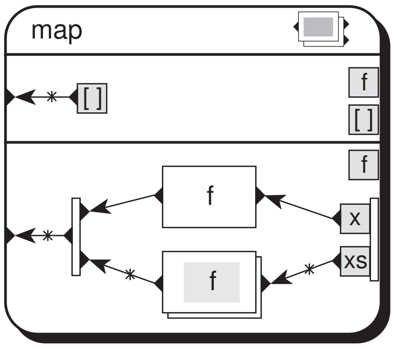


Figure 4.5. The *map* function

the appropriate location.

Figure 4.5 shows the following definition of Haskell's *map* function:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

As before, the function name and icon appear in the window's title bar. This time, however, the function definition has two clauses, separated in the visual syntax by a solid horizontal line. Note the use of the list constructor icon as a pattern at the right-hand side of the lower clause, as well as as a data constructor towards the left of this clause. The recursive call to *map* is denoted by the double-box icon. Its first argument, f, is shown *inside* the icon itself. This is an example of an argument slot—see section 4.5.1.

The "icons" I use in this thesis are rather plain for ease of drawing—more interesting icons would be used in a real graphical development system. Figure 4.6 shows the icons I use for some of the Haskell standard-prelude data constructors, while figure 4.7 shows icons for some standard-prelude functions on lists. Data constructors like $\texttt{Tuple}_2$ are produced

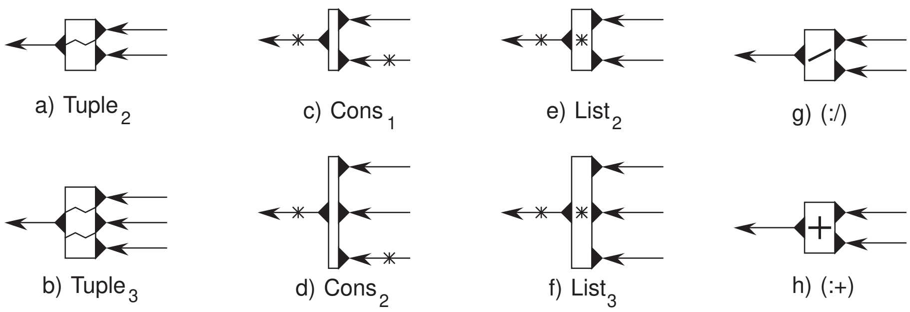


Figure 4.6. Icons for some standard prelude data constructors

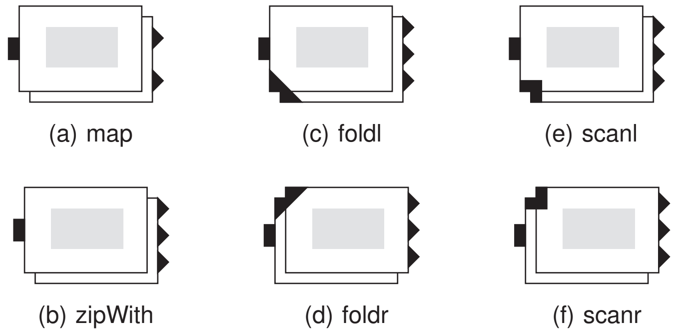


Figure 4.7. Icons for some standard prelude functions

by the desugaring translation (section 4.3.4).

The choice of iconic notation is I think a very personal one and one not readily systematised. In my choice of icons, I have used a visual theme to represent a class of icons and various (and arbitrary) decorations) to indicate specific icons. For example, a doubled box indicates a higher-order function over lists; higher-order functions over vectors (in chapter 5) have the obscured box filled black. Note that Visual Haskell does not depend on icons: they are "syntactic sugar," and any Visual Haskell program can be drawn with labelled boxes (the plain syntax described in section 4.4) instead of icons. Although excessive use of icons can certainly obscure program meaning to one unfamiliar with them, I believe that familiarity with a consistently-designed icon set yields a great improvement in ease of program understanding.

An icon-like annotation is also used in arcs to indicate type. For example, list-carrying arcs are decorated with an asterisk-like annotation. These annotations are a useful visual aid to program understanding, although I have not yet formalised their representation.

Figure 4.8. Function composition

Incidentally, I am in the habit of drawing Visual Haskell expressions so that the data flow is from right to left. The reason is simple: this is the same direction in which data "flows" in textual programs. For example, in $f \cdot g \cdot h$, data "flows" through h, g, and then f—see figure 4.8. Visual Haskell, however, does not care about the direction, and some of the diagrams in this thesis use left-to-right data flow instead.

A prototype visual editor was constructed by Ken Dawson as an undergraduate project [41]; the editor indicates that a two-view development system based on Visual Haskell is feasible. One interesting point noted during the construction of this prototype is that a visual parser is not needed to interpret visual programs. The prototype editor builds a graph representing the Visual Haskell program as the user "draws" the program. The editor is syntax-driven: in any given context, the editor only allows the user to draw pictures that make sense in Visual Haskell. To generate Haskell code from a completed Visual Haskell function, the system traverses the graph, generating (linear) Haskell code on the way—there is never any need to *parse* a two-dimensional picture. Cardelli made much the same observation, although his system translated abstract syntax trees into visual representations [31].

Despite the fact that Visual Haskell does not include all of Haskell, Dawson's system was able to generate executable Haskell functions. The type declaration for each function was inserted textually into a form associated with the function; other information in the form included the numbers of inputs and outputs. During generation of textual Haskell, the function type was simply written verbatim to the output file. Although Dawson's printer would only print acyclic graphs, this is an oversight that would be simple to correct.

4.3 Visual syntax preliminaries

Because Visual Haskell was designed to be an alternative notation for Haskell, this chapter specifies Visual Haskell in terms of Haskell. Although this approach may limit the scope of concepts that can be used in or introduced into this visual language—for example, ESTL's concept of inconsistency—it does have some advantages:

- Haskell has powerful features not found in visual dataflow systems: polymorphism and higher-order functions are two examples.
- It is likely to be easier to precisely define the semantics of Visual Haskell in terms of Haskell's semantics, than attempting to define a new semantics from scratch.
- It makes the notion of a "two-view" development system [121] feasible.
- Visual Haskell programs can be executed by standard Haskell compilers and interpreters; all that is required is a translator from visual to textual syntax.

The remainder of this section describes the means by which the visual syntax is specified.

4.3.1 Visual elements

A Visual Haskell program is composed of many visual elements, arranged on paper or on a computer screen. The surface on which elements are arranged is called the *canvas* (from [103]). Any valid program fragment is a *pict* (short for "picture"). Picts are recursively composed of simpler picts, and ultimately of primitive picts—boxes, icons, lines, strings, and so on. There are two special kinds of primitive pict: *ports* and *arcs*. *Ports* serve as connection points between picts. Output ports (figure 4.9a) are triangular if they carry a non-function value, and rectangular if they carry a function. Input ports (figure 4.9b) are triangular in expressions, and semi-circular in patterns. *Arcs* join picts via their ports. There are two kinds of arcs: *data arcs* (figure 4.9d) are used in expressions, while *binding arcs* (figure 4.9e) are used in patterns.

There are three key ways of composing complex picts. Two picts are *attached* if they are located immediately adjacent to each other (figure 4.9f). Every port, for example, is attached to a pict of some kind. Picts are *connected* if there is an arc from a port of one to a port of another (figure 4.9d and e); the connected picts, the ports, and the arc together form a new pict. And a pict *contains* another if the second is wholly enclosed by the first (figure 4.9c). Nickerson [102] claims that these three constructions—which he calls *adjoinment*, *linkage*, and *containment*—account for the majority of diagrams used in computer science.

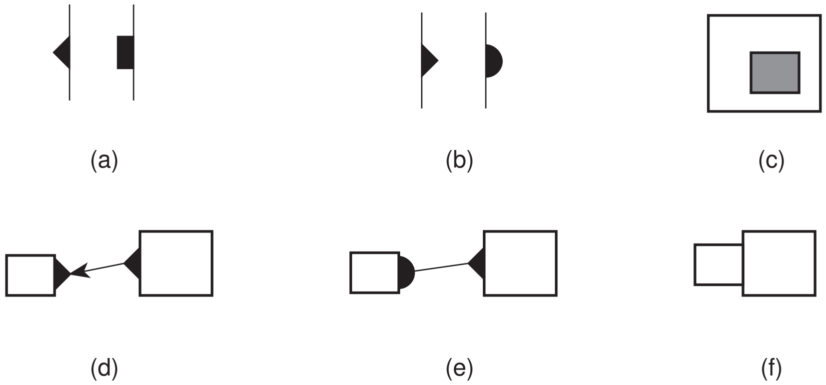


Figure 4.9. Simple visual elements: a) output ports; b) input ports; c) containment; d) data connection; e) pattern connection; f) adjoinment

4.3.2 Specifying the visual syntax

The syntax of Visual Haskell is specified as a translation from Haskell's abstract syntax to a concrete visual syntax. The abstract syntax used in this chapter is given in figure 4.10. As far as a syntax of Haskell goes, it is by no means complete, but it is enough for the purposes of this chapter. Some points to note:

– Sub-scripts distinguish unique occurrences of each production.

– Functions like $\mathtt{Tuple}_2$ and $\mathtt{Cons}_1$ are produced by a de-sugaring translation (section 4.3.4). In Haskell, these are equivalent to $(\_,\_)$ and $(:)$ respectively.

– A binary operator (op) enclosed in parentheses is treated as though it were a function identifier. The de-sugaring translation translates all operators into this form.

– The constant production (k) also includes floating-point literals, character literals, and string literals. String literals are treated as constants instead of as lists of characters.

– The *umatch* and *gmatch* productions represent Haskell's "match" phrases—see section 4.4.5.

– A semi-colon indicates concatenation. For example, the binding $d_1; d_2$ contains two bindings (each of which may itself be a series of bindings).

– Square brackets indicate an optional part. In the visual syntax, optional non-terminals are shown as though they were present; it is understood that visual non-terminals are actually present only if the textual ones are.

$$
\begin{array}{rcll}
v & \rightarrow & \text{(non-function identifier)} & \text{Values} \\
f & \rightarrow & \texttt{(}op\texttt{)} \mid \text{(function identifier)} & \text{Function names} \\
k & \rightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{[]} \mid \texttt{()} \mid \ldots & \text{Literal constants} \\
c & \rightarrow & \texttt{Tuple}_2 \mid \texttt{Cons}_1 \mid \ldots & \text{Data constructors} \\
op & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \mid \texttt{:+} \mid \texttt{:/} \mid \ldots & \text{Binary operators} \\
p & \rightarrow & v \mid k \mid c \mid f \mid p_1\ p_2 \mid v\texttt{@}p & \text{Patterns} \\
e & \rightarrow & v \mid k \mid c \mid f & \text{Expressions} \\
 & \mid & e_1\ e_2 & \\
 & \mid & e_1\ \texttt{.}\ e_2 & \\
 & \mid & \texttt{let}\ d\ \texttt{in}\ e & \\
 & \mid & \backslash\ umatch & \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ match_1; \ldots; match_k & \\
 & \mid & \texttt{if}\ g_1\ \texttt{then}\ e_1\ \texttt{else if}\ \cdots\ \texttt{else}\ e_k & \\
g & \rightarrow & e & \text{Boolean guards} \\
d & \rightarrow & d_1; d_2 & \text{Bindings} \\
 & \mid & p\ \texttt{=}\ e & \\
 & \mid & p\ \texttt{=}\ e\ \texttt{where}\ d & \\
 & \mid & p\ \texttt{|}\ g_1\ \texttt{=}\ e_1; \cdots;\ \texttt{|}\ g_k\ \texttt{=}\ e_k\ \ [\texttt{where}\ d] & \\
 & \mid & f\ match_1; \ldots; f\ match_k & \\
match & \rightarrow & umatch \mid gmatch & \text{Match phrases}
\end{array}
$$

Figure 4.10. Haskell's abstract syntax

– The backslash used in the λ-abstraction is not in constant-width typeface. It *should* be, but I am unable to get LaTeXto produce it.

The translation from abstract syntax to a picture is broken into three passes:

1. A textual de-sugaring pass (section 4.3.4). This removes some of Haskell's syntactic sugar to reduce the number of special cases. For example, infix operators are changed to prefix functions, and special syntax for lists and tuples is changed into data constructors.

2. A translation from Haskell's abstract syntax into the core visual syntax (section 4.4). Each production is rewritten into a concrete visual syntax.

3. A selective "visual sugaring" translation (section 4.5.1), in which a picture in core syntax is modified to improve its appearance and layout.

The visual syntax is quite abstract—it does not specify details such as the exact position of picts on the canvas, but only the topological relationships between them. For example, it states that two picts are connected, but does not state their relative or absolute positions on the canvas. The exact layout and positioning of picts will, we shall assume, be decided by some other mechanism called the *layout manager*. The layout manager will be some form of automatic graph drawing or screen layout algorithm, assisted interactively by a user.

4.3.3 A simple visual language

To assist in the notation of and motivation behind the visual syntax and sugaring sections (sections 4.4 and 4.5.1), I will give an example of a very simple language. Consider the following expression language, which has only variables, and functions of arity 1:

$$\begin{array}{rcl} v & \rightarrow & \text{(non-function identifier)} \\ f & \rightarrow & \text{(function identifier)} \\ e & \rightarrow & v \mid f\ e \end{array}$$

In a positional grammar approach, the *visual* syntax of this language could be specified using primitive visual operations. Suppose that the primitives *vbox* and *fbox* describe particular kinds of boxes used for variables and functions. The operation *name* produces the displayable string representing its argument; *contains* and *connect* represent the containment and linkage relations. Then, we could write

$$\begin{array}{rcl} v & \rightarrow & \mathit{contains}(\mathit{vbox}, \mathit{name}(v)) \\ f & \rightarrow & \mathit{contains}(\mathit{fbox}, \mathit{name}(f)) \\ e & \rightarrow & v \mid \mathit{connect}(e, f) \end{array}$$

Although I could translate Haskell's abstract syntax into a grammar such as this, I have chosen a more direct approach—the result of the translation *is* a visual representation, not just a textual description of one. Figure 4.11a gives the rewrite rules for the simple language. The first two rules are straight-forward: variables are represented by a grey box with a single output port; functions by a larger white box with one input port and one output port. The name of an identifier enclosed in quotes is meta-syntax for the equivalent displayable string.

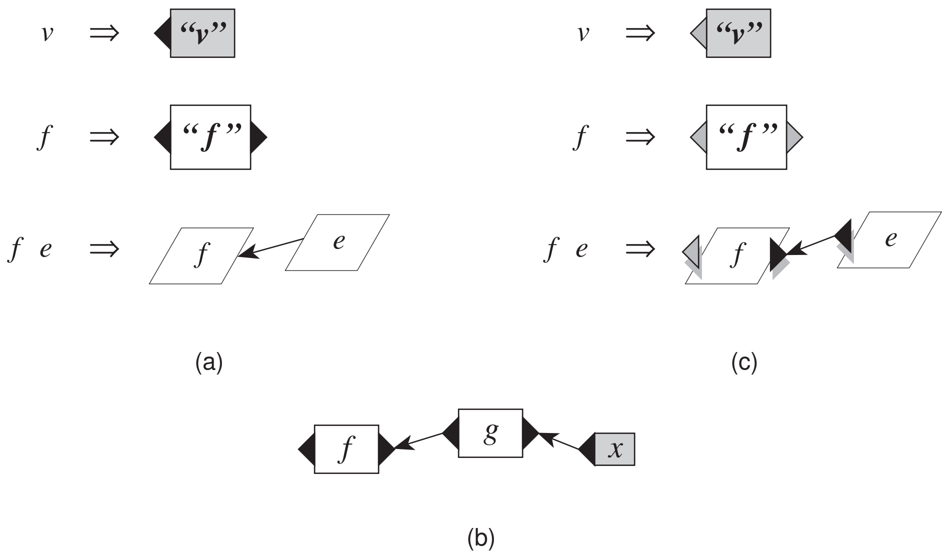


Figure 4.11. The visual syntax of a simple language: a) translation rules; b) a simple sentence; c) translation rules with port matching

Now, the third rule rewrites function application. A diamond-shaped graphic represents a *visual non-terminal*—that is, the visual equivalent of the production with which it is labelled. Every visual element other than non-terminals is *concrete visual syntax*—that is, elements that become part of the picture. In this case, the only such element is the arc that connects the output port to the input port. By recursively applying the rewrite rules, the complete picture is produced; figure 4.11b, for example, is the picture that represents the sentence $f\ (g\ x)$.

With currying and functions of more than one argument, this simple approach doesn't work. Two new concepts are needed: *live* and *dead* ports; and *matching* against already-existing ports. I will illustrate these concepts with a more complex visual syntax for the same language; functions with multiple arguments are left until section 4.4.1.

Figure 4.11c gives the translation rules with port matching. When a variable or function is encountered, it is created with *live* ports; these are coloured grey. Now, when two objects are connected, the connected ports are *killed*, and become black. The difficulty is that the ports already exist: if I draw them as black ports, then they will be new additions to the picture, and the picture will end up with grey ports *and* black ports. By matching against existing ports, we can avoid this difficulty. Consider the input port of e in the third rule of figure 4.11a: it is black, and partially obscures another grey port. In order to

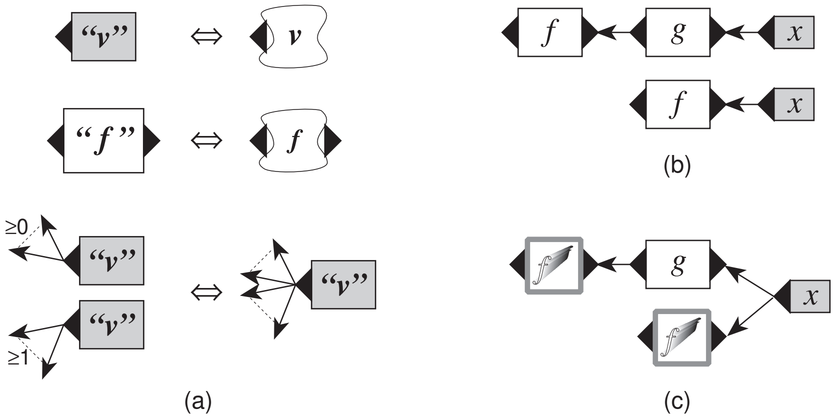


Figure 4.12. Sugaring the simple language: a) the sugaring rules; b) before sugaring; c) after sugaring

apply this rewrite rule, the obscured grey port must match a grey port on e. The result picture contains the obscuring port—that is, the black one. In contrast, the output port of f matches a live port, and remains live.

Note that the reason for port matching is to make it easier to support curried functions, not merely to change the colour of connected ports! Although this seems complex at first, it has removed the need to translate Haskell into an intermediate language, as in an earlier paper [117].

Having translated a program into a picture, it is unlikely that this picture will be as appealing as it could be. Icons, for example, are a popular ingredient in visual interfaces. We might also seek to remove redundancies left over from textual notation: multiple boxes for the same variable, for example. These changes to the picture do not affect its meaning, but are a kind of visual "syntactic sugar." This kind of modification is expressed with a set of bi-directional rewrite rules. Figure 4.12a shows a suitable set for the simple language. The first two rules allow a variable or function box to be replaced with an icon—the squiggly graphic is meta-syntax for the icon representing the object with which it is labelled. The third rule allows a variable box to be "shared" between applications.[1] In all rules, the left production is produced by the translation into core syntax, and the right is the alternative form. All rules are reversible, hence the double-ended arrows.

A sugaring rewrite is allowed only if the result is acceptable to the layout manager. If,

[1] To be pedantic, the language as given is too simple to even allow multiple variable occurrences. Assume that there are other productions that I have omitted.

$$e_1 \texttt{:} \cdots \texttt{:} e_n \texttt{:} e \Rightarrow \texttt{Cons}_\texttt{n}\ e_1 \cdots e_n\ e \tag{4.1}$$
$$\texttt{(}\ e_1, \ldots, e_n\ \texttt{)} \Rightarrow \texttt{Tuple}_\texttt{n}\ e_1 \cdots e_n \tag{4.2}$$
$$\texttt{[}\ e_1, \ldots, e_n\ \texttt{]} \Rightarrow \texttt{List}_\texttt{n}\ e_1 \cdots e_n \tag{4.3}$$
$$e_1 \,\grave{}\, f \,\grave{}\, e_2 \Rightarrow f\ e_1\ e_2 \tag{4.4}$$
$$e_1\ op\ e_2 \Rightarrow \texttt{(}\ op\ \texttt{)}\ e_1\ e_2 \tag{4.5}$$
$$\texttt{(}\ e\ op\ \texttt{)} \Rightarrow \texttt{(}\ op\ \texttt{)}\ e \tag{4.6}$$
$$\texttt{(}\ op\ e\ \texttt{)} \Rightarrow \texttt{(}\ op\ \texttt{)}^{[2,1]}\ e \tag{4.7}$$
$$\texttt{flip}\ f \Rightarrow f^{[2,1]} \tag{4.8}$$
$$\texttt{-}e \Rightarrow \texttt{negate}\ e \tag{4.9}$$

Figure 4.13. De-sugaring rules

for example, there is no icon defined for a particular function, then the layout manager will simply avoid that particular transformation; if merging two variable boxes causes an arc to cross another, then the merge will be disallowed. In other words, the sugaring rules are used selectively and intelligently. To illustrate, figure 4.12b shows a fragment of a visual program in core syntax; figure 4.12c shows how it might look after visual sugaring.

4.3.4 De-sugaring

The first phase of the translation removes some of Haskell's syntax for lists and tuples. The rewrite rules of figure 4.13 accomplish this. The amount of desugaring is minimal. The first three rules (equations 4.1 to 4.3) recognise a series of comma- or cons-operators and combine them into a single constructor; this allows the use of more meaningful icons. Equation 4.4 removes infix operators formed by back-quoting a function. Equations 4.5 to 4.7 translate binary operators and sections. Equations 4.7 and 4.8 produce a "permuted function"—see page 106. Equation 4.9 translates the only unary operator, negation.

4.4 The core syntax

The core syntax of Visual Haskell is specified by a set of rewrite rules from Haskell's abstract syntax directly into concrete visual syntax, in the style of section 4.3.3. The use of ports is more complex, and this is explained in section 4.4.1. The rules are given in five groups: simple expressions, structured expressions, patterns, bindings, and "match phrases," in that order. Note that the recursive re-application of rules implicitly selects the correct rule according to context—that is, the rule applied for an occurrence of say v is selected according to whether v occurs in an expression or a pattern.

4.4.1 Simple expressions

This section gives the syntax of simple expressions—that is, expressions that do not use containment. This small subset of Haskell is approximately equivalent to the language features of pure dataflow languages. Figure 4.14 gives the rewrite rules for these expressions: variables, constants, data constructors, function application, and function composition.

Variables and constants (figure 4.14a and b) are the same as in figure 4.11c. Function variables and data constructors (figure 4.14c and d) are also similar to figure 4.11c, but have n input ports, all live. n is the arity of the constructor or function; from here on, n will always mean the arity of the whole production under consideration. Unlike the earlier examples, however, the output port is rectangular, indicating that it is function-valued. Visual sugaring can be used to replace the boxes used here with icons—see section 4.5.1.

Recall that enclosing an object's identifier in double quotes is meta-syntax for its *name* as a displayable string. The name of a variable or function is its identifier. There are some special names: the name of a parenthesised operator is the operator without parentheses—for example, the name of `(+)` is "+"; the names of the constants `[]` and `()` are "[ ]" and "( )"; the name of the wildcard (_) is a blank string.

Application of a function-valued expression to an argument connects the corresponding picts (figure 4.14e). Because a function-valued expression can have more than one live input port, port matching is used to select the set of live ports. This is indicated by the dashed line between the input ports of e_1. The top-most live port is connected to the argument pict, and is killed, changing its colour to black. This is how currying is handled in the visual syntax: each time an additional argument is supplied to a function, the next live port is connected and killed; eventually, if all arguments are supplied, the function box has no remaining live ports.

Port matching is also used on the output ports of e_1 and e_2. The output port of e_2 matches a live port, which can be either triangular or rectangular, as indicated by the superposition of the two shapes. The port is killed. The output port of e_1 matches a

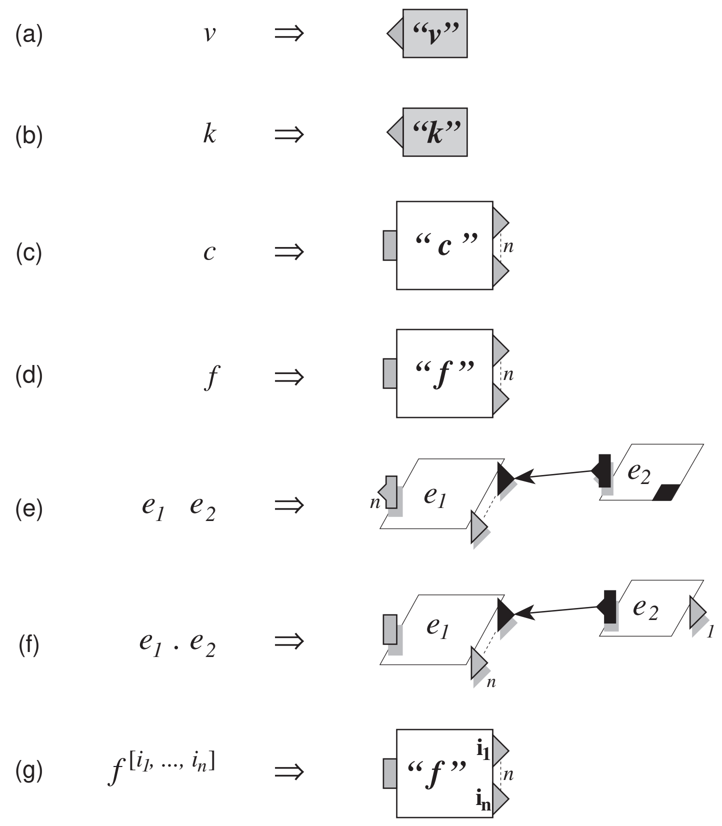


Figure 4.14. Visual syntax: simple expressions

rectangular port, since e_1 must be a function. The result port is annotated with a small n: this means that it is a triangle if n is zero, or a rectangle if n is greater than zero. Thus, if the newly connected port of e_1 was the last one—that is, the function is applied to all its arguments—the port changes from a rectangle to a triangle.

Once an expression has been used as an argument, it cannot have any live input ports. The black diamond in the lower corner of the e_2 non-terminal means that any remaining live ports are killed.

Function composition (figure 4.14f) is similar to function application. Because e_1 and e_2 must both be functions, the match shapes for the output ports are both rectangular. The main point to note is this: the first live port of the result pict is the port attached to e_2, not to e_1 as for function application. This is indicated by the "1" annotation on e_2's input port. If e_1 has more than one live port, then all but the first also remain live, and will be connected after the port attached to e_1. (Usually, though, e_1 will have no remaining live ports.)

In a visual language, it is easy to apply arguments in arbitrary order. To express this, I use the special syntax $f^{[i_1,\ldots,i_n]}$ to denote a function variable to which arguments are supplied in a different order. The j'th argument to $f^{[i_1,\ldots,i_n]}$ is the i_j'th argument to f. This I call "permuted application"; permuted functions are produced by some of the de-sugaring rules (section 4.3.4). The visual syntax of a permuted function variable is similar to that of a normal function, but with port j annotated with i_j. These numbers are concrete syntax—that is, they appear on the canvas. In a graphical development environment, permuted application would, I expect, be used more than in standard Haskell, because the visual representation does not impose an order of arguments as a textual representation does.

An example will clarify how function application and composition works. Consider the expression

```
((+) x . negate) y
```

Figure 4.15a shows its visual terminals; figure 4.15b show three steps in the translation, from top to bottom. In the first, $(+)$ is applied to x: the connected ports are killed. In the second step, this pict is composed with *negate*: the second port of $(+)$ is matched, and connected to the output port of *negate*. Note that the input port of *negate* remains live. Finally, the pict just produced is applied to y: the connected ports are killed, as expected. Note also that the output port of $(+)$ changes to a triangle, since the last input port of the whole pict has just been connected.

The alert reader may have noticed a source of ambiguity here: the picture at the bottom of figure 4.15b could also represent `x + (negate y)`.[2] The ambiguity arises because composition does not retain any visual grouping construct. A visual editor could perhaps best resolve the ambiguity by retaining the composition in its internal data structure; at the user's request, it would display the grouping as a containing box, and allow it to be removed or altered.

[2]This is a useful distinction only if Visual Haskell is intended to provide a complete syntactic equivalent to Haskell. In the original development of Visual Haskell, the idea of a "two-view" development system featured very heavily; for such a system, the difference between `((+) x . negate) y` and `x + (negate y)` is significant. In future development of Visual Haskell, I expect that this exact correspondence will not be necessary.

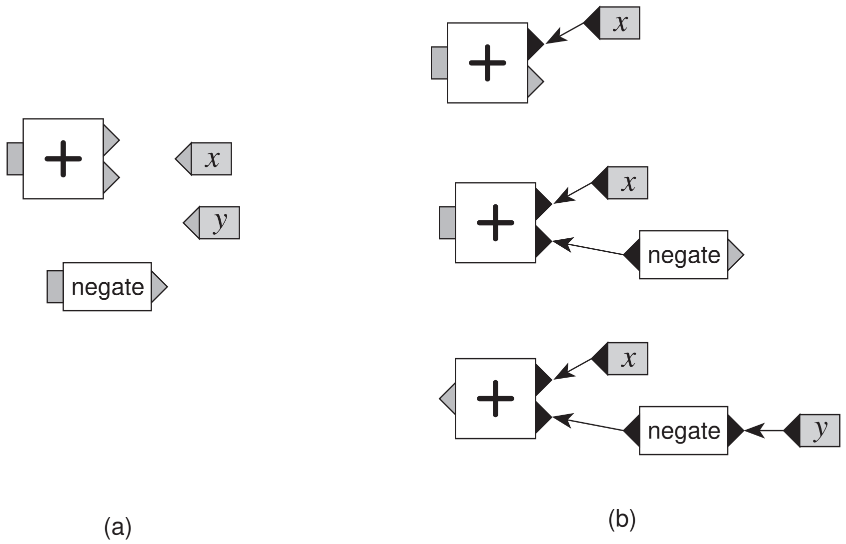


Figure 4.15. An example translation: `((+) x . negate) y`: a) visual terminals; b) three steps in the translation

4.4.2 Structured expressions

With the aid of some simple conditional actors such as *select* (page 56), it is possible to encode loops and other constructs such as conditional execution in a pure dataflow framework [82]. It is also possible to analyse a graph to extract the constructs encoded therein [27]. Nonetheless, Kodosky *et al* [82] argue that the dataflow paradigm should be extended to incorporate structured constructs. They add loops, a case construct, and a sequencing construct to the Labview language, calling the result "structured dataflow." General-purpose visual dataflow languages generally provide such constructs—see Hils' survey [59]. Special-purpose languages such as that of Ptolemy [88] do not, relying instead on the lower-level host language.

Haskell does not have explicit loops, since it uses recursion instead of iteration. Visual Haskell does, however, have several structures that resemble structured dataflow: *let*-expressions, conditionals (*if-then-else*), λ-abstractions, and *case*-expressions. Each of these is delimited by a containing box, or *enclosure*, and can be connected on the outside as any other expression can. A novel feature arises because Haskell is higher-order: if a structured expression is function-valued, it has one or more live input ports, and can thus be connected in the same way as a function variable. This is an improvement over the language of my earlier paper [117], which required explicit *apply* nodes for all structures

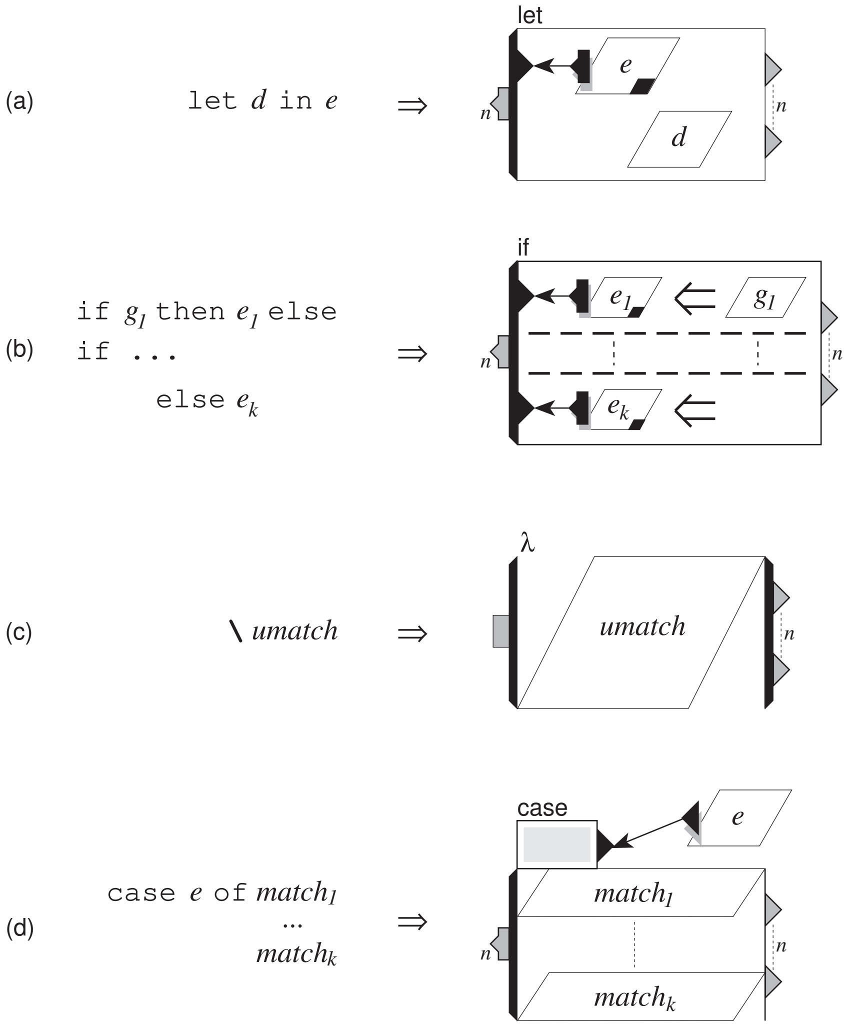


Figure 4.16. Visual syntax: structured expressions

except λ-abstractions.

The visual syntax of a *let*-expression is essentially a containing box labelled "let" (figure 4.16a), containing the expression result e and local bindings d. e is connected to a dummy port on the inside of the enclosure. If the whole expression has non-zero arity n, then the box has n live input ports attached to it. The output port is rectangular if n is non-zero, otherwise triangular. In the visual sugaring phase, sharing can be used to connect variables bound in d to uses of that variable in e. The area of the canvas over which sharing can occur is called a *region*; in this case, it is the entire inside of the box. To illustrate, figure 4.17a shows the expression

```
let t = w * x1 in (x0+t, x0-t)
```

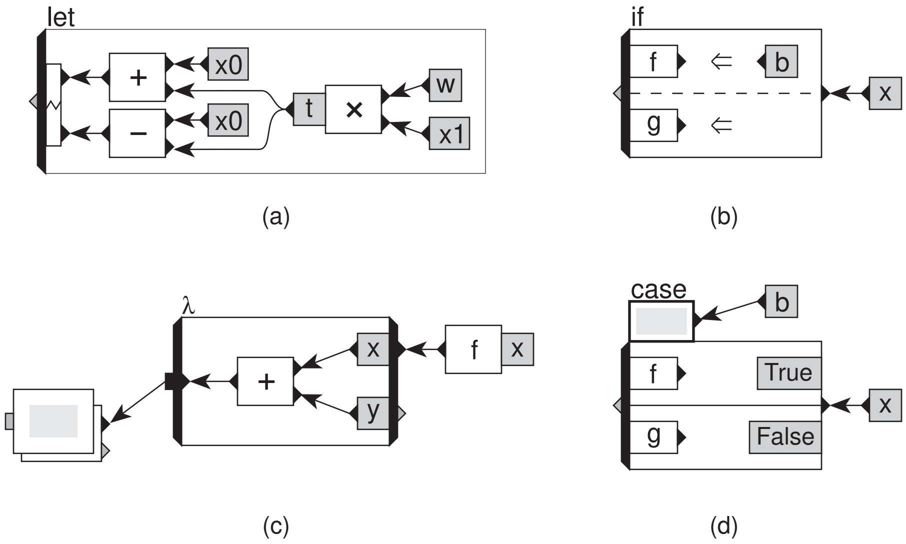


Figure 4.17. Examples of structured expressions (see text)

A series of conditional (*if-then-else*) expressions is grouped into a single visual construct (figure 4.16b). (Translation into *case*-expressions, as in the Haskell report [44], would be too cumbersome visually.) Each guard g_i and its consequent e_i are placed into one horizontal section of the containing box, but separated by the heavy arrow glyph. Sections are separated by a heavy dashed line. g_i and e_i are each in their own region—that is, there can be no sharing between them. There is no guard g_k, so its region is left entirely blank. As for *let*-expressions, a series of conditional expressions has one or more input ports if it is function-valued. To illustrates, figure 4.17b is a function-valued conditional expression applied to a variable:

```
(if b then f else g) x
```

λ-abstractions (figure 4.16c) are also a containing box; this time, however, the syntax of the box and its contents are delegated to another production, called *umatch* (see section 4.4.5). *umatch* is similar to the enclosing box of *let*-expressions, but contains one or more patterns for the λ-abstraction's arguments. Figure 4.17c includes a λ-abstraction:

```
map ((\x y -> x + y) (f x))
```

A *case*-expression (figure 4.16d) uses pattern-matching to select an expression from several alternatives. Each of these alternatives is a *match* clause (see section 4.4.5); they are "stacked" one above another in the order in which they are tested. The case value e is connected to a special input port above the stacked match expressions; the grey rectangle is an argument slot (section 4.5.1), so e can be placed within the *case* if the layout manager will allow it. Again, the shape of the output port depends on n, as does the number (if any) of input ports. Figure 4.17d is a function-valued *case*-expression applied to an argument:

```
(case b of
    True  -> f
    False -> g) x
```

Although I argued that these constructs have a precedent in structured dataflow, they are more general than structured dataflow. A structured dataflow construct behaves on the outside just like a dataflow actor: all of the data on which it operates comes in through input ports. Because Visual Haskell has scope and named variables, however, this is not necessarily true for the constructs just described. A λ-abstraction, for example, fulfils this condition only if it has no free variables. The λ-abstraction in figure 4.17c is like this; for comparison, the λ-abstraction `(\x -> x + y)` is not, since y occurs free.

A similar idea applies to the other constructs: to conform to the dataflow model, they must not use data not supplied to them explicitly. The conditional in figure 4.17b, for example, uses b even though b is not supplied to the construct by a dataflow arc. Compare this with the Labview *case*, which selects one of several *functions*, and then applies that to an argument [82]. The Haskell *case* can be used this way, as in figure 4.17d, but need not be. In this sense, then, the Visual Haskell *case* is more flexible. For λ-abstractions, a transformation known as *lambda-lifting* can be used to eliminate free variables [106]; it may be possible to find a similar transformation for *case*-expressions.

4.4.3 Patterns

Patterns "de-construct" arguments, and bind names to expressions (or parts of expressions). Haskell's pattern syntax is essentially a subset of its expression syntax; visually, patterns look like expressions connected "backwards." Every pattern has one input port and zero or more output ports; patterns are the only kind of object which can have more than one output port. Patterns are not common in visual languages, although Enhanced Show-and-Tell (ESTL) [99] supports a very similar facility based on *inconsistency*.

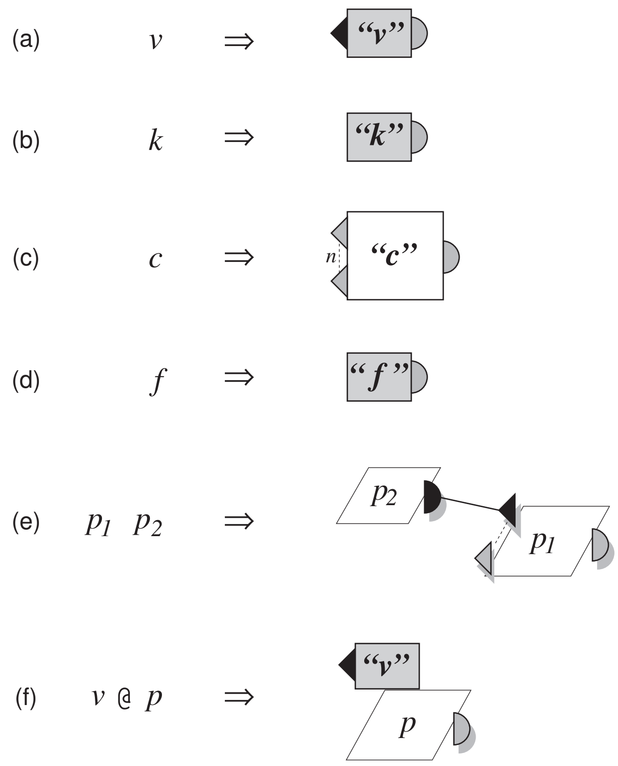


Figure 4.18. Visual syntax: patterns

Figure 4.18 gives the rewrite rules for patterns. Variables and constants are drawn as grey boxes, with a single live semi-circular input port (figure 4.18a, b, and d). Non-function variables also have a single output port. This port is not live—no patterns can be attached to it—but the port can still be used for sharing, since sharing does not require live ports (section 4.5.1). Data constructors are drawn as a white box with one input port and one or more live output ports (figure 4.18c). As for expressions, this box can be replaced by an icon in the visual sugaring (section 4.5.1).

Nested patterns are translated by figure 4.18e. Live output ports of a constructor are connected and killed in order, in the same way in which function application connects and kills live input ports. For example, the pattern `(:+)` x y firstly connects the first output port of `(:+)` to the input port of x; then the second output port to y. Note that some common nested patterns are removed by the desugaring translation (section 4.3.4). For example, $(x : y : ys)$ is translated into $\texttt{Cons}_2\ x\ y\ ys$.

Finally, *as*-patterns are drawn as the pattern with a variable box attached (figure 4.18f). The variable box has a single output port, which can be used for sharing.

4.4.4 Bindings

A *binding* binds a name to a value. Many visual dataflow languages only allow binding of names to functions (*function* bindings), and thus avoid the problem of free variables. Visual Haskell also allows names to be bound to values (*pattern* bindings).

A series of bindings is displayed in the same region (figure 4.19a). As noted earlier, the visual syntax does not specify the relative positions of bindings. In core syntax, the two bindings are not connected; with sharing, pattern bindings can be connected, and thus inter-mingled within the region. At the module level, the layout manager should place bindings one above the other. At other levels, it should place function bindings one above the other, and pattern bindings clustered together so that sharing can be used effectively.

The simplest kind of pattern binding has its value e connected to its pattern p (figure 4.19b). Any live input ports of e are killed. If a simple pattern binding has local definitions, its right-hand side is put into a containing box (figure 4.19c). The appearance is very like a *let*-expression, but without the "let" label, and with no input ports regardless of its arity.

A *guarded* pattern binding has the form p `|` g_1 `=` e_1`;`$\cdots$`;` `|` g_k `=` e_k [`where` d] (figure 4.19d). The visual syntax of the right-hand side is very similar to that for conditionals (figure 4.16b), with the addition of some bindings d. d is placed in its own region.

The final rewrite rule is for function bindings (figure 4.19e). Within the region in which it is defined, a function binding is shown as a small rounded rectangle containing the function's name, and with an appropriate number of dead input ports and a dead output port. For editing, the function definition is displayed in its own window: the title bar contains the function's name and its icon (if it has one). The body of the window is a stack of *match* phrases.

4.4.5 Match phrases

A match phrase, in Haskell, is a series of patterns, guards, and result expressions used in *case*-expressions and function bindings. There are two types: those without guards, and those with. These are represented in Visual Haskell by the productions *umatch* and *gmatch* respectively; the production *match* can be either of these (figure 4.10).

Figure 4.20a shows the textual and visual syntax of *unguarded* match phrases. Textu-

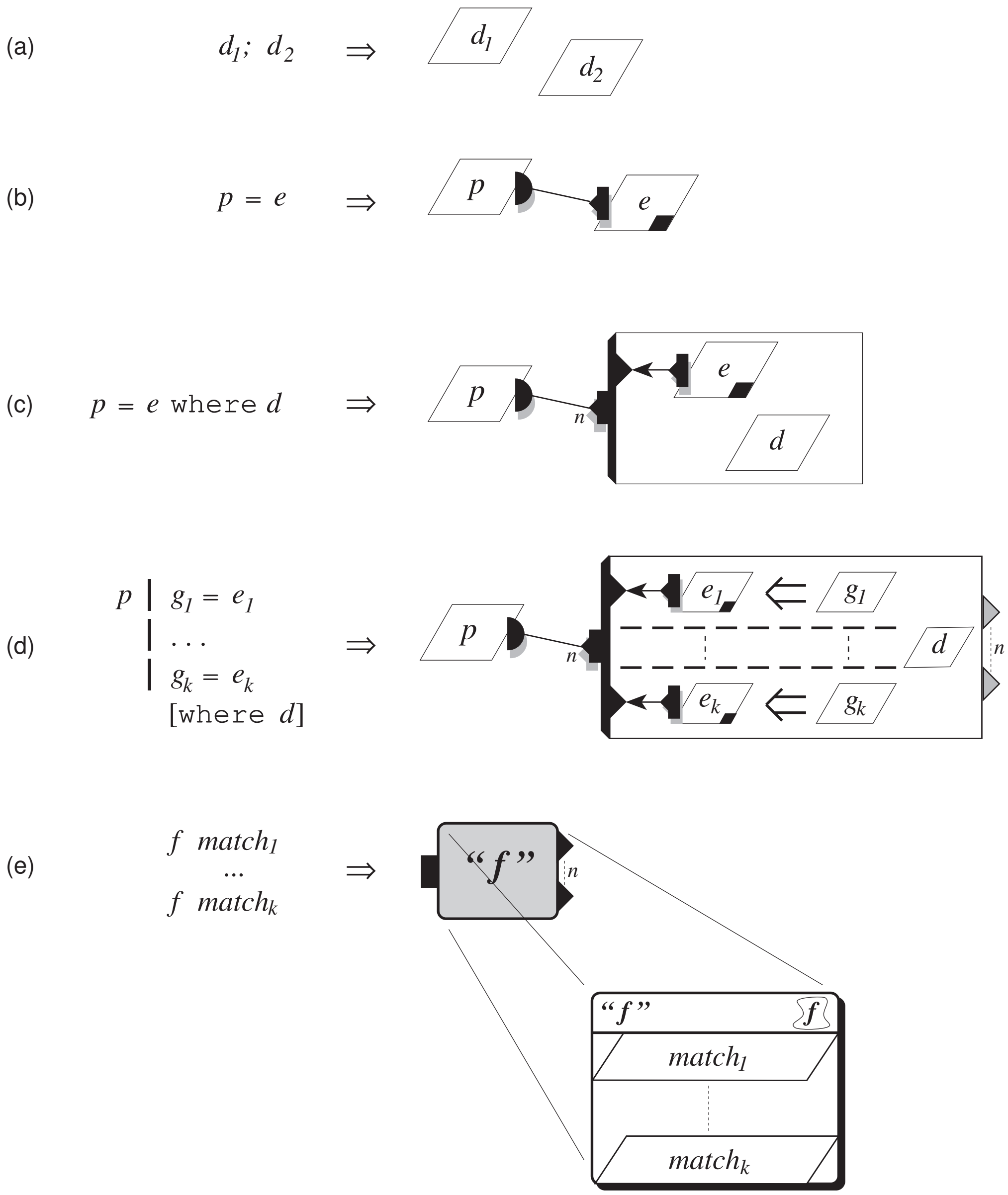
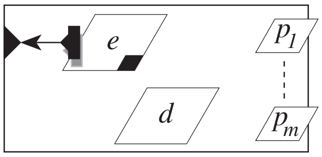


Figure 4.19. Visual syntax: bindings

$$
\begin{array}{llll}
\text{(a)} & \mathit{umatch} & \rightarrow & p_1 \cdots p_m \;\; \texttt{->} \;\; \texttt{let}\ d\ \texttt{in}\ e \\
& & | & p_1 \cdots p_m \;\; \texttt{->} \;\; e\ [\texttt{where}\ d] \\
& & | & p_1 \cdots p_m \;\; \texttt{=} \;\; \texttt{let}\ d\ \texttt{in}\ e \\
& & | & p_1 \cdots p_m \;\; \texttt{=} \;\; e\ [\texttt{where}\ d]
\end{array}
$$

$\Rightarrow$



$$
\begin{array}{llll}
\text{(b)} & \mathit{gmatch} & \rightarrow & p_1 \cdots p_m \;\; \texttt{|}\ g_1\ \texttt{->}\ e_1; \cdots ;\ \texttt{|}\ g_k\ \texttt{->}\ e_k \;\; [\texttt{where}\ d] \\
& & | & p_1 \cdots p_m \;\; \texttt{|}\ g_1\ \texttt{=}\ e_1; \cdots ;\ \texttt{|}\ g_k\ \texttt{=}\ e_k \;\; [\texttt{where}\ d]
\end{array}
$$

$\Rightarrow$

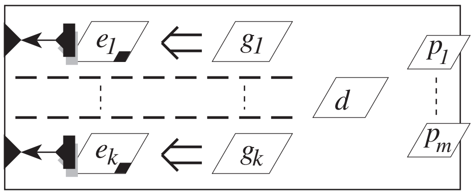


Figure 4.20. Visual syntax: match clauses

ally, an unguarded match is a set of patterns $p_1 \ldots p_m$, a delimiter `->` or `=`, an expression e, and an optional *where* clause containing local definitions d. The difference between the clauses with delimiter `->` or `=` is the context in which they occur: the former is used within *case*-expressions, while the latter is used in function bindings. Although Haskell distinguishes the two cases syntactically, Visual Haskell does not.

Visually, an unguarded match is similar to a *let*-expression, with the addition of the set of patterns. The expression, bindings (if they exist), and the patterns are all in the same region.[3]

A second type of unguarded match is also given in figure 4.20a; these contain a *let*-expression at the top level and have no auxiliary definitions. This simplifies the common case in which a *let*-expression occurs at the top level, as in $f\ p_1\ \ldots\ p_2$ `=` `let` d `in` e and $\backslash\ p_1\ \ldots\ p_2$ `->` `let` d `in` e; visually, the effect is to remove a redundant containing box.

Figure 4.20b shows the syntax of *guarded* match phrases. Textually, a guarded match is a set of patterns $p_1 \ldots p_m$, a delimiter `->` or `=`, a series of guards g_i and consequents e_i, and an optional set of bindings d. Visually, a guarded match is similar to a series

[3] d can in fact redefine variables in $p_1 \ldots p_m$—although Haskell has no problem with this, there should perhaps be a way of "crossing out" the over-ridden variables in $p_1 \ldots p_m$.

of conditionals, but with the addition of the bindings and patterns. The bindings and patterns are within the same region; as for conditionals, each g_i and e_i is isolated in its own region. Although not shown in the figure, the visual syntax for an *otherwise* guard is just blank space.

4.5 Improving the visual syntax

The core syntax can be enhanced in several ways. In this section, I will firstly give sugaring rules in style of section 4.3.3. These rules improve the appearance of Visual Haskell programs considerably. They do not, however, help with one of the key reasons for using a visual notation: the handle they give on *structural* properties of a program. For example, many of the examples in chapter 5 illustrate arrays of functions connected in various ways. To assist with this kind of program visualisation, more complex visual representations are needed, and I give three that I use in this thesis.

4.5.1 Visual syntactic sugar

Figure 4.21 lists the sugaring rules. As explained in section 4.3.3, the layout manager permits a sugaring rewrite only if the result pict can be sensibly laid out on the canvas. There are four sets of rules: for icons, variable sharing, attachment, and argument slots.

Icons A critical ingredient in any visual language is the use of meaningful symbols. Visual Haskell—in common with many other visual languages—optionally uses icons to represent application of certain functions; it also allows the use of icons for variables, constants, and data constructors. Figure 4.21a gives the sugaring rules for icons. (Recall that a quoted production is visual meta-syntax for a displayable string, and a squiggly box is visual meta-syntax for an icon.) This icons I use for standard prelude data constructors and functions were shown in figures 4.6 and 4.7; more icons for special cases will be introduced in following chapters where necessary.

Sharing In the core visual syntax, each occurrence of a variable in a textual Haskell program produces a unique pict on the canvas. If layout permits, these picts can be *shared*—that is, have more than one arc connected to them. There are two cases captured by the rewrite rules of figure 4.21b: in the first, the arcs from a variable box

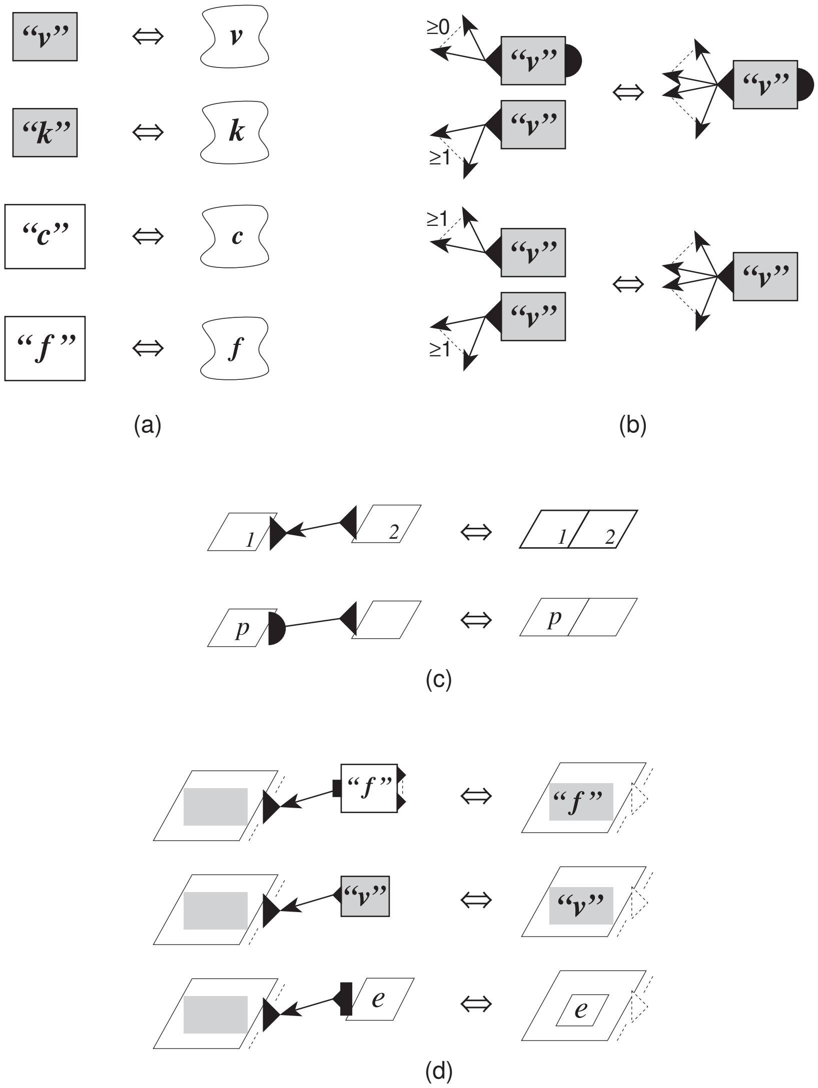


Figure 4.21. Sugaring rules: a) icons; b) sharing; c) attachment; d) argument slots

can be re-connected to a pattern variable box of the same name and with zero or more output arcs; the variable box is deleted from the canvas. In the second, the arcs from a variable box can be re-connected to another variable box of the same name with one or more output arcs; the first box is deleted from the canvas.

Attachment Picts can be attached instead of connected (figure 4.21c). Again, there are two rules. In the first, any two picts connected by a data arc can be attached; the arc and the two ports disappear from the canvas. The figure uses a blank diamond to denote *any* visual syntax; a numeral in the bottom-right corner distinguishes unique

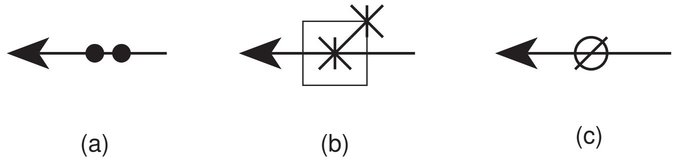


Figure 4.22. Illustrating type annotations: a) `(a,b)`; b) `[[a]]`; c) `Stream` (`Vector` α)

occurrences. In the second rule, any pict and a pattern connected by a binding arc can be attached.

Argument slots Enhanced Show-and-Tell (ESTL) [99] and DataVis [58] have *function slots*. A function slot is a position in the icon of a higher-order function in which a function argument can be placed. Visual Haskell has a generalised form of this idea, which I call *argument slots*: an icon can have slots for one or more of its arguments—whether functions or not. If layout permits, the argument connected to the corresponding input port is drawn in the argument slot—that is, within the icon.

There are three rules for argument slots. In the first, a function argument to a higher-order function has its *name* placed into the slot. This is used frequently for higher-order functions, such as the application of *map* in figure 4.4b. The second rule is similar but applies to variables. The third allows any arbitrary expression to be placed within an argument slot. This allows function icons to be placed into the slot; even complex expressions can be placed into the slot if layout permits. In all three rules, the port corresponding to the argument disappears.

Type annotations I use an informal system of annotations on arcs to indicate their types. Figure 4.22 illustrates a few annotated arcs: type variables (such as α) are represented by black circles; tupling by placing tuple element annotations side-by-side on an arc; lists by an asterisk-like annotation; streams (section 5.3) by an open circle; vectors (section 5.2) by a slash. If necessary, a type annotation is enclosed by a box, which can in turn be annotated.

4.5.2 Iteration boxes

Most general-purpose visual programming languages support iteration in some form or another. All of the visual dataflow languages in Hils' survey [59], for example, support

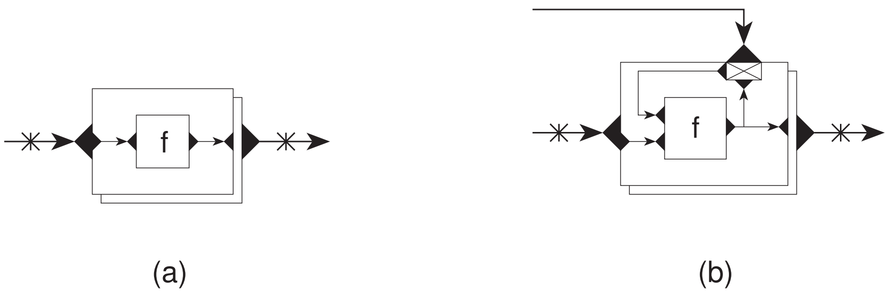


Figure 4.23. Illustrating iteration: a) *map*; b) *scanl*

iteration. One technique is the use of special iteration constructs; Labview, for example, incorporates *for*-loops and *while*-loops into its structured dataflow model in this way [82]. Each is a containing box with an internal dataflow graph, which executes each time through the loop. Other languages use cycles and special-purpose dataflow nodes; in Khoros, for example, the programmer places a LOOP glyph at the start of a section of flow-graph to be iterated, and a feedback connection to this glyph from the end of the flow-graph section [155].

Pure functional languages rely on recursion instead of iteration. In a higher-order language such as Haskell, patterns of iteration are captured by higher-order functions; applying the higher-order function is equivalent to coding an iterative loop in other languages (see sections 5.2.2 and 5.3.4). Visual Haskell, along with the visual functional language VisaVis [110], thus uses a function icon to represent iteration.

There is, however, a limit to how much information one icon can convey, and so in previous work [114] I have used a representation similar to structured dataflow iteration constructs. Two examples are given in figure 4.23. In this representation, the computation performed on each iteration is shown as a dataflow graph inside a generic icon representing the type of iteration. Each input to this graph is one element of an input aggregate (list, stream, or vector); each output is one element of an output aggregate.

For lists, the generic icon is a white-shadowed box, as used by the icon for *map*. In *map f* (figure 4.23a), for example, the internal graph is a function application—that is, *f* is applied to an input element to produce an output element. If the function maintains a "state" value between iteration, this is shown as a crossed box. In *scanl f* (figure 4.23b), for example, the state is one input to each iteration; it is also updated with the output element. Figure 5.22 shows a more complex example of this representation.

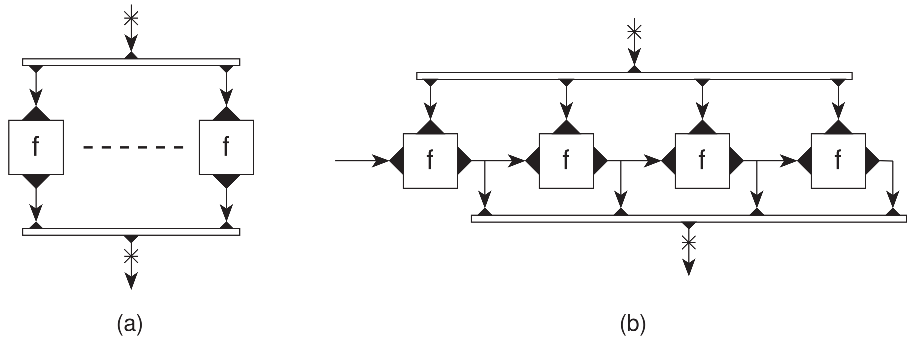


Figure 4.24. Unfolded higher-order functions: a) *map*; b) *scanl*

4.5.3 Unfolded higher-order functions

Higher-order functions combine functions, and are thus often more usefully thought of—in a visual language—as a means of constructing program graphs. For example, I gave in figure 4.14f a special syntax for function composition, which combines two function boxes in a meaningful way. Without this syntax, `f . g` would be represented as a box labelled "." connected to function-variable boxes labelled f and g.

Higher-order functions such as *map* and *scanl* can be thought of as representing an "array" of function applications—one to each element of a list. Figure 4.24a illustrates *map* shown in a manner that conveys this idea. The two narrow horizontal boxes represent deconstruction of a list into individual elements, and construction of a list from its elements. The dashed line symbolises elided function application boxes.

A second style of representation unfolds a higher-order function assuming certain properties of its argument. For example, figure 4.24b shows *scanl* as it would appear if its argument list contained four elements. Both of these styles show programs using higher-order functions in a very "structural" way; sections 5.4 and 5.5 use unfolded representations extensively.

I am unaware of any proposal to incorporate this kind of representation into a visual language. Although Lauwereins *et al* [85] use unfolded diagrams, they do not appear to be part of their visual interface yet. In Visual Haskell, these representations are still informal. To incorporate them into a visual interface, one could perhaps mark a list (or other aggregate data type) for "expansion." These unfolded representations would then be a valid sugaring rewrite rule.

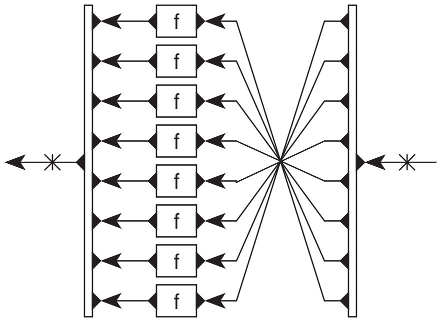


Figure 4.25. Wiring: *map* $f \cdot$ *reverse*

4.5.4 Wiring

A number of functions on lists re-arrange elements of their argument list. When used in conjunction with unfolded higher-order functions, it is helpful to draw these functions as "wiring." Figure 4.25 illustrates the expression *map* $f \cdot$ *reverse*. The long rectangles on the right and left represent list construction and de-construction respectively. *reverse* is shown as a mass of wires, and *map* f is shown unfolded.

The need for wiring is more apparent with the vector operations of section 5.2. Figure 5.5 gives example of additional wiring functions, while figure 5.9 uses them in the definition of the Fast Fourier Transform.

4.6 Summary

There is, I believe, significant value in a precise visual syntax for Haskell that can replace the *ad-hoc* notations that are sometimes used. This chapter presented such a syntax for a substantial portion of Haskell. The missing syntax will need to be provided to make Visual Haskell complete: apart from type declarations, syntax is needed for user-defined operators, modules, list comprehensions, and array comprehensions. In addition, a way of specifying more sophisticated syntax, as used in sections 4.5.2 to 4.5.4, is also needed. One approach I have explored is to translate abstract syntax into a low-level picture language like that used in picture layout grammars. This approach turned out to be too cumbersome for the core syntax, but may be acceptable if limited to special purposes.

Although I have used Visual Haskell mainly for program visualisation so far, the idea of a "two-view" development system is intriguing. In a two-view system, the programmer

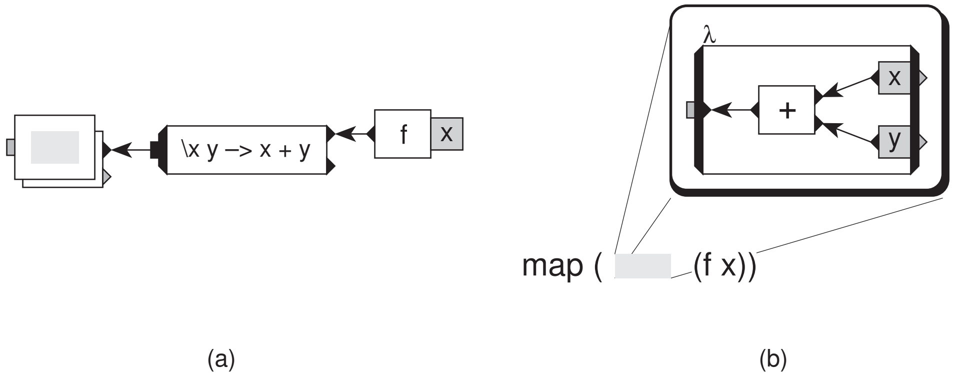


Figure 4.26. Mixing text and pictures: a) text-in-picture; b) picture-in-text

can switch between text and pictures at arbitrary levels of a program. For example, a picture can contain an expression in a text box, while a text window can contain a picture. Figure 4.26 illustrates the idea with two alternative views of the expression `map ((\x y -> x + y) (f x))` (illustrated as a complete picture in figure 4.17c).

Chapter 5

Static Process Networks

Real-time signal processing programs are oriented around two key aggregate data-types: arrays, which I will call vectors, and streams. Their base elements are almost invariably numbers. Streams are fundamental to a real-time signal processing program, even if they only exist as a data structure at the interface to external signals. Vectors are inherent in signal processing algorithms that operate on segments of a signal, such as transforms.

Digital signal processors are designed to operate most efficiently in loops, and it is therefore important that a high-level programming language be able to express iteration through vectors and streams clearly. The first section of this chapter develops a *Vector* data-type and associated functions, which capture the key operations on vectors. An example, the Fast Fourier Transform, illustrates their use.

Because Haskell is lazy, it is easy to write infinite data structures. Functional operating systems and process networks (sections 2.3.5 and 2.3.6) use lazy lists to model communications channels between processes. Here, I will use an explicitly-defined *Stream* data-type to model communication, and give six Haskell functions that can be implemented using the canonical SDF actors of section 3.2.5.

With these six functions, we can use the features of Haskell to increase the expressiveness with which we write dataflow networks: some higher-order functions encapsulate common types of processes; other higher-order functions capture common interconnection patterns, such as serial and parallel connection; yet others represent various linear, mesh, and tree-structured interconnection patterns. The use of these language facilities for constructing process networks was explored by Kelly [78]; here, I extend this work to real-time streams and signal processing. This style of programming could perhaps be called "data-parallel process programming," and provides a counter to the position that functional parallelism does not provide adequate abstraction from individual threads of computation.

Because all functions on streams are defined in terms of the six primitive functions, the programmer does not define recursive functions on streams. This style of programming is advocated by Backus [11] and by Bird [20], because: i) non-recursive programs tend to be more concise than recursive ones; and ii) there is less chance of making errors, such as forgetting a termination condition.

The final task of this chapter is to adapt the algebraic style of program transformation to streams and processes. The result is a set of provably-correct program transformations that can be used to alter the structure and degree of parallelism present in a process network program. We can use these techniques to eliminate apparent efficiency defects caused by the use of very-high-level programming language, and to tailor a program to a given hardware configuration for better execution efficiency.

5.1 Related work

At present, most "real-world" DSP programming is in assembler or C. Block-diagram systems are perhaps the most popular alternative programming technology. One of the drawbacks of block-diagram systems is that they are generally oriented towards particular hardware systems, and thus fall short of a generally-applicable production tool. Willekens *et al* argue that block-diagrams alone are inadequate for specifying DSP systems, since they do not permit expression of control flow; detailed computation is best specified in a textual language, leaving the block diagram to higher levels of description [154].

Another approach uses light-weight operating system kernels and support libraries [104, 39, 143]. This is particular evident in parallel systems, where the added complexities of multi-processor communication and synchronisation seem to encourage adoption of multi-tasking kernels.

The approach in which I am interested here is to use a programming language, but at a much higher level than C. Silage [57] is perhaps the best example of a language designed specifically for DSP programming; recent work explores higher-level signal processing programming concepts [142]. It has also evolved into a commercial product, DFL [154].

The most noticeable feature of Silage is its support for streams of data and delays. The expression $x@1$ is the signal x delayed by one sample. Arithmetic operations extend point-wise to streams. Silage also supports fixed-point data types and arithmetic, an important aspect of programming real DSP devices.

Freericks and Knoll use recursive functions to define signal-processing functions [81]. Higher-order functions can also be used to capture particular patterns of recursion [48]. An unusual aspect of their language is an explicit *suspension* construct that supports data-driven programming; this is used to code interfaces to real-time I/O channels. They propose to use partial evaluation in their compiler to eliminate the potential run-time overhead of recursion.

A recurring theme in DSP programming is the use of *streams* to represent discrete-time signals. Streams were first proposed by Landin as a means of separating the control structure of Algol-60 loops from the loop body [84]. Landin represents a stream as a pair of the head element, and a nullary function representing the rest of the stream. Stream elements are thus "evaluated when they are come to"—calculation of each successive loop control value and execution of the loop body proceed in an interleaved manner.

Burge further developed the idea of streams, as a way of structuring programs as a set of independent sub-programs [28]. He uses the same representation as Landin, and lists a number of functions that are now standard in functional programming languages: *maps* applies a function to each element of a stream; *generate* $f\ x$ produces the sequence $x, f\ x, f^2\ x$, and so on; *zips* produces a stream of pairs from two streams; *filter* removes elements that fail to satisfy a predicate; and so on.

Wendelborn and Garsden compare a number of stream implementations [153], and point out the difference between channels and streams, noting that the terms overlap somewhat in usage. A channel is destructive, since elements are appended to the channel. Kahn process networks and the dataflow process model use channels, since new tokens are appended to them. Streams are functional: the stream producer is part of the stream itself. Landin and Burge, for example, use a function to represent the remainder of the stream; lazy functional languages implement streams as recursively-defined, lazily-evaluated lists.

The ease with which lazily-evaluated streams lend themselves to signal processing has also been noted by Globirsch [52]. He uses lazy lists to simulate signals, and gives a number of simple filters in Haskell, suggesting that the conciseness of the programs makes Haskell an excellent tool for prototyping. Prototyping image processing programs in a functional language is advocated by Michaelson *et al*, who use four higher-order functions to code a range of image processing algorithms [97]. In related work, Bratvold compiles these higher-order functions into parallel code for a Meiko Transputer system [22].

5.2 Vectors

Vectors play an important role in signal processing. Recall from section 2.4.4 that the instruction sets of modern DSPs support very efficient vector operations. This section describes a *Vector* datatype implemented in Haskell, and a set of functions in Haskell that can be implemented efficiently on DSPs. These functions are typical of those present in what Sipelstein and Blelloch call "collection-oriented" languages [129].

Appendix A lists the code for the *Vector* module. As for the standard prelude, I will treat this code as a semantic definition only—in other words, the compiler "understands" the vector functions and generates efficient code for them. The key issue is avoiding laziness: if vector functions can be compiled in a hyper-strict context, we can avoid graph-building, and access vector elements through pointers.

5.2.1 The *Vector* datatype

The *Vector* datatype is defined in Haskell as

```
data Vector α = NullV
              | α :> Vector α
```

A vector thus has type *Vector* α, where α is the type of its elements. Defined like this, vectors are much the same as lists, but with different data constructors.

The *vector* function takes a list and produces a vector:

$$\mathit{vector}\ [x, y, z] \rightarrow (x :> y :> z :> \mathit{NullV})$$

Because this is clumsy, I will assume that Haskell has a special syntax for vectors, as it does for lists, and enclose vectors in angle brackets, thus: $\langle x, y, z\rangle$. Instead of *NullV*, I will write $\langle\,\rangle$.

I also find it convenient to annotate the vector type signature with vector lengths. For example, the function to perform an inner-product summation of two vectors has the type:

$$\texttt{ip} :: \texttt{Num}\,\alpha \Rightarrow \texttt{Vector}^k\,\alpha \rightarrow \texttt{Vector}^k\,\alpha \rightarrow \alpha$$

For input arguments, interpret these length annotations as *pre-conditions*: that is, they specify conditions on the arguments that must be satisfied for the function to produce a well-defined result. This is a key part of the "programming-by-contract" approach to software development [95]. In this case, the two input vectors must be the same length. Since a compiler is not required to generate correct code if pre-conditions are violated, an implementation can focus on generating the fastest code possible for two equal-length vectors (for example, by omitting a test for equal length).

If type annotations are attached to nested vector types, then the inner annotation specifies that all sub-vector are the same length. For example,

$$\texttt{xs} :: \texttt{Vector}\,(\texttt{Vector}^k\,\alpha)$$

is a vector of vectors, in which all sub-vectors are the same length, whereas the sub-vectors in

$$\texttt{ys} :: \texttt{Vector}\,(\texttt{Vector}\,\alpha)$$

may be different lengths. By annotating vector functions this way, we can choose either efficient implementation (xs) or generality (ys), as appropriate. It would be an interesting research topic to extend the Haskell type system to infer at compile-time whether or not these constraints are met.

5.2.2 Iterators

Vector iterators are higher-order functions that apply a function across all elements of a vector. In effect, each of them captures a particular pattern of iteration, allowing the programmer to re-use these patterns without risk of error. This is one of the most persuasive arguments in favour of inclusion of higher-order functions in a programming language [64].

Figure 5.1 shows the iterators unfolded, in the style of section 4.5.3. The diagrams illustrate the pattern of computation represented by each iterator in a very *structural* way; this viewpoint of higher-order functions is particularly important when using them to build networks of processes (section 5.4). The iterators all have complexity $O(n)$ in the length of the vector. Their types are listed in figure 5.2.

The simplest iterator is *mapV* (figure 5.1a), which is analogous to *map* on lists. For example,

$$mapV\ (+1)\ \langle 1, 2, 3, 4\rangle \rightarrow \langle 2, 3, 4, 5\rangle$$

iterateV (figure 5.1b) produces a vector of a specified length from an initial value and a function which produces each successive value. The first element of the vector produced by *iterateV* is the initial value; for example,

$$iterateV\ 5\ (+1)\ 3 \rightarrow \langle 3, 4, 5, 6, 7\rangle$$

To generate a vector of k complex numbers equally spaced around the unit circle, we can write:

$$mapV\ cis\ (iterateV\ k\ (+(2.0 \times pi/k))\ 0.0)$$

where $cis\ \theta = cos\ \theta + j\ sin\ \theta$. The *copyV* function creates a vector containing k copies of a given element; it is easily defined with *iterateV*:

```
copyV :: Int → α → Vector α
copyV k x = iterateV k id x
```

where `id x = x`.

foldlV (figure 5.1c) and *foldrV* "fold" a vector into a single value. For example, to sum the elements of a vector,

$$foldlV\ (+)\ 0\ \langle 1, 2, 3, 4\rangle \rightarrow 10$$

This particular expression is common enough to make it worth giving a name to:

```
sumV :: Num α ⇒ Vector α → α
sumV = foldlV (+) 0
```

Like some other vector iterators and many of Haskell's higher-order functions on lists,

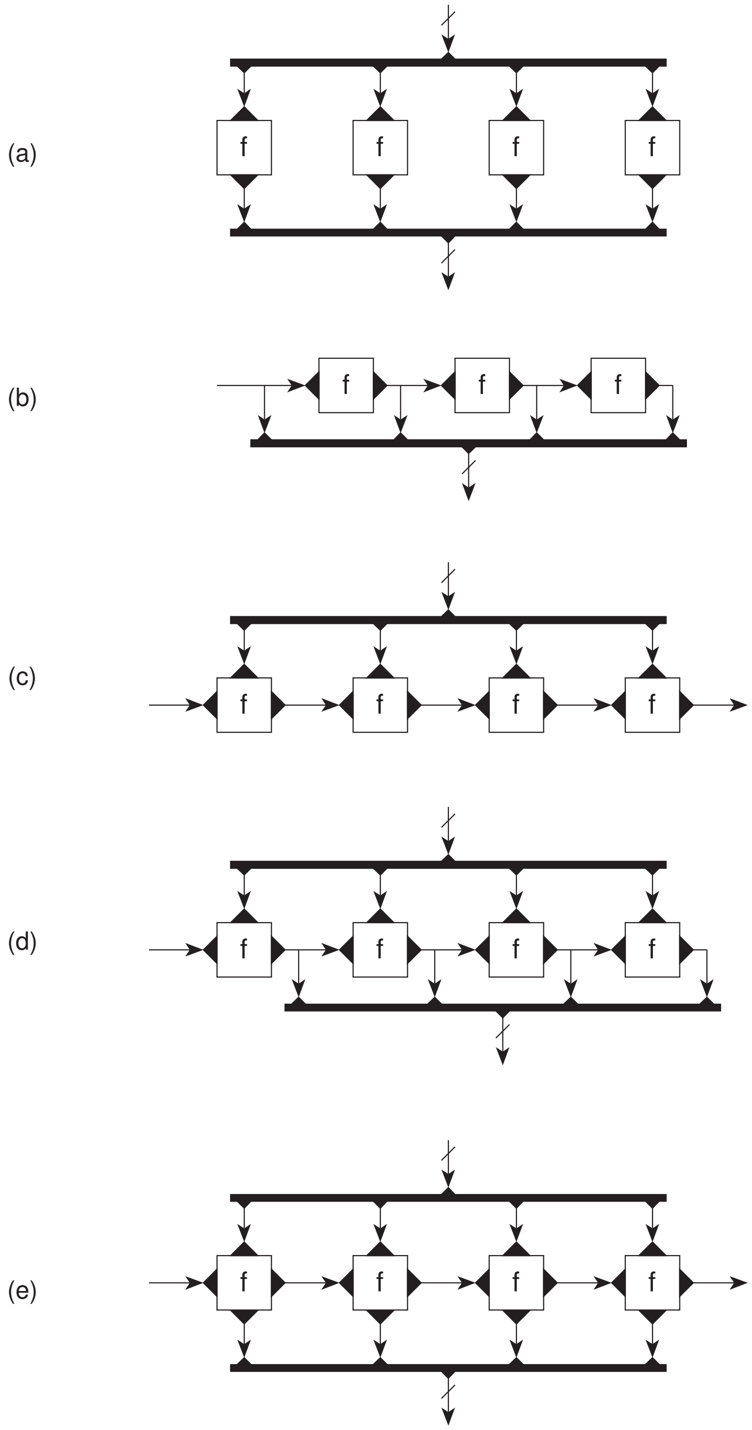


Figure 5.1. Unfolded vector iterators: a) *mapV*; b) *iterateV*; c) *foldlV*; d) *scanlV*; e) *meshlV*

$$
\begin{array}{lll}
\texttt{mapV} & :: & (\alpha \to \beta) \to \texttt{Vector}^k\ \alpha \to \texttt{Vector}^k\ \beta \\
\texttt{iterateV} & :: & \texttt{Int} \to (\alpha \to \alpha) \to \alpha \to \texttt{Vector}\ \alpha \\
\texttt{foldlV} & :: & (\alpha \to \beta \to \alpha) \to \alpha \to \texttt{Vector}\ \beta \to \alpha \\
\texttt{scanlV} & :: & (\alpha \to \beta \to \alpha) \to \alpha \to \texttt{Vector}^k\ \beta \to \texttt{Vector}^k\ \alpha \\
\texttt{meshlV} & :: & (\alpha \to \beta \to (\alpha, \gamma)) \to \alpha \to \texttt{Vector}^k\ \beta \to (\alpha, \texttt{Vector}^k\ \gamma) \\
 & & \\
\texttt{foldrV} & :: & (\beta \to \alpha \to \alpha) \to \alpha \to \texttt{Vector}\ \beta \to \alpha \\
\texttt{scanrV} & :: & (\beta \to \alpha \to \alpha) \to \alpha \to \texttt{Vector}^k\ \beta \to \texttt{Vector}^k\ \alpha \\
\texttt{meshrV} & :: & (\beta \to \alpha \to (\gamma, \alpha)) \to \alpha \to \texttt{Vector}^k\ \beta \to (\alpha, \texttt{Vector}^k\ \gamma)
\end{array}
$$

Figure 5.2. Iterator type signatures

the folding functions come in "left-handed" (*foldlV*) and "right-handed" (*foldrV*) flavours. The left-handed flavour iterates from the first element to the last; the right-handed flavour iterates from the last element to the first. Note, however, that the right-handed functions are not quite mirror-images of the left-handed ones, as the typing of the function argument changes—this can be seen by examining figure 5.2.

scanlV (figure 5.1d) and *scanrV* "scan" a function across a vector: the result produced on each application of the function argument is written to the output vector and used as one input to the next application. For example, to produce a "running sum" of elements in a vector: [1]

$$\mathit{scanlV}\ (+)\ 0\ \langle 1, 2, 3\rangle \to \langle 1, 3, 6\rangle$$

meshlV (figure 5.1e) and *meshrV* capture a "mesh-like" pattern of iteration; they are like a combination of *mapV* and *scanlV* or *scanrV*. The argument function produces a pair of values: the first is input into the next application of this function, and the second is the output value. As an example, consider this expression:

$$\mathit{meshlV}\ (\lambda a\ x\ .\ (x, a))\ 0\ \langle 1, 2, 3\rangle \to (3, \langle 0, 1, 2\rangle)$$

[1] Readers familiar with functional programming will notice that this definition differs from the standard Haskell scanning functions on lists. In comparison,

$$\mathit{scanl}\ (+)\ 0\ [1, 2, 3] \to [0, 1, 3, 6]$$

The vector versions are defined as they are so that the output vector is the same length as the input vector: this affords some efficiency improvement if the input vector can be over-written with the result.

$$
\begin{array}{ll}
\texttt{(+++)} & :: \texttt{Vector}\ \alpha \rightarrow \texttt{Vector}\ \alpha \rightarrow \texttt{Vector}\ \alpha \\
\texttt{concatV} & :: \texttt{Vector}^j\ (\texttt{Vector}^k\ \alpha) \rightarrow \texttt{Vector}^{jk}\ \alpha
\end{array}
$$

$$
\begin{array}{ll}
\texttt{(:>)} & :: \alpha \rightarrow \texttt{Vector}\ \alpha \rightarrow \texttt{Vector}\ \alpha \\
\texttt{(<:)} & :: \texttt{Vector}\ \alpha \rightarrow \alpha \rightarrow \texttt{Vector}\ \alpha \\
\texttt{(>>)} & :: \alpha \rightarrow \texttt{Vector}^k\ \alpha \rightarrow \texttt{Vector}^k\ \alpha \\
\texttt{(<<)} & :: \texttt{Vector}^k\ \alpha \rightarrow \alpha \rightarrow \texttt{Vector}^k\ \alpha
\end{array}
$$

Figure 5.3. Combiner type signatures

Note that the second element of the result pair is the input vector shifted right. We can, if we wish, define the ($\gg$) operator (see next section) as

$$\texttt{(>>)} :: \alpha \rightarrow \texttt{Vector}^k\ \alpha \rightarrow \texttt{Vector}^k\ \alpha$$

```
x >> xs = snd (meshlV (\a x -> (x,a)) x xs)
```

5.2.3 Combiners

Combiners do not perform any computation, but combine vectors and elements into new vectors. Their type signatures are listed in figure 5.3.

$+\!\!+\!\!+$ joins two vectors into a single vector. *concatV* joins a vector of vectors into a single vector. Note that its type signature requires that all sub-vectors be the same length. For example,

$$\langle 1,2,3\rangle +\!\!+\!\!+ \langle 4,5,6\rangle \rightarrow \langle 1,2,3,4,5,6\rangle$$

$$\mathit{concatV}\ \langle\langle 1,2,3\rangle, \langle 4,5,6\rangle\rangle \rightarrow \langle 1,2,3,4,5,6\rangle$$

The remaining combiners implement shift-register-like operations: (:>) and (<:) attach an element to the left or right of a vector respectively; ($\gg$) and ($\ll$) shift an element into the left or right of a vector respectively.

$$0\ :>\ \langle 1,2,3\rangle \rightarrow \langle 0,1,2,3\rangle$$

$$0\ \gg\ \langle 1,2,3\rangle \rightarrow \langle 0,1,2\rangle$$

A smart compiler will sometimes be able to produce $O(1)$ implementations of the combiners. Consider the expression $e_1 +\!\!+\!\!+ e_2$. If the destination address registers for e_1

$$
\begin{array}{lll}
\texttt{lengthV} & :: & \texttt{Vector}\ \alpha \rightarrow \texttt{Int} \\
\texttt{atV} & :: & \texttt{Vector}\ \alpha \rightarrow \texttt{Int} \rightarrow \alpha \\
\texttt{selectV} & :: & \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Vector}\ \alpha \rightarrow \texttt{Vector}\ \alpha \\
\texttt{groupV} & :: & \texttt{Int} \rightarrow \texttt{Vector}^{jk}\ \alpha \rightarrow \texttt{Vector}^{j}\ (\texttt{Vector}^{k}\ \alpha) \\
\texttt{transpose} & :: & \texttt{Vector}^{j}\ (\texttt{Vector}^{k}\ \alpha) \rightarrow\ \texttt{Vector}^{k}\ (\texttt{Vector}^{j}\ \alpha) \\
\texttt{zipV} & :: & \texttt{Vector}^{k}\ \alpha \rightarrow \texttt{Vector}^{k}\ \beta \rightarrow \texttt{Vector}^{k}\ (\alpha, \beta) \\
\texttt{unzipV} & :: & \texttt{Vector}^{k}\ (\alpha, \beta) \rightarrow (\texttt{Vector}^{k}\ \alpha,\ \texttt{Vector}^{k}\ \beta)
\end{array}
$$

Figure 5.4. Selector type signatures

and e_2 are set to point into the memory area allocated for the result of the whole of the above expression, the vectors produced by e_1 and e_2 can be produced directly into the result of (+++). No copying at all is required! To do this will require some sophisticated analysis.

The shift operators can be implemented with $O(1)$ complexity, using the modulo addressing hardware of modern DSP devices (page 43). Thus, the $(\ll)$ operator will write its right argument at the current position in the vector and increment the address register; if it moves outside the vector memory, it is reset by the addressing hardware back to the first position in the memory. A function that reads the vector operates in a similar way, reading from the current position—one position "past" the last element written—up to the last element written. The $(\gg)$ operator is similar, but decrements the address register before performing the write.

5.2.4 Selectors

Selectors do not perform any computation, but just re-arrange vector elements. Figure 5.4 lists their types. In Visual Haskell, most selectors can be drawn as "wiring" (section 4.5.4). Figure 5.5 shows some selectors in this way; as for the unfolded vector iterators, this visual representation provides a very "structural" and intuitive grasp of their operation.

The first function, *lengthV*, returns the length of a vector:

$$\mathit{lengthV}\ \langle 0, 1, 2, 3, 4\rangle \rightarrow 5$$

```
evensV v   = selectV 0 2 (lengthV v `div` 2) v
oddsV v    = selectV 1 2 (lengthV v `div` 2) v
takeV k v  = selectV 0 1 k v
dropV k v  = selectV k 1 (lengthV v - k) v
headV v    = v `atV` 0
tailV v    = dropV 1 v
lastV v    = v `atV` (lengthV v - 1)
initV v    = takeV (lengthV v - 1) v
reverseV v = selectV (lengthV v - 1) (-1) (lengthV v) v
```

Figure 5.5. Additional selector functions

atV selects an individual element of a vector:

$$\langle 0,1,2,3,4\rangle \ \texttt{`atV`}\ 3 \ \rightarrow\ 3$$

selectV selects linearly-spaced elements of a vector. The first argument is the position of the start element; the second the stride between elements; the third the number of elements. Its result is undefined if any elements are "outside" the vector. For example,

$$\mathit{selectV}\ 1\ 3\ 2\ \langle 0,1,2,3,4\rangle \rightarrow \langle 1,4\rangle$$

With *atV* and *selectV*, many other useful functions are easily defined (figure 5.5): to select the odd or even elements of a vector; to take or remove the first k elements of a vector; to split a vector into two equal halves; to produce the "head" and "tail" of a vector; to produce the last element of a vector, and all elements except the last; and to reverse a vector.

groupV splits a vector into a vector of vectors; elements are discarded if the length of the new sub-vectors does not exactly divide the length of the input vector. For example,

$$\mathit{groupV}\ 2\ \langle 0,1,2,3,4,5,6\rangle \rightarrow \langle\langle 0,1\rangle,\langle 2,3\rangle,\langle 4,5\rangle\rangle$$

transpose swaps the two outer dimensions of a vector. For example,

$$\mathit{transpose}\ \langle\langle 0,1\rangle,\langle 2,3\rangle,\langle 4,5\rangle\rangle \rightarrow \langle\langle 0,2,4\rangle,\langle 1,3,5\rangle\rangle$$

zipV and *unzipV* combine two vectors into a vector of pairs, and vice versa. For example,

$$\mathit{zipV}\ \langle 0,1,2\rangle\ \langle 3,4,5\rangle \rightarrow \langle (0,3),(1,4),(2,5)\rangle$$

$$\mathit{unzipV}\ \langle (0,3),(1,4),(2,5)\rangle \rightarrow (\langle 0,1,2\rangle\ \ \langle 3,4,5\rangle)$$

These can be used to implement "butterfly" access patterns, as used in the Fast Fourier Transform algorithm. The two functions *duals* and *unduals* are drawn as "wiring" in figure 5.6. They are defined as:

```
duals     :: Vector^2k α → Vector^k (α, α)
duals v    = zipV (takeV k v, dropV k v)
             where
             k = lengthV v `div` 2

unduals   :: Vector^k (α, α) → Vector^2k α
unduals v = let (x,y) = unzipV v in x +++ y
```

duals requires that the length of its argument is even. Here is an example:

$$\mathit{duals}\ \langle 0,1,2,3,4,5\rangle \rightarrow \langle (0,3),(1,4),(2,5)\rangle$$

zipV and *unzipV* are also useful for defining iterators that accept or produce more than one vector argument. For example, we can define the function *zipWithV*, which applies a function point-wise across two input vectors, as

```
zipWithV :: (α → β → γ) → Vector^k α → Vector^k β → Vector^k γ
zipWithV f xs ys = mapV (\(x,y) -> f x y) (zipV xs ys)
```

Any binary operator or function can be extended to operate on vectors in this way; for example,

$$\mathit{zipWithV}\ (+)\ \langle 1,2,3\rangle\ \langle 4,5,6\rangle \rightarrow \langle 5,7,9\rangle$$

A useful example of this kind of operation is the vector inner-product:

```
ip :: Num α ⇒ Vector^k α → Vector^k α → α
ip v w = sumV (zipWithV (*) v w)
```

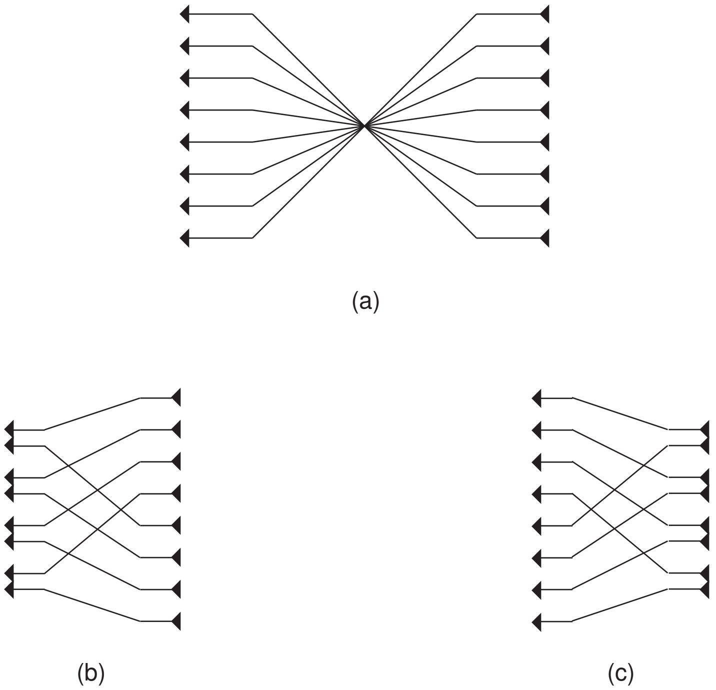


Figure 5.6. Selectors as "wiring": a) *reverseV*; b) *duals*; c) *unduals*

If implemented naively (by copying elements) most of the selectors have $O(n)$ complexity. A smart compiler could, however, reduce most of the selectors to $O(1)$ complexity. To see why, recall that a DSP can increment two address registers by the contents of two index registers in each instruction. Suppose that elements of a vector are located at contiguous addresses in memory. An expression such as $mapV\ (+1)$ will generate a single-instruction loop that, on each iteration, loads and adds one to a vector element, stores the result of the previous iteration, and increments both the source and destination address registers by one. Now consider the expression $mapV\ (+1) \cdot odds$. Before the loop, the source address register is incremented by one (to point to the element at offset one in the vector), and the loop count halved. The loop itself is the same, except that the source address register is incremented by two instead of one. This is easily generalised to arbitrary strides by using index registers.

5.2.5 Example: the Fast Fourier Transform

The Fast Fourier Transform (FFT) algorithm is one of the most important algorithms in signal processing. Although formulated as a recursive "divide-and-conquer" algorithm,

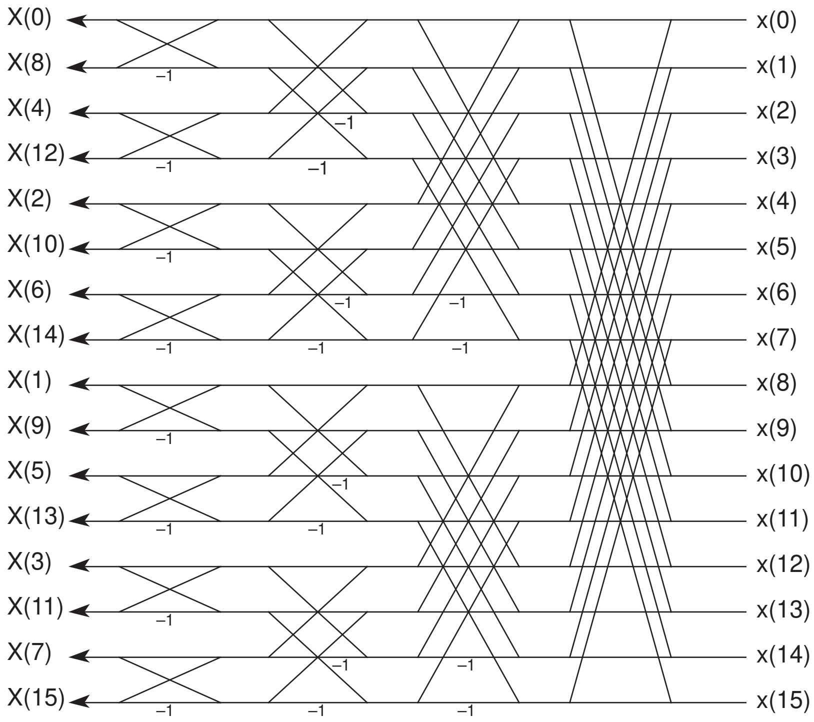


Figure 5.7. The 16-point FFT butterfly diagram

implementations of the algorithm are usually coded in an iterative manner for efficiency. As an aid to understanding the iterative algorithm, a so-called "butterfly diagram" can be drawn; figure 5.7 shows the diagram for the FFT coded here, for a 16-point input vector. This particular form of the FFT is a complex, radix-2, decimation-in-time algorithm with bit-reversed output [134].

Figure 5.8 is the FFT algorithm in Haskell, coded directly from the butterfly diagram. The top level of this function has a guard that checks that the supplied $\log_2$ vector length is correct. To explain how the rest of this code works, I will start at the inner functions and work outwards. Bit-reversed re-ordering is performed by the *bitrev* function, which assumes that the length of its argument vector is a power of two. It can be defined recursively as

```
bitrev :: Vector α → Vector α
bitrev ⟨x⟩ = ⟨x⟩
bitrev xs = bitrev (evens xs) ++ bitrev (odds xs)
```

Usually, however, we would expect an implementation to provide it as a primitive

```
fft :: Int -> Vector (Complex Float) -> Vector (Complex Float)
fft k xs | n == 2 ^ k = (bitrev . foldrV stage xs . iterateV k (* 2)) 2
    where
    stage :: Int -> Vector (Complex Float) -> Vector (Complex Float)
    stage k = concatV . zipWithV segment (takeV m twiddles) . groupV k
              where m = n `div` k

    segment :: Complex Float -> Vector (Complex Float)
               -> Vector (Complex Float)
    segment twid = unduals . mapV (butterfly twid) . duals

    twiddles :: Vector (Complex Float)
    twiddles = (bitrev . mapV (cis . negate) . halfcycle) n

    n = lengthV xs
```

Figure 5.8. The FFT function

function, using the reverse-carry addressing of modern DSPs.

The "twiddle" factors, *twiddles*, is a vector of complex numbers situated on the unit circle. For $n = 16$, the value of twiddles is:

$$\langle W_{16}^0, W_{16}^4, W_{16}^2, W_{16}^6, W_{16}^1, W_{16}^5, W_{16}^3, W_{16}^7 \rangle$$

where

$$W_n^m = e^{-2\pi m/n}$$

twiddles is obtained by applying the *negate* and *cis* functions to negative phase angles on the unit circle, and applying a bit-reversal re-ordering to the result.

The next level processes a single "segment." *segment* groups a segment into pairs of "dual" nodes, performs a butterfly operation on each pair, and then regroups the nodes into the original ordering. Note how partial application is used to distribute the twiddle factor for this segment to all applications of *butterfly*. (*butterfly* is defined on page 22.) Figure 5.9a illustrates (unfolded) a segment of the second stage of a 16-point FFT.

The next level of the FFT is a single "stage" of the algorithm—there are four of these for a 16-point FFT. A stage consists of a number of segments m where $m = 1, 2, 4, 8$, and so on. *stage* divides the data vector into segments, applies *segment* to each segment and

(a)

(b)

(c)

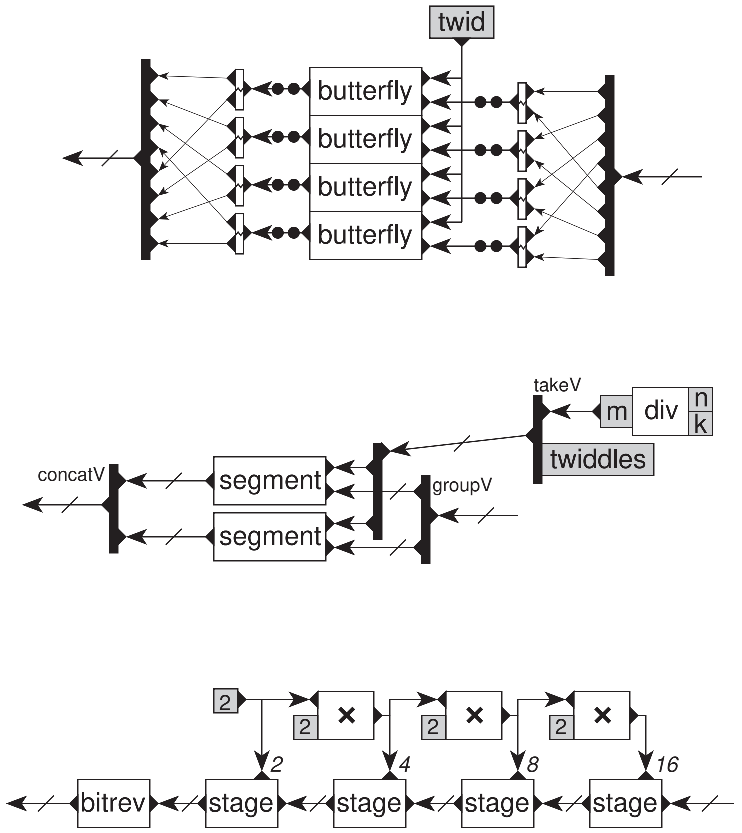


Figure 5.9. Parts of the 16-point FFT in Visual Haskell: a) *segment*; b) *stage*; c) *fft*

a single twiddle factor, and combines the result into a single vector again. Figure 5.9b illustrates (unfolded) the second stage of a 16-point FFT.

The next level is what we might call the "pipeline" level, since it applies a series of stages one after the other; this pattern is captured by *foldrV*. The argument to each stage is the length of segments within that stage: for a 16-point FFT, for example, the vector required is $\langle 16, 8, 4, 2\rangle$. The *iterateV* function generates this vector (in reverse order). Figure 5.9c illustrates this level of the FFT.

5.3 Streams

The Haskell *Stream* datatype defined in this section allows us to simulate dataflow process network programs in Haskell. As for vectors, I will make some assumptions about how the compiler will treat this datatype. The Haskell code for the *Stream* module is listed in appendix A.

5.3.1 The *Stream* datatype

The *Stream* datatype declaration is almost the same as that for *Vector*:

```
data Stream α = NullS
              | α :- Stream α
```

As we saw in sections 2.3.5 and 2.3.6, pragmatic implementations of process networks in functional programming languages evaluate each element of a stream in a hyper-strict context—that is, elements are fully evaluated before transmitting them to another process. Let us therefore assume the same for *Stream*.

The *stream* function produces a stream from a list:

$$\mathit{stream}\ [x, y, z] \rightarrow (x\ \texttt{:-}\ y\ \texttt{:-}\ z\ \texttt{:-}\ \mathit{NullS})$$

I will assume that Haskell has a special syntax for streams, and enclose streams in curly brackets, thus: $\{x, y, z\}$. Instead of *NullS*, I will write $\{\,\}$. In this chapter, I consider only infinite streams.

Annotations on streams indicate their sample rate. To see why, consider the process that sums corresponding elements of two input streams:

$$\texttt{summer :: Num}\ \alpha\ \Rightarrow\ \texttt{Stream}^n\ \alpha \rightarrow \texttt{Stream}^n\ \alpha \rightarrow \texttt{Stream}^n\ \alpha$$

If the two inputs had different sample rates, then buffer memory would very quickly become exhausted. The annotations, treated as pre-conditions, ensure that sample rates are always correct.

If a stream contains vectors, than an annotation on the vector type is a constraint on the lengths of the vectors. Thus, $\texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha)$ is a stream of rate n, containing vectors of length k. These annotations will also be useful when transforming process networks. As for vector lengths, extending the Haskell type system to infer correct sample rates would be most interesting.

5.3.2 Process primitives

In section 3.2.5, I argued that any SDF actor—and therefore any SDF network—can be built with delays and instances of five actor schemata. In this section, I give a Haskell function for each of these actors. The types of the six functions are given in figure 5.10. Recursive definitions are given in section A.

The "cons" operator (`:-`) attaches an element to the front of a stream. It is equivalent to placing an initial value in a stream—that is, a unit delay. For example,

$$x \;\texttt{:-}\; \{a, b, c\} \to \{x, a, b, c\}$$

It is common in signal processing for delays to have an initial value of zero, so define a new function named (confusingly) *delay*:

```
delay       :: Num α ⇒ Stream α → Stream α
delay xs     = zero :- xs
```

groupS breaks a stream into a stream of vectors of length k, where k is its first argument. *groupS* is directly equivalent to the *group* actor. This is the basic mechanism by which the sample rate of streams can be decreased. If the length of the input stream is not a multiple of k, the final incomplete vector is discarded. For example,

$$\mathit{groupS}\ 2\ \{a, b, c, d, e\} \to \{\langle a, b\rangle, \langle c, d\rangle\}$$

concatS is the reverse, concatenating a stream of vectors into a stream of elements. Unlike *concat*, *concatS* does not need an argument giving the length of the vectors in the stream, as it calculates them at run-time. (The vector-length assertion in the type states that all vectors are the same length.) For example,

$$\mathit{concatS}\ \{\langle a, b\rangle, \langle c, d\rangle\} \to \{a, b, c, d\}$$

zipS combines two streams point-wise into a pair of streams. For example,

$$\mathit{zipS}\ \{a, b, c\}\ \{1, 2, 3\} \to \{(a, 1), (b, 2), (c, 3)\}$$

unzipS is almost the reverse of *zipS*: it produces a pair of streams from a stream of

$$
\begin{array}{lll}
\texttt{(:-)} & :: & \alpha \rightarrow \texttt{Stream}^n\ \alpha \rightarrow \texttt{Stream}^n\ \alpha \\
\texttt{groupS} & :: & \texttt{Int} \rightarrow \texttt{Stream}^{nk}\ \alpha \rightarrow \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha) \\
\texttt{concatS} & :: & \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha) \rightarrow \texttt{Stream}^{nk}\ \alpha \\
\texttt{zipS} & :: & \texttt{Stream}^n\ \alpha \rightarrow \texttt{Stream}^n\ \beta \rightarrow \texttt{Stream}^n\ (\alpha, \beta) \\
\texttt{unzipS} & :: & \texttt{Stream}^n\ (\alpha, \beta) \rightarrow (\texttt{Stream}^n\ \alpha, \texttt{Stream}^n\ \beta) \\
\texttt{mapS} & :: & (\alpha \rightarrow \beta) \rightarrow \texttt{Stream}^n\ \alpha \rightarrow \texttt{Stream}^n\ \beta
\end{array}
$$

Figure 5.10. Types of stream functions

pairs. For example,

$$unzipS\ \{(a,1),(b,2),(c,3)\} \rightarrow (\{a,b,c\},\{1,2,3\})$$

zipS and *unzipS* are equivalent to the actors *zip*(2) and *unzip*(2) respectively. Haskell's type system does not allow a function to accept a variable number of arguments. This restriction is overcome by assuming that there exists a whole family of zipping and unzipping functions: *zipS3*, *zipS4*, and so on. In theory, this does not affect my claim that the six functions given in this section are the minimum set of functions needed to write any SDF actor, since triples, four-tuples, and so on can be represented by nested pairs, and the zipping functions defined in terms of *zipS* and *unzipS*.

mapS is equivalent to the *map* actor. It applies a function to each element of a stream:

$$mapS\ (+1)\ \{1,2,3,4\} \rightarrow \{2,3,4,5\}$$

In section 3.2.3, I discussed the difference between "parameter" and "stream" arguments. The functional programming language deals quite well with the two different types of argument: as long as the stream arguments follow the parameter arguments, it is easy to create parameterised stream functions that are equivalent to instantiated actors: (*groupS* 4), for example, is the function that divides its stream argument into 4-vectors. Slightly more sophisticated are examples like *mapS* $(*4)$, which multiplies each element of its stream argument by four. There is no need for a "constant stream" of fours, as would be required by languages such as Lucid [144].

A further point to note is a difference in the way that processes are instantiated. In section 3.2.3, an actor is instantiated by binding it to a unique vertex name; stream arguments are supplied via the graph topology. With the Haskell functions of this section, however, a process does not exist as a unique entity until *all* arguments have been supplied. I will call a function that contains streams in its input or output types a *process-function*; once all arguments are supplied, the process-function applied to its "parameter" arguments becomes a *process*. For example, *mapS* and *mapS* f, applied to no further arguments, are process functions; *mapS* f xs is a stream, produced by the process *mapS* f.

Because a process-function does not become a process until all arguments are supplied, process-functions can be supplied as arguments to higher-order functions. For example, we can write *mapV* (*mapS* f) xs to produce a vector of processes; section 5.4 explores this idea.

5.3.3 An example

A first-order recursive digital filter is defined by

$$y(n) = a_0 x(n) + a_1 x(n-1) - b_1 y(n-1)$$

where a_0, a_1, and b_1 are the filter coefficients. This can be reformulated as

$$w(n) = x(n) - b_1 y(n-1)$$

$$y(n) = a_0 w(n) + a_1 w(n-1)$$

The block diagram of this simple filter is shown in figure 5.11a. Define *scale* and *summer* (page 60) as Haskell functions:

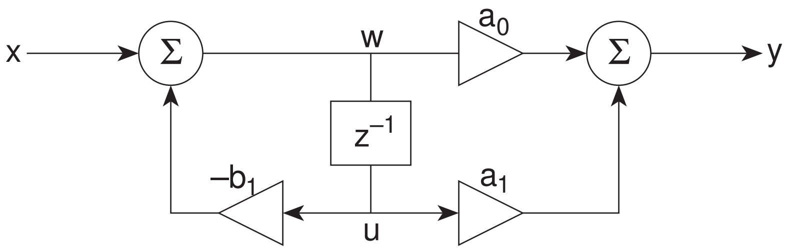
```
scale        :: Num α ⇒ α → Stream α → Stream α
scale a xs    = mapS (* a) xs

summer       :: Num α ⇒ Stream α → Stream α → Stream α
summer xs ys = mapS (\(x,y) -> x + y) (zipS xs ys)
```

Then, we can define the first-order IIR filter in Haskell as:

```
iir1 :: Num α ⇒ α → α → α → Stream α → Stream α
iir1 a0 a1 b1 x = let
```

(a)



(b)

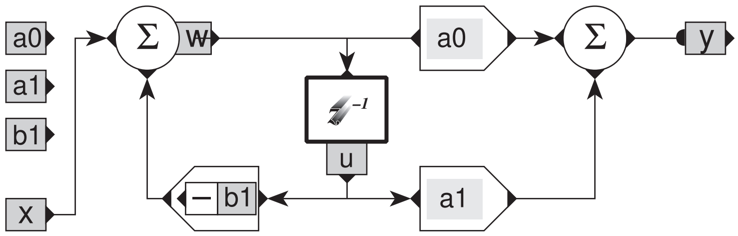


Figure 5.11. A first-order recursive filter: a) in block-diagram notation; b) in Visual Haskell

```
        w = summer x (scale (- b1) w)
        u = delay w
        y = summer (scale a0 w) (scale a1 u)
    in
        y
```

Figure 5.11b shows this definition in Visual Haskell, laid out to mimic the block diagram, and with icons used for *scale*, *summer*, and *delay*. In this diagram, and from here on, I omit the containing window for function definitions. This example demonstrates that Visual Haskell can be made to look like block diagrams, and should therefore be able to tap the same appeal to intuition as block diagram systems. Note the importance of defining suitable functions to aid the visual representation of programs. In the textual program, it would have been as easy to write, for example,

```
u = 0 :- w
y = summer (mapS (* a0) w) (mapS (* a1) u)
```

instead of

```
u = delay w
y = summer (scale a0 w) (scale a1 u)
```

but we would then not have been able to use icons as effectively.

5.3.4 Process constructors

Of the primitive functions, *mapS* is the only higher-order function. Additional higher-order functions are readily defined in terms of the primitives. I call higher-order functions on streams *process constructors*, because of their key role in constructing process networks. Their types and definitions are listed in figure 5.12; their definitions are shown diagrammatically in figure 5.13.

zipWithS (figure 5.13a), *zipOutS*, and *zipOutWithS* (figure 5.13b) are point-wise process constructors like *mapS*, but build processes with more than one input or output. For example, a point-wise addition process is written

$$\mathit{zipWithS}\ (+)$$

A process that outputs the sum and difference of two inputs is written:

$$\mathit{zipOutWithS}\ (\lambda x\ y\,.\,(x+y, x-y))$$

The other process constructors maintain a state. Although the definitions given in figure 5.12 use a feedback loop to carry the state around a purely-functional actor, recursive definitions could also be given. For example, *scanS* could be defined as

```
scanS f a NullS   = NullS
scanS f a (x:-xs) = let x' = f a x in x' :- scanS f x' xs
```

The two definitions are equivalent; I have chosen the state-as-feedback form so that these new functions can be treated as hierarchical dataflow actors.

iterateS (figure 5.13c) and *generateS* (figure 5.13d) produce a stream from a single input value; the key difference is that *iterateS* outputs the state on each application of its argument functions, while *generateS* produces each output value and the next state on

```
zipWithS :: (α → β → γ) → Stream α → Stream β → Stream γ
zipWithS f xs ys  = mapS (\(x,y) -> f x y) (zipS xs ys)

zipOutS :: (α → (β, γ) → Stream α → (Stream β, Stream γ)
zipOutS f xs = unzipS (mapS f xs)

zipOutWithS :: (α → β → (γ, δ)) → Stream α → Stream β
                         → (Stream γ, Stream δ)
zipOutWithS f xs ys = unzipS (mapS (\(x,y) -> f x y) (zipS xs ys))

iterateS :: (α → α) → α → Stream α
iterateS f a  = let ys = a :- (mapS f ys) in xs

generateS :: (α → (α, β)) → α → Stream β
generateS f a = let (zs,ys) = zipOutS f (a :- zs) in ys

scanS :: (α → β → α) → α → Stream β → Stream α
scanS f a xs = let ys = zipWithS f (a :- ys) xs in ys

stateS :: (α → β → (α, γ)) → α → Stream β → Stream γ
stateS f a xs = let (zs,ys) = zipOutWithS f (a :- zs) xs in ys
```

Figure 5.12. Process constructor definitions

each application. For example, the process that produces the stream $\{0, 1, 2...\}$ is written

$$\mathit{iterateS}\ 0\ (+1)$$

A useful function defined in terms of *iterateS* generates a stream containing an infinite number of copies of its argument:

```
repeatS :: α → Stream α
repeatS = iterateS id
```

scanS (figure 5.13e) and *stateS* (figure 5.13f) also propagate a state value between iterations. For example, the process

$$\mathit{scanS}\ 0\ (+)$$

produces the running sum of its input stream. A more interesting process is one that,

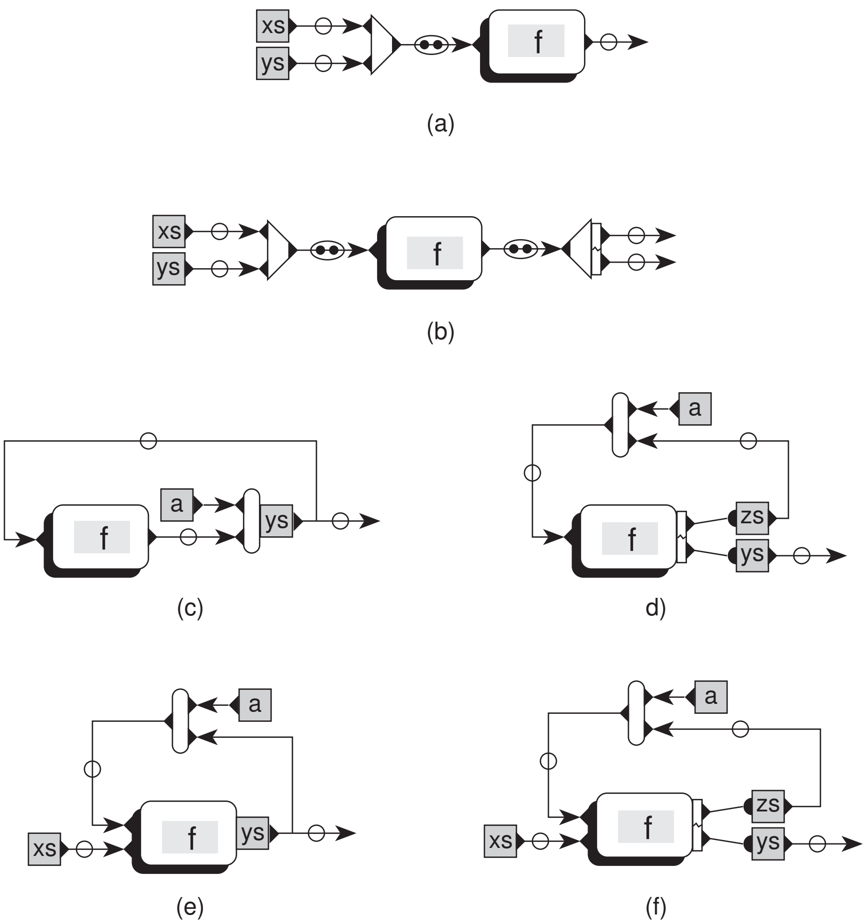


Figure 5.13. Process constructors: a) *zipWithS*; b) *zipOutWithS*; c) *iterateS*; d) *generateS*; e) *scanS*; f) *stateS*

given some integer k, outputs, for each input value, a vector containing that value and the past $k-1$ values:

```
slide   ::  Num α  ⇒  Int → Stream α → Stream (Vector α)
slide k = scanS (<<) (copyV k 0)
```

For example,

$$\mathit{slide}\ 3\ \{1,2,3,4\} \rightarrow \{\langle 0,0,1\rangle, \langle 0,1,2\rangle, \langle 1,2,3\rangle, \langle 2,3,4\rangle\}$$

This is a common operation in signal processing. To illustrate, the equation defining

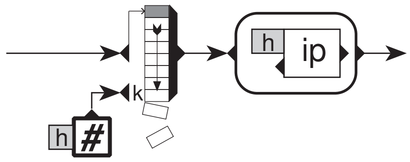


Figure 5.14. The FIR filter function

a k'th-order finite-impulse-response (FIR) filter with impulse response h is:

$$y(n) = \sum_{i=0}^{k} h(i)x(n-i)$$

In other words, for each input value, output the inner product of the impulse response and the most recent k input values. In Haskell:

```
fir   ::  Num α ⇒ Vector α → Streamⁿ α → Streamⁿ α
fir h = mapS (ip h) . slide (lengthV h)
```

fir is illustrated in figure 5.14. The icon for *slide* is supposed to evoke a stack with items "falling off" the bottom. There is a large amount of redundant communication between the two processes: each input value is sent from the *slide* process to the *scanS* process $\#h$ times. This apparent overhead can be removed using program transformation—see section 5.5.2.

Consider now a recursive filter of arbitrary order, shown in block diagram form in figure 5.15a. Its equation is

$$w(n) = x(n) - \sum_{i=1}^{k} b(i)w(n-i)$$

$$y(n) = \sum_{i=0}^{k} a(i)w(n-i)$$

where k is the filter order. This diagram is difficult to draw in a formal notation because of the arbitrary order. As for the FIR filter, we group all the z^{-1} delays into a single state with *slide*. In the following definition, illustrated in Visual Haskell in figure 5.15b, I have used two applications of *slide* for clarity:

```
iirn ::  Num α ⇒ Vector α → Vector α → Streamⁿ α → Streamⁿ α
```

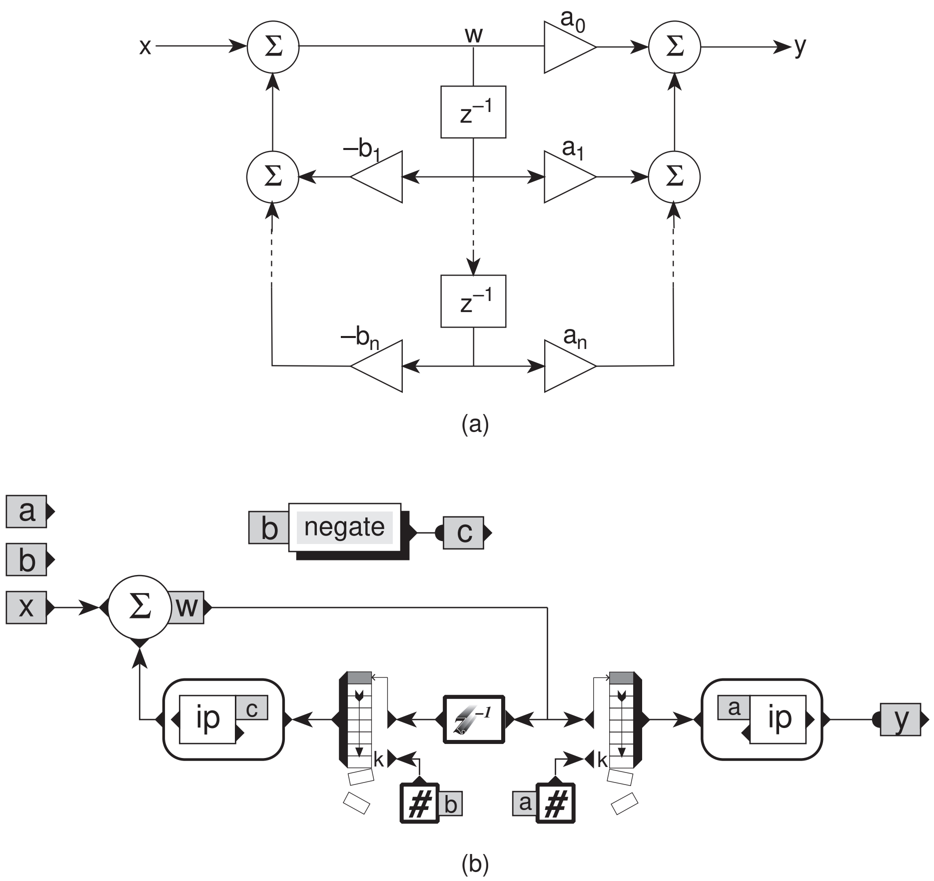


Figure 5.15. The n-th order recursive filter: a) block-diagram form; b) generalised form

```
iirn a b x
  = let
      c = mapV negate b
      w = summer x (mapS (ip c) (slide (lengthV b) (delay w)))
      y =           mapS (ip a) (slide (lengthV a)        w)
    in
      y
```

5.4 Process network construction

The previous section gave examples of *first-order* process networks. A first-order network is one in which a process-function never appears as an argument to a higher-order function—each process thus appears explicitly in the program text.

We can take advantage of the powerful features of the host functional language to increase the expressive power available to us for writing dataflow process network programs.

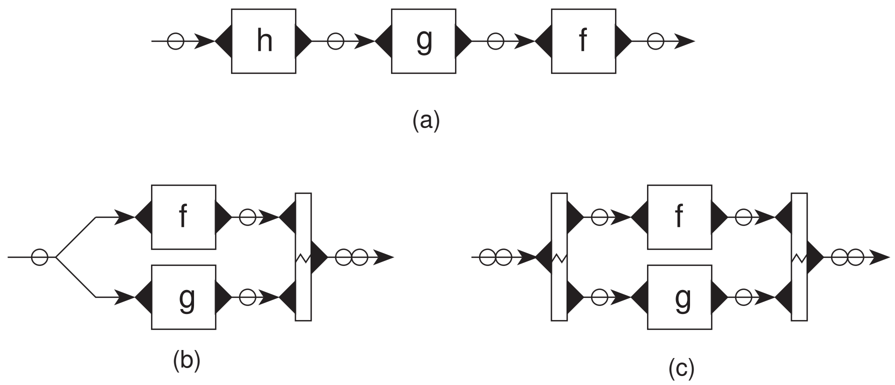


Figure 5.16. Simple network-forming functions: a) composition; b) *fan*; c) *par*

Higher-order functions are the most useful of these: they capture patterns of instantiation and interconnection between processes. This section is mainly concerned with illustrating the kinds of networks that can be produced this way; the resulting style of programming could perhaps be called *data-parallel process parallelism*.

5.4.1 Simple combinators

Function composition connects processes in series; a series of composed functions forms a pipeline. Figure 5.16a illustrates the pipeline $f \cdot g \cdot h$, where f, g, and h are processes. New higher-order functions to express other topologies are easily defined. For example, *fan* applies two processes to one stream; *par* applies two processes each to one stream of a pair. They are defined as follows:

fan $\quad$ :: $(\alpha \to \beta) \to (\alpha \to \gamma) \to \alpha \to (\beta, \gamma)$

`fan f g xs      = (f xs, g xs)`

par $\quad$ :: $(\alpha \to \beta) \to (\gamma \to \delta) \to (\alpha, \beta) \to (\gamma, \delta)$

`par f g (xs,ys) = (f xs, g xs)`

Figure 5.16b and c illustrate these two functions. Note, however, that this visual syntax has not been defined in chapter 4. An important feature of a visual language implementation for visualising these kinds of networks is thus user-defined visual syntax for higher-order functions.

5.4.2 Simple linear networks

Supplying a process-function as argument to any vector or list iterator gives rise to an array of processes (or process-functions). The expression

$$\mathit{mapV}\ (\mathit{mapS}\ f)\ \mathit{xs}$$

where $\mathit{xs} :: \texttt{Vector}\ (\texttt{Stream}^n\ \alpha)$ is a vector of streams, is a linear network of identical processes. The network is shown in figure 5.17a. A similar network results for point-wise constructors:

$$\mathit{zipWithV}\ (\mathit{zipWithS}\ f)\ \mathit{xs}\ \mathit{ys}$$

These examples used *mapS* or *zipWithS* as the process constructor. This is for ease of explanation—process constructors that maintain state can also be used. For example, the following expression builds a linear array of processes, each built with *scanS*:

$$\mathit{mapV}\ (\mathit{scanS}\ f\ a)\ \mathit{xs}$$

where a is the initial state of all processes. To build an array of processes with differing initial state, each taken from a vector *as*, we can write

$$\mathit{zipWithV}\ (\mathit{scanS}\ f)\ \mathit{as}\ \mathit{xs}$$

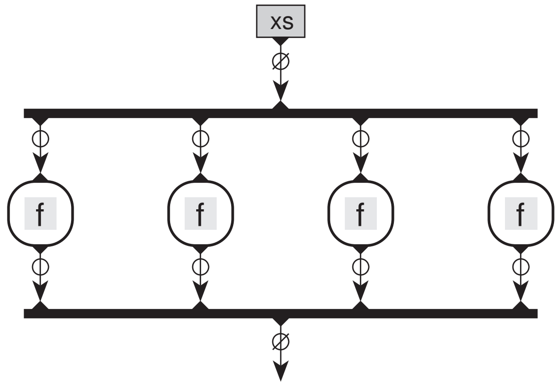
The previous examples built arrays of identical processes. We can, if we wish, apply a *vector* of functions *fs* so that each process in the network behaves differently:

$$\mathit{zipWithV}\ \mathit{mapS}\ \mathit{fs}\ \mathit{xs}$$

Figure 5.17b illustrates the network that results. This construction corresponds to *par*, using vectors of functions and data instead of pairs. The network corresponding to *fan*, in which a linear array of (different) processes is applied to a *single* stream *xs* is this:

$$\mathit{mapV}\ (\lambda f\,.\,\mathit{mapS}\ f\ \mathit{xs})\ \mathit{fs} \tag{5.1}$$

(a)



(b)

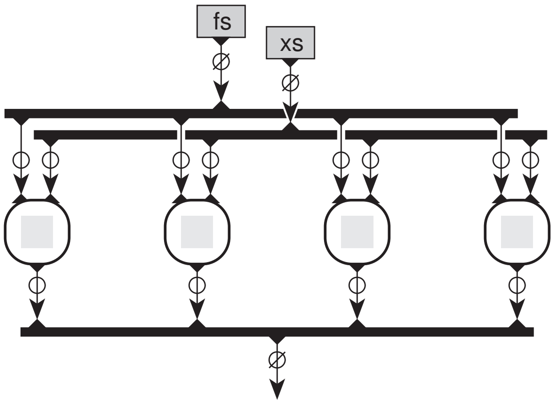


Figure 5.17. Linear process networks

5.4.3 Pipelines

Composition builds pipelines in which each process is explicitly represented in the text (figure 5.16a). It is easy to define a higher-order function that connects a vector of functions in series:

$$\texttt{series} \quad :: \texttt{Vector}\ (\alpha \to \alpha) \to \alpha \to \alpha$$
```
series fs = foldlV (.) id fs
```

If each f in *fs* is a process-function, then *series* produces a pipeline. For example, to create a pipeline of k identical processes, make k copies of the process and pipeline them:

$$\mathit{series}\ (\mathit{copyV}\ k\ p)\ \mathit{xs}$$

where $p :: \texttt{Stream}^n\ \alpha \to \texttt{Stream}^n\ \alpha$. One useful way of creating a pipeline of different processes is to *parameterise* a process by supplying each instance of the process with a different first argument, as in

$$\mathit{series}\ (\mathit{mapV}\ p\ v)\ xs \tag{5.2}$$

where p is the process and v is a vector of arguments. To see how this works, suppose that $p = \lambda i\,.\,\mathit{mapS}\ (f\ i)$ and $v = \langle 1, 2, 3\rangle$. The expression $(\mathit{mapV}\ p\ v)$ evaluates to

$$\langle \mathit{mapS}\ (f\ 1),\ \mathit{mapS}\ (f\ 2),\ \mathit{mapS}\ (f\ 3)\rangle$$

Unfolding *series* and then *foldlV* gives

$$(\mathit{mapS}\ (f\ 1) \cdot \mathit{mapS}\ (f\ 2) \cdot \mathit{mapS}\ (f\ 3))$$

which is shown in figure 5.18.

There is, however, a more direct way to achieve this result. We can relate *series* to *foldrV* with the following identity:

$$\mathit{series}\ (\mathit{mapV}\ f\ v)\ x \ \equiv\ \mathit{foldrV}\ f\ x\ v \tag{5.3}$$

The pipeline of parameterised processes in equation 5.2 becomes

$$\mathit{foldrV}\ p\ xs\ v \tag{5.4}$$

We could instead use *foldlV* to make data flow through the pipeline from left to right. Some care is needed, as the order of arguments to p must be swapped:

$$\mathit{foldlV}\ (\mathit{flip}\ p)\ xs\ v$$

The idea of parameterising a process network by mapping a process to a vector of parameters occurs often in pipelines, but is easily used for other network structures. For example, equation 5.1 is an example in which many instantiations of a process are applied to a single stream; to parameterise each process, we first produce a vector of parameterised processes, then map these to the stream xs:

$$\mathit{mapV}\ (\lambda p\,.\,p\ xs)\ (\mathit{mapV}\ p\ v)$$

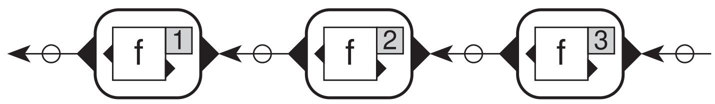


Figure 5.18. A pipeline process network

In this particular case, it is easy to write an expression for the desired network directly, so:

$$mapV\ (\lambda i\,.\,p\ i\ xs)\ v$$

The way in which the parameter to p affects it depends on how p is constructed. For example, if

$$p = \lambda i\,.\,mapS\ (f\ i)$$

then the parameter i becomes the first argument to the mapped function, f. If, however,

$$p = scanS\ f$$

then i is the initial state of p.

5.4.4 Meshes and systolic arrays

The remaining vector iterators, when used with a process-function argument, give rise to various mesh-structured networks. For example, the following expression, illustrated in figure 5.19a, "folds" a vector of streams into a single stream:

$$foldlV\ (zipWithS\ f)\ s\ ts$$

where f is a binary function. The vector generation functions, used in this way, produce a vector of streams from a single stream, as illustrated in figure 5.19b:

$$iterateV\ k\ (mapS\ f)\ s$$

The scanning vector functions produce a vector of streams from a vector of streams; as a result, they can be used to express two-dimensional arrays of processes. For example,

$$meshlV\ (zipOutWithS\ f)\ s\ ts$$

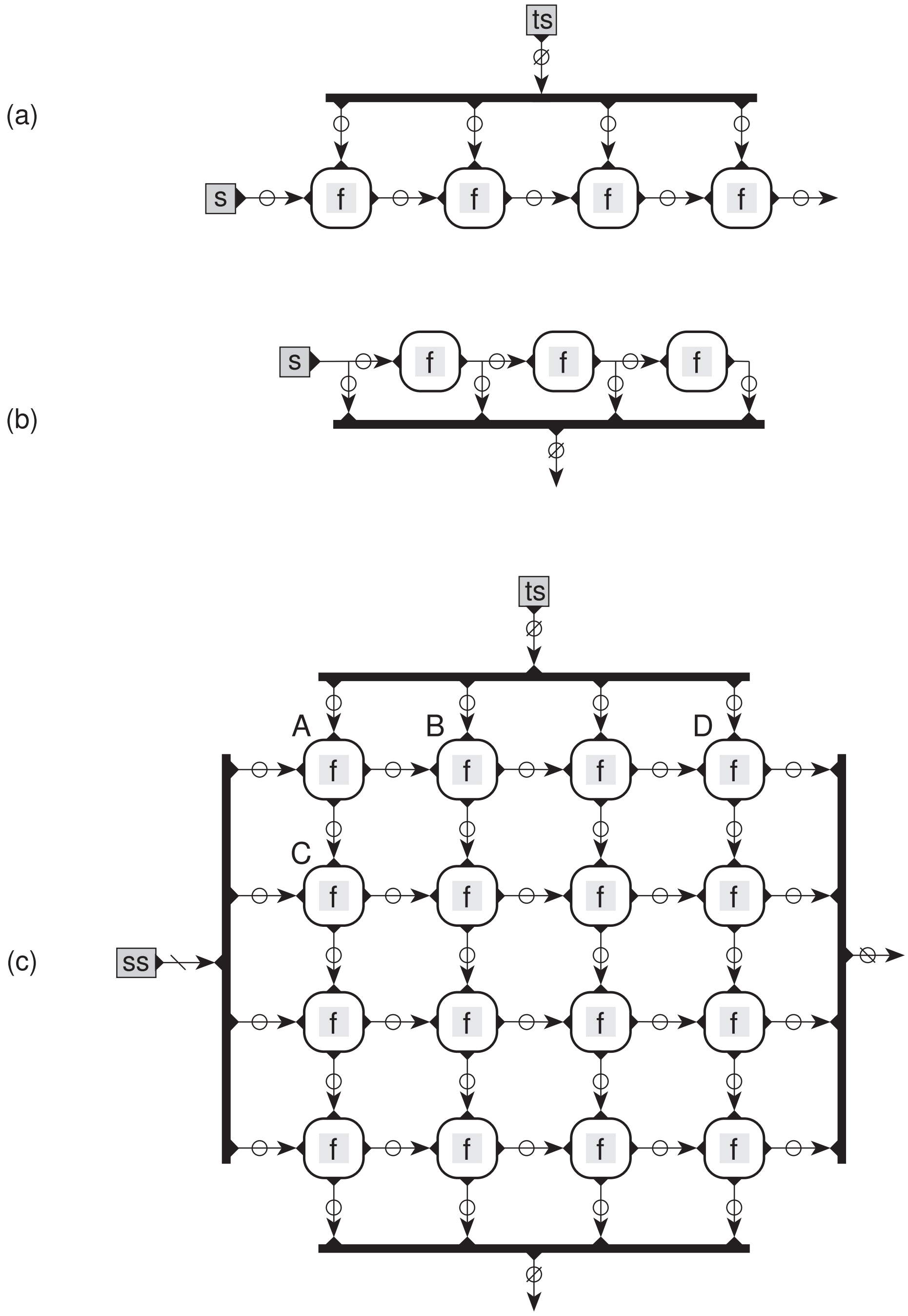


Figure 5.19. Mesh process networks

produces an array of processes with a mesh-like inter-connection pattern. By applying this function to a *second* vector of streams, we get a two-dimensional array of processes as shown in figure 5.19c. The expression for this network is

$$\mathit{meshrV}\ (\mathit{meshlV}\ (\mathit{zipOutWithS}\ f))\ \mathit{ss}\ \mathit{ts}$$

The two-dimensional mesh is very like a systolic array, in which a "wave" of computation proceeds across the array. Patterns such as these algorithms developed for hardware implementation, and there has been some interesting work on using functional languages for hardware design. Sheeran's μFP language, for example, is a functional language based on Backus' FP, and includes structuring operations similar to those presented here [128]. This language has evolved into the hardware design and verification language Ruby, in which circuits are represented as relations between signals (instead of as functions from signals to signals) [68]. Haskell has also been used to specify and simulate systolic arrays: McKeown and Revitt give higher-order functions for expressing systolic arrays and illustrate with a number of algorithms [93].

5.4.5 Network construction in dataflow systems

Recently, two block diagram systems have included network construction. In GRAPE-II, *geometric parallelism* specifies multiple invocations of a block [85]. In the visual language, these invocations all appear as one block. The number of invocations is specified in a textual language;[2] these values must be known at compile-time. The connection topology is also specified in the textual language. If the output of one invocation feeds into the input of the next, an additional feedback arc is also shown. This technique allows complex structures to be expressed without needing to provide higher-order functions.

Ptolemy has adopted a form of higher-order function notation [87]. Special blocks represent multiple invocations of a "replacement actor." The *Map* actor, for example, is a generalised form of *mapV*. At compile time, *Map* is replaced by the specified number of invocations of its replacement actor; as for GRAPE-II, this number must be known at compile-time. Unlike *mapV*, *Map* can accept a replacement actor with arity > 1; in this

[2] The referenced paper gives these values in a textual form. It is not clear whether they can be expressed in the visual language.

case, the vector of input streams is divided into groups of the appropriate arity (and the number of invocations of the replacement actor reduced accordingly).

The requirement that the number of invocations of an actor be known at compile-time ensures that static scheduling and code generation techniques will still be effective. Further work is required to explore forms of higher-order function mid-way between fully-static and fully-dynamic. For example, a code generator that produces a loop with an actor as its body, but with the number of loop iterations unknown, could still execute very efficiently.

5.5 Process network transformation

Because they lack side-effects, functional languages lend themselves very well to program transformation. In this section, I will demonstrate the algebraic style of program transformation, exemplified by the Bird-Meertens formalism (BMF) [20]. The key characteristic of this style is that it makes extensive use of a "catalogue" of known algebraic laws of functions, rather than relying on discovery through lower-level methods. BMF, also known as "Squiggol," is being developed as a "programming by calculation" method of developing programs from specifications. I will use Haskell as the notation for writing transformations, in a similar manner to [21], instead of the more concise notation of [20].

The key contribution of this section is to adapt the Bird-Meertens theory of lists to work with streams; the result is a powerful tool for restructuring process networks. Particularly powerful transformations become available for networks constructed with higher-order functions. I identify four distinct classes of transformation suitable for use with process networks. The transformations given here are common cases, and are by no means an exhaustive catalogue.

Transforming programs to improve performance is also advocated by Carriero and Gelernter [32], who transform between programs expressed in the most "natural" of their three categories of parallelism, into a more efficiently-implementable category. However, their transformations are *ad-hoc*, lacking the equational basis of the Bird-Meertens style.

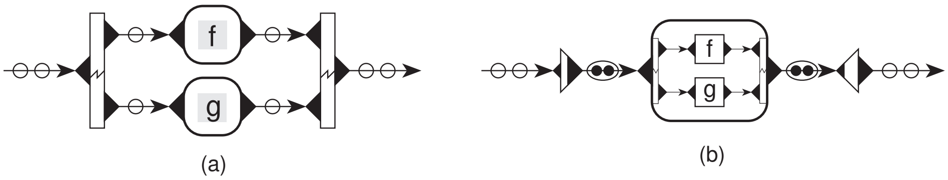


Figure 5.20. Illustrating network types

5.5.1 Type annotations

Transformations in the Bird-Meertens formalism (BMF) are, in general, independent of structural information about the arguments of the transformed functions, other than the types. Unfortunately, many of the transformations of this section are correct only if arguments satisfy certain structural constraints. In particular, we will be concerned with the lengths of vectors, and the data rates of streams. For example, consider the simple network (figure 5.20a)

$$par\ (mapS\ f)\ (mapS\ g) \tag{5.5}$$

We could merge the two processes into one (figure 5.20b), and transform equation 5.5 into

$$unzipS \cdot mapS\ (par\ f\ g) \cdot (uncurry\ zipS) \tag{5.6}$$

If the sample rates of the two streams are different, however, equation 5.6 will not operate as a real-time program. Because equation 5.6 "consumes" data from both streams at equal rates; data will accumulate on the input with the higher rate, resulting in buffer overflow or heap exhaustion. This problem can be isolated at *zipS*: the sample rates of the two streams do not meet the sample rate pre-conditions of *zipS*.

To specify that transformations are only valid if certain length or sample-rate conditions are satisfied, I will append the annotated type of the expressions on either side of a transformation. Thus, our example transformation would be written:

$$\begin{aligned} par\ (mapS\ f)\ (mapS\ g) &\equiv unzipS \cdot mapS\ (par\ f\ g) \cdot uncurry\ zipS \\ &:: (\mathtt{Stream}^n\ \alpha, \mathtt{Stream}^n\ \beta) \rightarrow (\mathtt{Stream}^n\ \gamma, \mathtt{Stream}^n\ \delta) \end{aligned}$$

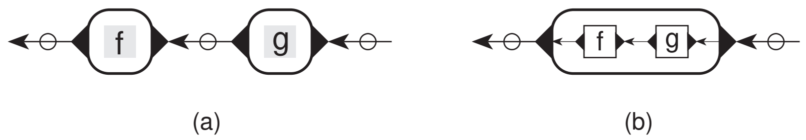


Figure 5.21. Process fusion

5.5.2 Fusion

A common example of an algebraic law on functions is *map distributivity*, so called because it expresses the fact that *map* distributes backwards through function composition:

$$map\ f \cdot map\ g \quad \equiv \quad map\ (f \cdot g)$$

This law states that applying g to each element of a list and then f to each element of the list, is equivalent to applying g then f to each element of the list. The correctness of this law depends only upon the properties of *map*, not upon properties of f or g. This is the essence of the algebraic style of program transformation: computational behaviours are captured by functions (with particular emphasis on higher-order functions); structural properties of expressions are then easily recognised; expressions are transformed using appropriate function identities. There are similar identities for other combinations of functions—*foldl* and *map*, for example; collectively, these are known as *loop fusion* identities.

The first class of process network transformations is a simple adaptation of these identities to process network. The simplest is

$$\begin{aligned} mapS\ f \cdot mapS\ g \quad &\equiv \quad mapS\ (f \cdot g) \\ &:: \quad \mathtt{Stream}^n\ \alpha \rightarrow \mathtt{Stream}^n\ \beta \end{aligned} \tag{5.7}$$

The left-hand side of equation 5.7 represents two processes (figure 5.21a); the right-hand side represents a single process (figure 5.21b). Transformation using this identity thus has the effect of increasing or decreasing parallelism, depending on the direction in which the transformation is applied. The types of all fusion identities are the same as equation 5.7, so will be omitted below.

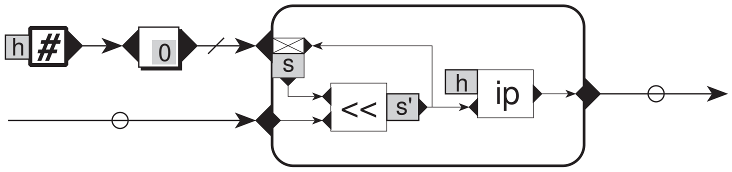


Figure 5.22. The transformed FIR filter

We can easily find more fusion identities. Fusion of two processes constructed with *mapS* and *scanS* is captured by:

$$mapS\ f \cdot scanS\ \ g\ a \quad \equiv \quad stateS\ (\lambda a\ x\ .\ let\ a' = g\ a\ x\ in\ (a',\ f\ a'))\ a \tag{5.8}$$

A more general identity captures fusion of two processes built with *stateS*:

$$\begin{aligned} stateS\ f\ a \cdot stateS\ g\ b \quad \equiv \quad stateS\ \ (\lambda(a,b)\ x\ .\ \ let \quad & (b',t) = g\ b\ x \\ & (a',y) = f\ a\ t \\ in \quad & ((a',b'),\ y) \\ )\ (a,b) \quad & \end{aligned} \tag{5.9}$$

To illustrate process fusion, consider the inefficient version of the FIR filter (page 147). Transformation into a more efficient version is straight-forward:

$$\begin{aligned} fir\ h \quad = \quad & \text{(definition of } fir\text{)} \\ & mapS\ (ip\ h) \cdot slide\ \ (lengthV\ h) \\ = \quad & \text{(unfold } slide\text{)} \\ & mapS\ (ip\ h) \cdot scanS\ \ (\ll)\ \ (copyV\ (lengthV\ h)\ 0) \\ = \quad & \text{(fusion: equation 5.8)} \\ & stateS\ \ (\lambda a\ x\ .\ let\ a' = (\ll)\ a\ x\ in\ (a',\ (ip\ h)\ a'))\ \ (copyV\ (lengthV\ h)\ 0) \\ = \quad & \text{(simplify)} \\ & stateS\ \ (\lambda a\ x\ .\ let\ a' = a \ll x\ in\ (a',\ ip\ h\ a'))\ \ (copyV\ (lengthV\ h)\ 0) \end{aligned}$$

This version of the FIR filter is harder to read than the original version. Figure 5.22 shows this version in Visual Haskell using an iterator view (section 4.5.2). Although we could have written this less-readable but more efficient version from scratch, it is readily derived from the easier-to-read and more maintainable version. As well as being easier to read, the original version uses a well-known utility function (*slide*).

A difficulty with the process fusion identities is that there seems to be no end to the number of identities that we may require. This problem was noted some time ago by Wadler in the context of lazy lists [145]. This led to the development of (successively) the techniques of listlessness [146] and deforestation [148], aimed at automatic fusion of list functions for program efficiency. More recently, Gill *et al* discovered an improved method [51], which is now part of some Haskell compilers.

It is not easy to see, however, how these techniques can be applied to our purposes, for two reasons. Firstly, they rely on expanding the functions in question to recursive first-order form; we, however, seek to retain the higher-order functions, since they are our key to efficient "imperative" implementation as dataflow actors. Secondly, the transformations are in the fusion direction only; as often as not, we are as interested in making processes smaller ("fission") instead of larger.

5.5.3 Parallelisation

The second class of identity transforms a process containing application of a vector iterator into a vector of processes, and vice versa. This class of identity does not appear in BMF, since they are needed only because the operational characteristics of vectors and streams differ.

I call this class of identity *horizontal parallelisation*. Two new zipping functions are needed, which zip a vector of streams into a stream of vectors, and vice versa:

$$\begin{aligned}
&\texttt{zipx} &&:: \texttt{Vector}^k\ (\texttt{Stream}^n\ \alpha) \rightarrow \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha)\\
&\texttt{unzipx} &&:: \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha) \rightarrow \texttt{Vector}^k\ (\texttt{Stream}^n\ \alpha)
\end{aligned}$$

The simplest parallelisation identity is this:

$$\begin{aligned}
mapS\ (mapV\ f) &\equiv zipx \cdot mapV\ (mapS\ f) \cdot unzipx\\
&:: \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha) \rightarrow \texttt{Stream}^n\ (\texttt{Vector}^k\ \alpha)
\end{aligned} \tag{5.10}$$

The left-hand side of equation 5.10 is a single process (figure 5.23a); the right-hand side is a vector of k processes (figure 5.23b). Since $zipx \cdot unzipx = id$ and $unzipx \cdot zipx = id$, it is straight-forward to modify this identity as follows:

$$
\begin{aligned}
&\mathit{unzipx} \cdot \mathit{mapS}\ (\mathit{mapV}\ f) \cdot \mathit{zipx} \\
&\quad \equiv \quad \mathit{mapV}\ (\mathit{mapS}\ f) \\
&\quad :: \quad \mathtt{Vector}^k\ (\mathtt{Stream}^n\ \alpha) \rightarrow \mathtt{Vector}^k\ (\mathtt{Stream}^n\ \alpha)
\end{aligned}
\tag{5.11}
$$

Equation 5.11 is probably more useful when starting with a vector of processes, whereas equation 5.10 is more useful when starting with a single process.

Consider now a process that repeatedly applies *foldrV*, such as might be obtained by calculating the sum of each vector in a stream:

$$\mathit{mapS}\ (\mathit{foldrV}\ (+)\ 0)$$

We can parallelise this process in a similar manner:

$$
\begin{aligned}
\mathit{mapS}\ (\mathit{foldrV}\ f\ a) \quad &\equiv \quad \mathit{foldrV}\ (\mathit{zipWithS}\ f)\ (\mathit{repeatS}\ a) \cdot \mathit{unzipx} \\
&:: \quad \mathtt{Stream}^n\ (\mathtt{Vector}^k\ \alpha) \rightarrow \mathtt{Stream}^n\ \alpha
\end{aligned}
\tag{5.12}
$$

Figure 5.23c shows the left-hand side of this identity; figure 5.23d the right-hand side. As for process fusion, horizontal parallelisation quickly leads us to the problem of a large number of combinations of higher-order functions. A technique for automatically deriving identities of this class would be helpful indeed.

5.5.4 Pipelining

The third kind of identity constructs or combines pipelines of processes, where each process is parameterised by a different value. Pipelining was explored extensively by Kelly [78]. The following identity relates a single process to a pipeline of parameterised processes, each constructed with *mapS*:

$$
\begin{aligned}
\mathit{mapS}\ (\mathit{series}\ \mathit{fs}) \quad &\equiv \quad \mathit{series}\ (\mathit{mapV}\ \mathit{mapS}\ \mathit{fs}) \\
&:: \quad \mathtt{Stream}^n\ \alpha \rightarrow \mathtt{Stream}^n\ \alpha
\end{aligned}
\tag{5.13}
$$

Suppose now that *fs* is produced by mapping a function to a vector of parameters, v.

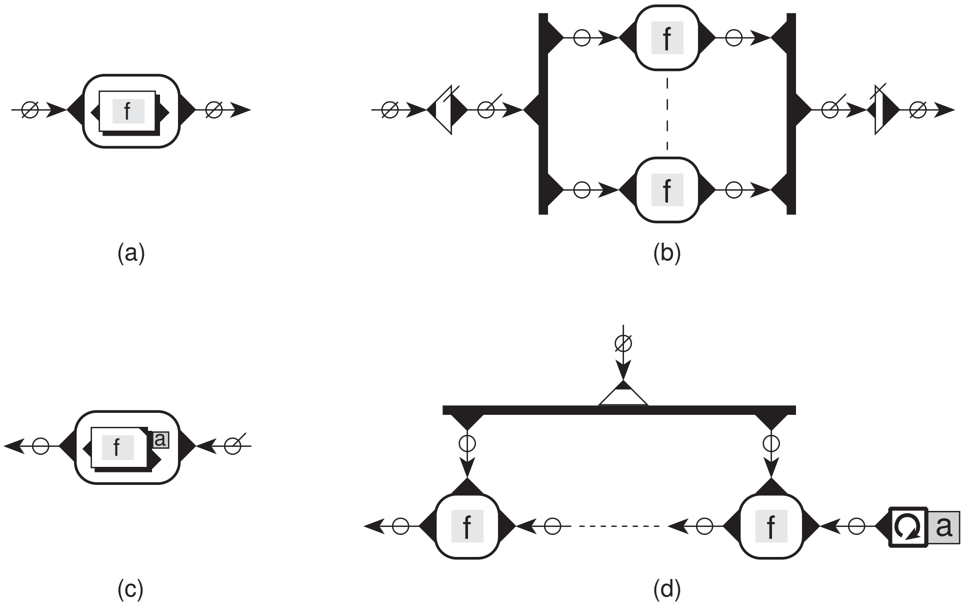


Figure 5.23. Horizontal parallelisation

The right-hand side then becomes

$$series\ (mapV\ mapS\ (mapV\ f\ v))$$

which, by map fusion, becomes

$$series\ (mapV\ (mapS \cdot f)\ v)$$

As we saw by equations 5.2 and 5.4, *foldrV* can be used instead of explicitly parameterising functions or processes. We can rewrite equation 5.13 in the equivalent form:

$$\begin{aligned} mapS\ (\lambda x\,.\,foldrV\ f\ x\ v)\ xs &\equiv foldrV\ (\lambda i\,.\,mapS\ (f\ i))\ xs\ v \\ &:: \mathtt{Stream}^n\ \alpha \end{aligned} \tag{5.14}$$

In this form, this identity is similar to horizontal parallelisation of the *fold* operators, except that the stream feeds into the end of the pipeline instead of a vector of streams feeding across the pipeline. I have used the structure on the left-hand side of equation 5.14 (figure 5.24a) in the FFT example of [114]; it is also similar to the FIR filter example of a second paper devoted to process network transformation [123]. The right-hand side

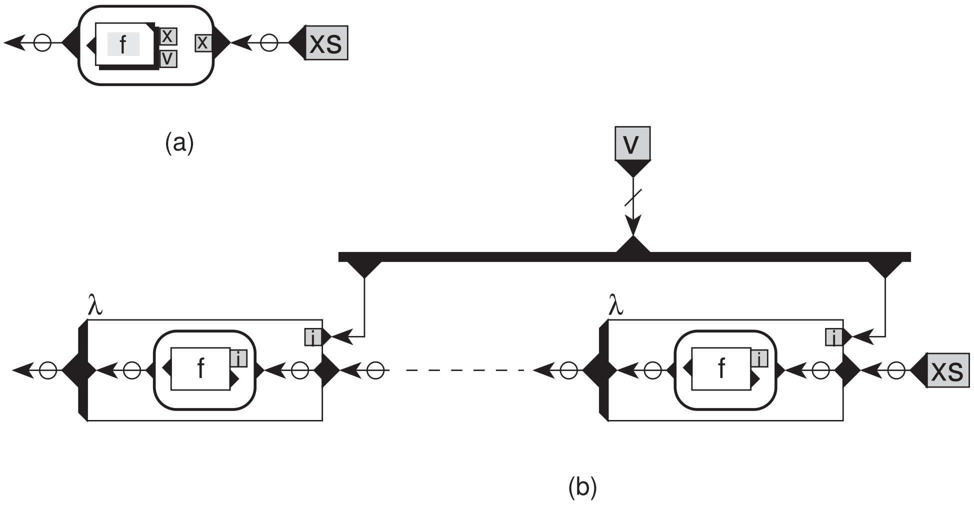


Figure 5.24. Pipelining

(figure 5.24b) is a pipeline of processes, each parameterised by its own i from vector v.

5.5.5 Promotion

The fourth class of identity is derived from Bird's "promotion" identities [21]. The *map promotion* identity expresses the idea that *map* can be "promoted" through concatenation:

$$map\ f \cdot concat \quad \equiv \quad concat \cdot map\ (map\ f)$$

In other words, concatenating a list of lists and applying f to each element is the same as applying f to each element of each sub-list, and concatenating the result.

Adapting this identity from lists to vectors, we get

$$\begin{aligned} mapV\ f \cdot concatV \quad &\equiv \quad concatV \cdot mapV\ (mapV\ f) \\ &:: \quad \texttt{Vector}^j\ (\texttt{Vector}^k\ \alpha) \rightarrow \texttt{Vector}^{j\times k}\ \beta \end{aligned} \tag{5.15}$$

Now, for any function

$$divideV :: \texttt{Vector}^{j\times k}\ \alpha \rightarrow \texttt{Vector}^j\ (\texttt{Vector}^k\ \alpha)$$

that satisfies

$$\begin{aligned} \mathit{concatV} \cdot \mathit{divideV} &= \mathit{id} \\ &:: \ \mathtt{Vector}^k\ \alpha \rightarrow \mathtt{Vector}^k\ \alpha \end{aligned} \tag{5.16}$$

we can rewrite equation 5.15 as

$$\begin{aligned} \mathit{mapV}\ f &\equiv \mathit{concatV} \cdot \mathit{mapV}\ (\mathit{mapV}\ f) \cdot \mathit{divideV} \\ &:: \ \mathtt{Vector}^k\ \alpha \rightarrow \mathtt{Vector}^k\ \beta \end{aligned} \tag{5.17}$$

Equation 5.17 gives us a slightly more useful perspective: to apply f to each element of a vector, divide it into sub-vectors, apply f to each element of each sub-vector, and join the result back into a vector.

With this, we can change the "grain size" of vectors of processes. For example,

$$\begin{aligned} & \mathit{mapV}\ (\mathit{mapS}\ f) \\ = \ & \text{(promotion: equation 5.17)} \\ & \mathit{concatV} \cdot \mathit{mapV}\ (\mathit{mapV}\ (\mathit{mapS}\ f)) \cdot \mathit{divideV} \\ = \ & \text{(horizontal parallelisation: equation 5.11)} \\ & \mathit{concatV} \cdot \mathit{mapV}\ (\mathit{unzipx} \cdot \mathit{mapS}\ (\mathit{mapV}\ f) \cdot \mathit{zipx}) \cdot \mathit{divideV} \end{aligned} \tag{5.18}$$

The result of equation 5.18, shown in figure 5.25, is a vector of processes with the size of each process determined by *divideV*. By adjusting the way in which *divideV* breaks up a vector, we can control the degree of parallelism obtained by this transformation.

A similar law holds for the fold functions; on *foldl*, the law is[3]

$$\mathit{foldl}\ f\ a \cdot \mathit{concat} \equiv \mathit{foldl}\ (\mathit{foldl}\ f)\ a$$

Laws of this kind are useful for controlling the grain size of processes in a pipeline – see the derivation of the pipeline FIR filter in [123] for an example.

[3] Bird [21, page 123] gives this law as a variant of the fold promotion law for non-associative operators.

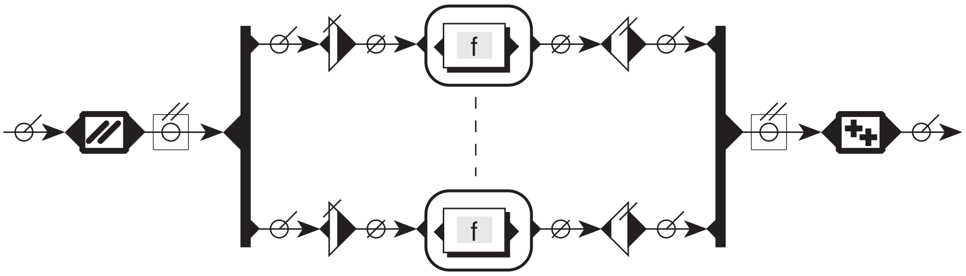


Figure 5.25. Promotion

5.6 Summary

This chapter presented an approach to programming dataflow networks in a functional programming language. I argued that vector functions in a functional language (section 5.2) provide potential for efficient execution on DSP devices, and showed how a small set of stream functions (section 5.3) can be used to write very concise definitions of typical signal processing functions.

Based on these functions, I demonstrated: i) how higher-order functions provide a very powerful notation for building networks of dataflow processes; and ii) that program transformation techniques developed for functional programming languages can be adapted to process networks. In particular, there are four key classes of transformation that have distinct effects on process networks, and I gave examples illustrating the use of these four classes. Note that, although I have not addressed it in this thesis, many similar transformations can be applied to vector functions—for concrete examples, see [123].

In functional languages, program transformation techniques are applied in two ways: i) for hand derivation and optimisation of programs, and ii) as an automatic optimisation tool in functional language compilers. Hand derivation can be quite difficult, as seen, for example, by my derivation of a parallel FIR in [123], and Jones' derivation of the FFT algorithm [67]. I believe the most appropriate path for further development of the transformations presented here would be in the context of a transformational dataflow programming tool: the programmer would choose from a catalog of transformations to be applied by the tool.

Chapter 6

Dynamic Process Networks

In the previous chapter, I considered static, SDF networks expressed in Haskell. The use of SDF networks in block-diagram systems like Ptolemy [88] is well-established, and SDF networks are adequate to describe the bulk of signal processing systems—Ptolemy's predecessor, Gabriel [17], supported only SDF.

As soon as we consider systems which interact with asynchronous events, however, synchronous dataflow is inadequate. For example, consider the "digital gain control" shown in figure 6.1. The gain control signal has a value only when a person changes the position of a gain control knob. In order to produce an appropriate output signal, we must be able represent the *times* of occurrence of gain control values relative to the input signal.

There are two key approaches to modelling time:

- Insert *hiatons* into the stream. A hiaton is a special token representing the passage of time [144].

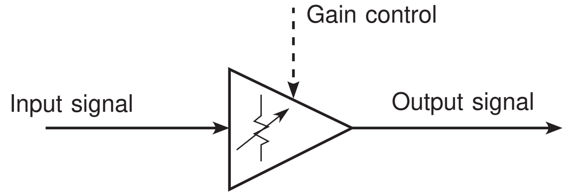


Figure 6.1. A simple digital gain control

– Attach a time-stamp to each token, denoting the real time at which the token occurs [24].

In this chapter, I develop a hybrid approach suitable for real-time implementation: a hiaton can represent any number of ticks of the "base clock" to which all timing of the stream is quantised. This approach avoids having processes spend all their time testing for hiatons; it also prevents unbounded latencies, a potential problem with real-time implementation of time-stamps.

The approach described here contrasts with other approaches to modelling time, in that it maintains a dataflow implementation focus through the use of time-passing tokens. Because it explicitly models time, it is more general than dynamic dataflow systems employing *switch* and *select* actors [26]. It contrasts with the approach taken in the family of "synchronous" (in a different sense to SDF) languages [13], which compile dataflow-like programs into finite-state machines.

Timed streams imply dynamic scheduling; they also give rise to dynamic process networks. (Although dynamic networks can be created using only synchronous stream functions, I have not been able to think of a realistic example.) A complete music synthesiser example provides ample opportunity to explore dynamic process networks: the synthesiser exhibits a very high level of asynchronous behaviour.

6.1 Related work

Prior to the development of SDF scheduling, block-diagram oriented simulation systems used dynamic actor scheduling. Messerschmitt's BLOSIM simulator [94], for example, fired actors in round-robin fashion; an actor that had insufficient input data was required to return to the simulator immediately.

A general approach to supporting multiple models of computation has been taken in Ptolemy: a *domain* is a graph that supports a given model of computation; this graph has an interface that allows it to be embedded within graphs of a different domain [25]. The "dynamic dataflow" (DDF) domain [42] implements dynamic scheduling of dataflow graphs. In the DDF domain, input channels to the graph are initialised with a fixed number of tokens, and the actors fired until no further firings can be made. This allows a DDF domain to be embedded within an SDF domain. An entirely different approach to time is taken in the discrete-event (DE) domain [42]. Here, actors operate on timed *events*; a global clock is used to order actor firings correctly in time, according to the times of occurrence of the actor's input events.

None of these approaches is, however, really suitable for figure 6.1. Pino *et al* [108] have recently developed the notion of "peek" and "poke" actors. These actors act as the interface between two independently-scheduled graphs, running on two different processors. For example, the multiplier in figure 6.1 would execute on a real-time DSP device, while the control signal would be written to a shared memory location by a control CPU. Because the reads and writes are unsynchronised, the amplitude of the output signal will change as expected provided that the latency of the control CPU processing is undetectable to a human listener.

The approach I develop in this chapter explicitly represents time. It thus has the advantage of precision over Pino *et al*'s method, although it will be more complex to implement. Other languages that explicitly model time include real-time extensions to Lucid, and the synchronous languages. Lucid [144] is a dataflow-like language, in which all data—even constants—occurs in streams. Faustini and Lewis [45] propose that each stream also have an associated stream of time windows, where a time window is a pair (a, b) that denotes the earliest time at which the corresponding daton can be produced, and the latest time it needs to be available. Skillicorn and Glasgow [131] use a similar approach: they associate *earliest* and *latest* time streams with each data stream, and construct isomorphic nets operating on these streams. From the earliest input streams, the earliest time at which any token is produced is calculated; from the latest output streams, the latest time at which any token can be produced is calculated.

The synchronous languages [13] also explicitly model time. They are based on the *synchrony hypothesis*: each reaction to an event is supposed to be instantaneous [15]. Inter-process communication in Esterel, for example, is done by broadcasting events to all processes; the broadcast and all responses to it are instantaneous. This simplifying assumption allows a precise specification of time semantics and compilation of programs into finite-state automata. The family of synchronous languages includes Esterel [15], Lustre [33], Signal [14], and RLucid, a version of Lucid extended with time-stamps [109].

6.2 Timed signals and streams

In section 2.4.1, a non-uniformly clocked signal $\widetilde{x}$ was defined at times given by its *clock*, $\widetilde{t_x}$. I will drop the $\widetilde{x}$ notation now, as it will be clear from context whether a signal is uniformly or non-uniformly clocked.

The clock t_x is *quantised* to a *base clock* $b_x = \{nT \mid n \geq 0\}$ for constant T. t_x thus only has values in b_x:

$$t_x(n) \in b_x, \qquad n \geq 0$$

If the signal is a sampled version of an analog signal x_a, then

$$x(n) = x_a(t_x(n)), \qquad n \geq 0$$

Because I am working within the framework of an existing language, I cannot add a time semantics to the language, as I could if starting afresh. Instead, I make time information explicit—that is, time is just data. Tokens that carry time information are called *hiatons*: $\triangle_n$ is a hiaton that denotes n ticks of the base clock, and $\triangle$ is a unit hiaton, which denotes a single tick. Tokens that carry signal data are called *datons*. A stream carrying datons and hiatons is a *timed stream*; here is an example:

$$\{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle$$

where the $\triangle$ subscript indicates a timed stream (as opposed to a synchronous stream).

The Haskell type definition for tokens is:

```
data Token α = Hiaton Int           —Hiaton
             | Daton α              —Single-value daton
             | Datons <Token α>     —Timed vector
             | Block <α>            —Synchronous vector
```

Unit datons, denoted by the *Daton* tag, occupy a single tick of the base clock; the datons in the example stream above are unit datons. A stream containing only hiatons and unit datons is called a *simple* stream. There are two other kinds of daton: a *timed vector* is a vector of hiatons and unit datons, written $\langle 4, \triangle_2, 7\rangle_\triangle$; a *synchronous vector* is a vector of data, each element of which is equivalent to a unit daton. The *duration* of a token is the number of base-clock ticks it occupies, and is given by the *duration* function:

```
duration :: Token α → Int
duration (Hiaton n) = n
duration (Daton x)  = 1
duration (Datons v) = sumV (mapV duration v)
duration (Block v)  = lengthV v
```

A timed stream is a stream of tokens. *NullT* is the stream terminator (equivalent to $\{\}$ on synchronous streams); `:-:` is the timed stream constructor:

```
data Timed α = NullT
             | Token α :-: (Timed α)
```

In an earlier paper [117], I used two representations of passing time: time-stamps, and unit hiatons. The problem with unit hiatons is that they imply that processes will do a great deal of work just marking time—that is, processing hiatons. Time-stamps, on the other hand, have indeterminate latency—some functions may not produce output until an indeterminate time after they have read the required input. This is unacceptable for real-time operation, and is the reason why I have adopted a hybrid approach.

Consider a timed *merge* function: this function produces a token when it receives a token on either input, but takes account of token durations so that the output token appears at the correct tick. Datons are passed to the output; hiatons result in a new hiaton (of possibly different duration) on the output. Because the arrival time of tokens is not synchronised to the token durations, timed merge cannot produce an output hiaton until the next input token is received—only then is the duration of the output hiaton known. If the duration of input hiatons is unbounded—as for time-stamped data—then the time between receiving an input daton and producing the following output hiaton is also unbounded.

For real-time operation, then, the duration of all tokens must be bounded by some limit, say L. Suppose that $L = 32$. The stream $\{1, \triangle_{74}, 2\}_{\triangle}$ does not have bounded latency, but the stream $\{1, \triangle_{32}, \triangle_{32}, \triangle_{10}, 2\}_{\triangle}$ is. A process that produces a real-time stream must ensure that it meets this requirement; and a process that consumes a real-time stream must ensure that its output streams have latency no greater than that of the input stream.

$$
\begin{array}{lll}
\texttt{(:-:)} & :: & \texttt{Token}\ \alpha \rightarrow \texttt{Timed}\ \alpha \rightarrow \texttt{Timed}\ \alpha \\
\\
\texttt{tmapT} & :: & (\alpha \rightarrow \texttt{Token}\ \beta) \rightarrow (\texttt{Int} \rightarrow \texttt{Token}\ \beta) \rightarrow \texttt{Timed}\ \alpha \rightarrow \texttt{Timed}\ \beta \\
\texttt{zipT} & :: & \texttt{Timed}\ \alpha \rightarrow \texttt{Timed}\ \beta \rightarrow \texttt{Timed}\ (\texttt{Token}\ \alpha,\ \texttt{Token}\ \beta) \\
\\
\texttt{tstateT} & :: & (\alpha \rightarrow \beta \rightarrow (\alpha,\ \texttt{Token}\ \gamma)) \rightarrow (\alpha \rightarrow \texttt{Int} \rightarrow (\alpha,\ \texttt{Token}\ \gamma)) \\
 & & \quad \rightarrow \alpha \rightarrow \texttt{Timed}\ \beta \rightarrow \texttt{Timed}\ \gamma \\
\\
\texttt{groupT} & :: & \texttt{Int} \rightarrow \texttt{Timed}\ \alpha \rightarrow \texttt{Timed}\ \alpha \\
\texttt{concatT} & :: & \texttt{Timed}\ \alpha \rightarrow \texttt{Timed}\ \alpha \\
\\
\texttt{concatvT} & :: & \texttt{Timed}\ \alpha \rightarrow \texttt{Stream}\ \alpha \\
\texttt{timeT} & :: & \texttt{Stream}\ \alpha \rightarrow \texttt{Timed}\ \alpha
\end{array}
$$

Figure 6.2. Types of timed stream functions

6.3 Functions on timed streams

This section presents a set of "primitive" functions on timed streams, as I did for synchronous streams in section 5.3.2. Again, the aim is to provide a set of functions with known dataflow actor equivalents, so that a process network can be translated into a dataflow network. Because, however, the variety of operations on timed streams is much greater than on synchronous streams, I cannot claim that these functions are complete in any way. Nor have I provided dataflow actor equivalents for these functions, as further work is required to find the most useful set of functions. The types of the functions presented in this section are listed in figure 6.2. Recursive definitions are given in appendix A.

6.3.1 Basic functions

The first three functions are similar to *mapS*, *zipS*, and *stateS* on synchronous streams. Because of the presence of hiatons, they are necessarily more complex. All three accept simple timed streams—that is, streams containing only hiatons and unit datons.

tmapT has two function arguments: the first is applied to the value of each unit daton, the second to the duration of each hiaton. Both produce a token—a hiaton or any kind of daton is acceptable. Using *tmapT*, we can define a version of *map* that only operates on unit datons and ignores hiatons:

```
mapT :: (α → β) → Timed α → Timed β
mapT f = tmapT (Daton . f) Hiaton
```

For example,

$$mapT\ \ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{2, \triangle, 4, 5, \triangle_2, 8\}_\triangle$$

tmapT can also be used to define *filterT*, which replaces unit datons that do not satisfy a predicate with unit hiatons:

```
filterT :: (α → Bool) → Timed α → Timed α
filterT p = tmapT
                (\x -> if p x then Daton x else Hiaton 1)
                Hiaton
```

The second function, *zipT* zips two timed streams into a timed stream of pairs; this stream contains a pair whenever either of its two input streams contains a unit daton. Because only one of the streams can contain a daton, one element of the pair can be a daton while the other is a unit hiaton. For example,

$$zipT\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle\ \{1, \triangle_4, 6, \triangle\}_\triangle \quad \rightarrow \quad \{(1,1), \triangle, (3,\triangle), (4,\triangle), \triangle, (\triangle, 6), (7, \triangle)\}_\triangle$$

With *zipT*, it is easy to define functions that operate on two timed streams. The first is the unfair timed merge discussed in section 6.2. The unfair merge outputs only the value from its first input if datons occur on both input streams at the same time:

```
mergeT :: Timed α → Timed α → Timed α
mergeT xs ys = tmapT merge Hiaton (zipT xs ys)
    where
    merge (Daton x, Daton y)  = Daton x
    merge (Daton x, Hiaton 1) = Daton x
    merge (Hiaton 1, Daton y) = Daton y
```

An operation found in synchronous languages with time semantics produces elements of one signal (the "data" signal) at the times at which elements of another signal (the "clock" signal) occur. In the Haskell version, *whenT*, a unit hiaton is output if the data signal has no value when the clock contains data. For example,

$$\{8, \triangle_2, 5, 4, \triangle, 2, 1\}_\triangle \text{ `whenT` } \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{8, \triangle, \triangle, 5, \triangle_2, 2\}_\triangle$$

The definition of *whenT* is very similar to that of the unfair merge:

```
whenT :: Timed α → Timed β → Timed α
whenT xs ys = tmapT when Hiaton (zipT xs ys)
    where
    when (Daton x, Daton y)  = Daton  x
    when (Daton x, Hiaton 1) = Hiaton 1
    when (Hiaton 1, Daton y) = Hiaton 1
```

tstateT is the final primitive of this first group; it propagates a "state" value along the stream. *tstateT* takes two function arguments: one applied to each daton value, and one to the duration of each hiaton. Both functions also accept a "state" argument, and both produce a pair containing an output token and the next state. For example, to replace each daton with the number of ticks since the previous daton occurred, propagate a counter as the state and output it when a daton occurs:

```
deltaT :: Timed α → Timed Int
deltaT = tstateT f g 1
    where
    f a x = (1,   Daton a)
    g a n = (a+n, Hiaton n)
```

For example,

$$deltaT\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{1, \triangle, 2, 1, \triangle_2, 3\}_\triangle$$

6.3.2 Timed vectors

A timed vector is a vector of tokens, marked in a timed stream by the tag *Datons*. There are two primitives for working with streams containing timed vectors: *groupT* and *concatT*. These functions are analogous to *groupS* and *concatS* on synchronous streams.

groupT produces a timed vector stream from a simple timed stream. Unlike *groupS*, the last vector may have a different duration to the rest. For example,

$$groupT\ \ 2\ \ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{\langle 1, \triangle\rangle_\triangle, \langle 3, 4\rangle_\triangle, \triangle_2, \langle 7\rangle_\triangle\}_\triangle$$

concatT joins a stream of hiatons, unit datons, and timed vectors back into a timed stream. To illustrate, let us use *concatT* to up-sample a simple timed stream. In the following definition, *tmapT* is used to replace each daton by a timed vector containing the daton and a hiaton, and to multiply the duration of each hiaton by the up-sampling factor. *concatT* joins the timed vector stream back into a timed stream.

```
upsampleT :: Int → Timed α → Timed α
upsampleT k = concatT . tmapT f g
    where
    f x = Datons ⟨Daton x, Hiaton (k-1)⟩
    g n = Hiaton (k*n)
```

For example,

$$upsampleT\ 2\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle\ \ \rightarrow\ \ \{1, \triangle, \triangle_2, 3, \triangle, 4, \triangle, \triangle_4, 7, \triangle\}_\triangle$$

A more complex function using *concatT* "retimes" a simple timed stream. This is useful for reducing the number of hiatons in streams produced by *filterT*. The first argument is the latency of the output stream; the function assumes that the latency of the input stream is no greater than this value. It works by propagating a count of the number of ticks since the last daton was output, and emitting a hiaton when a daton is encountered or the count reaches the latency:

```
retimeT :: Int → Timed α → Timed α
retimeT k = concatT . tstateT f g 0
    where
    f 0 x = (0, Daton x)
    f i x = (0, Datons ⟨Hiaton i, Daton x⟩)
    g i n | i+n < k   = (i+n,   Datons ⟨⟩)
          | otherwise = (i+n-k, Hiaton k)
```

For example,

$$\mathit{retimeT}\ 2\ \{1, \triangle, 3, 4, \triangle, \triangle, 7\}_\triangle \ \rightarrow\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle$$

The requirement that input latency be no greater than the output latency makes sense when real-time effects are considered. If the input stream has some latency L, then the timing of the output tokens will still, in effect, exhibit a delay of L; just because the output hiatons have a lower maximum duration does not make them advance in time. For the sake of following examples, assume there is a global variable named *latency* with value two.

6.3.3 Synchronous and timed streams

The last two functions of figure 6.2 operate on both timed and synchronous streams. *concatvT* concatenates a vector containing only unit datons and synchronous vectors into a synchronous stream. Two useful functions defined using *concatvT* are *fillT*, which inserts a specified value into empty slots of a stream, and *holdT*, which inserts the most recent daton value into empty slots:

```
fillT :: α → Timed α → Stream α
fillT a = concatvT . tmapT f g
    where
    f x = Daton x
    g n = Block (copyV n a)

holdT :: α → Timed α → Stream α
holdT a = concatvT . tstateT f g a
    where
    f v x = (x, Daton x)
    g v n = (v, Block (copyV n v))
```

For example,

$$\mathit{fillT}\ 0\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{1, 0, 3, 4, 0, 0, 7\}$$

$$\mathit{holdT}\ 0\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{1, 1, 3, 4, 4, 4, 7\}$$

The introduction to this chapter gave a simple example of a timed user control over a real-time signal processing function. The "digital gain control" is easily defined by:

```
digitalGain :: Num α ⇒ Stream α → Timed α → Stream α
digitalGain signal control
    = zipWithS (*) signal (holdT zero control)
```

This simple idea can be used to convert any kind of real-time process into one with dynamic user control.

timeT converts a synchronous stream to a timed stream by adding *Daton* tags. For example, we could use it to convert a synchronous boolean signal into a "clock" stream, defined only when the original stream contains the value *True*:

```
clockT :: Stream Bool → Timed Bool
clockT = retimeT latency . filterT (== True) . timeT
```

Note that the clock signal is retimed to *latency*. One use for timed streams may be to reduce the data rate of a signal. To do so, define a function that outputs a daton only when the input value changes. This function, *edges*, is defined in terms of *clockT* and *whenT*; its first argument is the starting value with which it compares the first value in the stream:

```
edges :: Eq α ⇒ α → Stream α → Timed α
edges a xs = retimeT latency
               ((timeT xs) `whenT` (clockT (zipWithS (/=) xs (a:-xs))))
```

For example,

$$\mathit{edges}\ 0\ \{1,1,3,4,4,4,7,7\} \rightarrow \{1,\triangle,3,4,\triangle_2,7,\triangle\}_\triangle$$

edges can be used with *deltaT* to find the distance between zero-crossings of a signal:

$$\begin{aligned}&(\mathit{deltaT} \cdot \mathit{edges}\ \ \mathit{True} \cdot \mathit{mapS}\ (\geq 0))\ \{0,1,2,-1,-1,-2,1,-1\}\\ &\quad\rightarrow\ \ \{\triangle_2,\triangle,4,\triangle_2,3,1\}_\triangle\end{aligned}$$

Suppose we wish to encode and transmit a signal as straight-line segments. The encoding produces as its first value the first value of the signal, as its second the slope of the first segment, and thereafter, the slope of the signal whenever it changes:

```
encode :: Num α ⇒ Stream α → Timed α
encode xs = edges huge (zipWithS (-) xs (zero:-xs))
    where
    huge = fromInteger (-100000000)
```

where *huge* is some value that cannot appear in the input stream.

For example,

$$\mathit{encode}\ \{0, 1, 2, 2, 2, 0, -2, -3, -3, -3, -3\} \quad \rightarrow \quad \{0, 1, \triangle, 0, \triangle, -2, \triangle, -1, 0, \triangle_2\}_\triangle$$

To reconstruct the signal, do a zero-order interpolation and then integrate:

```
decode :: Num α ⇒ Timed α → Stream α
decode = scanS (+) zero . holdT zero
```

For example,

$$\mathit{decode}\ \{0, 1, \triangle, 0, \triangle, -2, \triangle, -1, 0, \triangle_2\}_\triangle \quad \rightarrow \quad \{0, 1, 2, 2, 2, 0, -2, -3, -3, -3, -3\}$$

6.4 Dynamic process networks

Dynamic process networks arise when streams are "nested." Typically, synchronous streams are nested within timed streams—that is, each daton of the timed stream creates one or more processes that produce a new synchronous stream. This new stream is produced concurrently with other streams already initiated by the timed stream. After some time, the synchronous streams end, and the process or processes producing it disappear. This section describes primitive functions that make finite synchronous streams, "spawn" new processes, and combine streams. The types of these functions are listed in figure 6.3.

6.4.1 Finite synchronous streams

The first four functions of figure 6.3 are used to create finite synchronous streams. As the example in section 6.5 will show, synchronous streams are often created by taking an initial prefix of one stream, following it with another stream, and so on. Consider how we might do this with Haskell standard prelude functions on lists (described in section 2.2.1).

$$
\begin{array}{lll}
\texttt{doneS} & :: & \texttt{Generator}\ \alpha \\
\texttt{truncateS} & :: & \texttt{Stream}\ \alpha \rightarrow \texttt{Stream}\ \beta \rightarrow \texttt{Generator}\ \beta \rightarrow \texttt{Stream}\ \beta \\
\\
\texttt{takeFiniteS} & :: & (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha \rightarrow \texttt{Bool}) \rightarrow \beta \rightarrow \alpha \\
 & & \quad \rightarrow \texttt{Stream}\ \alpha \rightarrow \texttt{Generator}\ \alpha \rightarrow \texttt{Stream}\ \alpha \\
\texttt{takeAsLongS} & :: & (\alpha \rightarrow \texttt{Bool}) \rightarrow \texttt{Timed}\ \alpha \rightarrow \texttt{Stream}\ \beta \\
 & & \quad \rightarrow \texttt{Generator}\ \beta \rightarrow \texttt{Stream}\ \beta
\end{array}
$$

Figure 6.3. Types of finite stream functions

To define a function that produces a stream containing the first five elements of a list followed by an infinite number of copies of the fifth element, we write:

```
fiver xs = let ys = take 5 xs
           in  ys ++ repeat (last ys)
```

For example,

$$\mathit{fiver}\ [1,2,3,4,5,6,7,8,9,10] \rightarrow [1,2,3,4,5,5,5,\ldots]$$

If we interpret the functions used in *fiver* as processes, there are four: *take*, $++$, *repeat*, and *last*. $++$ and *last* do almost no work: $++$ outputs elements of one stream until it terminates, and then outputs elements of a second stream; *last* just waits for a stream to terminate, then outputs the last element.

Because a finite stream often has a second stream appended to it, I have defined functions on streams to take an additional function argument, of type $\alpha \rightarrow$ `Stream` α. A function of this type is called a *generator*; a Haskell type synonym makes it easier to recognise:

$$\texttt{type Generator}\ \alpha\ \texttt{=}\ \alpha \rightarrow \texttt{Stream}\ \alpha$$

For example, the stream version of *take*, *takeS*, outputs a given number of elements from its input stream; it then applies the generator to the last of these elements. Used this way, the generator is called a *stream continuation*. The generator becomes a new process which produces further elements into the output stream. If *takeS* is unable to produce any output elements (for example, the input stream is empty or $k = 0$), there is no element to which to apply the continuation. To work around this, *takeS* also accepts a value to be passed to the continuation.

For example, the above example, translated to synchronous streams, becomes

```
fiver xs = takeS 5 0 xs repeatS
```

which requires only two processes.

To terminate the output stream, use the *doneS* generator. *doneS* takes a single element, ignores it, and produces an empty stream. For example, to take the first five elements of a synchronous stream xs and terminate:

$$\mathit{takeS}\ 5\ 0\ \{1,2,3,4,5,6,7,8,9,10\}\ \mathit{doneS}\ \rightarrow \{1,2,3,4,5\}$$

takeS is defined in terms of a more complex primitive function, *takeFiniteS*; the definition of *takeS* and that of additional finite stream functions are listed in figure 6.4. *takeFiniteS* accepts the following arguments:

- A function from a state and the current input value to the next state.
- A predicate from a state and the current input value; the stream continuation is applied to the last output element when the predicate returns *True*.
- The initial state.
- The argument to the stream continuation if no output elements were produced.
- The input stream.
- The stream continuation.

takeWhileS is also defined in terms of *takeFiniteS*; it stops reading its input stream when a predicate fails. For example,

$$\mathit{takeWhileS}\ (<4)\ 0\ \{1,2,3,4,5,6,7\}\ \mathit{doneS} \rightarrow \{1,2,3\}$$

truncateS is a primitive; it stops reading its second argument stream when its first argument stream terminates. For example,

$$\mathit{truncateS}\ \{1,2,3,4\}\ \{a,b,c,d,e,f\}\ \mathit{doneS}\ \rightarrow \{a,b,c,d\}$$

```
takeS :: Int -> a -> Stream a -> Generator a -> Stream a
takeS = takeFiniteS (\n _ -> n-1) (\n _ -> n > 0)

takeWhileS :: (a -> Bool) -> a -> Stream a -> Generator a-> Stream a
takeWhileS p = takeFiniteS (\_ _ -> ()) (\_ x -> p x) ()

mapWhileS :: (a -> Bool) -> (a -> b) -> a
                -> Stream a -> Generator b -> Stream b
mapWhileS p f a xs = truncateS
                        (takeWhileS p a xs doneS)
                        (mapS f xs)

takeWhileDeltaS :: Num a => (a -> Bool) -> a
                   -> Stream a -> Generator a -> Stream a
takeWhileDeltaS p a (x:-xs) c
    = x :- truncateS
               (takeWhileS p a (zipWithS (-) xs (x:-xs)) doneS)
               xs
               c
```

Figure 6.4. More functions for making finite streams

With *truncateS*, we can define a few more finite stream functions, which will be used in the synthesiser example. *mapWhileS* (figure 6.4) applies a function to elements of its input stream; it stops reading the input stream and calls the continuation when a supplied predicate fails. For example,

$$\mathit{mapWhileS}\ (<4)\ (+1)\ 0\ \{1,2,3,4,5,6\}\ \mathit{repeatS} \quad \rightarrow \quad \{2,3,4,4,4,4,\ldots\}$$

takeWhileDeltaS is like *takeWhileS*, but stops reading the input stream and calls the continuation when the *difference* between successive elements fails to satisfy the supplied predicate. For example,

$$\mathit{takeWhileDeltaS}\ (<2)\ 0\ \{1,2,3,5,7,9\}\ \mathit{doneS} \quad \rightarrow \quad \{1,2,3\}$$

The final primitive in this group is *takeAsLongS*. This function stops reading a synchronous stream when a daton of a timed stream fails a supplied predicate. For example,

$$\mathit{takeAsLongS}\ (<6)\ \{1,\triangle_2,4,\triangle_2,7,\triangle\}_\triangle\ \{1,2,3,4,5,6,7,8\}\ \mathit{doneS} \quad \rightarrow \quad \{1,2,3,4,5,6\}$$

spawnT	::	$(\alpha \rightarrow$ Bool$) \rightarrow$ Timed $\alpha \rightarrow$ Timed (Timed α)
combineT	::	(Vector $\alpha \rightarrow \beta) \rightarrow$ Timed (Stream α) $\rightarrow$ Stream β

Figure 6.5. Types of dynamic process functions

6.4.2 Dynamic process functions

The last two primitive functions given in this section "spawn" new streams, and combine synchronous streams. Their types are given in figure 6.5. *spawnT* is the key to building dynamic process networks. For each daton that satisfies a given predicate, *spawnT* creates a new daton that is its argument stream from that daton on. Datons that do not satisfy the predicate result in a unit hiaton. For example,

$$spawnT\ odd\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \quad \rightarrow \quad \{\{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle, \triangle, \{3, 4, \triangle_2, 7\}_\triangle, \triangle, \triangle_2, \{7\}_\triangle\}_\triangle$$

Each stream within the top-level stream can produce new processes. Let us define a function which produces, for each odd daton in a timed stream, a synchronous stream containing 3 copies of that daton:

```
odder :: Num α ⇒ Timed α → Timed (Stream α)
odder xs = mapT proc (spawnT odd xs)
    where
    proc (Daton x :-: xs) = takeS 3 (repeatS x) doneS
```

For example,

$$odder\ \{1, \triangle, 3, 4, \triangle_2, 7\}_\triangle \rightarrow \{\{1, 1, 1\}, \triangle, \{3, 3, 3\}, \triangle, \triangle_2, \{7, 7, 7\}\}_\triangle$$

If we line these streams up to show the times at which elements are produced, we get

```
Time        1   2   3   4   5   6   7   8   9
            ---------------------------------
Values      1   1   1
                    3   3   3
                                    7   7   7
```

A nested stream of synchronous streams can be combined into a single synchronous stream with *combineT*. *combineT* accepts a function which is applied to a vector of values produced in that tick by all existing synchronous sub-streams; if there are no streams producing data at that time, this function will be applied to an empty vector. For example,

$$\mathit{combineT}\ \mathit{sumV}\ \{\{1,1,1\},\triangle,\{3,3,3\},\triangle,\triangle_2,\{7,7,7\}\}_\triangle \quad\rightarrow\quad \{1,1,4,3,3,0,7,7,7\}$$

6.5 A digital music synthesiser

This section illustrates the use of dynamic process creation to generate musical tones. The synthesiser receives a timed stream of tokens that represent "note-on" and "note-off" events; on each note-on event, it creates new processes to generate a note of the specified frequency. Eventually, the note decays to nothing, and the processes producing that note disappear.

There is a great deal of dynamism in this example, and readers familiar with the strict real-time constraints of DSP may doubt that it is possible to implement a program like this in real time. That it is possible has been shown by the implementation of a polyphonic synthesiser in C on a single TMS320C30, by Michael Colefax [36]. Colefax' program implements a synthesiser with a similar (although simpler) structure to that of the following Haskell program. It can generate up to 17 notes when generating sine waves, but only two with formant-wave-function (FWF) synthesis (section 6.5.4). With re-coding of selected parts of the FWF algorithm in assembler, considerably more notes should be achievable.

6.5.1 Notes

The input to the synthesiser is a timed stream of notes. The *NoteEvent* type and associated types and functions are listed in figure 6.6. This stream is typically produced by an interface to a music keyboard: when a key is pressed, a *NoteOn* daton is generated, containing the note ID, its frequency, and an amplitude value reflecting how hard the key was struck; when the note is released, a *NoteOff* daton is generated, containing the note ID. Some time after the note-off event, the note decays to nothing; a note thus exists from the time that its note-on event occurs until some time a corresponding note-off event

```
type Frequency = Float
type Amplitude = Float
type NoteID    = Int
type Envelope  = [(Float, Float)]

data NoteEvent = NoteOn NoteID Frequency Amplitude
               | NoteOff NoteID

isNoteOn (NoteOn _ _ _)      = True
isNoteOn (NoteOff _)         = False

isNoteOff noteid (NoteOff nid)
    | noteid == nid          = True
    | otherwise              = False
isNoteOff noteid _           = False

sampleRate :: Float
sampleRate = 32000.0
```

Figure 6.6. Note events and associated code

occurs. This is a polyphonic synthesiser, so an arbitrary number of notes can be active simultaneously.

The note event stream I will use in this example is:

$$\begin{aligned} notes \quad = \quad & \{\triangle_4,\ \mathit{NoteOn}\ 7\ 3000.0\ 0.25,\ \triangle_{15},\ \mathit{NoteOn}\ 3\ 5000.0\ 0.4, \\ & \triangle_{14},\ \mathit{NoteOff}\ 3,\ \triangle_{25},\ \mathit{NoteOff}\ 7, \\ & \triangle_{18},\ \mathit{NoteOn}\ 11\ 6000.0\ 0.1,\ \triangle_9,\ \mathit{NoteOff}\ 11,\ \triangle_{24}\}_\triangle \end{aligned}$$

The sample rate of the synthesiser is 32,000 Hz.

6.5.2 Envelopes

When a note-on event is received, the synthesiser starts generating a note of the appropriate frequency. The amplitude of the note is controlled by an "envelope," which is a series of straight-line segments, sampled at $1/T$. An envelope is specified as a list of (slope, target) pairs; when each segment is started, it proceeds towards the target value at the value indicated by the slope. A slope of zero indicates that the current value is to be held

```
ramp :: Float -> Generator Float
ramp slope = tailS . iterateS (+ (slope/sampleRate))

rampTo :: Float -> Float -> Float -> Generator Float -> Stream Float
rampTo start slope target cont
    | slope < 0.0 = takeWhileS (>= target) start (ramp slope start) cont
    | slope > 0.0 = takeWhileS (<= target) start (ramp slope start) cont
    | otherwise   = repeatS start

ramps :: Envelope -> Generator Float
ramps ((slope, target) : rest) start = rampTo start slope target (ramps rest)
ramps _ _                            = NullS

envelope :: Timed NoteEvent -> Envelope -> Envelope -> Stream Float
envelope notes@((Daton (NoteOn noteid _ _)):-:_) a d
    = attack decay
      where
      attack  = takeAsLongS (not . isNoteOff noteid)
                             notes
                            (0.0 :- ramps a 0.0)
      decay x = ramps d x
```

Figure 6.7. Ramp and envelope generators

indefinitely. The function *rampTo* of figure 6.7 generates a synchronous stream for a single segment of the envelope; the function *ramps* recursively processes a list of (slope,target) pairs to join a series of segment streams end-to-end.

A typical note-on envelope is an "attack-decay-sustain" envelope, such as

$$attack = [(6400.0, 1.01), (-3200.0, 0.49), (0.0, 0.0)]$$

When a note-off event is received, the note envelope decays from its current value to zero:

$$decay = [(-1000.0, 0.0)]$$

The complete note envelope is its attack envelope followed by its decay envelope. Function *envelope* of figure 6.7 generates the complete note envelope: it starts an envelope on a note-on event, and uses *takeAsLongS* to switch to the decay envelope when the corresponding note-off event occurs in the note event stream. The amplitude of each

```
siner     :: Generator Float
siner freq = mapS sin (phaser 0.0 freq)

phaser          :: Float -> Float -> Stream Float
phaser init freq = iterateS next init
    where
    next phase = fmod (phase + phinc) (2.0 * pi)
    fmod x m   = if x > m then x - m else x
    phinc      = (2.0 * pi * freq) / sampleRate

flatline     :: Generator Float
flatline freq = repeatS one

notegen :: Envelope -> Envelope -> Generator Float
                                -> Timed NoteEvent
                                -> Stream Float
notegen on off gen notes@((Daton (NoteOn _ freq ampl)) :-: ns)
    = mapS (* ampl) (zipWithS (*)
                      (envelope notes on off)
                      (gen freq))
```

Figure 6.8. Note generators

note's envelope is scaled by the amplitude parameter in the note-on event. The dashed line in figure 6.10 shows the superimposed envelopes of the above note event stream.

6.5.3 Note generation

Figure 6.8 lists the code associated with the generation of a single note. *siner* is a sine wave generator: given a frequency parameter, it generates an infinite sine wave of that frequency. *flatline* is used for testing envelope generators: it generates an infinite stream of ones.

The *notegen* function accepts an attack and decay envelope, a waveform generator (such as *siner*), and the note-event stream that starts with the current note. It uses *envelope* to generate the note envelope, the waveform generator to generate the note's waveform, and multiplies the two. It also scales the waveform by the note's amplitude. The note terminates when the envelope terminates.

The complete synthesiser is a pipeline of three stages—its code is given in figure 6.9. Its structure is identical to the dynamic process example on page 181. Each note-on event

```
synthesiser :: Envelope -> Envelope -> Generator Float
                        -> Timed NoteEvent -> Stream Float

synthesiser on off generate = combineT sumV
                              . mapT (notegen on off generate)
                              . spawnT isNoteOn
```

Figure 6.9. The top-level synthesiser

produces a new timed stream; for each of these, a note generator process is created to produce the synchronous stream corresponding to that note; finally, all of these synchronous streams are summed into a single synchronous stream.

The waveform generator used by the synthesiser is passed to it as the argument *generate*. Figure 6.10 shows the result of evaluating the expression

$$synthesiser\ attack\ decay\ siner\ notes$$

The dashed line in the figure is the sum of the note envelopes, produced by

$$synthesiser\ attack\ decay\ flatline\ notes$$

6.5.4 A formant-wave-function note generator

Although useful for testing, sine waves are musically uninteresting. Formant-wave function (FWF) synthesis is a time-domain method for simulating the excitation-pulse + resonant-filter model of sound synthesis [125]. For music synthesis, the FWF algorithm is relatively simple to implement, as demonstrated by the implementation of a monophonic synthesiser in assembler on a TMS320C25 fixed-point DSP chip [120]. Parameters of the algorithm have a direct relation to the shape of the frequency spectrum, which is helpful for programming musical sounds, and it can produce rich and interesting sounds.

A *formant* is a peak in the spectrum of a musical note, and is typical of instruments based on an excitation pulse and a resonant cavity, including brass, woodwind, organ, and the human voice. An impulse passed through a resonant cavity with a single spectral peak produces a sine wave with an exponentially decaying envelope; in the FWF algorithm,

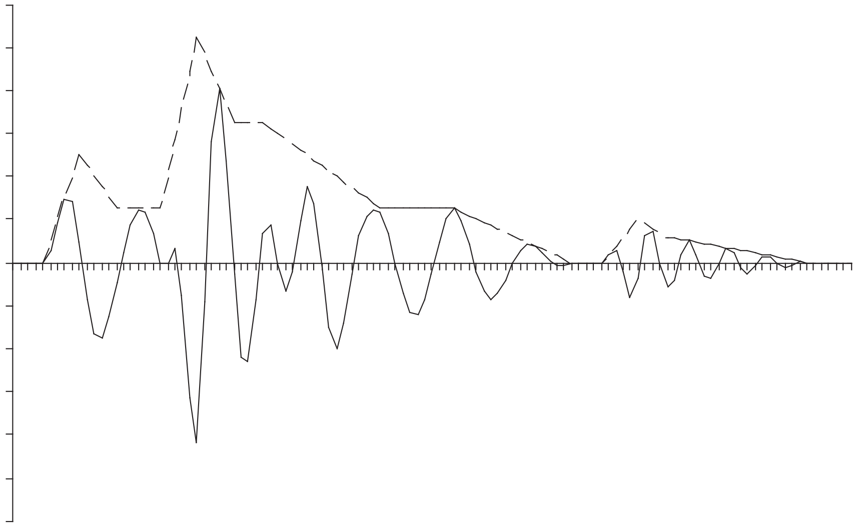


Figure 6.10. Sine-wave synthesiser output

this waveform is produced directly in the time domain. Typical instruments have three or four resonant peaks, the shape of which, and the time-variation in the shape of which, give the instrument its characteristic timbre. In this example, I will produce only a single, non-time-varying, formant.

Each formant-wave function (FWF) is the product of a sine wave at the resonant frequency, and an envelope, which determines the shape of the spectral peak [125]. The envelope is given by:

$$e(t) = \begin{cases} 0, & t \leq 0 \\ \frac{1}{2}(1 - cos(\beta t))e^{-\alpha t}, & 0 < t \leq \frac{\pi}{\beta} \\ e^{-\alpha t}, & t > \frac{\pi}{\beta} \end{cases} \tag{6.1}$$

The parameters α and β control the shape of the spectral peak. The exponential $e^{-\alpha t}$ never terminates, so, to minimise the computational burden, the envelope is changed to a linear decay when the rate of decay drops below some slope δ [120]. The modified envelope is then:

$$e(t) = \begin{cases} 0, & t \leq 0 \\ \frac{1}{2}(1 - cos(\beta t))e^{-\alpha t}, & 0 < t \leq \frac{\pi}{\beta} \\ e^{-\alpha t}, & \frac{\pi}{\beta} < t \leq \tau \\ e^{-\alpha \tau} - (t - \tau)\delta, & t > \tau \end{cases} \tag{6.2}$$

```
fwfEnvelope :: Float -> Float -> Float -> Stream Float
fwfEnvelope alpha beta delta = zipWithS (*) attack decay
    where
    attack = mapWhileS (<= pi)
                       (\p -> 0.5 - 0.5 * cos p)
                       0.0 (phaser 0.0 (beta/(2.0*pi)))
                       repeatS

    decay = takeWhileDeltaS
                (< (-delta/sampleRate))
                1.0 (1.0 :- mapS exp (ramp (-alpha) 0.0))
                (\x -> rampTo x (-delta) 0.0 doneS)

fwf :: [Float] -> Generator Float
fwf [alpha, beta, delta, fc] freq
    = zipWithS (*) (fwfEnvelope alpha beta delta) (siner fc)

periods :: Float -> Timed Float
periods freq = timed (cycle [Daton 0.0, Hiaton ((truncate period) - 1)])
               where
               period = sampleRate / freq

formanter :: [Float] -> Generator Float
formanter parms = combineT sumV . mapT (fwf parms) . periods
```

Figure 6.11. Formant-wave-function tone generation

where $\delta = \alpha e^{-\alpha t}$. Thus, $\tau = \frac{1}{\alpha}\ln(\frac{\delta}{\alpha})$ is the time at which the exponential decay changes to a linear decay.

Figure 6.11 lists the code for FWF tone generation. *fwfEnvelope* generates the envelope of a single formant-wave-function. The *attack* stream is the rising half-cosine wave until time π/β, followed by the last of those values repeated forever. The *decay* stream is the exponential decay until time τ, followed by a linear decay. The total envelope is the point-wise product of the two.

A single format-wave function is generated by *fwf*, which accepts a list of FWF parameters, and produces the point-wise product of the envelope and a sine wave at the resonant frequency, f_c.

An FWF waveform generator is a series of possibly-overlapping formant-wave-functions. A timed stream containing datons only at the times at which a new FWF is to start is

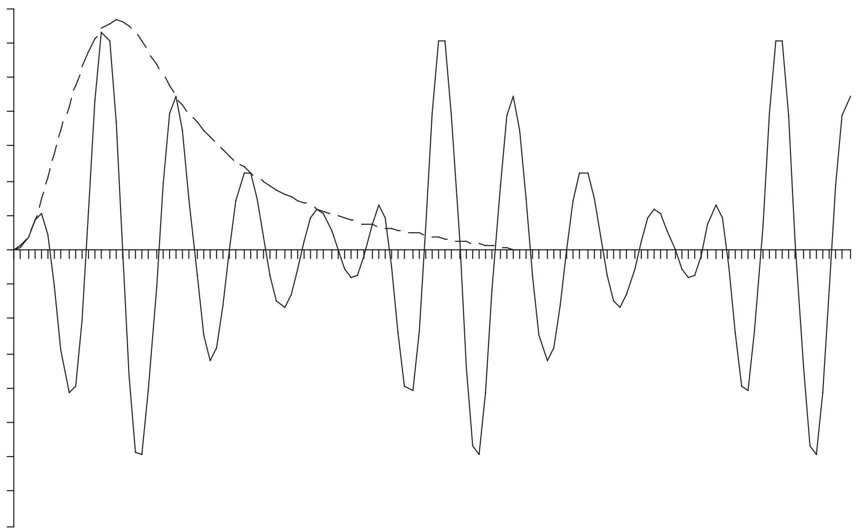


Figure 6.12. Formant-wave-function output

generated by the *periods* generator. (*periods* is quite coarse: it just truncates the period to the nearest integer number of ticks. This will result in incorrect tuning at higher frequencies, but it is adequate for this example.) The *formanter* function mimics the structure of the top level of the synthesiser: it produces a single FWF for each daton in *periods*, then sums them together into a single waveform.

Figure 6.12 shows a portion of the output of the FWF waveform generator. The dotted line is the envelope of a single FWF. Note that this waveform will be multiplied by the note's envelope. Thus, the FWF synthesiser exhibits asynchronicity both in the times at which notes are produced, and within each note, at the note's pitch period. To produce a single FWF, several processes are created; for each note, hundreds or thousands are created. This dynamism is inherent in the algorithm; it remains to be seen whether this degree of dynamism can be implemented in real time, or whether algorithms like this can only be implemented by removing some of this dynamism.

6.6 Summary

This chapter demonstrated the use of stream processing functions for writing dynamic process networks. The approach hinges on a particular choice of time representation suitable for real-time implementation: hiatons that represent an integral number of ticks of a base clock. Because of the complexity of this representation, I have been unable to produce a complete set of primitive functions (section 6.3), as I did for synchronous streams. Nonetheless, many useful examples were shown.

The functions that generate dynamic process networks (section 6.4) are even harder to limit to a few key functions. Those I defined and demonstrated in that section were suited to the kinds of network structure used in the example application (section 6.5), but other applications will likely require additional functions. It is not clear yet whether it will be possible to provide a sufficient set of primitive functions, or whether it will be necessary to allow programmers to define their own recursive functions. More work on other applications of dynamic process networks will be necessary to make further assessment of the validity of this approach.

Chapter 7

Summary

This thesis has covered a lot of ground, not all to the depth individual topics deserve. In this final chapter, I summarise the particular contributions made by the thesis, and indicate specific directions in which further work can proceed. Finally, I conclude with my thoughts on the value of the framework introduced in chapter 1 and expounded in remaining chapters.

7.1 Contributions

In chapter 1, I introduced a new framework for signal processing development, in which a high-level textual language, a visual language, and an efficient low-level execution model (dataflow process networks) combine to form an extremely powerful tool. Although there is no implementation of the complete framework, I believe I have laid much of the groundwork for further implementation. Apart from the working Haskell prototype code (appendix A and in the text), portions of the framework have been implemented: Dawson's prototype Visual Haskell editor [41]; the two implementations of SPOOK—Signal Processing Object-Oriented Kernel—described in [118] and [96] (see also page 9); and some portions of a compiler for modern digital signal processors [119].

Key to the framework is an efficient implementation model. The model, dataflow process networks, although widely used in signal processing development environments, lacks a formal description of its semantics. In chapter 3, I presented a formal syntax and semantics of dataflow actors and processes. As far as I know, this is the first attempt to present a formal semantics of dataflow actors and processes in this way. In the final portion of the chapter, I presented a new form of actor, phased-form actors. Although early in its development, this kind of actor offers new insights into the behaviour of dataflow actors, and shows promise as a path to improved semantic descriptions of dataflow networks.

Chapter 4 presented the Visual Haskell language. Visual Haskell is a substantial design for a visual language; it builds on the work in the functional programming community that has lead to the establishment of Haskell as the *de facto* functional programming language. It conforms to the most successful model of visual languages, pipeline dataflow. The formal syntax presented in chapter 4 covers a substantial portion of standard Haskell. I used Visual Haskell extensively in chapter 5 to illustrate functional programs; I believe these examples demonstrate that the language is usable and (within its stated limitations) complete. When used to draw functions on streams, it closely resembles block diagrams (see, for example, figure 5.11). Dawson's prototype implementation of the language [41] demonstrates the feasibility of constructing an editor for it.

Throughout the thesis, I have maintained that a functional programming language is an excellent vehicle for expressing dataflow and "block-diagram" style systems. Although this realisation is not new, I believe that, in chapters 5 and 6, I have explored this relation in the context of signal processing to a depth not considered previously. A key realisation during this work is the fact that any SDF network can be built from only delays and five actors (section 3.2.5). Because of this, just six functions are all that is required to represent any first-order SDF network in a functional programming language (section 5.3). This enables us to employ SDF scheduling techniques as an implementation technology for this restricted class of functional programs.

Having identified the utility of functional programming for dataflow, I explored the implications of two key facets of functional programming: higher-order functions, and program transformation (sections 5.4 and 5.5). Although well-developed in the functional programming community, these concepts were foreign to the signal processing community. The use of higher-order functions in signal processing has been implemented in the Ptolemy system [86], based on my work on this topic.

Finally, I have proposed and explored a novel representation of timed streams (chapter 6). A realistic and complete example illustrates how this representation can be used to express dynamically-changing process networks. Although this work is still evolving, it nonetheless illustrates a complex dynamic network.

7.2 Further work

There are several key directions in which further work can proceed.

The semantic description of dataflow actors in chapter 3 could benefit from a more concise notation. Additional sugaring on the syntax of standard-form actors would make it easier to write actors. For example, refutable state patterns would make it possible to write actors like *iota* as a standard-form actor—in other words, the state becomes part of the rule-selection process. Further work is needed on the relationship of phased-form actors and dynamic process networks. Because of the difficulty of finding a complete set of primitive functions over timed streams, the best approach would, I think, be to translate recursively defined functions on timed streams into phased-form actors. Further work is also need on phased-form actors—in particular, an algorithm that derives the phased form of a network of phased-form actors.

Although Visual Haskell (chapter 4) has proved to be an invaluable visualisation tool, and there has been a prototype implementation, there is still no complete implementation. It would be an interesting effort to construct a complete editor; an interesting approach may be build a complete Visual Haskell domain into the Ptolemy system for use in signal processing exploration and simulation, using say Tcl/Tk [103] as the graphical environment. Further work is also needed on iconic representations and type annotations, to find useful representations of the dynamic process networks of chapter 6.

Some of the features described in chapter 5 could be incrementally added to existing signal processing development systems: higher-order functions, and a simple catalogue of program transformations, for example. Still, the most effective means of implementing the dataflow-network-in-Haskell approach would be to add a new domain to the Ptolemy system. By doing so, there is no need to adapt functional I/O mechanisms (page 93) to real-time streams; instead, Haskell *expressions* represent dataflow networks, which are then embedded into an existing dataflow environment.

The work in chapter 6 is a first attempt at identifying the minimal set of functions needed to support timed dataflow. Further work is needed to make this set of functions complete, and to relate these functions to phased-form dataflow actors. Also, the relation of this work to synchronous languages [13] needs to be clarified. The exploration of dynamic process networks in sections 6.4 and 6.5 is intriguing, but further examples need to be developed. An important ingredient in further work along this line will be development of a formal model for dynamically-evolving dataflow networks.

7.3 Concluding Remarks

The complete framework presented in this thesis is somewhat idealistic. Given that users of such a framework are likely to be engineers, requiring that they learn a functional programming language is unlikely to be a welcome learning overhead. Lazy evaluation is probably out-of-place in a production signal processing environment, despite its utility in exploration and prototyping. And the complexities of modern DSP devices are such that generating assembler code efficient enough to meet performance expectations from high-level code (such as standard-form actors) requires analysis and optimisation technology that does not yet exist.

The real value of the framework lies perhaps in its use as an ideal model and as a framework within which specific areas of work can proceed (as I have done in the body of this thesis). I see two key directions in which practical development based on the framework can proceed: i) to use it to identify limitations in existing systems, and new features that can be added to them; and ii) to design a Haskell-like visual dataflow language that has an appropriate mix of strict and non-strict evaluation. I hope to be able to pursue both of these directions within the Ptolemy system, by extending it to support aspects of my ideal model, and by implementing the new dataflow language as a Ptolemy domain.

Bibliography

[1] William B. Ackerman. Data flow languages. *IEEE Computer*, pages 15–24, February 1982.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, August 1986.

[4] Alan L. Ambler, Margaret M. Burnett, and Betsy A. Zimmerman. Operational versus definitional: A perspective on programming paradigms. *IEEE Computer*, 25(9):28–43, September 1992.

[5] Analog Devices. *ADSP-21020 User's Manual*, 1991.

[6] Analog Devices. *ADSP-21060 SHARC Super Harvard Architecture Computer*, October 1993. Preliminary datasheet.

[7] Arvind and David E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[8] Arvind and Kattamuri Ekanadham. Future scientific programming on parallel machines. *Journal of Parallel and Distributed Computing*, 5:460–493, 1988.

[9] E. A. Ashcroft. Eazyflow architecture. SRI Technical Report CSL-147, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, April 1985.

[10] Tom Axford and Mike Joy. List processing primitives for parallel computation. *Computer Language*, 19(1):1–17, 1993.

[11] John Backus. Can programming be liberated from the Von Neumann style? *Communications of the ACM*, 21(8):613–641, 1978.

[12] Brian Barrera and Edward A. Lee. Multirate signal processing in Comdisco's SPW. In *ICASSP 91*, pages 1113–1116, May 1991.

[13] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In *A Decade of Concurrency: Reflections and Perspective*, pages 1–45. Springer-Verlag, June 1993.

[14] Albert Benveniste, Paul Le Guernic, and Christian Jacquenot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[15] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language. *Science of Computer Programming*, 19:87–152, 1992.

[16] Shuvra S. Bhattacharyya and Edward A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing*, 6, 1993.

[17] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, pages 28–45, October 1990.

[18] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[19] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[20] R.S. Bird. Lectures on constructive functional programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1988.

[21] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[22] Tore A. Bratvold. *Skeleton-Based Parallelisation of Functiona Programs*. PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, UK, November 1994.

[23] K. P. Brooks. Lilac: A two-view document editor. *IEEE Computer*, pages 7–19, June 1991.

[24] Manfred Broy. Applicative real-time programming. In R. E. A. Mason, editor, *Information Processing 83*, 1983.

[25] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *International Journal of Computer Simulation*, 1992. Special issue on "Simulation Software Development".

[26] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, Electrical Engineering and Computer Sciences, University of California Berkeley, 1993.

[27] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *28th Asilomar Conference on Circuits, Signals and Systems*, November 1994.

[28] W. H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25, January 1975.

[29] Margaret Burnett and Benjamin Summers. Some real-world uses of visual programming systems. Technical Report TR 94-60-7, Oregon State University, 1994.

[30] David Busvine. Implementing recursive functions as proceeor farms. *Parallel Computing*, 19:1141–1153, 1993.

[31] Luca Cardelli. Two-dimensional syntax for functional languages. In *Proc. Integrated Interactive Computing Systems*, pages 107–119, 1983.

[32] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[33] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declaration language for programming synchronous systems. In *14th ACM Symp. on Principles of Programming Languages, Munich, West Germany*, pages 178–188, January 1987.

[34] Rulph Chassaing. *Digital Signal Processing with C and the TMS320C30*. Topics in Digital Signal Processing. John Wiley and Sons, 1992.

[35] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman Publishing, 1989.

[36] Michael Colefax. A realtime polyphonic music synthesiser. Technical report, School of Electrical Engineering, University of Technology, Sydney, November 1993. Undergraduate thesis report.

[37] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea de Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3):56–66, March 1995.

[38] Stuart Cox, Shell-Ying Huang, Paul Kelly, Junxian Liu, and Frank Taylor. An implementation of static functional process networks. In *PARLE'92—Parallel Architectures and Languages Europe*, pages 497–512. Springer Verlag, 1992. LNCS 605.

[39] Alan D. Culloch. Porting the 3L Parallel C environment to the Texas Instruments TMS320C40. In A. Veronis and Y. Paker, editors, *Transputer Research and Applications 5*. IOS Press, 1992.

[40] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *PARLE'93—Parallel Architectures and Languages Europe*, pages 146–160. Springer Verlag, June 1993.

[41] Ken Dawson. Visual Haskell editor and parser. Technical report, School of Electrical Engineering, University of Technology, Sydney, November 1993. Undergraduate thesis report.

[42] Electronics Research Laboratory, University of California Berkeley. *The Almagest: Ptolemy User's Manual Version 0.5*, 1994.

[43] Marc Engels, Greet Bilson, Rudy Lauwereins, and Jean Peperstrate. Cyclo-static dataflow: Model and implementation. In *28th Asilomar Conference on Circuits, Signals and Systems*, November 1994.

[44] Paul Hudak et al. Report on the functional programming language Haskell, a non-strict purely-functional language, version 1.2. *SIGPLAN Notices*, May 1992.

[45] Antony A. Faustini and Edgar B. Lewis. Toward a real-time dataflow language. *IEEE Software*, pages 29–35, January 1986.

[46] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[47] Antony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[48] Markus Freericks and Alaois Knoll. Formally correct translation of DSP algorithms specified in an asynchronous applicative language. In *ICASSP 93, Minneapolis, USA*, pages I–417–I–420, April 1993.

[49] Daniel D. Gajski, David A. Padua, David J. Kuck, and Robert H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, pages 58–69, February 1982.

[50] H. Garsden and A. L. Wendelborn. Experiments with pipelining parallelism in SISAL. In *25th Intl. Hawaii Conf. on System Sciences*, January 1992.

[51] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Languages and Computer Architecture (FPCA) 93*, 1993.

[52] David M. Globirsch. An introduction to Haskell with applications to digital signal processing. Technical report, The MITRE Corporation, 7525 Colshire Drive, McLean, Virginia 22102-3481, September 1993.

[53] Eric J. Golin and Steven P. Reiss. The specification of visual language syntax. In *Proc. 1989 IEEE Workshop on Visual Languages*, pages 105–110, Rome, Italy, 1989.

[54] Kevin Hammond. Parallel functional programming: An introduction. In *Proc. 1st Intl. Symp. on Parallel Symbolic Computation (PASCO '94)*, pages 181–193, Hagenburg, Austria, 1994. World Scientific.

[55] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[56] P. Henderson. Purely functional operating systems. In *Functional Programming and its Applications.* Cambridge University Press, 1982.

[57] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the Silage language. In *ICASSP 90*, Alburqurque, New Mexico, April 1990.

[58] Daniel D. Hils. DataVis: A visual programming language for scientific visualisation. In *Proc. 1991 ACM Computer Science Conference*, pages 439–448, San Antonio, Texas, March 1991.

[59] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69–101, 1992.

[60] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[61] Paul Hudak. Para-functional programming. *IEEE Computer*, pages 60–70, August 1986.

[62] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[63] Paul Hudak and Mark Jones. Haskell vs Ada vs C++ vs Awk vs ...: An experiment in software prototyping productivity. Research report, Dept. of Computer Science, Yale University, July 1994.

[64] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[65] R. Jagannathan. Dataflow models. In E. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.

[66] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, Portland, Oregon, 1987. Springer-Verlag. LNCS 274.

[67] G. Jones. Deriving the fast Fourier algorithm by calculation. In K. Davis and J. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 80–102. Springer-Verlag, 1990.

[68] Geraint Jones and Mary Sheeran. Circuit design in ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.

[69] Mark Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 52–61, Copenhagen, Denmark, June 1993.

[70] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2):162–174, February 1989.

[71] Simon B. Jones. A range of operating systems written in a purely functional style. Technical Monograph PRG-42, Oxford University Computing Laboratory, September 1984.

[72] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[73] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Principles of Programming Languages 93*, pages 71–84, January 1993.

[74] John L. Kelly Jr., Carol Lochbaum, and V. A. Vyssotsky. A block diagram compiler. *The Bell System Technical Journal*, pages 669–678, May 1961.

[75] Gilles Kahn. The semantics of a simple language for parallel processing. In *Information Processing 74*, pages 471–475. North-Holland, 1974.

[76] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*. North-Holland, 1977.

[77] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations, determinacy, termination, and queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[78] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[79] Joel Kelso. A visual representation for functional programs. Technical Report CS-95/01, Murdoch University, Australia, December 1994.

[80] Takayuki Dan Kimura, Ajay Apte, Samudra Sengupta, and Julie W. Chan. Form/formula: A visual programming paradigm for user-definable user interfaces. *IEEE Computer*, 28(3):27–35, March 1995.

[81] Alois Knoll and Markus Freericks. An applicative real-time language for DSP programming supporting asynchronous data-flow concepts. *Microprocessing and Microprogramming*, 32:541–548, August 1991.

[82] Jeffrey Kodosky, Jack MacCrisken, and Gary Rymar. Visual programming using structured data flow. In *Proc. 1991 IEEE Workshop on Visual Languages*, pages 34–39, Kobe, Japan, October 1991.

[83] Konstantinos Konstantinides and John R. Rasure. The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3):243–252, May 1994.

[84] P.J. Landin. A correspondence between ALGOL60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8:89–101, 1965.

[85] Rudy Lauwereins, Piet Wauters, Merleen Ade, and J. A. Peperstraete. Geometric parallelism and cyclo-static data flow in GRAPE-II. In *5th Intl Workshop on Rapid System Prototyping*, Grenoble, France, June 1994.

[86] Edward A. Lee. private communication, 1993.

[87] Edward A. Lee. Dataflow process networks. Memorandum UCB/ERL M94/53, Electronics Reserach Laboratory, July 1994.

[88] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. Anonymous ftp from ptolemy.eecs.berkeley.edu, March 1994.

[89] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1):24–35, January 1987.

[90] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[91] Allen Leung and Prateek Mishra. Reasoning about simple and exhaustive demand in higher-order lazy languages. In *5th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 328–351, Cambridge, MA, August 1991.

[92] Andreas Maasen. Parallel programming with data structures and higher-order functions. *Science of Computer Programming*, 18:1–38, 1992.

[93] G. P. McKeown and A. P. Revitt. Specification and simulation of systolic systems functional programming. In *Proc. 6th Intl. Workshop on Implementation of Functional Languages*, University of East Anglia, Norwich, UK, September 1994.

[94] David G. Messerschmitt. A tool for structured functional simulation. *IEEE Trans. on Special Topics in Communications*, January 1984.

[95] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, October 1992.

[96] Matthias Meyer. A pilot implementation of the host-engine software architecture for parallel digital signal processing. Technical report, School of Electrical Engineering, University of Technology Sydney, and Technical University Hamburg-Harburg, November 1994. FTP from `ftp.ee.uts.edu.au` as `/pub/DSP/papers/spook.ps.gz`.

[97] G. J. Michaelson, N. R. Scaife, and A. M. Wallace. Prototyping parallel algorithms using Standard ML. In *Proc. British Machine Vision Conference*, 1995.

[98] Motorola Inc. *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, 1989.

[99] Marc A. Najork and Eric Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *Proc. 1990 IEEE Workshop on Visual Languages, Skokie, Illinois*, pages 215–220, October 1990.

[100] Marc A. Najork and Simon M. Kaplan. The CUBE language. In *Proc. 1991 IEEE Workshop on Visual Languages*, pages 218–224, Kobe, Japan, October 1991.

[101] Marc A. Najork and Simon M. Kaplan. Specifying visual languages with conditional set rewrite systems. In *Proc. 1993 IEEE Symposium on Visual Languages*, pages 12–18, August 1993.

[102] Jeffrey V. Nickerson. Visual programming: Limits of graphical representation. In *Proc. 1994 IEEE Symposium on Visual Languages*, pages 178–179, October 1994.

[103] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[104] Perihelion Software Ltd. *The Helios Parallel Operating System*. Prentice Hall, 1991.

[105] Simon L. Peyton-Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, February 1989.

[106] Simon L. Peyton-Jones and David Lester. A modular fully-lazy lambda-lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, May 1991.

[107] Keshav Pingali and Arvind. Efficient demand-driven evaluation, part 1. *ACM Trans. on Programming Languages and Systems*, 7(2):311–333, April 1985.

[108] Jose Luis Pino, Thomas M. Parks, and Edward A. Lee. Mapping multiple independent synchronous dataflow graphs onto heterogeneous multiprocessors. In *28th Asilomar Conference on Circuits, Signals and Systems*, November 1994.

[109] John A. Plaice. RLucid, a general real-time dataflow language. In *Proc. Formal Techniques in Real-time and Fault-tolerant Systems*, Nijmegan, the Netherlands, January 1992. Springer-Verlag. LNCS 571.

[110] Jorg Poswig, Guido Vrankar, and Claudio Moraga. VisaVis—a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, 5:83–111, 1994.

[111] Douglas B. Powell, Edward A. Lee, and William C. Newman. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. In *ICASSP 92*, pages V–553–V–556, 1992.

[112] John Rasure and Mark Young. Dataflow visual languages. *IEEE Potentials*, 11(2):30–33, April 1992.

[113] Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[114] H. John Reekie. Towards effective programming for parallel digital signal processing. Technical Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[115] H. John Reekie. Real-time DSP in C and assembler. FTP from `ftp.ee.uts.edu.au` as `/pub/prose/c30course.ps.gz`, 1993.

[116] H. John Reekie. Modelling asynchronous streams in Haskell. Technical Report 94.3, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, June 1994. FTP from `ftp.ee.uts.edu.au` as `/pub/prose/async-streams.ps.gz`.

[117] H. John Reekie. Visual Haskell: A first attempt. Technical Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, August 1994. FTP from `ftp.ee.uts.edu.au` as `/pub/prose/visual-haskell.ps.gz`.

[118] H. John Reekie and Matthias Meyer. The host-engine software architecture for parallel digital signal processing. In *Proc. PART'94, Workshop on Parallel and Real-time Systems, Melbourne, Australia*, July 1994. FTP from `ftp.ee.uts.edu.au` as `/pub/prose/host-engine.ps.gz`.

[119] H. John Reekie and John M. Potter. Generating efficient loop code for programmable dsps. In *ICASSP 94*, pages II–469–II–472. IEEE, 1994.

[120] Hideki John Reekie. A real-time performance-oriented music synthesiser. Technical report, School of Electrical Engineering, University of Technology, Sydney, November 1987. Undergraduate thesis report.

[121] John Reekie. Integrating block-diagram and textual programming for parallel DSP. In *Proc. 3rd Intl. Symp. on Signal Processing and its Applications (ISSPA 92)*, pages 622–625, August 1992.

[122] John Reekie and John Potter. Transforming process networks. In *Proc. MFPW'92, the Massey Functional Programming Workshop*, Palmerston North, New Zealand, August 1992. Massey University.

[123] John Reekie and John Potter. Process network transformation. In David Arnold, editor, *Parallel Computing and Transputers (PCAT-93)*, pages 376–383. IOS Press, November 1993.

[124] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Trans. Software Engineering*, 11(3):324–333, March 1985.

[125] Xavier Rodet. Time-domain formant-wave-function synthesis. *Computer Music Journal*, 8(3):9–14, 1984.

[126] Paul Roe. *Parallel Programming using Functional Languages.* PhD thesis, Dept. of Computing Science, University of Glasgow, 1991.

[127] Gary Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[128] Mary Sheeran. Designing regular array architectures using higher order functions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 220–237, Nancy, France, September 1985. Springer-Verlag. LNCS 201.

[129] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[130] David Skillicorn. Stream languages and dataflow. In J.-L Gaudiot and L. Bic, editors, *Advanced Topics in Dataflow Computing*, pages 439–454. Prentice-Hall, 1991.

[131] David Skillicorn and Janice Glasgow. Real-time specification using Lucid. *IEEE Trans. on Software Engineering*, 15(2):221–229, February 1989.

[132] D.B. Skillicorn. Parallelism and the Bird-Meertens formalism. FTP from qucis.queensu.ca in /pub/skill, April 1992.

[133] John A. Stankovic and Krithi Ramamritham. Introduction. In John A. Stankovic and Krithi Ramamritham, editors, *Hard Real-Time Systems*, chapter 1. IEEE Computer Society Press, 1988.

[134] W. D. Stanley, G. R. Dougherty, and R. Dougherty. *Digital Signal Processing.* Reston Publishing, 1984.

[135] William Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6(3):291–311, May 1986.

[136] V. S. Sunderan, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experience, and trends. *Parallel Computing*, 1994. To appear.

[137] Texas Instruments Inc. *TMS320C4x User's Guide*, 1991. Literature number SPRU063.

[138] Texas Instruments Inc. *TMS320C3x User's Guide*, 1992. Literature number SPRU031C.

[139] D.A. Turner. The semantic elegance of applicative languages. *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 85–92, 1981.

[140] David A. Turner. An approach to functional operating systems. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 199–218. Addison-Welsey, Reading, MA, 1990.

[141] Steven R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, C-23(12):1050–1071, December 1984.

[142] Ingrid M. Verbauwhede, Chris J. Scheers, and Jan M. Rabaey. Specification and support for multi-dimensional DSP in the Silage language. In *ICASSP 94*, pages II–473–II–476, Adelaide, Australia, April 1994.

[143] Eric Verhulst. Meeting the parallel DSP challenge with the real-time Virtuoso programming system. *DSP Applications*, pages 41–56, January 1994.

[144] W. W. Wadge and A. Ashcroft. *Lucid—the Dataflow Programming Language*. Academic Press, 1985.

[145] P. Wadler. Applicative style programming, program transformation and list operators. In *Functional Programming Languages and Computer Architecture*, pages 25–32. ACM, 1981.

[146] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *Proc. ACM Symp. on Lisp and Functional Programming*, 1984.

[147] Philip Wadler. How to replace failure by a list of successes—a method for exception-handling, backtracking, and pattern-matching in lazy functional languages. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer-Verlag, 1985. LNCS 201.

[148] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[149] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.

[150] Malcolm Wallace. *Functional Programming and Embedded Systems*. PhD thesis, Dept. Of Computer Science, University of York, UK, January 1995.

[151] Malcolm Wallace and Colin Runciman. Type-checked message-passing between functional processes. In *Proc. Glasgow Functional Programming Workshop*. Springer-Verlag, September 1994. Workshops in Computer Science Series.

[152] Kevin Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from functional prototypes—a case study. Technical Report 90/4, Heriot-Watt University, Edinburgh, UK, 1990.

[153] A. L. Wendelborn and H. Garsden. Exploring the stream data type in SISAL and other languages. In M. Cosnard, K. Ebcioglu, and J.-L. Gaudiot, editors, *Architectures for Fine and Medium Grain Parallelism*, pages 283–294. IFIP, Elsevier Science Publishers, 1993.

[154] Patrick Willekens, Dirk Devisch, Marc Van Canneyt, Paul Conflitti, and Dominique Genin. Algorithm specification in DSP station using Data Flow Language. *DSP Applications*, pages 8–16, January 1994.

[155] Carla S. Williams and John R. Rasure. A visual language for image processing. In *Proc. 1990 IEEE Workshop on Visual Languages*, pages 86–91, 1990.

Appendix A

Haskell Code

The *Vector* module

```
-- Types --

infixr  5 +++, :>

data Vector a      = NullV
                   | a :> (Vector a)

vector            :: [a] -> Vector a
vector []          = NullV
vector (x:xs)      = strict (:>) x (vector xs)      -- evaluate spine

nullV             :: Vector a -> Bool
nullV NullV        = True
nullV (_:>_)       = False

unitV  :: a -> Vector a
unitV x = x :> NullV

-- Iterators --

mapV              :: (a -> b) -> Vector a -> Vector b
mapV f NullV       = NullV
mapV f (x:>xs)     = f x :> mapV f xs

generateV         :: Int -> (a -> a) -> a -> Vector a
generateV 0 f a    = NullV
generateV n f a    = x :> generateV (n-1) f x where x = f a

iterateV          :: Int -> (a -> a) -> a -> Vector a
iterateV 0 f a     = NullV
iterateV n f a     = a :> iterateV (n-1) f (f a)

foldlV            :: (a -> b -> a) -> a -> Vector b -> a
foldlV f a NullV   = a
foldlV f a (x:>xs) = foldlV f (f a x) xs
```

```
scanlV           :: (a -> b -> a) -> a -> Vector b -> Vector a
scanlV f a NullV   = NullV
scanlV f a (x:>xs) = q :> scanlV f q xs where q = f a x

meshlV           :: (a -> b -> (a,c)) -> a -> Vector b -> (a, Vector c)
meshlV f a NullV   = (a, NullV)
meshlV f a (x:>xs) = (a'', y:>ys) where
                         (a', y)   = f a x
                         (a'', ys) = meshlV f a' xs

foldrV           :: (b -> a -> a) -> a -> Vector b -> a
foldrV f a NullV   = a
foldrV f a (x:>xs) = f x (foldrV f a xs)

scanrV              :: (b -> a -> a) -> a -> Vector b -> Vector a
scanrV f a NullV      = NullV
scanrV f a (x:>NullV) = f x a :> NullV
scanrV f a (x:>xs)    = f x y :> ys where ys@(y:>_) = scanrV f a xs

meshrV           :: (b -> a -> (c,a)) -> a -> Vector b -> (Vector c,a)
meshrV f a NullV   = (NullV, a)
meshrV f a (x:>xs) = (y:>ys, a'') where
                         (y, a'') = f x a'
                         (ys, a') = meshrV f a xs

-- Permutors --

(+++)            :: Vector a -> Vector a -> Vector a
NullV +++ ys       = ys
(x:>xs) +++ ys     = x :> (xs +++ ys)

lengthV          :: Vector a -> Int
lengthV NullV      = 0
lengthV (x:>xs)    = 1 + lengthV xs

atV               :: Vector a -> Int -> a
(x:>xs) `atV` 0     = x
(x:>xs) `atV` (n+1) = xs `atV` n

selectV          :: Int -> Int -> Int -> Vector a -> Vector a
selectV _ _ 0 xs   = NullV
selectV o s n xs   = xs `atV` o :> selectV (o+s) s (n-1) xs

groupV            :: Int -> Vector a -> Vector (Vector a)
groupV n v
    | lengthV v < n = NullV
    | otherwise     = selectV 0 1 n v :> groupV n (selectV n 1 (lengthV v - n) v)

concatV          :: Vector (Vector a) -> Vector a
concatV           = foldrV (+++) NullV

zipV                :: Vector a -> Vector b -> Vector (a,b)
zipV (x:>xs) (y:>ys)  = (x,y) :> zipV xs ys
zipV _ _              = NullV

unzipV              :: Vector (a,b) -> (Vector a, Vector b)
unzipV NullV          = (NullV, NullV)
```

```
unzipV ((x,y):>xys)   = (x:>xs, y:>ys) where (xs,ys) = unzipV xys

-- permutors (using additional functions) --

infixr  5 >>
infixl  5 <:, <<

(<:)                 :: Vector a -> a -> Vector a
xs <: x               = xs +++ unitV x

(<<)                 :: Vector a -> a -> Vector a
xs << x               = tailV xs +++ unitV x

(>>)                 :: a -> Vector a -> Vector a
x >> xs               = unitV x +++ initV xs
```

The *Stream* module

```
-- Types --

infixr  5 :-

data Stream a = NullS
              | a :- (Stream a)

stream       :: [a] -> Stream a
stream []     = NullS
stream (x:xs) = x :- stream xs

unitS            :: a -> Stream a
unitS x           = x :- NullS

nullS            :: Stream a -> Bool
nullS NullS       = True
nullS (_:-_)      = False

headS            :: Stream a -> a
headS (x:-_)      = x

tailS            :: Stream a -> Stream a
tailS (_:-xs)     = xs

-- Primitives --

mapS             :: (a -> b) -> Stream a -> Stream b
mapS f NullS      = NullS
mapS f (x:-xs)    = f x :- mapS f xs

groupS           :: Int -> Stream a -> Stream (Vector a)
groupS n NullS    = NullS
groupS 0 _        = NullS
groupS n xs
    | nullV v   = NullS
    | otherwise = v :- groupS n (dropS n xs)
      where v = takeSV n xs

concatS        :: Stream (Vector a) -> Stream a
```

```
concatS NullS   = NullS
concatS (v:-vs) = appendVS v (concatS vs)

zipS                 :: Stream a -> Stream b -> Stream (a,b)
zipS (x:-xs) (y:-ys)  = (x,y) :- zipS xs ys
zipS _ _              = NullS

unzipS               :: Stream (a,b) -> (Stream a, Stream b)
unzipS NullS          = (NullS, NullS)
unzipS ((x,y):-xys)   = (x:-xs, y:-ys) where (xs, ys) = unzipS xys

-- Useful functions --

appendVS :: Vector a -> Stream a -> Stream a
appendVS NullV s   = s
appendVS (x:>xs) s = x :- appendVS xs s

takeSV :: Int -> Stream a -> Vector a
takeSV k = tk k NullV
    where
    tk 0 v s       = v
    tk k v NullS   = NullV
    tk k v (x:-xs) = tk (k-1) (v <: x) xs

dropS          :: Int -> Stream a -> Stream a
dropS k NullS   = NullS
dropS 0 s       = s
dropS k (x:-xs) = dropS (k-1) xs
```

The *Timed* module

```
-- Types--

infixr  5 :-:

data Timed a = NullT
             | Token a :-: (Timed a)

data Token a = Hiaton Int
             | Daton  a
             | Datons (Vector (Token a))
             | Block  (Vector a)

-- Make timed stream from list --

timed       :: [Token a] -> Timed a
timed []     = NullT
timed (x:xs) = x :-: timed xs

timeT        :: Stream a -> Timed a
timeT NullS   = NullT
timeT (x:-xs) = Daton x :-: timeT xs

unitT  :: Token a -> Timed a
unitT t = t :-: NullT
```

```
-- Utilities need for "primitives" --

nullT       :: Timed a -> Bool
nullT NullT = True
nullT _     = False

-- Map on timed stream

tmapT :: (a -> Token b) -> (Int -> Token b) -> Timed a -> Timed b
tmapT f g (Daton x :-: xs)  = f x :-: tmapT f g xs
tmapT f g (Hiaton n :-: xs) = g n :-: tmapT f g xs
tmapT f g _                 = NullT

--  Zip two timed streams together --

zipT :: Timed a -> Timed b -> Timed (Token a, Token b)
zipT xs ys = zipT' 0 0 xs ys
    where
    zipT' 0 0 (Daton x :-: xs) (Daton y :-: ys)
                 = Daton (Daton x, Daton y) :-: zipT' 0 0     xs ys
    zipT' 0 0 (Daton x :-: xs) (Hiaton n :-: ys)
                 = Daton (Daton x, Hiaton 1) :-: zipT' 0 (n-1) xs ys
    zipT' 0 0 (Hiaton m :-: xs) (Daton y :-: ys)
                 = Daton (Hiaton 1, Daton y) :-: zipT' (m-1) 0 xs ys
    zipT' 0 0 (Hiaton m :-: xs) (Hiaton n :-: ys)
        | m == n = Hiaton m :-: zipT' 0 0     xs ys
        | m < n  = Hiaton m :-: zipT' 0 (n-m) xs ys
        | m > n  = Hiaton n :-: zipT' (m-n) 0 xs ys

    zipT' 0 n (Daton x  :-: xs) ys
                 = Daton (Daton x, Hiaton 1) :-: zipT' 0 (n-1) xs ys
    zipT' 0 n (Hiaton m :-: xs) ys
        | m == n = Hiaton m :-: zipT' 0 0     xs ys
        | m < n  = Hiaton m :-: zipT' 0 (n-m) xs ys
        | m > n  = Hiaton n :-: zipT' (m-n) 0 xs ys

    zipT' m 0 xs (Daton y  :-: ys)
                 = Daton (Hiaton 1, Daton y) :-: zipT' (m-1) 0 xs ys
    zipT' m 0 xs (Hiaton n :-: ys)
        | m == n = Hiaton m :-: zipT' 0 0     xs ys
        | m < n  = Hiaton m :-: zipT' 0 (n-m) xs ys
        | m > n  = Hiaton n :-: zipT' (m-n) 0 xs ys

    zipT' _ _ _ _ = NullT

-- Unzip a stream of pairs (not used!)

unzipT   :: Timed (Token a, Token b) -> (Timed a, Timed b)
unzipT ((Daton (Daton x, Daton y))  :-: xys)
         = (Daton x :-: xs, Daton y :-: ys)   where (xs,ys) = unzipT xys
unzipT ((Daton (Daton x, Hiaton 1)) :-: xys)
         = (Daton x :-: xs, Hiaton 1 :-: ys)  where (xs,ys) = unzipT xys
unzipT ((Daton (Hiaton 1, Daton y)) :-: xys)
         = (Hiaton 1 :-: xs, Daton y :-: ys)  where (xs,ys) = unzipT xys
unzipT ((Hiaton n) :-: xys)
         = (Hiaton n :-: xs, Hiaton n :-: ys)where (xs,ys) = unzipT xys
unzipT _  = (NullT,NullT)
```

```
-- State process on timed stream. Defined directly rather than --
-- in terms of tmapT, to get around ``break-up'' of hiatons.   --

tstateT :: (a -> b -> (a, Token c)) -> (a -> Int -> (a, Token c)) -> a -> Timed b -> Timed c
tstateT f g a (Daton x :-: xs)  = y :-: tstateT f g b xs where (b,y) = f a x
tstateT f g a (Hiaton n :-: xs) = y :-: tstateT f g b xs where (b,y) = g a n
tstateT _ _ _ _ = NullT

-- Concatenate a stream of tokens and token vectors --

concatT                   :: Timed a -> Timed a
concatT (Hiaton n :-: xs) = Hiaton n :-: concatT xs
concatT (Daton x  :-: xs) = Daton x  :-: concatT xs
concatT (Datons v :-: xs) = v `append` concatT xs
    where
    NullV   `append` ys = ys
    (x:>xs) `append` ys = x :-: (xs `append` ys)
concatT _ = NullT


-- Group a stream into vectors (k greater than 1) --

groupT k xs = groupT' 0 xs
    where
    groupT' 0 NullT = NullT
    groupT' i NullT
        | k == i = unitT (Hiaton k)
        | k < i  = Hiaton k :-: groupT' (i-k) NullT
        | k > i  = unitT (Hiaton i)

    groupT' 0 xs = v :-: groupT' j rest where (v,j,rest) = splitT k xs
    groupT' i xs
        | k == i = Hiaton k :-: groupT' 0 xs
        | k < i  = Hiaton k :-: groupT' (i-k) xs
        | k > i  = (Hiaton i `consT` v) :-: groupT' j rest where (v,j,rest) = splitT (k-i) xs

groupT _ _ = NullT

-- Take a vector off the front of a stream. Also returns the --
-- rest of the stream and an initial hiaton value for it.    --

splitT :: Int -> Timed a -> (Token a, Int, Timed a)
splitT 1 (Daton x :-: xs) = (Datons (unitV (Daton x)), 0, xs)
splitT k (Daton x :-: xs) = (Daton x `consT` v, j, rest) where (v,j,rest) = splitT (k-1) xs
splitT m (Hiaton n :-: xs)
    | m == n   = (Hiaton n, 0, xs)
    | m < n    = (Hiaton m, n-m, xs)
    | nullT xs = (Hiaton n, 0, NullT)
    | m > n    = (Hiaton n `consT` v, j, rest) where (v,j,rest) = splitT (m-n) xs
splitT _ _     = (Hiaton 0, 0, NullT)

-- (Join a token onto a token vector) --

consT :: Token a -> Token a -> Token a
consT x (Hiaton n) = Datons (x :> unitV (Hiaton n))
consT x (Datons v) = Datons (x :> v)
```

```
-- Join a vector of blocks back into sync stream. Hiatons not allowed. --

concatvT :: Timed a -> Stream a
concatvT (Daton x :-: xs) = x :- concatvT xs
concatvT (Block v :-: xs) = v `appendVS` concatvT xs
concatvT _              = NullS

-- ``Spawn'' new streams from a timed stream --

spawnT :: (a -> Bool) -> Timed a -> Timed (Timed a)
spawnT p NullT              = NullT
spawnT p s@(Daton x :-: xs)
    | p x                   = Daton s  :-: spawnT p xs
    | otherwise             = Hiaton 1 :-: spawnT p xs
spawnT p (Hiaton n :-: xs)  = Hiaton n :-: spawnT p xs

-- Combine timed sync streams into single top-level sync stream --

combineT :: (Vector a -> b) -> Timed (Stream a) -> Stream b
combineT c = concatvT . tstateT f g []
    where
    f ss s = (mdrop 1 (s:ss), Daton  (c (vector (map headS (s:ss)))))
    g ss k = (mdrop k ss,     Block (takeSV k (mmap c ss)))

mdrop :: Int -> [Stream a] -> [Stream a]
mdrop n = filter (not . nullS) . map (dropS n)

mmap :: (Vector a -> b) -> [Stream a] -> Stream b
mmap f ss = f (vector (map headS ss)) :- mmap f (mdrop 1 ss)

{-
    New functions for truncating streams...
-}
type Generator a = a -> Stream a

-- doneS:-: terminate a stream --

doneS :: Generator a
doneS = const NullS

-- truncate stream according to predicate and state.

takeFiniteS :: (s -> a -> s) -> (s -> a -> Bool)
                  -> s -> a -> Stream a -> Generator a -> Stream a
takeFiniteS f p s a t@~(x:-xs) c
    | nullS t        = c a
    | p s x         = x :- takeFiniteS' (f s x) xs x
    | otherwise     = c a
    where
    takeFiniteS' s NullS z = c z
    takeFiniteS' s (x:-xs) z
        | p s x         = x :- takeFiniteS' (f s x) xs x
        | otherwise     = c z

-- takeAsLongS :-: take items until predicate on control --
-- stream satisfied. First daton must satisfy.         --
```

```
takeAsLongS :: (a -> Bool) -> Timed a -> Stream b -> Generator b -> Stream b
takeAsLongS p (Daton y :-: ys) (x:-xs) c
    | p y       = x :- takeAsLongS' p ys xs c x
    | otherwise = error "takeAsLongS"
takeAsLongS p (Hiaton n :-: ys) (x:-xs) c
                = x :- takeAsLongS' p (Hiaton (n-1) :-: ys) xs c x

takeAsLongS' p (Daton y :-: ys) (x:-xs) c a
    | p y        = x :- takeAsLongS' p ys xs c x
    | otherwise  = c a
takeAsLongS' p (Hiaton 1 :-: ys) (x:-xs) c a
                 = x :- takeAsLongS' p ys xs c x
takeAsLongS' p (Hiaton n :-: ys) (x:-xs) c a
                 = x :- takeAsLongS' p (Hiaton (n-1) :-: ys) xs c x
takeAsLongS' p _ _ c a = c a


-- Truncate a stream to the same length as another

truncateS :: Stream a -> Stream b -> Generator b -> Stream b
truncateS (x:-NullS) (y:-ys) c    = y :- c y
truncateS (x:-xs)    (y:-NullS) c = y :- c y
truncateS (x:-xs)    (y:-ys) c    = y :- truncateS xs ys c
```