University of Exeter
Department of Computer Science

Developments in Dataflow Programming

Daniel Julius Maxwell

April 2018

Supervised by Dr Antony Galton & Prof. Jonathan Fieldsend

Submitted by Daniel Julius Maxwell to the University of Exeter as a thesis for the degree of Doctor of Philosophy in Computer Science, April 2018.

I certify that all material in this thesis which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other University.

(signature) .....

Abstract

Dataflow has historically been motivated either by parallelism or programmability or some combination of the two. This work, rather than being directed primarily at parallelism or programmability, is instead aimed at maximising the overall utility to the programmer of the system at large. This means that it aims to result in a system in which it is easy to create well-constructed, flexible programs that comply with the principles of software engineering and architecture, but also that the proposed system should be capable at performing practical real-life tasks and should be as widely applicable as can be achieved.

With those aims in mind, this project has four goals:

- to argue for a unified global dataflow coordination system, extensible to be able to accommodate components of any form that may exist now or in the future;
- to establish a link between the design of such a system and the principles of software engineering and architecture;
- to design a dataflow coordination system based on those principles, aiming where possible to embed them in the design so that they become easy or unthinking for programmers to apply; and
- to implement and test components of the proposed system, using it to build a set of three sample algorithms.

Taking the best ideas that have been proposed in dataflow programming in the past — those that most effectively embed the principles of software engineering — and extending them with new proposals where necessary, a collection of interactions and functionalities is proposed, including a novel way of using partial evaluation of functions and data dimensionality to represent iteration in an acyclic graph.

The proposed design was implemented as far as necessary to construct three test algorithms: calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort. The implementation was successful in representing iteration in acyclic dataflow, and the test algorithms generated correct results, limited only by the numerical representation capabilities of the underlying language. Testing and working with the implemented system revealed the importance to usability of the system being visual, interactive and, in a distributed environment, always-available.

Proposed further work falls into three categories: writing a full specification (in particular, defining the interfaces by which components will interact); developing new features to extend the functionality; and further developing the test implementation.

The conclusion summarises the vision of a unified global dataflow coordination system and makes an appeal for cooperation on its development as an open, non-profit dataflow system run for the good of its community, rather than allowing a proliferation of competing systems run for commercial gain.

To Avocado, for whom I hope computers will be easier to use.

Acknowledgements

I would like to thank my supervisors, Dr Antony Galton for his great attention to detail, being lightning-fast to understand and tease apart my logic, and insistence on clear explanations; and Professor Jonathan Fieldsend for solid competence, deep knowledge of programming languages, high standards and some timely advice on productivity.

Huge thanks to Clare for listening, sharing my excitement about the subject, being brave enough to read sections of my thesis and for her valiant efforts to explain the topic to our friends.

Thank you to my funders, EPSRC (grant number EP/K503046/1), and to the Department of Computer Science for awarding me the funding.

And thank you to my examiners, Dr Yulei Wu and Professor Ian Watson, for taking on that mammoth task.

Contents

List of Figures

Chapter 1

Introduction

1.1. Purpose

Dataflow captures much of the essence of how programming should be done. Its cause-and-effect nature reflects the way we think about the world, with nodes representing events or objects and their behaviours, incoming connections representing causes, outgoing connections and node actions representing results or outcomes, and chains of connected nodes representing processes or sets of instructions.

Dataflow has been founded on understandability from the start: its origin in the concept of a flow chart was itself motivated by wanting to communicate complex ideas between expert and non-expert users (Gilbreth and Gilbreth [1921]). It has been shown to be a productive paradigm for programming — even with a basic and incomplete interface, researchers have demonstrated productivity gains for both expert and non-expert users, that increase with the complexity of the tasks involved (Baroth and Hartsough [1995], Whitley [1997], Morrison [2010]).

Carriero and Gelernter [1989] made a case for regarding the coordination and computational components of a program as separate and orthogonal (see also Gelernter and Carriero [1992]). As a coordination system, dataflow has huge advantages. With nodes regarded as black boxes, the system is by its nature indifferent to the inner workings of nodes, allowing for, as Carriero and Gelernter argued, an entirely heterogeneous ensemble of components containing a diversity of systems, platforms and languages. A dataflow coordination system could be powerfully exploited to coordinate the full global ensemble of computing components that currently exists and much more besides.

Dataflow has historically been motivated either by parallelism or programmability or some combination of the two. This work, rather than being directed primarily at parallelism or programmability, is instead aimed at maximising the overall utility to the programmer of the system at large. This means that it aims to result in a system in which it is easy to create well-constructed, flexible programs that comply with the principles of software engineering and architecture (described in Chapter 3), but also that the system should be

capable at performing practical real-life tasks and should be as widely applicable as can be achieved.

The principles of software engineering and architecture are usually seen as instructions for programmers on how they should run projects or construct their programs. In this document, instead, the whole system is designed around them, with the intention of conceiving a system in which programmers are freed up to focus on their core logic rather than the architecture of their software.

As well as the architecture of the programs created, the architecture of the system itself plays a part. Its parallelisability is valuable for utilisation of processors, but also makes programs distributable over multiple machines, locations and users. The modularity of the system means that components within the same program can be divided between different providers, making them more easily interchangeable and leaving users free to choose the providers of components that best suit them. Such interchangeable components include user interfaces, storage components, processing resources and, in general, cloud service provision.

Much of the work in this document is aimed at improving the functionality of the system — thereby increasing its applicability, so that the widest possible range of tasks can be achieved using it. The use of partial evaluation to achieve iteration, for example, is partly presentational, allowing iteration to fit more easily within the the dataflow paradigm, but also affects the architecture of the system; resulting in functions being executed at different times and on different machines than would otherwise have been the case.

With those general aims in mind, the goals of this work are:

- to argue for a unified global dataflow coordination system, consisting of a protocol for communication between nodes and providing interfaces for interchangeable resources such as node hardware (storage and computation), user interfaces and a directory of available nodes, a system that would be extensible to be able to work with any form of computational components that may exist now or in the future;

- to establish a link between the design of such a system and the principles of software engineering and architecture, with those principles ideally embedded into the design of the system so that they become easy or unthinking for programmers to apply, even without necessarily knowing or understanding the principles themselves. While the list of principles and the manner in which they are embedded may be open to question, the importance of embedding them should be clear;

- to design a dataflow system based on such principles, utilising those ideas from the previous literature that most closely embed them, and adding new features and methods where needed (most notably using partial evaluation and data dimensionality to represent iteration in an acyclic graph); and

- to implement and test components of the proposed system, using it to build a set of three sample iterative algorithms (calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort).

Among the ideas previously proposed in dataflow that are utilised in this document are acyclicity (which leads to simplicity), node heterogeneity and separation between the programming language and the system of coordination between components. Some of the most important principles of software engineering are flexibility, modularity and platform independence. Dataflow is a good fit for such principles because it is inherently modular and therefore flexible.

The design of a unified global dataflow coordination system requires two classes of features and components: features that are intrinsic to the system, and components that are necessary but interchangeable. The intrinsic features have characteristics determined by the system design. Examples are the addressing of nodes, the protocol for communication between nodes, and the API (Application Programming Interface) through which nodes communicate with other components of the system and the outside world. A node must be able to communicate with the user interface via the API, nodes must be able to communicate with each other using the same protocol, they must understand the user interactions, and be able to receive inputs and deliver outputs in the required data formats.

However, the intention in the design is to retain the maximum flexibility by making most components of the system interchangeable. The system at large can be indifferent to the internal node implementation: whereas *some* internal implementation is needed, and one is built for the purpose of testing, it is interchangeable, and could equally well be exchanged for any other, provided it exhibits the required behaviour. Likewise, the user interface, the programming languages in which node functions are written, and the hardware used for data storage and function execution are all interchangeable. The system is indifferent to the implementation of all such components.

Whereas an implementation has been provided here for demonstration purposes, there is no suggestion that this implementation is in any way optimal or ideal; it is simply one of many that could have been constructed. Discussion of any idealised or optimised implementation of the system is largely omitted from this work. The application of the existing field of Functional Programming, and in particular Graph Reduction, would mostly fit within the realm of implementation and optimisation. The technique of Partial Evaluation, for example, discussed in Section 4.8, would most probably benefit from existing approaches to Graph Reduction used in Functional Programming.

This document attempts to make use of the solutions previously suggested that fit the goals most closely. As described in Section 2.3 (Side-Effects), although many in the past have adopted purely functional approaches there are many others (for example, Rumbaugh [1977]; Davis and Keller [1982]; Hudak [1989]; Johnston et al. [2004]), who have recognised the need for non-determinism in practical systems. Where approaches in this document deviate from the principles of purely functional languages, or from a notion of 'pure' dataflow, this has been done for the sake of achieving the project goal of maximising the utility of the system at large to its users. An example of this would be where nodes are allowed to maintain state, fetch external data and to have side-effects external to the system. By allowing them these abilities we sacrifice functional purity, but also gain useful capabilities for real-life applications.

In advocating a unified global dataflow coordination system, the conclusion of this thesis makes an appeal for cooperation: for those working on dataflow to collaborate to bring such a system into being, working together on the same open system rather than allowing a proliferation of competing or proprietary systems. The important principles of node heterogeneity and interchangeability of components and resources allow such a system to embrace a multitude of possible implementations of node internals and user interactions, all living within the same system of node coordination and communication. The conclusion argues, finally, that such a system can only work, or will work best, if it is open, available to all, and managed by a non-profit body in the interests of its users, on something like the model of the World Wide Web Consortium[1], rather than by private companies that would be subject to narrower goals.

1.2. What Is Dataflow?

The concept of dataflow is simple enough: a program is represented as a directed graph in which units of data or functionality, represented by nodes, are depicted as being connected by arcs, representing data dependencies between them.

It reflects the way we think about the world: the cause-and-effect nature of a node is similar to the way we think about objects and events, and the connections give us chains of actions, similar to the way we think about processes or sets of instructions.

Dataflow lends itself to visual representation, not least because visual representation gives us access to two dimensions. Whereas the one dimension available to us in text requires flow control commands to represent forks and convergence by moving the position of the control point, two dimensions represent these more easily, in a way that can be perceived and rapidly understood by the human eye, whilst also providing the data dependency chain necessary to be able to parallelise the operations involved.

Dataflow as a programming paradigm started being proposed and discussed by a number of people independently in the 1960s and late 1950s, although all were rooted in the concept of a flow-chart, which was seeded much earlier, as a business tool, in 1921.

Work on dataflow has usually been motivated either by parallelisability or programmability. Whereas dataflow does produce parallelisable code, this has not always translated into efficiency. Dataflow necessarily avoids any underlying state, which means data must be copied in order for nodes to perform their operations. Various proposals have been made to mitigate this problem, the simplest of which is simply to avoid fine-grained node operations, in which the copying overhead is higher and tend instead towards coarse-grained operations where the copying overhead is lower. Another proposal often cited is to improve efficiency by compiling sections of a dataflow graph into sequential (control-flow) code that can be executed more efficiently. How efficiency concerns are best addressed depends on the nature of the programs in question, the balance of priorities between the various aspects of overhead cost (communication, memory and computation), and the relative costs of having large numbers of small processors compared with small numbers of

[1] `https://www.w3.org/`

fast ones. Efficiency issues are not addressed in this document.

Greater success has been reported in achieving programmability with dataflow, with measurably better programming productivity at all levels of programming experience and complexity, but with the benefits increasing with the complexity of the tasks at hand (Baroth and Hartsough [1995], Whitley [1997], Morrison [2010]).

It is the desire to exploit and enhance these benefits in programmability and programming productivity that motivates the desire for a unified global dataflow coordination system, coordinating the global ensemble of computational components that already exists, and extensible to those not yet created or conceived of (including such biological computational machines as ourselves).

1.3. What Is A Coordination System

A Coordination System, in this context, is a system for coordinating computing components. Gelernter and Carriero [1992], in designing their language Linda (which they referred to as a *Coordination Language*), treated the coordination model as orthogonal to the computation model, such that the two could be chosen separately. The coordination language was described by them as a way of controlling asynchronous ensembles of activities — which could be programs, processes, threads or "*any agent[s] capable in principle of simulating a Turing Machine*". They saw ensembles as heterogeneous, incorporating "*different machines, or different computing models*", and their vision was grand: an activity could also, they said, be a person or another whole ensemble. As they put it, ensembles are "*fundamental and ubiquitous*" in computing, and their conception of a coordination system attempted to address how these heterogeneous components could be made to work together.

Linda involved creating what they called *tuples*: items of data that were that were 'cast adrift' into a global space, from which other processes would fetch them. There could also be what they called *active tuples*, meaning that an execution would take place before it produced a result which became a standard tuple.

Several different approaches to coordination have since been proposed and implemented. The most obvious succession to Linda was Laura (Tolksdorf [1998]). Laura focussed primarily on the implications of a system being not just distributed but also open, with no central control over the components that join or leave. Rather than having explicit connections between nodes, they used what they called a *service offer form* to define the name of a service together with the format of its expected inputs and outputs, which would be shared globally. These were then retrieved by submitting a *service request form*, for which a matching service offer would be found. This was aimed at achieving a loose coupling between providers and consumers of services, enabling any service to be substituted by another with the same name, and thereby accommodating dynamic joining and leaving of components in a system that exerted no overall control over such availability.

Others, over the years, have focussed on a variety of applications, environments and imple-

mentation issues. Opus (Chapman et al. [1997]) was an approach centred around entities referred to as 'Shared Abstractions'. Analogous to Objects in Object-Oriented Programming, these contained both properties and functions that could be called by other such entities. S-Net (Penczek et al. [2010]) was directed primarily at Signal Processing and the problems that prevail in that domain. PaCE (Caporuscio et al. [2012]) was a system composed of independent and autonomous nodes (which they called 'Actors'), distributed over a network, and focussed significantly on dealing with asynchronicity in such a system. Swift (Wozniak et al. [2013]) addressed implementation issues, aimed primarily at scalability. Not all approaches enabled iteration. PaCE, for example, did; S-Net did not. Most approaches accepted at least a limited form of non-determinism, although PaCE specifically excluded it. The most natural ancestors of the ideas discussed in this document are those put forward in the development of Linda and Laura (Gelernter and Carriero [1992]; Carriero and Gelernter [1989]; Tolksdorf [1998]) —- at least in some of their aims and ambitions, if not in their specific approaches.

The proposals in this document avoid being too specific about implementation or the nature of the communication protocol — believing instead that the need for global acceptance dictates that such details should be agreed collaboratively through wide discussion, in order to gain agreement and adoption. The aim is to conceive a generalised computing tool that can accommodate as many as possible of the optimisations that may emerge in future. Carriero and Gelernter commented that for Linda they would use or "*at any rate ... gratefully appropriate*" alternative approaches (Carriero and Gelernter [1989]), and a similar attitude to future development is advocated here.

The system described here deviates from other approaches in the use of partial evaluation to avoid explicit loops in the graph and in its conception of the system as forming a single, interactive, always-available constantly running program, responding in real-time to changes in both data and the program itself as they occur. The interactive nature of it has implications for non-determinism and referential transparency. In this system, the arrival of new input data signifies the correct moment to run, meaning that the result computed at that time is correct as corresponding to the inputs and outside environment that prevail at that moment.

Whereas previous efforts have been driven by a range of motivations — for example, parallelism, performance, expressiveness and programmability — this system is designed to be as general as possible, and must therefore be able to accommodate widest range of users' priorities. These potentially varying priorities are accommodated by giving nodes (or their owners) the flexibility to vary their behaviour accordingly.

Overall, the focus here is on the need for a single generalised distributed computing system. This document attempts to take a principled approach to decision-making, through the principles of software engineering and architecture (discussed in Chapter 3). However, anyone disagreeing with decisions presented here is urged to engage in discussion about how to broaden the applicability of such a system, wherever possible, in order to keep it as a unified generalised system rather than allowing a fragmentation of alternative approaches.

1.4. Document Structure

Chapter 2 (The History of Dataflow) describes the history of dataflow and the developments that have marked its progress. It highlights the use of partial evaluation, the question of allowing side-effects, the representation of iteration and the proposed use of dataflow as a coordination system.

Chapter 3 (Software Engineering) summarises the history of software engineering and the principles of software architecture. The goal is to establish a link between the principles and the design of a programming paradigm, ideally embedding the principles in the design and making it easy, unthinking or automatic for programmers to apply them.

Chapter 4 (Definition) attempts to use the best features of dataflow systems that have been proposed previously to design a skeletal system, based as far as possible on the engineering principles, and extending them with new features, in particular using partial evaluation and data dimensionality to represent iteration in acyclic dataflow.

Chapter 5 (Implementation) describes how the ideas in Chapter 4 have been put into practice. In most cases, a subset has been implemented, sufficient to prove that algorithms can be run but without including all features that would be necessary for a production or fully functional system run on untrusted code, and without any form of optimisation.

Chapter 6 (Evaluation and Results) gives an assessment of the definition and implementation for its programmability and ability to run a series of test algorithms successfully.

Chapter 7 (Further Work) describes three areas of further work: designing a full specification of the system; investigating additional features to enhance or extend the functionality; and further developing and testing the implementation.

Chapter 8 (Conclusion) concludes with a summary of the achievements and areas left open for further work, and finally an appeal for cooperation in developing a single system managed by a non-profit body in the interests of its users, rather than a multitude of competing private systems run for commercial gain.

The terms and abbreviations used in this document can be found in Appendix A, and a summary of the visual notation used to depict graphs can be found in Appendix B.

Chapter 2

The History of Dataflow

This chapter summarises the history of dataflow programming. It describes its origin and some of the most important developments and ideas that have been proposed. Ideas of particular importance to this project include the use of partial evaluation, the question of whether side-effects should be allowed, the representation of iteration and the proposed separation between the coordination and computation components of a programming environment.

2.1. Origin and Motivation

Dataflow was first conceived as a programming paradigm during the 1960s and late 1950s. Whiting and Pascoe [1994], in their history of dataflow languages, attributed its earliest invention to Karp and Miller [1966], and the coining of the term (independently) to Adams [1969], with Rodriquez [1969] (also independently) following. All three were preceded in publication date by Sutherland [1966]. Whiting and Pascoe also referred to work as early as 1958 (Young Jr and Kent [1958]) as having "*overtones of the data-flow model*".

There is an earlier precedent, in that all were rooted in the concept of a flowchart, attributed to Gilbreth and Gilbreth [1921], whose paper pre-dated the idea of dataflow in computation by decades.

In their paper, the Gilbreths described their notion of a flowchart (termed by them a 'process chart') as a clear way to view and understand the details of a business process, to help engineers and executives to improve processes and seek efficiencies. They envisaged process charts containing a multitude of symbols that could be adapted according to the process at hand, with any symbol made into its own more detailed sub-chart if required.

Their process charts were arranged as directed graphs, with flow from top to bottom. Initial nodes represented physical inputs or instructions, with subsequent nodes representing processes resulting, finally, in outputs — a physical item or an event. The examples they provided were acyclic: they did not explicitly preclude cyclicity, but because their process charts were aimed at producing an output, any loop would have to have an end point if the process chart's output were to be delivered.

They justified their creation in terms of its clarity and understandability. As they put it, their process charts enabled a process to be "*visualised all at once*", presenting information "*in such simple form that such information can become available to and usable by the greatest possible number of people in an organisation*". In particular, it satisfied, they claimed, the twin goals of being understandable by non-experts and useful for experts, saying it "*presents both simple and complicated problems easily and successfully*" and "*the process chart has met the tests of a satisfactory teaching device from the psychological standpoint, as well as of a satisfactory working device from an engineering standpoint*".

Of particular resonance in computing was the claim that a process chart "*makes possible the more efficient utilisation of similarities in different kinds of work*", which has echoes in the need for writing reusable code. As the Gilbreths noted in the final line of their report: "*Process charts pay*".

Sutherland [1966], in his PhD thesis, described dataflow as a graphical interactive system to be used for "*describing procedures in a two-dimensional programming language*", with procedures to be applied to "*data obtained from any source*". Although the programs Sutherland used to illustrate the concept used low level operations (mostly arithmetic), his vision was more far-sighted — Sutherland described what he called a 'macro' functionality, in which a new symbol could be defined to represent a combination of operators, allowing complex programs to retain their simplicity of representation.

Sutherland described dataflow as being beneficial in providing a "*natural way of expressing parallel operations*" pointing out that, whereas written languages are linear (one dimensional), a graphical language is two dimensional, and that we (humans), via our sense of vision, have "*considerable parallel input capability*". Karp and Miller [1966], who published their work later the same year, described a directed graph as a "*natural way to depict the sequencing of parallel computation*".

Sutherland mentioned in passing a key difference between dataflow and sequential programming: that in dataflow the system could "*[repeatedly scan] the total procedure to determine which parts are ready to be activated next*" (Sutherland [1966]). This difference was expressed in a later survey paper (Yazdanpanah et al. [2014]) by saying "*dataflow architectures use the availability of data to fetch instructions rather than the availability of instructions to fetch data*". Agerwala and Arvind [1982] expressed the difference differently, by contrasting the absence in dataflow of the shared global memory or single program counter that exist in a von Neumann architecture.

Much of the early work was aimed particularly at parallelism, and often using fine-grained dataflow. Dennis and Misunas [1975] built on the model of Karp and Miller [1966] by proposing a processor architecture aimed at improving the efficiency of concurrent operations, attempting to overcome some of the problems encountered when adapting von Neumann machines for parallel computation. Ackerman and Dennis [1979] built a language, named VAL, with similar aims, commenting that most previous languages "*reflect the storage structure of the von Neumann concept*". Arvind and Gostelow [1982], likewise, devised an approach aimed at overcoming previous bottlenecks to parallelism, in their

case through an interpreter for functional languages. Dennis [1980] made a general case for dataflow multiprocessors as a solution to those processor bottlenecks. However, the suspicion was starting to be raised that other advantages, such as modularity, extensibility and programmability ('easier program verification') could eventually come to eclipse the advantages of parallelisability (Agerwala and Arvind [1982]).

In later work, some effort was dedicated to development of textual dataflow languages. Davis and Keller [1982] compared the two approaches, noting that textual dataflow has often been motivated by concurrent execution, whereas in visual dataflow 'human engineering' became the primary motivating factor, reiterating points made by Agerwala and Arvind [1982], and by Sutherland [1966] and others before them.

Davis and Keller also decoupled the basic concept of dataflow (actions sequenced by the "*data availability firing rule*") from the visual representation of this behaviour as a directed graph. They specified an arc as a 'conceptual medium', differentiating their treatment of it from that of others, including Karp and Miller [1966], who viewed arcs more as physical storage queues that might have limited capacity. However, they did still treat arcs very much as pipes, introducing conditional constructs ('selectors' and 'distributors') that would, respectively, choose which of two incoming arcs to 'absorb' a token from, or choosing which of two outgoing arcs to 'emit' a token to.

Cox et al. [1989] later published a critique of textual programming languages in general, making an argument for greater use of graphics in programming and introducing their own creation, Prograph. They reiterated another point made previously by Sutherland, that text is inherently one-dimensional whereas visualisations are two-dimensional. Text-based programming forces the programmer, they argued, "*to consider how to linearly organise every program, whether or not the algorithm requires it*". They went further, by pointing out that textual programming has inherited a complex syntax from its natural language counterparts, but with the detrimental flaw of being inflexible and unforgiving, "*forcing the programmer to deal with small syntactic details, rather than the important concepts of algorithms*". They cited Aczél and Daróczy [1975] in pointing out the low information density of textual languages.

Cox et al. also had concerns about the use of variables, on the basis that they embed the additional concept of scope and, they argued, introduce unnecessary confusion and potential for error; problems which are exacerbated when used to express object orientation. Pictorial representations, they argued, can overcome these problems. Without going into as much detail, Hudak [1989], in his survey of functional programming languages, listed a range of motivations including both programmability and parallelisability.

The theme of usability, particularly by non-expert programmers, comes up repeatedly. Suárez [2013], in an undergraduate thesis, complained of a mismatch between the programming skill needed to program high performance computers and the programming skill that could reasonably be expected of 'domain experts' — people who have devoted their time to learning their fields rather than the technical skill of representing their knowledge in code.

McPhillips et al. [2009] conveyed a similar belief in the title of their paper, "*Scientific workflow design for mere mortals*". Their paper described a category of software the authors called 'scientific workflow systems', typically based on dataflow. They used an open source system called Kepler[1] for their examples. They identified a list of what they saw as desirable characteristics, and defined a framework for design of programs, following the principles described by Morrison [2010] in the earlier (1994) version of his book. The desirable characteristics were: clarity, well-formedness, predictability, recordability, reportability, reusability, scientific data modelling, and automatic optimisation. Similar principles appear amongst the established principles of software engineering good practice, described in Section 3.4.

2.2. Partial Evaluation

Sutherland [1966] pointed out one of the capabilities that gives dataflow its most important tool. In discussing what might happen if an input is not provided, he mooted the idea of dummy inputs, pointing out that any operator could then be used as a flow output, so that the operator could be initiated later, once its inputs became available.

Rumbaugh [1977] attempted to develop the idea into a more formal framework. He defined the system as being composed of purely functional (side-effect free) nodes, and introduced switches and joins, as flow control structures. This involved modelling the connections between nodes as pipes, along which 'tokens' (items of data) would flow. His definition reiterated Sutherland's 'macros' in different words, by stating that a dataflow graph could be "*an acyclic graph of dataflow programs*" (in other words, that a node could itself be a dataflow program).

He also reiterated the principle that a portion of a dataflow graph could be conveyed as an item of data. As Rumbaugh put it, "*any data flow program (without tokens) can be considered to be a procedure, which itself can be held as a token value*". As he saw it, a portion of a dataflow graph with missing tokens could be conveyed through the graph and later invoked by using an 'Apply' operator, together with values for the missing tokens. In Functional Programming, a similar idea is conveyed by the term 'higher order'. Darlington and Reeve [1981] defined higher order functional languages as those with "*the ability to pass functions as arguments and return them as values*". Such a scheme must necessarily be paired with a mechanism by which to evaluate those functions with a set of arguments at a later time (usually referred to as 'applying' them). In that paper, the authors claimed their own graph reduction scheme, named ALICE, as one that could support higher order languages.

Whereas Sutherland, and later Rumbaugh, only mentioned in passing this idea of conveying functions together with associated data through the graph, Keller [1980] went into detail on the subject. He called it a 'serial composition', and wrapped it inside an 'envelope', together with the tokens that had already been provided. He related this to the concept of closures, in which a function can be packaged with a set of variables, and 'Cur-

[1] `https://kepler-project.org/`

rying', which describes the representation of a function of many arguments as a chain of functions of one argument each[2].

Keller also, in his paper, designated what he saw as three fundamental data types: atoms (units of information), tuples (lists of atoms) and graphs (sections of a dataflow graph, which he also referred to as functions), along with a set of functions for operating on these data types. The notion of functions as values was reiterated later by Davis and Keller [1982].

In Functional Programming, graph reductions and their sub-type Beta-reductions (the substitution of variables in a Lambda expression with their values[3]) provide a form of partial evaluation. Generally, graph reduction (for example, of the kind described at length by Peyton Jones [1987]) provides a method of evaluating functional statements by evaluating the end branches of its function tree and replacing the functions with values, reducing the size of the tree with each step until the function is evaluated in full. After each stage of this process, the result is a reduced version of the original tree. Each step could reasonably be described as a partial evaluation of the tree.

Graph reduction techniques have, over a long period of time, suffered from accusations of inefficiency (for example, by Darlington and Reeve [1981], who attributed this partly to their being computed on hardware designed for a von Neumann architecture). Darlington and Reeve, in describing their scheme for performing graph reduction (named 'ALICE'), claimed inspiration from preceding dataflow projects in their acknowledgements. Comparing it with dataflow, they suggested that their scheme allowed them to "*support a wider range of applicative [functional] languages*". They also argued that it provided easier support for data structures (although the method they suggested of handling data structures could be applied equally to dataflow).

Hughes [1982] addressed the efficiency of graph reduction through their concept of 'super-combinators', a way of implementing graph reduction for which the authors claimed greater efficiency than pre-existing methods. They devised a method of achieving what they called 'fully-lazy evaluation', contrasting it with simple 'lazy evaluation' by saying that whereas fully-lazy evaluation ensured that every expression would be evaluated at most once, simple lazy evaluation would ensure only that "*every expression passed as a parameter to a λ-expression is evaluated at most once*". Their technique, they claimed, overcame a collection of disadvantages of previous methods, including avoiding the need to make multiple passes over expressions containing nested Lambda expressions and reducing the overhead involved in linking steps in the execution.

2.3. Side-Effects

Hudak [1989] described the existence of an underlying state as a key difference between

[2]The term 'Currying' is a reference to Haskell Curry, who developed the idea in his book, *Combinatorial Logic* (Curry et al. [1972]). Others (eg. Reynolds [1972]), have suggested that Moses Schönfinkel would be more rightly credited with the idea.

[3]Beta-reductions have continued to be widely deployed in Lambda Calculus. Lambda Calculus is otherwise not covered in this document.

imperative and declarative languages. Whereas imperative programs "*are characterised as having an implicit state that is modified (i.e. side effected) by constructs*", declarative languages, he suggested, "*are characterised as having no implicit state, and thus the emphasis is placed entirely on programming with expressions*". With reference to functional programming in particular, he went on to suggest that "*what is important is the functional programming style, in which [features such as higher-order functions, lazy evaluation, pattern matching and various kinds of data abstraction] are manifest and in which side effects are strongly discouraged but not necessarily eliminated*". He acknowledged, however, a "*very large contingency of purists ... who believe that purely functional languages are not only sufficient for general computing needs but also better because of their 'purity'*". Indeed, in early papers on dataflow it was sometimes taken as axiomatic that functional languages, including dataflow languages, must be side-effect free (Vegdahl [1984]; Agerwala and Arvind [1982]; Whiting and Pascoe [1994]). Hudak's characterisation of imperative and declarative languages, however, while restricting a declarative language by definition from using state as a means of computation, did not necessarily preclude components of declarative programs (particularly dataflow programs) from causing side-effects external to the program, or from maintaining their own internal state, provided those side-effects and state were not used as means of conveying data between components.

Part of the problem with a restriction on side-effects is its inhibiting of required forms of non-determinism from the system. Many of those who have worked on dataflow languages have recognised the need for some form of non-determinism (for example, Rumbaugh [1977]; Davis and Keller [1982]; Johnston et al. [2004]), despite in many cases developing systems that precluded it. Dennis [1974] advocated the inclusion of non-determinism in dataflow through an explicit non-deterministic construct, such that its absence could guarantee determinism where needed. Building partly on that work, Arvind et al. [1977] argued that explicit distinction between determinate and non-determinate parts of a programme would in fact give dataflow a key advantage over conventional languages, preventing non-determinacy from being able to "*sneak in through the back door*". They argued, in fact, that the explicit scheduling used in dataflow allows for better handling of non-determinism in general.

The choice of whether to allow side-effects divides languages that are purely functional (which do not allow side-effects) from those that take a more pragmatic approach, recognising the occasional need for side-effects and allowing them, but to some extent sacrificing their functional purity and referential transparency in the process. Hudak [1989] gave ML and Scheme as two examples of such languages.

Barth et al. [1991] divided the functional language approaches (that had been devised up to that point) into two categories: those they described as "*purely functional (strict or non-strict) with implicit parallelism or annotations for parallelism*"; and those they described as "*strict functional + sequential evaluation + imperative extensions + threads and semaphores for parallelism*". ML and Scheme would, they suggested, be in the second category. To those existing categories they added their own approach, called 'M-Structures', and put it in a third category, described as "*non-strict functional + implicitly parallel evaluation + M-structures + occasional sequentialisation*".

In Imperative Programming, work was directed towards solving the problems of using shared memory in a parallel or distributed environment. In 1993, Herlihy and Moss [1993] published their concept of *Transactional Memory* as a solution to the problems caused by locking techniques in shared data structures. The concept behind Transactional Memory was for certain blocks of code to be considered atomic; such that the entire block and its resulting changes to memory could be committed to memory as a unit if 'valid' (that is, if no other changes had taken place during execution), or aborted and the execution repeated if not. Although this solution mitigated some of the problems of locks, such as scalability and the possibility of deadlock, Harris et al. [2005] identified the problem that its abstractions "*cannot be composed together to form larger abstractions*". They proposed a solution, referred to as 'Composable Memory Transactions', which involved explicitly distinguishing memory operations (which can be safely re-executed if needed), from input/output operations (which cannot). They supplemented this with a programming construct (named 'retry') for triggering re-execution of a block of code only when the variables it depends upon have changed, and another (named 'orElse') for choosing one of two abstractions to run. Together these enabled abstractions to be safely composed. Harris et al. implemented their idea in Concurrent Haskell, a language which they described as a "*pure, lazy, functional language*", but with a mechanism to allow side-effects explicitly where needed.

Søndergaard and Sestoft [1990] drew a detailed distinction between four terms which are often associated, but which they argued need to be separated. The terms were: determinacy, definiteness, unfoldability and referential transparency. One such property should not necessarily, they argued, imply anything about the others. They cover "*different aspects of languages, and they all provide useful dimensions along which to characterise programming languages*". They proposed, for example, that a language could be non-determinate, definite, and referentially transparent.

In dataflow, connections (represented visually as arcs connecting two nodes) are used to express explicitly where and how pairs of nodes are permitted to communicate. For nodes to interact, directly or indirectly, in any other way, would undermine that principle. While this precludes the use of shared state, as with functional languages, functional purity can potentially be sacrificed to allow state or side-effects explicitly where needed, provided that principle of explicit communication between nodes is not broken. Johnston et al. [2004] suggested that most dataflow languages had indeed been determinate up to that point, naming only Valid, Cajole and DL1 (a set of examples that Whiting and Pascoe [1994] concurred with) as examples of those with, as they put it, "*very limited non-determinate features*". Nevertheless, they acknowledged the need for non-determinacy ("*if dataflow is to become an acceptable basis for general-usage programming languages, non-determinacy is essential.*"). Accordingly, they named it as an open issue in their survey.

2.4. Data-Driven vs. Demand-Driven Execution

Davis and Keller [1982] drew a distinction between two models of execution: data-driven and demand-driven. In data-driven execution the graph is executed when new data be-

comes available at the inputs; in demand-driven execution it is triggered by requests for data at the outputs. They thought of it in terms of a 'demand token' that flows against the direction of the arcs.

Treleaven et al. [1982] claimed that, at that time, the search was on for "*the next generation of computer to replace von Neumann organisation*". They considered the two variations of dataflow systems — data-driven and demand-driven — as potential candidates, and listed examples of each (13 data-driven and 8 demand-driven) that had been implemented or were under way at that time. They identified advantages and disadvantages of each, but noted that for a machine to achieve general-purpose goals, it might need to utilise the advantages of both, together with those of the more traditional control-flow mechanism in a synthesis of the three. As they put it, the three organisations, "*instead of being competitive, seem in fact to be complementary organisations*". In their minds, there would need to be a centralised organisation that defined multiple self-contained computing elements that could be plugged together to increase concurrency.

Darlington and Reeve [1981] discussed the issue using the terms *eager evaluation* (data-driven) and *lazy evaluation* (demand-driven). Acknowledging the advantages and disadvantages of each (for example, the potential of eager evaluation to waste resources but also to utilise under-utilised resources to pre-evaluate expressions), they chose a hybrid approach for their 'ALICE' project, which "*allows the user to determine which of the two modes is appropriate to each expression evaluated*". Peyton Jones [1987] pointed out that other languages, such as ML and Hope, also took a hybrid approach by using eager evaluation by default but lazy evaluation when requested by the programmer.

2.5. Iteration

Mosconi and Porta [2000] distinguished between two forms of iteration, which they called 'horizontally parallel', and 'temporally dependent' or 'sequential'. Horizontally parallel iteration (referred to in this document as 'independent' iteration) can be dealt with using what they called a 'for-all' construct — a command which executes a set of instructions once for every value in an array. For 'temporally dependent' iteration, which they also called 'sequential' (referred to in this document as 'dependent' iteration), they described two approaches that have been used — one using recursion and the other using switches and cycles.

The Vipers system (Ghittori et al. [1998]) was an example of a system that implemented both forms of iteration, using switch nodes for dependent iteration and a 'for-all' structure (referred to by them as 'for-each') for independent iteration. Dennis and Misunas [1975], similarly, achieved iteration using 'actors' to perform various forms of flow-control to direct flow back into the body of the iteration or out of it when finished. Mosconi and Porta noted the conflict that occurred when iteration was achieved using cycles, violating the principle that dataflow should be acyclic. There is a tendency, they pointed out, "*to consider cycles with visual program graphs as something to be avoided at all costs*". They went on to classify dataflow paradigms as 'pure' or not.

Keller [1980] discussed iteration in terms of loops in dataflow. He showed that any loop can be replaced with an appropriate recursion and that recursion can in turn be eliminated using an 'Apply' operator as described by Rumbaugh [1977] to create loops over an individual node, making the graph at large acyclic. He described as an example the application of this principle to calculating a factorial. Departing from Keller's previous work (Keller [1980]), Davis and Keller [1982] returned to the idea of cyclic data flow graphs, although for the sake of visual simplicity they then attempted to eliminate cycles from the visual display by naming the cyclical sections of the graph as subroutines. Auguston and Delgado [1997] proposed a solution involving a line signifying a loop over an array, with text describing the operation to be performed on its members, but using switches to achieve dependent iteration.

Despite the approaches above, along with many other similar solutions, iteration has continued to be cited as an open problem. The issue is that it has generally been achieved using special purpose nodes (eg. Rasure and Williams [1991]), which are limited in functionality, loops over the graph with switch nodes (eg. Ghittori et al. [1998]; Mosconi and Porta [2000]; Arvind and Gostelow [1982]), which lose visual simplicity, or boxes that enclose a section of the graph (eg. McLain and Kimura [1986]; National Instruments Corporation [1998]), which find a compromise but suffer both drawbacks to a lesser degree. Johnston et al. [2004] listed many of the approaches that had been taken up until that time, but suggested that they do not fit neatly into the dataflow paradigm, and furthermore that the efficient execution of such loops continues to cause a problem. It is a related but separate issue to decide how such structures should be expressed visually, and they listed examples too, of approaches that had been taken to this problem.

2.6. Pessimism ... and the Recovery

Vegdahl [1984] noted some of the inefficiencies in functional programming; in particular, the overhead involved in passing parameters and the lack of destructive updating ("*to return a modified version of a structure, it is necessary (logically) to return a new copy of the structure with the modification*"). In mitigation, he pointed out that these may reflect the implementations rather than being inherent to the execution model, and highlighted a number of techniques that had been proposed by others for improving efficiency. Amongst these was the idea of transforming a program to an alternative form at compilation time, combining the programmability of dataflow with the speed of sequential (imperative) programming.

Nikhil et al. [1989] proposed a partial solution to this problem using a concept they called 'I-Structures'. I-Structures were arrays of arrays in which each new write created a new array position, meaning that new data was stored only when it changed, rather than being copied routinely during transitions between nodes or at the start of every iteration in a loop. This reduced the overhead, but only partially and at the cost of increased memory and a loss of programmability.

Peyton Jones [1987] made a case for lazy evaluation as a way to avoid unnecessary computation, but also pointed out the down-side that it comes at the cost of execution speed.

The choice depends on the balance of priorities between execution speed and resource cost. As a result, some languages, he pointed out, use eager evaluation by default but support lazy evaluation when explicitly requested by the programmer.

By the early 1990s, there was a growing feeling that dataflow appeared not to be fulfilling its potential. Lee and Hurson [1993] published a paper exploring why this might have been. In their abstract, they suggested that "*a direct implementation of computers based on the dataflow model has been found to be a monumental challenge*". The problems they discussed revolved predominantly around the resource overhead, and in particular the communication overhead in parallel computations and the copying and resource allocation needed when dealing with data structures, questioning whether dataflow may have been *too* successful at finding concurrency. One school of thought at the time was that the solution lay in compiling technology rather than hardware support, making dataflow the programming solution to overlay other underlying technology. However, they believed that the search for solutions posed an "*arduous challenge*" and required "*extensive study*".

In the conclusion to their survey, Whiting and Pascoe [1994] quoted one of the early proponents of dataflow, Alan Davis of the University of Utah, as reflecting the gloomy assessment of the time by saying that dataflow had been "*mostly ... a failure*". However, they struck a more optimistic note in pointing out a number of its successes and its influence on developments in other fields.

Only a year after their previous pessimistic paper, Lee and Hurson published an article suggesting there was renewed interested in dataflow, despite its well-documented shortcomings (Lee and Hurson [1994]). A key factor in this resurgence, they suggested, was a convergence between dataflow and control-flow models — the use of dataflow for coordination between control-flow nodes. Their paper described a categorisation of dataflow as centralised or distributed. It referred to machine architecture rather than a wider system architecture, but pointed towards the more dramatic idea of distributing a dataflow system over more than just components of a single machine.

Hils [1992], in his survey of dataflow programming, expressed optimism for dataflow, but also listed a number of weaknesses, including that dataflow languages tend to take up a lot of screen-space relative to the amount of information conveyed, requiring procedural abstraction to hide unimportant details. He also expressed his view that any language must have a large library of predefined functions in order to be powerful. Bernini and Mosconi [1994] agreed, arguing it would be crucial for widespread adoption.

The onset of optimism seemed to have gained ground again a little later, when Baroth and Hartsough [1995] published their report on using visual programming in the real world. They touted "*productivity improvements of four to ten times compared to conventional text-based programming*", attributing the gains to improved communication between customer, developer and computer. Their tests used two products: LabView — one of the few dataflow products of the time that is still commercially available today; and HP's Vee. They described their customers as non-programmers, but who were able to understand the visual dataflow diagrams well enough to assist the developers in programming together,

and described new recruits being able to learn the paradigm quickly, to the point of being able to deliver working code to their customers. As well as supporting communication, they found that dataflow brought benefits in ease, reduced number of mistakes, and flexibility, although they expressed some doubt about long term scalability (having not yet experimented with larger or longer-lived applications). Morrison [2010], whose first edition was published at a similar time, in 1994, also justified work on his concept of 'Flow-Based Programming' in terms of programming productivity, and it was one of the motivations behind the development of DFScala nearly two decades later (Goodman et al. [2013]).

Whitley [1997] attempted to sum up the empirical evidence on the benefits of dataflow. He complained about the prevalence of unsupported speculation regarding ease of use. While the invention of flowcharts was motivated by improving understandability, its benefits had not been widely tested with code. Whitley cited evidence in research results that (with some caveats) found "*a significant advantage for all dependent variables and for all levels of algorithm complexity*", with benefits that "*increased as the algorithms became more complex*", by a ratio of up to 2.5 in the more complex cases and also with fewer errors. Looking for possible disadvantages, he cited findings that the effect depended on the task, and that if the visual interface were poorly designed then the beneficial effects would not appear, but found that, at worst, studies were suggesting a lack of productivity gain rather than any productivity cost. Providing support to the view of Baroth and Hartsough [1995], he concluded that dataflow has benefits in communicating with non-programmers, while also emphasising the productivity gains for expert programmers, even for simple tasks but with increasing benefits with increased complexity. He made a plea for greater use of the scientific method in designing interfaces, although acknowledged that this is not always required for success, citing spreadsheets as an example of huge success without empirical support.

Seeking to combine the efficient memory use (particularly with respect to data structures) of the von-Neumann model, with the parallelism and power-efficiency of dataflow for scheduling between components, Yazdanpanah et al. [2014] argued in favour of using a hybrid of the two architectures. The authors expressed a preference for ultimately combining or unifying the two. In making their case, they created a taxonomy of scheduling models, based on how the two approaches were combined. A reproduction of their taxonomy is shown in Figure 2.1.

Furthering this goal, Gajinov et al. [2014] published a benchmarking suite (named 'DaSH') for hybrid models, applying a collection of sample problems referred to as the Berkeley Dwarfs (devised originally by Asanovic et al. [2009]). They found shortcomings and expressed dissatisfaction with the API support of the three models tested but, like many before them, expressed hope for the future of dataflow.

Kyriacou et al. [2006] provided one example of Yazdanpanah et al.'s '*dataflow / control flow*' model with an approach they termed 'Data-Driven Multithreading' (DDM). They pointed out that the sequencing overhead was relatively lower when using coarse-grained nodes (as they put it — "the effect of the data-driven sequencing overhead is amortised"), albeit at the cost of some parallelism. In tests they found that efficiency improved as gran-

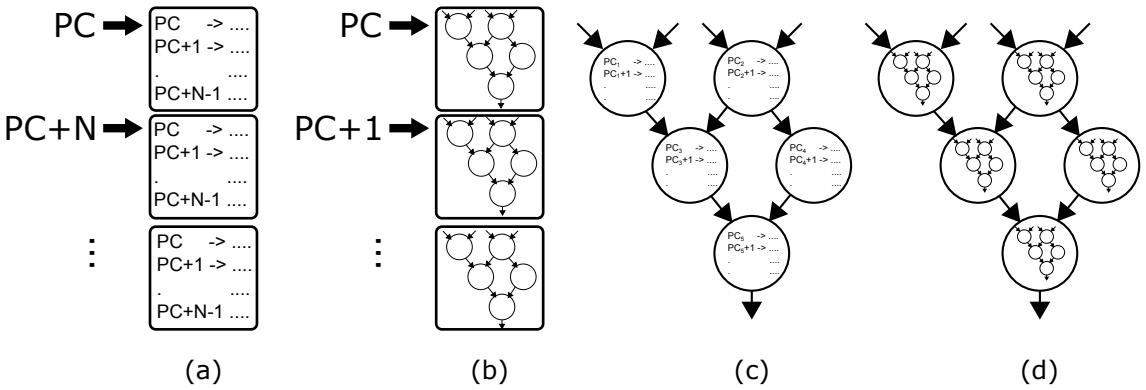

Figure 2.1. Reproduction of Yazdanpanah et al. [2014] Fig 2 — "*Inter- and intrablock scheduling of organizations of hybrid dataflow/ von-Neumann architectures. (a) Enhanced control flow, (b) control flow/ dataflow, (c) dataflow/control flow, and (d) enhanced dataflow. Blocks are squares and big circles.*"

ularity increased, until the point where the computational operations took long enough to allow scheduling operations to complete. While their exact results and peak granularities are specific to the architectures and algorithms used, the broader principle that granularity is beneficial up to a point, is more widely applicable.

The Codelet Program Execution Model developed by Zuckerman et al. [2011] also used Yazdanpanah et al.'s '*dataflow / control flow*' model. As they put it, a Codelet was "*a collection of machine instructions that can be scheduled 'atomically' as a unit of computation*". They were to be, they suggested, more coarse than traditional dataflow but more fine-grained than a traditional thread. Codelets were also grouped together into containers, referred to as '*Threaded Procedures*', in the process combining two of Yazdanpanah et al's four models — the '*dataflow / control flow*' model with the '*enhanced dataflow*' model.

The Teraflux project (Solinas et al. [2013]; Giorgi et al. [2014]), incorporating the coarse-grained DFScala language (Goodman et al. [2013]), made simultaneous use of many of the earlier innovations, including both DDM and the Codelet Program Execution Model. This encompassing approach resulted in Teraflux delivering both good performance and good scaling (within the ranges tested) as the number of cores was increased.

2.7. Coordination Languages

Carriero and Gelernter [1989] discussed the concept of a coordination language, aiming for generality by separating the coordination model from the computation model. They described the coordination component as "*orthogonal to the base language in which it's embedded*", and implemented a combined computation and coordination language in a system they called Linda.

The coordination aspect of it was similar to one proposed slightly later (independently) by Rasure and Williams [1991] in their visual language *Cantata*. At a similar time, Schwanke et al. [1989] also discussed the idea of a coordination system as a way of maintaining large software systems, visualised as a directed acyclic graph. In their vision, the graph would

be created automatically from the program and the program could in turn be recreated from the graph, making it possible to use the graph to edit the underlying program.

Carriero and Gelernter made the suggestion that both the computation and the coordination components are necessary for any language to be considered complete. In their model, the coordination language controls an ensemble of asynchronous activities, each of which could be a "*program, process, thread or any agent capable in principle of simulating a Turing Machine*", suggesting that this could be a person or another whole ensemble. They pointed out that ensembles are already ubiquitous, existing in some form whenever a human and computer communicate, or when computers communicate with each other. Their most contentious claims, they suggested, were that the coordination and computation components were orthogonal, and that it would be possible and desirable to create a coordination system that is general, being able to coordinate other ensembles of any kind.

They attributed Kahn and Miller [1988], approvingly, as having made the case for 'conceptual economy' as a way to retain flexibility. In their words: "*simple, economical languages tend to be supple and powerful, complex ones tend to be rigidly inflexible — a stubborn fact that emerges screaming from programming language history, only to be repeatedly ignored*". This was part of the reason for wanting generality in their coordination system, to avoid the apparent "*intellectually crippling*" consensus that "*every twist and turn in the hardware development path ... calls for a new language or programming model*".

They argued later (Gelernter and Carriero [1992]) that a coordination language would need to be able to coordinate an "*interconnected computer jungle*" of components, featuring "*diversity with respect to language, hardware platform, physical location, even basic computing model*". While others had argued for coordination and computation to be integrated and provided together within the same language, they made a case for the opposite, separating the coordination language to accommodate the diversity of components in an asynchronous ensemble of computers.

Johnston et al. [2004] pointed out in their survey paper that, by 2004, the notion of dataflow as a coordination system had gained increasing traction, citing Vipers (Bernini and Mosconi [1994]) as one such system. They also made it one of their five conclusions that, with the advent of heterogeneous distributed systems, dataflow as a coordination system was an important area of research.

2.8. Dataflow Classification

Whiting and Pascoe [1994] provided a useful classification of dataflow languages along seven axes according to their features. The axes were:

- The dataflow execution model

  The authors distinguished between a static execution model, a dynamic execution model, and a hybrid of the two.

  A static model is one in which a node is not allowed to compute its next value until

its previous value has been consumed by its destination nodes. This was described as making formal proofs easier, and was therefore more attractive to some researchers. A dynamic system is one that allows a node to execute as soon as its input data is available, but because multiple outputs can be sent along the same connection, it requires what the authors called 'colour' — a tag uniquely identifying each token.

- Representation of iteration

  Languages seemed to have chosen tail-recursion, special purpose iteration nodes, or switch nodes. They also mentioned what they called 'data-independent' iteration, in which every iteration is independent of every other iteration, and which can be performed using a 'for-all' loop.

- Recursion

  Most implementations, they suggested, had supported recursion, although some, such as SISAL[4], did not.

- Data structures

  The value-passing model can appear "*unpleasingly inefficient*", the authors suggested. The approaches to this problem included the 'I-Structures' method used by the Id language, as described above.

- Non-determinacy

  Determinacy is the property of a computation that, for a certain set of inputs, it will always proceed through the same set of steps, producing the same internal states as it does so and therefore producing the same output. An alternative (weaker) definition is the property that a computation will always produce the same output for a given set of inputs, regardless of the underlying steps. Johnston et al. [2004] used booking systems and database access systems as examples of applications in which non-determinacy would be needed, in both cases because the outcome depends on some internal or external state that is independent of the inputs.

- I/O support

  This had not been well addressed in dataflow, with the authors suggesting this may have been due to its mundanity. Their description suggested there may also have been reticence about introducing non-determinacy as a result.

- 'Lazy', 'Lenient' and 'Eager' evaluation

  While dataflow is fundamentally based on eager evaluation, this has the drawback of leading to infinite recursion. Some languages, they suggested, allowed programmers to explicitly choose portions of a program to use lazy evaluation. Others had adopted a middle-ground between the two extremes, which they referred to as 'lenient evaluation'.

[4]'Streams and Iteration in a Single Assignment Language, McGraw et al. [1983]

2.9. Implementations

A great many implementations of the dataflow concept have been built over the years. This section highlights a small number of them.

An early attempt to design hardware for dataflow was made by Dennis and Misunas [1975], using the concept of 'actors' — nodes which were capable of merging, splitting or directing flow around the graph. Building on their ideas, Davis [1978] proposed a system they named 'Data Driven Machine 1' (DDM1), later classified by Vegdahl [1984] as a 'Tree-Structured Machine'. It was extensible to an arbitrary degree but, as described by its designers, its major disadvantage (and the major disadvantage of any tree-structured machine) was that the "*hard-wired, fixed-tree structure prohibits reallocation of unused PSEs [processor-store elements] to other branches*", meaning it needed significant redundancy in the size of the tree in order to capitalise fully on the concurrency of the algorithms in question. The designers also cited the need for redundant data storage as a major drawback.

Another early implementation, focussing on the software rather than hardware, was the programming language VAL (Ackerman and Dennis [1979]), which evolved eventually into SISAL (McGraw et al. [1983]). By the creators' account, Val was designed primarily for concurrency, in an attempt to move away from languages that closely reflect the von Newmann model of the underlying hardware. The introduction to its manual expressed a desire for a general decoupling of the design of the language from the computer hardware on which it could be expected to run. A similar motivation was expressed later by Nikhil et al. [1986] as an explanation for the development of their dataflow language *Id Nouveau*.

McGraw [1982] published a description and assessment of Val, which expressed approval for its general aims and aspects of its implementation, in particular its 'for-all' feature, which allowed a very high level of concurrency ("*orders of magnitude more concurrency than a machine can exploit*") in independent iteration. He noted, however, that optimisation techniques required further study. It was McGraw who went on to develop SISAL.

Arvind and Gostelow [1982] identified a further opportunity for concurrency in the fact that, in dependent iteration, it may still be possible to execute more than one iteration concurrently. In their proposed system, named the 'U-Interpreter', they suggested achieving this using a version of dataflow that used switches, allowing an arc to store more than one data token, and allowing concurrent invocations of the same node. They did not build an implementation, but suggested that their system could have been implemented on the Manchester machine, which was cited by them as having been under way since 1980 and was described in more detail by its creators a few years later, in 1985 (Gurd et al. [1985]).

Davis and Keller [1982] listed a handful of implementations that had been created by the time their paper was published, including Dynamo (Forrester [1961]), FGL ('Function Graph Language', Keller and Yen [1981]) and GPL ('Graphical data-driven Programming Language', Davis and Lowder [1981]). FGL and GPL, he noted, allowed the programmer to "*draw the program and execute its graphical form*". He lamented the burdensome overhead of graphical displays, but assumed that with improvements in hardware such obstacles

would quickly dissolve. He also pointed out that it was the display of text within nodes that required the greatest resolution, signposting the need, even with very high resolution displays, for a user interface with zoom and pan functionality and allowing pertinent data to be displayed and irrelevant data hidden to avoid unnecessary screen clutter.

By 1984 there had been a wider proliferation of dataflow implementations. Vegdahl [1984], in his survey paper of proposed architectures for functional language execution, listed 23 functional languages under the five different categories, including six data-driven dataflow implementations and a further four demand-driven machines. He also pointed that all still suffered from problems of efficiency.

'The Manchester Project' (Gurd et al. [1985]) was one of those. Its creators claimed "*impressive speedup for programs with sufficient parallelism*". Their system was data-driven and allowed cyclical graphs, using switches ('branch instructions') to exit the loop when appropriate conditions were met, using constructs similar to the 'distributors' described by Davis and Keller [1982]. They had a limited number of 'instruction processors', which were separate from the nodes themselves, meaning that nodes were added to a queue and would be executed when a processor next became available. They evaluated their results for the speedup obtained when adding processors. The results for their test algorithms were slower than a sequential system, but they asserted their belief that system performance could eventually exceed that of conventional language systems "*for a variety of applications*".

Expressing a desire to be consistent with interfaces that programmers will be familiar with, Cox et al. [1989] created a dataflow implementation, Prograph, that was influenced by Microsoft's Windows operating system as it was in 1989. Although the authors tried not to be influenced by textual programming languages, the operations made available were similar to those in textual languages, with their object-oriented class definitions closely reflecting object-oriented programming in textual languages. While arguing that visual languages can be more easily understood, they introduced a range of symbols, resulting in graphs that could not be intuitively understood and required a key to interpret. Like the development of flow-charts that led eventually to the introduction of a 26-page international standard with tables of symbols (International Organization for Standardization (ISO) [1985]), the creators of Prograph appeared to have lost some of the motivating simplicity of the concept.

The Cantata visual programming environment (Rasure and Williams [1991]) was claimed by its creators to be a 'multiparadigm language', with the aim of being able to choose whichever of three paradigms (dataflow, text or form) most suited any particular component of a problem. Dataflow was the principal paradigm, coordinating connections between nodes that utilised either forms for data input or textual languages to define functionality. The code associated with a particular node could be written in a language of the user's choice (compiled to run on the machine in question), could be arbitrarily large and could be imported from external libraries if necessary. They aimed to reduce the relative size of the overhead of dataflow by using the dataflow component at a high level where the computation was large in comparison with the overhead (although did not restrict the

programmer to a predefined level of granularity). They used switches to enable loops and a 'visual hierarchy of workspaces' to reduce visual complexity.

By the time Hils [1992] published his review paper, he was able to refer to 15 different implementations as examples, divided into six application domains. The domains were: sound processing, constructing user interfaces, image processing, science (including LabView, which is still in use today), graphics and general purpose. Of these six domains, at least three — sound engineering, science and graphics — have popular products today that use dataflow as their programming interface.

Vipers (Bernini and Mosconi [1994]) partially implemented the approach advocated by Gelernter and Carriero [1992], by separating the coordination from the computation components. Although they used one sequential language to define node functions, their vision was that any language could be used, and the nodes triggered and coordinated through Vipers without the programmer needing any knowledge of the languages used. A significant amount of effort appeared to have been devoted to devising suitable visual representations, although they stressed that the interface should be reconfigurable, even by users while the system was running.

Forms/3 (Burnett et al. [2001]) was described by its creators as an attempt to explore the boundaries of functional programming and spreadsheets. Starting with spreadsheets as a foundation and seeking to solve its problems, the solutions they suggested had similarities to dataflow graphs. They pointed out the benefit of being able to see the changes on the screen as values are computed.

DFScala (Goodman et al. [2013]) was a library for building dataflow graphs in the Scala programming language, created at the University of Manchester. The authors noted a revival in interest in dataflow programming preceding it. It was part of a large-scale multi-institution project named Teraflux (Giorgi et al. [2014]), commenced in 2009, investigating 'teradevices' (those with 1000 billion transistors and 1000+ cores) and 'extreme-scale' systems. Teraflux took an integrated approach, targeting all aspects of system design, from the programming model at the high level, to hardware design at the low level. DFScala was the dataflow programming component of this project. It was acyclic, side-effect-free, achieved iteration using recursion and allowed a dataflow graph to be nested within another, although it did not fulfil Sousa's (Sousa [2012]) desire for dynamically expanding and collapsing sub-graphs. It was coarse-grained, enabling it to avoid the worst overhead costs, and was further compiled for more efficient execution. Accordingly, it was one of the first implementations for which the creators claimed objectively good performance, as opposed to simple speed-up with increasing concurrency (Goodman et al. [2013]).

Bravo et al. [2014] wrote about their implementation of dataflow (named 'Derflow') in Erlang. They described problems achieving determinacy in Erlang and proposed solving them using a single-assignment data store, in an approach with echoes of the concept of the 'I-Structures' (Nikhil et al. [1989]).

A project with a very different objective but dataflow-like features emerged with the publication of "*Ur/Web: A Simple Model for Programming the Web*" the following year

(Chlipala [2015]). The author noted that the World Wide Web had evolved from document delivery to an architecture for distributed programming, explaining his aim to build a language infrastructure that would make that easier. He aimed to abstract away some details of web programming such as communication with the server, and utilised ideas on reactive programming by implementing the user interface in dataflow. He listed six applications that, at the time, were using Ur/Web for commercial applications, and claimed good performance for some applications on multi-processor machines (but conceded poor performance in tests involving multiple database writes).

More recent products that use dataflow include Nuke (video compositing)[5], Max6 (sound engineering)[6], Intel TBB[7], Tensorflow[8], and Google's Cloud Dataflow[9] product.

2.10. Open Problems

Johnston et al. [2004] published one of the most comprehensive surveys of the field to date. They covered the origins of dataflow, its evolution through experimental hardware architectures and execution models, described approaches to problems, including iteration, synchronisation and efficiency, and summarised the notable implementations. Amongst the developments they described was the advent of hybrid dataflow, combining dataflow coordination of nodes combined with von-Neumann execution within nodes, suggesting that hybrids could deliver better performance than either model alone.

They listed four open problems, as they saw them at that time: provision of iteration in textual dataflow languages, iteration in visual dataflow languages, use of data structures and non-determinacy. With data structures they also listed examples of approaches to operating on them efficiently, but express dissatisfaction with progress at that time. They believed that "*any practical implementation of dataflow must include an efficient way of providing data structures*". The determinacy of dataflow, they noted, had often been promoted as an advantage, but they observed that in practical applications non-determinacy is often mandatory, leaving the successful control of non-determinacy as an open problem.

Sousa [2012] wrote a short survey which identified two additional open problems — abstraction of complex graphs, and debugging. He suggested grouping nodes hierarchically as a solution to the first problem, and visual feedback as a solution to the second. He saw the lack of good quality visual editors as the primary problem to be solved.

Vegdahl [1984] had, much earlier, made the point that it is the existence of a static tree in functional languages that makes them, in principle, easier to debug, compared with the dynamically changing underlying state that characterises sequential languages. He pointed out that one language in particular, Poplar (Morris et al. [1980]), included a function specifically for debugging which allowed nodes to be tested for expected values as they were computed.

[5] `https://www.foundry.com/products/nuke`
[6] `https://cycling74.com/products/max/`
[7] `https://software.intel.com/en-us/intel-tbb`
[8] `https://www.tensorflow.org/`
[9] `https://cloud.google.com/dataflow/`

Bainomugisha et al. [2013] included, in their survey on reactive programming, a discussion of the problems when dataflow is used to build a user interface, and highlighted two as unsolved problems: multi-directionality, and what they called 'glitches' — when an incorrect output momentarily appears, as can happen particularly in distributed systems. They identified glitches as "*a potential sweet spot for future research*".

Multi-directionality issues apply in particular to user interfaces. The example the authors provide is a tool to convert between Celsius and Fahrenheit, with two data entry boxes, in which whenever a user enters one, the other should update. This seems to have been addressed in the past by allowing multi-directionality with some termination clause, or by introducing phases into the interface, each phase being one-directional in a different direction, or bypassing the problem by attaching the directionality to more abstract components such as 'event sources'.

2.11. Summary

This chapter summarises the key concepts in dataflow. Areas of particular interest are the concept of dataflow as a coordination language (Section 2.7), the search for a representation of iteration consistent with the dataflow paradigm (Section 2.5) and the partial evaluation of functions (Section 2.2). The next chapter discusses the principles of software engineering and architecture that can be applied to the design of a programming paradigm.

Chapter 3

Software Engineering

This chapter summarises previous literature on software engineering and describes the evolution of the development methodologies and the principles of software architecture. The purpose is for us to be able to use those principles to design a paradigm which, ideally, embeds them so tightly that it becomes easy, unthinking or automatic for a programmer to apply them without needing detailed knowledge of programming good practice. The aim is to establish a link between principles and the programming paradigm design, such that changes in the consensus on software engineering good practice will lead to changes in the design. Both the principles and the manner in which they are applied may be open to debate. The importance of applying them should be less controversial.

3.1. Origins

In 1968, the NATO Science Committee[1] sponsored a conference in Garmisch, Germany to try to solve the prevailing problems in software (Naur and Randell [1969]). The conference was named the 'NATO Software Engineering Conference', coining the term 'Software Engineering' in the process. The organisers described their choice of that term as "*deliberately provocative*". A glimpse of how this may have been perceived at the time is provided by the fact that, at a follow-up conference a year later, a participant wrote a parody that likened it to trying to 'engineer' great works of art (Randell [1996]). Despite such mockery, the term caught on (Randell [1996]).

According to the report on the 1968 conference (Naur and Randell [1969]), participants wanted to tackle the reputation of software projects for cost and schedule overruns, a problem that some participants referred to as the 'software crisis' or 'software gap' (meaning the gap between project estimates and delivery). The idea of a software crisis also took hold and was still in vogue nearly three decades later, with the publication of articles and papers such as 'Software's Chronic Crisis' (Gibbs [1994]) and 'The Systems Engineer and the Software Crisis' (Johnson [1996]).

[1] The NATO Science Committee was formed in 1958 to "*improve corporate effectiveness in the area of the Alliance's scientific and technological resources*", and operated until 1996, when its functions were taken over by the NATO Research and Technology Organisation. A brief history, provided by NATO, is available at: `https://www.sto.nato.int/Pages/drg-history.aspx`

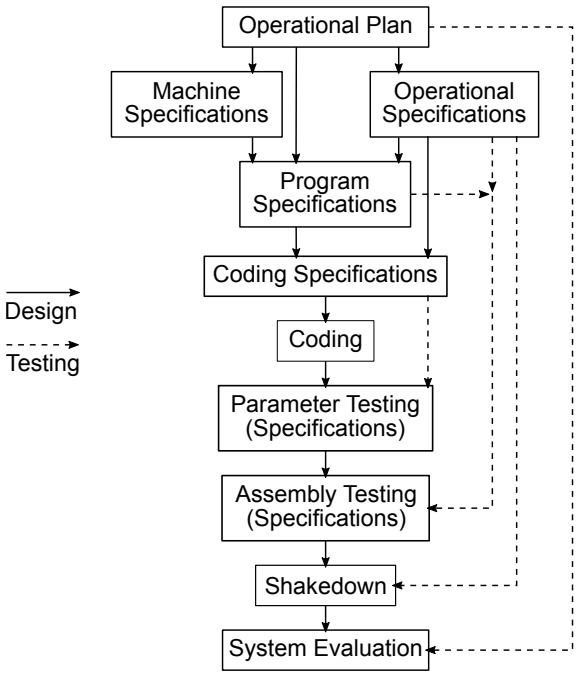

Figure 3.1. An example of a stage-wise process model, reproduced from Benington [1956]. Benington provided this as an example of a program production process, involving specification of the whole system before embarking on any coding of the system.

The field of Software Engineering came to focus substantially on efforts to solve this problem through better project management. To this end, a number of project management methodologies were proposed (also called 'process models' or 'development methodologies').

3.2. Development Methodologies

One early account of a project management methodology was given by Benington (Benington [1956]), illustrated in Figure 3.1. Its aim was to reduce the risk and uncertainty through upfront design and analysis; it involved a series of (mostly) sequential steps, starting with planning and specification, followed by coding and ending with testing and evaluation. Processes similar to this have been called, by some, the 'stage-wise' model (Boehm [1986]).

There has been a certain amount of confusion over the origin of and terminology used for stage-wise models of this form. The term 'waterfall' is often used to refer to a similar sequential process, with Royce [1970] credited as its inventor. Royce's paper included a diagram (reproduced in Figure 3.2), versions of which are usually used to illustrate the waterfall process without mentioning his own comment about it, that the process it describes is "*risky and invites failure*". Boehm [1976], who has also been credited with the waterfall model (for example, by Basili and Rombach [1988] and Giddings [1984]), included a similar diagram, calling it the 'Software Life Cycle'. Others, such as Holthouse and Greenberg [1978], referred to diagrams of this form as the 'Traditional Software Life Cycle'. Royce actually advocated iteration in the development process, and his recommended process model involved completing every step twice, as shown in Figure 3.3.

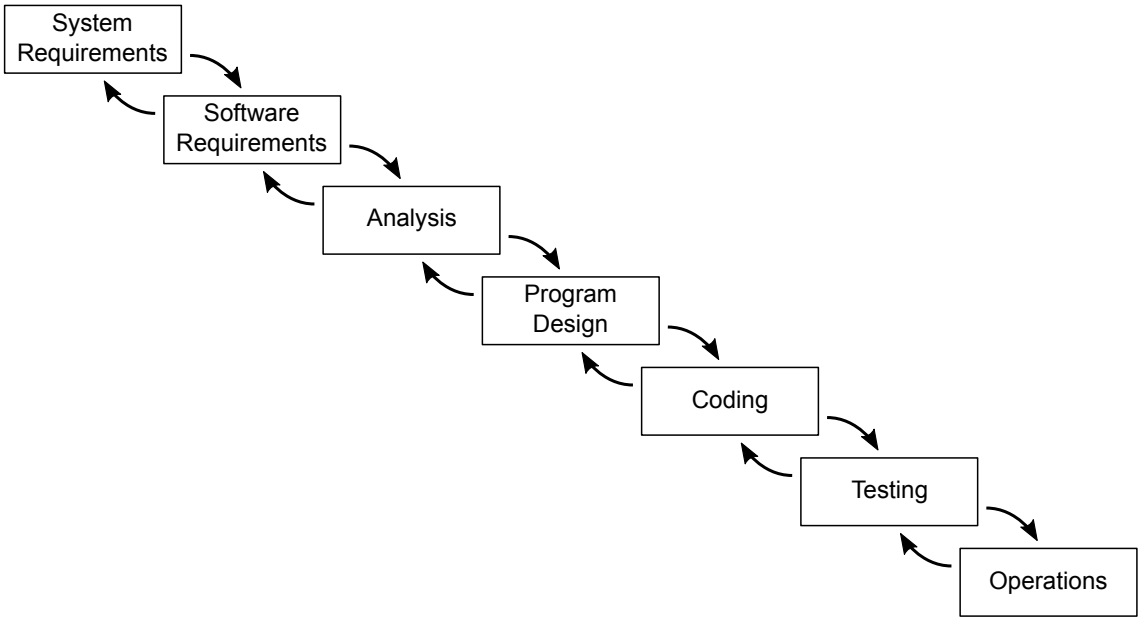

Figure 3.2. Not Royce's recommendation. Royce commented about this process that it is "*risky and invites failure*". Reproduced from Royce [1970].

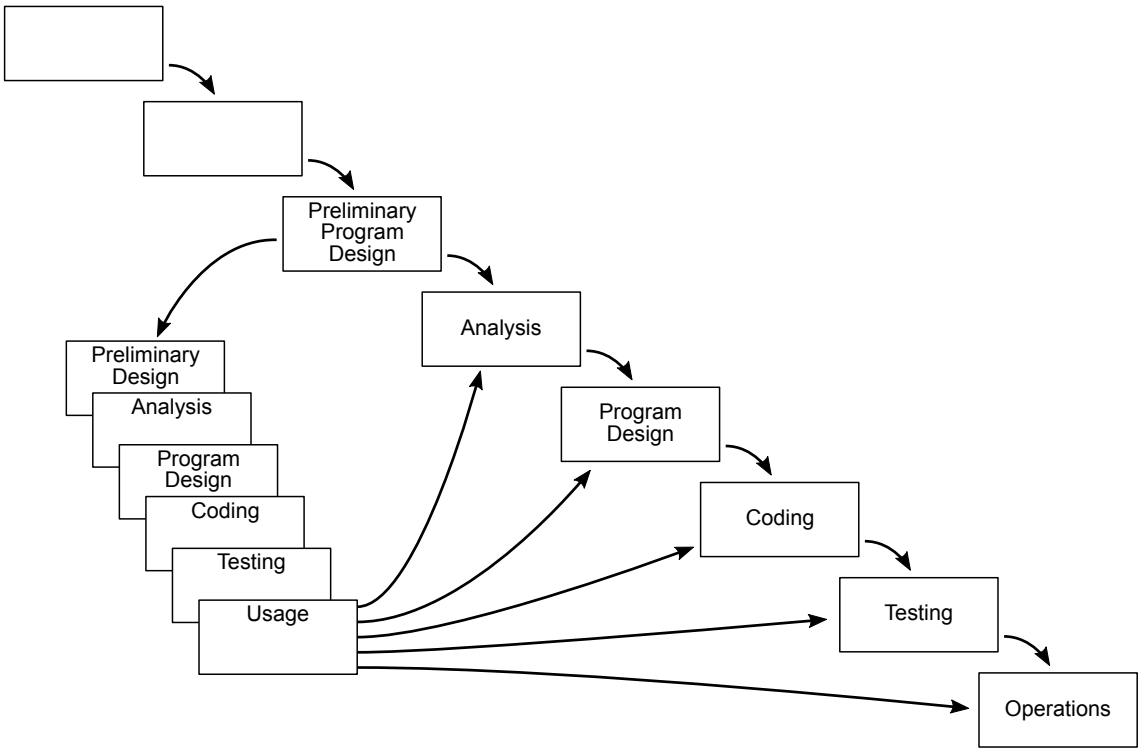

Figure 3.3. Royce's recommended process model. He advocated completing "the entire process ... in miniature, to a time scale that is relatively small with respect to the overall effort", so that the "the version finally delivered to the customer ... is actually the second version". Reproduced from Royce [1970].

Project management methodologies emanating from Computer Science and Software Engineering have, in general, been iterative. Iterative methodologies for managing software projects have included 'Stepwise Refinement' (Wirth [1971]), 'Iterative Enhancement' (Basili and Turner [1975]), the 'Domain Dependent Life Cycle' (Giddings [1984]), the Spiral model (Boehm [1986]), the Rational Unified Process (Kruchten [1995, 1996])[2], Scrum (Schwaber [1995]), Extreme Programming ('XP' — Beck [1999b,a]) and Agile (Beck et al. [2001]).

Even at the time of the earliest of those, iteration in software development was not a new innovation. The report on the 1968 NATO Software Engineering Conference stated that "*the need for feedback was stressed many times*" and Boehm [2006] referred to the avoidance of sequential processes as a key lesson of the 1950s. According to a history of the subject written by Larman and Basili [2003], the idea of iterative development had origins as early as the 1930s, was used for a significant military hardware project in the 1950s, and the recollection of a team member suggested iterative methods being used in software in 1957.

The wider adoption of iterative methodologies may, however, have been inhibited by the desire of government and commercial procurement departments for upfront estimation. With iteration being inherently open ended, estimation procedures dating back to the late 1960s were prone to assuming a stage-wise approach (for example, Nelson [1967]; Wolverton [1974]; Lord et al. [1977]). UK government project management guidelines encouraged a stage-wise development model through the project management methodology PRINCE2 (1996) and before that PRINCE (1989) and PROMPT (1979) (Newman [1997]; Weaver [2007]). US military documents encouraged or imposed similar principles through published standards such as MIL-STD-498 (1994) and others dating back to the early 1970s (US Department of Defense [1994, 1985a,b, 1983, 1972]; Cooper [1981]).

Despite many dissenting voices and being contradicted by empirical evidence, variations of the stage-wise development process continued to be seen by many as the 'correct' way to develop software. Exemplifying this, Madden and Rone [1984] described a variation of the stage-wise model as 'ideal', even while explaining why the stage-wise approach could not work for the Space Shuttle project and had to be abandoned in favour of an iterative process.

The problem seems to have been that, in opting for upfront specification, managers had failed to realise the high likelihood of change taking place within the project and its environment, and that a great deal of the uncertainty and risk originated at the very root of the project, in the project goals or specification. As described by Gilb [1981], the project goals are "*usually poorly stated and incomplete*". Boehm [1986] put it in more general terms, saying of the waterfall model that it "*does not adequately address the concerns of ... organising software to accommodate change*". On a similar theme, Cave and Salisbury [1978] stressed the high likelihood of change on long-running projects and the need for

[2] At the time, Kruchten called it the Rational Development Process, 'Rational' being the name of the company he worked for. The term 'Rational Unified Process' and its acronym RUP were attached to it later.

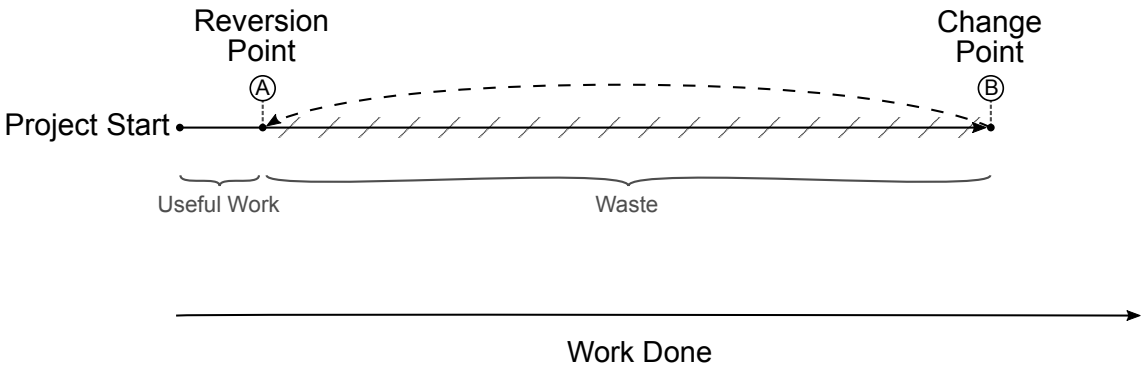

Figure 3.4. An Efficiency Chart. The horizontal axis represents work done; the project starts at the left and moves to the right as it progresses. At point B, the 'Change Point', new information is discovered that requires a change of direction and invalidates some of the work already done. As a result, the project must revert to some earlier point — point A, the 'Reversion Point'. This chart shows the amount of useful work done and the work wasted as a result of having to revert from point B to point A.

"*continual user interaction, refinement, and review of specifications*".

A detailed upfront specification meant that a large amount of work followed on from potentially flawed goals. This served to delay the point at which errors were discovered and maximise the work wasted before they were. Equally detrimental to the project was that a rigidly defined specification was unable to bend when technical difficulties were encountered. Whereas a small change to the specification can sometimes make a difficult project achievable, an immovable specification can lead instead to uncontrollably escalating costs and timescales. It is better to allow the specification to change and the management to make new prioritisation decisions when needed.

Errors in the specification, which can occur for many reasons, may not be discovered until a working version of the software is delivered and users are able to test it. In the worst case, this might mean having to rewind the project almost to the beginning; and even then, when a new specification is written, it might yet be subject to further change. This process can result in a great deal of wasted work. Furthermore, the longer a project lasts the more likely it is that external changes in the business, technological or legal environment of the project will invalidate previous decisions, requiring yet more changes to the project or its goals.

I propose the introduction of a type of diagram with which to view this process, shown in Figure 3.4. This could be called an *efficiency chart*. Movement along the horizontal access represents work done. At point 'B', labelled the 'Change Point', a change has to be made. This could be because knowledge is uncovered, such as an error in the specification or an incorrect technological assumption; or because an event has occurred, such as a change to the business or technological environment. The project is set back and has to revert to some earlier point, labelled point 'A', the 'Reversion Point'.

The position of reversion point represents the head-start in the new direction provided by the work already done. The work to the left of the Reversion Point is still useful; the work to the right of the Reversion Point has been wasted and would not have been done had the new information been available at the start.

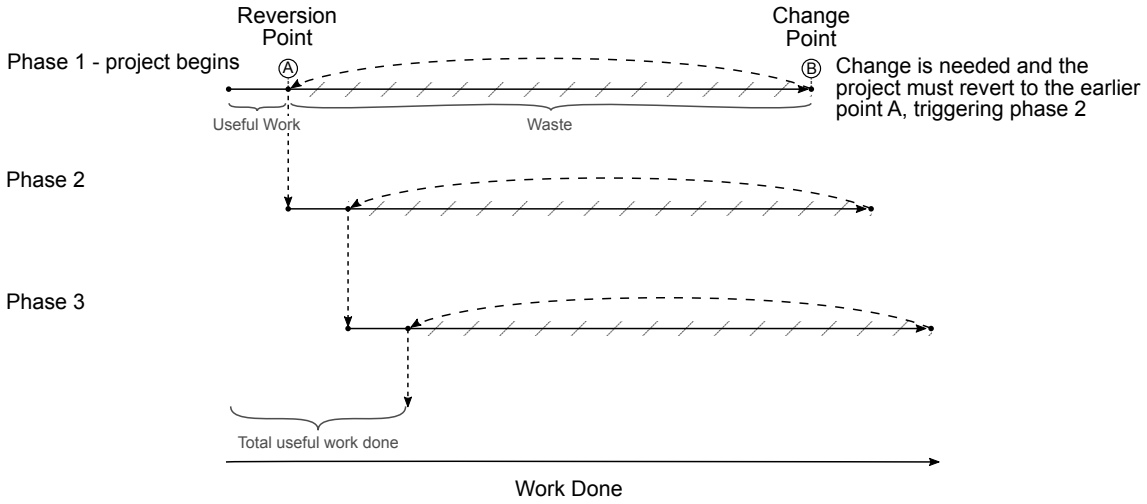

Figure 3.5. Multiple phases of work. A series of phases of work in which each change to the project triggers a new phase. The project begins at the top left with Phase 1. At point B, the first 'Change Point', a change is needed and the project reverts to the earlier point A, its 'Reversion Point'; at which point a new phase begins. The total useful work done, shown at the bottom, is the work that would have been required to make the same progress had the knowledge that triggered the change points been available at the start. The total waste is the sum of the shaded areas.

If Figure 3.4 represents the first phase, we could plot sequential phases of the project as shown in Figure 3.5. Here we see a similar process repeated three times, labelled as phases 1, 2 and 3. Each phase features a change point, triggering reversion to a previous position and the start of a new phase. This chart shows the total useful work done — the work that would have been required to get to the current position had the information that triggered all previous change points been available at the start — together with the work wasted in each phase. There could be many such phases in a project.

Although Figure 3.5 gives the impression that a phase necessarily takes a project forwards, there is no reason why a change could not also cause a reversion to before the start of the current phase, eliminating the progress of the current phase and eating into the progress of a previous one. The project will continue in this manner, some phases bringing about progress, others not, until the project achieves its goals or is halted for some other reason.

The purpose of a development methodology is to maximise the efficiency with which the project goals can be achieved, while accommodating the fact that those goals themselves may change during the project. As suggested by Boehm [1986], the biggest component of this task has always been the need to manage change in the project. Such changes can emanate from a variety of places, including errors in the project goals or specification, the discovery of technical knowledge or changes in the business, financial or technological environment of the project.

As is clear from Figure 3.5, the efficiency of the response to this change can be improved by doing two things: moving Change Points to the left (uncovering changes sooner) and moving Reversion Points to the right (minimising the impact of those changes). Kent Beck, in his book *Extreme Programming Explained* (Beck [1999b]), made the assertion that the cost of making a change to software increases exponentially as the project progresses. Possible hyperbole aside, his general point, that change become more difficult as a project

progresses, highlights the importance of uncovering change sooner, and the cost of failing to do so.

The assumption on which iterative development methodologies are based is that the best way to uncover the most important changes is to obtain rapid feedback — to build a working version and put it to the test as quickly as possible. This uncovers errors in the goals and specification by giving users a cheap working version, and uncovers bad technological assumptions by putting them into practice in skeleton form.

It is this thinking, and perhaps the validation of it, that has seen recommended iteration times shrink over time. Whereas the stage-wise process (Benington [1956]) did not use iteration at all, Royce [1970] proposed that a project should have at least two iterations; and methodologies specifically intended to be iterative have included Madden and Rone [1984], who proposed six month iterations and Schwaber [1995], who proposed iterations lasting a few weeks. Many development teams have put the principle into practice by using time-boxed iterations as short as a day or less (Larman and Basili [2003]; Meyer [2014]).

The choice of iteration time is a balance between the cost of obtaining feedback and the value of having feedback. As automated deployment tools have improved, the cost of obtaining feedback has shrunk. For web-based applications, the technique of 'continuous integration' enables the immediate automated roll-out of changes to end users as they are made, making iterations of a day or less more common (Meyer [2014]).

Although development methodologies are designed to provide general guidance, they do not substitute entirely for intelligent decision-making about which activities should occur when. They encourage managers to focus on making the high-value discoveries early on, and the iterative methodologies in particular direct managers towards unknowns in the goals and specification. However, while goals and specifications are a focus, what constitutes high-value knowledge does still vary from one project to another. We can say, more generally, that to minimise waste projects should be planned in such a way as to make the most critical discoveries early on, and to maximise the flexibility to change when those discoveries are made. It is the goal of the development methodology, in general, to narrow the gap between each change point and corresponding reversion point.

One component of narrowing the gap is about how the project activities are organised; another is about how the software is structured — its architecture — which is discussed in the next section.

3.3. Software Architecture

In the late 1980s the field of Software Architecture emerged as the branch of Software Engineering concerned with system and software structure. Zachman [1987] used (and sought to define) the term 'information systems architecture', through a direct analogy with the processes involved in classical architecture and the architect's role in translating requirements of the client and the real-world constraints into plans for the builder. He sought to

distinguish the discussion about systems architecture from the orthogonal discussion about development methodologies (which he called 'strategic planning' methodologies) that had been the focus of much of the previous literature in Software Engineering. Before this point, software architecture was more often referred to as the software 'structure'.

Shaw [1989], two years later, had started referring to it as 'Software Architecture', and presented a roadmap for defining and formalising a set of architectural styles that would put the field on a firmer footing. Perry and Wolf [1992] took a step along that road, attempting to determine how Software Architecture should be defined. At that time they still felt the need to justify their use of the term, but expressed their belief that the 1990s would be be, as they put it "*the decade of software architecture*". They listed a few architectural styles as examples but avoided any definitive list, instead leaving open the possibility of new styles being developed.

In an introduction to the fledgling field of Software Architecture in 1993 (Garlan and Shaw [1993]), the authors listed what they saw as the common architectural styles at that time, while acknowledging that real-world systems would usually utilise several. They also made reference to overriding architectural principles (without listing them), which they claimed were already in use by software engineers.

By 2006, Shaw and Clements [2006] claimed that the previous fifteen years had been a golden age of Software Architecture, in which the term had caught on, spawned a proliferation of ideas and become a mature discipline. It had entered university courses, moving over that time from post-graduate to undergraduate level. As pointed out by Kruchten et al. [2006], Software Architecture conferences had been held, journals had published special issues on the subject, and a good handful of books had been published.

Books on Software Architecture list both the architectural principles that guide the design of a computer system, and the architectural styles that can be utilised. The principles of software architecture are fundamental to system quality. A good design should obey all of the principles. The architectural styles are, rather, a way of classifying systems; a menu from which system designers can pick. The choice of architectural style is driven by the requirements and constraints of the project, accommodating the organisation and distribution of the system's users and components. A system may utilise one or more architectural styles, or devise new ones if required, but should, in any configuration, obey the principles. The architectural principles and styles are discussed in the following sections.

3.4. Architectural Principles

As with the software development methodology, it is useful to view the architectural principles from the perspective of the efficiency chart, shown again in Figure 3.6. Like the development methodology, the architectural principles are also aimed at narrowing the gap between the change point and reversion point. Whereas the development methodology does so through project management, the architectural principles do it through the structure of the software. This means making it easier to discover knowledge early on, with principles such as 'testability'; and making the software more flexible with principles

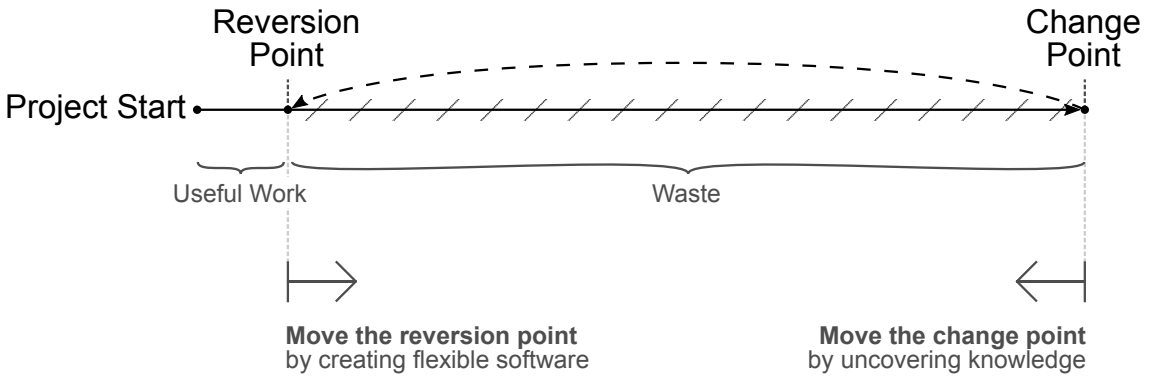

Figure 3.6. Narrowing the gap. Seen on an efficiency chart, software engineering has two goals: to move change points to the left by uncovering knowledge; and to move reversion points to the right by making the system flexible.

such as — surprisingly — 'flexibility'. Good management and good architecture are both necessary to be able to narrow the gap: 'testability' is no use if nobody is trying to test anything and trying to test things will be an arduous process if the software is not testable.

We can classify the principles by their goal. Some are aimed at making it easier to uncover knowledge (with the intention of moving change points to the left), some are aimed at increasing flexibility (with the intention of moving reversion points to the right), and many are lower level structural characteristics aimed at helping to achieve those higher level goals. Finally, there are characteristics that could be classed as desirable traits, whose relative importance varies depending on the project environment.

3.4.1. Uncovering Knowledge

These principles are aimed at moving the change point in the efficiency chart to the left by making it easier to uncover relevant knowledge (and, therefore, to make the required changes) earlier in the project.

Team Communication

In his book 'Domain Driven Design' (Evans [2004]), Evans' key insights revolved around the importance of communication in software development, in particular between what he called 'domain experts' (usually clients) and software developers. He described projects in which key terminology was understood differently by the two groups for prolonged periods of time. His suggested solution was to develop a simplified 'domain model' and corresponding 'ubiquitous language', through extensive discussion between the two groups, to express as clearly and simply as possible the important parts of the business as well as the technical functionality involved in implementation. This helps programmers to better understand the requirements and constraints, and helps clients to spot when the project deviates from their goals.

Evans went into greater detail on the subject, but the point that underlies it is that software development is a collaborative pursuit, in which communication is of critical importance. Meier et al. [2009] also listed team communication as a benefit of the Domain Driven architectural style, explaining that it helps to communicate knowledge between team members.

Incrementality

Incrementality is the main goal of the development methodologies discussed in Section 3.2. In the context of the architectural principles, incrementality means that the structure of the software should be able to accommodate making many small changes and extensions to its functionality rather than having assumptions so deeply embedded that small changes to assumptions result in large changes to the code. This may mean, for example, avoiding unnecessary details at an early stage and building components to be flexible and interchangeable. The software should, at all stages, be flexible enough to accommodate later changes and extensions to its functionality.

The principle of incrementality is sometimes expressed through a related principle as the need to 'minimise upfront design'. Vogel et al. [2011] suggested that attempts to design a system upfront often fail, blaming communication problems between software designers and domain experts, and a tendency of software designers to focus too much on technical details without noticing their embedded assumptions about the requirements. If application requirements are unclear or subject to change, heavy upfront design based on those shaky foundations is likely to be wasteful.

The suggestion to 'minimise upfront design' should not necessarily be taken as an excuse to be lazy. Rather than being thoughtless in design decisions, its key point is to avoid committing too deeply to design decisions that are likely to change; decisions should be made with a view to that potential change. That said, in some cases it may be cheaper to build twice than to build with every flexibility built-in — it is a decision the designer should consciously make.

Reference to Use Cases

This principle, included by Vogel et al. [2011] in their list of principles, suggests that use cases help to reduce the number of specification errors. Vogel et al. also pointed out that it helps reduce the likelihood of an architecture exceeding the requirements of the system. It contributes to the principle of team communication, as well as helping domain experts or project sponsors themselves understand the goals they are setting for the project.

Testability

Testability both uncovers knowledge and improves flexibility. It helps to expose bugs early in the development process, which brings change-points forwards, and improves flexibility by making the software less susceptible to breakage when changed. Meier et al. [2009] recommended, specifically, using automated quality assurance techniques.

Structural characteristics such as modularity, abstraction and explicit communication (discussed below) contribute to testability by providing clear units of functionality on which tests can be carried out and clear behaviours that can be tested. Automated tests contribute to the goal of incrementality (discussed above) as well as maintainability and traceability (discussed below) by catching errors and providing an explicit expression of how the module being tested is expected to behave.

3.4.2. Increasing Flexibility

These principles are aimed at moving the reversion point in the efficiency chart to the right by making the software more flexible. The principle of testability falls into both categories by uncovering knowledge and improving flexibility and, since it was included in the previous section, is not included separately in this list. Other principles that contribute to flexibility are discussed below.

Flexibility

Flexibility is the most self-evident principle in this category. It is the ultimate goal of all others and is stressed repeatedly throughout the literature on software architecture (for example, by Gamma et al. [1995]; Evans [2004]; Vogel et al. [2011]). It is also the most important way to achieve the principle of incrementality. Some texts include the similar principle of 'extensibility' (for example, Meier et al. [2009]; Gamma et al. [1995]) or the instruction to 'design for change' (Vogel et al. [2011]; Gamma et al. [1995]).

As simple as it is to express the need for flexibility, its implementation is not necessarily easy; it requires the other principles and of all the structural characteristics discussed in Section 3.4.3. As pointed out by Vogel et al. [2011], it is difficult to know where to expect changes to occur. They suggested that, although clues may be provided by ambiguities in the specification or the designer's prior experience of similar systems, the best way to accommodate unexpected change is to apply the principle of low coupling (introduced below, in Section 3.4.3).

Simplicity

Vogel et al. [2011] called it 'avoidance of superfluous complexity', Meier et al. [2009] called it 'low complexity' and 'management of complexity'. All stressed the need for software systems to be easily understood. As Vogel et al. pointed out, "*complex architectures are prone to error*".

Simplicity contributes to flexibility and improves communication between members of the team and between software developers and domain experts. As with the principle of flexibility, most of the other principles are ultimately aimed at helping to achieve it, reducing complexity and achieving some level of simplicity — of structure, of behaviour, and of interactions between components.

The actual simplicity of a system may in fact be less important than its understandability by those working with it. Some complexities lend themselves to human aptitudes better than others. Conceptual tools, visualisations and expressive terminology all help to make a system more understandable. Programming environments have a big part to play. If a programming environment can present an appropriate subset of the system, it can increase its understandability significantly, whether that is a simplified overview or focussed detail of a single module. A range of levels of detail is needed to be able to understand every aspect of the system, in every case making important details visible while hiding the rest.

Others principles also help: modularity, abstraction, high cohesion, low coupling and explicit communication, all introduced below in Section 3.4.3, contribute to the understandability of a system.

Maintainability

Maintainability — being cheap and easy to maintain — is inherently desirable, since all software systems must eventually move into a phase of being maintained. Cave and Salisbury [1978] considered it so important that they included it as one of only two attributes measured when quantifying software quality (together with availability). Meier et al. [2009] described maintainability as one of the goals of software architecture, and they described many of their recommendations as being aimed at improving maintainability.

Maintainability usually requires flexibility, since the need for maintenance only comes about when something in the external environment changes and the system must be changed accordingly. As with flexibility and simplicity, most of the structural characteristics described in Section 3.4.3 contribute to it in some way.

Traceability

Vogel et al. [2011] listed traceability as one of the key principles of software architecture. It refers to the ability to trace the origin of architectural decisions. There are generally two approaches to achieving traceability: through documentation, and through the use of self-documenting code. As described in the principle of 'self-documentation' in Section 3.4.3, traceability through self-documentation is generally seen as preferable. This means using well-designed structures, well-chosen and consistent names and appropriate commenting within the code, all of which reduce the time-consuming need to maintain and refer to separate documentation to understand the structures that have been built.

Reusability

Reusability of components makes it easier to make changes because it reduces the number of places in which changes have to be made, and thereby reduces the likelihood of errors being made. Meier et al. [2009] referred repeatedly to the importance of reusability, and listed a principle they called 'Don't Repeat Yourself' (or 'DRY') as one of their five key principles of software architecture. Shaw and Garlan [1996] included reusability as a key requirement of an architectural style, and Gamma et al. [1995] stressed its importance, with many of their design patterns aimed at improving reusability.

The principle of reusability applies not just within software systems but also between them. Van Der Linden and Müller [1994] and Garlan et al. [1995] (and many others) argued for wider interchangeability and interoperability of components that would enable large-scale software systems to be built from previously existing reusable building blocks provided by others. Both discussed approaches to achieving that goal, suggesting the use of other architectural principles (such as abstraction, high cohesion and low coupling, all introduced below in Section 3.4.3) to help achieve it.

Interchangeability

Meier et al. [2009] listed interchangeability as a benefit of some of their software architectures (such as a component-based architecture). This is aided by a principle discussed by Gamma et al. [1995] which they referred to as 'program to an interface, not an implementation'. It is also assisted by the principles of low coupling and explicit communication, both introduced in the next section.

3.4.3. Structural Characteristics

These principles relate to the software and system structure and are aimed at making it easier to deliver the higher level principles of uncovering knowledge (Section 3.4.1) and increasing flexibility (Section 3.4.2).

Modularity

This principle was described by Vogel et al. [2011] by saying "*the architecture should consist of well-defined system building blocks with clearly distinguishable functional responsibilities*". They went on to describe its contribution to other principles, including interchangeability, high cohesion, flexibility, extensibility and reusability. Similarly, Wirth [1971] argued that the degree of modularity of a system determines its adaptability to change or extension.

Modularity also makes it easier for large teams to work on a project by increasing the number of separate components that team members can work on simultaneously without risk of interfering with each other's work. Some of the design patterns described by Gamma et al. [1995], such as the Builder pattern, were described as helping to improve modularity.

Abstraction

Abstraction usually means exposing the important details of an object while hiding the rest. It was included as an architectural principle or used as a basis for other principles by Gamma et al. [1995], Evans [2004], Vogel et al. [2011] and Meier et al. [2009]; with some also expressing it using phrases such as 'hiding implementation details' or 'information hiding'. Meier et al. described a principle they called 'encapsulation' as having similar goals.

A similar principle is one listed by Meier et al. [2009], referred to as the 'Law of Demeter' ('LoD') or the 'Principle of Least Knowledge', that the programmer should avoid components utilising knowledge of the implementation details of other components. It achieves broadly the same effect as abstraction, but putting the onus on the user of the module to avoid using internal details rather than on the module to hide them.

Another similar principle was listed by Gamma et al. [1995], which they called 'program to an interface, not an implementation', suggesting that a programmer should commit only to an interface, not its implementation details. It was a common theme of the design patterns in their book.

Abstraction was described by Shaw and Garlan [1996] as a key requirement of an architectural style and was used by Shaw [1995] as a criterion on which to assess architectural styles. As pointed out by Meier et al. [2009], abstraction helps achieve the principles of low coupling and explicit communication (both described below).

High Cohesion

Vogel et al. [2011] described cohesion as "*a measure of how much a building block is self-contained*. Gamma et al. [1995] described it as gathering related operations together and, by implication, separating unrelated ones. This principle is sometimes described as the 'single responsibility' principle (for example, by Meier et al. [2009]) or simply 'isolation' (by Gamma et al. [1995])

High cohesion ensures that if something related to a particular area of functionality needs to change, the code for achieving is all in the same place and is not intricately linked (or 'coupled') with other unrelated functionality. For this reason it is often paired with the principle of low coupling (described below), and the two are sometimes tied together by another principle, not listed independently here, referred to as 'Separation of Concerns'.

Meier et al. [2009] included 'Separation of Concerns' as a principle its own right; and Shaw [1995] compared architectural styles by assessing them partly on the extent to which they were able to separate concerns within a system. The principle of 'Separation of Concerns' asserts that parts of the software 'concerned with' different aspects of its functionality (or acting on unrelated objects) should be kept separate and that code concerned with related functionality should be kept together. As described by Meier et al. [2009], its aim is to "*minimise interaction points to achieve high cohesion and low coupling*". It is of little value in itself unless it does indeed achieve high cohesion and low coupling. As Meier et al. pointed out, separation at the wrong boundaries would defeat the point by causing low cohesion and high coupling.

Low Coupling

Also referred to as 'weak' or 'loose' coupling. The strength of coupling between two components is the extent to which changes in one affect or require changes in the other. By reducing that need, low coupling improves flexibility. Meier et al. [2009] and Vogel et al. [2011] listed it as a principle, and it was a goal behind many of the design principles described by Gamma et al. [1995]. Other principles such as abstraction and high cohesion help to reduce coupling between components.

The principle of low coupling applies not just between modules but between the program and the platform on which it runs — its hardware and operating system. Gamma et al. [1995] argued that platform dependencies should be limited as a way to increase flexibility and improve maintainability. Java is an example of a programming language designed to do this, by running programs within a platform-specific virtual machine. Although the program is still dependent on the presence of a virtual machine, and the virtual machine is platform-specific, this structure obscures the underlying architecture from the program, making the program indifferent to the platform on which it runs (Lindholm et al. [2014]).

Explicit Communication

Explicit communication provides clarity about how components interact with other parts of the system, which helps achieve the principles of abstraction and interchangeability.

Meier et al. [2009] made multiple recommendations on this theme, such as "*be explicit about how layers communicate with each other*", "*understand how components will communicate with each other*" and "*define a clear contract for components*". Gamma et al. [1995] and Evans [2004] also made multiple recommendations along similar lines.

Acyclicity

Circular dependencies increase coupling and make it difficult to keep track of dependencies between components. Vogel et al. [2011] included the avoidance of circular dependencies as a sub-principle of low coupling, pointing out that circular dependencies involve "*particularly high coupling*" and that "*none of the circularly dependent building blocks can*

be understood or tested without understanding or testing the entire cycle". Evans [2004] agreed, pointing out that one-directional flow makes software easier to understand; his book assumed use of a layered architecture that made this easier to achieve.

Consistency

Consistency reduces the need for programmers to memorise every characteristic of their components and their interfaces. It makes functionality more predictable and makes the program easier to learn and understand. It is stressed repeatedly in the literature on software architecture and is seen in recommendations from Meier et al. [2009] such as "*keep design patterns consistent within each layer*", "*establish a coding style and naming convention for development*" and "*keep the data format consistent within a layer or component*";

It also manifests itself as part of the motivation behind a principle included by Vogel et al. [2011] which they called the principle of 'convention over configuration'. It is better, they explained, for components to use standard assumptions that cover most common usages, so that only adjustments to those standard assumptions have to be configured.

Self-Documentation

Self-documentation reduces the time-consuming need to maintain separate documentation, which is prone to becoming outdated. It was included as a principle by Vogel et al. [2011], and it was a key theme of Evans [2004] that code should be expressive, designed to communicate with people as much as with the machine. Evans recommended minimising the number of separate design documents and stressed the importance of keeping them synchronised with the code. Some sources (such as Meier et al. [2009]) included a related principle they called 'minimise upfront design'; and Aitken and Ilango [2013] found the reduction in separate design documents to be a key difference between agile methodologies[3] and more traditional software engineering techniques.

Vogel et al. quoted Meyer [1988] in saying that the designer should "*try to make every item of information about the system building block part of the system building block*". Well chosen names and naming conventions help to make code self-documenting, as do the other principles, such as abstraction, explicit communication, high cohesion, low coupling and consistency.

3.4.4. Desirable Characteristics

In discussing the principles of software architecture, various desirable characteristics of software are often named as goals. Alongside the obvious performance measures such as speed and low resource use are a number of others that are beneficial to varying degrees depending on the project environment. A generalised system for programming will be used in a wide range of environments so needs to be be flexible to those diverse priorities. Desirable characteristics often mentioned (including by Vogel et al. [2011] and Meier et al.

[3]By 'agile', they meant methodologies that were broadly consistent with the principles of the Agile Manifesto, published in 2001 (Beck et al. [2001]). They included amongst these the Rational Unified Process (Kruchten [1995, 1996]), Scrum (Schwaber [1995]), Extreme Programming (Beck [1999b,a]) and others.

[2009], in describing the relative benefits of the different architectural styles) include those shown below.

- **Scalability** The ability to increase its scale without suffering unacceptable performance costs.
- **Distributability** The ability to distribute its components over multiple machines and locations. As well as being beneficial in its own right, it can add resilience, speed, parallelisability and can improve response times (if it means a service provider can be physically closer to its users).
- **Availability** For a program to be continuously available so that other programs can connect to its interface at any time.
- **Interoperability** For a program to be able to operate easily with other related programs.
- **Discoverability** When a service advertises its existence and publishes the specification of its interface, enabling clients to connect to it automatically. If all programs and modules were services, and were distributed, available, discoverable and interoperable, it would reduce the need for programmers to rewrite functionality that had been written before, and would deliver the much sought-after ability (for example, by Van Der Linden and Müller [1994] and Garlan et al. [1995]) to compose new programs from reusable components.

3.5. Architectural Styles

Architectural styles are chosen based on the goals and constraints of the project, and the balance of priorities of their relative benefits. In Chapter 3 of their book, Meier et al. [2009] described eight architectural styles ('client / server', 'component-based', 'domain driven design', 'layered', 'message bus', 'n-tier / 3-tier', 'object-oriented' and 'service-oriented') which, in practice, are often combined. They listed the benefits of each, including amongst them their ability to deliver many of the architectural principles listed in Section 3.4, such as flexibility, simplicity, reusability and testability, as well as desirable characteristics such as scalability, discoverability and interoperability.

Evans [2004] assumed as a basis for his book the use of a layered architecture and the architectural style that gave the book its title, Domain Driven (or Model Driven) Design. In his model of layered architecture, each layer was a service provider to the layer above, providing high cohesion within layers with abstraction and low coupling between them. Evans's book focussed heavily on the principle of team communication, and in particular on the importance of rich ongoing communication between 'domain experts' (usually clients) and software experts.

Vogel et al. [2011] provided a longer list of architectural styles including, in addition to those listed above, Dataflow, Repositories, Rich client, Thin client, Peer-to-peer, Publish/Subscribe, Middleware, Service-Oriented, Security, and Cloud Computing.

3.6. Summary

This chapter describes the evolution of software development methodologies, highlighting their common emphasis on iterative and incremental development; and distilled and categorised the principles of software architecture applied in designing the structure of computer systems. It introduced the concept of an efficiency chart, showing the process of developing software as a series of blocks of work interrupted by 'change points', at each of which new knowledge or a change in the project environment invalidates some of the work done and means the project must revert to some earlier point, referred to as the 'reversion point', causing the work in between to be wasted (see Figure 3.7). The software development methodologies and the principles of software architecture are both aimed at 'narrowing the gap' — reducing the amount of work wasted when a change point occurs. This is done by moving change points left (uncovering new knowledge and required changes sooner) and moving reversion points right (maximising the previous work that remains useful by making the software flexible).

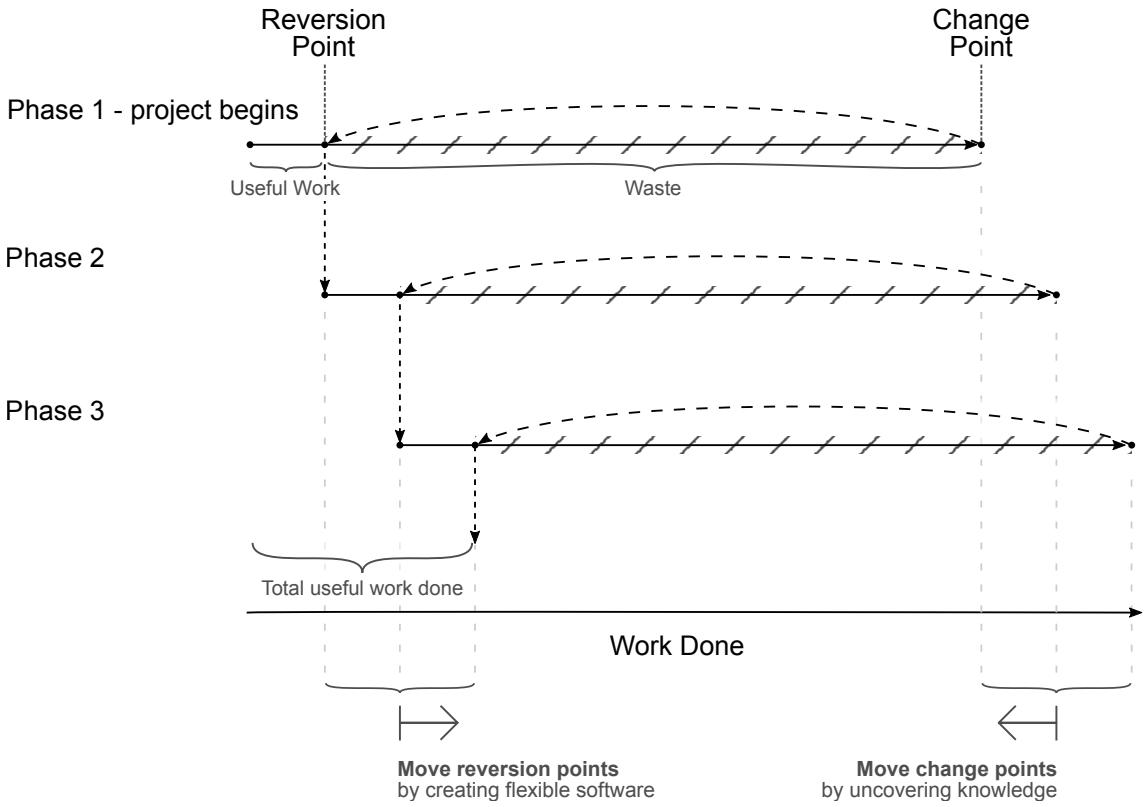

Figure 3.7. An Efficiency Chart. The project begins at the top left with Phase 1. At the first 'Change Point', a change is needed and the project must revert to the 'Reversion Point', with only a portion of the work being salvageable. The process repeats itself, with each phase taking the project a step closer to its goal. Development methodologies and architectural principles improve the efficiency of this process — 'narrow the gap' — by helping uncover knowledge sooner (moving change points to the left) and making the software more flexible (moving reversion points to the right).

In the next chapter the principles of software architecture and iterative development are used to select the most important existing concepts and extend them, developing a model of dataflow as a system of coordination over a distributed network of independent autonomous nodes. Throughout the process, the goal of 'narrowing the gap' is used as a

guide in the design of the system, with the ultimate aim of designing a unified global dataflow coordination system that is simple, understandable, highly programmable and flexible, and can be applied in the widest possible range of applications.

Chapter 4

Definition

Chapter 2 described the history of dataflow with some of its significant advances; Chapter 3 outlined the goals of Software Engineering and the principles of Software Architecture. In this chapter, the two are brought together. Guided by engineering and architectural principles, we unify and extend the best aspects of previous solutions in this field, and discuss how a dataflow system might be used to coordinate communication between a distributed network of autonomous nodes.

Where applicable, the visual notation used to describe it is introduced as we progress. For reference, it is also described in Appendix B. For simplicity, visual components that are unimportant to the discussion at hand are excluded.

4.1. A Coordination System

As described in Section 2.7, Carriero and Gelernter [1989] suggested that the coordination component of a programming system should be regarded as orthogonal to its computation component. Taking this idea, our first step in defining a dataflow system is to think of it as a coordination system rather than a language.

This allows us to make it platform-independent (obeying the principle of low coupling between the system and its hardware, as described in Section 3.4.3) and distributable (one of the desirable characteristics listed in Section 3.4.4), with its only completely necessary requirement being interoperability between nodes. Dataflow in general helps to achieve the principle of incrementality listed in Section 3.4.1 by enabling the programmer to start with the end-product — a node with the described functionality — and work backwards by filling in its functionality later. The inherent distributability of dataflow also means that a program (in this case, a dataflow graph) can be distributed not only over multiple machines, but over multiple owners. Within one dataflow graph, different nodes could be controlled by different people.

Restricting ourselves to a coordination system moves many of the implementation details out of our problem domain. The system itself has no need to know (or care) how data is

stored, what form data takes or how computations are computed. We only need to know that data and instructions can be communicated appropriately between nodes.

Such a system could, for example, be used for computing stock valuations, based on inputs and functions provided by a whole ensemble of different machines and users. One could imagine a stream of currency exchange rate data coming from one provider, weather forecasts coming from a second, a function for forecasting future prices from weather forecasts, current prices and interest rate data from a third and prices for raw ingredients coming from several others. The final output would depend on up-to-date data from every participant; with each node recomputing its output in response to every update to its inputs and passing its outputs in turn to its dependent nodes. An example of such a system is described in a little more detail in Section 4.18.

The most basic definition of our dataflow system will be that it is composed of nodes (making it modular, complying with the principle of modularity listed in Section 3.4.3), with nodes being autonomous data-containing computation-capable entities, connected by directed arcs, each of which connects two nodes and symbolically represents a dependency of one node on data from the other (complying with the principle of explicit communication, also listed in Section 3.4.3). From this base we can start to fill in the details.

Two classes of features are required. First is the set of node behaviours and interactions that are intrinsic to the system. Intrinsic characteristics include the protocol by which nodes communicate with each other, the outward node behaviours and the interfaces through which different components of the system, including nodes, must interact. Features in this category, whilst open to discussion at this stage, must eventually be resolved into an exact specification so that components can be built that comply with the requirements and can interact consistently. This document does not define details of a protocol, but assumes one can be designed in future. We will take for granted that nodes must have an addressing system that allows them to be identified uniquely and to direct messages to each other.

The second class of features is the set of interchangeable components of the system. The user interface, the internal implementation of node behaviour, data storage and processors are interchangeable components. Whereas the system needs a user interface, nodes need an internal implementation and they need access to computing resources, there is no need for the system to specify exactly how the user interface should look, how the internal node implementation should be built or what form the computing resources should take. Components of this type could equally well be substituted with alternatives, with more than one implementation of each existing concurrently within the same system. Features of this type discussed in this document are suggestions only, and are amongst many that could be built.

4.2. Functional Purity

As described in Section 2.3 (Side-Effects), although there are those who believe that purely functional languages are better for their purity, many others (for example, Rum-

baugh [1977]; Davis and Keller [1982]; Hudak [1989]), have recognised the need for non-determinism in practical systems and several (such as Dennis [1974] and Arvind et al. [1977]) have proposed doing so in dataflow through the provision of explicit non-deterministic structures. Whiting and Pascoe [1994] listed six languages that allowed non-determinism of some kind. One problem with the notion of functional purity is that it appears not to have had a universally agreed definition. Sabry [1998] noted that two frequently-used informal definitions (referential transparency and independence of order of evaluation) themselves lack universally agreed definitions — echoing a similar point made earlier by Søndergaard and Sestoft [1990]. In attempting to formalise a definition of functional purity, Sabry based his instead on the commonalities between the languages that people seemed to agree were strictly pure. In dataflow specifically, Johnston et al. [2004] defined a notion of *pure dataflow* as including both determinism and the exclusion of side effects (relating it closely to functional purity), whilst acknowledging that "*if dataflow is to become an acceptable basis for general-usage programming languages, non-determinacy is essential*".

Since the design of a purely functional language or a pure dataflow language has never been the intention here, we do not explore the formal or informal definitions in any greater depth. As described in Chapter 3 (Software Engineering), rather than attaching rigidly to any notion of purity, the aim of this work is to devise a system to maximise overall utility and programmability by designing it around the principles of software engineering and architecture. It is led, primarily, by consideration of how such a system might be used. It deviates from more traditional discussions of dataflow in being envisaged as a live, interactive system, existing in a dynamic environment in which it must continue to operate effectively even with programs in active development, and with data, node contents and the connections between them potentially subject to frequent change.

It is worth pointing out a few things about the system being designed here. First that we can, if needed, include in the node definition a mechanism with which to identify nodes explicitly as being deterministic or not, in an approach with similarities to those proposed by Dennis [1974] and Arvind et al. [1977]. In our design process we make a presumption that nodes will *not* have side-effects and will result in deterministic outputs, but include capability to have side-effects and be non-deterministic where needed. To account for the need for side-effects, a discussion of deliberately triggering nodes is included in Section 4.7. Likewise, non-deterministic nodes could be executed in the same way, or could be triggered by external sources or run to a schedule as discussed briefly in the Further Work chapter (see Section 7.1.2 — Subscription Types, Scheduling and Throttling). In this way, the programmer can deliberately cause non-deterministic code to run by providing an updated input as a trigger, or could prevent it from running by omitting to provide that input. Conversely, nodes whose outputs are purely deterministic and free of side-effects can be provided with their required input values and 'reduced' through the partial evaluation process, as they would be in a normal graph reduction.

The second point to make is that we do not, in any case, have complete freedom to choose the order of evaluation in this environment. Since the graph is dynamic rather than static, the time at which each node is executed is determined by the moment in time at which

its inputs arrive. The output that node generates at that time is by definition the correct result at that time, corresponding to that particular update at that time but not to other updates at other times.

Finally, where limited freedom to choose the order of execution does exist (for example, in independent iteration, or where a node receives several partially evaluated functions as inputs that in turn share some of their inputs), the different branches could potentially be executed concurrently, which could indeed result in unpredictabilities in their timings. Since Dennis [1974] and Arvind et al. [1977] both used the example of a ticket booking system, we can perhaps do the same. Imagine we want our graph to contact an external system to book ten tickets, separately, for named individuals, for a flight on which only five seats are available. In a dataflow system, they can be booked in parallel (as described in Section 4.11.1 — Independent Iteration), which has advantages but also results in unpredictable differences in timing: we have no way of knowing which five passengers will end up with seats. This uncertainty is a natural consequence of parallelising the booking. In general, we would expect these uncertainties to be known and well-understood by the programmer and accepted as a deliberate trade-off against the parallelism. In the example of the booking system, it remains within their power to structure the program differently if they want to guarantee an order in which seats will be booked.

Our design incorporates a few behaviours, to be used individually or in combination, that would usually be seen as functionally impure. One is for nodes to maintain state, which could be used to compute values that depend on previous executions, to append new incoming values to an outgoing list, or in any other way the programmer sees fit. Another is for a node to fetch values from outside the system; for example, a node could query an external sensor for data or an external database for its current state. Finally, a node could trigger external actions; for example, as described above, booking a ticket and returning a confirmation, or simply sending a message to an external system.

These deviations from traditional notions of functional purity have been chosen in order to fit more closely with the practicalities of real problems and the characteristics of the underlying environment. Problems such as synchronisation and latency, which would be nonsensical in a traditional functional language, are acutely important in this dynamic distributed environment, and the discussion on those subjects in Section 4.16 is simply a recognition of the physical characteristics of the system and the underlying universe that such a system would, in the real world, have to face. The deviations from functional purity and the freedoms given to nodes — to store state, fetch data and act externally — are chosen deliberately to enhance the overall power and flexibility of such a system.

4.3. What Are Nodes And Connections?

Since the system is a means of coordination, linking a distributed network of autonomous nodes, there are few constraints on what a node could do, other than that it should obey a defined communication protocol in its interactions with other nodes (obeying the principle of explicit communication listed in Section 3.4.3). This protocol would have the purpose of standardising, across all nodes and implementations, the means of conveying the messages

between nodes discussed in this Chapter, together will all the messages between users or user interfaces and nodes, through which the nodes and the node graph would be controlled. These messages would include:

- data updates between nodes;
- notifications between nodes;
- identification and authentication between nodes;
- the creation and destruction of nodes by users or user interfaces;
- the setting of node parameters by users or user interfaces;
- creation of connections between nodes by users or user interfaces;
- the constraints and limits on external actions by node functions;
- the behaviour of any external node API (through which nodes might expose their functionality to the world outside the system).

Since the purpose of connections is to represent data dependencies, a node must have a data output port. It does not need to, but may, have inputs ports, through which it will receive data from other nodes. Being distributed means that nodes may reside on different machines, in different locations and be owned by different people. Being autonomous builds on the notion of node heterogeneity described by Gelernter and Carriero [1992]. The principle is that the system should be indifferent to the workings of its components, provided they obey a few rules of behaviour and the defined communication protocol. Autonomy takes the principle a step further by saying not only that many different types of nodes may be included but that many aspects of their behaviour may be decided by their owners, or even by nodes themselves based on their owners' stated priorities. In particular, this allows nodes to optimise for cost, speed and resource use. The notion of node autonomy is also used to refer more generally to the flexibility of the system to accommodate a wide range of node behaviours. This widens the applicability of the system and helps comply with the principle of flexibility listed in Section 3.4.2 and the principles of abstraction, high cohesion and low coupling listed in Section 3.4.3.

As described in Section 2.3 (Side-Effects), it has long been a point of discussion in the literature whether dataflow, regarded as a functional language, should be purely functional and free of side-effects. As Hudak [1989] pointed out, there are functional languages that have opted for functional purity as well as those, such as ML and Scheme, that have taken a more pragmatic approach, sacrificing functional purity and a degree of referential transparency in order to allow state and side-effects where needed.

In this document the choice has been made to sacrifice functional purity, and referential transparency in some cases (see Section 4.8.6 — Referential Transparency), in order to capture the benefits of node state and external side-effects, provided the principle of explicit communication between nodes is maintained. On this basis, internal node state and external side-effects by nodes are allowed, provided they do not act together on a

shared program state or use the side-effects as a way of bypassing the principle of explicit communication between nodes (that it should take place only through their connections using the protocol defined for them).

Nodes in our system can, therefore, maintain their own internal state, manipulate that state, use it to calculate their outputs, cause side-effects external to the system or fetch external data. Nodes may also perform external actions, including sending data, storing data, fetching data, and performing external actions of any other kind, provided the integrity of node connections is maintained and these actions are not used to bypass them.

The requirement to avoid using external communication as a way to bypass connections between nodes can never be rigidly enforced. It can only be good-practice guidance that users should avoid such behaviour. Perhaps the most safeguarding against this that could be expected of the system itself is that it capture inadvertent circularity that relies on external systems, by monitoring nodes for unexpectedly high usage or unexpectedly rhythmic patterns of updates.

Because the system does not specify anything about the underlying platform, the system at large is completely indifferent to the computer architecture, and therefore able to embrace any model of computation, past, present or future. Taking this platform-independence to an extreme, there is no reason for the underlying hardware to be silicon-based; it could just as well be biological. We could, for example, have a node whose purpose is to tell us how a particular person is feeling. Let's call the person Bob. When executed, the function is to ask Bob how he feels and report the result. Bob, who is the hardware in this computation, would decide how he feels and reply, and the response would be provided as the output to the node. As a unit, this entity, illustrated in Figure 4.1, would be a valid node.

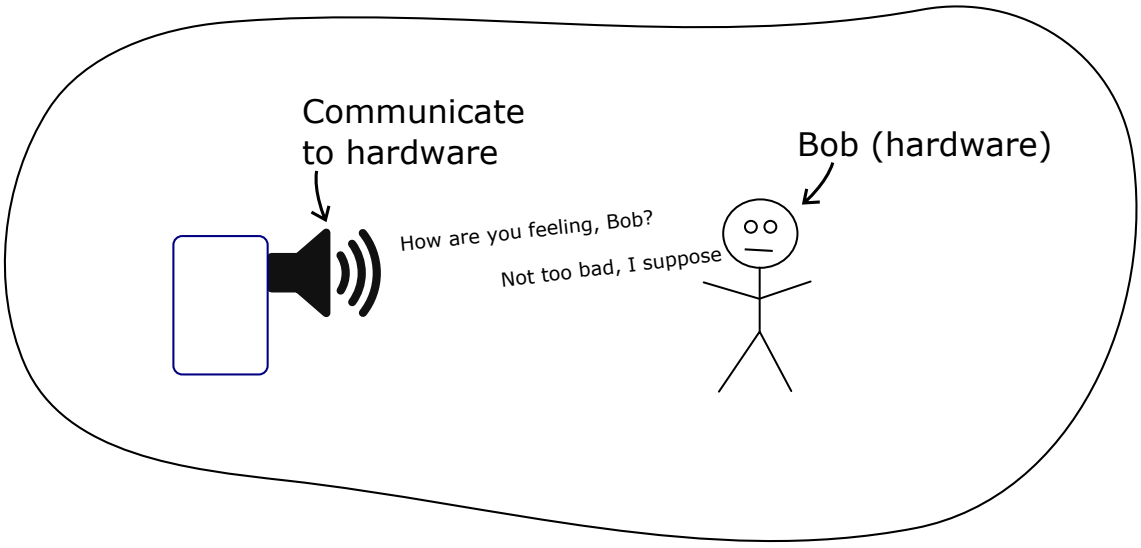

Figure 4.1. The 'How Is Bob Feeling?' node. A node's computational power could come from any source. A human being would make a valid computational component of a node. The programming language used to specify the task must be appropriate to the hardware; in this case, spoken English.

To combine this with external effects, if Bob is a competent chef, we might pass him an

'algorithm', in the form of a recipe, and request that he produce some number of such items. In Figure 4.2, Bob is provided with an apple pie recipe and a quantity, and in response creates some apple pies.

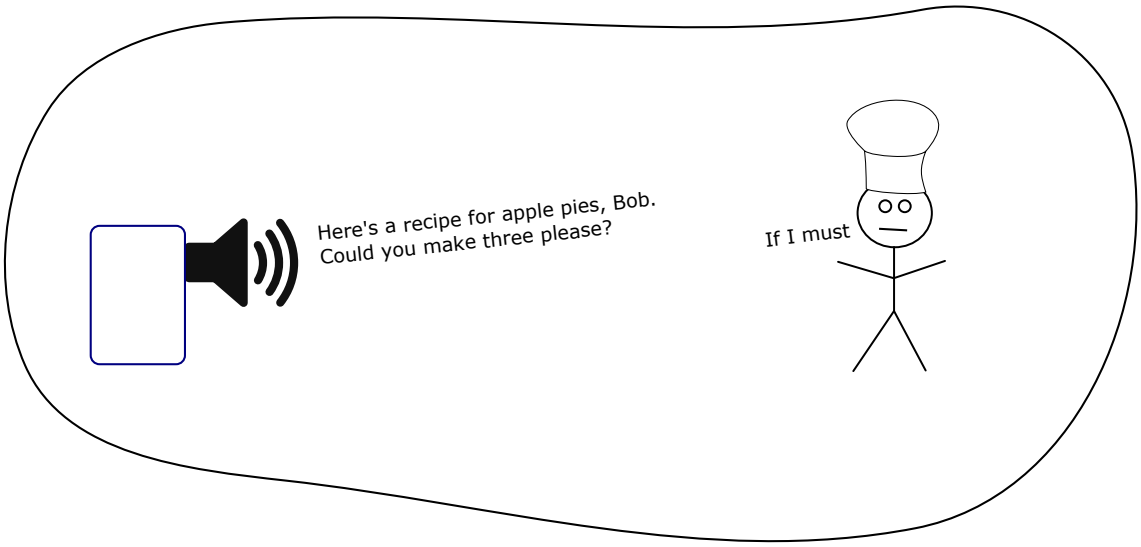

Figure 4.2. Bob's pie-making node. A node could have the job of creating some output or external effect; and the system is indifferent to the means of producing it. A human being would make a valid computational component responsible for producing the output and reporting the result.

The most important property of a node is its content. Its content is a unit of data. The system is indifferent to the type of data, and there is no requirement as to where the data should be stored. It can be stored away from the machine hosting the node, provided that the node is able to access and modify the data when needed. This flexibility helps achieve scalability (listed as a desirable characteristic in Section 3.4.4).

The system needs a number of data types that are universally understood by all nodes; these will be defined as needed, and additional types may be added at any time in future. In our system, these universally understood data types will include the usual primitives (number, string, Boolean), and also functions.

Since it is a coordination system and not a language, and the system itself is indifferent to underlying language and hardware, a function is defined very loosely as comprising four things: an algorithm, a set of zero or more inputs, information about how to execute the algorithm (for example, the language the system is written in or the hardware needed to run it), and the ability for the function to produce a data output[1]. The inputs belonging to a node's function are referred to as its 'root' inputs.

Nodes should have the capability to execute these functions, but not necessarily on the machine where the node resides. The work of executing a function may be delegated to some other machine, provided the node has the ability to trigger execution on request, and to retrieve outputs when finished. Again, this achieves scalability.

In addition to its content, we will give nodes some optional fields of meta-data to add

[1] A node always has an output port, whereas a function may or may not produce an output value. If it does, the function's output value is provided as the node's output; if not, the node's output value will be null.

usability. These include a name by which a user may refer to a node (but which need not be unique), documentation to describe how to use the node and the meanings of its inputs and output, and parameters to control the behaviour of the node.

A minimal set of behaviours of a node, and the interactions between nodes needs to be defined; the rest can be left to the autonomy of nodes to decide. Although nodes are allowed to cause side-effects external to the system, they will only interact with each other through their connections.

The hardware-independence of both data and function execution gives owners the ability to choose the hardware, or supplier from which those services could be provided. They may switch suppliers based on their own balance of priorities, such as cost, security, level of redundancy, speed, or any number of other factors on which such a decision may be made.

A 'connection' in the graph is a directed arc connecting two nodes and provides a symbolic representation of a flow of data, describing a data dependency of the downstream node on the upstream node. Reflecting a consensus in the literature — described in Chapter 2 — that cycles are best avoided, and the principle of acyclicity listed in Section 3.4.3, we will apply the principle to our system that graphs must be acyclic. This provides clarity and simplicity of programs and better obeys the principles of explicit communication (Section 3.4.3) and simplicity (Section 3.4.2). Where iteration is needed, it will be defined explicitly and deliberately (see Section 4.11), not implicitly or unintentionally.

4.4. Separation Between Nodes And Resources

While a node can be thought of as an autonomous entity, with its own behaviour and a point of contact for communication with the rest of the system, that entity does not need to 'own' its own computing resources. If it did, it would risk leaving those resources idle most of the time. The only requirement of a node is that it must have access to resources when needed, not that it must control them completely.

Instead of dedicating resources to individual nodes, a better option is to make a pool of shared resources available to a collection of nodes. This makes more efficient use of memory and processing power, making resources more easily available when needed, and makes for easy scalability (listed as a desirable characteristic in Section 3.4.4) by adding additional, possibly remote, resources to the pool. It also maximises the flexibility of the system by allowing a user to choose whatever hardware or cloud provider they wish to provide their nodes with the resources they need (reducing coupling between the software and its platform, obeying the principle of low coupling listed in Section 3.4.3).

4.5. Visual Representation

Nodes and connections are represented visually as described below. For reference, the visual notation is summarised in Appendix B.

4.5.1. Nodes

A node is represented visually, in this document, as lens-shaped object, either shallow or deep, as shown in Figure 4.3. This is different from the more common depiction of node graphs, in which nodes are depicted as circles or squares, and has been chosen to make more space for inputs at the top of the node, and to draw a clear distinction between inputs at the top, and outputs at the bottom.

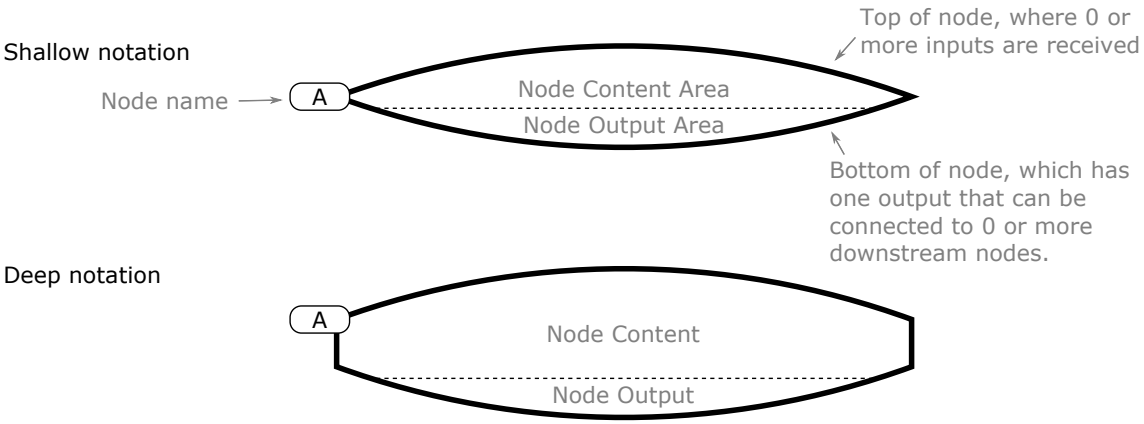

Figure 4.3. Nodes in shallow and deep notation. A node can be stretched vertically as far as necessary to contain the content and node output.

Node names are not required, but are sometimes shown for ease of reference in the discussion. For clarity, a convention is used in this document that node names always begin with an upper case letter. Node content and node output are both optional in the drawings, and either or both are sometimes excluded if not applicable to the discussion at hand. Nodes that are peripheral to the discussion may only be partially shown, or not at all, to exclude unimportant details from the drawing.

4.5.2. Node Connections

Data within a graph is depicted as flowing from top to bottom. At the top of a node, zero or more inputs can be made available, each of which can receive, at most, one value or connection. An input is depicted as a 'V' shape in the top part of a node, and may be labelled with a name if necessary. Where input names are included, a convention is used in this document that they always start with a lower case letter.

At the bottom of a node, its output can be connected to zero or more downstream nodes. Connections between nodes are shown as lines flowing from the output (the bottom part) of one node to an input of another. Neither inputs nor outputs need necessarily be connected. Inputs can also be provided with a value directly, which is depicted by showing the value above the input in question. Figure 4.4 shows a node with three inputs (one not connected, one connected to an upstream node and one provided with a value directly) and an output with multiple connections.

If a node's content is a function, it will have 'root' inputs corresponding to the arguments of the function. A node may also have further inputs that are derived from its root inputs. These derived inputs will either be inherited from an upstream node (see Section 4.8) or derived from a root input's input parameters (see sections 4.9 and 4.11.2).

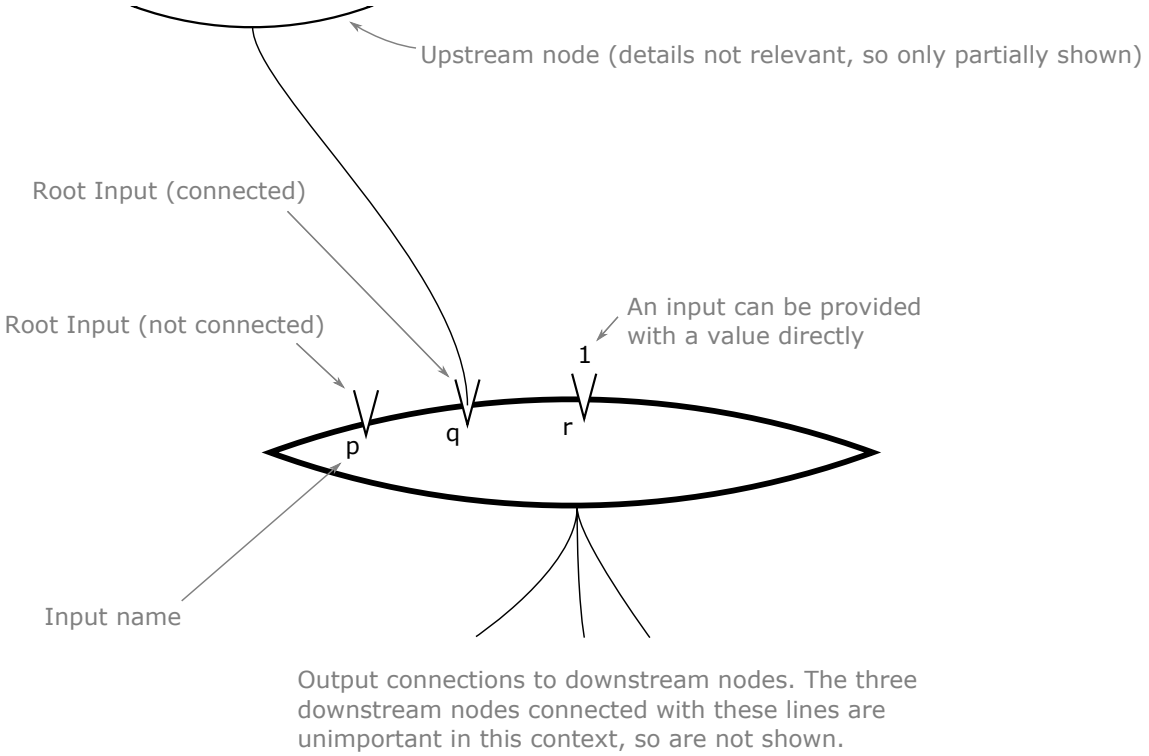

Figure 4.4. Inputs, outputs and connections. Inputs are shown as a 'V' shape in the top of the node, together with an associated name label where appropriate. Inputs can be provided with values either by connecting them to an upstream node (as in the case of input 'q') or by providing a value directly (as with input 'r'). Nodes have a single output value which can be connected to multiple downstream nodes.

A node with just one input that provides its input, unaltered, as an output, is referred to as an 'ID' node. Its input may have any name. ID nodes are depicted with their content shown as 'ID', as shown in Figure 4.5.

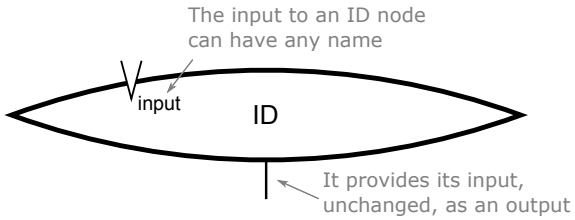

Figure 4.5. ID Node. An ID node is a node with one input, which provides its input, unaltered, as an output. Its content is shown as 'ID'. Its input may have any name.

4.6. The Service-Provider Model

Node interactions are based on a 'service-provider' model. This model of interaction was as advocated by Evans [2004] and described by Meier et al. [2009] as delivering the principles of abstraction and low coupling (listed in Section 3.4.3) and distributability (listed as a desirable characteristic in Section 3.4.4). Meier et al. [2009] described a service-oriented architecture as being one in which "*applications ... expose and consume functionality as a service using contracts and messages*". In this context it means that each node is a service provider to its downstream connected nodes (that is, it offers services and agrees contracts to provide those services to its downstream nodes); and is a service user of its upstream

connected nodes.

A range of services may be provided to downstream connected nodes including, for example, subscriptions (meaning the downstream node receives new outputs when available, possibly subject to constraints specified in the subscription contract) or that the downstream node may at any time request a value. It may also agree a contract with a downstream node to keep itself current, perhaps with associated financial transactions or agreements, as discussed in Section 4.15.

One of the consequences of the use of the service-provider model is that nodes are largely indifferent to and ignorant of their downstream nodes (complying with the principles of abstraction and low coupling described in Section 3.4.3). Accordingly, connections are properties of inputs, not outputs.

If a node has no knowledge of its downstream connected nodes, the system needs a way for data to flow down the graph. This can be done by adding the concept of subscriptions. A downstream node that requires data from an upstream node can 'subscribe' to it, notifying the upstream node that it wants to be informed of updates to its value. Each node will maintain a list of its subscribers, and will notify every subscriber whenever new data is available. The benefit of using this model is that it allows nodes to 'switch off'. Subscriptions signify that a client is interested in a node's updates. If a node has no subscribers it can 'switch off' by unsubscribing from its upstream nodes. It will then no longer receive updates or waste resources computing values that nobody needs, and will no longer require upstream nodes to compute values on its behalf either.

Using subscriptions, whole sections of a graph may end up switching off and becoming dormant, yet remain available when needed. Nodes will then refresh themselves only when data is requested by their downstream 'clients', rather than computing an update every time new data becomes available. Although it may be premature to discuss optimisation, if we have an always-on fully-available interconnected system, it is important to be able to avoid unnecessary computation.

This delivers the best of both the data-driven and demand-driven execution models described by Davis and Keller [1982] (see Section 2.4). As pointed out by Treleaven et al. [1982], the two models are complementary rather than competitive and, as they also pointed out, a system that utilises all three execution models (including also the sequential control-flow model used by most conventional programming languages) may be better able to achieve general purpose goals. Our system achieves this by utilising a hybrid demand-driven and data-driven dataflow mechanism to coordinate scheduling between control-flow nodes.

Using the service-provider model, each node is dependent on the service provided by its upstream nodes for its ability to compute its own output value. Given that nodes may be owned by different owners, we may also need node connections to be supported by contracts between node owners determining the quality of the service provided. The nature of the contract does not need to be dictated by the system, but for consistency it may be desirable for it to provide set contracts, so that such agreements can be automated. The service-

provider model supports such contracts by enabling cost accounting, in which usage could be measured and associated costs or charges assigned to nodes and their human owners.

It was mentioned in Section 4.3 (What Are Nodes And Connections?) that nodes are autonomous, leaving the owner free to optimise the behaviour of a node according to their resource and cost priorities. This applies equally to the service provision: service users could be offered the ability to pay for more expensive prioritisation preferences. For example, a user might want to pay more to be notified of every data update than just to receive the current output value; or to pay more for the computation to be executed on faster hardware than on slower hardware. Any such service could be charged by a node to its clients. The service-provider model need not just apply internally: nodes could equally well provide their service externally, delivering data and notifications in response to requests or subscriptions, as a chargeable service if required.

The model proposed here differs from many traditional approaches. In a traditional model of dataflow a node graph represents a static program. It purpose is to run, produce an output and then stop. Referential transparency allows the freedom to choose an execution strategy, with the knowledge that any execution strategy or means of graph reduction will lead to the same output. In our model, the graph represents an interactive program that may be distributed over many users and locations and constantly maintains its outputs, computing them dynamically as changes take place within the graph (to its inputs, node contents and graph structures). When a change takes place to a node's content or inputs, its output value and those of its downstream nodes cease to be current. The change and resulting outputs must therefore be propagated through the graph as quickly as possible in order to minimise latency between the originating update and the corresponding changes to outputs in the downstream graph. By design, computation of each node is triggered by arrival of data at its inputs; and functions are, by definition, substitutable by the values they create when computed at that time. Provided the user has not violated the principle of explicit communication by encoding hidden communication between nodes through their external connections, differences in outputs for different execution strategies will relate only to the differences in the time at which continuously varying external variables are sampled. This variability is intrinsic to the sampling of external variables and would apply to any system that included that capability.

While this deviates from traditional models of functional programming and dataflow, it is also highly pragmatic. It provides necessary access to the external world (to be declared explicitly when needed) and provides greater flexibility in the development process, closer alignment with the principles of Software Engineering described in Chapter 3 (Software Engineering) and can accommodate the frequent updates to data inputs, functionality and objectives that need to be accommodated in real-life projects.

4.7. Triggering Execution

One of the defining features of dataflow (as expressed by Yazdanpanah et al. [2014]) is that execution is triggered by the availability of data, rather than the availability of instructions. When new data arrives at a node, it recomputes. For a node that is deterministic and

side-effect free, this works perfectly. Its sole purpose is to produce an output value, and we can guarantee the same input will always produce the same output. The node can take a 'lazy' approach to evaluation: each new input value can be compared with the previous input value and computation avoided if the two consecutive input values are the same. Although this would be a deviation from the traditional dataflow model, in this dynamic system (as discussed in Section 4.2), it can be a useful optimisation for side-effect free nodes.

However, as discussed in Section 4.3, nodes are not always deterministic and side-effect free, and this lazy approach works less well for nodes that are not. For nodes that are non-deterministic, depend on external data, modify internal state or cause external side-effects, we need to be able to trigger execution of a node even if its inputs are unchanged.

In Section 4.3 we asked Bob's pie-making node to make three apple pies. What if we need another three? As the inputs are the same, a lazy approach to evaluation would prevent the node from calculating. And yet, we still need three more pies.

An example sometimes used when discussing side-effects is that of a ticket booking system (see Section 2.3 — Side-Effects). In a ticket-booking system, the number of available tickets is an external state that must be queried by the node, meaning that the result of booking one ticket may not be the same as the result of booking the next. As with the pies, the function must be executed even if the inputs are the same.

One way of achieving the required triggering for nodes that need it, whilst retaining the more efficient lazy execution for the rest, would be for the programmer to provide an input to the node in question for some arbitrary 'trigger data'. This input would have no purpose other than to cause the node to execute. The value provided to it could be an integer that incremented each time, or a time-stamp, or anything else that suited the situation.

Would a continuously incrementing integer constitute a memory leak? The memory footprint of an integer increases only logarithmically with the magnitude of the number, so would not have an undue impact but, if concerned, the number could be made to loop, provided the modulus of the loop were high enough to preclude the possibility of the node mistaking one update for another. With suitable triggering being achievable within the existing functionality, there is no need for additional or explicit functionality to achieve this aim.

4.8. Partial Evaluation

Partial evaluation of a form was mentioned in conjunction with some of the earlier work on dataflow programming (Sutherland [1966]; Rumbaugh [1977]; Keller [1980]). In those earlier contexts, it was often discussed in terms of 'envelopes', and 'composition' of graphs. Keller [1980] compared it to two related concepts: closures, and 'Currying'. Davis and Keller [1982] pointed out that this concept (referred to by them as "*data objects that can represent functions*") can "*enhance the power of data flow programming considerably*",

explaining that "*many versions of the encapsulated graph can be generated, each customised by different import values*".

Most of the terms that were suggested (with the exception of Currying) conveyed details about the implementation rather than the effect. Although Currying is in some ways similar, it is not similar enough, and does not leave us with as much generality as we need. This document instead uses the term 'Partial Evaluation', which expresses the effect of the concept rather than the implementation. This term is more relevant to the user and leaves our implementation options open. The implementation described in Chapter 5 is only one of many that could have been used; any other that achieves the same goal could equally well fall within the term.

4.8.1. Full vs Partial Evaluation

To define our terms, 'full' evaluation is where you start with a function, provide values for all of its inputs, execute the function and then provide its return value (if applicable) as an output. The purpose of a partial evaluation is for a node to provide a function as an output in every other case. In a partial evaluation, a node containing a function will produce, as an output, another function containing only the inputs whose values were not provided to the node. This concept is illustrated in Figure 4.6.

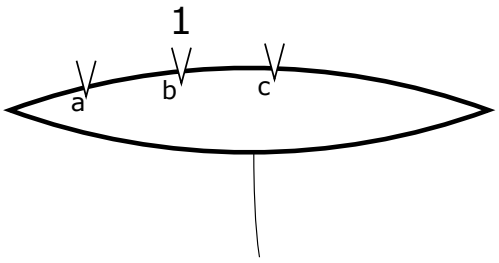

Figure 4.6. Partial evaluation. When a node partially evaluates, it generates an output that subsumes the provided values, and has an input for each of the node's inputs whose values were not provided. In this example, input 'b' has been provided with a value and the output is a function of 'a' and 'c'.

In this example, the node has three inputs, labelled 'a', 'b' and 'c'. Input 'b' is provided with the value '1' and the node partially evaluates. It subsumes the value of the input 'b' and provides, as an output, a new function with inputs 'a' and 'c'. Partial evaluation helps comply with the principle of reusability (sometimes called the 'Don't Repeat Yourself' principle) listed in Section 3.4.2.

4.8.2. Inheriting Inputs

Having generated a function as an output of a node, we want to be able to use that function. When a partially evaluated function is received by a downstream node, its inputs still need values before it can be fully evaluated. To represent this, the inputs of the received function are 'inherited' by the receiving node, and are referred to in the downstream node as 'derived' inputs. Derived inputs are depicted in node diagrams as shaded and slightly smaller than root inputs, as shown in Figure 4.7.

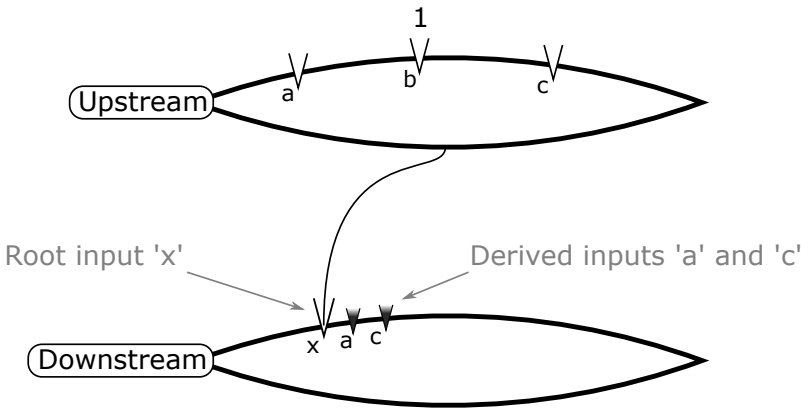

Figure 4.7. Derived inputs. When an input is connected to a node whose output is a partially evaluated function, the node 'inherits' the inputs of the function it receives and displays them as 'derived' inputs. Derived inputs are shaded and slightly smaller than root inputs.

If the example shown, the 'Downstream' node inherits inputs 'a' and 'c' from the 'Upstream' node. If these two derived inputs are provided with values then the 'Downstream' node can fully evaluate and produce an output. Alternatively, it may in turn be left to partially evaluate and output another function with inputs 'a' and 'c'. A derived input behaves the same way as a root input: it can be provided with a value or can be connected to an upstream node, including one that delivers a partially evaluated function, causing the receiving node to inherit further inputs.

4.8.3. Inheriting Content

The architectural principle of Reusability, listed in Section 3.4.2, is helpful in testing and development (see Section 4.17). The features of partial evaluation and input inheritance make nodes reusable, but only when mediated by ID nodes. As shown in Figure 4.8, functionality defined by one node can be reused by multiple others, but this use of an ID node requires the node to have the ID function as its content and to display the ID function's argument as its root input.

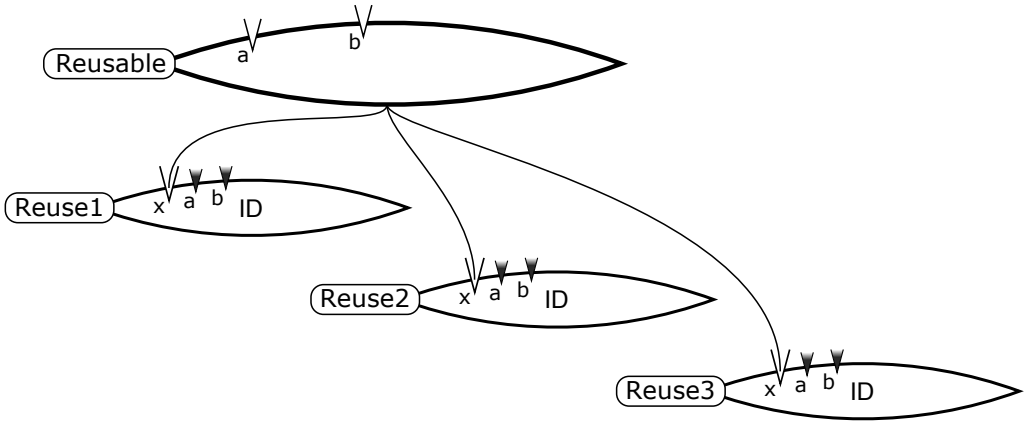

Figure 4.8. Reusing nodes. The functionality defined by the 'Reusable' node is reused by three other nodes ('Reuse1', 'Reuse2' and 'Reuse3'), each of which is an ID node with one input, 'x', inheriting the inputs 'a' and 'b' from the 'Reusable' node.

To simplify this process and eliminate unnecessary clutter, a node's *content* can instead be connected to an upstream node. When this is done, the node's content is set to the

value received from the upstream node and, if that value is a partially evaluated function, it will inherit the inputs of that function.

A connection to a node's content is depicted as a connection directly to the top side of the node. Since this sets a node's content, a node can only have one connection of this type, and cannot also have its own content. When the content is connected to an upstream node, the inputs inherited from the upstream node are all shown with equal status as root inputs.

In the example shown in Figure 4.9, the 'Reusable' node's functionality is reused by three other nodes, by connecting their content to the upstream node. They then inherit the inputs of the partially evaluated function they receive and replicate the functionality of the upstream node.

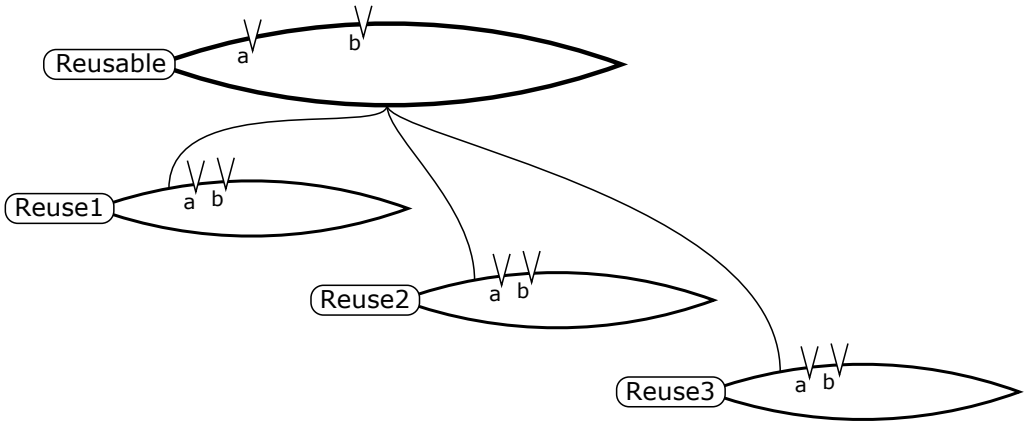

Figure 4.9. Connecting a node's content to an upstream node. In this example, the 'Reusable' node's output is a partially evaluated function, which is reused by three other nodes, 'Reuse1', 'Reuse2' and 'Reuse3', by connecting the content of each to the 'Reusable' node. They all, in turn, inherit the inputs of the partially evaluated function they receive.

4.8.4. Input Name Clashes

In the case of a name clash, where two inherited inputs have the same name, or where an inherited input has the same name as a root input, the inherited inputs are uniquely identified using their node of origin and name, in curly brackets, as shown in Figure 4.10. This is referred to as a 'fully resolved' input name.

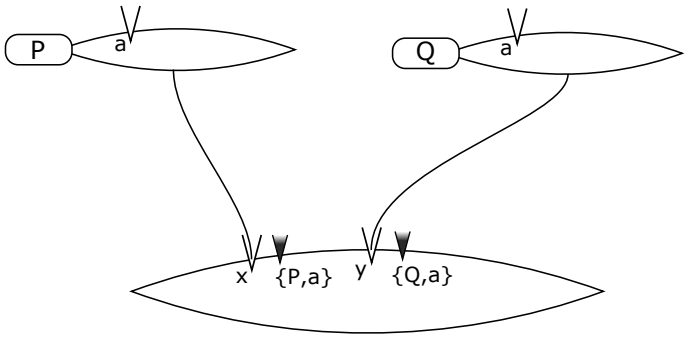

Figure 4.10. Fully resolved input names. Inputs can be uniquely identified by their name and node of origin, depicted in curly brackets as {nodeName,inputName}. This notation is used if the input cannot be uniquely identified from the input name alone.

4.8.5. Input Unification

Each input can be only inherited once. If an input is inherited via two separate routes, as shown in Figure 4.11, it is re-combined. In this instance, a different partially evaluated function containing the input has been received via each of the two routes. If a value is now provided for that input, it must be applied to both partially evaluated functions. In the example in Figure 4.11, the input 'a' originates at node 'P'. It arrives at node 'Q' via input 'x' and input 'y', but appears only once in Node 'Q'.

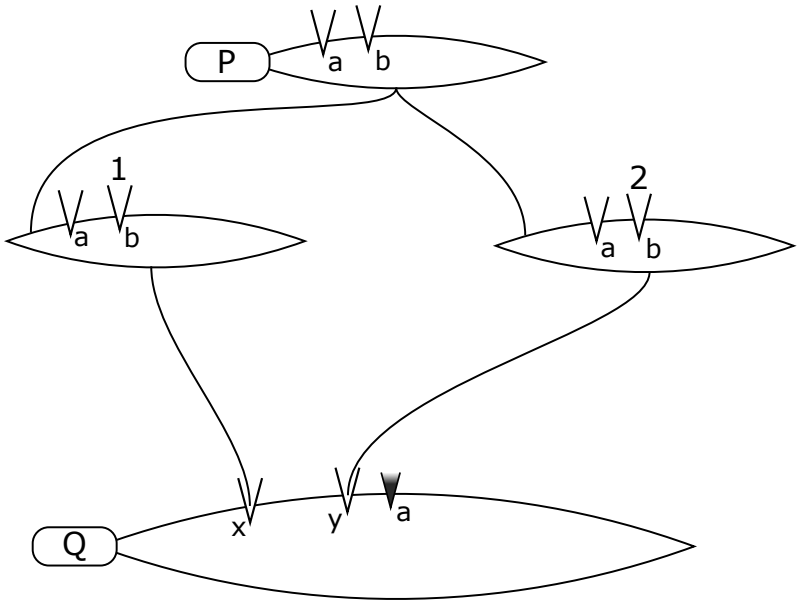

Figure 4.11. Inputs that arrive by multiple routes. Each input, identified by its name and node of origin, can only be inherited once by each node. If it is inherited via two routes, as in this case, it is displayed only once.

4.8.6. Referential Transparency

Referential transparency is usually defined in terms of replaceability of sub-expressions with their resulting values, and is sometimes characterised as a key difference between imperative and functional languages (Hudak [1989]; Sabry [1998]). The concept of partial evaluation depends on the ability to replace a function with its resulting value; allowing node state, side effects and external data sources, as discussed in Section 4.3 (What Are Nodes And Connections?) would seem to undermine that ability. Indeed, by some the existence of side-effects is used as a shorthand for referentially opaque[2].

To resolve this apparent contradiction it is worth pointing out that although the system described here allows nodes to have their own state, that state is not shared between nodes. The intention is, additionally, that state, side-effects and external data sources should be explicit, and therefore only ever included as deliberate features of a program rather than as a mechanism by which nodes could interact with each other or as a means of bypassing the explicit connections between nodes that are intended to be the only means of communication between them. Where purely deterministic behaviour is required, it can be achieved by building the program without using those features.

[2]Søndergaard and Sestoft [1990] summarised and provided a good discussion on the varying definitions of the term *referentially transparent* that have been used.

Where those features *are* needed, using them sacrifices the referential transparency and determinism of the program, but doing so will be amongst the goals of the program. Using those features would not mean we sacrifice all control over the values or that they would become random or meaningless. Instead, inputs to nodes are used to control when nodes execute and are used, in some cases, solely for the purpose of triggering that execution, as described in Section 4.7 (Triggering Execution).

Whereas each execution of a non-deterministic node may result in a different value, in any particular execution of such a node, the result of executing it will be by definition correct at that time and replaceable by the value obtained at that time. The node outputs the value that was correct at the time it was computed, together with a stamp identifying the time and node at which the triggering change originated, and transmits it through the graph with only that promise; that it was correct at the time it was produced. It is distinguished from a future (different) value by being produced at a different time in response to arrival of a different input value.

4.9. Expected Inputs

If we have a function but not its inputs, partial evaluation allows us to provide its inputs later, downstream. Sometimes we need to be able to reverse this, by setting inputs for an as-yet unknown function and providing the function later, downstream.

Returning to the example of Bob's Pies used in Section 4.3, imagine that we know we want three pies but have yet to decide on the recipe (or algorithm) for making the pies. We know we need a recipe, and that it should take 'quantity' as an input. If we had the recipe in advance, we would set up our node as shown in Figure 4.12. In this scenario, we have set up our 'Main' node, which has an input named 'recipe'. We then connect 'recipe' to the upstream node, at which point we inherit the input 'quantity' and can provide it with the value '3'. However, if we do not yet have the upstream 'Recipe' node, we cannot inherit the 'quantity' input, and there is no way for us to provide the value (Figure 4.13).

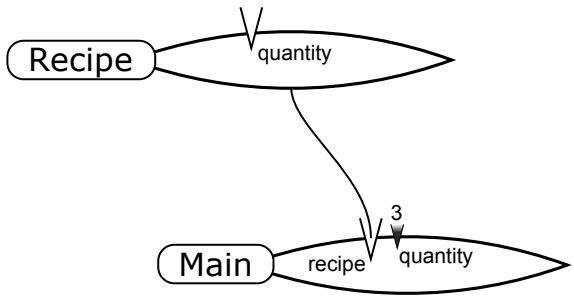

Figure 4.12. Inheriting the 'quantity' input. Once you have the recipe, you can set the quantity required.

To provide the capability to do this, we need a new feature, which we will call 'expected inputs'. The purpose of the 'expected inputs' feature is to be able to tell a node that we expect one of its inputs to receive a partially evaluated function and to specify the names of the inputs we expect that partially evaluated function to have. Upon doing so, the node should provide a way for us to provide values for those inputs, even though the partially

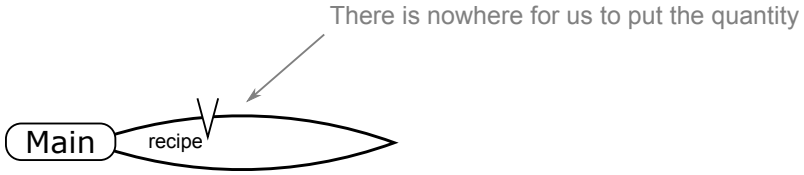

Figure 4.13. Without connecting the recipe, it would be impossible to set the quantity required. Sometimes the quantity might be known but not the recipe. This requires a new feature, named 'expected inputs'.

evaluated function in question is not yet available. To do this, we have to provide a list of names of the inputs we expect, as a parameter of an input. We will call this the 'expected inputs' parameter.

An input parameter is denoted visually, in this document, with a box connected to the input, as shown in Figure 4.14. The 'expected inputs' parameter is a list of names of the inputs we expect the partially evaluated function eventually received by that input to have. In the example shown, there is just one expected input, 'quantity'.

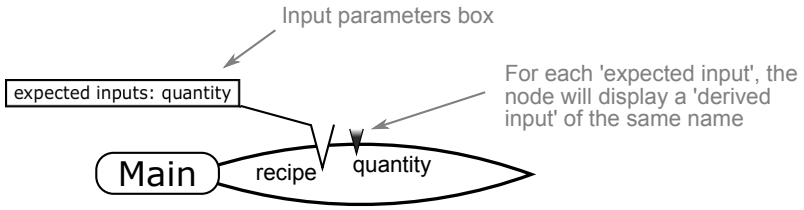

Figure 4.14. The input parameters label. Input parameters appear as a label attached to the input in question. The 'expected inputs' parameter is a list of input names. The node generates a derived input for each item in that list. In this case, there is only one expected input, named 'quantity', and the node generates a derived input for it.

In response to this property having been set, to allow the user to set a value for it, the node should generate a 'derived' input for each expected input (also shown in Figure 4.14). If this node is partially evaluated and connected to downstream nodes, this derived input gets inherited by downstream nodes in the same way as any other input. Likewise, the input parameter gets transmitted through the graph (Figure 4.15).

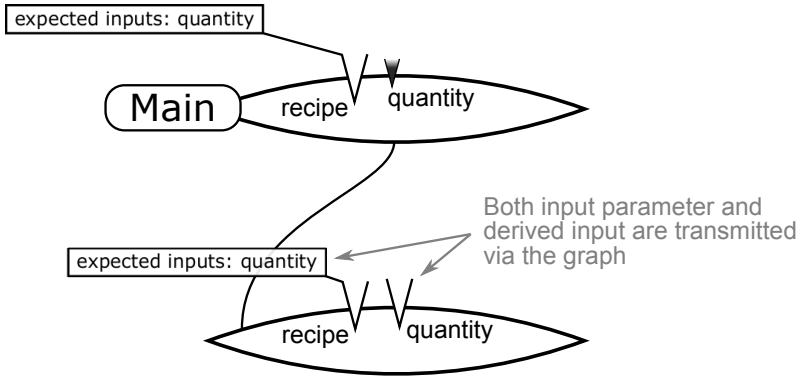

Figure 4.15. Transmission of parameters and derived inputs. Input parameters are inherited by downstream nodes together with the input they are associated with, and derived inputs are inherited by downstream nodes in the same way as other inputs. In this example, the derived input appears in the downstream node as a root input because its content is connected to the upstream node.

A derived input behaves the same as other inputs. If a value is provided for it then, when partially evaluated, the value is stored internally in the partially evaluated function and the derived input will not appear as an input to downstream nodes (Figure 4.16).

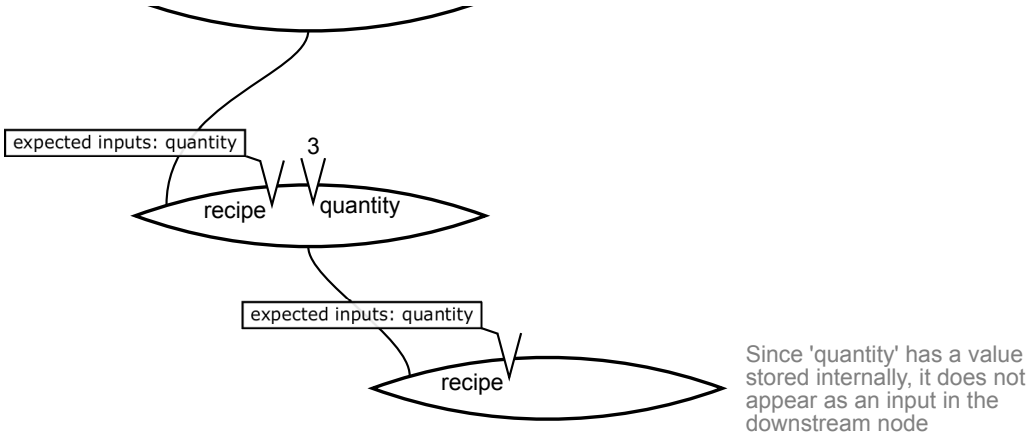

Figure 4.16. Internal storage of the connected value. If the derived input is provided with a value, it is stored internally in the partially evaluated function until the expected function is received. In this example, the 'quantity' input is provided with the value '3', which is then transmitted in the partially evaluated function and does not not appear as an input downstream.

When the expected function arrives, its inputs may not correspond with the expected inputs in the input parameter. To allow for this, we need the ability to associate the expected inputs with the inputs the function actually has. This is referred to as 'nominating' which of its inputs should correspond to each expected input.

Again, this requires us to introduce parameters, this time for connections. The 'nominated inputs' parameter of a connection will be a list of associations between expected inputs and actual inputs of the incoming function. The 'nominated inputs' connection parameters are depicted visually in this document as a box connected to the connection. One input of the incoming function must be nominated for each expected input. This is shown, applied to our example, in Figure 4.17.

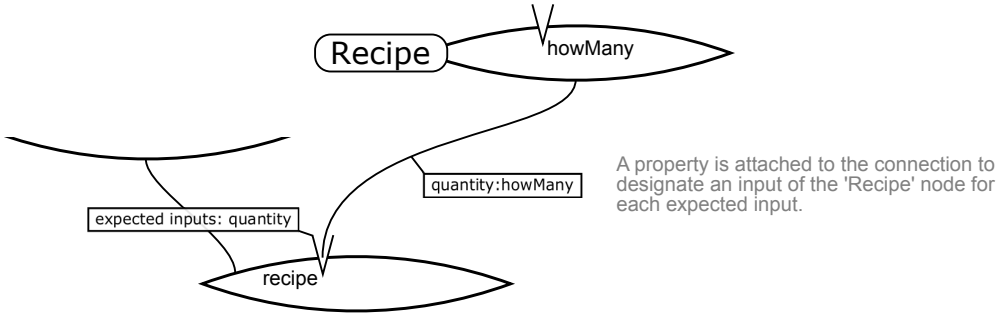

Figure 4.17. Nominated inputs. When an input with an 'expected inputs' parameter is connected to a node, an input of the incoming function must be nominated as the one to be associated with each expected input. This is depicted using a label attached to the connection. In this example, the expected input named 'quantity' is associated with the 'howMany' input of the incoming partially evaluated function.

In this document, nominated inputs are depicted if they would not otherwise be obvious to the reader, and only with as much information as is needed to uniquely identify the input in question. In cases where multiple inputs have the same name, they can be distinguished by their originating node. There are three possible ways to show a nominated input:

1. If there is only one expected input, and the incoming function has only one input, there is no need to nominate it (Figure 4.18).

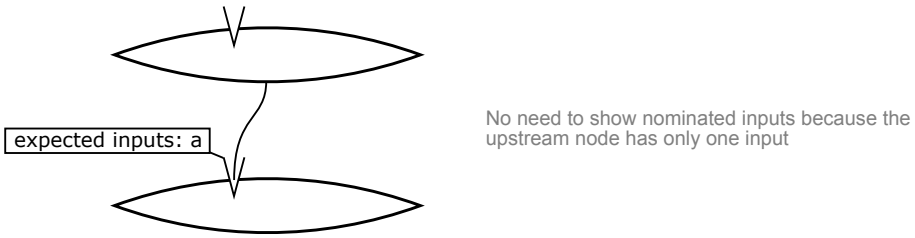

Figure 4.18. Implicit nomination. In cases where the incoming function has only one input, the nomination is not required, and is not always shown in drawings.

2. If the input name being nominated is unique, it can be nominated using only its name (Figure 4.19).

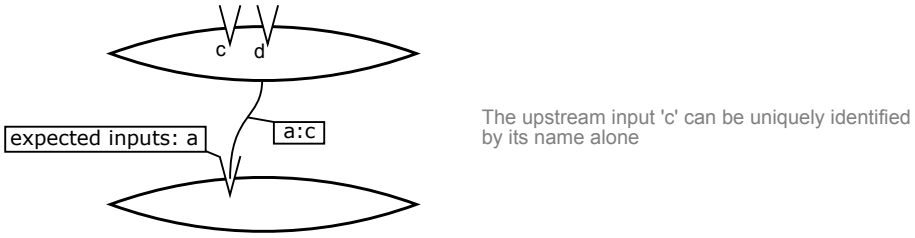

Figure 4.19. Name-only nomination. In cases where the nominated input name is unique, it can be nominated using its input name alone.

3. If the input name being nominated is not unique, its fully resolved name must be used (specifying its node of origin as well as the input name). This is done by using curly braces containing the node at which the input originates, followed by the input name (Figure 4.20).

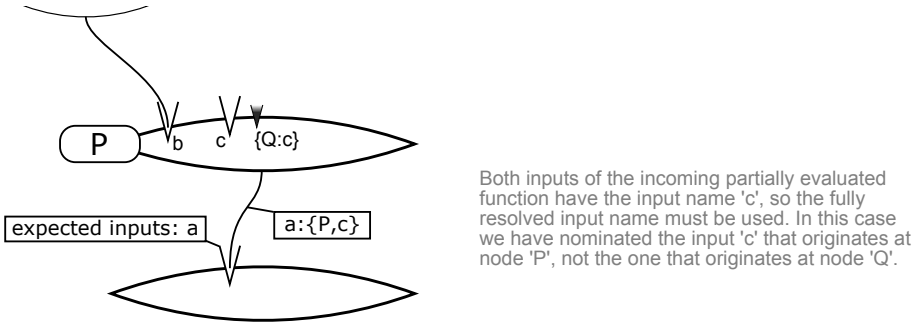

Figure 4.20. Originating Node / Input Name nomination. In cases where the input name being nominated is not unique, it can be uniquely identified using its originating node name and input name, in curly braces.

4.10. Dimensions

We are going to adopt three ‘shapes’ in which we will consider data to occur: units, lists, and tables. This is an extension of the three fundamental data types originally listed by Keller [1980], which included atoms, tuples and graphs. Atoms were similar to our ‘units’, tuples were similar to our ‘lists’ and graphs were somewhat similar to our partially evaluated functions, described in Section 4.8.

A Unit is a container for any item of data as we would normally know it. A number, a block of text, a file, or an object comprising properties. A unit is zero-dimensional. It may contain any other item of data, including a list or a table.

A List is an ordered array of units. As with any unit, each could contain a raw data of any kind, a list or a table. A list is one-dimensional.

A Table is a hyperrectangle — a multi-dimensional rectangle of units of data. It can have zero or more dimensions.

A table's shape is defined by a list of its dimensions, each member of which is itself a list defining that dimension's column headings. In this document, the word 'column' is used to refer generically to columns and rows since, in a multi-dimensional table there is no useful distinction between the two. A depiction of a table structure is shown in Figure 4.21.

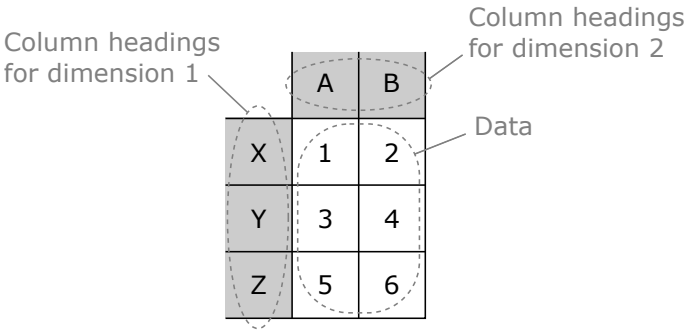

Figure 4.21. Table structure. Tables are depicted with shaded column headings on the left and top of the table, providing a cross-reference for the table data. Tables of more than two dimensions are not depicted visually in this document.

In this example, there are two dimensions, so the dimension list would consist of two members, each of which would be a list: the first dimension would be the list '[X,Y,Z]' and the second dimension would be the list '[A,B]'. There is no set order in which the table's dimensions must be displayed when depicted visually.

Multi-dimensional tables of more than two dimensions are difficult to depict visually due to the limitations of paper, space, and common human experience. Tables of more than three dimensions are even more difficult to depict. Data visualisation tools usually circumvent the problem by choosing two dimensions and summarising over the rest, or showing subcategories within column headings or combining both approaches. In this document, no data of more than two dimensions is depicted, and the problem is therefore avoided.

4.11. Iteration

As described in Section 2.5, three main approaches to iteration have been considered in the past: special purpose nodes, enclosures and switch nodes. Special purpose nodes are nodes which perform an iteration internally. They provide fields for the function, the initialisation statement and the termination condition, analogous to a 'for' statement in a C-family programming language. Cantata (Mosconi and Porta [2000]) took this approach.

Enclosures are where a section of the graph is enclosed by a special node which performs the looping operation over the graph it encloses. LabView (National Instruments Corporation [1998]) took this approach.

Switch nodes are the most common approach, but depend on a model of dataflow that allows cyclicity (violating the principle of acyclicity listed in Section 3.4.3). With switch nodes, instead of using the service-provider model as we have done here (see Section 4.6), the connections must be seen as pipes, along which the data 'flows', with the node directing flow down one pipe or another, providing a cyclical graph with an escape condition. The U-Interpreter (Arvind and Gostelow [1982]) used this approach.

Special purpose nodes provide the better solution because they avoid cyclical graphs, but without the benefit of partial evaluation they lose flexibility and reusability. Here we propose a slightly different approach — a variation of the special-purpose node concept, but which maintains acyclicity, flexibility and reusability by using partial evaluation.

Mosconi and Porta [2000] proposed breaking down the problem into two types of iteration, which they called 'horizontally parallel' and 'temporally dependent', or 'sequential'. These classifications are useful, and are adopted here using the simpler terms 'independent' (horizontally parallel) and 'dependent' (sequential). Every iteration can be classed as either one or the other of these two types.

- **Independent** — every iteration is independent of every other iteration;
- **Dependent** — there is at least one iteration that depends on the result of at least one other iteration.

A series of iterations in which every iteration is independent of every other iteration would not be called iterative in a mathematical sense. However, in a sequential programming environment, because there is no additional cost involved in using iterative flow control constructs to perform independent iterations, they are frequently used for independent iteration. However, because independent iterations can be executed in parallel and dependent iterations cannot, it is worth distinguishing between the two in a parallel programming environment. Independent iteration is the easier to deal with, and is often addressed using some form of 'for-all' loop (as described by McGraw [1982]). We will deal with independent iteration first.

4.11.1. Independent Iteration

For independent iterations to be parallelised, the inputs for all iterations must be available at the start. This may involve a set of different data inputs for the different iterations, or a set of different functions for the different iterations (or both).

Dimensions provide a natural way to represent the functionality of a 'for-all' loop. If a function is written to receive zero-dimensional inputs, what happens when data arrives that has more dimensions? In this situation, where the incoming data has more dimensions than the input can process, we can refer to the input as being 'dimensionally overloaded'.

Imagine the data delivered to an input is a list of zero-dimensional members. The function can compute the result for any particular member of the list but not the list as a whole. A simple way to deal with this situation would be to compute the result once for each member of the list, creating a corresponding list of output values as a result.

Figure 4.22 illustrates this with a function that adds the number '1' to its input. In this example, it receives a list of numbers as an input, meaning the input is dimensionally overloaded, so it executes its function once for each member of the list, generating a list as an output.

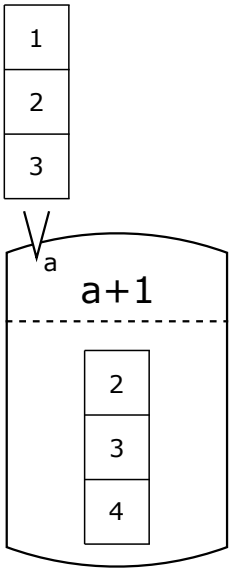

Figure 4.22. List in — list out. In this example, input 'a' expects zero-dimensional data but receives one-dimensional data, so is overloaded by one dimension. It responds by computing its output once for each dimension in the list — creating a list as an output.

Similarly, if a node receives a two dimensional table at one of its inputs, it can perform the node function once for each item of data in the table, and output a corresponding table of results (Figure 4.23).

The arithmetic is very simple: for each input that is dimensionally overloaded, the number of dimensions by which it is overloaded is added to the number of dimensions in the output. If lists added in this way lead to tables as outputs, each list becomes a column header of the output table. Figure 4.24 shows a node that has two lists (one dimension each) as inputs, resulting in a table of two dimensions as an output, using the input lists as column headings. The number of dimensions in the output table is a simple sum of the numbers of dimensions by which its inputs are overloaded.

It might be that what varies between iterations is not a number, but the function being applied to the number. In Figure 4.25, we have three nodes, 'Square', 'Root' and 'Log', which calculate the square, root and log, respectively, of their inputs. All three inputs are connected to an upstream ID node with the input 'a', meaning they all inherit the input 'a'. Their outputs are, in turn, connected to the 'MakeList' node, which takes three inputs and generates a list of the three received values. The 'MakeList' node also inherits the input 'a'.

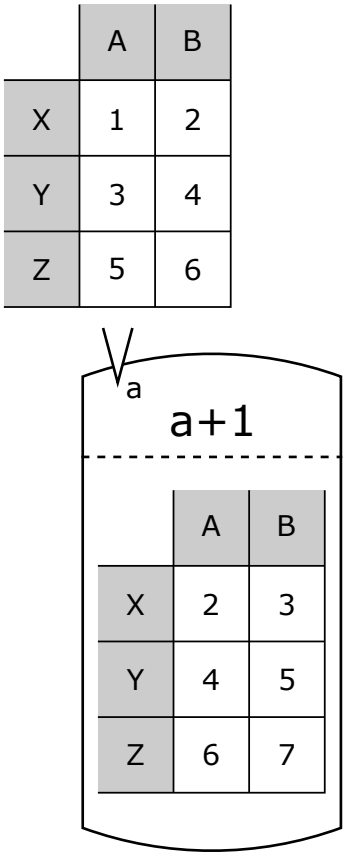

Figure 4.23. Table In — Table Out. In this example, input 'a' expects zero-dimensional data but receives two-dimensional data, so is overloaded by two dimensions, adding two extra dimensions to the output and resulting in another table.

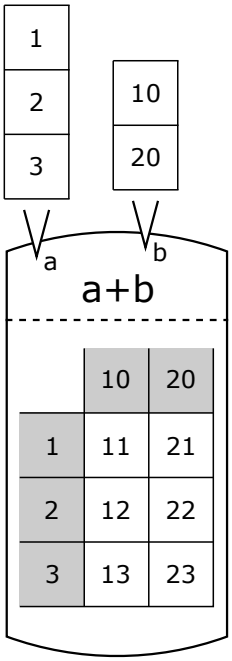

Figure 4.24. Multiple Lists In — Table Out. Both inputs 'a' and 'b' expect zero-dimensional data but receive one-dimensional data, meaning they add one dimension each to the output, resulting in a two-dimensional table with the input lists as column headers.

The purpose of this structure is that it results in an output that consists of a list of three different functions of the same input. When this is connected to input of the 'AddOne'

node, the 'AddOne' node inherits input 'a'. In this example, the inherited input 'a' is provided with the value '2' and a list of three values is generated as the output of the 'AddOne' node.

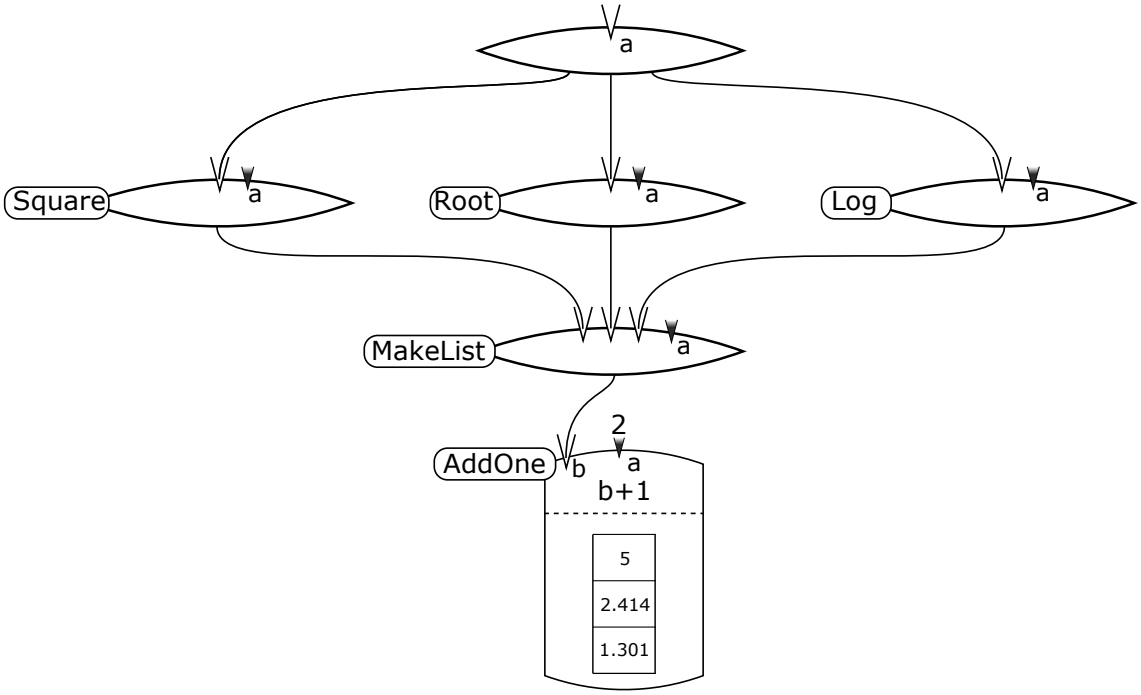

Figure 4.25. Applying a list of functions to an input. In this example, the graph is used to create a list of three functions of the same input, 'a'. When connected to the 'AddOne' node's input 'b', input 'b' is dimensionally overloaded. If we then provide a value for input 'a', the 'AddOne' node generates a list of three values as its output.

Sometimes, we need to allow node functions to process dimensional data directly. This applies for some very simple uses, such as where we want to obtain a sum or average of a list of numbers. We need to be able to 'tell' the node that we want it to pass dimensional data into the node function, rather than intercepting it.

This requires a second input parameter (in addition to the 'expected inputs' parameter introduced in Section 4.9) to signify the number of expected dimensions. This is called the 'dimensions' parameter. The 'dimensions' parameter for an input signifies the maximum number of dimensions the function expects to receive on that input, and can be an integer 0 or greater, or can be set to 'Infinity'. As with the 'expected inputs' parameter, the 'dimensions' parameter is specified in a label attached to the input (Figure 4.26), which will also contain the 'expected inputs' parameter if applicable.

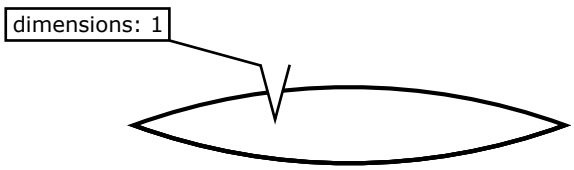

Figure 4.26. Visual depiction of the 'dimensions' input parameter. The 'dimensions' input parameter is shown, together with other input parameters, in a label attached to the input in question.

The default value for this parameter, if not set explicitly (and if not depicted in diagrams), is zero. The 'dimensions' parameter defines the maximum number of dimensions that can

be processed by the node's function. If the number of dimensions received is fewer than or equal to the 'dimensions' parameter, all received data will be passed into the function. If the number of dimensions received is greater than the 'dimensions' parameter, the input is overloaded and only the set number of dimensions will be passed into the function at a time. The first set number of dimensions will be passed in, and the node will iterate over the rest.

As an example, we can modify the example given in Figure 4.24. In Figure 4.27, input 'a' is set to expect one dimension — a list — which it then sums in the function. When this input is provided with a table, it is overloaded by one dimension. The first dimension ([X,Y,Z]) gets passed into the function, and it iterates over the second dimension ([A,B]). Input 'b' does not have a 'dimensions' parameter, meaning it will take the default value of zero and, since it is provided with a list (one dimension), is also overloaded by one dimension, so iterates over that list. The result is a table with two dimensions, with [A,B] as the list of column headings for its first dimension and [10,20] as the list of column headings for its second dimension.

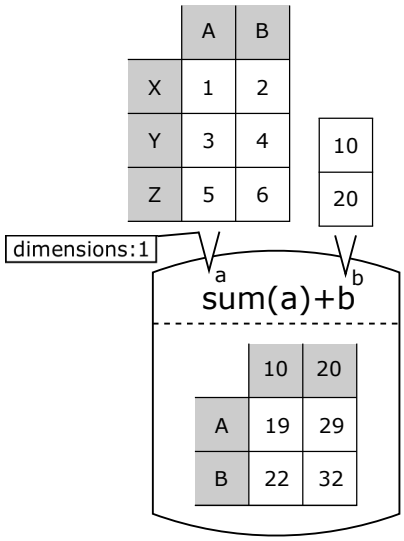

Figure 4.27. Dimensionally overloading an input with a 'dimensions' parameter. In this example, inputs 'a' and 'b' are overloaded by one dimension each, meaning they add a dimension each to the output, resulting in a two dimensional table.

Any of the functionality described so far could be combined with any other functionality in any combination; meaning a node could use dimensional overloading as well as being partially evaluated and using 'expected inputs' to designate an input as expecting a function.

4.11.2. Dependent Iteration

Dependent iteration is where at least one iteration is dependent on the result of at least one other. Some previous approaches to this functionality have involved setting up loops in the graph, with a node acting as a 'switch' to break the loop. This breaks the principle of acyclicity (Section 3.4.3) and relies on a model of dataflow in which connections act as pipes, rather than using the service-provider model.

With partial evaluation, that approach and its shortcomings can be avoided. To perform

an iteration, we must execute some function repeatedly, starting with some initial input, then in each subsequent iteration use the previous iteration's output as the new input, until a termination condition is satisfied, at which point the final output value is delivered as a result. To achieve this, we need to be able to do four things:

- designate a node as an iteration node;
- specify the input over which it should iterate;
- specify a starting value to be applied to the iteration function's input in the first iteration;
- specify a termination condition (or, inversely, a continuation condition), to be applied to the input before each iteration, to decide whether the iteration should terminate. Once terminated, the node should provide the most recent result of the iteration function as an output (or provide the starting value as an output if the iteration terminates before the first iteration).

Here we will depict an iterating node diagrammatically by showing a 'rotation' symbol (🔄) above the designated input, indicating that the node is an iteration function, that it should iterate over this input, and that the input is no longer able to receive connections. An iterating node can only iterate over one input.

When an input is designated as an iteration input, the node must generate inputs on which it can receive a starting value and termination condition. They are given shorthand names '~sv' for the starting value and '~tc' for the termination condition, using a tilde as a prefix (a character that will be forbidden in user-defined input names), to ensure uniqueness. Like inherited inputs, they are depicted and referred to as 'derived' inputs. The starting value mirrors the input parameters of the input being iterated over; the termination condition input has predetermined input parameters of '0' for the 'dimensions' parameter and the item 'nominatedInput' for the 'expected inputs' parameter. If the node has other open inputs (inputs that are not connected), it must partially evaluate and transmit them to its downstream nodes, as it would normally.

An iterating node is depicted diagrammatically as shown in Figure 4.28. The input being iterated over ('input' in this example) is shown with the 'rotation' symbol above it and the two derived inputs ('~sv' and '~tc') have been generated. Although the 'expected inputs' parameter is shown in this example, it is not always be shown in diagrams, since all '~tc' inputs have the same input parameters.

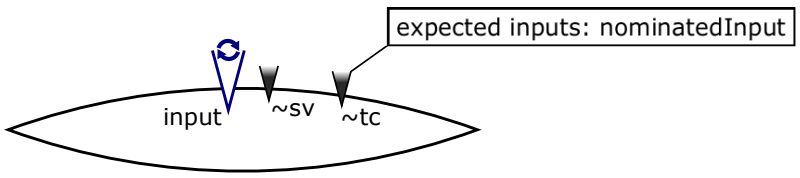

Figure 4.28. Depiction of dependent iteration. A node is depicted as having been set to iterate over a particular input with a 'rotation' symbol placed over the input in question. The node will then generate two derived inputs, for the starting value ('~sv') and termination condition ('~tc'). The termination condition has an expected input named 'nominatedInput'.

A simple example of this functionality would be to use an iteration to count to ten. In the example shown in Figure 4.29, the main function adds one to its input. It is set to iterate over its input, is given the starting value '0' and, as a termination condition, a function which outputs true if its input is '10'.

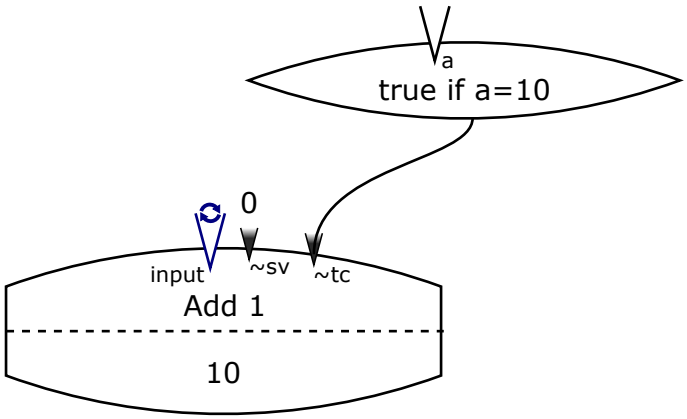

Figure 4.29. Using dependent iteration to count to ten. In this example, a node whose function adds one is set to iterate over its input. Its starting value is '0' and it is set to terminate when its output reaches 10, at which point it outputs '10'.

This function starts with the number '0'. It tests this against the termination condition, receives the output 'false', proceeds to the iteration function, which adds one, producing the output '1'. It then applies this to the input of the termination condition again, which outputs 'false', and the new number is applied again to the iteration function; and so on. On the final iteration, the iteration function produces the output '10'. This is applied to the input of the termination condition, which this time outputs 'true', telling the node to terminate the iteration, at which point the node will output the most recent result of the iteration function, which is '10'.

A common example of iteration would be to use a counter to perform an iteration a set number of times. This is more similar to the way a normal 'for' loop works. An example is shown in Figure 4.30.

In this example, the 'Iteration' node has one root input, named 'dataAndCounter', which expects an object containing a 'data' property and a 'counter' property. Its functionality uses this input to iterate the data, increment the counter, and output a combined object containing the new values.

When we set its 'dataAndCounter' input to iterate, it generates the '~sv' and '~tc' inputs. We connect the '~sv' input to a node that delivers starting values of both; and we connect the '~tc' input to a node that outputs a partially evaluated function that returns 'true' when the counter reaches a chosen limit. Because the 'Tc' node has two inputs, we need to explicitly nominate the 'feedback' input as the expected input for '~tc'.

The output of the 'Iteration' node is a node that has three inputs: 'startingData', 'startingCounter' (both inherited from the 'Sv' node) and 'limit' (inherited from the 'Tc' node). We could use this output to perform the iteration with any starting data, starting counter and limit we chose.

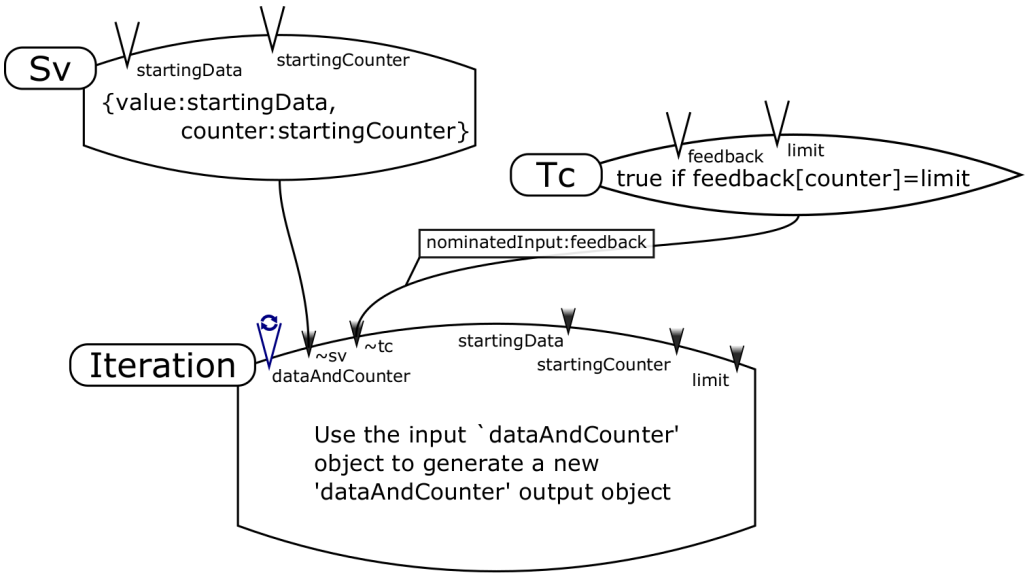

Figure 4.30. A structure similar to a 'for' loop. This structure creates a node that has inputs for the starting data, starting counter and the 'limit' (which determines the counter value at which the loop should terminate).

As with the features introduced previously, the dependent iteration feature can be combined with all others. This means that the node can be partially evaluated, any inputs can have 'expected inputs', and the node can receive functions or dimensional data on any of its inputs.

4.12. The Generalised Iteration Node

This section illustrates how a generalised iteration node can be built using the features described in this chapter. The aim of a generalised iteration node is to create a node that can be used for any iteration, with similarity to the iterative structures we could expect most users to be familiar with.

4.12.1. Common Iterative Structures

The most common iterative structures in most programming languages are 'for' loops, 'do/while' loops and recursion. JavaScript, as an example of a C-family language, provides all three. An example of a 'for' loop is shown in Snippet 4.1.

Snippet 4.1

```
for (let i = 0; i < 10; i++){
  ... // iteration statements
}
```

The 'for' loop in JavaScript contains four statements: the first three contained in parentheses, followed by the iteration statement, comprising a collection of statements enclosed in curly braces. In parentheses, we have:

- the initialisation statement ('let i = 0' in our example). This is most often used to define some form of counter;

- the continuation condition ('i < 10' in our example). This returns a value, coerced to a Boolean which, if 'truthy', will allow the iteration to continue and if 'falsy' will terminate the iteration;
- the step function ('i++' in our example). This is executed after the iteration statement in each iteration, and is usually used to increment or 'step' a counter from one iteration to the next.

The 'for' loop works by executing the initialisation statement, followed by the continuation condition (exiting the loop if false), then the iteration statements and finally the step function, repeating until the continuation condition returns false. JavaScript's 'do/while' loops are similar to its 'for' loops, but without the initialisation and step function statements. They come in two variations, giving the programmer the option to evaluate the continuation condition either before the iteration statement (in a 'while' loop) or after iteration statement (in a 'do...while' loop). Examples of both are shown in Snippet 4.2.

Snippet 4.2

```
// while loop
let testVal = 0;
while (testVal < 10){
  ... // iteration statements
}


// do...while loop
let testVal = 0;
do {
  ... // iteration statements
} while (testVal < 10);
```

The 'while' and 'do...while' loops include only an iteration statement (in the curly braces in these examples) and a continuation statement. The same functionality as the 'for' loop can be achieved by preceding the loops with an initialisation, and including the step function within the loop. The 'do...while' loop allows the programmer to guarantee that the iteration statement will run at least once, whereas the 'while' statement allows the loop to be exited before the iteration statement is first run.

Recursion in JavaScript, and other programming languages, allows a function to call itself. An example is shown in Snippet 4.3.

Snippet 4.3

```
// First define the recursive function
function recursiveFn(a){
  // ... iteration statements

  if (a === 0) return 0;
  else return recursiveFn(a - 1) + 1;
}

// ... then call it.
let result = recursiveFn(10);
```

There are six important components of a recursive function.

- **The iteration statements** ('// ... iteration statements' in our example). These are executed in every iteration.
- **The termination (or, inversely, continuation) condition** ('if (a === 0)' in our example), which determines whether the recursion should continue.
- **The base case statements** ('return 0;' in our example), which are executed when the recursion terminates.
- **The continuation case statements** ('return recursiveFn(a - 1) + 1' in our example), which are the statements to be executed if the recursion does not terminate. These will include the recursive function call ('recursiveFn(a - 1)' in our example).
- **The step function** ('a - 1' in our example). This determines the new value used for the argument in the internal function call.
- **The initial value** to be provided as an argument to the recursive function (10, in our example, which is provided within the initial function call: 'let result = recursiveFn(10)').

Recursion is available in JavaScript (and many other languages) for convenience, but is not a necessary programming construct: it is a form of loop, and is implemented by the language using hidden loops. Since they are implemented with loops, it is always possible to implement a recursive algorithm as a loop in a way that is at least as efficient as the recursive version of it. Keller [1980] showed, specifically for dataflow, that it is possible to use a variation of the partial evaluation concept (what he called the 'Apply' operator) to eliminate recursion. Although this performs the recursion from a computability point of view, it does not necessarily address the potential for parallelism.

Recursion, with its potential to enhance convenience and parallelism (and in particular with respect to multiple recursion) will be reserved for further work. Instead, in this section, we address how to achieve generalised loop-like iteration in dataflow.

4.12.2. Loop-Like Iteration

A generalised iteration node is created here in eight steps, described below.

4.12.2.1. Step 1 — An 'ID' Node

The starting point for a generalised iteration node is that we need an open input to which a later user can connect the iteration function. We create this using an 'ID' node — a node with just one input that provides its input unchanged as an output. An example is shown in Figure 4.31, which has a single input named 'iterationFunction'.

4.12.2.2. Step 2 — Setting Expected Inputs

We want this central ID node to expect to receive the iteration function on its input. In each iteration, the iteration function should take the previous value as an input and

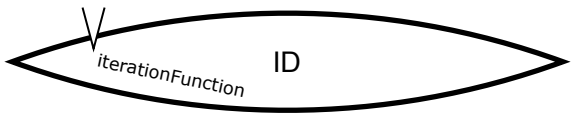

Figure 4.31. Step 1. The central node is an 'ID' node, meaning it returns its input unaltered.

output the new value. As described in Section 4.9, we achieve this by setting the 'expected inputs' input parameter to expect an input, in this case named 'prev'. As shown in Figure 4.32, when we set the input parameter, the node generates derived inputs corresponding with the expected inputs, in this case generating one derived input named 'prev'.

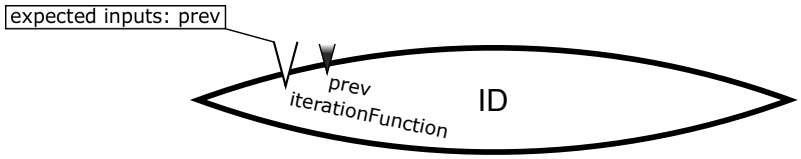

Figure 4.32. Step 2. The 'expected inputs' parameter of the 'iterationFunction' input is set to 'prev'. This tells the node it should expect to receive a function with one input. In response, it generates a derived input of the same name.

4.12.2.3. Step 3 — Iterating Over the 'prev' Input

It is this new derived input, 'prev', which we want to iterate over. We can use the 'dependent iteration' functionality defined in Section 4.11.2 to achieve this (Figure 4.33).

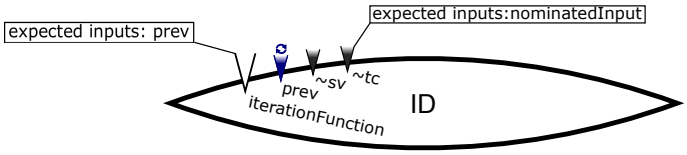

Figure 4.33. Loop-like iteration, step 3. The node is set to iterate.
Step 3. The node is set to iterate over its input named 'prev', which prompts it to generate derived inputs for the starting value ('~sv') and termination condition ('~tc') of the iteration.

The 'prev' input is designated as an iteration input (depicted here with the 'rotation' symbol displayed above it). In response, the node generates derived inputs for the starting value and termination condition, named '~sv' and '~tc'. As described in Section 4.11.2, the termination condition has the expected inputs parameter 'nominatedInput'.

This is enough for a simple iteration. However, iterations more often use some form of counter or supplementary data to compute new values or to decide when to terminate. To make it as easy as possible for users, we want to make our generalised node more similar to the 'for' loops described in Section 4.12.1.

4.12.2.4. Step 4 — Splitting and Recombining the Data

Since a node can iterate over only one input, we need a way for that input to convey not only the previous value but also a counter. We do this by combining both into a single item of data. To perform the calculations, we will then split this combined item of data into its 'value' and 'counter' components, before recombining them again as an output,

ready to be fed back into the next iteration. This action to split and then recombine data requires a graph of the form illustrated schematically in Figure 4.34.

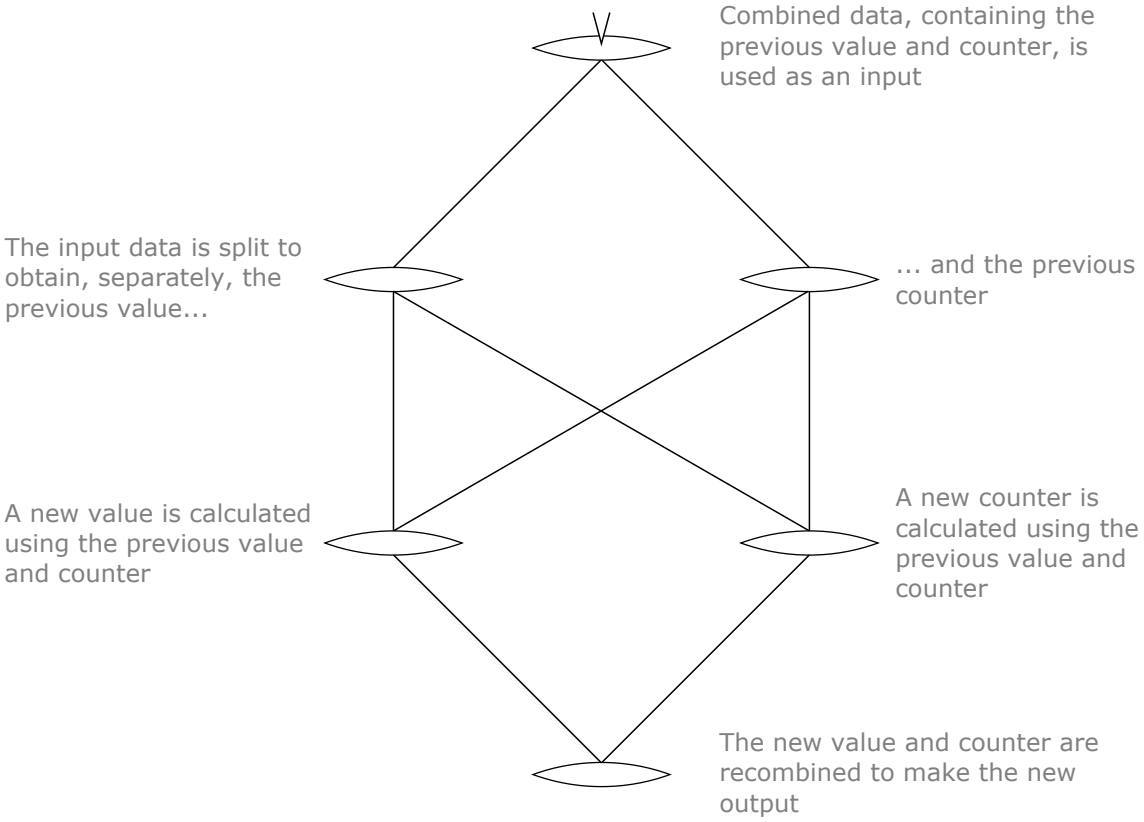

Figure 4.34. Generalised Iteration — The split-recombination form. The general form of an iteration requires a combined data input containing the previous value and counter, which are then split to obtain them separately so that they can be operated on separately, before recombining the new value and counter into a combined object.

In order to build this structure, the splitting and combination actions require three components:

- the 'Combiner' node — a node that takes a value and counter and combines them into a single object;
- the 'GetValue' node — a node that takes a combined object and obtains the value;
- the 'GetCounter' node — a node that takes a combined object and obtains the counter.

These three components are shown in Figure 4.35. In our version of the split-recombination graph, we need the user to be able to provide, as well as the iteration function, the function to calculate the new value and the function to calculate the new counter, using the previous value and counter each time to calculate the next.

In Figure 4.36, we have a skeleton of the required graph superimposed on the split-recombination form (shown by the grey dashed line), with the three components outside that section of the graph. In this structure, we have the combined input at the top (labelled 'comb'), a node to obtain the value (derived from the 'GetValue' node), a node to obtain the counter (derived from the 'GetCounter' node), and the combiner at the bottom.

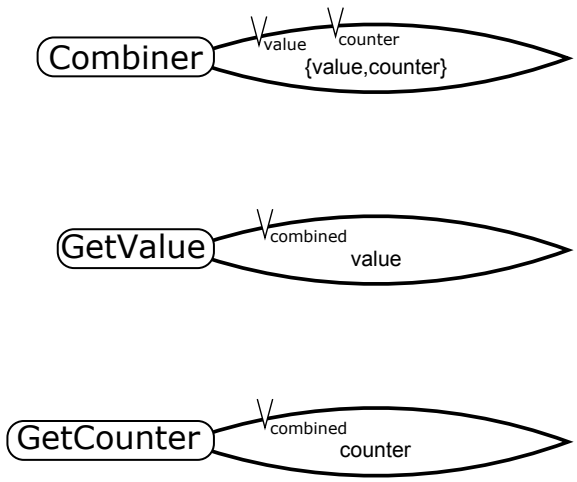

Figure 4.35. The generalised iteration requires three components: the 'Combiner' node, which takes a value and counter and combines them into a single object; the 'GetValue' node, which extracts a value from a combined object, and the 'GetCounter' node, which extracts the counter from a combined object.

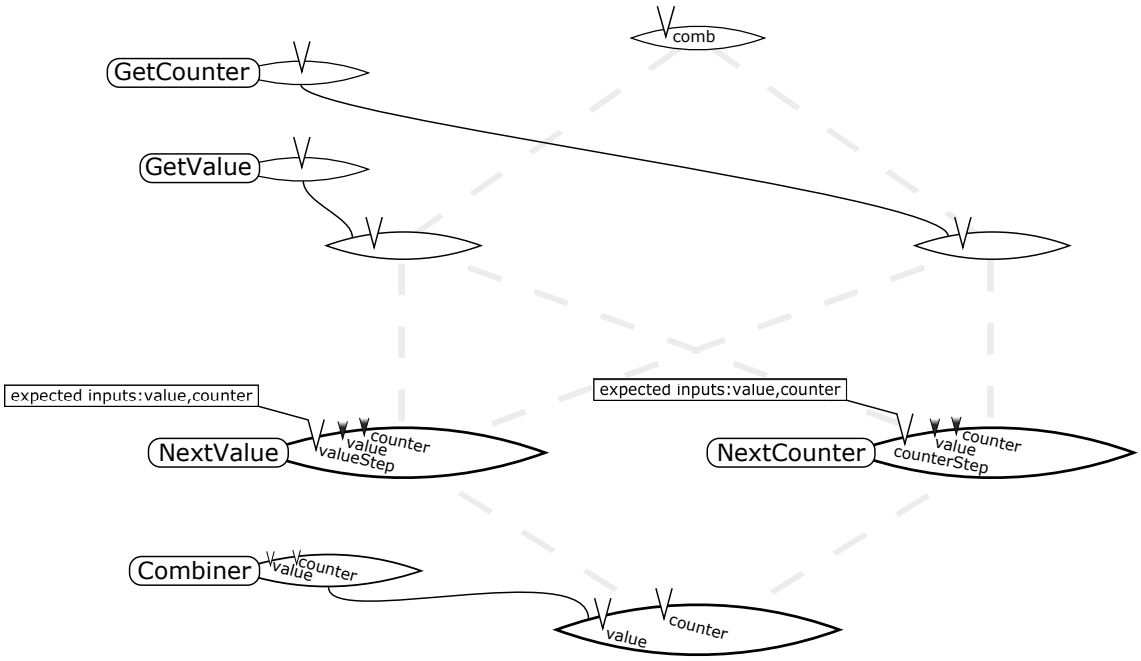

Figure 4.36. Step 4. Using the Split-Recombination form together with the three components, we can set up the skeleton of our general iteration graph.

In the lower left side of the split-recombination graph, the 'NextValue' node has an open input named 'valueStep' with the expected inputs 'value' and 'counter'; and has generated the two corresponding derived inputs accordingly. In the lower right side of the graph, the 'NextCounter' node, similarly, has an open input labelled 'counterStep' with the same expected inputs, 'value' and 'counter'. These two inputs, 'valueStep' and 'counterStep', will be inherited through the graph and enable us to define later how we want to calculate the new value and counter from the previous value and counter.

4.12.2.5. Step 5 — Connecting the Split-Recombination Graph

When we connect the nodes together in this graph, as shown in Figure 4.37, the open inputs are all inherited by their downstream nodes and therefore all appear in the final node. To create a clearer output at the end, a final node has been added, labelled 'IterationFn', which shows the inherited inputs.

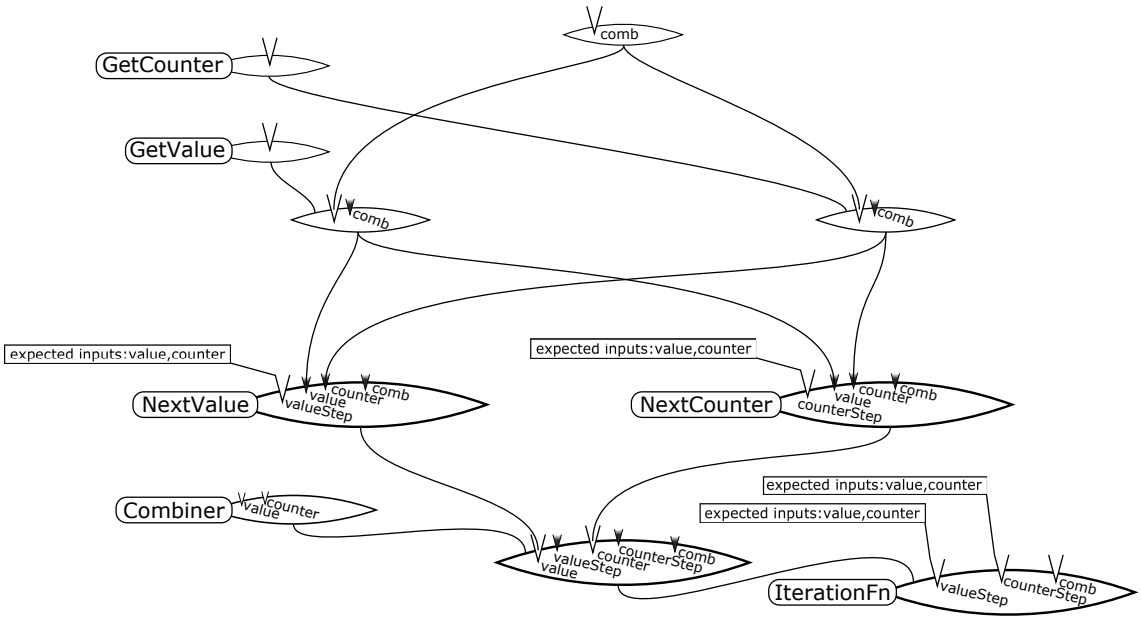

Figure 4.37. Step 5. The nodes are connected together. The open inputs cascade down the graph and are inherited by the 'IterationFn' node at the end.

This final node is the one that can be connected to the input labelled 'iterationFunction' in Step 3 (Figure 4.33). The node in Step 3 also has inputs for the starting value ('~sv') and termination condition ('~tc'). These require similar processes of splitting and combining data.

4.12.2.6. Step 6 — Defining the Termination Function

The termination condition input shown in Step 3 (Figure 4.33) expects a function with one input (called 'nominatedInput'). We want a future user to be able to provide this function and can set up a graph to make that easier. Like the value step and counter step functions discussed in Step 4, we want the user to be able to provide a function that takes the previous value and counter as inputs.

Shown in Figure 4.38, the core of this functionality is a node with an input labelled 'terminationFn'. This input has 'value' and 'counter' as expected inputs, which are therefore generated as derived inputs. As in Step 4, the 'GetCounter' and 'GetValue' components are used to separate values from initially combined data. The combined input, labelled 'comb', cascades through the graph and is inherited at the end. As before, a final node is added for clarity, in this case labelled 'TerminationFn'. This is the node that will be connected to the termination condition input (labelled '~tc') in Step 3.

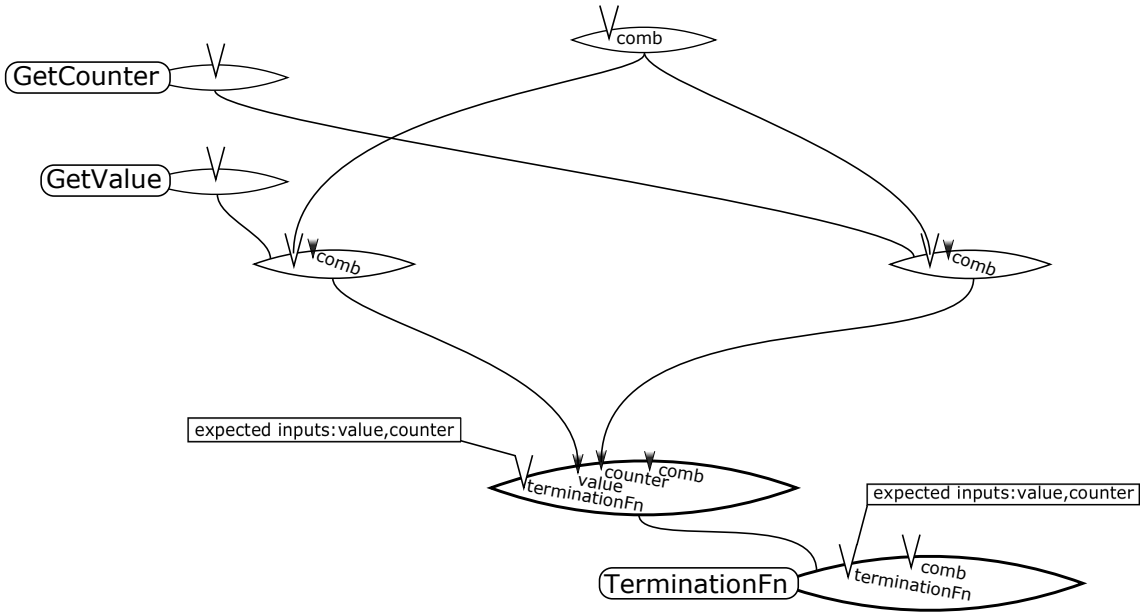

Figure 4.38. Step 6. The termination function takes a similar but simpler form, splitting the combined object into its components to make separate inputs to the termination function, which will return 'true' to terminate or 'false' otherwise.

4.12.2.7. Step 7 — Defining the Starting Value

For the starting value, we want our future user to be able to define the starting value and starting counter separately. We can do this by using our 'Combiner' component. This is simple enough, except that its input names, 'value' and 'counter' are less expressive than would be ideal. To make these input names more communicative, we can connect these inputs to upstream nodes with differently named inputs, effectively renaming them.

This is shown in Figure 4.39, with the inputs renamed in this way to 'startingValue' and 'startingCounter'. Once again, a final node is added, named 'StartingValueFn', so that we can clearly see the end product with its set of inputs.

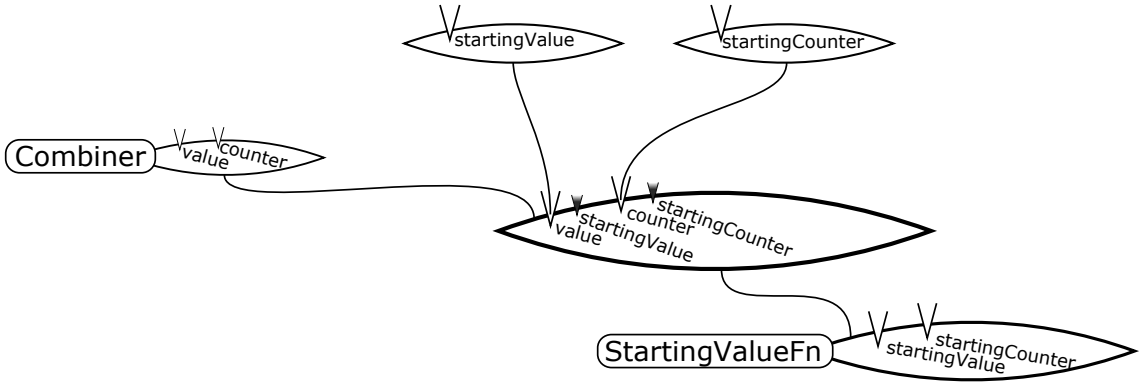

Figure 4.39. Step 7. The starting value node is simple, but in this case two additional nodes have been added which have the effect of changing the names of the inputs to make them more expressive: 'value' being renamed to 'startingValue' and 'counter' being renamed to 'startingCounter'.

4.12.2.8. Step 8 — Connecting the Components

Finally, these three new components — 'IterationFn', 'TerminationFn' and 'StartingValueFn' — can be connected to the component shown in Step 3 to obtain our end result.

In Figure 4.40, these three nodes are connected to the central ID node. When using the 'expected inputs' functionality, we have to specify which input of the connected function corresponds to which expected input. So, when the 'IterationFn' node is connected to the 'iterationFunction' input, we must use the connection parameter to specify that input of the received function named 'comb' should be treated as the expected input 'prev'. The ID node then inherits the other two inputs of the connected function — 'valueStep' and 'counterStep' — both of which have expected inputs 'value' and 'counter' (recall Step 5, Figure 4.37).

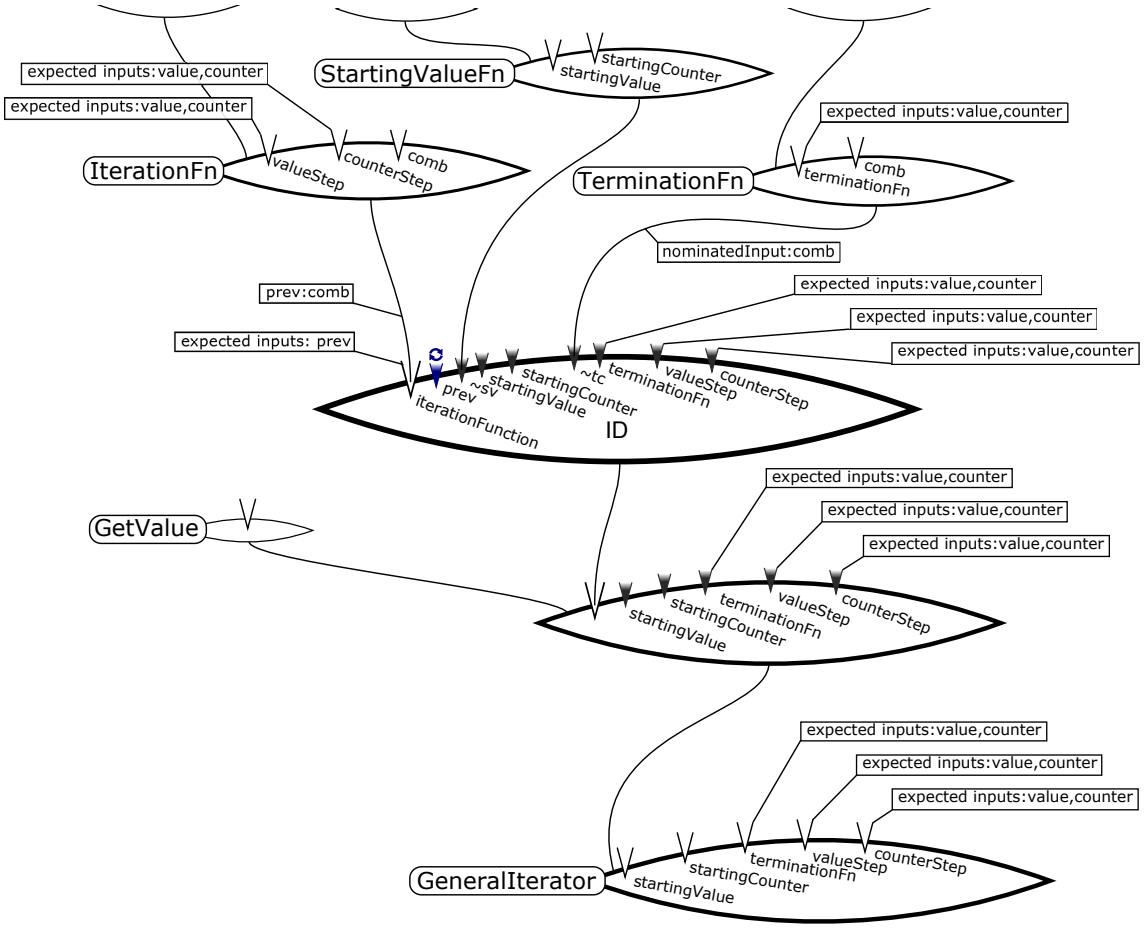

Figure 4.40. Step 8. The components are combined to create the 'GeneralIterator' node as a result.

The termination function input, named '~tc', has an expected input (not shown) named 'nominatedInput', which is associated via the connection parameter to the input of the connected function called 'comb'. The ID node inherits the other incoming input, called 'terminationFn' which, like 'valueStep' and 'counterStep', has expected inputs named 'value' and 'counter' (recall Step 6, Figure 4.38).

Since the output of this node is the combined object (containing both the final value and the final counter), we can use one of our components to extract just the value. The 'GetValue' component is used again here to extract just the value, and inherits all the open inputs.

Finally, as in previous cases, a node is added to the end to give us a clearer view of the end product, in this case named 'GeneralIterator'. This is our end result: a node that

can be used to perform any iteration, with five inputs ('startingValue', 'startingCounter', 'terminationFn', 'valueStep' and 'counterStep'), of which three have an expected input setting. The result is shown in Figure 4.41.

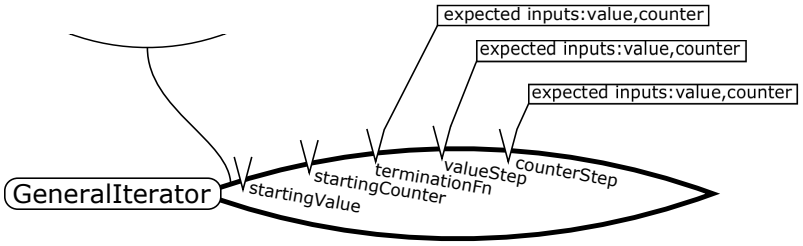

Figure 4.41. The Generalised Iteration Node. The final node contains inputs for the starting value, starting counter, termination function, value step and counter step. The last three have expected inputs of 'value' and 'counter'.

Having defined this general iterator once, if used in a unified global dataflow coordination system, it could be reused in any other iteration with a single value and counter. Future programmers could use this component rather than having to redefine it themselves.

Whereas this iteration iterates over two values — a 'value' and a counter — a similar technique could be used to define a general iterator with three, four or any other number of values, or to modify this iterator to create one capable of handling more components. An even more generalised iterator could be conceived, capable of performing an iteration with an arbitrary number of values. For example, we could imagine functionality in which an input to the general iterator tells it how many values are involved in the iteration, and the node would generate the required number of inputs (a starting value and step function) for each. Thus, an iteration with two components (as in the example described above) would generate four inputs in addition to the termination function.

To achieve this would require additional features. Section 7.2.1 (Manipulating Partially Evaluated Functions and Graphs) describes a hypothesised feature that could contribute to such functionality, though the extent to which it would be useful in this situation is still open to debate. As the number of components increased, the generalised version of the iteration would become increasingly cumbersome, making it arguably easier to create one from scratch.

4.13. Function Isolation

The fact that nodes run functions received via their inputs exposes them to potential harm from malicious upstream nodes. It could result in code that leaks data, or changes the outcome of subsequent executions.

To prevent this, the system needs a way of isolating untrusted code. To some extent, it is a matter of node autonomy for them to defend themselves and their execution engines from potential harm, so it does not need to be dictated by the system. However, since it is a universal problem, it is worth considering how this can be done, and establishing a standardised approach.

There are several methods by which code can be isolated, with varying levels of resource cost. The simplest method to implement, provided by some operating systems, is to run code in a virtual machine. This requires creating a new virtual machine for each execution, and destroying it when completed. It is an effective method, but carries a high resource cost. Another method would be convey functions in the form of their abstract syntax tree, to be compiled by the function using a compiler that only includes safe language features. This would avoid the virtual machine overhead. A third possible approach would be to make the code safe at the time it is created or edited, so that it can be run safely without needing a virtual machine — doing the work at save-time rather than run-time.

The same code may be executed multiple times, at time-critical moments. This means a virtual machine would incur its overhead repeatedly. However, we can expect code to be created or edited only infrequently in comparison, and at non-time-critical moments; work done instead at the time the code is saved would only incur its overhead infrequently. One way to do incur the overhead at save-time rather than run-time would be to parse the code at save-time using a language parser that only included the safe language features. This is illustrated in Figure 4.42.

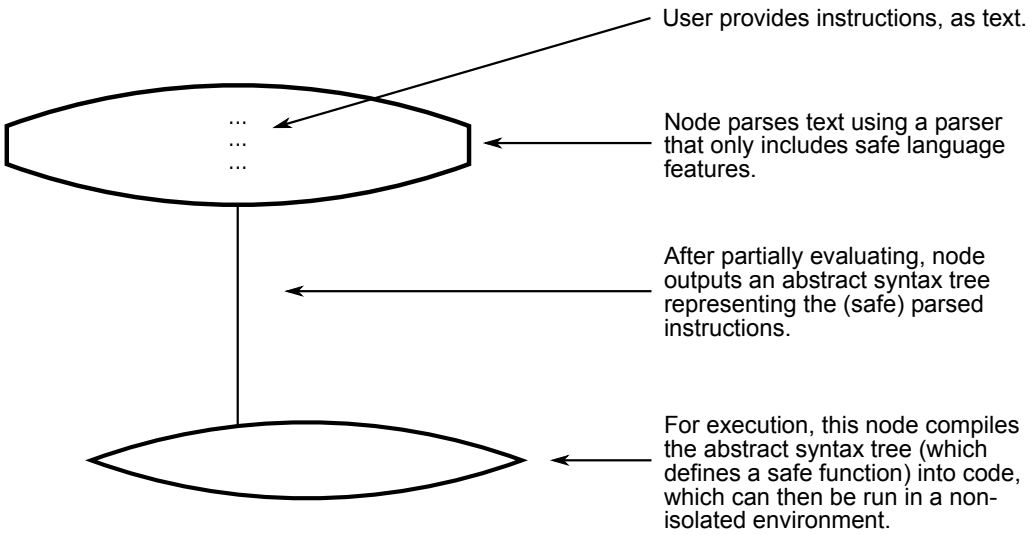

Figure 4.42. Parsing code for non-isolated environments. Untrusted code could be parsed and recompiled into a trusted version.

Code that has been made safe could then be run in a less isolated but more efficient environment. The Google Caja product[3] provides something like this functionality. This approach depends to some extent on trusting inputs that arrive from other nodes, but allows for the possibility of setting up networks of trusted host machines, through which code safety could be certified.

For nodes that do not trust each other, the advantages are not lost. Where a node has multiple inputs, they do not necessarily all arrive at once. If one input received a function, the node could pre-emptively parse that function to make it safe, before receiving values on its other inputs, enabling it to perform a fast execution when the last of its inputs arrived.

[3] `https://developers.google.com/caja/`

A similar approach provides us with a way to make nodes indifferent to the language used by the user to enter instructions. A safe parser could be written to parse a language in which untrusted parsers could be written, which would make the system extensible to other as-yet unwritten languages — enabling arbitrary language parsers to be written by anyone who wanted to write node functions in a language not previously available. Combined with a central register of these third-party languages, it would mean a node function could be written in any language for which a parser had been written, which could in turn be executed by any node.

Such a system is would be built up in layers. In the bottom layer, we have the underlying hardware, which has its own base language. Written in this base language is the safe parser, which defines the language suitable for running untrusted code. This is the 'parser parser'; that is, a parser for parsing the untrusted code that defines a third party parser.

Using this parser parser, it is safe for an untrusted user to submit untrusted code that defines a new language. The parser parser will parse the untrusted parser into a safe version of it written in the base language. This safe version of the parser could then be used to run the untrusted code submitted by untrusted users as node functions. The two key steps: first defining, and then using, the new language are illustrated in Figure 4.43.

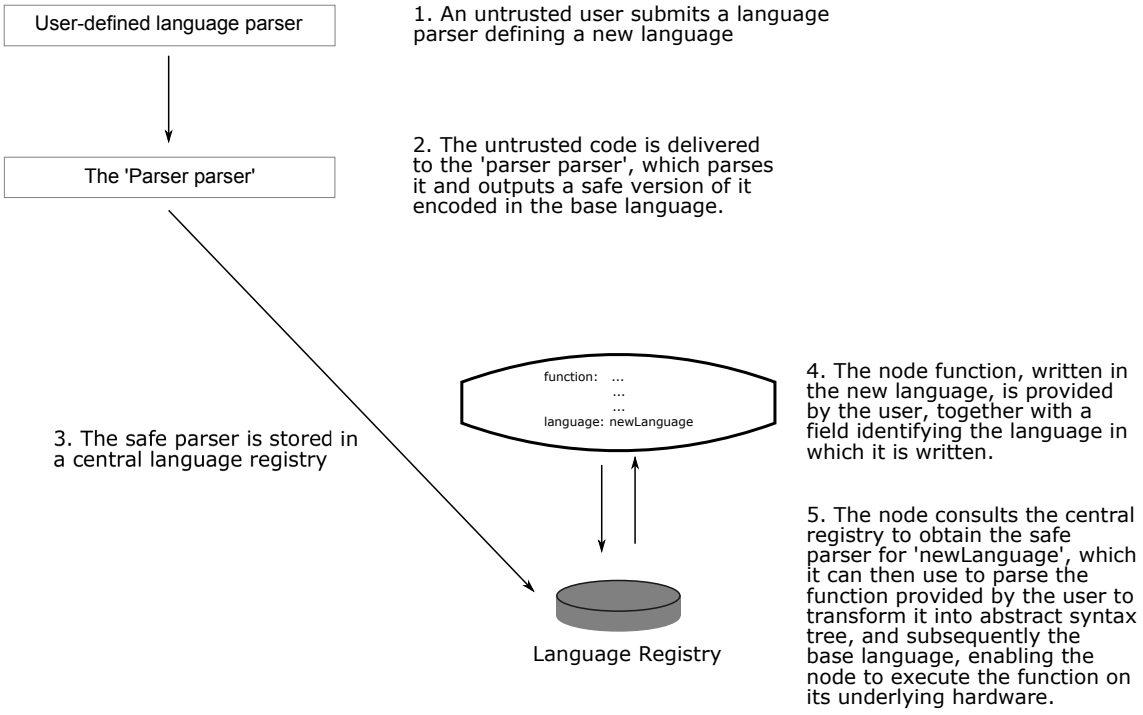

Figure 4.43. Defining a new language. A parser can be used to make code safe. A parser for parsing a user-defined language parser would enable any user to define a language that other users could use to write functions.

This infrastructure depends on there being a single universally understood language, executable by all hardware. As discussed in Section 4.1, this cannot be relied upon either.

Parallel language parsing infrastructure could be written for more than one type of hardware. It could be that some functions require one of a list of types of hardware, and a node receiving them would have to find suitable hardware on which to execute them. The loca-

tion of applicable hardware could, perhaps, be stored along with the language definition in a central registry. Many of these implementation issues fall beyond the scope of this work, but the aim of achieving this level of flexibility is desirable, and the infrastructure needed to achieve it seems at least plausible.

4.14. Notifications and Time-Stamps

As discussed in Section 4.2 (Functional Purity), the system is envisaged as a live, dynamic interactive system, in which nodes may at any time be updated, both during development and operational use. While this deviates from the traditional model of dataflow, in which a graph is seen as statically representing a single execution of a program, this model is chosen for its utility and closer representation of real-world programming realities. However, the propagation time between upstream and downstream nodes does have the undesirable effect of introducing latency. This can lead to two types of error:

- out-of-date errors — where a node's output value is out of date but was correct at some previous point in time;
- incorrect-value errors — where inputs arrive out of order and as a result a node generates an output value that does not correspond to any synchronous set of inputs and was not correct at any point in time.

Out-of-date errors occur between the moment an upstream node is updated and the moment a dependent downstream node's output is updated accordingly. Because there is always a propagation time, this type of error is a certainty: it will occur with every update, and the best we can hope for is to mitigate it by minimising the amount of time for which it occurs.

Incorrect-value errors happen when an output corresponds to a combination of inputs that never occurred. Imagine we have the graph shown in Figure 4.44. In this graph, an update from Node A will arrive at Node B via two routes — one long and one short.

In this situation, an update from Node 'A' could arrive at Node 'B' and, depending on the relative speeds of transmission and computation, Node 'B' could update its value and report its result before the new input value arrives via the longer route. Only two values can be correct at Node 'B' — the one corresponding to the old value at Node 'A', or the one corresponding to the new value at Node 'A'. If Node 'B' outputs a value corresponding to the old value via its first input but the new value via its second input, it is an incorrect-value error.

Both types of error can be mitigated if we introduce the concept of an 'update notification'. An update notification is a message sent by a node to its subscribers whose purpose is to propagate the information that *an update has taken place*, even if the updated data itself is not yet ready. In order to reconcile notifications with their corresponding data updates, both must contain data to uniquely identify the originating update. This could be done using the node of origin with a time-stamp, or a user interface identifier with a time-stamp.

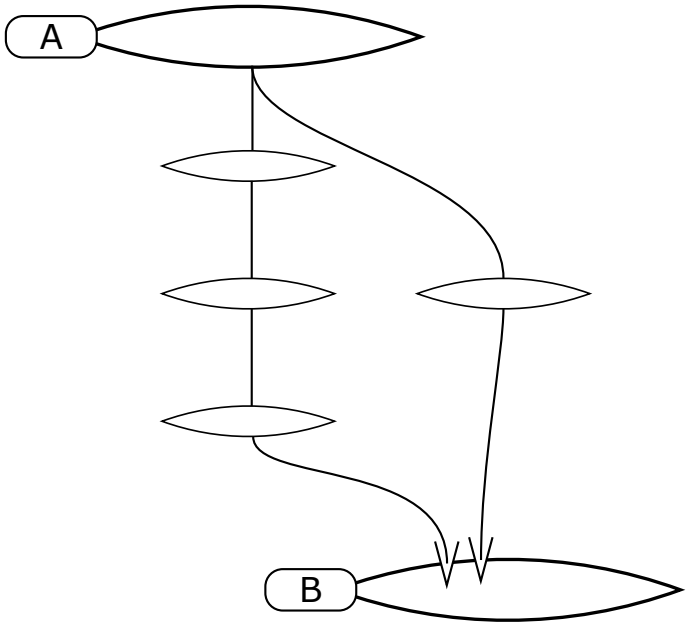

Figure 4.44. Update arriving via two routes. In this structure, when Node 'A' is updated, the new data is likely to arrive at Node 'B' sooner via the short route than the long route. This could cause Node 'B' to execute and compute a new output before the new value arrives via the longer route. This would generate an incorrect-value error.

When a node is updated, or when it receives an update notification from an upstream node, it will immediately (before starting to compute its output) send an update notification on to its subscribers. In this way, the notification is able to propagate quickly through the graph without being delayed by the computation. It does not speed up the delivery of the corresponding data but does allow nodes to 'know' that new data is imminent and to make decisions accordingly.

If a node receives a request for its output in the interval between receiving a notification update and having new data available, it can choose to provide the previous (most recently available) value, or wait and deliver the upcoming value when it is ready, depending on the priorities of its clients or owners.

Incorrect-value errors can in some cases be eliminated. In the graph shown previously, in Figure 4.44, it is likely (depending on the propagation times) that Node 'B' will receive the notification update via the long route before it receives new data via the short route and has time to compute a new output value. When it receives the notification via the long route, it can immediately recognise that the second update notification originates with the same update as the first, and can wait until both have arrived before computing its new value. Only if there are more extreme differences in the lengths of routes is it likely to experience an incorrect-value error, and even then for a shorter period of time than it would without update notifications.

It is possible that a change to a node's input may not change its output. This could be a characteristic of that particular computation, or could be a result of using triggers as described in Section 4.7. In either of these scenarios, to avoid downstream nodes having to recompute their values unnecessarily, it is helpful to be able to cancel an update

notification. To account for this, we can introduce an update notification cancellation message. If this message is received by a node then, if no other changes have taken place, it can send the cancellation message on to its subscribers and revert to its cached output value if it has one. This avoids having to recompute its value or resend previous data updates to its subscribers.

4.15. Subscription Types

If updates occur with high frequency, it brings about the possibility that the interval between updates could be shorter than the time taken to compute outputs. Propagation times, particularly in networked environments introduce the additional complication that updates could arrive out of order. These two issues present nodes with a choice about how to serve their subscribers.

The first decision a node owner has to make is whether the node should produce an output for every set of inputs, even if that means computing values for inputs already known to be outdated. This would be needed if, for example, the feed of values were being used to assemble a time series. If so, the node will need to request the same of its upstream nodes and, if the interval between updates is shorter than the computation time, will have to trigger multiple computations in parallel (which is possible with the separation between nodes and their resources discussed in Section 4.4).

If historical output values are not needed, the second decision the owner has to make is whether the node should be allowed to output values that are known to be out-of-date. If not, the node can abort a computation if a new notification is received while it is under way, and ignore data it receives that it knows to be outdated. This will produce more up-to-date outputs but has the danger that, in a high frequency environment (in which the interval between updates is less than the computation time), computations might never finish without being superseded and aborted.

Between these two extremes of computing every update and computing only those believed to be current, the decision is more complex. The node can make choices about whether to propagate outdated notifications if they arrive out of order; whether to commence computations on outdated data if superseding notifications have already been received or computations started; whether to abort computations if superseding notifications are received or computations started while they are under way; or even whether to ignore newer data updates that arrives while a computation is under way. These decisions can be decided by node owners or agreed through contracts — in this case, types of subscriptions — between nodes and their internal or external clients. Nodes are, however, dependent on their upstream nodes for their ability to deliver their service. Each node would, therefore, need to request at least as high a level of service of its upstream nodes as it delivers to its clients, causing high service level requests, like subscriptions themselves, to escalate up the graph.

A variation of the problem is if updates occur with a higher frequency than is needed by a downstream node. The most extreme version of this problem is if the data is a continuous

series. In this case, it may be necessary to throttle the updates using a subscription that contains a specific frequency or a maximum frequency with which updates should be delivered.

4.16. Synchronisation

Distributability is a key component of the system. With nodes hosted on different machines, their clocks will not be perfectly synchronised. If update notifications include the identity of the machine on which the update originated, we can reliably compare time-stamps that originate in the same place. For those that originate in different places, the problem remains.

This problem was discussed briefly by Bainomugisha et al. [2013], who suggested two solutions. Their first suggestion was to use a centralised clock, which provides a consistent time-stamp but, they pointed out, creates a single point of failure, a bottleneck and a communication overhead. Their second suggestion was to treat events that occur within a certain time interval of each other as having occurred simultaneously, pointing out that it might only be feasible for programs with relatively infrequency changes.

Because simultaneity is transitive property but temporal proximity is not, if the number of updates is high it could result in a chain of updates, all treated as having occurred at the same time, which actually stretches the 'effectively-simultaneous' interval far beyond what was intended. In the graph shown in Figure 4.45, we have three nodes, 'A', 'B', and 'C', all connected to a single downstream node, 'D'.

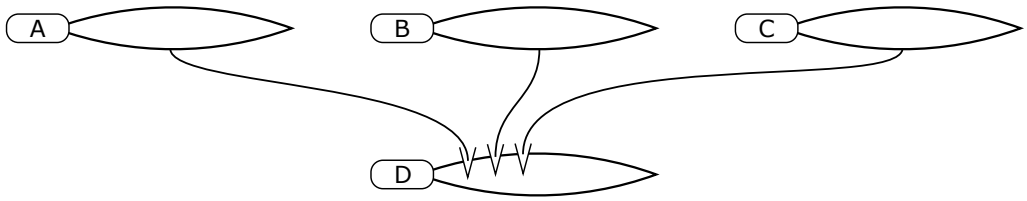

Figure 4.45. Multiple connections with high frequency updates. In this scenario, Node 'D' has three upstream nodes, all delivering high frequency updates.

As illustrated in Figure 4.46, if several inputs all update with high frequencies, where the interval between them is within the 'effectively simultaneous' interval, it sets up a situation in which successive updates end up having to be treated as if they occurred simultaneously, and setting up a chain of 'simultaneous' events that might never end.

Node autonomy can help. The importance of sequencing varies greatly between applications, as do the frequencies of updates, the latencies in communication between machines and the accuracies of computer clocks. In most situations, where the frequencies of updates are low compared with transmission times and differences between clock times, and there are low consequences of presenting an incorrect sequence or an output that is temporarily in error, the problem can be ignored and time-stamps taken at face value.

In the small number of situations in which consequences and update frequencies are high, models can be structured and node behaviour configured accordingly. In these situations,

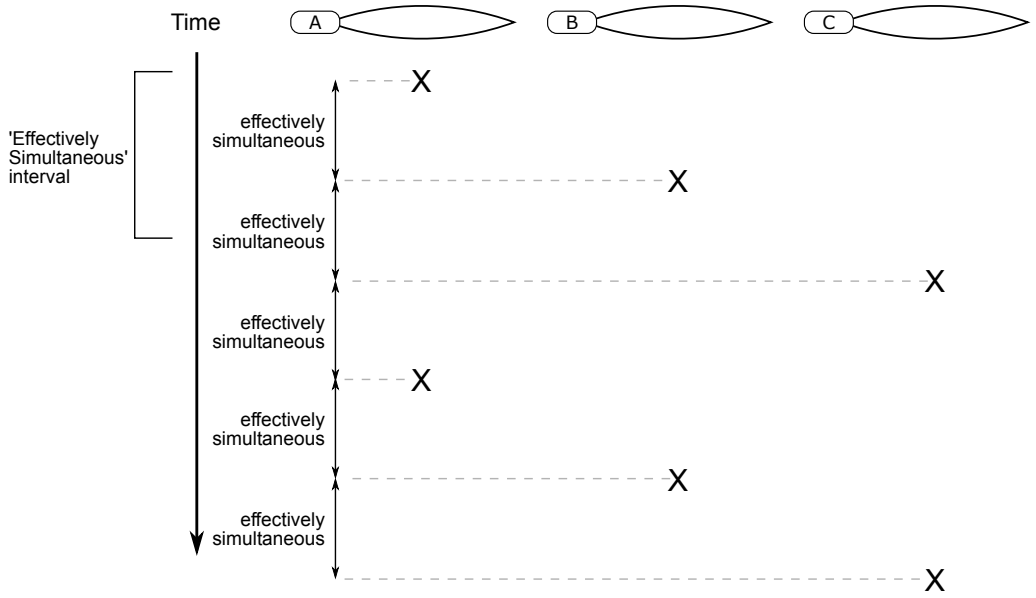

Figure 4.46. High frequency updates time chart. In a scenario with multiple high-frequency updates to a node, if we were to consider updates within a certain interval to have been effectively simultaneous, it could create an endless chain of 'simultaneous' events stretching far beyond the intended interval.

it may be that nodes have to be hosted together on the same machine, or in the same location, and more effort put into obtaining and coordinating accurate time-stamps on those machines; and in some cases, the communication overhead involved in using a central time server for a set of nodes may be worth the gain.

In some situations it may be an adequate solution to regard updates that take place within a certain interval as having occurred simultaneously. There is no need to constraint nodes to set solutions. In a distributed environment, the time reported by an update notification tells another node something about when it happened but does not provide exact information or clearly-defined time-box. Perhaps one way to view it is to think of a time-stamp as indicating a probability distribution of time over which an update can be suspected of having taken place. This could, perhaps, be supplemented with information gleaned elsewhere about particular machines in order to assemble a most-likely sequence of updates. Although the existence of standard solutions may ultimately be helpful, the question can largely be left to node autonomy and remain open to future extension and modification.

4.17. Testing and Development

Testability is one of the architectural principles listed in Section 3.4.1. Interactivity of the graph provides the quickest and simplest form of support for this, meaning mistakes are revealed as they happen and the developer can always see the results. Partial evaluation, expected inputs and independent iteration can be used to improve the testing functionality. These are discussed in the sections below.

4.17.1. Testing with Partial Evaluation

Partial evaluation can be used to integrate testing into the development process without additional infrastructure. To see how this can be done remember that, with partial evaluation, the output of any particular node will be a function of all the open inputs in its upstream graph. A developer can set up tests on any node, which can be checked and verified without affecting the rest of the graph. This is illustrated in Figure 4.47.

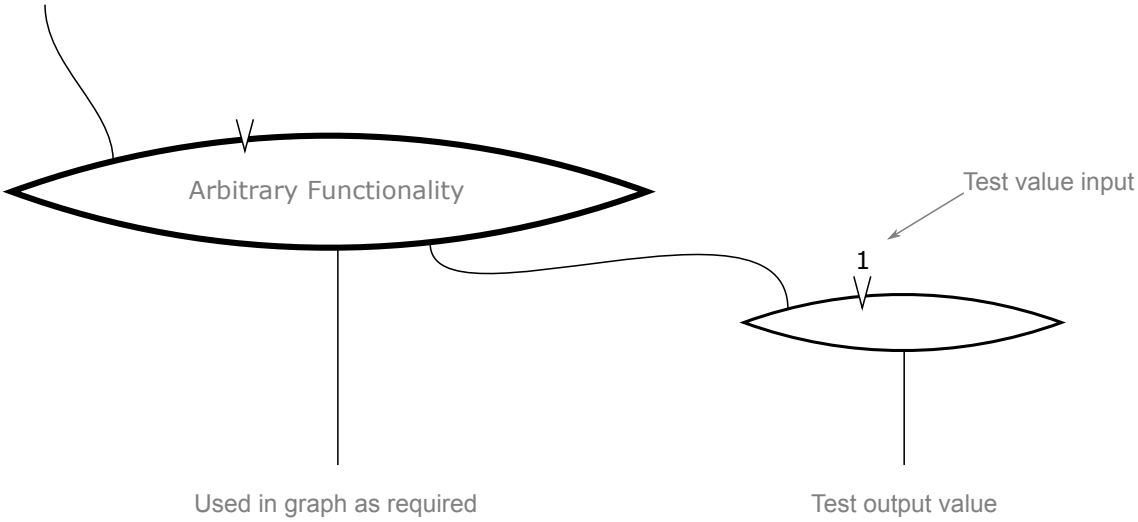

Figure 4.47. Verifying a node's functionality. Without interrupting the flow of the graph, a node can be tested to verify it generates the expected outputs.

This can be extended to obtain a verification that the result is as expected by producing an output value of 'true' or 'false' (two ways of doing this are shown in Figure 4.48).

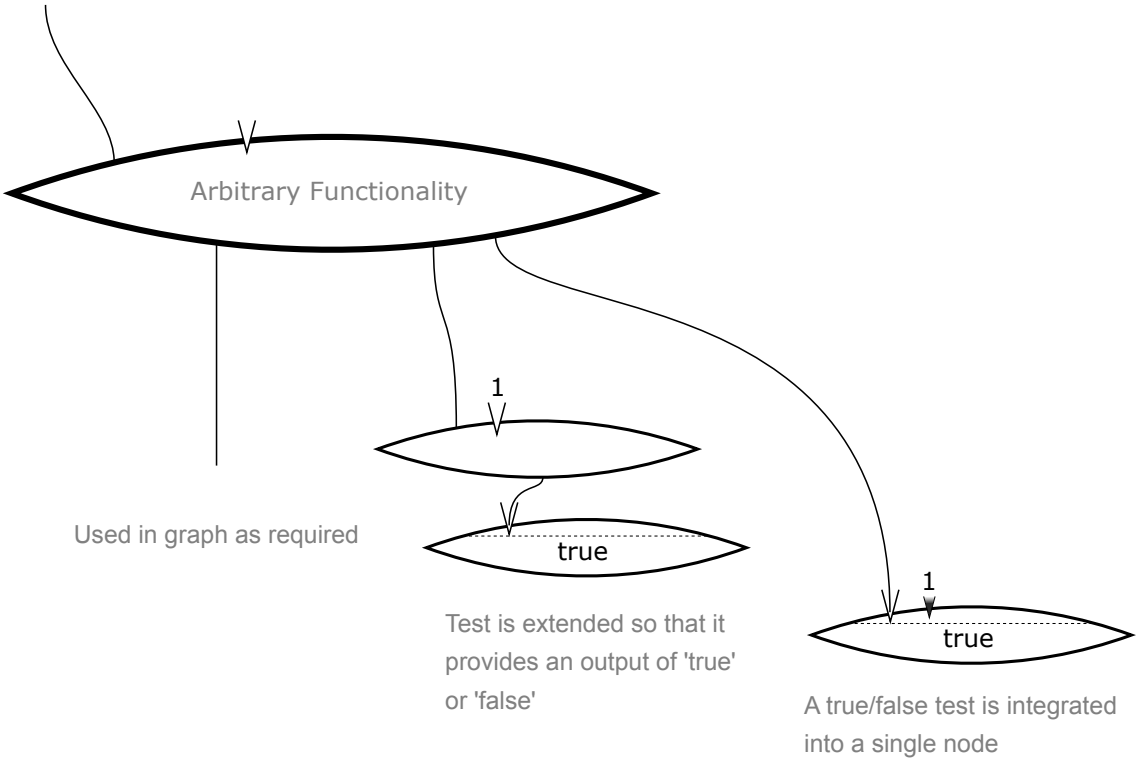

Figure 4.48. Obtaining a true/false test result. The test can be extended with another node to generate an output of 'true' or 'false', depending on whether the test generates the correct result. This can be done by adding an additional node, or the test can be integrated into a single node.

It can be extended further to obtain a summary of multiple tests, or global results of all tests, telling the developer whether all tests in a graph have passed, and if not then which ones have failed. This is illustrated in Figure 4.49, in which partially-displayed nodes on the left are part of the main node graph. We can construct any number of individual tests, and summaries of them as required. The 'Test Summary' node in this example can be used to provide a summary of the results and list any tests that have failed across the whole graph.

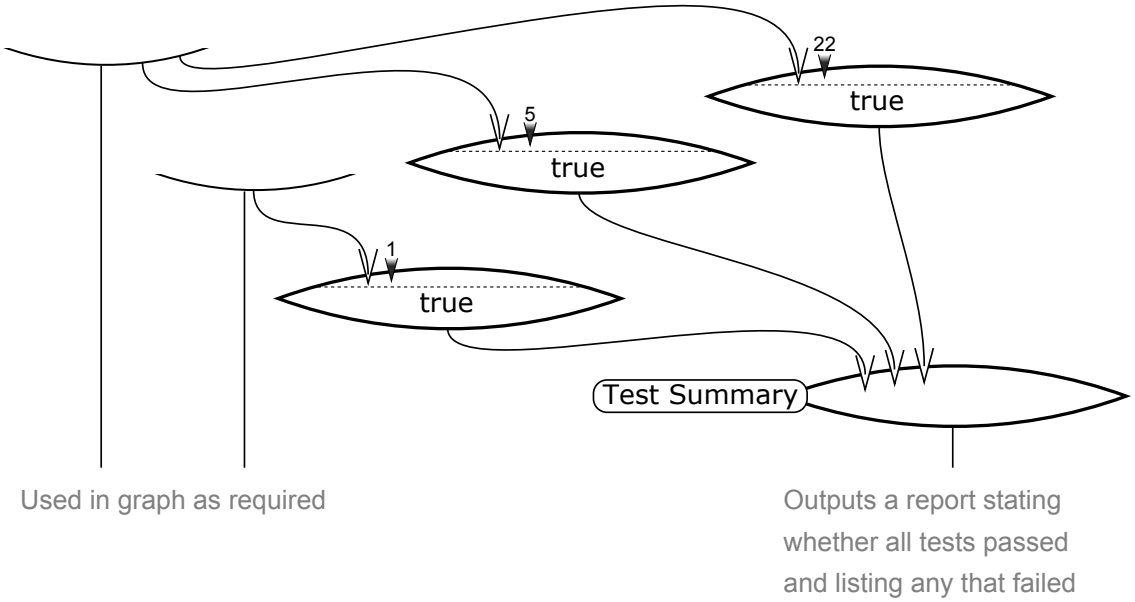

Figure 4.49. Summary test results. A summary node can be created to aggregate the results of all test nodes, outputting a report stating which of the tests have failed, and providing the programmer with single point of reference to determine whether the program is performing correctly.

4.17.2. Testing with Expected Inputs

The 'expected inputs' feature, described in Section 4.9, improves the ability to build tests by making it possible to build a test in the absence of the function being tested, and to reuse the test on multiple different functions to compare results. Imagine we want a node whose job is to double a number. Without yet having the node we want to test, we are going to write a test for it. Using the 'expected inputs' feature, we can set up the test as shown in Figure 4.50.

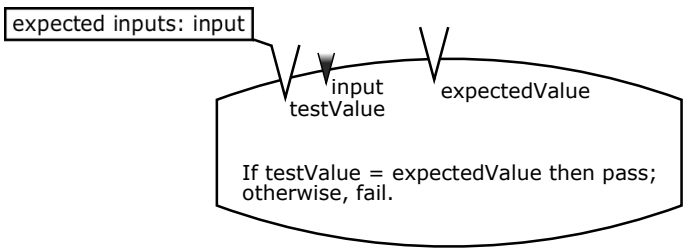

Figure 4.50. Tests using the 'expected inputs' feature. With expected inputs, a generic test can be created in the absence of the node being tested.

This node includes two root inputs, called 'testValue' and 'expectedValue', and a function which simply outputs 'pass' if they are the same or 'fail' if they are different. We set 'testValue' to have an expected input named 'input'; in response, the node generates 'input'

as a derived input. We can now use 'input' and 'expectedValue' to try corresponding inputs and outputs. If we were to add values to the 'input' and 'expectedValue' inputs, the node would partially evaluate and we would get, as an output, a function which has one input ('testValue'), which in turn provides a 'pass' or 'fail' output when that function is provided (Figure 4.51).

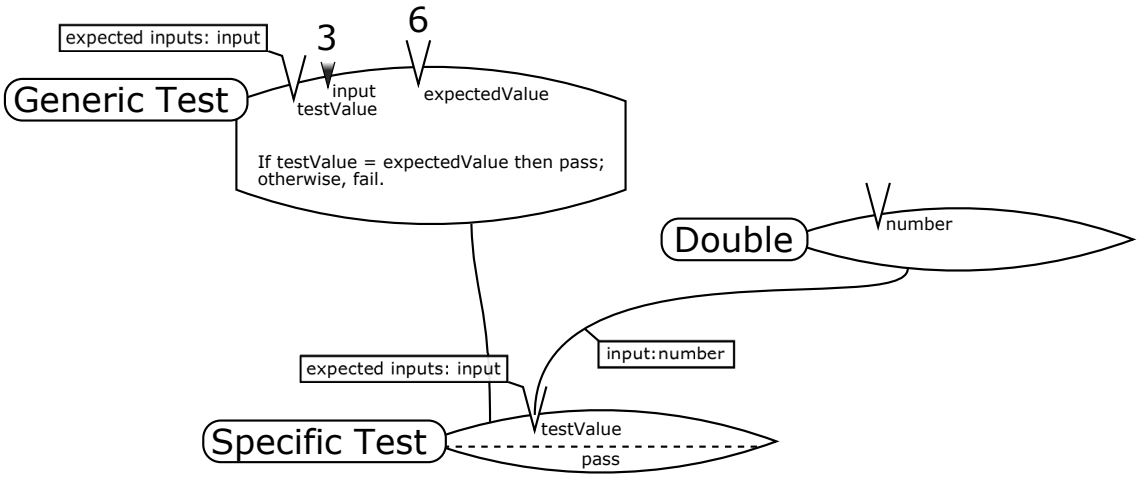

Figure 4.51. Using a generic test node. The 'testValue', 'input' and 'expectedValue' inputs can be provided in any order. In this example, the 'input' and 'expectedValue' inputs are provided directly, but the 'testValue' is left open and inherited downstream by the 'Specific Test' node, and when connected to the 'Double' node, it provides 'pass' as an output.

In this example, the 'Generic Test' node has been provided with test input and output values of '3' and '6'. It outputs a partially evaluated function, which is connected to the 'Specific Test' node. The 'Specific Test' node has the node being tested (the 'Double' node) connected to it with the appropriate input nominated, and it outputs 'pass' as result.

An alternative configuration would allow us to reuse the generic test to verify more than one combination of input and output values. In Figure 4.52, the node to be tested could be connected to the 'Generic Test' node when ready, and the results for both sets of test values would be calculated; or they could be connected to the two downstream nodes separately.

In Figure 4.53, expected values are applied instead to the 'Test' node, and its output is used to verify that two separate nodes with the same functionality, 'Double1' and 'Double2', produce the correct result. In this example, both are shown to pass the test.

4.17.3. Testing with Independent Iteration

Tables, combined with the 'dimensional overloading' feature used for independent iteration (described in Section 4.11.1) can improve the process further. With dimensional overloading, rather than using a separate node to test each pair of input and output values, we can use a table to test every set together.

In Figure 4.54 we have an alternative node graph, this time designed to accept pairs of inputs and expected results on the same input. In this graph, the 'Comparison' node has only two root inputs: 'testFn' and 'expected'. It compares the two values and provides the output value 'true' if they are the same, or 'false' otherwise.

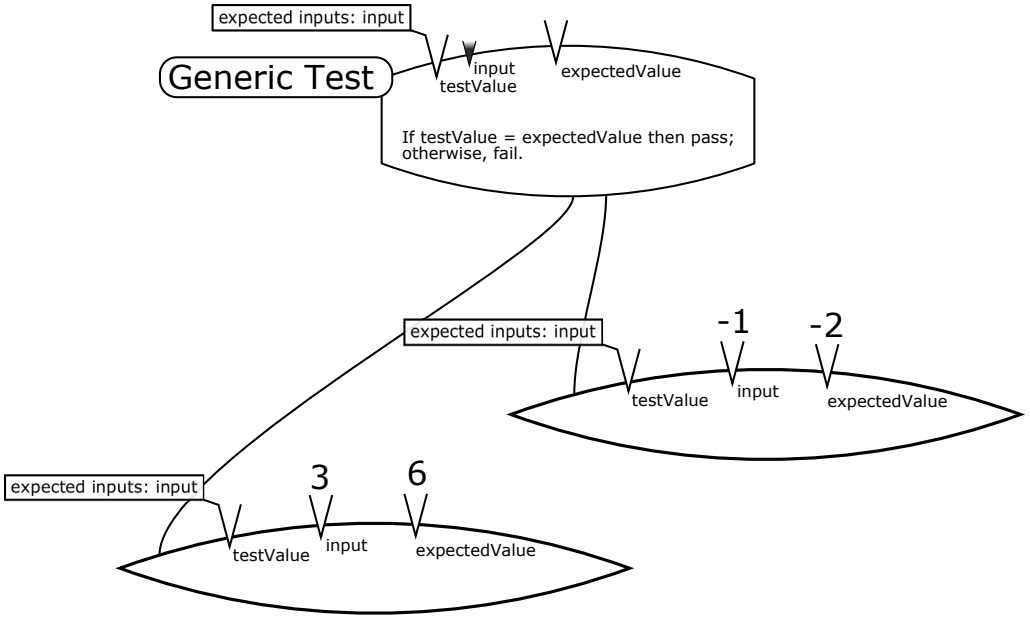

Figure 4.52. Testing multiple values. In another extension of the generic test, in this example all inputs are left open, so that different combinations of input and output values can be tested.

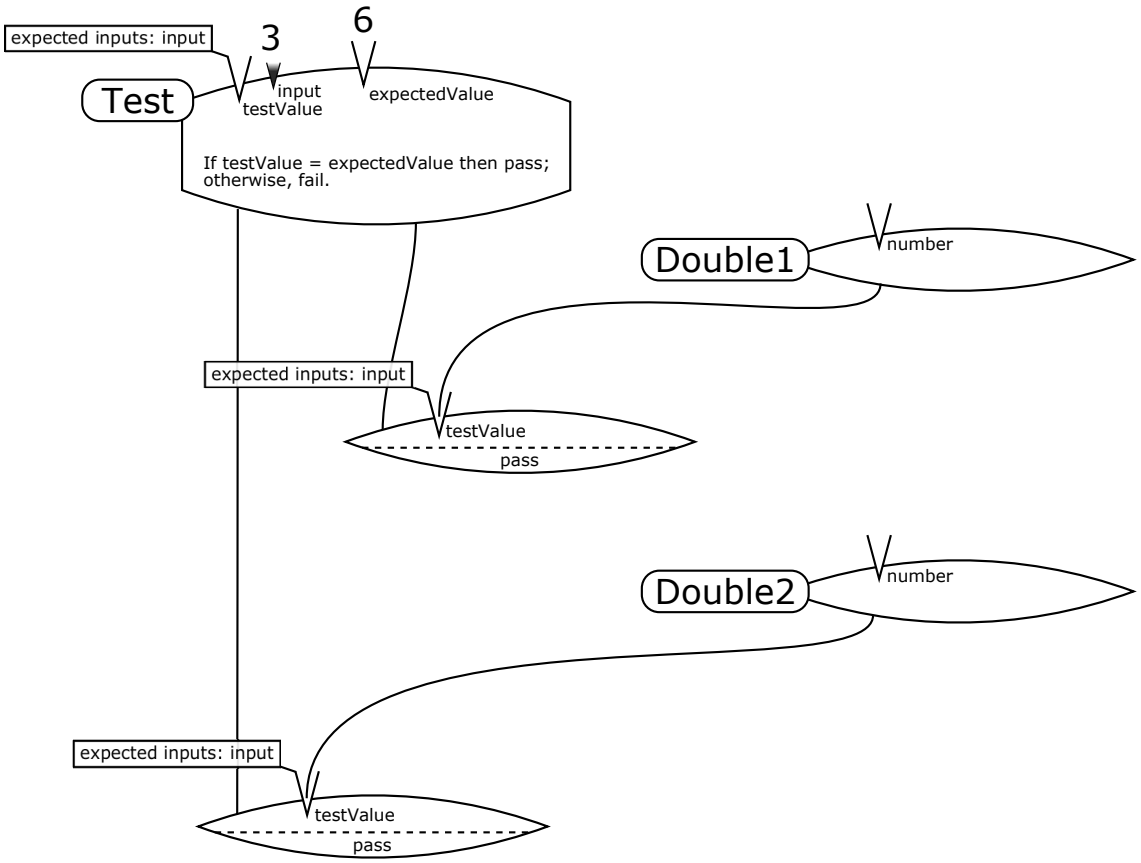

Figure 4.53. Testing multiple nodes. In this example, the same combination of input and expected output values is used to test two different nodes, 'Double1' and 'Double2'.

To make it into a test, the 'testFn' input has its 'expected inputs' parameter set to expect one input, named 'input'. At the top of the graph, we have the InputResult node, whose purpose is only to accept an input and corresponding expected result, in the form {input,result}, and output it unchanged. From its output, separate nodes have the job of extracting the 'input' and 'output' parts of this combined object. These are then applied

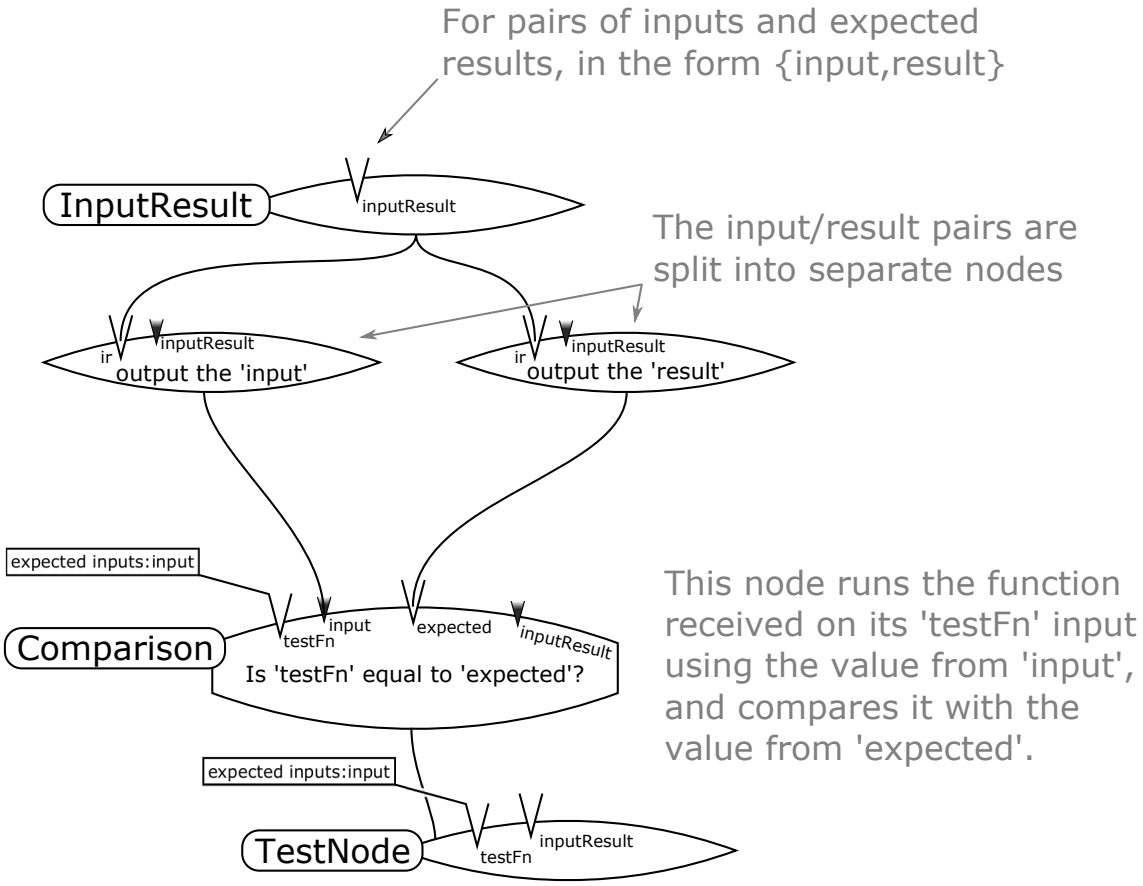

Figure 4.54. A test for use with dimensions. The resulting 'TestNode' node allows the user to apply a list of input/result pairs, together with a function to be tested, and will generate a list of results as an output, corresponding to the list of input/result pairs.

to the 'input' and 'expected' inputs of the 'Comparison' node, which then inherits the 'inputResult' input.

The final output of the graph is a node named 'TestNode', with just two inputs: 'testFn' and 'inputResult'. This is a generic node for testing pairs of inputs and outputs for any other node. Figure 4.55 shows how it can be used.

In this graph, a node whose purpose is to double a number is being tested. It is called 'DoubleANumber'. The TestNode's 'testFn' input is connected to it. The 'testFn' input has an 'expected input'. Because the 'DoubleANumber' node only has one input, its input is nominated implicitly and there is no need to explicitly associate it with the expected input.

This leaves the 'inputResult' input. This is where dimensions come into play. Because the number of dimensions is not specified for the 'inputResult' input, it takes the default value of '0'. The data provided to it is a list (one-dimensional) of input/result pairs, meaning the input is 'overloaded' by one dimension and the node therefore generates a one-dimensional output (another list).

Setting this up need not always be so complicated. The 'TestNode' node is generic enough that it can be reused to test any node that has just one input. Whereas this example shows a test involving just three input/result pairs, it scales up or down with very little

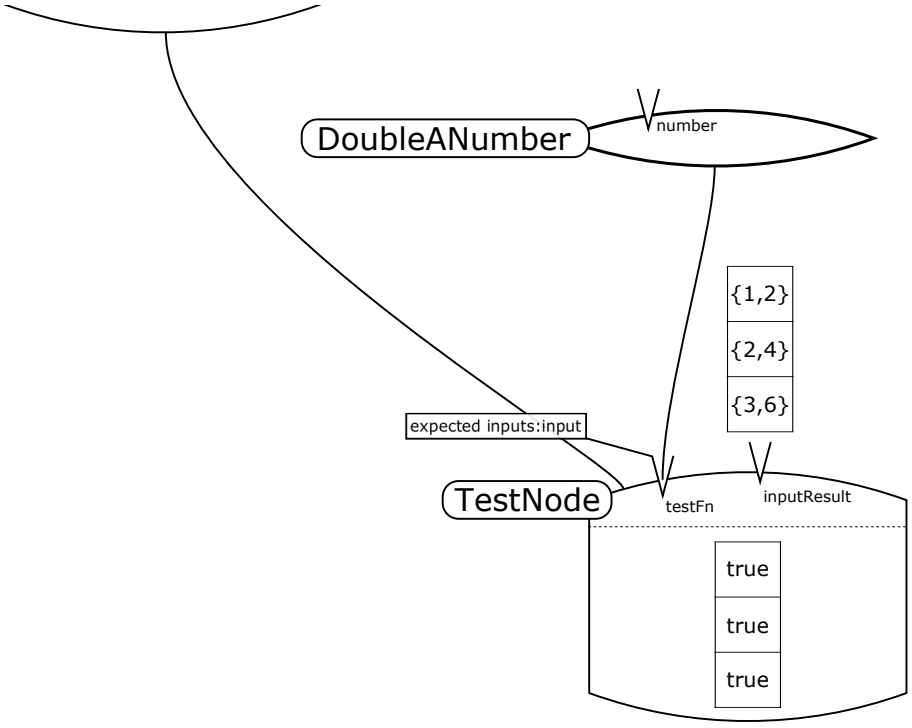

Figure 4.55. Testing with dimensions. In this example, a node whose purpose is to double a number is tested on a list of three pairs of inputs and results. It outputs a list telling us that the test has been successful in all three cases.

extra work, and could equally be used to test any number of pairs, simply by adding or removing items from the list.

4.18. Example Application

To see how a system with some of the features described in this chapter might work, we will use the example of a node whose purpose is to buy or sell a stock based on three pieces of information: whether or not it currently owns the stock, the current estimated value of the stock and the current price of the stock. If it owns the stock and the price is higher than the value, the node will sell it; if it does not own the stock and the price is lower than the value, the node will buy it. Otherwise it does nothing.

Manifested in this example we see three important concepts: state (knowing whether or not the node currently owns the stock), external action (the ability to buy or sell it) and external inputs (the current stock market price of the stock). We could depict this node graph very simply, as shown in Figure 4.56.

In this example, the value is set to a fixed amount, 100 (currencies and other units are excluded). The 'Price' node could work in a number of ways: it could periodically query the stock exchange to fetch the current price and convey that; or it could in some way subscribe to a data feed from the stock exchange and update its value as new prices arrive. While we are building this node graph, the user interface might want to be able to see, interactively, the changes to values taking place within the nodes. In order to achieve that, the user interface must subscribe to the nodes. For as long as the user interface is subscribed to the nodes in this way, the buying and selling action will take place. However,

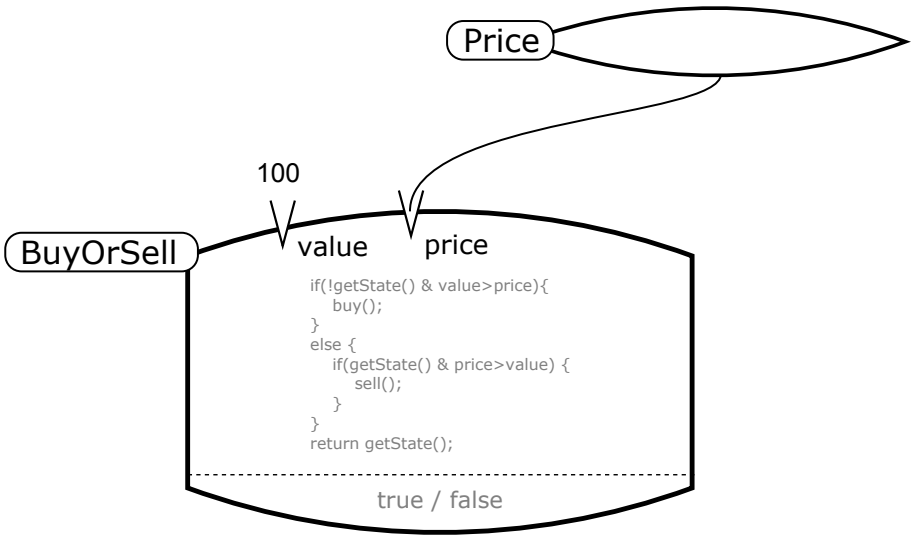

Figure 4.56. Example application. A node to buy or sell a stock based on its valuation, its current price and whether or not we already own it. This example requires us to be able to set or retrieve state for the node. In this case, the state is true if it currently owns the stock or false otherwise.

in order for the action to continue when no longer being viewed by the user interface, we can set the 'BuyOrSell' node to 'on' (as described in Section 5.1.7 — Subscriptions). It will then continue buying and selling the stock indefinitely. If we set the 'BuyOrSell' node to 'off' and the user interface stops viewing the node then, when the next data update notification arrives from the 'Price' node, the 'BuyOrSell' will unsubscribe from it and could potentially be saved on disk until it is next queried for a value.

If, on the other hand, we wish to pause this node to prevent it from acting even while performing its computations and displaying outputs, we could add another input, based on which it will perform its actions or not. We could call this new input 'active' (distinct from the 'on' setting, which determines whether the node computes at all and remains subscribed to its inputs). This new version is shown in Figure 4.57.

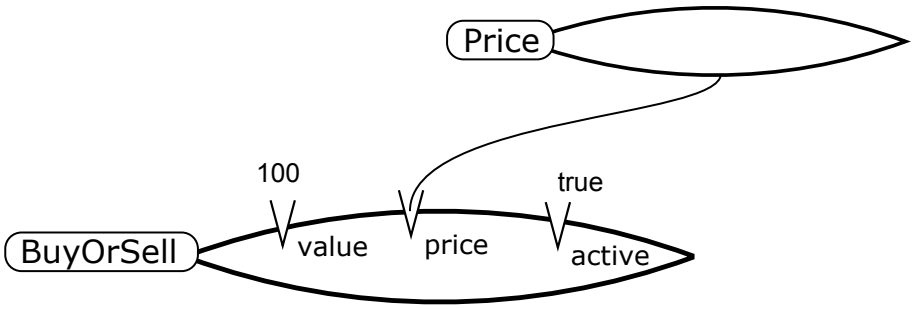

Figure 4.57. Example application — with an input to make it active or inactive. The node will continue to compute its output, but the 'active' flag is used to determine whether the node will actually perform its external actions.

We are now going to add a distributed component to the graph. Imagine that instead of using a fixed valuation for the stock we are going to engage an external provider to provide us with a current value. The external provider maintains their own estimate of the current value of the stock based on three inputs: the current central bank interest rate, which is updated monthly; a currency exchange rate, which (like the stock price) is managed by the node — queried periodically or by subscribing to receive a data feed as

prices are updated; and the price of some other commodity — say, oil. We are going to assume that the oil price is used to calculate a cap on the valuation of our stock. Say, for example, it is decided that our stock is never worth more than the price of a barrel of Brent Crude. This means that a third input, the price of Brent Crude, is needed, but does not always affect the price. The graph and function for this node might look something like that shown in Figure 4.58.

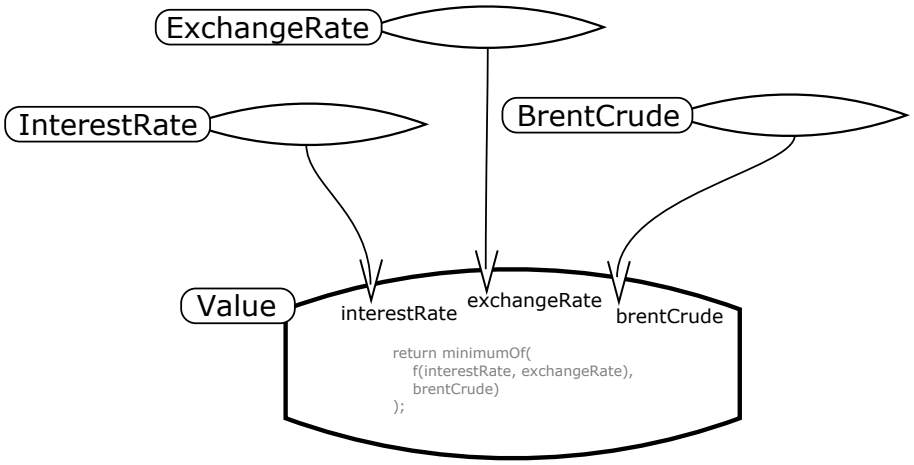

Figure 4.58. Example application — stock valuation. The valuation, provided by a separate supplier, is a function of two inputs: the interest rate and a currency exchange rate; and is capped by the the price of a commodity, in this case a barrel of Brent Crude.

This node, owned by a separate organisation, could be hosted in an entirely different location. Nevertheless, our 'BuyOrSell' node needs to connect and subscribe to it, and on doing so will start to receive data update notifications and subsequently new data whenever new inputs to the valuation node are received. Since the 'brentCrude' input is used to cap the price it only causes a change to the overall valuation if it drops below the value otherwise calculated which, most of the time, might not happen. Whenever the Brent Crude price is updated, an update notification would first be sent through the graph. Subsequently, the 'Value' node would recalculate its value and find that its output was unchanged. Rather than proceeding to send this identical value and allowing downstream nodes to recalculate their values, it could instead send a cancellation notification, notifying downstream nodes that the previously notified update had been withdrawn and allowing them to mark their previous values as being current again. This would enable both the 'Value' node and the 'BuyOrSell' node to avoid new computations, and would avoid having to send the same data again over the network. The overall graph would be of the form shown in Figure 4.59.

Although this provides an accurate depiction of the whole graph, this is not what any one user of the system would see. The owner of the 'Value' node would have no need to see the workings or inputs of the 'BuyOrSell' node, and the owner of the 'BuyOrSell' node would have no need to see the inner workings or connected inputs of the 'Value' node, other than to be aware of its remaining inputs after being partially evaluated. As a result, these two users would have different restricted views of the same graph, with only the directly-connected nodes being visible to the other user. The sub-graphs visible to the two users are shown in Figure 4.60.

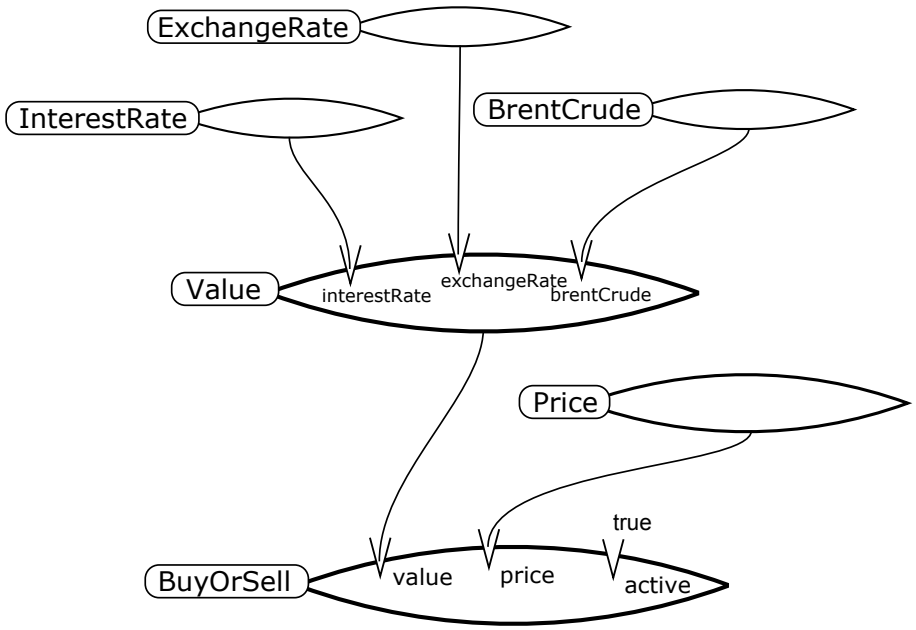

Figure 4.59. Example application — the complete graph for buying and selling a stock based on a valuation provided by a separate provider.

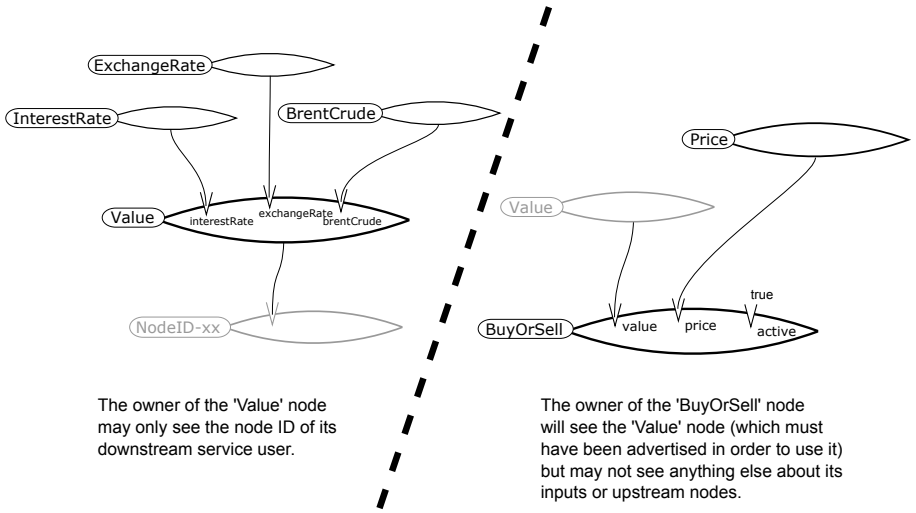

Figure 4.60. Example application — two restricted views of the same graph. While the owner of the 'BuyOrSell' node can see the advertised name and description of the node it is connecting to, the owner of the 'Value' node may see nothing more than the node ID and address of the downstream node connecting to it.

Having a distributed system, we can now see instances of the issues of synchronisation and node preferences discussed in Sections 4.14 (Notifications and Time-Stamps) and 4.16 (Synchronisation). If new data arrived from one input, while other new data was working its way through the node graph, we would probably not want the node to execute a trade based on out-of-date information. The notification system reduces the chance of this happening by ensuring that the decision-making node is notified of the updated data promptly. The principle of node autonomy could be used to ensure the computations (and therefore trades) are aborted when update notifications arrive rather than proceeding with a trading decision that could be out of date.

This graph also demonstrates the issue of imperfectly synchronised clocks over a dis-

tributed system. Imagine we want to record a time-series of valuations, prices and stock trading decisions. This would require a new node, dependent on the 'Value' node, the 'Price' node and the 'BuyOrSell' node. It would need a type of subscription in which every new value was delivered, regardless of whether it had been superseded. The node would tell us how effectively the 'BuyOrSell' node was reacting to those changes (and whether latency was causing it to miss potential trading decisions). The time-stamps would be needed to order data updates, which may not arrive in strict chronological order. However, with time-stamps being generated on different (imperfectly synchronised) machines, an approach would be needed to decide how to order those updates sufficiently close to each other to fall within the clock synchronisation uncertainties. Some approaches, discussed in Section 4.16, have been proposed by others, but the wider synchronisation issue is left for further work.

4.19. Further Work

Because the system is designed as a coordination system, indifferent to the implementation of the nodes within it, it means any computational capability can be integrated within a node. However, there are improvements to be gained by moving functionality from node-level to graph-level and providing a graph-level interface for building it. Much of the work of improving the system, now and in the future, has and will revolve around selecting features to be integrated at graph-level. Further work is needed in a few areas in particular, including:

- data structures — the ability to create, process and manipulate object oriented-like data, relational database-like data and graph data;
- tables — a more fully developed handling of multi-dimensional tables ('hyperrectangles');
- grouping — developing the possible uses for groupings of nodes, including to define a 'collection' (an unordered list), to visually collapse a part of the graph, to define compound objects, as groupings to define distinct 'models' or access rights, and to define graph segments to be used as units of data;
- optimisation — efficiency has long been a problem, particularly with iterations and data structures, and will require ongoing work to resolve.

The possible future directions of future work are discussed more fully in Chapter 7.

4.20. Summary

This chapter describes the possible structure and design of a unified global dataflow coordination system. Applying the principles of software engineering (discussed in Section 3.4), it assembles the best of the ideas proposed in the literature (discussed in Chapter 2) and extends them, in particular using partial evaluation (Section 4.8) and dimensionality (Section 4.10) to represent dependent and independent iteration (Section 4.11). Section

4.12 shows how, together with a new feature referred to as ‘expected inputs’ (Section 4.9), a generalised iteration node can be created providing functionality similar to a ‘for’ loop.

The next chapter describes how these concepts have been implemented as far as needed to be able to construct three test algorithms: calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort.

Chapter 5

Implementation

Selected functionality described in the previous chapter has been implemented in order to demonstrate that it can work. It has been used to build, in dataflow, three test algorithms: calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort. The features demonstrated include partial evaluation, as described in Section 4.8, independent iteration as described in Section 4.11.1 and dependent iteration as described in Section 4.11.2. These have been used not only to demonstrate that the algorithms produce accurate results, but also that the system can parallelise the computation by making use of all available processors.

5.1. Feature Implementation

As described in the previous chapter, the definition comprises intrinsic features, which must be consistent across all nodes (such as the protocol by which nodes communicate with each other and the set of required node behaviours), and interchangeable components (such as the internal node implementation), any of which could be substituted and alternatives used concurrently within the same system. The implementations of both are intended to be demonstrative rather than, at this stage, a definite proposal.

Some features that were unnecessary for the purpose of testing were omitted. These include the user interface and a full implementation of the 'expected inputs' feature described in Section 4.9. The user interface is an interchangeable component that would certainly improve usability but does not contribute to the core capability of the system. A partial implementation of the 'expected inputs' feature was included, but was not fully integrated with the iteration functionality. These, along with a range of other suggested features, are deferred for further work and described more in Chapter 7.

5.1.1. Technology Choices

The technology choices for the implementation are driven by a number of key considerations.

- **Platform Flexibility** — full platform independence is not necessarily needed at

this stage, but an ability to run on multiple platforms is beneficial.

- **Parallelisability** — since this is a key benefit of the system, an ability to parallelise code is important.
- **Ability to parse and recompile functions** — functions need to be parsed into abstract syntax tree form, passed around, then recompiled and executed.
- **Ability to run code in a web browser** — since it is envisaged that a user would interact with the system via a web browser, it is important that it should be possible to construct a dataflow graph and run functions in the browser as well as on a server.

This last constraint, needing the ability to run code in a web browser, is the most restrictive, since browsers only speak JavaScript[1]. It is not an absolute constraint, since it would be possible to write a separate implementation for the browser, but this creates extra work that can be avoided.

For this reason, JavaScript has been chosen as the language in which the system as a whole is written. It is written using the server-side JavaScript platform NodeJS, which can be run in Windows, Mac or Linux, providing a degree of platform flexibility. NodeJS provides the ability to find out how many cores are available and to create separate processes that execute in parallel, providing in addition a level of isolation between processes.

JavaScript has two additional advantages. The first is that functions are a first order data type and can therefore be treated in the same way as any other data type: they can be used as variables, passed into functions as arguments and used as function return values. The second advantage of JavaScript is its use of prototypal inheritance. Prototypal inheritance allows node object properties to be attached to a single prototype rather than to each instance of it, enabling object instances to have a very low memory footprint. In a system that may require many thousands of nodes, this efficient handling of object instances is a significant advantage.

This implementation was tested on Windows 7. In NodeJS, the programmer is to a large extent insulated from the differences between operating systems, and the same code could be run on any other operating system on which NodeJS could be installed with only very few changes. NodeJS uses more recent versions of JavaScript than can be relied upon in the browser, and the code utilises convenient features that are relatively new in JavaScript but would be unavailable in most browsers. However, with the help of web development tools such as *Browserify*[2] and *Modernizr*[3], the code could be adapted to run in any modern browser. The system is controlled through an interactive text-based API (as yet, there is no graphical user interface for it).

[1] The section of the W3C HTML5 specification on scripting (Section 4.11 - `https://www.w3.org/TR/2014/REC-html5-20141028/scripting-1.html`) specifies Javascript as the default scripting language, and the only one whose 'type' attribute browsers are required to recognise.

[2] `http://browserify.org/`

[3] `https://modernizr.com/`

5.1.2. Function Execution

The underlying language has capabilities that we would not want untrusted user-contributed code to have. Section 4.13 describes how untrusted code could be made safe to run and, using a similar technique, how it could be made possible for untrusted users to contribute whole new languages to the system. This uses a structure in which a language parser, defining a new (safe) language, parses function code into the form of an abstract syntax tree for storage and transmission and a compiler compiles it from that into the underlying language for execution.

The implementation demonstrates this technique, using the open source parser *Esprima*[4] to convert node functions written in JavaScript to an abstract syntax tree form (in the 'ESTree Spec' format[5]), and using the open source compiler *Escodegen*[6] to compile code into the system language (JavaScript) from its abstract syntax tree form. Although the parser leaves the unsafe as well as the safe features of JavaScript intact, it does demonstrate the technique and leaves open the future possibility to edit it or write an alternative to add safety. It is convenient for testing that the function language is the same as the system language, but there is no other reason why the two should be the same.

5.1.3. Partial Evaluation

In its simplest implementation, partial evaluation (or something that looks like it) could be achieved by wrapping one function in another. Say we have the function, 'f', shown in Snippet 5.1, and want to partially evaluate it with the value $a = 1$.

Snippet 5.1

```
function f(a,b,c){
  ...
}
```

The simplest way to do this would be to wrap the function 'f' in another function, 'g', as shown in Snippet 5.2. The problem with this approach is that the partially evaluated function 'g' is 'bigger' (and requires more work to evaluate) than the original function 'f'.

Snippet 5.2

```
function g(b,c){
  function f(1,b,c);
}
```

Since partial evaluation reduces the scope of the function's possible inputs and outputs, it has the potential, in the right implementation, to reduce the amount of computation needed to obtain the result. The example in Snippet 5.2 has the opposite effect. In order to achieve the appropriate reduction in work, we need to be able to manipulate function 'f'. For this purpose, it is useful that the function is parsed and converted into abstract syntax tree form, as described in Section 5.1.2. In its abstract syntax tree form, it is possible to edit the function directly.

[4] Created by Ariya Hidayat and available online at `http://esprima.org/`
[5] Described at `https://github.com/estree/estree`
[6] Created by Yusuke Suzuki and other contributors, and available online at `https://github.com/estools/escodegen`

In a simple version of this approach, a partially evaluated version of a function can be created by inserting a variable declaration for the provided value at the start of the function, as shown in Snippet 5.3. This leaves open the possibility of more elaborate manipulations and optimisations of the function including, for example, executing branches of the abstract syntax tree for which all variables are available, trimming branches that are no longer applicable, and the application of existing graph reduction techniques taken from Functional Programming, where relevant.

Snippet 5.3

```
function f(b,c){
  var a=1;
  ...
}
```

5.1.4. Data Transmission

Although the implementation runs on one machine, it is designed to test functionality that could be distributed over multiple machines. Two techniques are used in the sending of data to better simulate the behaviour of a distributed system.

The first is to make communication between nodes asynchronous, meaning a message can be sent to another node without having to wait for that message to be received before continuing. The second is to convert data to a format that can be transmitted over a network — in this implementation, the JSON[7] format is used. This incurs a computational overhead but has the advantage that, being a text-based format, it is immutable and therefore safe from being treated, inadvertently, as shared state between nodes. JSON does not allow for every type of data that might be needed in dataflow, making it necessary to encode and specify types within the string. Examples of two such JSON strings are shown in Snippet 5.4.

Snippet 5.4

```
'{"type":"string","data":"Test String"}'
'{"type":"date","data":"1977-08-15T02:16:00.000Z"}'
```

Partially evaluated functions require a more complex format. The first step is to convert the abstract syntax tree (AST) form into JSON. Partially evaluated functions then require additional data, including input parameters, input values, iteration parameters and connection parameters, all of which must be conveyed together with the abstract syntax tree when a partially evaluated function is transmitted from one node to another.

In addition, when a partially evaluated function is received by a node which is in turn partially evaluated, the partially evaluated function object conveyed through the graph as a result must encapsulate both the root function and the received function. To accommodate this, a partially evaluated function is encoded as an object which, at its top level, contains a pair of recursive objects — one containing the input values and one containing the function statements (the AST). The partially evaluated function ('NodeFunction') object has, at the top level, the shape shown in Snippet 5.5

[7] JavaScript Object Notation, defined at `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

Snippet 5.5

```
NodeFunction {
  args, // Object of the type `Args'
  fns   // Object of the type `Fns'
}
```

The 'Args' object has the shape shown in Snippet 5.6, with the 'subs' property containing an additional 'Args' object for each applicable top level argument.

Snippet 5.6

```
Args {
  origin,          // A string specifying the node at which
     this function was first defined.
  arr,           // An ordered list of strings specifying
     the arguments of the underlying function.
  dimensionsObj,       // For each argument for which a `
     dimensions' number is set, this provides the number.
  expectedInputs,      // For each argument for which `
     expectedInputs' are set, this provides an array of
     strings, which convey the inputs that any function
     received will be expected to have.
  nominatedInputs,     // For each input which expects
     inputs and that has received a value, this describes
     which of the received inputs corresponds to each of
     the expected inputs.
  iterationParameters,  // If this function is set to
     iterate, this contains the iteration parameters.
  subs           // An object containing an Args object for
     each applicable member of `arr'.
}
```

The 'Fns' object, shown in Snippet 5.7, has the same basic form, containing top level data and a 'subs' property, which contains another 'Fns' object for each applicable top level argument.

Snippet 5.7

```
Fns {
  main, // The AST for this node's function.
  val,  // In some cases, where function has already been
     evaluated, this stores the resulting value.
  subs  // This stores a `Fns' object for each input of the
      function for which a function has been received.
}
```

For transmission, the whole 'NodeFunction' object is converted to text, and all data within it converted into data objects of the shape described in Snippet 5.4. When received by a node, the whole 'NodeFunction' object is reassembled from JSON format and interrogated to obtain a list of its inputs.

5.1.5. Input Unification

As described in Section 4.8, where the same input is received via multiple routes, or is contained more than once in the same partially evaluated function, the multiple instances

of it are ‘unified’; meaning that the node treats them as one. When the node is queried for a list of its inputs, it will only list each input once; and when a value is received for an input, it is applied to all instances of that input. Figure 5.1 illustrates this process.

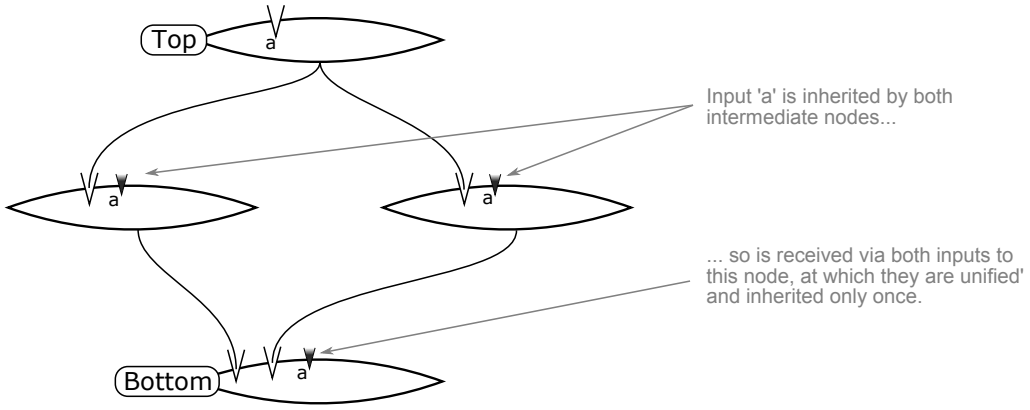

Figure 5.1. Unification of inputs. An input that is inherited at a downstream node via more than one route is inherited only once.

In this example, the node labelled ‘Top’ has one input, which is inherited by two separate nodes, whose outputs are then connected to separate inputs of the same downstream node (labelled ‘Bottom’). Despite input ‘a’ having arrived via two routes, it is inherited only once.

5.1.6. Parallelisation

NodeJS reports the number of processors available, so it is possible to adapt the number of processes used to the number of processors available. The main process is used for user interaction, node-to-node communication, and all node operations except function execution, leaving the remaining processors available solely for function execution.

Allocation of processes to processors is handled automatically in NodeJS, so although processes cannot be assigned to a particular processor explicitly, the automatic handling is effective enough that they are usually assigned appropriately. In NodeJS, additional processes are referred to as ‘child processes’. Communication between the main process and child processes takes place through ‘messages’. The main process can send a message to any child process; child processes can only send messages to the main process. In all processes, a function is assigned to a listener, which is triggered when a message is received.

Each child process waits, idle, until a message arrives. A message, when it arrives, contains a function, which is then executed and a message sent back to the main process containing the result or an error report.

The main process maintains a queue of functions to be executed. As soon as it receives a result from a child process, it sends that process the next function in the queue. The main process also schedules a time-out for each function, terminating the child process if that time-out is reached without receiving a response. For testing purposes, a time-out period of two seconds was used, but this is a matter for node autonomy and could be varied to suit the circumstances.

Because the child processes can communicate only through explicit messages, the main process is not vulnerable to malicious activity by node functions. However, child functions are vulnerable to their own code being modified by malicious node functions. This vulnerability would have to be resolved before being exposed to untrusted users.

As discussed in Section 5.1.4, node functions can contain other node functions, meaning that any particular node function may be parallelisable, and multiple child processes can be used in execution of a single node. The structure of a graph provides some level of parallelisability; in testing it is easy to build graphs that allow parallelisation of the order of around ten. Dimensional overloading, however, provides a higher degree of parallelisation. Each dimension by which a node's inputs are overloaded allows that node to parallelise its computation by an additional factor equal to the length of that dimension. For example, if an input is dimensionally overloaded by three dimensions, each of length 10, the parallelisability of the computation is multiplied by a factor of 10^3.

In tests using dimensionality, it is easy to overload a node by an order of ten or more dimensions. If each of ten overloading dimensions had a length of 10, this would multiply parallelisability by a factor of 10^{10}. This would be multiplied again by the degree of parallelisability inherent in the graph. We quickly find ourselves able to build graphs that allow parallelisation by more than the number of processors likely to be available.

This means that in many cases the availability of processors, rather than the parallelisability of the program, will be the limiting factor. This could lead to long queues of functions waiting for execution, with an associated memory overhead. It seems likely that for a production dataflow system to be viable, it would need to optimise for these limits dynamically. It is easy to envisage solutions, but for the purpose of testing this problem is left unresolved. This enormous potential parallelisability does, however, illustrate how powerful this approach to parallelisation can be, and the possible benefits to be gained in moving to machines that have a larger number of smaller processors, rather than a smaller number of larger ones.

5.1.7. Subscriptions

Nodes have the ability to subscribe to other nodes, as described in Section 4.6. The intention is that such functionality should also enable external entities or components to subscribe to nodes. A user interface could be one such component, which would subscribe to the list of nodes it is viewing at any point in time.

When a node receives a subscription, it registers the subscribing node in an internal list of its subscribers. Whenever the node receives an update notification or its content is updated, it will immediately notify its subscribers, and later deliver its updated value when it becomes available.

When a node accepts a subscription it agrees, in effect, a contract to remain up-to-date and to provide update notifications and data updates to the subscriber. In order to be able to fulfil this agreement, it needs to know that its own upstream nodes are up-to-date, so must immediately subscribe to them. In this way, subscriptions escalate from the original

subscribing node to all nodes in its upstream graph.

When a node is subscribed to its upstream nodes it means that, subject to latency, it can be sure that its value has not been superseded. This remains the case until the next update notification arrives, at which point the node marks itself as out-of-date and awaits new data before it can be marked as up-to-date again.

In order to make best use of knowledge about their current state, nodes take a lazy approach to unsubscribing. When the last subscriber to a node unsubscribes, if the node is up-to-date at that moment, it remains subscribed to its upstream nodes. This allows it to maintain knowledge about whether its value is current until another update notification arrives. If it unsubscribed at this point, it would no longer receive notification updates, and would have no way to know whether it was up-to-date. Remaining subscribed during this time means that, if requested, it can provide a value without having to perform a calculation or request new data from upstream nodes.

As soon as an update notification arrives, however, knowing there are no subscribers, there is little point in computing a new value, or in fetching the new value from the upstream node. Instead, it can record that it is out-of-date and immediately unsubscribe from its upstream nodes. Those nodes, in turn, if they have no other subscribers, will remain subscribed to their upstream nodes until they next receive a notification, at which point they too will unsubscribe. In this way, whereas subscriptions escalate immediately up the graph, unsubscriptions move only one step for every update notification that takes place. In a globally distributed node graph, the system could accumulate a large number of infrequently used nodes. As described in Section 4.6, this feature enables unused segments of the graph to 'switch off'.

Subscriptions, like many aspects of the system, can also be a matter for node autonomy. If a node owner considers it important that a node should provide rapid responses, and is prepared to pay the resource overhead in keeping it subscribed, a node can be marked as 'on'. In this case, it will remain subscribed to its upstream graph regardless of whether it has any subscribers. In graphs created for testing, the 'on' setting was applied to the nodes at the bottom of every graph, causing the graph to remain active during testing.

5.1.8. Iteration and Dimensionality

Iteration of both types — independent and dependent — was implemented as described in Section 4.11. Tables and dimensionality (described in Section 4.10) were used for independent iteration; and a subset of the expected inputs feature (Section 4.9) used to define the termination condition in dependent iteration (Section 4.11.2). A full implementation of the 'expected inputs' feature is reserved for further work (See Section 7.3.3).

5.1.9. Notifications and Time-Stamps

Notifications and time-stamps are described in Section 4.14. An object containing an identifier and a time-stamp is used to identify each action through the API. This object is transmitted with every update notification, data update and function execution that

takes place. The API is a useful source for the identifier because it runs on a single processor, in series, meaning that each action can be uniquely identified by its identifier and time-stamp.

5.1.10. Subscription Types and Synchronisation

Section 4.15 discusses potential responses to high frequency updates. The implementation includes just one type of subscription which uses an assumed set of priorities. Nodes only produce outputs that are not known to be out-of-date: a computation that is under-way when a superseding update notification arrives is cancelled, and if notifications or data arrive that are known to be out-of-date they are ignored. Since the implementation runs on a single machine, all time-stamps use the same clock and the problems of synchronisation described in Section 4.16 do not occur.

5.2. Code Structure

The implementation has six key components:

- The node definition;
- The node registry;
- The communication manager;
- The computation manager;
- The execution processes;
- The API.

NodeJS has a main process, and can create additional 'child processes' on demand. All components run in the main process with the exception of the execution processes, which run in child processes created by the computation manager. The computation manager queries the operating system to find out how many processors are available and, after allowing one processor for the main process, creates a child process for each additional processor (with a minimum of one). The six components are discussed in more detail below.

5.2.1. The Node Definition

The behaviour of a node is determined by an object definition, instantiated once for each node in the system. As discussed in Section 5.1.1, one of the advantages of JavaScript is its efficient prototypal inheritance model, which allows node behaviour to be stored in the prototype of an object, rather than in its instances. The reduces the memory footprint of a node and means that only the information that varies from one node to another — its content, inputs, connections and meta-data — need to be stored in each instance.

Although NodeJS does not directly report the memory footprint of an object, tests observing the overall memory footprint showed that this technique can use a fraction of

the memory of the object-oriented model and enables correspondingly more nodes to be created. Nodes in the test system use approximately 6KB each, enabling around 167,000 nodes per Gigabyte of available memory, depending also on the data load of each node. The node definition includes the functionality required to:

- accept requests from and report results to the API;
- store content and meta-data;
- parse functions to establish the inputs;
- send data to and receive data from other nodes;
- partially evaluate functions;
- handle iterations;
- handle subscriptions.

5.2.2. The Node Registry

The node registry maintains a list of the nodes currently in existence. Each node has a unique identifier, used by the API to uniquely identify it.

5.2.3. The Communication Manager

The communication manager mediates communication between nodes, ensuring that it happens asynchronously. This helps to simulate the characteristics of a distributed system, and ensures the correct ordering of events in the graph. The communication manager abstracts away details of communication between nodes, enabling substitution with distributed communication when needed.

5.2.4. The Computation Manager

The computation manager runs on the main process. When a node needs a function to be executed, it sends the function to the computation manager as a computation request. The computation manager maintains a pool of child processes (the 'execution processes') for executing functions. Each computation request is converted into a 'task' and added to a task queue. Items in the queue are then assigned to child processes in request order. When a child process completes a task, it reports the result (or reason for failure) to the computation manager and is added back to the pool of available processes ready for its next task. The computation manager, in turn, reports the result to the requesting node.

Each task has a designated time-out period; with a default of two seconds that can be overridden or disabled by the requesting node. If a task is sent to a child process and the time-out period elapses before it returns a result, the computation manager aborts the task by terminating the child process on which it is running. After terminating a child process, the computation manager immediately starts a new one to maintain the same number in the pool.

A node can also request that a computation be cancelled. This may happen if, for example, the node receives a new update notification that makes the previously requested computation obsolete. The computation is then removed from the task queue or, if already under way, the child process on which it is running is terminated.

5.2.5. The Execution Processes

The execution is designed to be as lightweight as possible in both its size and its use of resources, keeping resources free for the computations themselves. The execution process receives a function from the computation manager, compiles and runs it, and sends a message back reporting the result.

5.2.6. The API

The API ('Application Programming Interface') exists to make the programmer's task easier, presenting a simpler interface, abstracting away some of the underlying detail. It creates and destroys nodes and simplifies references to them.

Most commands to the node graph are asynchronous. Using asynchronous commands normally requires additional work of the programmer to ensure that each command completes before the next one commences. The API relieves the programmer of this burden by automating it, storing all commands in a queue and executing each task only when the previous task has been completed.

The API also provides a shortcut in connecting nodes. Consider the unconnected graph shown at the top of Figure 5.2 (labelled as its 'Initial state'). We have three nodes, 'A', 'B' and 'C'. If we want to end up in the final state shown at the bottom of the same figure (labelled as 'Step 2'), this requires two steps: the first is to connect input 'a' to node 'B' and wait for Node 'A' to inherit input 'b'; the second is to connect the inherited input 'b' to Node 'C'. The API automates this, allowing inherited inputs to be connected in the same step as the root input from which they are inherited.

The API generates update identifiers, as described in Sections 4.14 and 5.1.9, which are included with every request to an underlying node, and consequently with every update notification and data update exchanged between nodes, allowing events arriving at nodes to be correctly ordered. In total the API has 26 functions, listed below, which are described in more detail in Appendix C.

api Returns a placeholder for referring to and performing actions on a node.

createNode Creates a new node with specified content and parameters.

deleteNodes Deletes the specified nodes.

graphStatus Reports a list of nodes in the graph and the number of tasks in the queue.

listNodes Returns a list of the nodes in the graph.

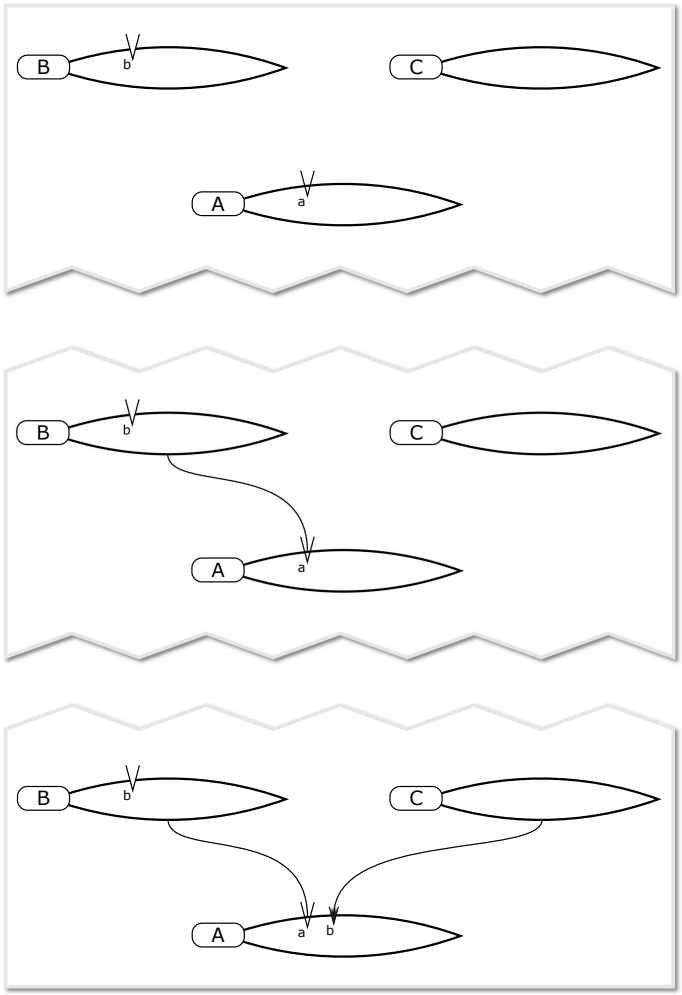

Figure 5.2. Connecting a root and inherited input simultaneously. This would normally require two steps: first connecting the root input, then waiting for it to inherit the derived input before connecting that. The API provides the functionality to do this in one step.

ready Returns a JavaScript 'Promise' (a JavaScript device for handling asynchronous functions), which resolves when the task list has been completed.

instance Creates a new 'ID' node, connected to the specified node. This function is used when a node's partially evaluated output needs to be used more than once in the graph, and is particularly useful for testing.

node Returns a placeholder for the node being acted on.

nominateInput Used to nominate an input for use with the Expected Inputs feature.

nominateInputs Used to nominated more than one input for use with the Expected Inputs feature.

calculateTop A shorthand for calculateWith which uses a simpler syntax but can only be used to provide values for root inputs.

calculateWith Calculates the output of a node using specified values for its open inputs, but without affecting the node or connecting its open inputs.

deleteNode Deletes the specified node.

getConnections Returns a list of inputs and their connections.

getContent Returns the current content of the specified node.

getCurrentInputs Returns the current inputs to a node.

getInputs Updates and returns the list of inputs, including all inherited inputs.

getOpenInputs Returns a list of the inputs that are currently unconnected.

getRootInputs Returns a list of just the root inputs.

getStatus Returns the current status of a node, including its id, name, content, inputs, current value, its 'on' flag (see Section 5.1.7) and whether its output is up-to-date.

getTitle Returns the current name of the specified node.

getValue Calculates the value of the specified node, fetching values of connected nodes in order to do so if necessary.

setConnections Sets connections on a specified node.

setContent Sets the content of the specified node.

setTitle Sets the name of the specified node.

top A shorthand with a simpler syntax for setting only the root connections of a node.

5.3. Test Algorithms

To demonstrate that the implementation can perform computations and, in particular, iterations, it has been used to build three test algorithms: calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort.

The factorial graph is similar to the generalised iteration graph described in Section 4.12, using the split-recombination form to update a counter and the calculated value in each iteration. The graph to generate terms of the Fibonacci sequence is simpler overall but has been chosen to demonstrate that we can make use of more than one previous value to calculate each new value. The merge-sort algorithm is the most complex; as with the factorial algorithm, it uses the split-recombination form, but adds the further complexity of needing sub-iterations within its main outer iteration. The three test algorithms are described below. The code for all three is included in Appendix D and can be found online in the code repository supporting this thesis[8].

[8]The repository can be found at: `https://bitbucket.org/danieljmaxwell/developments-in-dataflow-programming` and the test file that runs the code is at the relative path: *test/mocha/bin/examples.js*

5.3.1. The Factorial Algorithm

The purpose of the factorial algorithm is to provide the factorial of its input as an output. In sequential code, an algorithm to find the factorial of a number might look something like that shown in Snippet 5.8 (written in JavaScript). This function produces the correct value for factorials of non-negative integers up to the limit of integer representation in JavaScript[9].

Snippet 5.8

```
function factorial(input){
  var output = 1;
  for(let i = 1; i <= input; i++){
    output *= i;
  }
  return output;
}
```

In dataflow, the graph takes a similar form to that used for the generalised iteration discussed in Section 4.12. We start with the central ID node on which the iteration will take place, as shown in Figure 5.3.

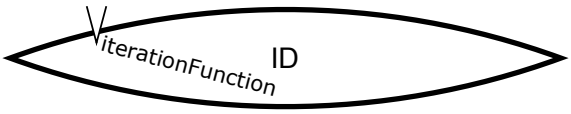

Figure 5.3. The central ID node. We start with a central ID node (one which provides its sole input, unaltered, as an output) with an input named 'iterationFunction'.

We need to connect its input, named 'iterationFunction', to a function defining how each iteration's value will be calculated from the last, so our next step is to define that function. The calculation of the next value from the previous one requires two components: a counter and a previous value (equivalent to variables 'i' and 'output' in Snippet 5.8). Nodes can only iterate over one input, so we need to combine both components into a single data object. In common with the generalised iteration graph described in Section 4.12, we can achieve this using a combined object together with a graph in the split-recombination form to split the combined input, calculate the new value and counter, and then recombine them.

To recall, the split-recombination form is as shown in Figure 5.4. It takes a combined data object containing the previous value and counter as an input, which is then split into its components so that the new value and counter can be computed, before being recombined back into a combined object.

Also in common with the generalised iteration graph in Section 4.12, we need three components: the 'Combiner' node to assemble a value and counter into a combined object, the 'GetValue' node to obtain the value from the combined object and the 'GetCounter' node to obtain the counter from the combined object. These three components are shown in Figure 5.5.

[9] JavaScript guarantees that integers will be represented correctly for positive integers up to $2^{53} - 1$. This allows for factorials of integers up to 18. When representing higher integers (up to $1.79E + 308$), JavaScript rounds as determined by the IEEE-754 double-precision floating point specification.

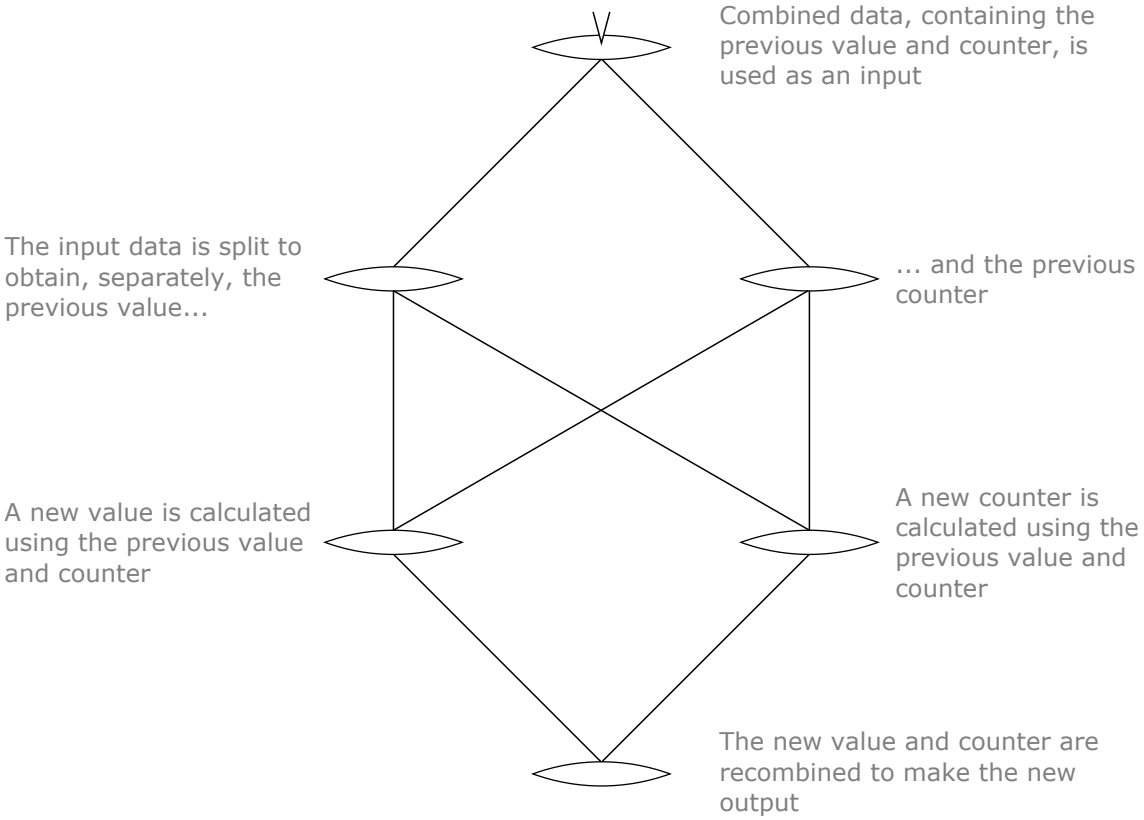

Figure 5.4. The Split-Recombination Form. Dependent iteration can only iterate over a single input, whereas the computation requires two inputs — a previous value and a previous counter. This is achieved by using a combined object that contains both as an input to the iteration function. It is then split into its two components so that they can be operated on separately, before recombining them back into a combined object.

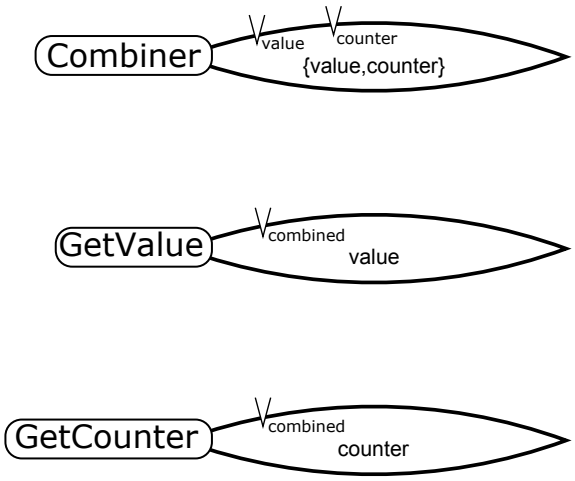

Figure 5.5. The iteration requires three components: the 'Combiner' node, which takes a value and counter and combines them into a single object; the 'GetValue' node, which extracts a value from a combined object, and the 'GetCounter' node, which extracts the counter from a combined object.

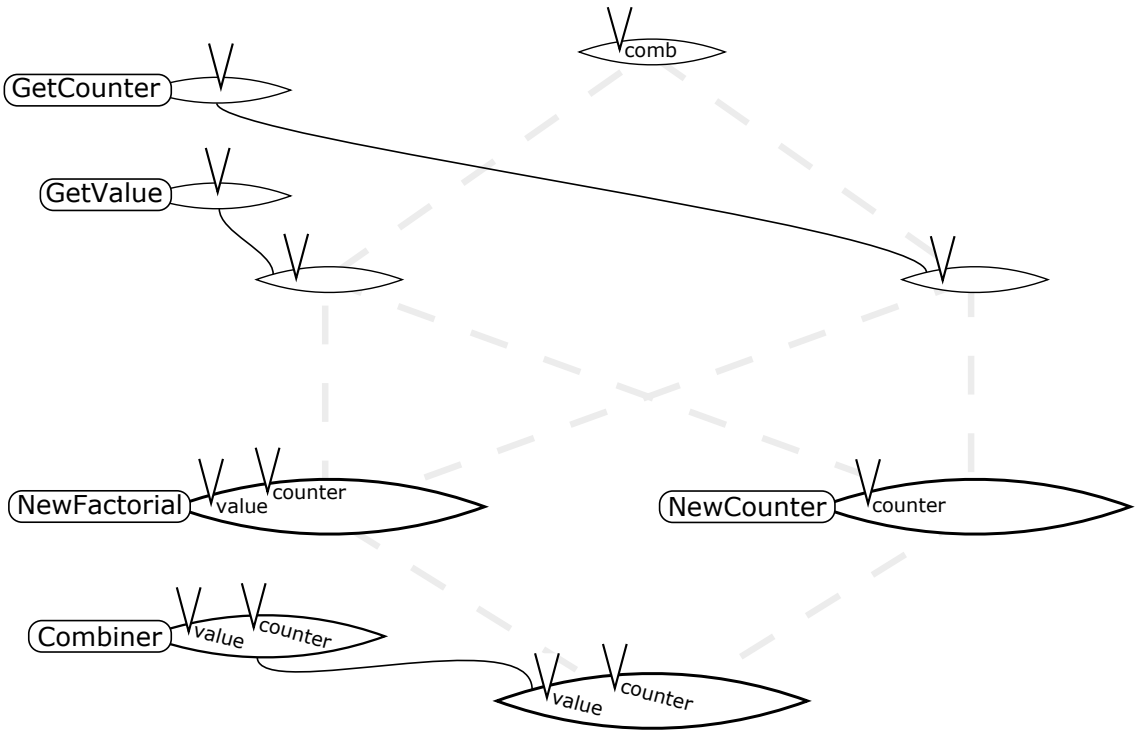

Figure 5.6. Components are assembled into the split-recombination form but not yet connected. The 'GetValue' and 'GetCounter' components are used to split the combined input ('comb') to obtain the value and counter separately. These will be used by 'NewFactorial' to calculate the new factorial. The 'NewCounter' node depends only on the previous counter, so has only one input. The 'Combiner' component will then be used to combine the new value and counter into a single object.

The components are assembled into the split-recombination form as shown in Figure 5.6. An ID node at the top has just one input, 'comb', at which we want it to receive a combined object containing the previous value and counter. The 'GetValue' and 'GetCounter' components are used to obtain the value and counter, respectively, from the combined object. The 'NewFactorial' node takes both the previous value and previous counter as inputs, multiplies them together and outputs their product. The 'NewCounter' node takes only the previous counter as an input, and increments it by one. Finally, the 'Combiner' component is used to assemble the new value and new counter into a combined object, which it provides as its output.

In Figure 5.7 the graph has been connected, the 'comb' input is inherited by its downstream nodes and, to show the remaining open inputs more clearly, a final node, named 'IterationFn', has been added to the end.

We then connect the 'IterationFn' node to the central ID node introduced in Figure 5.3. As shown in Figure 5.8, the ID node inherits the 'comb' input. We now set the ID node to iterate over this inherited input, and it will generate the two derived inputs '~sv' and '~tc' as a result.

We can now set about defining starting value and termination condition nodes. For the Factorial function, the starting value is always the same — a combined object with the value and counter both initially set to '1'. The termination condition is similar to the generalised iteration node described in Section 4.12, although in this case depends on the

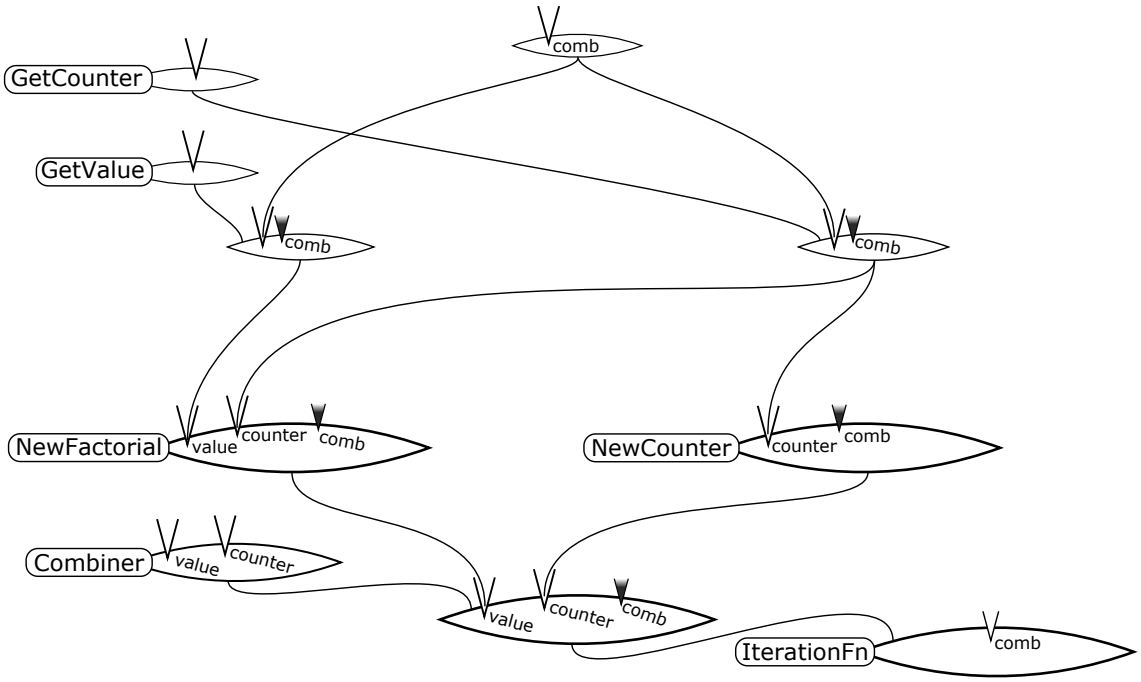

Figure 5.7. Split-Recombination form with the nodes connected. The final node takes a single input, 'comb', a combined object containing both the previous value and the previous counter. It will use them to calculate the new value and counter and provide a new combined object as an output.

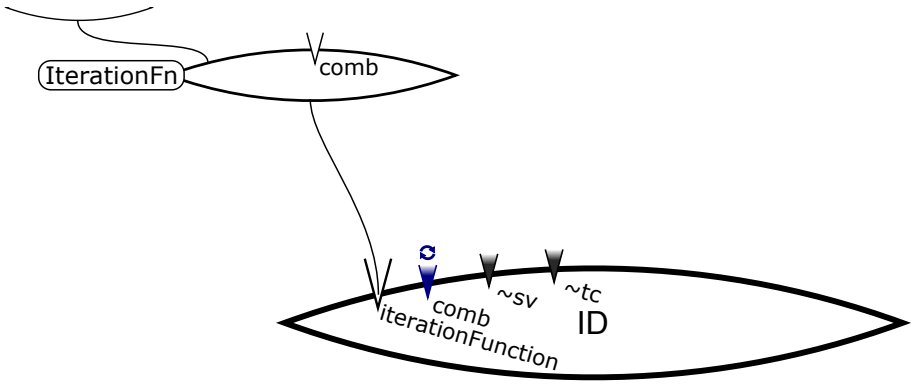

Figure 5.8. Connecting the iteration function. The 'IterationFn' node is connected to the 'iterationFunction' input of the central node, which inherits the 'comb' input. The central node is then set to iterate over the 'comb' input, and the starting value ('~sv') and termination condition ('~tc') inputs are generated.

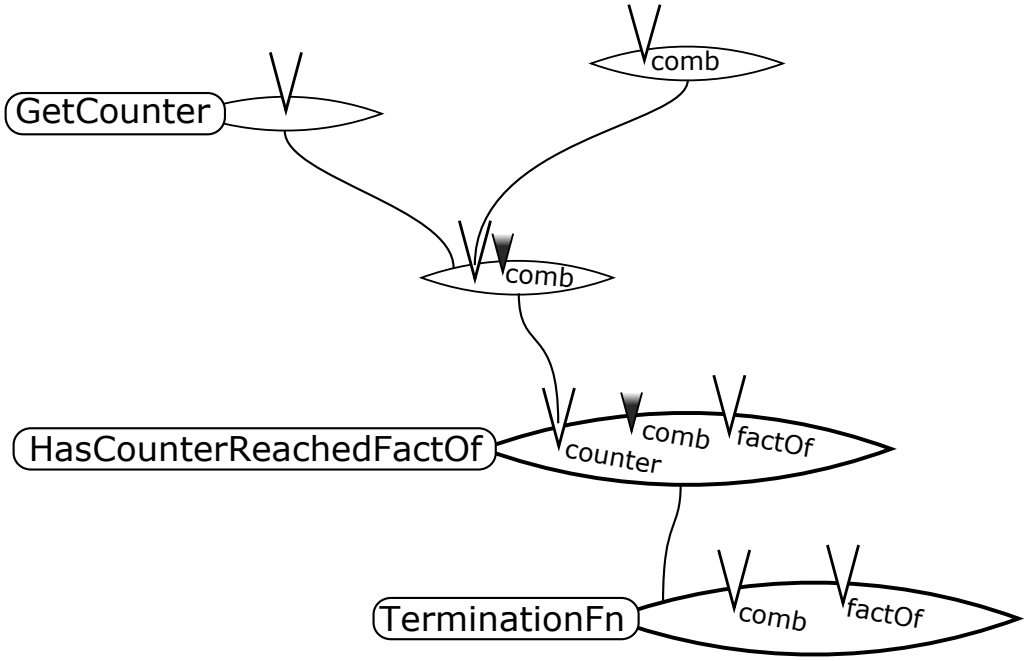

Figure 5.9. The termination condition for the factorial algorithm depends only on the counter, not the value. The 'HasCounterReachedFactOf' node tests whether the counter (provided by the 'counter' input) has reached the value we want to find the factorial of (provided via the 'factOf' input).

previous counter but not the previous value. The function should terminate when the counter reaches the value we want to find the factorial of.

The graph depicting the termination condition function is shown in Figure 5.9. It is based on a node (named 'HasCounterReachedFactOf') which compares its 'counter' input with its 'factOf' input and outputs 'true' if they are equal. The counter is obtained from the combined object (the input named 'comb') using the 'GetCounter' component. An additional node, named 'TerminationFn', is added to the end for clarity. The 'TerminationFn' node takes two inputs: 'comb', the combined value that will be the output of the iteration; and 'factOf', the number we want to find the factorial of, which will be tested against the counter each time it runs.

Finally, the components are connected together. In Figure 5.10, the 'StartingVal' node provides the combined object containing the initial value and counter, the 'IterationFn' node provides the iteration itself, the 'TerminationFn' node tests whether the counter has reached the required value; and we then use the output of the 'GetValue' node to obtain the resulting value from the combined output.

The 'Factorial' node is the end result. It has one input, named 'factOf', at which we can provide the number we want to find the factorial of, and it will output the result. In tests, the graph described did indeed correctly calculate factorials for non-negative integers up to the limit of integer representation in JavaScript[10].

[10]See footnote 9 (page 115).

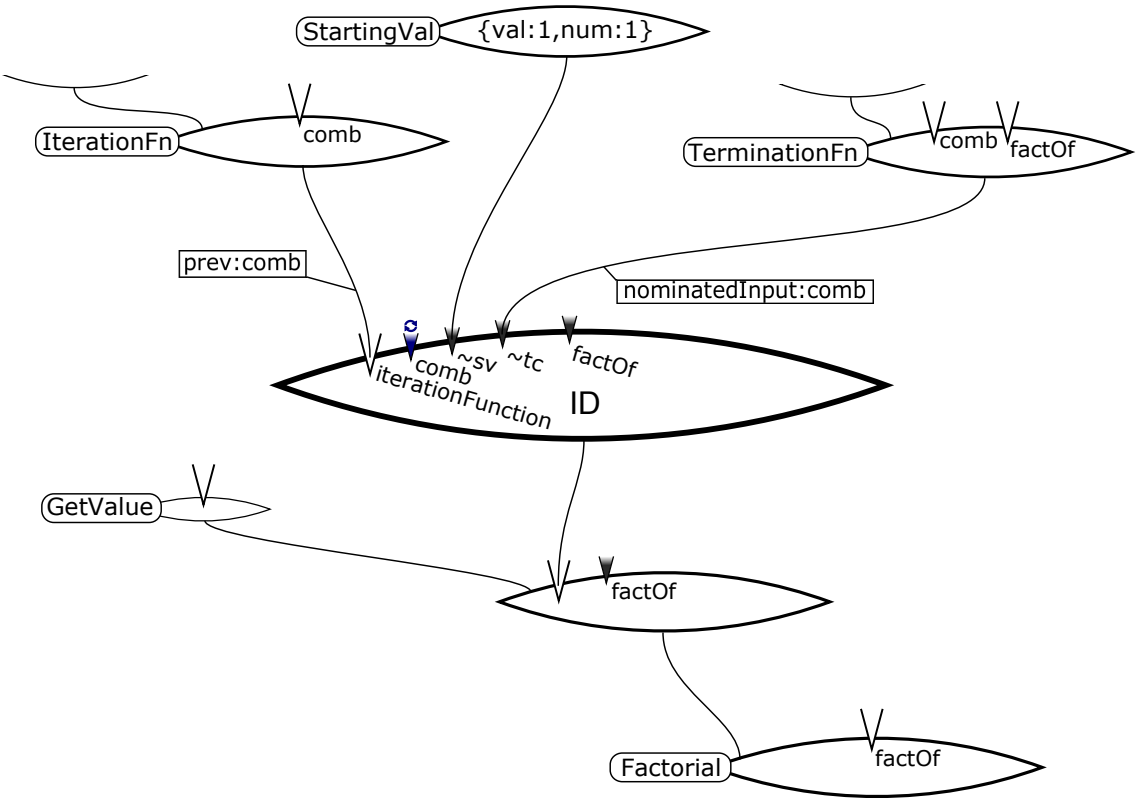

Figure 5.10. The components of the factorial algorithm are now connected together, resulting in the 'Factorial' node at the end, which takes 'factOf' — the number we want to find the factorial of — as an input.

5.3.2. The Fibonacci Sequence

The Fibonacci Sequence, the sequence of numbers starting with $(0, 1)$, in which each subsequent value is the sum of the preceding two, differs from the Factorial algorithm in needing reference to more than one previous value in order to calculate the next. In other ways it is simpler: no counter is needed, meaning we do not need the split-recombination form that was used to calculate the factorial.

As shown in Figure 5.11, the algorithm is based on the 'AppendItem' node, which has one root input, named 'arr'. On this input, we want it to receive an array containing the sequence generated up to that point and to append the next value. We want it to append the value '0' to the end of the array if the array is empty, the value '1' to the end of the array if it has only one member, and in every other case append the sum of the last two values to the end of the array.

We now set this node to iterate over its 'arr' input, at which point it will generate the two derived inputs, '~sv' and '~tc'. The starting value, '~sv' is given an empty array ('[]'); and the termination condition is connected to the node 'TerminateAtN', which returns 'true' when the length of the array received on its input 'arr' is equal to or greater than the value of its input 'n'. When the '~tc' input of the 'AppendItem' node is connected to the 'TerminateAtN' node, we must specify 'arr' as the input we want to associate with the expected input named 'nominatedInput'. The other input, 'n', is inherited by the 'AppendItem' node.

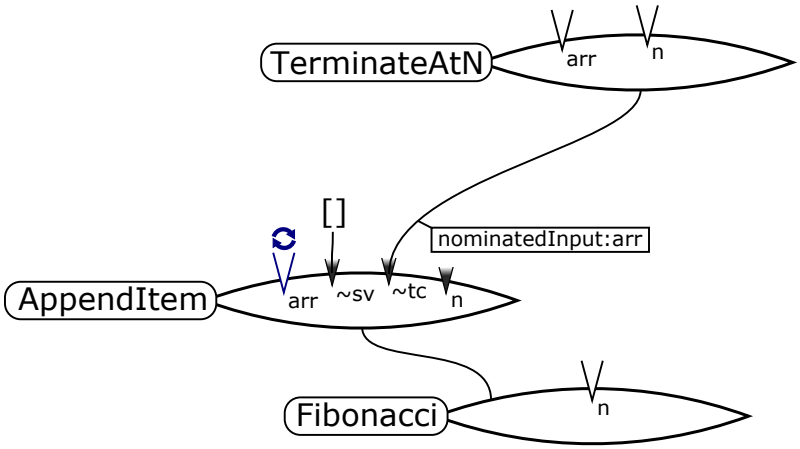

Figure 5.11. The Fibonacci sequence algorithm is based on the 'AppendItem' node, which takes an array of values as an input and provides a copy of it as an output, with the next value appended to the end. We set it to iterate over its 'arr' input, causing it to generate the starting value ('~sv') and termination condition ('~tc') inputs. The starting value is an empty array; and the termination condition is a function which returns true when its input reaches the length given by its input 'n'. The end result of the graph is the 'Fibonacci' node, which takes 'n' — the length of the sequence we want to generate — as an input.

As with previous examples, a final node is added for clarity; in this case named the 'Fibonacci' node, which has one input, 'n'. When provided with a value for 'n' it will generate an array containing the first n terms of the Fibonacci sequence. This was tested successfully with values of 'n' from 0 to 79. For values greater than this, the final item in the list exceeds the highest integer that can be exactly represented in JavaScript[11].

5.3.3. The Merge-Sort Algorithm

The merge-sort algorithm is more often used as a demonstration of a recursive function. However, since recursion can be implemented using iteration (Keller [1980]), the merge-sort algorithm can be too. The purpose of the merge-sort algorithm is to take an unsorted list as an input and produce a sorted copy of it as an output. The usual (recursive) mechanism for this is to split the list in half, for the function to then call itself on each half and merge the two returned (sorted) half-lists to make a full sorted list. It might look something like the abbreviated algorithm shown in Snippet 5.9.

Snippet 5.9

```
function mergeSort(unsortedArr){

  // First test for the base case:
  if(unsortedArr.length === 1) {
    // If the base case applies, return the argument
       unchanged.
    return unsortedArr;
  } else {
    // Otherwise:

    // First split the array into two halves
    var firstHalf = getFirstHalf(unsortedArr);
    var secondHalf = getSecondHalf(unsortedArr);
```

[11] As for the Factorial algorithm (see footnote 9, page 115), the range of values for which the Fibonacci sequence algorithm generates correct values is limited by the underlying programming language.

```
        // Use mergeSort to sort each half
        var sortedFirstHalf = mergeSort(firstHalf);
        var sortedSecondHalf = mergeSort(secondHalf);

        // Now merge them together

        // This is done by comparing the first items of
            sortedFirstHalf and sortedSecondHalf and,
            whichever is lower, removing it and adding it to
            the output array. This procedure is repeated
            until the sortedFirstHalf and sortedSecondHalf
            arrays are both empty.
        var outputArr = makeMergedArray(sortedFirstHalf,
            sortedSecondHalf);
        }
        return outputArr;
    }
}
```

Every time this function calls itself, it must wait for the function call to finish and return control before continuing. This means that the first instance of it to actually complete is the base case. Whereas in reading order the recursive version of this function appears to start with the whole unsorted array before progressively splitting and sorting smaller instances of it, the sequence of events during execution is to merge pairs of single item arrays, then two item arrays, then four, and so on, until the whole list is sorted.

The iterative version of the algorithm uses the same execution order, starting with single-item arrays, merging them, and doubling their length each time until the whole list is sorted. Unlike the recursive version it reads in the same order that it executes. The iterative version might take something like the form shown in Snippet 5.10 (written in sentences, for simplicity).

Snippet 5.10

```
function mergeSort(unsortedArr){
    // Merge the first with the second item, then third
        with fourth, etc

    // Merge the first 2 items with the second 2, then
        third 2 with fourth 2, etc

    // Merge the first 4 items with the second 4, then
        third 4 with fourth 4, etc

    // Repeat, doubling the number of items merged each
        time, until it is greater than or equal to the
        length of the list.

    // Return the resulting list.
}
```

As with the factorial algorithm the dataflow graph to perform this iteration takes a similar form to the generalised iteration node described in Section 4.12. Once again, we start with

the central ID node on which the iteration is going to take place, shown in Figure 5.12.

Figure 5.12. The central ID node. The merge-sort algorithm starts with a central ID node (one which returns its sole input unchanged).

We now need to define the iteration function for this node. In common with the factorial and generalised iteration algorithms, successive values need to be calculated using a counter as well as the previous value, so we must use the split-recombination form.

A skeleton of the iteration function is shown in Figure 5.13. Like the factorial and generalised iteration algorithms, it uses three components: the 'Combiner', 'GetValue' and 'GetCounter' nodes. In addition, it uses two others: the 'Double' node, whose functionality is to double its 'counter' input; and the 'SplitMerge' node, whose functionality is to receive an array on its 'value' input, split it into pairs of arrays of a length determined by its 'counter' input, merge each pair of arrays into a single sorted array, then append the resulting sorted arrays together into one single array of numbers. This functionality involves its own iterations, which can be built in a similar way.

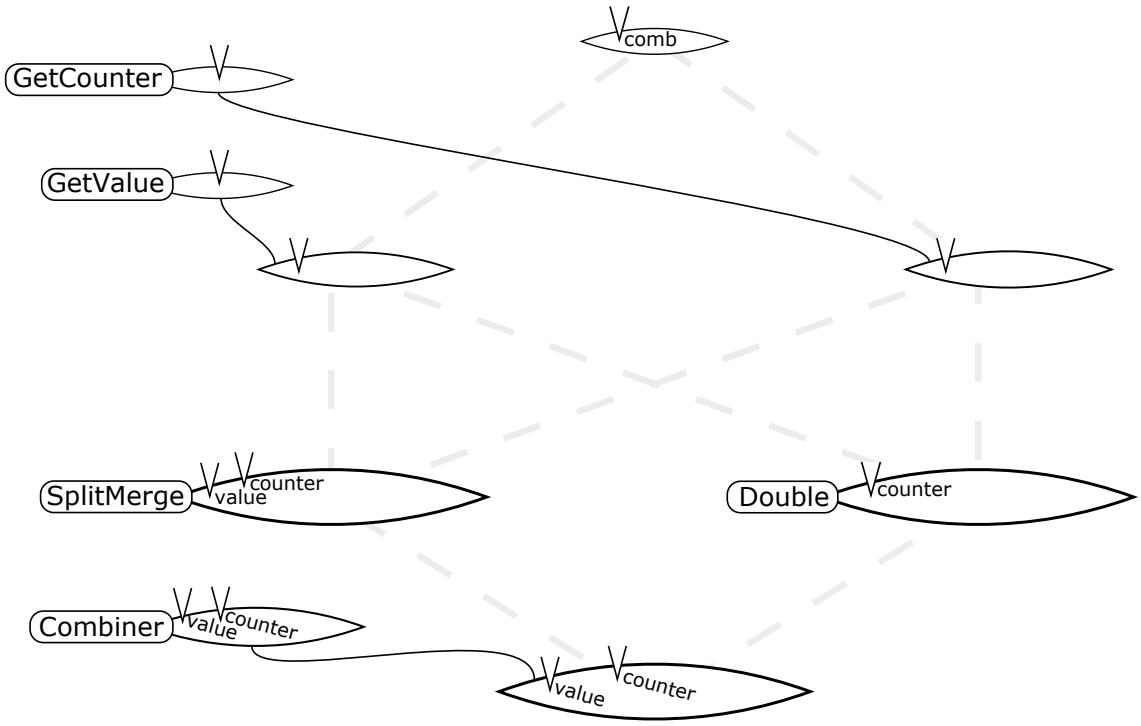

Figure 5.13. Iteration function skeleton. The iteration takes the split-recombination form, in which a combined input containing a value and counter is split into its components so that the new value and new counter can be calculated, before recombining back into a single object.

The 'Double' node depends only on the previous counter, not the previous value, so has just one input rather than two. When the nodes are connected, as shown in Figure 5.14, the 'comb' input is inherited by its downstream nodes. To clearly show the remaining open inputs a final node, named 'IterationFn', is added at the end.

We need this iteration function to be executed first with a counter value of 1, then doubling

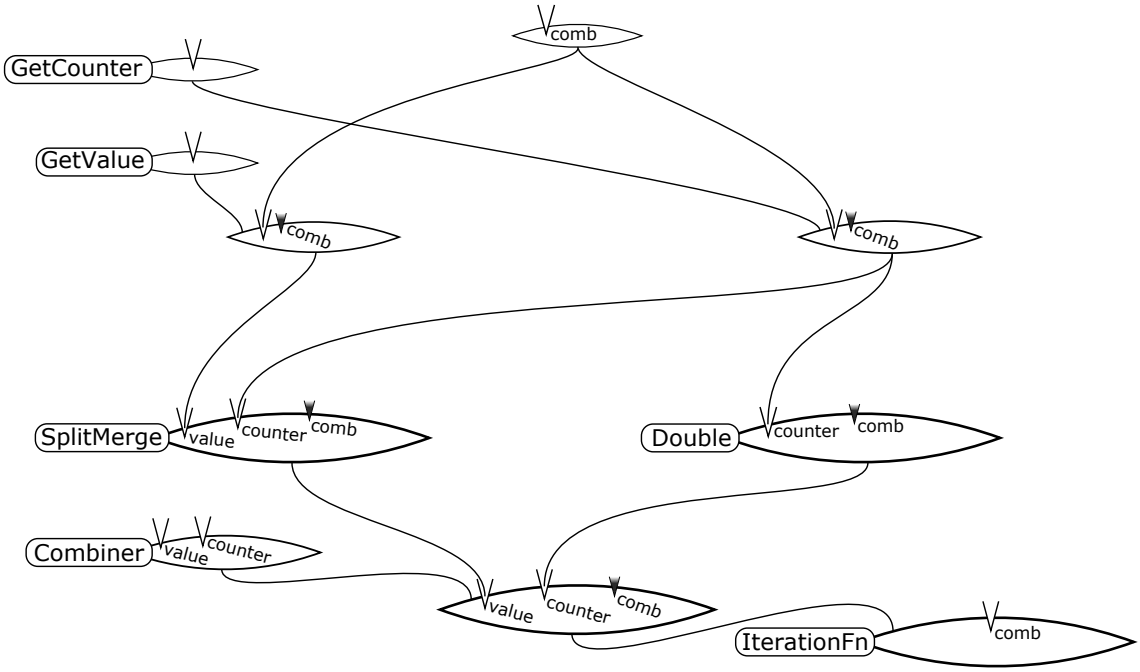

Figure 5.14. Iteration function with connections. Once connected, the open inputs cascade through the graph and are inherited by downstream nodes. The end result is the 'IterationFn' node, which has one input, named 'comb', at which we expect it to receive a combined object containing an array and a counter.

the counter each time until the counter value equals or exceeds the length of the array. After each iteration, its output will be an array in which each set of members of length $(2 \times counter)$ will be sorted. We now connect this resulting 'IterationFn' node to the original central ID node, which inherits the 'comb' input, as shown in Figure 5.15. We set the central ID node to iterate over the inherited 'comb' input, and it generates the two derived inputs, '~sv' and '~tc'.

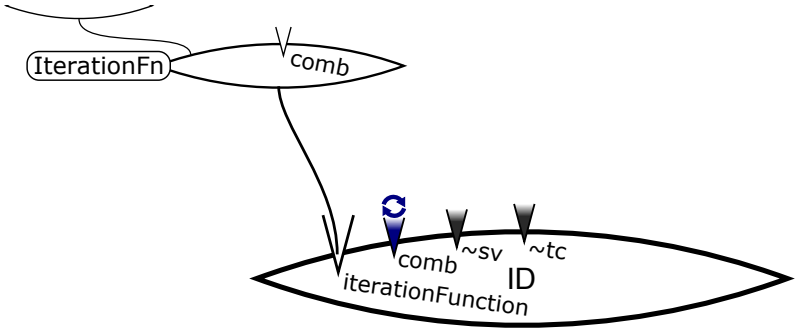

Figure 5.15. Connecting the 'IterationFn' node. We connect the 'IterationFn' node to the central ID node, which inherits the 'comb' input. When we set the central ID node to iterate over the 'comb' input, it will generate the derived inputs for the starting value ('~sv') and termination condition ('~tc').

The starting value and termination condition now need to be defined. The starting value must be a combined object containing the initial array and initial counter. The initial counter is always 1 and the initial (unsorted) array is provided by the user. As shown in Figure 5.16, we use the 'Combiner' component to combine the two parts. The value '1' is provided for the initial counter, and an ID node with the input 'unsortedArr' is connected to its 'value' input, to provide a more meaningful name when inherited later.

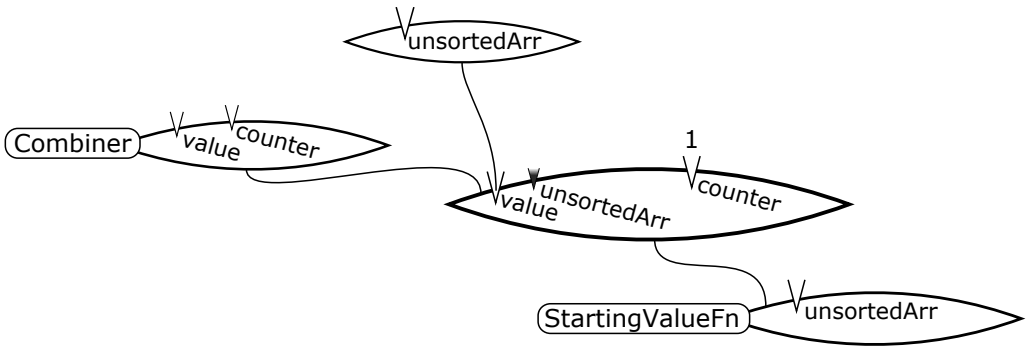

Figure 5.16. Defining the starting value. The counter is provided with the value '1' and the 'value' input is connected to an upstream ID node to provide the more meaningful name 'unsortedArr'.

For the termination condition, we want the iteration to terminate when the counter is equal to or greater than the length of the array. To define the termination condition in dataflow, we need to use the split-recombination form again. As shown in Figure 5.17, we split the combined object to obtain the array and the counter separately. The comparison is carried out by the node named 'DoesCounterExceedValueLength', which compares the length of the incoming array (which arrives on the 'value' input) to the value of its 'counter' input, and outputs a Boolean value; true if the counter is equal to or greater than the length of the array, or false otherwise.

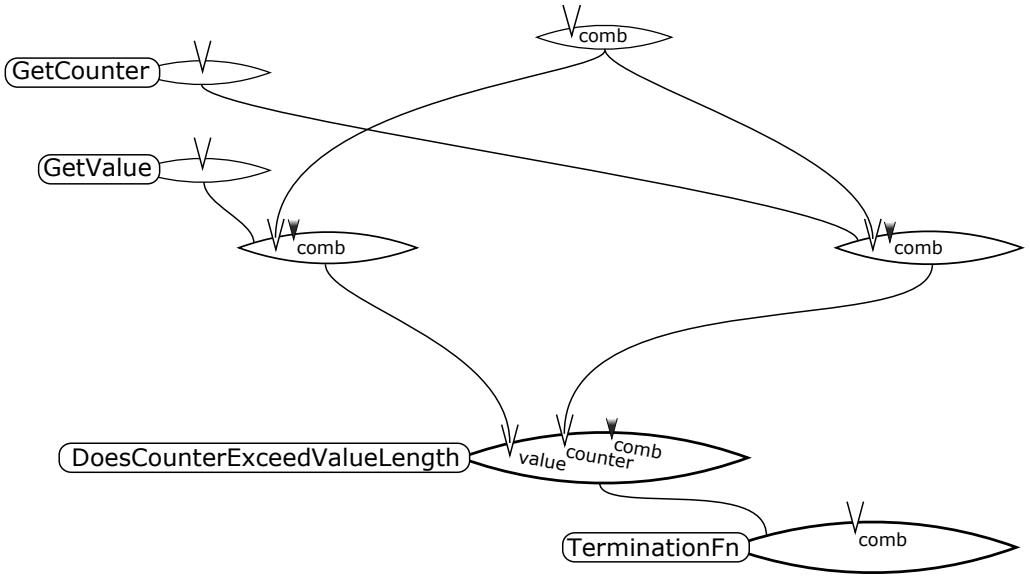

Figure 5.17. Defining the termination function. The termination function takes a similar shape to the split-recombination form used previously. The combined input is split into a value (an array) and a counter, which are compared, outputting 'true' if the counter is equal to or greater than the length of the array, or false otherwise.

Finally, the components are connected together, as shown in Figure 5.18. In this graph, the 'comb' input of the 'TerminationFn' node is the nominated input for the '~tc' input to the central node, so is not inherited. The 'unsortedArr' input of the 'StartingValueFn' node is inherited all the way through the graph. The end result is a node named 'MergeSort', which has just one input, named 'unsortedArray'. This final node takes an unsorted array as its input and provides a sorted copy of it as an output.

The algorithm was tested with 100 arrays of 10,000 randomly generated numbers, positive

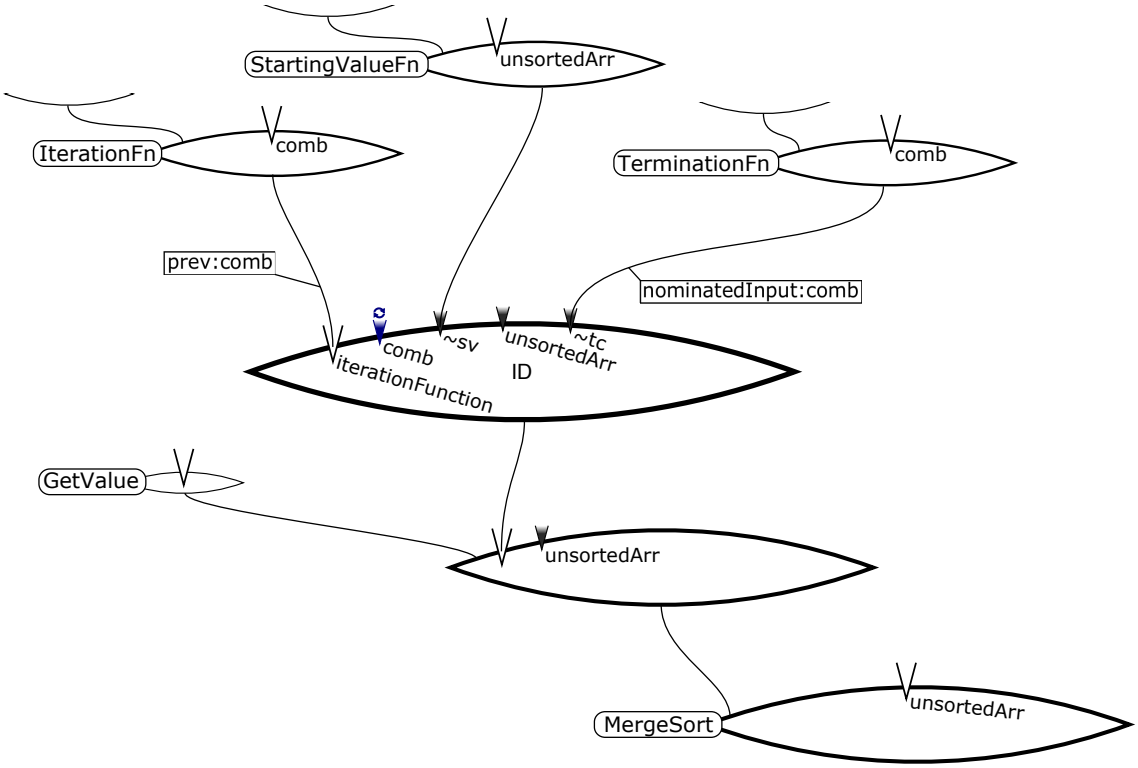

Figure 5.18. Connecting the components together. The components are connected to create a final 'MergeSort' node which has one input, named 'unsortedArr'.

and negative, some using integers and some using floating point numbers. In every case, the algorithm delivered a correctly ordered array as an output.

The embedding of an independent iteration inside a dependent one has the effect of synchronising successive dependent iterations. We could depict the execution order as a tree, as shown in Figure 5.19, in which a parent node can only execute when its child nodes have completed. In our implementation, the levels are synchronised, meaning that each level is able to commence only when all nodes in the level beneath it have completed. A more efficient implementation would enable each node to commence when its own child-nodes had completed, without waiting for other nodes at the same level.

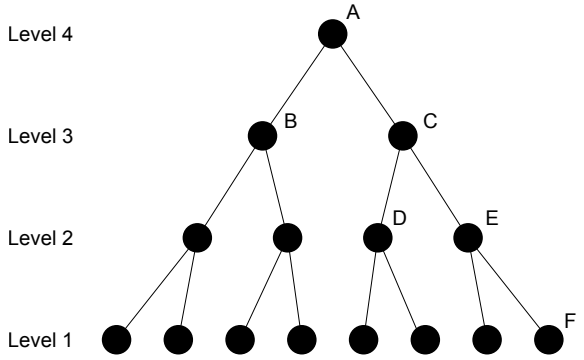

Figure 5.19. An Execution Tree. Each node can only begin executed when its child nodes have been completed. The node labels are used in Figures 5.20 and 5.21.

Whether this makes a difference to overall execution time depends on where the slowest nodes lie. In Figure 5.20, the slowest nodes at each level, depicted as white, lie on the same path. In this case the two approaches would have the same overall execution time.

However, in Figure 5.21, the slowest nodes lie on different paths. In this case, an optimal approach, Node D would be able to start execution as soon as its child nodes had completed. In the implementation used, it would have to wait for Node F to complete before it could commence. Likewise, Node B would, in an optimal implementation, be able to begin execution without waiting for Node D to complete. Although both exploit the same degree of parallelism, the synchronisation of levels used here introduces delays whose magnitude would depend on the computation in question.

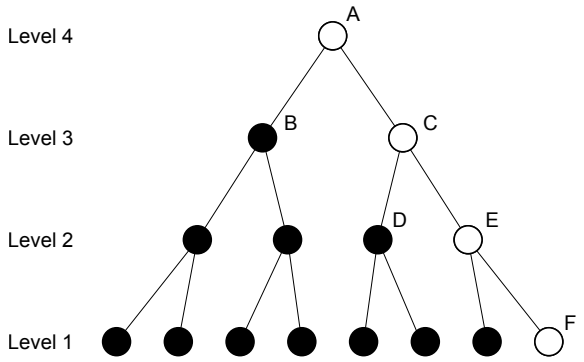

Figure 5.20. In this execution tree, the slowest node at each level is unshaded. The total execution time, in this example, would be the same for either execution approach, with Node A having to wait for Node C, which has to wait for Node E, which has to wait for Node F.

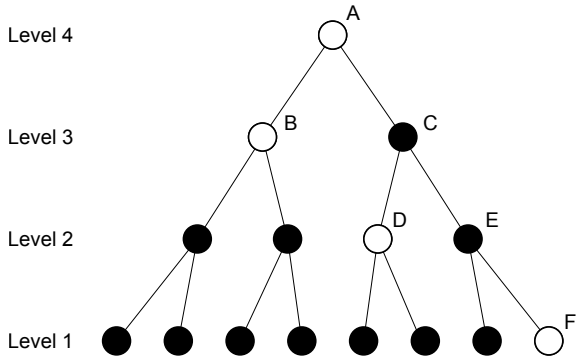

Figure 5.21. In this example, the slowest node at each level (the unshaded nodes) lies on a different path (with the exception of Node A, which is on every path). For this example, a more efficient execution strategy would enable Node D to begin execution when its own child nodes had completed, without waiting for Node F to finish. Likewise, in an optimal implementation, Node B would be able to begin when its child nodes had completed without waiting for Node D. This would lead to a faster total execution time for an optimal approach than when levels of the tree are synchronised.

5.4. Summary

The implementation, which is publicly available[12], successfully demonstrates that the three test algorithms can be built and deliver the correct results, using partial evaluation, dimensionality and dependent iteration. As pointed out in Section 4.8, partial evaluation makes algorithms reusable. Once one user has built and shared a generalised iteration node, such as that described in Section 4.12, any other user will be able to connect to it and reuse it, rather than having to build each iteration from scratch. As more nodes are built and shared, and the pool of available nodes grows, the value of the system will

[12]The implementation can be found in a Mercurial repository at the following address: `https://bitbucket.org/danieljmaxwell/developments-in-dataflow-programming`

increase and more elaborate functionality will become easier to achieve. The next chapter provides a broader evaluation of the system and its implementation, and a description of the experience of using it.

Chapter 6

Evaluation and Results

Chapter 4 (Definition) described how a distributed dataflow coordination system might work and proposed a method of representing dependent iteration that is consistent with the principle of acyclicity. Chapter 5 (Implementation) described an implementation of such a system and its use in building a set of three test algorithms — calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort. This chapter describes the experience of using the implementation, and highlights some of the improvements that could be made and principles applied in onward development.

6.1. Programmability

As described in Chapter 2 (The History of Dataflow), much of the work on dataflow programming (and the flow-chart concept it is based on) has been motivated by programmability. The experience of using the implementation highlighted the need for three characteristics of the system in order to fully exploit that potential improvement: visualisability, interactivity and availability.

6.1.1. Visualisability

After working with both diagrammatic and text-based representations of dataflow graphs, it became clear that dataflow is easier to use visually than in text. In most cases, it was found that the easiest way to create a dataflow graph in a text-based interface was to draw the graph on paper first, and then use the diagram to write the text. Correspondingly, it was found that the easiest way to make sense of a text-based graph definition was to use the textual program to draw a diagram in order to visualise the flow of data and the sequence of events.

The visualisation is a necessary component, not an optional extra. That is not to say, however, that the visualisation needs to be rigidly defined. As described in Chapter 4, the visualisation is an interchangeable component: any number of visualisations could exist concurrently within the same system. They could look similar to the diagrams in this document, or could look completely different; they could be in two dimensions, or three;

and could use shapes, interactions and types of hardware as yet un-dreamt of.

6.1.2. Interactivity

Interactivity is the ability of the programming interface to respond immediately to changes and display the changes to resulting values in the graph in as close to real-time as can be achieved. It enables extremely rapid feedback on the functionality and output of the program and provides the programmer with constant verification of the functionality they are creating. Chapter 3 (Software Engineering) discussed the importance of uncovering changes early for the sake of project efficiency. The principle applies as much to the discovery of programming errors on a second-by-second basis as it does to the discovery of specification errors and changes in the project environment. It is easier to identify the source of a bug if it is discovered immediately after the action that caused it and is a reason why testability, listed as one of the architectural principles in Section 3.4.1 (Uncovering Knowledge), is so important.

As described in Section 4.17 (Testing and Development), the 'partial evaluation', 'expected inputs' and 'independent iteration' features all improve testability. The use of multiple processors described in Section 5.2 (Code Structure) means that calculations can be completed on separate processors without obstructing the user interface or programming activity. The use of subscription types, as described in Section 4.15 (Subscription Types), means that new computations can be triggered even if likely to be quickly superseded and cancelled, making it relatively low-cost to trigger an update immediately after every change, even when those changes are happening in quick succession. The fact that a change to the graph results in only limited recalculation further adds to these gains, producing fast results when changes are made.

Although these features contribute to the ability to achieve interactivity, interactivity still needs to be an explicit (though to some extent flexible) decision in the design of the system. The text-based implementation's interactive programming interface added considerably to the ease of programming. Interactivity in a visual interface would make it easier still.

6.1.3. Availability

Availability is one of the desirable characteristics listed in Section 3.4.4 (Desirable Characteristics). The system of subscriptions described in Section 4.6 (The Service-Provider Model) enables availability to be achieved without unnecessary resource use — sections of a graph can be switched off if not being used.

In a distributed system, where components of a single program may be hosted on different machines and controlled by different people, it is important that components of the program should be available when needed, even if temporarily 'switched off'. In a networked environment with distributed ownership, this can never been guaranteed, but the system of partial evaluation and the caching of node inputs within each node adds robustness (at some resource cost) and improves availability across the system as a whole.

6.2. Speed

Speed has not been, at this stage, an objective of the design. However, it is an important consideration of any computer system. While dataflow has the advantage of being parallelisable, it also brings with it a resource overhead. The granularity of the computations in nodes affects the impact of this overhead. A finely granular system has greater opportunity for parallelisability but a greater overhead, and a coarsely granular system has a lower overhead but sacrifices some of its parallelisability. The dataflow system described in this document allows a programmer to write any amount of code within a node, leaving the level of granularity to the programmer and their balance of priorities.

The implementation has not been subjected to any form of optimisation. It is a dataflow system implemented in a sequential language (JavaScript), on a platform (NodeJS) which parses it into another sequential language (C) in order to be run on hardware that is designed for sequential execution. When communicating between nodes, a partially evaluated function is encoded as an object (in its Abstract Syntax Tree form), which is then converted into text, transmitted to the next node before being parsed and converted back into its Abstract Syntax Tree form. On execution, this process is repeated in order to send the function to the execution process. On arrival at the execution process, it is compiled back into a JavaScript function and executed, at which point NodeJS will parse it again to transform it back into Abstract Syntax Tree form in the back end of the language interpreter, before being executed. An iteration will repeat this process once for every iteration. The whole procedure is extremely inefficient and the test algorithms described in Chapter 5 can be executed substantially faster when written in sequential code directly in the underlying language.

Some of this inefficiency is in the implementation, some is a result of being a layer on an underlying sequential system, and some is intrinsic to dataflow. While much of the performance overhead can be optimised, in particular by avoiding unnecessary copying of data, some inefficiencies will certainly remain. It seems likely that a coarse granularity would be needed to make dataflow workable until some of the most glaring inefficiencies have been addressed. Furthermore, since it is in dependent iteration that dataflow has the biggest handicap, a cautious approach to using dependent iteration in performance-sensitive applications would be well-advised, until the performance can be better-optimised.

Peyton Jones [1987], in his book on implementation of functional languages, expressed the view that lazy evaluation was a "*critically important feature for functional programming*", but also pointed out that it came at the expense of execution speed. The implementation described in Chapter 5 uses eager evaluation, completing each execution as soon as the data is available to do so. Given the trade-off between processor time and execution speed this involves, it makes sense for the decision between eager and lazy evaluation to be left to node autonomy (and ultimately based on the priorities of the programmer or user). The capability to choose lazy evaluation does not yet exist in the implementation, so would need to be included.

The purpose of this project is not to create a fast or optimal implementation of dataflow,

but rather to demonstrate that the test algorithms can be achieved at all, in a way that is consistent with the dataflow paradigm. The task of optimisation remains one for further work.

6.3. Distributability

The implementation was built with future distributability in mind. The main way this has been done is by converting messages between nodes into a format that could be transmitted over a network. Nodes also take a defensive approach to other nodes, copying both outgoing and incoming data to protect against mutable data being changed by untrusted code within other nodes. Although this helps verify that the system can work in a distributed environment, it also introduces unnecessary work when nodes are running trusted code on the same machine. There are numerous possible approaches to such problems, and in a real distributed environment, much of this overhead would be unnecessary and could be reduced or eliminated entirely.

6.4. Summary

The use of partial evaluation, data dimensionality and the 'expected inputs' feature were successful in providing the basis for a representation of both dependent and independent iteration in dataflow, and the implementation showed that they could be used to define the test algorithms and generate correct results (within the constraints of the programming language used). Use of the implementation revealed the importance of three characteristics to the usability of the system: visualisability, interactivity and, in a distributed system, availability.

As has been widely discussed in the field of dataflow, performance remains an issue; one which previously proposed solutions could be expected to mitigate, both through use of coarse-gained dataflow programs and through compilation. This second of these has been successful at mitigating the dataflow overhead in previous implementations, such as DFScala (Goodman et al. [2013]).

The internal node implementation, the method of computing functions, and the provision of computing resources, although included in the implementation, are interchangeable components rather than being intrinsic to the system described. Any alternative internal node implementation or hardware could equally well be used concurrently with any other, within the same dataflow coordination system, providing they complied with a consistent communication protocol and set of minimal behaviours required by the system.

The next chapter lists some of the problems that remain to be solved and features that remain to be built, and makes suggestions of new features that could usefully be added in future.

Chapter 7

Further Work

This chapter describes three areas of further work. The first, discussed in Section 7.1, is a full specification of the system. This includes protocols determining the required behaviour of system components and the interfaces through which the components will communicate with each other and the outside world. The second, discussed in Section 7.2, is to investigate additional features to enhance or extend the functionality described in Chapter 4 (Definition). The final area of further work, discussed in Section 7.3, is the further development and testing of the implementation.

7.1. Full Specification

To bring the vision of a unified global dataflow coordination system to fruition would require the behaviour discussed in Chapter 4 (Definition) to be resolved into a full specification of the protocols and interfaces through which nodes and system components would interact. This would define the minimal behaviour requirements of system components, but leave sufficient flexibility for node autonomy, differing user priorities, the full diversity of current and future hardware and the wide range of applications to which the system may be applied.

The interfaces should include external interfaces through which nodes could be queried or controlled and internal interfaces through which interchangeable components could be attached. Chapter 5 (Implementation) describes example implementations of some of the interchangeable components, but the specification should allow for multiple implementations of the same components to exist concurrently on the same system, and no implementation should restrict users in their choice of components.

Examples of components that should be interchangeable include the choice of cloud supplier to store node data; separately, a cloud supplier to perform computations; and the choice of user interface through which to interact with the graph. The choice could be based on speed, proximity, price, reliability, security, legal framework, or any other combination of concerns that a user considers to be important. Similarly, programmers could be offered a choice of programming language, which could be broadly similar to the language

described here, or could be completely different if required. The purpose would be to make the system as flexible and extensible as possible, minimising any constraints on its use and future potential.

A full specification would require fuller development of some of the ideas discussed in Chapter 4, including:

- side effects and internal state (Section 4.3);
- subscription types, scheduling and throttling (Section 4.15); and
- synchronisation (Section 4.16).

It would also require a handful of practical problems to be solved, including:

- prevention of cyclical graphs;
- privacy, security and access rights; and
- cost accounting.

These subjects are discussed in more detail below.

7.1.1. Side-Effects and Internal State

Section 4.3 discusses the subject of side-effects. The first step in enabling side-effects would be to allow nodes to store data internally. An additional step that would allow richer functionality would be to allow side-effects outside the system.

This would require some form of protection against the possibility of malicious code in a partially evaluated function leaking data without the knowledge of the user. Requiring the explicit permission of a user to perform external actions might provide some protection against this possibility. A set of predetermined actions provided by the function language would be the safest way to deliver this functionality, and would provide safety in a way that is consistent with the security measures applied to the function language as a whole.

7.1.2. Subscription Types, Scheduling and Throttling

Section 4.15 (Subscription Types) describes a set of choices determining the behaviour of nodes when update notifications and data updates arrive either out of order or with high frequency (with new data arriving before the previous data has been processed). The decision of how to deal with such a situation is largely a matter of user preference, but a node does require sufficient information from its upstream graph, through its subscriptions, to support the decisions made. To enable this, the system needs a wider variety of types of subscription to communicate those preferences between nodes.

Alternative solutions to the problems resulting from high frequency updates could include throttling and scheduling. With throttling, a maximum frequency could be set, so that some updates would be ignored if the frequency exceeded a set level. With scheduling, a node could be set to provide an update on a regular schedule, or at predetermined times.

As well as providing a solution to high frequency updates, scheduling could also be used to sample continuously changing external data or, where an external action is needed at a set time, allowing a one-off or periodic future execution. In the example of Bob's pie-making node, discussed in Section 4.3, this could be used to request the required pies at a set time.

7.1.3. Synchronisation

The issue of synchronisation, discussed in Section 4.16, relates to the problem of clocks not being synchronised between distributed nodes. The exact choice of solution should be left to node autonomy, for users to decide based on priorities. However, subscriptions and the meta-data contained in update notifications and data updates must contain sufficient information to communicate those choices and the resulting uncertainties, to support the full range of possible choices. Furthermore, node interfaces may also need to provide additional clock and latency data in some cases, depending on the choices made.

7.1.4. Preventing Cyclical Graphs

It is part of our definition of dataflow (Chapter 4) that the graph should be acyclic. The complexity of determining whether any particular graph is acyclic scales linearly with the number of connections in the graph (Jungnickel [2008] provides a proof of this[1]). However, this does not completely solve the problem. In a globally connected dataflow graph with many distinct users, we could expect a large number of edges with new ones being added at a high rate, each requiring protection against circularity. Even though the check is only linear with respect to the number of edges, checking every change for circularity across the whole graph could prove prohibitively expensive.

A number of alternative approaches are possible, with varying levels of speed, reliability and resource cost. One approach could be to interrogate the upstream graph of every new connection that is created. This would enable the node or user interface to verify that the particular change in question did not introduce any new circularity. A lighter-touch approach could be to introduce a new message type which would be sent by a node to its upstream nodes in the event of a new connection being made. These 'new-connection' messages, containing the origin and update ID of the change in question, would be passed on by each node to its upstream nodes until reaching the top of the graph. If a node received a new-connection message that had originated at itself, it would signify that a circularity had been created and it could cancel the change in question. In a test implementation it is possible to solve the problem using a resource-intensive solution, such as checking the whole graph when a connection is made. In a production system, a more efficient mechanism will be needed.

A more difficult type of loop to eliminate is where an external action by a node leads indirectly to one of its upstream nodes being changed. Some protection against this could be achieved by setting a limit to the frequency of updates that might be expected and

[1] Theorem 2.6.6, on page 48 of Jungnickel's book, is that the topological sorting algorithm he provides can be used to determine whether a given graph is acyclic and that the complexity of doing so is proportional to the number of connections (referred to by him as 'edges') in the graph.

notifying the node's owner if that frequency is exceeded, although this would only mitigate rather than completely eliminate the problem.

The choice of a suitable combination of strategies will be a balance between the magnitude of the problem and the cost of finding and eliminating it within the applicable time scales. As with many other priority choices within the system, it may be best to allow flexibility through the core functionality and leave it to node owners to choose priorities. Cost accounting (discussed below in Section 7.1.6) could help to ensure that users are incentivised appropriately to avoid creating their own cyclical structures.

7.1.5. Privacy, Security and Access Rights

The specification needs a mechanism by which access rights to nodes can be controlled. The options should include keeping them private or public, sharing with specific people or groups, or enabling collaborative editing with specified individuals or groups, along with a mechanism for transferring ownership of a node.

Additionally, there may be a need, in some cases, for a level of privacy that would prohibit a node from being transmitted to prevent its content from being read. In such cases, it may be necessary to keep the partially evaluated function on trusted hardware, where it would be executed on behalf of its service users when needed.

7.1.6. Service Agreements and Cost Accounting

Section 4.6 (The Service-Provider Model) discusses the need for contracts between the owners of nodes to support the service-provider model of interaction. Associated with this is the likely need for one node owner to be able to charge another for providing services to their nodes. Furthermore, the interchangeable components of the system include computing resources, such as processors and memory, which are likely to be charged based on the level of use. A range of pricing options might be needed, including per-usage charges; and the interfaces between nodes and system components would need a system of contracts and accounting to support the corresponding agreements between their owners. The easier and more automated such a system can be, the more smoothly it will run.

The platform used in the implementation, NodeJS, makes it possible to measure the processing time and memory used by its child processes, and can therefore measure the resources used in each execution of a node. Many other platforms and languages allow similar functionality. This information could be used to measure and therefore charge for the resources used by nodes. An additional benefit of detailed resource-use data is that it could also be used to assess and optimise the efficiency of nodes or models.

7.2. Additional Features

This section discusses additional features to enhance or extend the functionality described in Chapter 4 (Definition).

7.2.1. Manipulating Partially Evaluated Functions and Graphs

In the same way that some languages, including JavaScript, treat functions as a first order data type, allowing one function to accept another as an argument value, it would be useful for a node to be able to accept a partially evaluated function as an input. This partially evaluated function could then be passed to the node's internal function to be analysed or manipulated. This would require an input parameter specifying that the node should pass the partially evaluated function to its internal function; and the node language would need to be given the ability to read, manipulate, re-generate and return a partially evaluated function as an output.

Another even more useful step would be the ability to pass a segment of a graph to a node's input. This would require a way to encapsulate a section of a graph so that it could be passed as a unit of data via a connection and, as with partially evaluated functions, would require an input parameter and suitable functionality to be given to the function language. This would allow for a graph to be able to add, edit, remove or reposition nodes, to set their parameters, or execute the graph, and return a new or edited graph as an output.

This ability to operate on a graph might also contribute to the functionality described in Section 4.12 (The Generalised Iteration Node) of generating nodes for which the number of inputs the parameters of those inputs could be configurable based on one or more of its other inputs.

7.2.2. Grouping

There are a handful of problems to which grouping is a potential solution, although it may not necessarily offer solutions to all concurrently. These are discussed below.

7.2.2.1. Collapsing a Graph Segment

The first possible use for groupings is providing the ability to visually collapse a graph. One of the problems with a dataflow graph is that when the whole graph is viewed it can take on the appearance of an untidy spider's web of connections. When writing code in text, the programmer can organise code by grouping it into functions or modules, making it possible to view both the high level dependencies between modules and also to focus attention on a small subset of the program where needed. Dataflow needs a similar mechanism to allow programmers to choose a level of detail or overview at which to view the graph.

One of the benefits of dataflow is its ability to summarise the product of a whole section of graph in a single node. An additional ability to collapse and expand sections of the graph would provide the potential to see either an overview or detailed view within the same style of interface. The exact mechanisms need to be thought through. Examples of questions that need to be answered are:

- Should the programmer be able to select any combination of nodes and collapse them, or should there be some constraints?

- Once collapsed, should the connections between one collapsed group and another reflect all of the connections between the nodes within those groups?
- How should the system handle the appearance of a cyclical graph between collapsed groups? If groupings were unrestricted and connections appeared between groups, it would be possible to have a set of groups with cyclical connections, even where their underlying nodes were not cyclical.

7.2.2.2. Unordered Collections

A second possible use for groupings of nodes would be to define an unordered collection of nodes. Imagine we want to find the sum of an initially unknown number of nodes. An elegant way of achieving this would be if a group could be defined in the user interface, whose output was an aggregation of the outputs of the nodes within it, which could be connected to the input of a node whose job it was to find the sum. An example is shown in Figure 7.1.

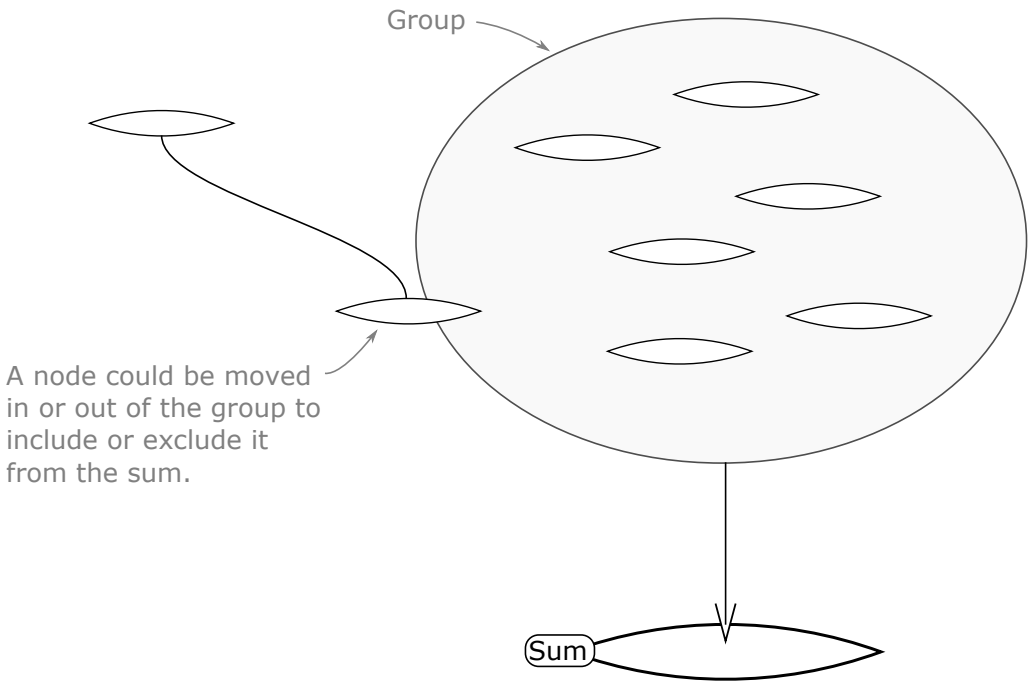

Figure 7.1. Using a group to define a collection. This allows a group to be used as a collective input to a node, so that nodes can be added to or removed from the input, and the number of nodes contained does not need to be known in advance.

In this example, the group is shown in grey, containing six nodes. If we assume that a group can have an output, we can now connect this output to the input of the 'Sum' node. By using groups in this way, it becomes easy to set up an aggregation, and easy to change its contents by dragging a node into or out of the group. In effect, this forms connections between nodes, but using groups to simplify the user interface for setting up such connections.

7.2.2.3. Batch Operations

The next possible use of a group is for the purpose of batch operations in the user interface. Like unordered collections, this is a user interface device rather than necessarily having

any underlying meaning. It would be similar to the method of selecting files to perform batch operations on them (such as moving or copying) in file management systems. The use of a group for achieving this could provide a selection that persists between sessions. Labels (discussed in Section 7.2.4) could provide a similar mechanism.

7.2.2.4. Defining Models

A related use of groups is to provide functionality analogous to opening and closing files. If a collection of nodes were enclosed within a group, the group could be collapsed to appear in the graph as a single node, and expanded again to view the nodes within it when needed. A collection of nodes enclosed by a group in this way could be referred to as a 'model'. Groups could themselves be collected within other groups to create a system of filing similar to the folders or directories used in most operating systems. Access settings for a group could be used by its owner to allow other users to view or edit the nodes within it, potentially enabling collaborative real-time editing. This use of groups also provides a possible visual mechanism for defining a segment of a graph to be consumed by other nodes (as described in Section 7.2.1), although that conflicts with the use of groups for defining unordered collections described in Section 7.2.2.2.

7.2.2.5. Defining Data Structures

A final, possibly more complex, use for groups is in defining data structures. This is discussed in more detail in Section 7.2.3.

7.2.3. Representation of Data Structures

Whereas a unit of data, described in Section 4.10 (Dimensions), can hold any type of data it is nevertheless sometimes beneficial to be able to represent data types at graph level as well as within data units. This can be useful in providing a way to operate on and manipulate those data structures at graph level, and can sometimes enable additional functionality. An example is where tables represented at graph level are used to provide independent iteration (described in Section 4.11.1 — Independent Iteration). The work remains to decide how object-oriented-like data, relational database-like data and graph data might be represented at graph level.

One possible method for enabling object-oriented-like representation in a dataflow graph might be to use groups to define 'objects', which could then be reused elsewhere in the graph. As an example, imagine we want to define a new data type specifying the behaviour of complex numbers. We could start by creating a group containing two nodes, named 'R' and 'I', representing the real and imaginary parts of a complex number. As shown in Figure 7.2, a group is used to define this object. From it, two instances are created.

We can imagine the creation of functions designed to operate on instances of this type; they could, for example, perform arithmetic operations on complex numbers by extracting, separately, the real and imaginary parts of an instance, performing the required real number arithmetic on the components, and using the results to build a new complex number instance as an output.

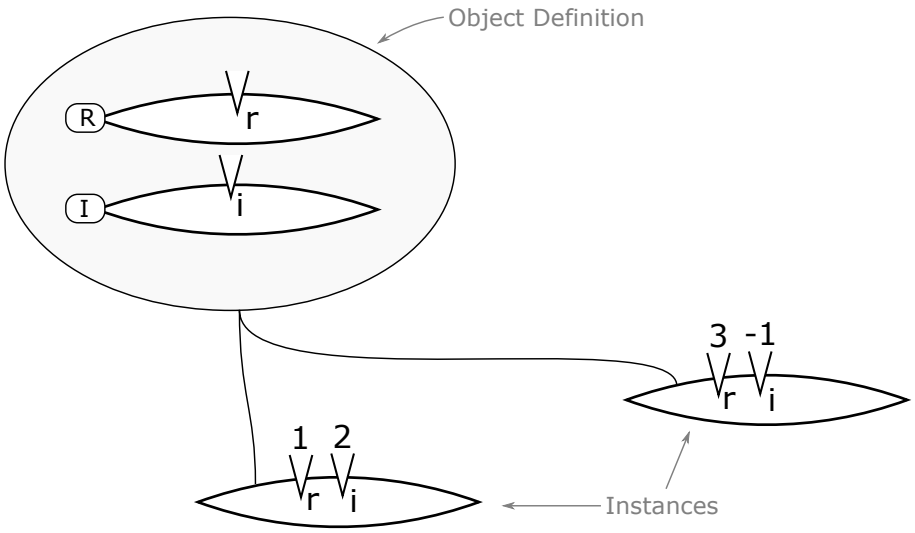

Figure 7.2. Using a group to define an object. In this example, the group is used to define the form of a complex number. It contains two components, 'R' and 'I', representing the real and imaginary parts of the number. The group is used to create two instances, which inherit the inputs of the node's components, whose values can then be set.

There are also likely to be benefits in representing relational and graph data at graph level. For relational data, one avenue of further work could be to establish whether multi-dimensional tables could play a part in representing normalised relational data at graph level.

7.2.4. Labelling

One of the possible applications of grouping is to define unordered collections, as described in Section 7.2.2.2. Labels could be used in the same way, with the benefit that, whereas groups are mutually exclusive, labels are not. Each node could have more than one label, allowing labels to define overlapping collections of nodes. As with groups, the unordered collections defined this way could be used to aggregate nodes to be applied to an input. Users could also label nodes as a way of categorising them, to make it easier to search for and find suitable nodes when needed. How this might work, and whether it turns out to be useful or efficient, remains to be seen.

7.2.5. Visualisations and Interactive Components

It would be useful for a node to be able to contain a generalised visualisation, to which data generated in the graph could be connected. Since the user interface is assumed to be web-based, the code defining a visualisation would have to run in a browser. This introduces slightly different practical privacy and security issues than would apply to other node functions, but that problem seems unlikely to be insurmountable. Functionality would have to be provided to enable creation of a visualisation while constraining its dimensions to the box representing the node in question and preventing it from accessing information about the rest of the graph.

An example might be if a node were created that took a table of data as an input and presented it as a graph. Figure 7.3 illustrates how such a visualisation might look. The

visualisation appears within a node, to which three data inputs have been connected, one providing the data and, in this case, two providing other parameters of the graph. Whenever new data arrived from any of the inputs, the graph would update correspondingly.

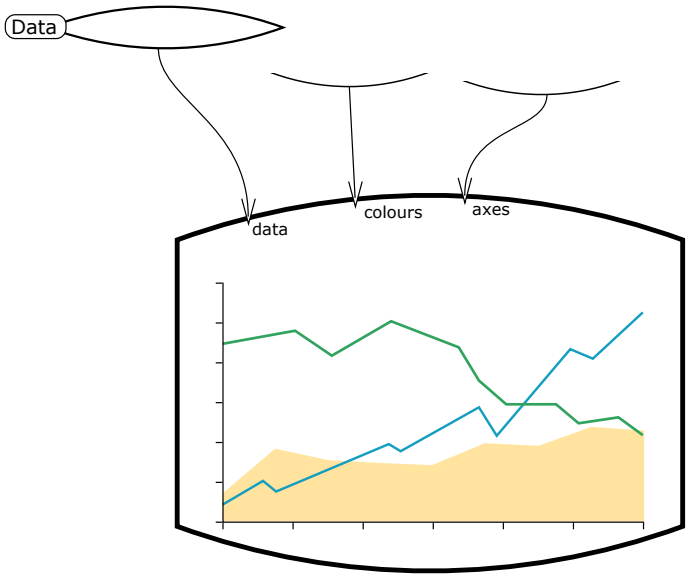

Figure 7.3. A visualisation node. In this example, the visualisation is a graph. It takes data as an input, but also allows its user to control the appearance of the graph by providing 'colours' and 'axes' as additional inputs.

It is possible that a graph containing many visualisations would quickly become cluttered. To avoid this problem, nodes could be made resizeable or collapsible, showing only a thumbnail representation of the visualisation when in collapsed form, but a full-screen view when needed. The precise form of interaction could be left to the designers of a user interface to decide.

An expansion of this concept would be the idea of adding interactive components. It is common for web-based graphs to alter the visualisation in response to a mouse being hovered or the graph touched or clicked. Interactive components could also have an output, so they could be made to output an edited version of their inputs, depending on the user's interactions.

To take this idea further, could an interactive component be used to override the native method of editing a node input? An interactive component for editing a number would have to use the existing value of the node as its starting value, but on editing would need to update the node with a new value. An interactive component of this kind occurs both before and after the underlying node in the node graph — taking the node's value as an input but also providing it with an output.

One way of representing this might be to think of interactive components as overlaying the underlying node. In Figure 7.4, we have an input node containing a percentage, set to the number '50'. We want a different interaction for it, so create an interactive component and overlay the existing node with the component. Once in the correct position overlaying the underlying node, the interactive component reads from the underlying node when first loaded and writes to it whenever a user interacts with the component, allowing the

resulting value to cascade through the rest of the graph as it does so.

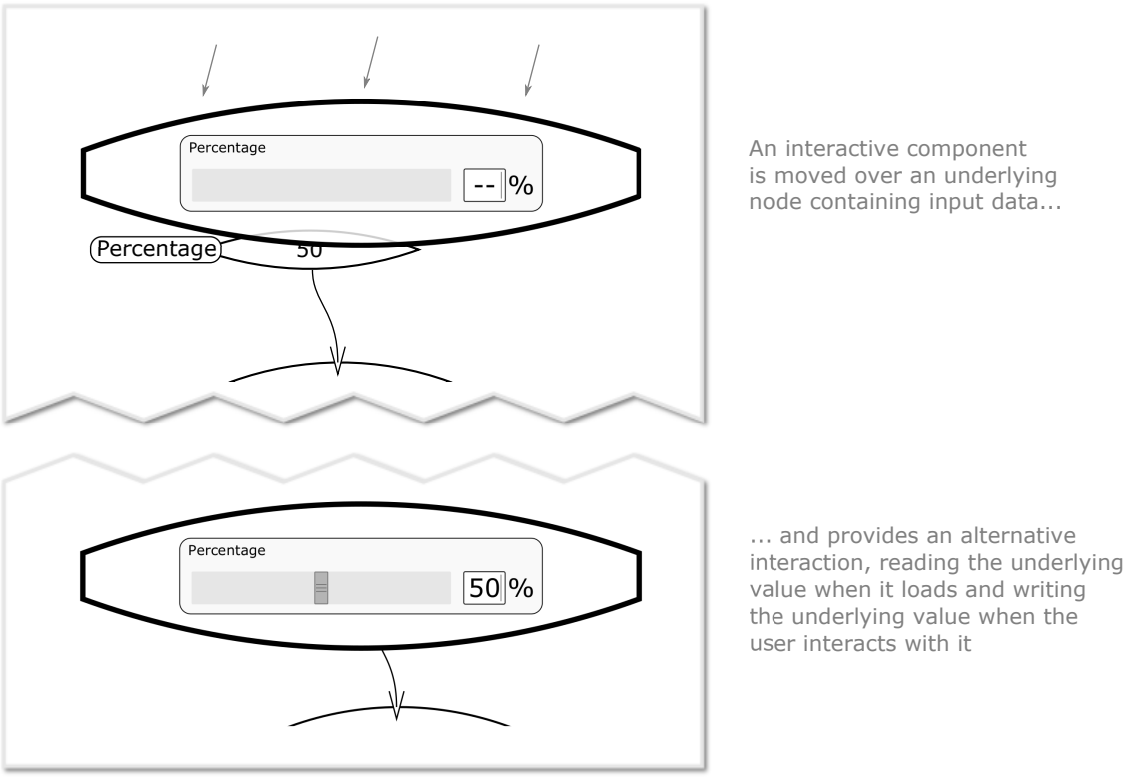

Figure 7.4. Use of interactive components to replace the interaction for an underlying node. The interactive component is moved over the underlying node, thereby replacing the native interaction for controlling its content with a new one.

7.2.6. Recursion

Although recursion is a form of iteration and can be represented in iterative form (Keller [1980]), it is an open question whether explicit recursion would be considered beneficial by users. If so, work would be needed to choose a manner of interaction that is easy to use and best complies with the principles of dataflow and software architecture.

The representation of recursion as iteration may also limit its potential parallelism. As discussed in Section 5.3.3 (The Merge-Sort Algorithm), the embedding of an independent iteration inside a dependent iteration has the effect of synchronising the execution such that each depth-level of an execution tree must be completed in full before execution of the next level can begin. In some cases this will not affect the total execution time, but in many cases it will slow it down. Whether it affects the total execution time, and by how much, depends on the functions in question. Investigation of recursion should therefore encompass investigation of the extent of this handicap and whether it can be eliminated by using a different approach or by implementing explicit recursion functionality.

7.2.7. Multi-Directionality

Multi-directionality was listed as an open problem in reactive programming (for which dataflow is a common programming paradigm) by Bainomugisha et al. [2013]. They provided examples of applications needing multi-directionality that included a unit conversion

(in which a change to either unit would trigger a change to the other) and pairs of user interface components required to maintain a fixed distance apart (in which one being dragged by the user would require the other to be moved correspondingly).

Another example would be where, in a user interface, a user is offered two different components to control the same underlying value. Figure 7.5 shows an example of such an interface in which a number, representing a percentage, can be edited using a slider or a text box.

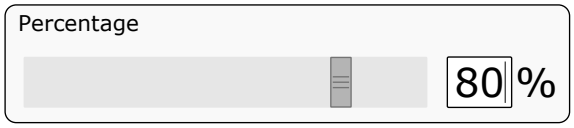

Figure 7.5. A component with two interactions controlling the same underlying value. When the user interacts with one, the other should also reflect the change.

Only two of the fifteen languages surveyed by Bainomugisha et al. had any form of multi-directional capability. Those that did achieved it by modelling the change as emanating from an event source. From the event, an action could be triggered (a change to the underlying number or a calculation of the unit conversion) which would in turn be reflected by the user interface components. The use of event sources side-steps the issue by using a uni-directional flow from an invisible component (the event source) to deliver behaviour that at first glance appears to be multi-directional. A similar approach could be used to achieve the appearance of multi-directionality in a dataflow system, perhaps connecting a node's input to an invisible event listener.

Another solution mentioned by Bainomugisha et al. was the approach of using switching constructs, which would enable the program to respond to an event by switching the direction of flow appropriately, so that information could flow from the object at which the event originated to the other components. This could be achieved using the manipulation of partially evaluated functions and graphs discussed in Section 7.2.1 (Manipulating Partially Evaluated Functions and Graphs). Neither approach has been directly addressed or tested as part of this work, so would require further investigation to fully resolve.

7.2.8. Optimisation

As pointed out in Chapter 6 (Evaluation and Results), the system currently suffers from a significant performance handicap when the test algorithms are compared with equivalent sequential implementations. This is unsurprising: the implementation was not designed to be optimal. There is significant potential for optimising the system, in both its software and hardware. In particular, there is potential to reduce the number of times functions are parsed and recompiled and data copied during iteration.

Dependent iteration is the source of much of the performance handicap suffered by dataflow; in particular the lack of destructive updating (as described by Vegdahl [1984]). In a sequential (control-flow) language, 'for' loops sit within a context, which contains a set of variables that can be modified by the iteration statement. The objective of the loop is to leave that context in the desired final state.

In dataflow, no such context exists; instead, the purpose of the loop is to produce an output value. This difference makes the outcome of an iteration clearer in dataflow than in control-flow, but also requires data to be copied in each iteration, making each iteration much less efficient than in control-flow loops. This is one of the reasons why iteration in dataflow has often been listed as an open problem (for example, by Johnston et al. [2004]).

Various approaches to solving this problem have been proposed, including by Nikhil et al. [1989], who devised a mechanism for this purpose named 'I-Structures', Giorgi et al. [2014], who used the concept of 'Transactional Memory' (originally devised by Herlihy and Moss [1993]) to ensure the integrity of data when using in-place updates and Harris et al. [2005], whose 'Composable Memory Transactions' involved explicitly distinguishing 'memory operations' (which could be safely re-executed if needed), from 'input/output operations' (which could not). An approach termed 'Compilation', combined with other techniques, also appears to have been successful at addressing this problem for The TERAFLUX project, with its DFScala language being amongst the first to have claimed objectively good performance rather than simply speed-up with increasing parallelisation (Giorgi et al. [2014]; Goodman et al. [2013]).

This document does not address efficiency. However, the ability of nodes to maintain state (described in Section 4.3 — What Are Nodes And Connections?), combined with the fact that dependent iterations take place sequentially in a predicable order, allow the possibility of making in-place data updates to a node's own state in each iteration rather than copying the data output each time. This would significantly reduce the computational overhead involved. It is also possible that the termination condition and iteration functions, which are the same in every iteration, could be allowed to persist between executions, avoiding having to reassemble them from their abstract syntax tree forms for each iteration.

A range of optimisations could also be applied to internal node functions during partial evaluation, using code analyses similar to those carried out by typical software development tools, code optimisers and compilers. For example, unnecessary computation could be avoided by trimming branches that no longer affect the output. Consider the function shown in Snippet 7.1: it uses a computationally expensive branch ('`let c = expensive(a)`') to set the variable 'c', which is used only if the condition '`a > 1`' is met. Imagine that, during partial evaluation, we provide the argument '`a`' with the value '`0`'. If the function '`expensive(a)`' does not cause external effects or modify the node's internal state, and if the variable 'c' is not subsequently used, then the expensive branch is redundant and can be removed without affecting the function's output. Alternatively, if, during partial evaluation, we provide the argument '`a`' with the value '`2`', the expensive branch can be computed in advance and replaced with its resulting value, reducing the amount that remains to be computed during the final (full) evaluation of the node.

Snippet 7.1

```
function(a,b){
  let c = expensive(a); // A computationally expensive
      function of `a'
  if a > 1 {
    return b + c;
  }
  else {
    return b + 1;
  }
}
```

Optimisations of this type, while potentially powerful, would have to account for the use of deliberately obsolete inputs as triggers (as described in Section 4.7 — Triggering Execution) and the resource priorities of node owners (including, potentially, a choice between eager and lazy evaluation, as discussed in Section 6.2 — Speed). The task of optimisation involves both the specification and the implementation of the system. Whereas the system can accommodate different implementations of its components, it is less flexible with regards to the specification, making it more important to consider the role of the specification in the optimisation of the system at an early stage.

7.3. Further Work on the Implementation

This section discusses the work that would be needed to build and test a more complete implementation of the features described in Chapter 4 (Definition), adding to the features that were implemented for testing and described in Chapter 5 (Implementation).

7.3.1. Distributability

The system was implemented with distributability in mind, but as yet runs on just one machine. The option to make it distributable was kept open through use of a separate component for node-to-node communications (described in Section 5.2 — Code Structure), and conversion of data and partially evaluated functions into a format, for transmission, that can be easily transmitted over a network.

In order for the system to be fully distributed, the main additional component needed is an addressing and identification system. In the current implementation, nodes are given an identifier that is unique with respect to the machine in question, but could clash with identifiers on other machines. A simple approach, sufficient for the purpose of testing, would be to identify nodes using an address for the host machine combined with that node identifier. Another approach, possibly suited to a production system, might involve allocating identifiers centrally and keeping a lookup table matching unique node identifiers to locations.

7.3.2. Component Interchangeability

Interchangeability is an architectural principle included in Section 3.4.2 (Increasing Flexibility), helping to deliver the principles of flexibility and incrementality (Section 3.4.1 — Uncovering Knowledge); and it is an attribute of the system discussed in Chapter 4

(Definition). Interchangeability would require a full definition of the interfaces between the system components to enable components to be switched for alternatives.

7.3.3. Expected Inputs

The 'Expected Inputs' functionality, described in Section 4.9 (Expected Inputs), provides the ability for a programmer to designate a particular input as expecting to receive a partially evaluated function, and to name the inputs it expects that incoming function to have. This provides flexibility by making it possible to choose the point in the graph at which to connect that expected function. This functionality is required, in particular, for the generalised iteration node described in Section 4.12 (The Generalised Iteration Node).

It could also be used to make a small change to the merge-sort algorithm described in Section 5.3.3 (The Merge-Sort Algorithm) to enable it to accept a sorting function as an input — which would be used to determine how the list should be sorted. The sorting function would compare two items and determine which should appear first in the list; any transitive relation could be used for this purpose.

The expected inputs functionality was partially implemented, but not fully integrated with the dependent iteration functionality (described in Section 4.11.2 — Dependent Iteration). Integrating the expected inputs feature fully with the rest of the system would also enable the generalised iteration node described in Section 4.12 (The Generalised Iteration Node) to be built.

7.3.4. Function Isolation

It is important when running untrusted code that the system and other nodes are protected from malicious or accidentally harmful behaviour. Section 4.13 (Function Isolation) describes a method of isolating untrusted code which also enables untrusted third-party languages to be added, using a parser within a parser. For the purpose of testing, all code is trusted, and this functionality has not been needed.

The implementation included a skeleton of the required functionality, in which functions written in JavaScript are parsed, converted into their Abstract Syntax Tree (AST) form, and compiled back into a function for execution. However, the parser used defines a version of JavaScript which does not curtail any of JavaScript's unsafe capabilities. Although it does demonstrate the use of the mechanism of defining a language and parsing code, it does not make untrusted code safe to run. The next step would be to create a parser that defines a safe language. It could be a version of JavaScript, as was used, or a version of any other language. Since JavaScript is widely used and relatively easy to learn, it makes a good candidate.

7.3.5. Tables

Tables were implemented as far as necessary for the purpose of building the test algorithms, using implicit creation of dimensional data. A fuller implementation would provide an explicit mechanism for creating dimensional data. This would require a 'list' data-type

for use inside functions, together with supporting functionality for list manipulation. The function language would also benefit from the addition of other, richer, table-manipulation functionality.

7.3.6. Visualisability, Interactivity, Availability

As described in Section 6.1 (Programmability), experimentation revealed that the system needs visualisability, interactivity and availability in order to maximise its programmability. Delivering these requires a user interface connected to an always-on back end. Subscriptions, described in Section 5.1.7, are a useful component in this, providing the ability for the user interface itself to subscribe to the nodes it is displaying at any point in time, receive data updates as they happen, and thereby providing the necessary interactivity. It would enable a user to make a change to one node and see the effect cascade through the graph as updated outputs became available.

7.3.7. A Directory of Components

In a globally distributed dataflow coordination system, there are huge benefits to be had from the use of nodes created by others, but it needs a way for programmers to find them. A directory and search system would be needed to help programmers find relevant nodes. As with the world wide web, for which search engines are useful but not integral to the system, there is no reason why a directory should necessarily be integral to the system. Although it would be beneficial for any particular user interface to be integrated with a directory through which useful nodes could be found when needed, the directory and search mechanism could be provided independently.

7.3.8. Further Testing

Additional tests are needed to assess aspects of the system beyond those covered by the test algorithms described in Chapter 5 (Implementation). To assess programmability we might be interested, for example, in how long it takes a user to create a selection of test algorithms, and their reported experience of doing so. We might also be interested in the balance of priorities between programmability and speed for various applications. That information would make it possible to prioritise optimisation efforts to achieve those gains that would most enhance the usefulness of the system.

While it is important that design decisions should be driven by the principles of Software Engineering and Architecture described in Chapter 3 (Software Engineering), the principles themselves are not beyond question. They provide a shortcut to avoid unnecessarily granular testing of every decision, but they may be subject to refinement over time. The resulting system should still be subjected to a degree of testing against those overall aims of improving programmability, optimising the experience of using the system and enabling appropriate prioritisation of resources.

7.4. Summary

This chapter divides the further work into three areas, categorised as defining the full specification of the system, describing additional features that would add functionality, and refining, completing or improving the existing implementation.

Part of the purpose of this document is to argue for the creation of a unified global dataflow coordination system. In such a system, the principles of node autonomy and interchangeability mean the requirements imposed by the specification should be minimal, relating mostly to the communication between nodes and interactions between components of the system. Whereas the specification must be well thought through, the interchangeable components, such as the user interface and internal node implementation need only minimal implementations in order to provide the basic functionality; the rest can be left to the preferences and values of those who might wish to add their own improvements later.

The next chapter concludes by summarising the work that has been done, expressing again the vision of a unified global dataflow coordination system, and making an appeal to those involved to cooperate to create a single open non-profit dataflow system run for the benefit of its users, enabling the widest possible applicability such that, ideally, any computational component in the world can become a part.

Chapter 8

Conclusion

The goals of this work are:

- to argue for a unified global dataflow coordination system;
- to establish a link between the design of that system and the principles of software engineering and architecture;
- to design a dataflow system, drawing on previous ideas and adding new proposals where required (most notably using partial evaluation and data dimensionality to represent iteration in an acyclic graph);
- to implement and test components of the proposed system, using it to build a set of sample algorithms.

While the use of dataflow for coordinating components has been discussed before, the notion of creating a unified global dataflow coordination system is a novel contribution. Likewise, the notion of linking the design of such a system to the principles of software engineering (so that they become easy or unthinking for programmers to apply) is new; as is the use of partial evaluation and data dimensionality to represent iteration in an acyclic graph.

The vision of a unified global dataflow coordination system is first introduced in Chapter 1, and the rest of the document is based on this founding principle. The importance of design decisions being based on principles of software engineering is, similarly, introduced at the beginning and used throughout.

The principles of software engineering and architecture, described in Chapter 3, are inherently empirical and to some extent subjective, so are certainly open to question, and the consensus may change over time. However, the value of such principles being embedded in the system, so that their implementation becomes easy or automatic and unthinking, should be clear. As programming becomes easier and increasingly accessible to people with less formal training, the importance of embedding such principles can only increase.

There are two classes of component in the system: intrinsic features — those dictated

by the system such as the manner of interaction between nodes and the interfaces between system components; and interchangeable components, such as the user interface, the hardware and the internal node implementations, for which alternative components or implementations could be used in parallel within the same system. Additionally, decisions on prioritisation of costs and resources can in many cases be left to node autonomy (Section 4.3 — What Are Nodes And Connections?) and ultimately end users to decide: not all applications or users share the same priorities and the system must accommodate their choices.

The design of a dataflow system described in Chapter 4 (Definition) features, in particular, the novel use of partial evaluation and dimensional data to represent iteration in acyclic dataflow. Partial evaluation, sometimes by different names, has been mentioned occasionally in the literature (see Section 2.2 — Partial Evaluation). This document uses partial evaluation to make nodes more reusable, and to simplify the process of programming iteration in dataflow, both dependent (where an iteration depends on the results of the previous iterations) and independent (where all iterations in a loop can be executed simultaneously). It is suggested that this representation is simpler and more flexible than those that have been proposed previously.

The functionality was implemented and tested (described in Chapter 5 — Implementation) by using it to construct three test algorithms: calculating a factorial, generating terms of the Fibonacci sequence and performing a merge-sort. The test algorithms produced correct results for values within the limitations of the underlying programming language used.

The proposed further work, described in Chapter 7, is divided into three categories. The first is to write a full specification defining the protocols that determine the required behaviour of system components, and the interfaces through which the components will interact with each other and the outside world. The main obstacles are providing a mechanism for controlling side effects and node state, resolving the full range of subscription types needed and solving the problems of synchronisation caused by divergent clocks in a distributed system.

The second category of further work is to investigate additional features that would add functionality to the system. Areas for investigation include how to convey a graph as data, how grouping could be used to collapse segments of a graph, how to represent data structures, enabling visualisations within nodes and how performance could be optimised, particularly through the design of the specification.

The third category of further work is to implement the remaining features described in Chapter 4 (Definition) and conduct further tests of the resulting system. This includes looking for ways to test its programmability, usability and applicability to the widest possible range of applications. The experience of using the (text-based) implementation (described in Chapter 6 — Evaluation and Results) highlighted a number of characteristics that will be important to any later, fuller implementation of the system. These include, in particular, that the system should be visual, interactive and always-available; visual

because graphs are easy to understand visually, interactive because it makes it immediately clear when and where mistakes are made, and available to enable interconnectivity between distributed nodes.

Reiterating one of the goals of the project, the hope is that this work helps to advocate the creation of a unified global dataflow coordination system that could make programming in general easier, more widely available and more accessible to all. A crucial step in the realisation of such a system is the full specification of the protocol by which nodes will interact and their interfaces for connection to interchangeable components such as user interfaces and computing resources. With a minimal specification of node behaviour, components can be built and a system can start to be assembled that is flexible, distributable, extensible to current and future computing technology, indifferent to hardware architectures and able to accommodate the full range of user priorities. With an appropriate protocol and interface, any computing or storage resources could be plugged into it, any user interface attached, and new languages and types of hardware could be added as required.

Many attempts have been made in the past to build dataflow systems that are designed to run locally, integrating the computational as well as coordination components. However, what is needed in order to bring about a truly heterogeneous system is an open protocol, available to all, defined and managed not by a commercial organisation, but by a non-profit body run in the interests of its users, perhaps on something like the model of the World Wide Web Consortium[1]. To that end, I urge people working on dataflow in future not to add to the diversity of incompatible systems that already exist, but instead to work together on building and designing the parts of a system that can work as one, to make the vision of a unified global dataflow coordination system come true.

[1] `https://www.w3.org/`

Appendices

Appendix A

Nomenclature

The terms and abbreviations used in this document are given below in alphabetical order, together with their meanings.

API Application Programming Interface. The set of functions through which a programmer can interact with a system.

AST Abstract Syntax Tree. A function encoded as an object.

Autonomy See Node Autonomy

Cascade When information propagates down the graph from top to bottom, it is described as 'cascading' down the graph. This applies to data updates (Section 4.6), update notifications (Section 4.16) and inherited inputs (Section 4.8).

Computation Manager A node can request of the computation manager that a function should be executed. A node can make multiple simultaneous requests. Section 5.2.4.

Connection A conceptual link between the input or content of one node and the output of another. Section 4.3.

Content A node has content, which can be a function or other value. Section 4.3.

Derived Inputs Node inputs that are generated either by being inherited via a connection to an upstream node or by having been implied through iteration or input parameters. Sections 4.8 and 4.11.2.

Dimensions (of data) Data comes in units (zero-dimensional), lists (one-dimensional) and tables (zero or more-dimensional). In the case of tables, the table's shape is defined by a number of lists; each list contributes one dimension. Section 4.10.

Downstream Used to refer to any part of a graph that is in the chain of connections below (emanating from the output of) the node in question.

Escalate When information propagates up the graph from bottom to top, it is described as 'escalating' up the graph. This applies to subscriptions (Section 5.1.7).

Expected Dimensions An input parameter (named 'dimensions') that determines the maximum number of dimensions that the node's function is capable of processing. It may be an integer (0 or greater) or unlimited. Section 4.11.1.

Expected Inputs An input parameter used to specify that the programmer expects the value received at a particular input to be a function. It is a list of input names of the function that the programmer expects this input to receive. Section 4.9.

Function A set of instructions which takes one or more inputs, and may be written in any programming language that can be understood by some hardware.

Graph Or node-graph. A network of connected nodes.

Inherited Input When a node receives a partially evaluated function through one of its inputs, it inherits the inputs of the function it receives. Section 4.8.

ID Node A node with one input that provides its input, unchanged, as an output.

Input If a node's content is a function, the node has an input corresponding to each argument of that function. In addition, the node may generate additional (derived) inputs, depending on its input parameters and the values received from its upstream nodes. Section 4.3.

List A list is an ordered array of units of data. Section 4.10.

Name A node can have a name, which is purely to assist the programmer and has no other meaning. It does not need one, and names need not be unique.

Node An autonomous unit of functionality, which contains a function or data, and may also be a custodian of related data. It must obey set rules of behaviour and interaction with other nodes, but also has significant scope for autonomy. Section 4.3.

Node Autonomy This refers to the freedom of nodes to choose their behaviour and resource prioritisation based on the preferences of their owners or users. Section 4.3.

Nominated Input A connection parameter used to associate an expected input to one of the inputs of an incoming partially evaluated function. Section 4.9.

Notification (Update Notification) An update notification is a message sent by a node to its subscribers as soon as it becomes aware that its output value is out of date. Section 4.16.

Originating Node The node of origin for an input is the node at which that input is first defined, before being later inherited by its downstream nodes. Section 4.16.

Output Each node has an output value. If the node's content is a function, its output value will be the partially evaluated function or the computed result of executing that function. If its content is data, the output is that data. Section 4.3.

Parameters Inputs and connections have parameters, which determine how the node behaves. Sections 4.9 and 4.11.1.

Partial Evaluation If a node's content is a function and values are not provided for all of the inputs of that node, the node will partially evaluate the function, resulting in a new function which has one input for each input whose value was not provided. Section 4.8.

Root Inputs The set of a node's inputs corresponding to its function's arguments. Section 4.3.

Subscription Each node maintains a list of subscribers, and notifies those subscribers as soon as it becomes aware that its output is no longer up-to-date. A node may subscribe to any of its directly connected upstream nodes. Section 5.1.7.

Table A zero or more-dimensional grid of data. The shape of a table is defined by a list of lists. The number of lists determines the number of dimensions of the table. The items of each list form the column headings of the table. Section 4.10.

Unit (of data) Any item of data. Lists and tables are units of data containing units of data. Section 4.10.

Upstream Any part of the graph in the chain of connections above (arriving at an input of) the node in question.

Appendix B

Visual Notation

This appendix provides a quick reference for the visual notation used to depict node graphs. The features involved are described more fully in Chapter 4 (Definition). Most visual features are optional. Only those relevant to the discussion at hand are included in diagrams; obvious or irrelevant details are usually left out.

Nodes

Nodes are depicted as a lens-shape (Figure B.1), in which inputs are received at the top of the shape and outputs delivered at the bottom. The node's name, if applicable, is shown as a label to the left of the node. A convention is used in this document that node names start with an upper case letter. See Section 4.3.

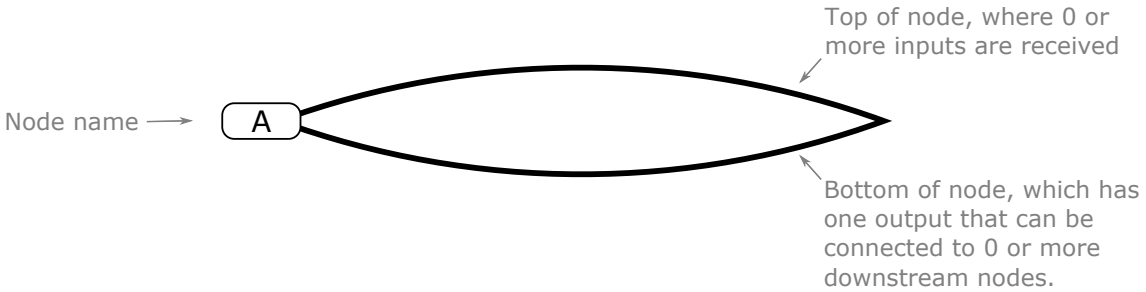

Figure B.1. Lens-shaped depiction of a node. The node's name, if applicable, appears in a label to the left. Input connections are depicted arriving in the top half of the node and output connections leave from the bottom.

Node Content

Nodes have 'content' and an output value, either or both of which may be shown within the node. Where both are included, the content is shown at the top and output at the bottom, separated by a dashed line. In order to ensure sufficient space, nodes can be depicted in either shallow or deep form, depending on the space needed (Figure B.2). A node may be elongated as far as necessary to contain its content and output. See Section 4.3.

Node Inputs

Inputs are depicted as a 'V' shape overlaid in the top part of the node, and labelled with a name if necessary (Figure B.3). A convention is used in this document that input names start with a lower case letter. See Section 4.3.

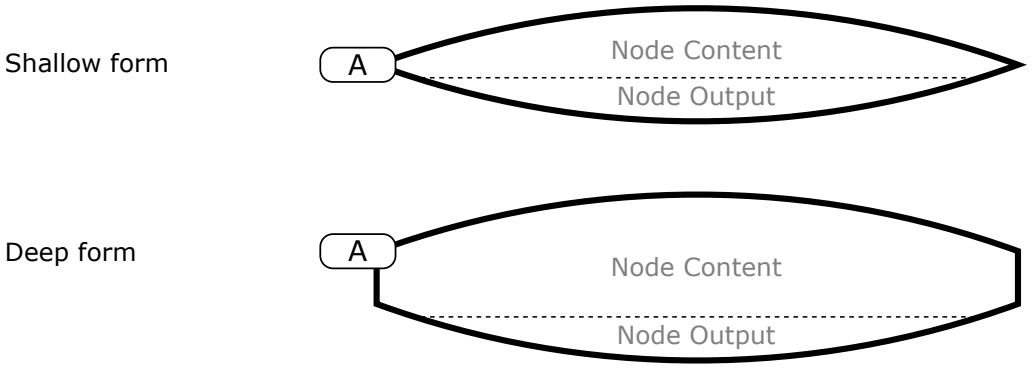

Figure B.2. Shallow and deep node notation. The node's height is elongated as far as necessary to accommodate the required content, input labels and output values.

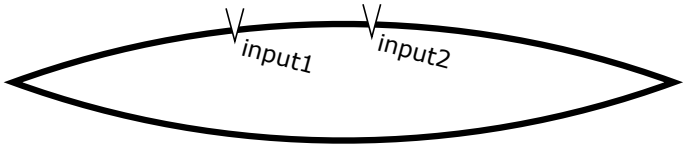

Figure B.3. Node inputs. Inputs are depicted as a 'V' shape in the top part of the node, with an adjacent input name if applicable.

Node Values and Connections

Inputs are provided with a value directly, or can be connected to another node's output. (Figure B.4). Connections between nodes are shown as lines connecting the output (the bottom part) of one node to an input of another. Flow in the graph is always from top to bottom. See Section 4.3.

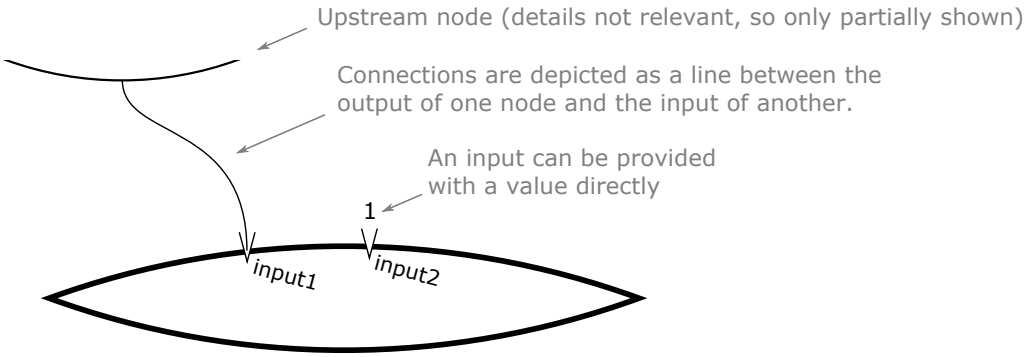

Figure B.4. Providing inputs with values. An input can be provided with a value by being connected to an upstream node, depicted as a line connecting the output of another node to the input; or an input can be provided with a value directly, depicted by writing the value above the input.

Root and Derived Inputs

Derived inputs (described in Sections 4.8 and 4.11.2) are shown as smaller than root inputs, and shaded, as shown in Figure B.5. See Sections 4.8, 4.9 and 4.11.2.

Input Parameters

Two different parameters can be set for an input, named 'expected dimensions' and 'expected inputs'. They are shown in a box connected to the corner of the input in question (Figure B.6). Where the 'expected inputs' list has been set, a derived input will appear for each expected input. See Sections 4.8, 4.9 and 4.10.

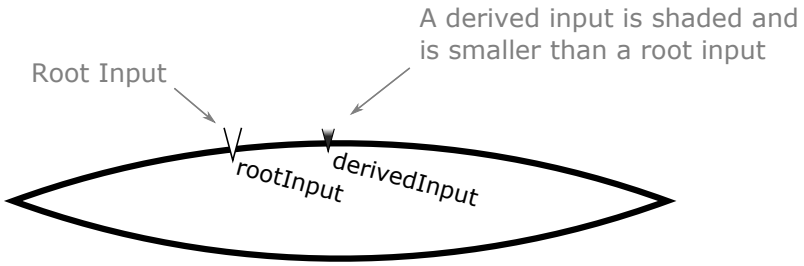

Figure B.5. Derived inputs. A derived input appears when a partially evaluated function arrives via a connection, when an input has expected inputs, or when a node is set to iterate over an input.

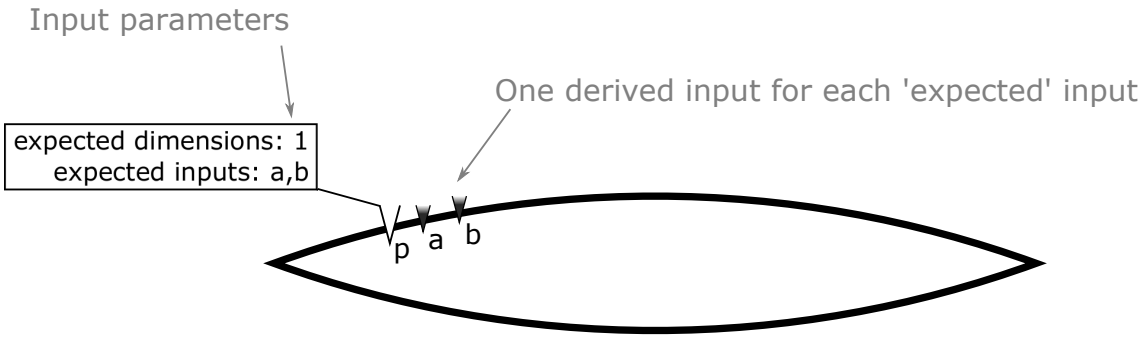

Figure B.6. Input parameters are shown in a label attached to the input they relate to. There are two possible types of input parameter: 'expected dimensions' and 'expected inputs'.

Connection Parameters

Connection parameters are shown in a box connected to the connection line. Each parameter is a 'nominated input', associating an expected input with an input of the incoming partially evaluated function. If the input of the incoming partially evaluated function can be uniquely identified by its input name alone, only its input name is used. Otherwise, its originating node and input name are both used, in curly brackets. Both formats are shown in Figure B.7. See Section 4.9.

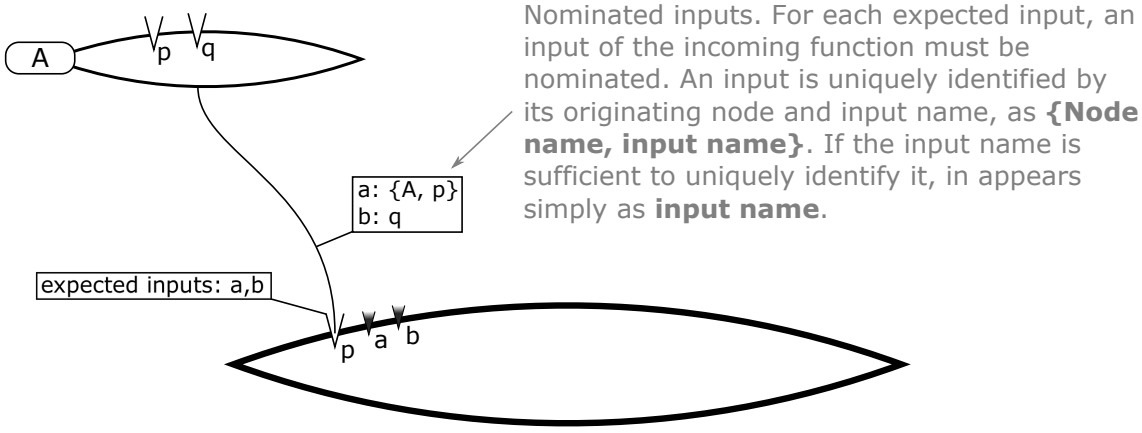

Figure B.7. Nominated inputs. When an input whose 'expected inputs' parameter has been set is connected to an upstream node, an input of the incoming partially evaluated function must be nominated to correspond to each expected input.

Inherited Content

A node may 'inherit' content by connecting it directly to an upstream node (that is, not via an input), making it, in effect, an instance of the upstream node. The inputs of the upstream node are inherited and are depicted in the inheriting node as root inputs. A

direct connection is depicted as a connection directly to the downstream node, as shown in Figure B.7. A node can only have one incoming connection of this type. See Section 4.8.

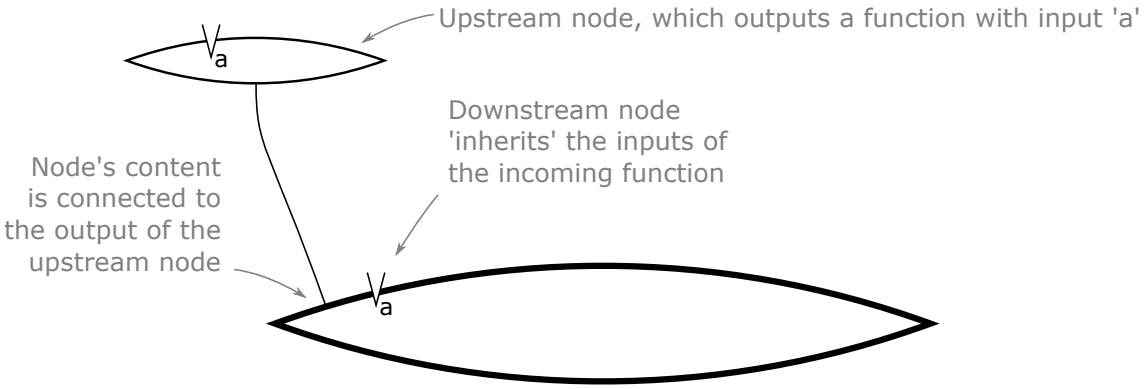

Figure B.8. Inherited content. Instead of setting a node's content, it can be set to inherit its content directly from another node. This is depicted by connecting an upstream node to some area on the top half of a node. The node will then inherit the inputs of the upstream node, all of which will be shown as root inputs, regardless of their status on the upstream node.

Tables

Tables are depicted as a grid, with the column headings shown with a grey background and the table data with a white background (Figure B.9). Tables of more than two dimensions are not depicted visually in this document. See Section 4.10.

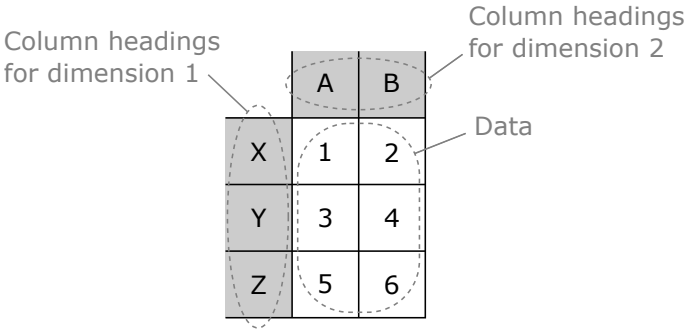

Figure B.9. Tables (of two dimensions) are depicted with column headings to the left and top, shaded grey, and the table data unshaded. Tables of three or more dimensions are not depicted visually in this document.

Dependent Iteration

Iteration is depicted in the graph with a 'rotation' symbol (🔄) appearing above an input, as shown in Figure B.10. A node can only iterate over one input. When an input has been designated as an iteration input, it can no longer receive a value and therefore cannot be connected to an upstream node. It will generate two additional (derived) inputs: '~sv' (the 'starting value') and '~tc' (the 'termination condition'). See Section 4.11.2.

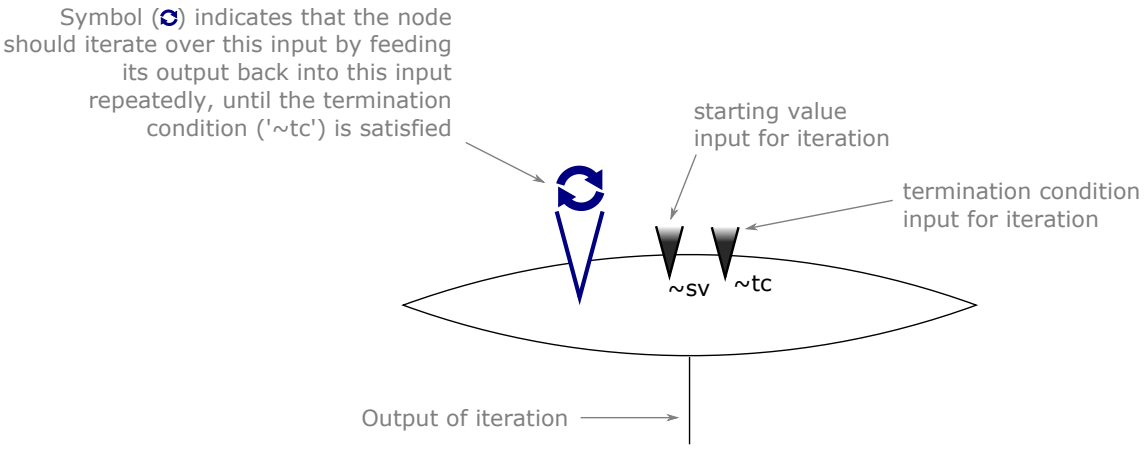

Figure B.10. Node iteration. A 'rotation' symbol appearing over an input denotes that a node will iterate over that input. When a node is iterating, two additional ('derived') inputs are generated in order to receive the starting value ('~sv') and termination condition ('~tc').

Appendix C

API Documentation

This appendix reproduces the README file included as part of the Mercurial Repository linked to this thesis[1].

Developments in Dataflow Programming — Test Implementation

Built by Daniel Maxwell, 2018.

Description

This repository contains an implementation of the dataflow system described in the PhD Thesis submitted by Daniel Maxwell to the University of Exeter in April 2018.

It contains the code defining nodes and the interactions between them, tests, and a set of three example algorithms, contained in the file test/mocha/bin/examples.js.

All implementation code is contained in the 'bin' directory and all test code is contained in the 'test' directory.

Requirements

- The implementation is written in and therefore requires the installation of NodeJS[2] (version 6.4.0 or later). It has been tested on Windows 7 and Linux. It should work (but cannot be guaranteed to) on other platforms.
- The code is stored in a Mercurial repository (`https://bitbucket.org/danieljmaxwell/developments-in-dataflow-programming`), so requires Mercurial[3] to create a clone

[1] The repository can be found at: `https://bitbucket.org/danieljmaxwell/developments-in-dataflow-programming` and the README file itself is at the top level, with the name *README.md*. It uses the Markdown format (described at `https://confluence.atlassian.com/bitbucketserver/markdown-syntax-guide-776639995.html`) and is displayed fully formatted on the repository's home page. This version relates to changeset '17d556be60d9'.

[2] `https://nodejs.org/`

[3] `https://www.mercurial-scm.org/`

of the project folder.

How to Install

- Install NodeJS[4] if you don't already have it.
- Install Mercurial[5] if you don't already have it.
- Clone the project folder from the repository:
- Open a command prompt in the location where you want the project folder to be installed and type:

```
hg clone https://bitbucket.org/danieljmaxwell/developments-in-
    dataflow-programming
```

- Open a command prompt in the project folder and install the required modules by typing:

```
npm install
```

How to Run the Tests

The tests, written in the Mocha[6] framework, are in the '*../test/mocha/*' directory. The structure of this directory reflects the structure of the top level project directory. The most important test files are '*./test/mocha/bin/api.js*', '*./test/mocha/bin/nodeDef.js*' and '*./test/mocha/bin/examples.js*'. Run them all by opening a command prompt in the top level of the project folder and typing:

```
cmd
npm test
```

The examples alone can be run using the command:

```
cmd
npm run-script examples
```

Any test can be run individually with the commands:

In windows:

```
cmd
node_modules\.bin\mocha ./file_path/test_file_name.js
```

In Unix-family systems:

```
cmd
node_modules/.bin/mocha ./file_path/test_file_name.js
```

[4] https://nodejs.org/
[5] https://www.mercurial-scm.org/
[6] https://mochajs.org/

How to use the API

You can use the API in interactive mode by starting NodeJS at the top level of the project directory with the command:

```
cmd
node
```

... and attaching the api object to a variable by typing:

```
const api=require('./bin/api');
```

This assigns an object containing the api functions to the 'api' constant. The 'api' object provides graph functions (relating to the graph as a whole) and allows you to create nodes. When creating a node, it will return a placeholder object which has a set of functions for operating on that node.

Synchronous Graph Functions

Synchronous graph functions are executed as soon as you call them and return their result.

api(findNode[,nominatedInputs]);
Returns a placeholder object, which provides a reference to a node that can be used to perform other functions on it.

Arguments

- findNode — a number or string containing the identifier or name of the node you want a reference to. It first tries to find a with an identifier matching the 'findNode' argument. If not found, it searches for a node with a name matching the 'findNode' argument.
- nominatedInputs — The inputs to be nominated if the node is connected to an input with expected inputs. It has the shape: 'expectedInput:incomingInput[,...]'. The 'incomingInput' can be a string specifying the name of the input on the incoming function to be nominated; if there is more than one input with the same name, it can use the fully qualified name as an array of the shape: '[originatingNode,inputName]'. The originating node is the node at which the the input 'inputName' is first defined. The nominated input parameter does not affect the underlying node, it only attaches to the placeholder object and is used only when using the placeholder object to connect an input to the node it refers to.

Example

```
api(0,{expectedInputName:'anInputOfNodeZero'});
```

or:

```
api(0,{expectedInputName1:'anInputOfNodeZero',expectedInputName2:'
    anotherInputOfNodeZero'});
```

or:

```
api(0,{expectedInputName:['originatingNode','anInputOfNodeZero']});
```

api.createNode([content[,options]]);

Creates a new node and returns a placeholder object, providing a reference to the newly created node.

Arguments

- content — The content of the node (optional). It can be a function, number, string, array or object.
- options — An object containing the options for the node (optional). Containing:
- name — a string defining the name of the node (optional).
- dimensions — an array of integers indicating, where content is a function, how many dimensions to expect for each argument of the function (optional). The default for any argument whose dimension is not specified is zero.
- expectedInputs — an array of arrays of strings indicating, where content is a function, for each argument, that a function is expected on that argument and listing the arguments that that function is expected to have (optional). eg. $[['a','b'],['c']]$ would indicate that we should expect a function on the first argument, with arguments named 'a' and 'b', and a function on the second argument with an argument named 'c'.

None of the node parameters have to be defined at the time of creation: the content and all options can be changed later if needed.

Example

```
api.createNode(1);
```

or:

```
api.createNode(
  (a,b)=>a[0] + b,{
    name:'myNode',
    dimensions:[1,0],
    expectedInputs:[undefined,['c']]
  }
);
```

api.deleteNodes([node[,node...]]);

Deletes the listed nodes and returns a list of those that did not exist. Include each node to be deleted as a separate argument.

Arguments

- node — A placeholder object or a string or number containing the ID of the node to be deleted.

Example

```
api.deleteNodes(0,1);
```

or:

```
let nodeZero=api(0);
api.deleteNodes(nodeZero);
```

api.graphStatus();

Returns an object containing a list of the nodes in the tasks and the number of tasks outstanding.

Example

```
api.graphStatus();
```

Asynchronous Graph Functions

Asynchronous functions are queued and executed in the order in which you call them. They return a JavaScript Promise, can be used to trigger an action or obtain a result when an asynchronous task has finished. To do this, follow the function with '.then(functionToBeTriggered)'.

Example

```
anAsyncFunction().then(result=>{console.log(result);});
```

See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises` for more about how to use promises.

api.listNodes();

Returns a promise which resolves to an array of the (string) identifiers of the nodes that exist.

Example

```
api.listNodes()
  .then(result=>{console.log(result);});
```

api.ready();

Returns a promise which resolves to 'undefined' when the tasks already in the queue have been completed.

Example

```
api.ready()
  .then(()=>{console.log('ready');});
```

Node Functions

Node functions are attached to the placeholder object:

```
let placeholder = api.createNode(0);
placeholder.setContent(1);
```

Those that perform actions return a Promise that has all other node functions attached to it, so that they can be chained. For example:

```
placeholder.setTitle('newNode').setContent(1);
```

Synchronous Node Functions

placeholder.instance();

Simulates the creation of an instance of the node by creating an ID node with the input 'input', and connecting it to the node on which this function is called, and returns a reference to the new 'instance'.

Example

```
let myNode = api.createNode();
let myInstance = myNode.instance();
```

placeholder.node();

Returns a reference to the node on which this function is called. This is useful to add to the end of an asynchronous function if you want to obtain a reference to the node to assign to a variable.

Example

```
let nodeA = api.createNode(a=>2*a).setConnections([nodeX]).node();
```

placeholder.nominateInput(input);

When using a placeholder to connect an input with just one 'expected input' to a node (for example, the ' tc' input in an iterative node), this specifies the name of the nominated input of the node being connected. It can use the just name or can identify the input in question by using its node of origin and name. If just the name is used and the node has more than one input with that name, it will use the first input it finds with that name.

Arguments

- input — the name (string) of the input being nominated or an array containing, in order, the originating node and name of the input being nominated.

Example

```
let myPlaceholder = api(0).nominateInput('anInputOfNodeZero');
```

or:

```
let myPlaceholder = api(0).nominateInput(['originatingNode','
    anInputOfNodeZero']);
```

placeholder.nominateInputs(inputs);

When using a placeholder to connect an input with one or more 'expected inputs', this can be used to nominate the input to be used for each. It can use just the name of each or can identify each input by using its node of origin and name. If just the name is used and the node has more than one input with that name, it will use the first input it finds with that name.

Arguments

- inputs — an object containing, for each expected input, the name of the input being nominated or an array containing, in order, the originating node and name of the input being nominated.

Example

```
let myPlaceholder = api(0).nominateInputs({
  expectedInputName1:'anInputOfNodeZero',
  expectedInputName2:'anotherInputOfNodeZero'
});
```

or:

```
let myPlaceholder = api(0).nominateInputs({
  expectedInputName:['originatingNode','anInputOfNodeZero']
});
```

Asynchronous Node Functions

Asynchronous functions are queued and executed in the order in which you call them. They return a JavaScript Promise, can be used to trigger an action or obtain a result when an asynchronous task has finished. To do this, follow the function with '.then(functionToBeTriggered)'.

Example

```
anAsyncFunction().then(result=>{console.log(result);});
```

See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises` for more about how to use promises.

The Promises returned by node functions have the other node function attached to them, allowing them to be 'chained'. They are executed in the order in which they appear in the chain.

Example

```
let myNode = api.createNode();
let nodeValuePromise = myNode.setTitle('a').setContent(1).getValue();
nodeValuePromise.then(result=>{console.log(result)});
```

placeholder.calculateTop([value1[,value2,...]]);

Obtain the output of the given node by executing it with the given root input values. Returns a promise which resolves to the output of the node.

Arguments

- value1, value2, ... — The values to set the corresponding root inputs to, or nodes to connect them to, in order.

Example

```
let myNode = api.createNode(input=>input+1);
myNode.calculateTop(1); // Returns a promise which resolves to the
    value 2.
```

placeholder.calculateWith([value1[,value2,...]]);

Obtain the output of the given node by executing it with the given input values. Returns a promise which resolves to the output of the node.

Arguments

- value1,value2,... — The values to set the inputs to or nodes to connect them to. Each value is an array of the form '[inputName,value]' or '[origin,inputName,value]', where 'origin' is a placeholder object or string indicating the reference of the node at which that input originates (only needed if the input cannot be uniquely identified from the input name), 'inputName' is a string containing the input being set, and 'value' is the value it should be set to. All inputs must be set in order for the node to fully evaluate. Otherwise, it will output a partially evaluated function.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.calculateWith(['a',1],[myNode,'b',2]); // Returns a promise
    which resolves to the value 3.
```

placeholder.deleteNode();

Deletes the node referred to by the placeholder.

Example

```
let myNode = api.createNode();
myNode.deleteNode();
```

placeholder.getConnections();

Returns a list inputs whose values have been set, together with the values or connections that they have been set to.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.top(1);
myNode.getConnections(); // Returns a list containing input 'b' but not
     input 'a'.
```

placeholder.getContent();

Returns a promise that resolves to the content of the node.

Example

```
let myNode = api.createNode(1);
myNode.getContent(); // Returns a promise that resolves to the value 1.
```

placeholder.getCurrentInputs();

Returns a promise that resolves to a list of the node's inputs, together with the 'origin', 'consumables', 'dimensions' and 'expectedInputs' values for each. It will not fetch up-to-date values from connected nodes, and will return only its current knowledge about its inherited inputs.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.getCurrentInputs(); // Returns a promise that resolves to a list
     showing its current knowledge of its root and inherited inputs.
```

placeholder.getInputs();

Returns a promise that resolves to a list of the node's inputs, together with the 'origin', 'consumables', 'dimensions' and 'expectedInputs' values for each. If some inputs are connected to upstream nodes, it will fetch up-to-date values for them in order to ensure it correctly lists all inherited inputs.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.top(1);
myNode.getInputs(); // Returns promise that resolves to a list
    containing all inputs, including up-to-date inherited inputs.
```

placeholder.getOpenInputs();

Returns a promise that resolves to a list of the node's open (unconnected) inputs, together with the 'origin', 'consumables', 'dimensions' and 'expectedInputs' values for each.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.top(1);
myNode.getOpenInputs(); // Returns a promise that resolves to a list
    containing input 'b' but not input 'a'.
```

placeholder.getRootInputs();

Returns a promise that resolves toa list of the node's root inputs, together with their 'origin', 'consumables', 'dimensions' and 'expectedInputs' settings.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.getRootInputs(); // Returns a promise that resolves to a list
    containing both inputs but would not display inherited inputs.
```

placeholder.getStatus();

Returns a promise that resolves to the current status of the node. The status includes its name, content, list of arguments, current value and 'isOn' value (indicating whether it remains active regardless of whether it has subscribers).

Example

```
let myNode = api.createNode();
myNode.getStatus();
```

placeholder.getTitle();

Returns a promise that resolves to the current name/title of the node.

Example

```
let myNode = api.createNode(1,{'name':'node1'});
myNode.getTitle(); // Returns a promise that resolves to 'node1'.
```

placeholder.getValue();

Calculates the value of the node and returns a promise that resolves to the resulting value.

Example

```
let myNode = api.createNode(1);
myNode.getValue(); // Returns a promise that resolves to the value 1.
```

placeholder.setConnections([value1[,value2,...]]);

Sets the inputs of the given node to values or connects them to other nodes. Returns a promise that resolves to a list of the rejected connections when done. Connections are rejected if the input in question does not exist.

Arguments

- value1,value2,... — The values to set the inputs to or nodes to connect them to. Each value is an array of the form '[inputName,value]' or '[origin,inputName,value]', where 'origin' is a placeholder object or string indicating the reference of the node at which that input originates (only needed if the input cannot be uniquely identified from the input name), 'inputName' is a string containing the input being set, and 'value' is the value it should be set to. All inputs must be set in order for the node to fully evaluate. Otherwise, it will output a partially evaluated function.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.setConnections(['a',1],['b',2]);
```

placeholder.setContent(content);

Sets the content of the node. Returns a promise that resolves to 'undefined' when done.

Arguments

- content — a function or other value to set the content to.

Example

```
let myNode = api.createNode();
myNode.setContent(1);
```

placeholder.setTitle(name);

Sets the name/title of the node. Returns an promise that resolves to 'undefined' when done.

Arguments

- name — a string containing the new name of the node.

Example

```
let myNode = api.createNode();
myNode.setTitle('newName');
```

placeholder.top([value1[,value2,...]]);

Sets the root inputs of the given node with the given values. Returns a promise that resolves to 'undefined' when done.

Arguments

- value1,value2,... — The values to be set or nodes (placeholder objects) to connect the inputs to.

Example

```
let myNode = api.createNode((a,b)=>a+b);
myNode.top(1,2); // Returns a promise that resolves to the value 3.
```

Appendix D

Examples

This appendix provides the code used to implement the three examples described in Chapter 5. The full implementation can be found in a Mercurial repository[1]. The tests use the mocha test framework combined with the assertion testing module provided by NodeJS[2]. The Mocha framework wraps its tests in two outer functions — the first one called 'describe', and within that one called 'it', as shown in Snippet D.1.

```
describe('Description of item being tested',function(){
  it('Description of the test',function(){
    const a = 1;
    assert.strictEqual(a,1,'string to be displayed if test fails'
        );
  });
});
```

In these tests, the 'describe' function contains the three specific algorithm tests as shown in Snippet D.2.

```
describe('Algorithm tests', function(){
  it('merge-sort', function(){...});

  it('factorial', function(){...});

  it('fibonacci sequence', function(){...});
});
```

Mocha handles asynchronism by allowing the test either to call a function ('done()') or to return a JavaScript Promise. In the examples shown below, the 'Promise' method is used, in most cases using the 'Promise.all()' function to return a single promise that completes when all promises created within the test have also completed.

[1] The repository can be found at: `https://bitbucket.org/danieljmaxwell/developments-in-dataflow-programming` and the test file that runs the code shown in this appendix is at the relative path: *test/mocha/bin/examples.js*. The version of the code printed here relates to changeset '17d556be60d9'.

[2] Mocha is available at: `https://mochajs.org/` and details of the assertion testing module in NodeJS can be found at: `https://nodejs.org/api/assert.html`

The code stored in the repository defines an API, which the tests below use to build graphs. The API is defined in the file *bin/api.js*, and its usage is described in the file *README.md* (which is also included with this document in Appendix C).

The three tests are shown separately below, in Snippet D.3 (Merge-Sort, starting immediately below), Snippet D.4 (Factorial, starting on page 180) and Snippet D.5 (Fibonacci Sequence, starting on page 181). To verify the functionality is as expected and to enable debugging, tests for individual nodes are also included within the code.

Merge-Sort Test

```
1  it("merge-sort",function(){
2    // This test should not finish until all promises have been completed.
         Here we create an array to contain the list of promises.
3    const promises = [];
4
5    //----------------- Creating the graph --------------------//
6
7    // The graph uses objects of the form {pairs,remainder}
8    // First we need components to manipulate these objects.
9
10   // A component to get the list of pairs from an object
11   const getPairs = api.createNode(obj => {return obj.pairs});
12
13   // Test the getPairs node works
14   promises.push(
15     getPairs.calculateWith(["obj",{pairs:[[1,2],[3,4]],remainder
           :[5,6,7]}]).then((result)=>{
16         assert.strictEqual(JSON.stringify(result),"[[1,2],[3,4]]");
17     })
18   );
19
20   // Create a node to get the remainder from the object
21   const getRemainder = api.createNode(obj => obj.remainder);
22
23   // Test the getRemainder node works
24   promises.push(
25     getRemainder.calculateWith(["obj",{pairs:[[1,2],[3,4]],remainder
           :[5,6,7]}]).then((result)=>{
26       assert.strictEqual(JSON.stringify(result),"[5,6,7]");
27   }));
28
29   // In order to unify these two different inputs called 'obj', we
         connect both to a single ID node with the same input.
30
31   // First create an id node
32   const inputObj = api.createNode(obj => obj);
33
34   // Then connect both previous nodes to it.
35   getPairs.top(inputObj);
36   getRemainder.top(inputObj);
37
38   // We need to create a new pair from an array, using the array and
         desired size of each member of the pair
39   const makeAPair = api.createNode((arr, size) => {
```

```
40      // If a function expects to receive one-dimensional data, it could
            either be an array or come in the form of a table.
41      // This function checks which it is and returns the data as a simple
            array.
42      function getArrData(tableOrArr){
43        if ("data" in tableOrArr) {
44          // In this case tableOrArr is a table, and we return the array
                data from within it;
45          return tableOrArr.data[0];
46        } else {
47          // In this case tableOrArr was an array to begin with, so we can
                simply return it.
48          return tableOrArr;
49        }
50      }
51      arr = getArrData(arr);
52      return [arr.slice(0, size),arr.slice(size, size * 2)]
53    }, {dimensions: [1]});
54
55    // Test the makeAPair node
56    promises.push(makeAPair.calculateTop([1,2,3,4,5],2).then((result)=>{
57      assert.strictEqual(JSON.stringify(result),"[[1,2],[3,4]]");
58    }));
59
60    // We also need to be able to work out the new remainder after taking a
          new pair away from it.
61    const newRemainder = api.createNode((arr, size) => {
62      // If a function expects to receive one-dimensional data, it could
            either be an array or come in the form of a table.
63      // This function checks which it is and returns the data as a simple
            array.
64      function getArrData(tableOrArr){
65        if ("data" in tableOrArr) {
66          // In this case tableOrArr is a table, and we return the array
                data from within it;
67          return tableOrArr.data[0];
68        } else {
69          // In this case tableOrArr was an array to begin with, so we can
                simply return it.
70          return tableOrArr;
71        }
72      }
73      arr = getArrData(arr);
74      return arr.slice(size * 2)
75    }, {dimensions: [1]});
76
77    // Test the newRemainder node
78    promises.push(newRemainder.calculateTop([1,2,3,4,5,6],2).then((result)
          =>{
79      assert.strictEqual(JSON.stringify(result),"[5,6]");
80    }));
81
82
83    // We now need to connect the makeAPair node and the newRemainder node
          to the rest of the graph.
```

```
84  // But they both have a 'size' input that we need to be unified. So we
        now create a new node for that and connect them all up.
85
86  // First we create the new ID node with a 'size' input.
87  const inputSize = api.createNode(size => size);
88
89  makeAPair.top(getRemainder, inputSize);
90  newRemainder.top(getRemainder, inputSize);
91
92  // Test the makeAPair and newRemainder nodes again, this time with an
        object input.
93  promises.push(makeAPair.calculateWith(["obj",{remainder:[1,2,3,4,5]}],[
        "size",2]).then((result)=>{
94    assert.strictEqual(JSON.stringify(result),"[[1,2],[3,4]]");
95  }));
96  promises.push(newRemainder.calculateWith(["obj",{remainder
        :[1,2,3,4,5]}],["size",2]).then((result)=>{
97    assert.strictEqual(JSON.stringify(result),"[5]");
98  }));
99
100
101 // We need a node to append a new pair to the existing pairs list
102 const newPairsList = api.createNode((existingPairs, newPair) => {
103
104   // If a function expects to receive one-dimensional data, it could
        either be an array or come in the form of a table.
105   // This function checks which it is and returns the data as a simple
        array.
106   function getArrData(tableOrArr){
107     if ("data" in tableOrArr) {
108       // In this case tableOrArr is a table, and we return the array
            data from within it;
109       return tableOrArr.data[0];
110     } else {
111       // In this case tableOrArr was an array to begin with, so we can
            simply return it.
112       return tableOrArr;
113     }
114   }
115   existingPairs = getArrData(existingPairs);
116   newPair = getArrData(newPair);
117
118   existingPairs.push(newPair);
119   return existingPairs;
120 }, {dimensions: [1, 1]});
121
122
123 // Test the newPairsList node
124 promises.push(newPairsList.calculateTop([[[1,2],[3,4]]],[[5,6],[7,8]]).
        then((result)=>{
125   assert.strictEqual(JSON.stringify(result),"
        [[[1,2],[3,4]],[[5,6],[7,8]]]");
126 }));
127
128 // Connect the newPairsList node into the graph
```

```
129  newPairsList.top(getPairs, makeAPair);
130
131  // Test the newPairsList node with its connections to the rest of the
         graph
132  promises.push(newPairsList.calculateWith(
133    ["obj",{pairs:[[[1,2],[3,4]]],remainder:[5,6,7]}],
134    ["size",2]
135  ).then((result)=>{
136    assert.strictEqual(JSON.stringify(result),"
           [[[1,2],[3,4]],[[5,6],[7]]]");
137  }));
138
139  // takes the new list of pairs and the remainder and combines back into
         an object of shape {pairs,remainder}
140  // Once connected, this takes an object of shape {pairs,remainder},
       then makes a new object in which one extra pair is moved to pairs
       from remainder.
141  const combineIntoObj = api.createNode((pairs, remainder) => {
142    // If a function expects to receive one-dimensional data, it could
         either be an array or come in the form of a table.
143    // This function checks which it is and returns the data as a simple
         array.
144    function getArrData(tableOrArr){
145      if ("data" in tableOrArr) {
146        // In this case tableOrArr is a table, and we return the array
             data from within it;
147        return tableOrArr.data[0];
148      } else {
149        // In this case tableOrArr was an array to begin with, so we can
             simply return it.
150        return tableOrArr;
151      }
152    }
153    pairs = getArrData(pairs);
154    remainder = getArrData(remainder);
155    return {pairs, remainder}
156  }, {dimensions: [1, 1]});
157
158  combineIntoObj.top(newPairsList, newRemainder);
159
160  // Test the connected combineIntoObj node
161  promises.push(combineIntoObj.calculateWith(
162    ["obj",{pairs:[[[10,11],[12,13]],[[14,15],[16,17]]],remainder
           :[18,19,20,21,22,23,24,25]}],
163    ["size",2]
164  ).then((result)=>{
165    assert.strictEqual(JSON.stringify(result),"{\"pairs
           \":[[[10,11],[12,13]],[[14,15],[16,17]],[[18,19],[20,21]]],\"
           remainder\":[22,23,24,25]}");
166  }));
167
168  // Here we set this object to iterate by connecting the 'obj' input to
         itself.
169  combineIntoObj.setConnections(["obj",combineIntoObj]);
170
```

```
171   // Test the combineIntoObj node by checking the number of inputs.
172   promises.push(combineIntoObj.getOpenInputs().then((result)=>{
173     assert.strictEqual(result.length,3,"this object should have three
          inputs");
174   }));
175   promises.push(combineIntoObj.instance().getOpenInputs().then((result)
        =>{
176     assert.strictEqual(result.length,3,"this object should have three
          inputs");
177   }));
178
179
180   // With the iteration, and the tc and sv set, it should move all of the
        remainders into pairs
181   combineIntoObj.setConnections(
182     ["~sv",{pairs:[],remainder:[1,2,3,4,5,6,7,8,9]}],
183     ["size",2],
184     ["~tc",api.createNode((obj)=>{
185       return !obj.remainder.length
186     })]
187   );
188
189   // proving that it does...
190   promises.push(combineIntoObj.getValue().then((result)=>{
191     assert.strictEqual(result.pairs.length,3,"There should be three '
          pairs' in the result");
192     assert(result.remainder.length===0,"There should be nothing in the
          remainders");
193   }));
194
195   // takes an obj of shape {pairs,remainder} and a size and returns an
        object in which ALL the remainder items have been moved to pairs of
        the given size
196   const objIntoPairs = combineIntoObj.setConnections(
197     ["~sv",api.createNode(obj=>obj)],
198     ["size",undefined]
199   ).instance();
200
201   // Takes an obj of shape {pairs,remainder} and a size outputs an array
        of arrays where each sub-array is now in order.
202   // Intended to take a table (dim 1) of such objects and then return a
        table (dim 1) of merged pairs
203   const mergePair = api.createNode((pairs)=>{
204     const result = [];
205     if(!Array.isArray(pairs)) {pairs = pairs.data[0]};
206
207     pairs.forEach((pair)=>{
208       while (pair[0].length || pair[1].length) {
209       let takeLeft;
210       if(pair[1][0]===undefined) {takeLeft=true};
211       else if(pair[0][0]===undefined) {takeLeft=false};
212       else {takeLeft = (pair[0][0]<pair[1][0])};
213       result.push(takeLeft?pair[0].shift():pair[1].shift());
214       }
215     });
```

```
216     return result;
217   }).setConnections(["pairs",getPairs],["obj",objIntoPairs]).node();
218
219   const doubleSize = api.createNode(size=>size*2).top(inputSize).node();
220
221   // Takes an obj of shape {pairs,remainder} and a size, and outputs an
          obj of shape {arr,size}, where size is doubled and arr is the
          remainders, having been split and recombined
222   const recombined = api.createNode((arr,size)=>{
223     return {arr,size};
224   }).top(mergePair,doubleSize);
225
226   const arrSize = api.createNode(obj=>obj);
227   const getRemainderObj = api.createNode(obj=>{
228     return {pairs:[],remainder:obj.arr};
229   }).top(arrSize).node();
230   const getSize = api.createNode(obj=>obj.size).top(arrSize).node();
231
232   // Now it takes an object of shape {arr,size} and returns one of the
          same shape after being processed
233   recombined.setConnections(["obj",getRemainderObj],[inputSize,"size",
          getSize]);
234
235   const iterationNode = recombined.instance();
236
237   const startingValue = api.createNode(arr=>{
238     return {arr:arr.data[0],size:1};
239   },{dimensions:[1]});
240   const terminationCondition = api.createNode(obj=>{
241   if(!obj.arr) {return true};
242   else {return obj.size>=obj.arr.length})};
243   iterationNode.setConnections(
244     ["obj",iterationNode],
245     ["~sv",startingValue],
246     ["~tc",terminationCondition]
247   );
248
249   // We now, finally, create the desired merge-sort node
250   const mergeSortNode = api.createNode(obj=>obj.arr).top(iterationNode);
251
252   // We have two alternative tests to demonstrate it works - a long test
          and a short test.
253   // The long test uses a set of 100 randomly generate arrays, which are
          loaded from file.
254   // The short test uses just arrays of just 12 members. A switch
          variable, longTest, is used to decide which to run.
255
256   let unsortedArr = [];
257   let sortedArr = [];
258   if(longTest) {
259     this.timeout(0);
260     unsortedArr = require(path.join(process.cwd(),'test','mocha','bin','
          Unsorted.json'));
261     sortedArr = require(path.join(process.cwd(),'test','mocha','bin','
          Sorted.json'));
```

```
262   } else {
263     unsortedArr = [[0,5,39,0,3,5,6,4,4,23,2,8]];
264     sortedArr = ["[0,0,2,3,4,4,5,5,6,8,23,39]"];
265   }
266
267   let testCounter=0;
268   // Now define a function to test every array in turn
269   function nextTestValues(){
270     if(testCounter<unsortedArr.length){
271       mergeSortNode.setConnections(["arr",unsortedArr[testCounter]]);
272
273       return mergeSortNode.getValue().then((result)=>{
274         assert.strictEqual(JSON.stringify(result),sortedArr[testCounter],
              "merge-sort test array " + (testCounter+1));
275         testCounter++;
276         return nextTestValues();
277       });
278     }
279   }
280
281   // Run the test and store the result in the promises array.
282   promises.push(nextTestValues());
283
284   return Promise.all(promises);
285 });
```

Factorial Test

```
1  it("factorial",()=>{
2
3    // We start with a node containing a variable that combines both the
         value (val) and the counter (num).
4    const combinedInput = api.createNode(combObj=>combObj);
5
6    // We then create two nodes to separate out the components. getVal gets
          the value from the combined object...
7    const getVal = api.createNode(combObj=>combObj.val).top(combinedInput).
         node();
8
9    // and getNum gets the counter ('num') from the combined object.
10   const getNum = api.createNode(combObj=>combObj.num).top(combinedInput).
         node();
11
12   // The computation of the next value is found by multiplying the
         previous value by the new counter. In this line, we create the node,
          set its connections and then return the newly created node to be
         stored in the variable 'newVal'.
13   const newVal = api.createNode((oldVal,oldNum)=>{return oldVal * oldNum
         }).top(getVal,getNum).node();
14
15   // The new counter is the previous counter incremented by one.
16   const newNum = api.createNode(oldNum=>oldNum+1).top(getNum).node();
17
18   // The value and counter are then recombined into a single object.
         Again, we create the node, set its connections in the same line, and
          this time return an instance of it to the variable.
```

```
19    const recombinedInput = api.createNode((val,num)=>{
20      return {val,num};
21    }).top(newVal,newNum).instance();
22
23    // The iteration terminates when the counter ('num') reaches the number
          we want the factorial of.
24    const tc = api.createNode((factOf,output)=>{
25      return output.num>factOf
26    }).nominateInput("output");
27
28    // We set the recombinedInput node to iterate, and at the same time set
          its starting value and termination condition inputs.
29    recombinedInput.setConnections([combinedInput,"combObj",recombinedInput
          ],["~sv",{val:1,num:1}],["~tc",tc]);
30
31    // Finally, we create the factorial node, which takes one input, '
          factOf'.
32    const factorial = api.createNode(obj=>obj.val).top(recombinedInput);
33
34    // Here we test it with the highest number for which the result is a
          number that can be represented exactly by JavaScript.
35    return factorial.calculateWith(["factOf",21]).then((result)=>{
36      assert.strictEqual(result,5109094217170944000,"The factorial should
            be returned correctly");
37    });
38  });
```

Fibonacci Sequence Test

```
1   it("fibonacci sequence",()=>{
2
3     // An array to hold the promises for testing
4     const promises = [];
5
6
7     // This node appends the next value to the end.
8     const iterate = api.createNode(arr=>{
9       if("data" in arr) {arr = arr.data[0]};
10      if(!Array.isArray(arr)) {
11        arr = []
12      } else {
13        arr.push((arr.length<2)? arr.length : (arr[arr.length-1] + arr[arr.
              length-2]));
14      }
15      return arr;
16    },{dimensions:[1]});
17
18    // Test the iterate node - this verifies that it appends the correct
          value to the end of the example array
19    promises.push(iterate.calculateTop([1,1,2,3]).then((result)=>{
20      assert.strictEqual(JSON.stringify(result),"[1,1,2,3,5]");
21    }));
22
23    // Connect the 'iterate' node's input to itself, to make it iterate.
24    iterate.top(iterate);
25
```

```
26    // Define a node to act as the termination condition. This returns true
          when the length of the array equals n.
27    const tc = api.createNode((arr,n)=>arr.length>=n).nominateInput("arr");
28
29    // Set the starting value to be an empty array and the termination
          condition to be the node we have just defined.
30    iterate.setConnections(["~sv",[]],["~tc",tc]);
31
32    // Calculate it with n (the desired length of the array) set to 10.
33    promises.push(iterate.calculateWith(["n",10]).then((result)=>{
34      assert.strictEqual(JSON.stringify(result),"[0,1,1,2,3,5,8,13,21,34]",
            "Should correctly generate the first ten terms of the Fibonacci
            sequence");
35    }));
36
37    return Promise.all(promises);
38  });
```

Bibliography

Ackerman, W. B. and Dennis, J. B. (1979). VAL–A Value-Oriented Algorithmic Language (Preliminary Reference Manual). Technical report, Massachusetts Inst Of Tech Cambridge Lab For Computer Science.

Aczél, J. J. and Daróczy, Z. (1975). On measures of information and their characterizations. Technical report, New York: Academic Press.

Adams, D. A. (1969). A computation model with data flow sequencing. *Doctoral Dissertation, Stanford University*.

Agerwala, T. and Arvind, N. I. (1982). Data flow systems. *Computer*, 15(2):10–14.

Aitken, A. and Ilango, V. (2013). A Comparative Analysis of Traditional Software Engineering and Agile Software Development. *2013 46th Hawaii International Conference on System Sciences*, pages 4751–4760.

Arvind, A., Gostelow, K., and Plouffe, W. (1977). Indeterminacy, monitors, and dataflow. *ACM SIGOPS Operating Systems Review*, 11(5):159–169.

Arvind, A. and Gostelow, K. P. (1982). The U-interpreter. *Computer*, 15(2):42–49.

Asanovic, K., Wawrzynek, J., Wessel, D., Yelick, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., and Sen, K. (2009). A view of the parallel computing landscape. Technical Report 10, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Auguston, M. and Delgado, A. (1997). Iterative constructs in the visual data flow language. In *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No.97TB100180)*, pages 152–159. IEEE.

Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S., and De Meuter, W. (2013). A survey on reactive programming. In *ACM Computing Surveys (CSUR)*, volume 45, page 52. Citeseer.

Baroth, E. and Hartsough, C. (1995). Visual programming in the real world. In *Visual Object-Oriented Programming*, pages 21–42. Manning Publications Co.

Barth, P. S., Nikhil, R. S., and Others (1991). M-structures: extending a parallel, non-

strict, functional language with state. In *Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer.

Basili, V. R. and Rombach, H. D. (1988). The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on software engineering*, 14(6):758–773.

Basili, V. R. and Turner, A. J. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, 1.

Beck, K. (1999a). Embracing Change with Extreme Programming. *IEEE Computer*, 32(10):70–77.

Beck, K. (1999b). *Extreme Programming Explained: Embrace Change*. Pearson Education.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., and Others (2001). Manifesto for Agile Software Development. http://agilemanifesto.org/.

Benington, H. D. (1956). Production of Large Computer Systems. *Symposium on Advanced Programming Methods for Digital Computers*, 5(4):299–310.

Bernini, M. and Mosconi, M. (1994). VIPERS: a data flow visual programming environment based on the Tcl language. *Proceedings of the workshop on Advanced Visual Interfaces*, pages 243–245.

Boehm, B. (1976). Software Engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241.

Boehm, B. (2006). A View of 20th and 21st Century Software Engineering. *Proceedings of the 28th International Conference on Software Engineering SE - ICSE '06, Shanghai, China*, pages 12–29.

Boehm, B. W. (1986). A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–42.

Bravo, M., Li, Z., Van Roy, P., and Meiklejohn, C. (2014). Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM.

Burnett, M., Atwood, J., Djang, R. W., Gottfried, H., Reichwein, J., and Yang, S. (2001). Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Functional Programming*, 11(2):155–206.

Caporuscio, M., Funaro, M., and Ghezzi, C. (2012). PaCE: a data-flow coordination language for asynchronous network-based applications. In *International Conference on Software Composition*, pages 51–67. Springer.

Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

Cave, W. C. and Salisbury, A. B. (1978). Controlling the Software Life Cycle – The Project Management Task. *IEEE Transactions on Software Engineering*, SE-4(4):326–334.

Chapman, B., Haines, M., Mehrotra, P., Zima, H., and Van Rosendale, J. (1997). Opus: A coordination language for multidisciplinary applications. *Scientific Programming*, 6(4):345–362.

Chlipala, A. (2015). Ur/Web: A Simple Model for Programming the Web. In *POPL '15 The 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 153–165, Mumbai, India. Association for Computing Machinery (ACM).

Cooper, J. (1981). MIL-STD 1679 , Weapon System Software Development. *The Journal of Systems and Software*, 2:319–327.

Cox, P. T., Giles, F. R., and Pietrzykowski, T. (1989). Prograph: a step towards liberating programming from textual conditioning. In *Visual Languages, 1989., IEEE Workshop on*, pages 150–156. IEEE.

Curry, H. B., Feys, R., Craig, W., Hindley, J. R., and Seldin, J. P. (1972). *Combinatory logic*, volume 2. North-Holland Amsterdam.

Darlington, J. and Reeve, M. (1981). ALICE a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 65–76. ACM.

Davis, A. L. (1978). The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 210–215. ACM.

Davis, A. L. and Keller, R. M. (1982). Data flow program graphs. *IEEE Computer*, 15(2):26–41.

Davis, A. L. and Lowder, S. A. (1981). A sample management application program in a graphical data driven programming language. *Digest of Papers Compcon Spring*, 81:162–167.

Dennis, J. (1974). First version of a data flow procedure language. In *Robinet B. (eds) Programming Symposium. Lecture Notes in Computer Science, vol 19.*, pages 362–376. Springer, Berlin, Heidelberg.

Dennis, J. (1980). Data Flow Supercomputers. *Computer*, 11:48–56.

Dennis, J. B. and Misunas, D. P. (1975). A Preliminary Architecture for a Basic Data-Flow Processor. *ACM SIGARCH Computer Architecture News*, 3(4):126–132.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional.

Forrester, J. W. (1961). *Industrial Dynamics.* Cambridge: MIT Press.

Gajinov, V., Stipić, S., Erić, I., Unsal, O. S., Ayguadé, E., and Cristal, A. (2014). DaSH: a benchmark suite for hybrid dataflow and shared memory programming models. In *Proceedings of the 11th ACM Conference on Computing Frontiers - CF '14*, pages 1–11. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26.

Garlan, D. and Shaw, M. (1993). An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, 1(January):1–39.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 35(2):96.

Ghittori, E., Mosconi, M., and Porta, M. (1998). Designing new programming constructs in a data flow VL. In *Proceedings of VL*, page 78, Nova Scotia, Canada. IEEE.

Gibbs, W. W. (1994). Software's Chronic Crisis. *Scientific American*, 271(3):86–95.

Giddings, R. V. (1984). Accommodating Uncertainty in Software Design. *Communications of the ACM*, 27(5):428–434.

Gilb, T. (1981). Evolutionary Development. *ACM SIGSOFT Software Engineering Notes*, 6(2):17.

Gilbreth, F. B. and Gilbreth, L. M. (1921). Process charts-first steps in finding the one best way. *American Society of Mechanical Engineers (ASME), New York, NY*.

Giorgi, R., Badia, R. M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G. R., Garbade, A., Gayatri, R., Girbal, S., Goodman, D., Khan, B., Koliaï, S., Landwehr, J., Lê, N. M., Li, F., Lujàn, M., Mendelson, A., Morin, L., Navarro, N., Patejko, T., Pop, A., Trancoso, P., Ungerer, T., Watson, I., Weis, S., Zuckerman, S., and Valero, M. (2014). TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8):976–990.

Goodman, D., Khan, S., Seaton, C., Guskov, Y., Khan, B., Luján, M., and Watson, I. (2013). DFScala: High level dataflow support for scala. *Proceedings - 2012 2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing, DFM 2012*, 1(249013):18–26.

Gurd, J. R., Kirkham, C. C., and Watson, I. (1985). The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52.

Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM.

Herlihy, M. and Moss, J. E. B. (1993). *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM.

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101.

Holthouse, M. A. and Greenberg, S. G. (1978). Software Technology for Scientific and Engineering Applications. In *Computer Software and Applications Conference*.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411.

Hughes, R. J. M. (1982). Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10. ACM.

International Organization for Standardization (ISO) (1985). ISO 5807: 1985 Information Processing-Documentation Symbols and Conventions for Data, Program and System Flowcharts, Program Network Charts and System Resources Charts. *Geneva: ISO*.

Johnson, D. M. (1996). The systems engineer and the software crisis. *ACM SIGSOFT Software Engineering Notes*, 21(2):64–73.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34.

Jungnickel, D. (2008). *Graphs, Networks and Algorithms*. Springer, 3rd edition.

Kahn, K. M. and Miller, M. S. (1988). Language design and open systems. *The Ecology of Computation. Elsevier Science Publishers BV (North-Holland)*.

Karp, R. M. and Miller, R. E. (1966). Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411.

Keller, R. M. (1980). *Semantics and applications of function graphs*. Book published by University of Utah, Department of Computer Science.

Keller, R. M. and Yen, W.-C. J. (1981). A graphical approach to software development using function graphs. In *IEEE COMPCON*, volume 81, pages 23–26.

Kruchten, P. (1995). The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):9.

Kruchten, P. (1996). A Rational Development Process. *Crosstalk*, 9(7):11–16.

Kruchten, P., Obbink, H., and Stafford, J. (2006). The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30.

Kyriacou, C., Evripidou, P., and Trancoso, P. (2006). Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188.

Larman, C. and Basili, V. R. (2003). Iterative and incremental development (IID). *IEEE Computer*, pages 47–56.

Lee, B. and Hurson, A. R. (1993). Issues in Dataflow Computing. *Advances in Computers*, 37(C):285–333.

Lee, B. and Hurson, A. R. (1994). Dataflow Architectures and Multithreading. *Computer*, 27(8):27–39.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2014). *The Java Virtual Machine Specification — Java SE8 Edition*. Addison-Wesley Professional.

Lord, R., Millar, J., and Kahane, R. (1977). A Procedure for the Estimation of Software Development Costs. *Annual Review in Automatic Programming*, 8:211–227.

Madden, W. A. and Rone, K. Y. (1984). Design , Development , Integration : Space Shuttle Primary Flight Software System. *Communications of the ACM*, 27(914-925).

McGraw, J., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauert, J., Dobes, I., and Hohensee, P. (1983). SISAL: Streams and Iteration in a Single-Assignment Language. Language reference manual, Version 1.1. Technical report, Lawrence Livermore National Lab., CA (USA).

McGraw, J. R. (1982). The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82.

McLain, P. and Kimura, T. D. (1986). Show and Tell User's Manual Report Number: WUCS-86-04. *All Computer Science and Engineering Research*.

McPhillips, T., Bowers, S., Zinn, D., and Ludäscher, B. (2009). Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551.

Meier, J., Hill, D., Homer, A., Jason, T., Bansode, P., Wall, L., Boucher Jr, R., and Bogawat, A. (2009). *Microsoft Application Architecture Guide*. Microsoft Corporation.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall International.

Meyer, M. (2014). Continuous Integration and its Tools. *IEEE Software*, 31(3):14–16.

Morris, J. H., Schmidt, E., and Wadler, P. (1980). Experience with an applicative string processing language. *7th Principles of programming languages ({POPL})*, pages 32–46.

Morrison, J. P. (2010). *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, 2nd edition.

Mosconi, M. and Porta, M. (2000). Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2000):67–104.

National Instruments Corporation (1998). Labview User Manual.

Naur, P. and Randell, B. (1969). Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. In *NATO Software Engineering Conference*, page 231.

Nelson, E. (1967). Management Handbook for the Estimation of Computer Programming Costs.

Newman, P. (1997). PRINCE2. In *IEE Half-Day Colloquium on Professionalism in Project Management (Digest No: 1997/373)*, pages 2–3, London.

Nikhil, R. S., Pingali, K., and Arvind, K. P. (1986). *Id Nouveau*. Massachusetts Institute of Technology, Laboratory for Computer Science.

Nikhil, R. S., Pingali, K. K., and Others (1989). I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632.

Penczek, F., Herhut, S., Grelck, C., Scholz, S.-B., Shafarenko, A., Barrère, R., and Lenormand, E. (2010). Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2085–2094.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc.

Randell, B. (1996). The 1968 / 69 NATO Software Engineering Reports. In *Dagstul Seminar 9635 on History of Software Engineering*, pages 1–7.

Rasure, J. R. and Williams, C. S. (1991). An integrated data flow visual language and software development environment. *Journal of Visual Languages & Computing*, 2(3):217–246.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM.

Rodriquez, J. E. (1969). A Graph Model For Parallel Computations. Technical report, Massachusetts Inst Of Tech Cambridge Electronic Systems Lab.

Royce, D. W. W. (1970). Managing the Development of large Software Systems. In *IEEE Wescon*, pages 1–9.

Rumbaugh, J. (1977). A data flow multiprocessor. *IEEE Transactions on Computers*, 100(2):138–146.

Sabry, A. (1998). What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22.

Schwaber, K. (1995). SCRUM Development Process. In *OOPSLA Workshop on Business Object Design and Implementation*, pages 117–134.

Schwanke, R. W., Altucher, R. Z., and Platoff, M. A. (1989). Discovering, visualizing, and controlling software structure. *ACM SIGSOFT Software Engineering Notes*, 14(3):147–154.

Shaw, M. (1989). Larger scale systems require higher-level abstractions. *ACM SIGSOFT Software Engineering Notes*, 14(3):143–146.

Shaw, M. (1995). Comparing Architectural Design Styles. *IEEE Software*, 12(6):27–41.

Shaw, M. and Clements, P. (2006). The Golden Age of Software Architecture. *IEEE Software*, 23(2):31–39.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, volume 1. Prentice Hall, Inc.

Solinas, M., Badia, R. M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G. R., Garbade, A., Girbal, S., and Others (2013). The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 272–279. IEEE.

Søndergaard, H. and Sestoft, P. (1990). Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517.

Sousa, T. B. (2012). Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium in Informatics Engineering, University of Porto*.

Suárez, A. F. (2013). *Domain Specific Languages for High Performance Computing: A Framework for Heterogeneous Architectures*. PhD thesis, Universitat Politècnica de Catalunya (UPC) - BarcelonaTech.

Sutherland, W. R. (1966). *The on-line graphical specification of computer procedures*. PhD thesis, Massachusetts Institute of Technology.

Tolksdorf, R. (1998). Laura—A service-based coordination language. *Science of Computer Programming*, 31(2-3):359–381.

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. (1982). Data-driven and demand-driven computer architecture. *ACM Computing Surveys (CSUR)*, 14(1):93–143.

US Department of Defense (1972). Military Standard MIL-STD-1512.

US Department of Defense (1983). Military Standard DOD-STD-1679A.

US Department of Defense (1985a). Military Standard DOD-STD-2167.

US Department of Defense (1985b). Military Standard MIL-STD-483A.

US Department of Defense (1994). Military Standard MIL-STD-498.

Van Der Linden, F. J. and Müller, J. K. (1994). Creating Architectures with Building Blocks. *IEEE Software*, 12(6):51–60.

Vegdahl, S. R. (1984). A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, C-33(12):1050–1071.

Vogel, O., Arnold, I., Chughtai, A., and Kehrer, T. (2011). *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. Springer Science & Business Media.

Weaver, P. (2007). The Origins of Modern Project Management. In *Fourth Annual PMI College of Scheduling Conference*.

Whiting, P. G. and Pascoe, R. (1994). A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59.

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142.

Wirth, N. (1971). Program Development by Stepwise Refinement and Related Topics. *Communications of the ACM*, 14(4):221–227.

Wolverton, R. W. (1974). The Cost of Developing Large-Scale Software. *IEEE Transactions on Computers*, C-23(6):615–636.

Wozniak, J. M., Armstrong, T. G., Wilde, M., Katz, D. S., Lusk, E., and Foster, I. T. (2013). Swift/T: Scalable data flow programming for many-task applications. In *ACM SIGPLAN Notices*, volume 48, pages 309–310. ACM.

Yazdanpanah, F., Alvarez-Martinez, C., Jimenez-Gonzalez, D., and Etsion, Y. (2014). Hybrid Dataflow / von-Neumann Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1489 – 1509.

Young Jr, J. W. and Kent, H. K. (1958). An abstract formulation of data processing problems. In *Preprints of papers presented at the 13th national meeting of the Association for Computing Machinery*, pages 1–4. ACM.

Zachman, J. A. (1987). A Framework for Information Systems Architecture. *IBM Systmes Journal*, 26(3):454–470.

Zuckerman, S., Suetterlein, J., Knauerhase, R., and Gao, G. R. (2011). Using a codelet program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69. ACM.