A MATHEMATICAL THEORY OF SYNCHRONOUS CONCURRENT ALGORITHMS

Benjamin Criveli Thompson

Submitted in accordance with the requirements for the degree of Doctor of Philosophy

The University of Leeds
Department of Computer Studies

September, 1987

A MATHEMATICAL THEORY OF SYNCHRONOUS CONCURRENT ALGORITHMS

Benjamin Criveli Thompson

Submitted in accordance with the requirements for the degree of Doctor of Philosophy

September, 1987

ABSTRACT

A *synchronous concurrent algorithm* is an algorithm that is described as a network of intercommunicating processes or modules whose concurrent actions are synchronised with respect to a global clock. Synchronous algorithms include *systolic algorithms*; these are algorithms that are well-suited to implementation in VLSI technologies.

This thesis provides a mathematical theory for the design and analysis of synchronous algorithms. The theory includes the *formal specification* of synchronous algorithms; techniques for proving the correctness *and performance* or *time-complexity* of synchronous algorithms, and formal accounts of the *simulation* and *top-down design* of synchronous algorithms.

The theory is based on the observation that a synchronous algorithm can be specified in a natural way as a *simultaneous primitive recursive function over an abstract data type*; these functions were first studied by J. V. Tucker and J. I. Zucker. The class of functions is described via a formal syntax and semantics, and this leads to the definition of a functional algorithmic notation called *PR*. A formal account of synchronous algorithms and their behaviour is achieved by showing that synchronous algorithms can be specified in PR. A formal account of the performance of synchronous algorithms is achieved via a mathematical account of the time taken to evaluate a function defined by simultaneous primitive recursion.

A synchronous algorithm, when specified in PR, can be transformed into a program in a language called *FPIT*. FPIT is a language based on abstract data types and on the *multiple* or *concurrent assignment statement*. The transformation from PR to FPIT is phrased as a compiler that is proved correct; compiling the PR-representation of a synchronous algorithm thus yields a *provably correct simulation* of the algorithm. It is proved that FPIT is just what is needed to implement PR by defining a second compiler, this time from FPIT back into PR, which is again proved correct, and thus PR and FPIT are formally *computationally equivalent*. Furthermore, an autonomous account of the length of computation of FPIT programs is given, and the two compilers are shown to be *performance preserving*; thus PR and FPIT are computationally equivalent in an especially strong sense.

The theory involves a formal account of the top-down design of synchronous algorithms that is phrased in terms of correctness and performance preserving transformations between synchronous algorithms specified at different levels of data abstraction. A new definition of what it means for one abstract data type to be 'implemented' over another is given. This definition generalises the idea of a *computable algebra* due to A. I. Mal'cev and M. O. Rabin. It is proved that if one data type D is implementable over another data type D', then there exists correctness and performance preserving compiler mapping high level PR-programs over D to low level PR-programs over D'.

The compilers from PR to FPIT and from FPIT to PR are defined explicitly, and our compiler-existence proof is constructive, and so this work is the basis of theoretically well-founded software tools for the design and analysis of synchronous algorithms.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr. John Tucker for guiding me through a most interesting course of study. In the past four years John has provided me with knowledge and sound training that I know will serve me well in years to come; for this I will be grateful always. I also thank him for patient encouragement that kept me calm at times when I found the completion of this thesis a very great strain: I value his friendship.

I am indebted to parents, Anne and Alan, and to my sister Laura, for their love which has always been a source of strength.

I am grateful for the opportunity to thank Dr. Willi Riha for originally stimulating my interest in theoretical computer science. His infectious enthusiasm for the subject lead me to become a Ph. D. student, and this thesis is the result.

Thanks are also due to my fellow postgraduates Steven Eker, Neal Harman, Keith Hobley, Andy Martin, and Karl Meinke, for providing a stimulating environment in which to work. Special thanks are due to Clive Jervis for useful technical conversations, and to Keith Hobley for proof reading the technical chapters of this thesis in their entirety.

I am also pleased to acknowledge the financial support of a Science and Engineering Research Council research studentship and also a research assistantship (the latter under the Alvey project for information technology).

In memory of my Mother, Anne.

CONTENTS

In science nothing capable of proof ought to be accepted without proof.

Richard Dedekind

... anything which can be done will be done...

Kurt Vonnegut

CHAPTER 1
INTRODUCTION

Recent advances in computer hardware have led to the manufacture of devices comprising a (large) number of separate components that compute in parallel. The advantage that a concurrent device has over a sequential device is one of speed of course: it takes less time to solve a given task if different parts of the task can be tackled concurrently rather than one at a time. The advantage of using concurrent devices is offset however by the fact that it is difficult to design and analyse concurrent systems of processes (for the basic reason that it is difficult to analyse one part of a concurrent system whilst simultaneously keeping track of what is happening in other parts of the system).

The theory of sequential computation has led to greater understanding of what is solvable by means of a (sequential) computer, and complementary theories are now sought to surmount the conceptual difficulties involved in concurrent computation, and to thereby exploit the advantages offered by concurrency. Most of the *theoretical* research into concurrency is concerned with *asynchronous* concurrency: in an asynchronous concurrent system the components of the system behave autonomously with respect to each other. Significant examples of mathematical models of asynchronous computation are CCS (Milner[1980]), CSP (Hoare[1985]), and ACP (Bergstra and Klop[1986]); these are models with an 'arbitrary interleaving' semantics of concurrency.

This thesis concerns *synchronous* concurrency: in a synchronous concurrent system the components of the system are forced to compute and to communicate with each other in system-wide simultaneous steps. Whilst there is much interest in synchronous concurrency, much of the published research is concerned with (small families of) examples, and is conducted on an informal basis with little theoretical content. In this thesis we will present a *general theory* that encompasses the formal *specification* and *verification* of synchronous concurrent algorithms and upon which we build mathematical accounts of *simulation* and *top-down design*. The theory is applied to examples and to the design of software tools.

Very Large Scale Integration (VLSI) technologies, as described in Mead and Conway[1980] for example, support the implementation of both synchronous and asynchronous concurrent algorithms as digital devices. Such devices can be extremely complex and may contain in excess of five hundred thousand transistors (Beyers[1981]). Due to the availability and overwhelming complexity of VLSI systems, the need of structured design methodologies for VLSI is acute (Rem[1981], Mead[1983]) and this need has precipitated much research into formal methods for VLSI systems. Whilst this thesis is not specifically concerned with VLSI, synchronous VLSI systems are for us an important source of motivation and examples. Work on VLSI systems that is relevant to our research, in particular work on *systolic algorithms*, is discussed below in Section 1.2; for work not directly related to this thesis the reader is referred to the useful general bibliography Rosenberg[1985].

In the next section we will describe this thesis chapter by chapter. Afterwards we will review related research in two sections: in Section 1.2 we review other research that is concerned with the specification and design of computer hardware (VLSI); in Section 1.3 we describe some of the practical applications of our work in the context of a software engineering project under development here in the Department of Computer Studies at the University of Leeds.

1.1 OVERVIEW AND DISCUSSION.

In Chapter 2 we begin with an informal model of computation that we call *synchronous current algorithms* or 'synchronous algorithms' for short. A synchronous algorithm is an algorithm that is described as a network of intercommunicating processes or *modules* whose concurrent actions are synchronised with respect to a global clock T measuring discrete time $t = 0,1,2,....$ To begin with, a synchronous algorithm is described as processing data taken from a single set A. (In Chapter 3 we generalise the model by allowing A to be a *many-sorted* family of sets so that one synchronous algorithm can process many different kinds of data.) A module m comprises a *processor* and a *store*: the processor is capable of computing a *total* function $f_m : A^n \longrightarrow A$; the store is capable of holding a *single* datum from the set A. The function f_m *specifies* a module m in the following sense: whenever an n-tuple $a = (a_1,...,a_n)$ of input data is available to m, the module computes $f_m(a)$ and places this

value in its store. As case studies we will investigate two systolic algorithms for sorting, although it will become apparent that our synchronous algorithms are more general than systolic algorithms.

In Section 2.4 we take our first steps towards the formal specification of a synchronous algorithm or network: we show how from the communication structure of a synchronous network, and from the functional specifications of the network's modules, we can automatically construct a variety of functional specifications of the behaviour of the entire network. The most important of these functional specifications is a uniquely defined function that we call the network's *value function* which we will now explain.

Synchronous networks process infinite sequences or *streams* of data: we use a mapping $\underline{a} : T \longrightarrow A^n$ to represent the input to a network with $n > 0$ inputs; here $T = \{0,1,2,...\}$ is the clock with respect to which the network's modules are synchronised, and the intention is that $\underline{a}(t) \in A^n$ is the input available to the network at each time $t \in T$. In addition to input data, the behaviour of a synchronous network is dependent on the values initially held in the stores of the network's modules; that is, the values held in the network at time $t = 0$. Modules hold a single datum, and so we use a vector $x = (x_1,...,x_k) \in A^k$ to represent the values initially held by a network with $k > 0$ modules; here the intention is that x_i is the value held by the ith module of the network at time $t = 0$ for $i = 1,...,k$.

To specify the behaviour of a synchronous network over time, we need to specify the value held by each of the network's modules at each time $t \in T$, and this is the role played by the network's value function: if N is an n-input, k-module synchronous network over data set A, then N's value function is a map V_N of functionality

$$V_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^k$$

(Here '$[T \longrightarrow A^n]$' is the collection of all functions from T into A^n; generally '$[X \longrightarrow Y]$' is the notation we use for the collection of all functions with domain X and codomain Y.) For given arguments $t \in T$, $\underline{a} : T \longrightarrow A^n$, and $x = (x_1,...,x_k) \in A^k$, our interpretation of the expression '$V_N(t,\underline{a},x)$' is straightforward: '$V_N(t,\underline{a},x)$' is read as 'the values held in network N at time t when the input to the network is $\underline{a}$ and the initial values were $x_1,...,x_k$'. Notice that V_N is vector-valued: $V_N(t,\underline{a},x)$ is a vector in A^k with the intention that the ith coordinate of $V_N(t,\underline{a},x)$ is the value held by the ith module of N at time t for $i = 1,...,k$.

As we have described it, the vector x denotes the network's initial values, and so V_N always satisfies the equation

$$V_N(0,\underline{a},x) = x \qquad (1)$$

for any $\underline{a} : T \longrightarrow A^n$ and any $x \in A^k$. In fact, V_N satisfies this equation because (1) is how we *define* $V_N(t,\underline{a},x)$ at $t = 0$ (for any $\underline{a} : T \longrightarrow A^n$ and $x \in A^k$). To define $V_N(t,\underline{a},x)$ at subsequent times t we use a form of *primitive recursion*: *because of the network's synchronous operation, the values held by the network's modules at a time $t+1$ can always be effectively determined from the values held by the network's modules at time t.* We will explain how $V_N(t+1,\underline{a},x)$ is defined from $V_N(t,\underline{a},x)$ in due course. The important point to notice here is that for a given network N, its value function V_N tells us everything there is to know about the network's behaviour over time and so a mathematical theory of synchronous algorithms is provided by a formal account of 'value functions'.

In Chapter 3 we give a formal account of the *simultaneous primitive recursive functions generalised to an abstract data type*. These functions, first defined and studied in Tucker and Zucker[1987] (work of 1979), are the mathematical setting for the study of value functions.

We begin by modelling data and operations (ultimately those used by a synchronous algorithm) as a *many-sorted Σ-algebra* A; this A comprises some data sets $A_1,...,A_n$ and some operations on these sets, and Σ (the *signature* of A) is the syntax of A in the sense that it comprises *names* for the operations of A. In order to define the simultaneous primitive recursive functions we assume that A contains a copy of the natural numbers $\{0,1,2,...\}$, the successor function on this set, the Booleans $\mathbb{B} = \{tt,ff\}$, and the logical operations *not* and *or*; such algebras we call *standard*.

The class PR(A) of all simultaneous primitive recursive functions over a (Σ-) algebra A is built up from the operations of A by means of four function-building tools, viz *definition-by cases*, *sequential* and *parallel composition*, and *simultaneous primitive recursion*. Simultaneous primitive recursion allows us to define k functions $f_1,...,f_k$ simultaneously by primitive recursion from functions $g_1,...,g_k$ and $h_1,...,h_k$; that is, $f_1,...,f_k$ are defined by a system of equations of the form:

$$\begin{aligned} f_1(0,a) &= g_1(a) \\ &\vdots \quad \vdots \quad \vdots \\ f_k(0,a) &= g_k(a) \end{aligned}$$

and

$$f_1(t+1,a) = h_1(t,a,f_1(t,a),\ldots,f_k(t,a))$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$f_k(t+1,a) = h_k(t,a,f_1(t,a),\ldots,f_k(t,a))$$

Here $t \in T = \{0,1,2,\ldots\}$ and $a \in A^n$ for any $n > 0$.

PR(A) is officially defined via a syntax PR(Σ) (Σ the signature of A), and a formally defined semantics:

$$\mathrm{PR}(A) = \{ [\![\alpha]\!]_A : \alpha \in \mathrm{PR}(\Sigma) \}$$

wherein $[\![\,.\,]\!]_A$ is the semantic evaluation mapping that sends each $\alpha \in \mathrm{PR}(\Sigma)$ (a PR *scheme* over Σ) to a function on A. With this definition of PR(A), it should be clear that a function f on A is officially simultaneous primitive recursive over A iff there exists some $\alpha \in \mathrm{PR}(\Sigma)$ such that $[\![\alpha]\!]_A = f$.

We can now explain how we provide a formal account of the concept of a 'value function'. From a synchronous network N we can abstract a Σ-algebra $A = A_N$. This A comprises:

- the natural numbers as a set $T = \{0,1,2,\ldots\}$ and the successor function on T to represent the clock with respect to which the modules of the network are synchronised;
- the Booleans $\mathbb{B}$ and the usual logical operations on $\mathbb{B}$ for testing, and
- the functions on A that specify the modules of N together with the sets that comprise their domains and codomains.

Suppose the sets involved in A are T, $\mathbb{B}$, and data sets $A_1,\ldots,A_n$ say. The next step towards the formal specification of N is to form the algebra $\underline{A}$. This $\underline{A}$ is obtained from A by adding $[T \longrightarrow A_1],\ldots,[T \longrightarrow A_n]$ as new data sets, and new operations $eval_1,\ldots,eval_n$: here the operation $eval_i : T \times [T \longrightarrow A_i] \longrightarrow A_i$ is defined by $eval_i(t,\underline{a}) = \underline{a}(t)$ for each $t \in T$ and $\underline{a} : T \longrightarrow A_i$ for $i = 1,\ldots,n$. The signature of $\underline{A}$ is $\underline{\Sigma}$ which is formed from Σ (the signature of A) by adding names for these new operations.

Formal specification of N is achieved by showing that its value function is simultaneous primitive recursive over $\underline{A} = \underline{A_N}$, that is, by showing there is some $\alpha_N \in \mathrm{PR}(\underline{\Sigma})$ such that $[\![\alpha_N]\!]_{\underline{A}} = V_N$. Since we identify a synchronous network with its value function this α_N serves as a formal (syntactic) specification of N, and '$[\![\alpha_N]\!]_{\underline{A}}$' is a formal (mathematical) expression for the behaviour of N.

This approach to the specification of synchronous algorithms has two significant consequences: first, whilst simultaneous primitive recursion over an abstract data type is not without its intricacies, primitive recursion is a simple idea, and primitive recursive specifications lead to simple inductive proofs of algorithms. Secondly, one might imagine that primitive recursion over an algebra that includes streams would complicate matters; however, many facts about such algebras can be obtained via the following strategy: if we can prove that some statement about PR(B) holds for *any* algebra B, then that statement must be true in the case $B = \underline{A_N}$ as a simple corollary. (This is the strategy behind the proofs of the compiler theorems found in Chapters 7 and 8 for example; see below.)

Also in Chapter 3, we begin our account of the *performance* (synonymously: *complexity* or *execution time*) of synchronous algorithms. The complexity of synchronous algorithms is important for their

application to hardware design. Starting with Thompson[1980] various models for the area-time characteristics of VLSI circuits have been developed. Some of these models contain conflicting assumptions (see Baudet[1983]) and are the subject of experimental work (see Dew and Tucker[1983]. An informal complexity model can be incorporated into the formal definition of the simultaneous primitive recursive functions. We begin by defining a *performance measure P* for an algebra *A*: a performance measure is simply a collection of functions (*performance estimations*) that tell us how long it takes to execute each operation of the algebra on any given input. Performance measures can be nearly arbitrary for the reason that this allows us to choose an account of the complexity of operation evaluation that best reflects our intuition concerning how that operation is implemented. For example, if we think of an operation as being atomic or indivisible, then we can choose a performance measure in which that operation takes one time unit to evaluate on any input. Performance measures appear in Nielson[1984] and generalise the treatment of algorithm complexity in computation over algebraic structures found in Asveld and Tucker[1982].

We provide a complexity theory for synchronous algorithms by defining a length-of-computation function for each PR scheme α; in symbols this function is denoted '$\lambda_P(\alpha)$' with the intention that $\lambda_P(\alpha)(a)$ is the time taken to evaluate α on input a (with respect to performance measure P). An account of the performance of a synchronous network N is thus simply provided by choosing a performance measure P for the algebra $\underline{A_N}$, and then by considering $\lambda_P(\alpha_N)$.

Subsequently, unqualified, we use 'PR' to mean a formal system comprising a syntax PR(Σ), a semantics $[\![.]\!]_A$, and a complexity theory λ_P, for some (standard) signature Σ, (standard) Σ-algebra A, and performance measure P (for A) respectively.

Chapters 4 and 5 are devoted to applications of PR. In Chapter 4 we show that PR-specifications of synchronous algorithms can be used in mathematical verifications by proving the correctness of the two sorting algorithms of Chapter 2. These proofs are important for the following reason: the correctness is best explained in terms of a variety of alternative sorting networks and transformations from one network to another; we use PR to specify each of the networks encountered, and in addition to giving the reader some feel for our notation, this work shows that PR-specifications make complicated correctness proofs manageable and indeed, mathematically satisfying.

In Chapter 5 we further test out our formalism by using PR to specify and establish the correctness of a variety of synchronous algorithms. First, we consider an algorithm for *convolution* found in Brookes[1983]; this algorithm has parallel loading of data. Second, we consider an alternative algorithm for convolution that has serial loading of data. Thirdly, we consider a new algorithm for recognising *palindromes*. Finally we consider the *matrix-vector multiplication* algorithm of Kung and Leiserson (taken from Chapter 8 of Mead and Conway[1980]).

Chapters 6, 7, and 8, are devoted to transformations between *equivalent representations* of synchronous algorithms. We think of these chapters as the mathematical foundations of a *design environment* for synchronous algorithms (see Section 1.3).

In Chapter 6 we give a scientific account of simulating synchronous algorithms. Simulation is no theoretical substitute for formal verification of course, but it is an important aspect of the design and analysis of complex algorithms. An obvious idea for providing a simulation of a synchronous algorithm is to *implement* PR in some executable programming language L say. This idea must be treated with some caution however, especially with respect to algorithm performance. Suppose α is the PR-specification of some synchronous algorithm, and suppose α is implemented as an L-program S. Of course, we can simulate our synchronous algorithm α by executing its simulation S on sample input data. However, that S runs quickly or otherwise on a given input tells us nothing about the performance of α *unless* we have been told what relationship exists between the complexity of S and the complexity of α. Indeed, a similar remark can be made about the behavioural aspect of simulation: unless we have some guarantee that the behaviour of S faithfully represents the behaviour of α, experiments with S tell us nothing about the behaviour of α.

These remarks suggest that like the specification language PR, a simulation language should itself be a formal language with an independently-defined semantics and complexity theory. The language we use is called *FPIT*. FPIT is a formal, structured programming language based on abstract data types and on the *multiple* or *concurrent assignment statement*. (These are statements of the form

$$x_1,\ldots,x_n := e_1,\ldots,e_n$$

whose intended interpretation is that each expression e_i is evaluated and assigned to x_i in parallel for $i = 1,\ldots,n$.) To simulate our PR-specified synchronous algorithms we construct a *compiler* as a mapping c from PR into FPIT that we prove is *correctness and performance preserving*: for each PR scheme α, $c(\alpha)$ is an executable FPIT version of α that is firstly provably equivalent to α, and secondly has the same execution time as α; in this situation it is proper to infer properties of the behaviour and performance of a synchronous algorithm from the corresponding properties of its simulation.

Whilst PR is our official specification language for synchronous algorithms, it turns out that synchronous algorithms can in fact be directly specified in FPIT without using PR as an intermediate stage, and thus FPIT can be viewed as an alternative means of specifying or defining the behaviour of a synchronous algorithm. Now, we have said that the behaviour of a synchronous algorithm is officially defined by means of a PR scheme and so we must ask: what is the relationship between FPIT and PR? Does a specification of a synchronous algorithm in one system tell us any more about the algorithm than its specification in the other? In Chapter 7 we will answer this question (in the negative) by establishing that FPIT and PR are *equivalent* specification languages. Note that the existence of a correct and performance preserving compiler from PR into FPIT implies that every synchronous network (when specified in PR) can be simulated in FPIT with no change in performance, and thus the simulation can be regarded as an alternative but equivalent means of formalising the network: thus FPIT is at least as good as PR for specifying synchronous algorithms. To show that PR is at least as good as FPIT we will define a second compiler, this time from FPIT into PR, which we will again prove is both correct and performance preserving, and thus PR and FPIT are *formally computationally equivalent*: whatever we can define using PR we can simulate with FPIT, and with equivalent performance, and moreover, whatever we can simulate using FPIT we can define with PR, and with equivalent performance. Thus in the case of

defining or modelling synchronous algorithms, PR and FPIT are *formally equivalent representation systems.*

Chapter 8 concludes our mathematical work on synchronous algorithms with a formal account of top-down design. In algorithm design generally, we are advised to initially design an algorithm for a given task at a high level of computational abstraction: use of high-level 'primitive' operations makes the algorithm easy to understand and therefore more amenable to formal verification. To provide an implementation of the algorithm at a lower level of abstraction we implement the algorithm's high-level operations by programs defined over more primitive or lower-level operations, and then we combine these implementations with the original algorithm in some way to yield the required implementation of the original algorithm. The essence of 'top-down design' is that the way in which we combine the algorithm with the implementations of its operations should be such that the correctness of the implemented algorithm *follows automatically* from the correctness of the original algorithm and the correctness of the implementations of the operations; this obviates the need for verification of the algorithm when implemented at lower levels where the resulting design may be so complex as to be impossible to formally verify in practice. Indeed, one can make a similar remark about algorithm performance: it is usually extremely tedious (and therefore error-prone) to calculate the complexity of an algorithm when implemented at a low level of abstraction. In the same way that correctness at a low level should follow from the correctness at a high level, we want to be able to *predict* the complexity at a low level from complexity at a high level.

Our theory of top-down design for synchronous algorithms has both of these properties: algorithm correctness and performance at low levels of abstraction are guaranteed by theorems (Implementation Theorem 8.4.1 and Hierarchy Theorem 8.4.3 in particular). In essence, our strategy for achieving this theory is to first provide a theory of top-down design for PR schema, and then to apply this work to synchronous algorithms.

We begin in Section 8.1.1 by investigating the conceptual issues involved in the top-down design of synchronous algorithms. In the next three sections we present a theory of top-down design for PR, central to which is a new definition of what it means for one (high level) algebra to be 'implementable' in PR over another (low level) algebra. This definition generalises the idea of a *computable algebra* due to A. I. Mal'cev and M. O. Rabin (see Mal'cev[1961] and Rabin[1960] respectively). In the Mal'cev-Rabin theory, an algebra A is said to be computable (over the natural numbers $\mathbb{N}$) if (a) A can be Gödel-numbered or coded via a surjection $\alpha: \mathbb{N} \longrightarrow A$, (b) each operation of A can be computed (with respect to α) via a partial recursive function over $\mathbb{N}$, and (c) the relation $\equiv_\alpha$ defined by $n \equiv_\alpha m$ iff $\alpha(n) = \alpha(m)$ is recursive. The main purpose of this theory was to study effective computability over algebraic structures such as groups, rings, and fields in mathematics; it is now a standard tool in the theory of data types (see Meseguer and Goguen[1985]). In contrast to the Mal'cev-Rabin definition, we define an algebra A to be (PR-) computable over *another* algebra A' if, first, A can be coded via a surjection $\gamma: A' \longrightarrow A$, and second, each operation of A can be computed (with respect to γ) via a simultaneous primitive recursive function over A'. This new definition leads to the idea of a *hierarchy*

$A, A', A'', \ldots$ of algebras wherein each algebra is implementable over its successor. Indeed, our definition can be seen as the starting-point for a generalisation of computable algebra in which primitive recursive operations on A are replaced by generalisations of partial recursive functions on A; this new theory has applications to the theory of *parameterised* data types.

We will now explain the connection between this PR theory and the top-down design of synchronous algorithms.

Suppose that N is a synchronous network whose modules are specified by the operations of a (high level) algebra A. Now suppose that each module m of N can be implemented by a synchronous network $N(m)$ whose modules are specified by the operations of (lower level) algebra A'. Intuitively, we can replace each module of N by the network that implements it, and this substitution will lead to a new network N' whose modules are specified by the operations of A'. Suppose the signatures of A and A' are Σ and Σ' respectively. Then N will be formalised as a scheme $\alpha_N \in \mathrm{PR}(\underline{\Sigma})$, and N' will be formalised as a scheme $\alpha_{N'} \in \mathrm{PR}(\underline{\Sigma}')$. Also, each $N(m)$ will be formalised by some $\alpha_{N(m)} \in \mathrm{PR}(\underline{\Sigma}')$. Now, from our PR theory (Implementation Theorem 8.4.1 to be precise), it is a fact that if we transform α_N by substituting $\alpha_{N(m)}$ for each occurrence in α_N of the operation of A that $N(m)$ implements, then this transformation is a correctness preserving map $c : \mathrm{PR}(\underline{\Sigma}) \longrightarrow \mathrm{PR}(\underline{\Sigma}')$. Furthermore, $[\![c(\alpha_N)]\!]_{A'} = [\![\alpha_{N'}]\!]_{A'}$, which, in words says that the intuitive idea of network-for-module substitution is formalised by scheme-for-operation substitution, thus providing the required theory of top-down design.

Finally, with Chapter 9 we end with some concluding remarks and some directions for future work.

1.2 RELATED WORK.

In this section we will review research on the mathematical specification, verification, and design of concurrent algorithms that is relevant to this thesis. Research that is otherwise related to our work is reviewed in the appropriate chapter. (For example, in Chapter 7 we verify the two compilers that compile from PR to FPIT and back again; the subject of 'compiler correctness' is therefore discussed in that chapter.)

After some historical remarks, we review research on *systolic algorithms* in two sections. Systolic algorithms are a restricted kind of synchronous algorithm that have received much attention due to their suitability for implementation in, and therefore exploitation of, VLSI technologies: in Section 1.2.1 we review research on systolic algorithms and their specification and verification, and in Section 1.2.2 we review research on *design tools* for systolic algorithms. Finally, in Section 1.2.3 we review contemporary research on the specification and formal verification of computer hardware in general.

Perhaps the earliest examples of synchronous algorithms are the *neurone nets* of McCulloch and Pitts[1943]. McCulloch and Pitts were interested in mathematical models of neural activity (an interest fuelled, no doubt, by the seminal work Turing[1936] in which we find the first mathematical characterisation of a physical device), and neurone nets were devised for exactly this purpose. A neurone net is a simplified discrete model of neural activity comprising 'cells' (abstract neurones) connected together by

'fibres' (abstract *axons* and *dendrites*, the 'wires' by means of which electrical pulses are sent from neurone to neurone).

Other early examples of synchronous algorithms are *cellular automata* introduced by von Neumann in 1948 (see the volume Neumann[1987]). Von Neumann was interested in how one might "abstract the logical structure of life" and proposed a mathematical model for analysing how a network of processes could evolve over time. One of his principal concerns was the possibility of *self-reproducing* automata; his ideas have since been popularised in the well-known 'Game of Life' (see Conway, Berlekamp, and Guy[1982]): 'Life' is intuitively a synchronous process and is easily seen to be a synchronous algorithm (in our technical sense) when played on a finite grid.

Von Neumann was aware of the significance of his research as the foundations of parallel computation, Minsky and Pappert carried further the study of neural networks (Minsky and Pappert[1969]), and the mathematics underlying such networks of processes was examined in Hennie[1961] and Codd[1968]. However, this line of research does not seem to have been carried further for the reason that it was ahead of its time: it was not feasible to make these networks with pre-transistor technology.

There has been a recent resurgence of interest in synchronous concurrent algorithms that is due to two factors, both arising from VLSI: first, it is now possible (in principle) to implement large neural networks as physical devices (as witnessed by research on *connectionist* and *Boltzmann machines*; see Aarts and Korst[1987] for an overview). Second, in the design of computers it has long been acknowledged that using synchronous circuitry alleviates some of the complexities in circuit design (for example, timing problems at the circuit level). Today, in custom VLSI circuits, using synchronous circuitry is a VLSI design strategy that is epitomised in the concept of a *systolic algorithm* which we will now discuss.

1.2.1 Systolic Algorithms.

A principal application area for VLSI devices is the implementation in hardware of algorithms that solve particularly computation-intensive tasks; typically, the kind of task that arises in signal and image processing for example. Implementing an algorithm in hardware increases computation speed of course, especially so if the hardware is dedicated to that one specific task. With the advent of VLSI technologies (wherein it is possible to place many processors in close proximity to each other), it is now possible to conceive of hardware implementations that involve a high degree of concurrency, increasing computation speed even further.

Systolic algorithms, pioneered by H. T. Kung and his colleagues at Carnegie-Mellon University, are algorithms that are tailored to VLSI implementation. Essentially, a systolic algorithm is a synchronous algorithm with the following distinguishing features:

- the algorithm involves only a few simple types of module;
- the algorithm is formulated in such a way that the modules only need to communicate with their nearest neighbours;
- the algorithm is laid out (usually as a two-dimensional array) in such a way that the interconnections form a simple, regular pattern (for instance, see Figure 2.8 depicting a systolic sorting

algorithm), and

- the algorithm uses each piece of input data many times.

The first three of these features lead to efficient VLSI implementations: to fully exploit the VLSI medium, a VLSI design is likely to involve a large number of modules; to minimise design and implementation costs it is therefore expedient that the algorithm involves only a few simple types of module. Also when a large number of modules are used, communication costs become significant: in VLSI it is the number of interconnecting channels and hence the physical area required by these channels that dominates the efficiency (power, time, and area) of the implementation (Sutherland and Mead[1977]); local communication and regular structure minimises these costs of course.

The fourth feature, that a systolic algorithm uses each piece of input data many times, concerns the 'computation-intensive' quality of the problems for which VLSI implementations are sought:

A problem is said to be *compute-bound* if, in the computations required to solve the problem, the number operations that must be executed exceeds the number of memory accesses. Thus, intuitively, compute-bound problems need fast processing, and this is offered by VLSI. However, existing (general-purpose) computers have a *von Neumann architecture*; that is, an architecture that comprises a memory and a single processor: to compute an operation (for example, an addition or a multiplication), the input data must first be loaded into the processor from memory, then the processor computes a result, and this result is then stored back into memory. Intuitively then, with a von Neumann architecture the speed of computation is limited by the speed of input/output (or *memory bandwidth*): there is no point in having a fast processor if data cannot be supplied/retrieved quickly enough to support the processor. The von Neumann configuration is thus not well-suited to the implementation of algorithms for compute-bound problems. However, by using a systolic architecture, the memory bandwidth versus processing speed problem (the 'von Neumann bottleneck') is circumvented by using each piece of input data many times.

Beginning with the seminal paper Kung and Leiserson[1979], many researchers have become interested in systolic algorithms due their attractiveness as a paradigm of VLSI design. Preliminary exploratory work on systolic algorithms was concerned with (particular) systolic algorithms for particular computational problems, and many such algorithms are now known: see Kung[1982] and Fischer and Kung[1985] for surveys of systolic algorithms for a wide variety of problems.

As the concept of a systolic algorithm is now well-developed, contemporary research has turned to larger and more theoretical questions concerning systolic algorithms. One may summarise the issues addressed by contemporary research via the the following two questions:

- How do we formally analyse systolic algorithms?
- How do we design a systolic algorithm for a given task?

The first of these questions concerns the provable correctness of systolic algorithms (a desirable property for any algorithm). Whilst the regularity of structure and data-flow make a systolic algorithm easy to implement in VLSI (or at least amenable to implementation), this does not necessarily mean that it is easy to understand a given systolic algorithm; typically, the sequence of actions performed by a

systolic algorithm form a very complex pattern, and the correctness of the algorithm need not be at all obvious.

The question asks for a mathematical model or notation for systolic algorithms in which we can establish an algorithm's correctness by proving theorems about the algorithm's behaviour. One approach to this verification problem is to use an already existing 'tried and trusted' theory of concurrency, such as ('synchronous' versions of) CCS as in Hennessy[1986] and Backhouse[1983], or ACP as in Weijland[1987], or our own notation PR. A criticism of the ACP and CCS-type approaches is that these notations are principally *a*synchronous specification tools with an 'arbitrary interleaving' interpretation of concurrency wherein parallel computation is conceptually reduced to sequential computation plus nondeterminism. In practice this means that nondeterministic choice appears in specifications of systolic algorithms (indeed, synchronous algorithms in general), in effect cluttering the specification with low level information about *how* the modules of an algorithm are synchronised: intuitively, this is not something we want to know when we are given that the modules *are* synchronised. In contrast, synchronousness and 'true parallelism' are built into PR (via the simultaneous primitive recursion mechanism) and this leads to concise and rather elegant specifications of algorithm behaviour as we will see in Chapters 2 and 5.

An alternative approach is to devise a new model which is tailored to systolic algorithms. Work in this direction is Kung and Lin[1983] and Melhem and Rheinboldt[1984]. Both of these approaches suffer from the fact that the underlying mathematics is somewhat unclear, although we note that Brookes[1983] begins to further explore the work of Kung and Lin.

1.2.2 Design Tools for Systolic Algorithms.

The design of complex algorithms needs software tools for exploring the implications of making different implementation choices. We will discuss software tools based on our work on PR in Section 1.3; here we review existing research into software tools for systolic algorithms.

The second of our two questions above concerns what is often referred to as the 'synthesis' of systolic algorithms; that is, the systematic construction of a systolic algorithm for a given task from a specification of that task; often, use of the term 'synthesis' implies that the construction is 'guaranteed' to produce a correct design. There is much work on the synthesis of systolic algorithms to be found in the literature; again, the regular structure of a systolic algorithm does not necessarily mean that it is easy to devise a systolic algorithm for a given task: the design of nontrivial systolic algorithms requires considerable expertise and ingenuity.

The basic idea behind the concept of synthesis (of systolic algorithms) is to circumvent the difficulty of conceiving a systolic design for a given task by first coding an algorithm that solves the task in some (familiar) algorithmic notation, and then to use some mechanical transformation (software tools) to map the algorithm to a systolic implementation. Intentionally, the algorithmic notation (hereafter referred to as the 'source language') is a familiar one so as to make the original algorithm comprehensible and amenable to formal verification, and the transformation is a correctness preserving map. Ideally, the source language should be as general-purpose as possible (to facilitate the expression of as many

algorithms as possible). However, the problem of finding an appropriate transformation for a truly general-purpose source language is a very difficult one, and in practice source languages are tailored to the synthesis problem.

Uniform recurrence equations (Karp, Miller, and Winograd[1967]) are commonly used as the starting point for synthesis, see for example: Mongenet and Perrin[1987]; Quinton[1987]; Li and Wah[1985]; Rajopadhye and Fujimoto[1987]; Delosme and Ipsen[1987b], and Guerra and Melhem[1986].

An alternative choice of source language is a language of imperative programs of a special form (usually nested loops), see for example: Cappello and Steiglitz[1984]; Fortes and Moldovan[1986]; Miranker and Winkler[1984]; Lam and Mostow[1985], and Huang and Lengauer[1987]. Other research in this area is: Snepscheut[1985], whose work is cast in *trace theory* (Rem[1983]), and Chandy and Misra[1986a], whose idea is to use traditional tools taken from the design of sequential algorithms (specifically, the concept of a 'loop invariant').

A related research area is the synthesis of systolic algorithms from concurrent algorithms that are already described as a network of modules, but that are not systolic. See for example: Kung and Lin[1983] who build on Weiser and Davis[1981] and Johnsson and Cohen[1981] (see also Brookes[1983]); Leiserson and Saxe[1983], and Ramakrishnan, Fussel, and Silberschatz[1985].

Synthesis is a very difficult problem, and one that this thesis does not address. However, we feel that much of this research would be improved by formal definitions of the source language, transformation, and model of systolic computation employed, since without these there can be no precise sense (and certainly no formal sense) in which the transformation can be correctness preserving. Few of the researchers mentioned above give an effective characterisation of their source language, and none give a precise definition of what is meant by a 'systolic algorithm'. Not only does this mean that the correctness implicit in the concept of synthesis is at best rather woolly, but it also makes it impossible to provide a comparative survey of the relative merits of the above approaches: we cannot tell whose source language is the 'most general-purpose', nor who has the 'most implementable' concept of a systolic algorithm.

To the interested reader we recommend Quinton[1987], Huang and Lengauer[1987], and Rajopadhye, Purushothaman, and Fujimoto[1986] as the most tractable introductions to the subject. We also note that some of the above research is at a more advanced stage than others in the sense that software tools have been produced: Gachet, Joinnault, and Quinton[1987] describes the system DIASTOL based on the work of Quinton; Mongenet and Perrin have a system called SYSTOL (see Mongenet and Perrin[1987]); Huang and Lengauer also have a software system (see Huang and Lengauer[1987]); Moldovan has a system called ADVIS described in Moldovan[1984], and Delosme and Ipsen have two systems, SAGA and CONDENSE, described in Delosme and Ipsen[1987a].

1.2.3 Hardware Specification and Formal Verification.

Our work on synchronous algorithms is inspired by the problems of verifying algorithms that can be (or are) implemented in hardware (that is, as physical devices). The verifications found in this thesis are mathematical (semantic) proofs rather proofs based on (syntactic) axioms and rules of inference or deduction. In this section we will review research that does use such formal reasoning about hardware. A substantial research project for us is to re-examine our correctness proofs to find logical languages and proof systems that formalise the reasoning involved in these proofs, and to develop software tools such as 'proof-checkers' to support the verification procedure; we think of the notes below as preparation for this future work. (See Section 1.3 and Chapter 9.)

The basic idea behind research into formal reasoning about hardware is that if one has a clear cut mathematical logic, then one can map an algorithm of interest into the (term) language of the logic and then use the logic's axioms and rules of inference to prove theorems about the algorithm, in particular, to verify it with respect to some correctness specification (in the language of the logic). A key idea here is that the application of rules of inference can be automated; this is a particularly attractive idea when the algorithm of interest is either large or complicated and correctness proofs are likely to be lengthy. For example, it is reported in Gordon[1987] that Cohn's verification of the VIPER microprocessor (see Cohn[1987] and Cullyer[1987] respectively) used over a million deductions in the correctness proof.

There is much research on this topic to be found in the literature, too much for us to give an exhaustive survey; we will review only what we feel to be a representative sample of this work.

Work Based on First-Order Logic. First-order logics are distinguished by the fact that one can only quantify over data objects from the underlying data type. An early example of using first-order logic to reason about hardware is Barros and Johnson[1983] in which four kinds of asynchronous circuit are proved equivalent. Eveking[1985] develops an approach based on first-order logic that addresses the question of how specifications at different levels of abstraction are related. Hunt[1986] describes the verification of a complete microprocessor (the FM8501, roughly equivalent to a PDP-11); Hunt uses Boyer-Moore logic (a first-order logic; see Boyer and Moore[1979]) to describe his microprocessor, and a theorem-prover for the Boyer-Moore logic is used to verify the design.

Work Based on Higher-Order Logic. Higher-order logics are distinguished by the fact that one can only quantify over functions and relations (second-order logic), and/or functions of functions (third-order logic) etc. Higher-order logics originate in research into the foundations of mathematics (Church[1940]), but the insight that higher-order logic is useful specifying and verifying hardware is attributed to K. Hanna; in Hanna and Daeche[1986] Hanna and Daeche argue the appropriateness of higher-order logic for describing hardware at a very low level of abstraction: circuit timing issues can be addressed within their model for example. Hanna's work (collectively known as the VERITAS project) includes a theorem-prover described in Hanna and Daeche[1984].

Using higher-order logics to specify and verify hardware at a higher level of abstraction is the work of M. Gordon and his colleagues at the University of Cambridge. Gordon argues the case for higher-order logic in Gordon[1986], and its use is exemplified in Joyce[1987] and Cohn[1987] which

describe the formal verification of microprocessors. Gordon's work (collectively known as the HOL project) includes an impressive interactive theorem-prover based on LCF and its metalanguage ML. (LCF and ML are due to R. Milner, but these languages have evolved many times before being incorporated in HOL; see Gordon[1987] for a short history.)

Work Based on Temporal Logic. A temporal logic is a formal system for reasoning about the occurrence of events in time, and as such it is a natural choice of formalism for hardware specification and verification. The earliest example of using temporal logic for this purpose is apparently Bochmann[1980] (published as Bochmann[1982]). Another early example is Malachi and Owicki[1981] in which temporal logic is used to formalise Seitz's so-called 'weak conditions' for composing asynchronous circuitry (see Seitz[1980]).

Mishra and Clarke[1983] describes using a kind of temporal logic called *Computation Tree Logic* (CTL) in the automatic verification of asynchronous circuits. The basic idea is that from a circuit one can abstract a state transition graph comprising the states that the circuit can be in, and the transitions between these states that the circuit admits. The truth or falsity of a formula in CTL (one that asserts the circuit's correctness in particular) is determined relative to a state transition graph and this process can be automated via a program called a 'model-checker'. This work is notable for the fact that the model-checker is very efficient in the sense that it is linear in the size of the formula being checked and the number of states in the state transition graph.

Other notable work in this area is Moskowski[1983] which describes the use of *Interval Temporal Logic* (ITL) to reason about hardware. We are not aware of a proof-checker for ITL, although we note that Moskowski[1986] describes the use of ITL as a programming language (TEMPURA), and thus hardware specified in (a subset of) ITL can be executed (simulated).

Other Work. As pointed out in Camilleri, Gordon, and Melham[1986], there are dangers involved in using a logic that is based on a circuit model that is too inaccurate: the authors give an example of a circuit that will not work (reliably) in practice, but that can nonetheless be formally verified. The problem here (noted by the authors) is that in the model it is assumed that values (voltages) in the circuit are well-defined zeros and ones, whereas in practice there is a range of voltages to be found in an electrical circuit. An early attempt to model this fact is Bryant[1984] in which transistors can 'hold' zero, one, or 'X', the latter denoting an intermediate voltage not corresponding to zero or to one. Bryant's work was originally devised for simulation rather than formal verification purposes; it has been extended and reformulated in Winskel[1987] as a formal multi-valued logic in which low level (electrical) aspects of circuits can be represented. (We also note that Bryant has continued his work in Bryant[1986].)

Milne[1982] describes a calculus (CIRCAL) in the style of CCS for reasoning about circuits; low level details can be specified in this calculus (see Milne[1983]) although it is not necessarily limited in this way: Milne[1986] describes how his calculus can be used in top-down design.

Johnson[1984] uses equations to define circuits; the equations are given a semantics via fixed-point theory and are incorporated in a language called DAISY that is based on LISP.

Finally, Sheeran[1983] describes a functional notation based on FP (Backus[1978]) to specify circuits; algebraic laws are provided for formal reasoning purposes.

1.3 THE DEDEKIND PROJECT.

This thesis is a contribution to a project on methodologies, mathematical theories, and software tools for synchronous algorithms that is being developed under the supervision of Dr. J. V. Tucker here in the Department of Computer Studies at the University of Leeds. In particular, this thesis is basic to the design of a software environment for the design and analysis of synchronous algorithms. The environment is provisionally called DEDEKIND after Richard Dedekind, inventor of primitive recursive functions (see Dedekind[1888]). In this section we will sketch this project, describe existing software tools and mention others that will be included later. Note that this is not an *ad hoc* list of programs: each tool is included *only* on the basis of a clear cut mathematical idea.

As we remarked earlier, PR and FPIT can both be viewed as specification languages. Under current plans DEDEKIND will include two languages: PRESS (for Primitive Recursive Synchronous Systems) based on PR, and CARESS (for Concurrent Assignment Representation of Synchronous Systems) based on FPIT. DEDEKIND has two notations because PR, as a language of function definitions, leads to specifications that are suitable for mathematical reasoning and verification; whilst FPIT, as an imperative language, is suitable for programming. Indeed, it is in the spirit of programming that CARESS is being developed. CARESS has been implemented in C on a VAX 11/780 by our colleague A. R. Martin. Martin and Tucker[1987] describes the practical use of CARESS and its associated preprocessors for describing synchronous algorithms.

As we remarked earlier, PR can be compiled into the language FPIT to provide simulations of synchronous algorithms. The proof that PR and FPIT are computationally equivalent underwrites the idea that we can flip between PR- and FPIT-representations of a synchronous algorithm at will. A prototype compiler from PR to FPIT has been implemented by Martin but a compiler from FPIT back to PR awaits realisation.

Another feature of our mathematical research that has practical implications is our account of algorithm complexity or performance. The length of computation function λ_p (see Section 1.1) is 'syntax-directed': it is possible to compile a PR scheme α into a program which computes $\lambda_p(\alpha)$ (a function from input data into time) on any input data. Indeed, the same will done for FPIT programs; it will be interesting to empirically compare the performance of equivalent PR- and FPIT-specifications of the same algorithm (given that the compilers from PR to FPIT and from FPIT to PR are both formally performance preserving).

Finally, our account of top-down design in Chapter 8 is a first step towards an enhanced (structured) version of PR for expressing PR specifications of hierarchically structured synchronous algorithms: the mathematics of that chapter supports the idea that algorithms specified in such a notation can be compiled back into PR thus providing a simulation at any level of abstraction (given the equivalence of PR and FPIT).

There is other ongoing theoretical and experimental research worth noting for completeness.

A graph-theoretic model of synchronous computation has been developed by our colleague K. Meinke (see Meinke[1987]). Meinke's work is inspired by the fact that people often describe a synchronous algorithm by means of an informal 'picture' of the algorithm; these pictures can be formalised using (directed) graphs, and computation by such graphs is the basic subject of Meinke's work. Pictorial representations of a synchronous algorithm are often helpful for explaining the structure and data flows involved: it is possible that DEDEKIND will have a third autonomous notation for specifying synchronous algorithms, one based on Meinke's model that reflects *architectural* concerns. Indeed, pictorial representations give the designer some intuition for the area required by a synchronous algorithm. One can draw a 'floorplan' of a PR scheme in a way that is reminiscent of μFP (Sheeran[1983]). Our colleague S. M. Eker has investigated floorplanning from PR schema as part of his undergraduate studies (Eker[1986]).

The verification of synchronous algorithms is further developed with respect to special-purpose hardware for graphics and signal-processing devices. A basic theme is proofs of correctness of algorithms that have been developed or derived by step-wise refinement. In Eker and Tucker[1987] a study of the verification of incremental line-drawing algorithms is undertaken using PR as the primary specification tool. This study involves the derivation of *Bresenham's algorithm*, by geometrical transformations and program transformations, and direct proofs in the style of Chapters 4 and 5. Our colleague K. M. Hobley is studying the relationships between correctness proofs that arise in the design, by step-wise refinement, of a digital correlator specified in Harman and Tucker[1987] (see below). This research, like that of Chapters 4 and 5, is prerequisite to attacking formal and mechanical verification. Our colleague C. A. Jervis is currently working (Jervis[1988]) on Hoare logics for languages that closely resemble FPIT. Possible future work on proof systems and logical languages for PR and FPIT is discussed in Chapter 9.

Finally, *language-independent* theories of specification are being investigated by our colleague N. A. Harman. Harman and Tucker[1987] provides a taxonomy of formal specifications and mathematical methodologies encountered in the incremental design of a digital correlator. Harman has also studied counters and microprocessors (including the VIPER microprocessor; see Cullyer[1987]). The simultaneous primitive recursive functions on an abstract data type are also used in analysing specifications. Ultimately this work will be incorporated into DEDEKIND; indeed software tools for executing specifications are already being developed by A. R. Martin.

CHAPTER 2
SYNCHRONOUS ALGORITHMS

In this chapter we will discuss the concept of a synchronous algorithm and identify issues central to synchronous computation in general.

In Section 2.1 we will describe informally, and in general terms, a model of synchronous computation which we call *synchronous networks*. These networks comprise a collection of *modules* which are connected together by communication links that we call *channels*; the networks are synchronous in the sense that computation by and communication between modules is synchronised with respect to an explicitly defined *clock*.

In Section 2.2 we show how synchronous algorithms can process *streams* of data, and discuss the correctness of algorithms with respect to certain specifications. These specifications define important characteristics of algorithms such as *initialisation time* and *period*.

In Section 2.3 we investigate two synchronous sorting algorithms as case studies.

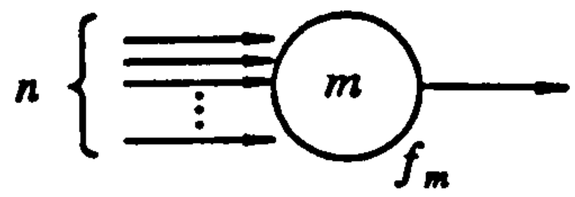
In Section 2.4 we take our first steps towards the formal specification of a synchronous algorithm: we show how from the communication structure of a synchronous network, and from functional specifications of the network's modules, we can systematically construct functional specifications of the complete network.

In Section 2.5 we outline the principal objectives of this thesis. In essence, we seek a theory of synchronous computation which has regard for the formal *specification*, *verification*, *simulation*, and *hierarchical design* of a synchronous algorithm.

2.1 SYNCHRONOUS NETWORKS.

A synchronous algorithm is a parallel algorithm over a set A that is described as a network of *modules* which are synchronised by means of a *clock* and which communicate via interconnecting *channels*. The reader can look ahead to Figures 2.1, 2.5, and 2.8 for examples of the kind of network we have in mind. The networks comprise *modules*, *channels*, *sources*, and *sinks*, which we will now describe in turn:

We imagine a module to be an atomic computational device comprising a *store* and a *processor*: the store is capable of holding a *single* datum from the set A, and the processor is capable of computing some *total single-valued* function defined over A. In a figure, a module is typically represented by:

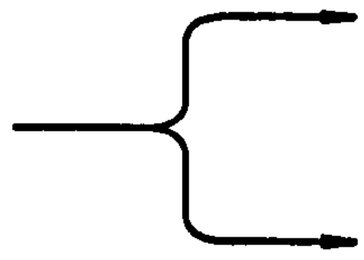


Of course, the arrows in this and subsequent figures represent the channels that are the means by which data is passed from module to module.

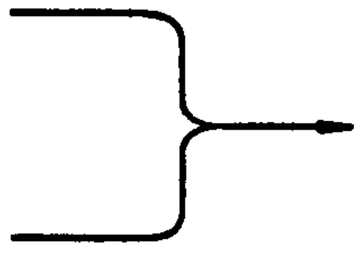
The above figure depicts a module m, which is behaviourally specified by the map $f_m : A^n \longrightarrow A$. This map serves to specify m's behaviour in the following way:

Initially, we imagine m's store to be holding some value $a \in A$ which m is attempting to propagate along its output channel; simultaneously, m is expecting input to arrive on its input channels. (In this situation we will say 'm is ready to compute, holding value a'.) Subsequently, when some vector $(a_1,...,a_n)$ of input data is made available to m on its input channels (one datum per channel), the module performs a sequence of actions which we call a *step*. A step comprises reading the current input whilst simultaneously propagating the value held by m's store along its output channel; once the input $(a_1,...,a_n)$ has been read, m then computes the value $f_m(a_1,...,a_n)$ which is placed in m's store, overwriting the previous value; thereafter m is ready to compute, holding value $f_m(a_1,...,a_n)$.

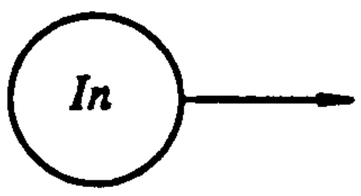
A network's *channels* we imagine to be *uni*directional communication links between the network's modules. We will assume the channels have *bandwidth* 1; that is, a single channel may only carry a single datum $a \in A$ (and not, for example, a vector $(a_1,...,a_n) \in A^n$). Thus a module specified by an n-ary function has n input channels. Additionally, we allow channels to branch (finitely) as for example:



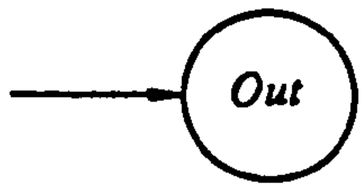
with the intention that a datum on a branching channel is reproduced on each branch; however, we do not allow channels to merge together as for example:



The remaining constituents of a network to be described are the *sources* and the *sinks*. Intuitively, sources and sinks are the only points at which data held in the network is visible to an external observer. A source, or *input module*, is typically represented in our figures by:



Sources supply data to the network and are the only points at which a datum may enter the network. A sink, or *output module*, is typically represented by:



The purpose of a sink is merely to identify which of the network's channels are regarded as outputs. Note that since channels can branch, one source can supply many modules (with the same input data), but since channels may not merge together, one sink can only have input from one module; in other words, there is a unique module which supplies any given sink with output data.

Example. Consider the network N of Figure 2.1. N is a network depicting a synchronous algorithm over the set $\mathbb{N}=\{0,1,2,...\}$ of *natural numbers*. N has source *In*, sink *Out*, and N involves modules which compute the natural number operations: successor (s); predecessor (p); addition (+), and multiplication (×). Additionally, N has six distinct channels, three of which branch into two.

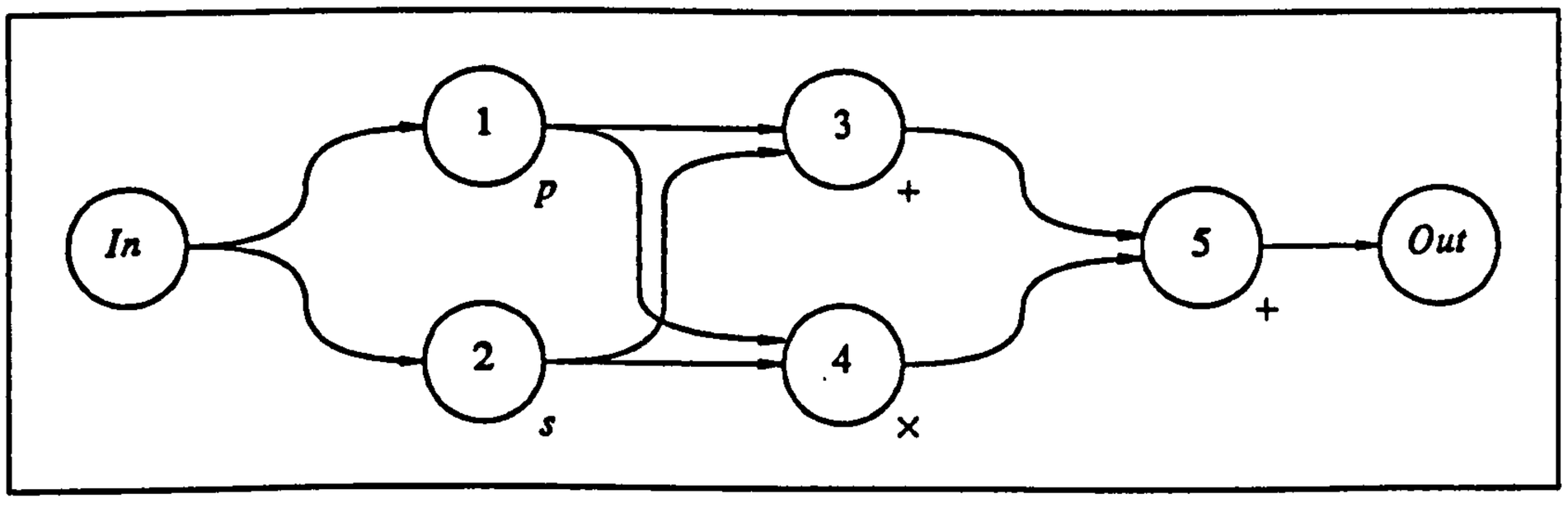


Figure 2.1 - The network N.

□

We can now describe what is 'synchronous' about a synchronous network or algorithm.

Let N be an n-source network over data set A. Initially, we imagine N's sources to be (instantaneously) supplying $(a_1,...,a_n)\in A^n$ to the network, and we imagine each module m of N to be ready to compute, holding some value $x=x_m\in A$; in this situation we say N is *initialised*.

At the instant that N is initialised, every module is being supplied input at precisely a time when each expects or requires input; in this way every module starts to perform its first step simultaneously. Suppose that the time taken, according to some clock C, for *every* module to perform *any* step is less than or equal to some *fixed* constant $\tau>0$. Intuitively, it follows from this assumption that since the

modules start a step simultaneously with respect to C, they can be made to end the step simultaneously too. Thus the modules of a network can be *synchronised*: at every step, each module supplies data to its neighbours just as the neighbours require input. Furthermore, notice that when the modules are synchronised, a module which requires a vector $(a_1,\ldots,a_n)$ of input data receives each a_i simultaneously for $i=1,\ldots,n$.

Before we consider an example, it is worth further considering the implications of our 'constant time' hypothesis above.

Suppose every module performs each step in time at most τ; then the modules are synchronised initially (that is at time 0 with respect to C), and then at times $\tau,2\tau,3\tau,\ldots$. We can normalise τ to unity (that is, take $\tau=1$) and our hypothesis, after normalisation, becomes a *performance abstraction* that defines a new *virtual clock* T measuring discrete time $t=0,1,2,\ldots$. Notice that for a specific algorithm depicted as a network N, the choice of τ is dependent on the particular modules that N employs, and thus τ is properly τ_N, and hence T is T_N; for this reason we refer to T as *the algorithm's clock.*

As we have explained it, the idea behind a functional specification $f_m : A^n \longrightarrow A$ of a module m is that if the input to m at a given time t is $a=(a_1,\ldots,a_n)$, then $f_m(a)$ is the value held by m at time $t+1$; such a module is *autonomous* with respect to the clock T since the value produced by a module is independent of the time t. Later, we will see examples of modules that are *non*autonomous with respect to T; accordingly, a nonautonomous module m has a functional specification of the form $f_m : T \times A^n \longrightarrow A$ when m has n input channels. (Nonautonomous modules will not have an explicit (extra) input channel to carry the time t; the current time we imagine to be globally available to all modules.) Similar to an autonomous module specification, the intention behind a nonautonomous specification f_m is that $f_m(t,a)$ denotes the value held by m on completion of a step which *began* at time t (at which time the input $a=(a_1,\ldots,a_n)$ was available to the module); that is, $f_m(t,a)$ denotes the value held by m at time $t+1$.

Example. Let us reconsider the network N of Figure 2.1. As we have explained above, if we assume each of the network's modules to perform a step in unit time, then we obtain a clock $T=T_N$, where each $t \in T$ is a time point at which the modules are synchronised.

The network N is intended to compute or implement the function $f : \mathbb{N} \longrightarrow \mathbb{N}$ defined by

$$f(n) = (n-1)+(n+1)+(n-1)\times(n+1)$$

To see how N computes $f(n)$ for any argument $n \in \mathbb{N}$, suppose that the source supplies n to the network; for simplicity we imagine that N's source constantly supplies n to the network for $t=0,1,2,\ldots$. Also suppose that at $t=0$ the ith module holds some $x_i \in \mathbb{N}$ for $i=1,\ldots,5$. Then we may tabulate the value held by the ith module at time t for $i=1,\ldots,5$ and $t=0,\ldots,3$ as in Table 2.2 (For $t \geq 3$ the system is stable in the sense that no new values are produced.)

time	module				
	1	2	3	4	5
0	x_1	x_2	x_3	x_4	x_5
1	$n-1$	$n+1$	x_1+x_2	$x_1\times x_2$	x_3+x_4
2	$n-1$	$n+1$	$(n-1)+(n+1)$	$(n-1)\times(n+1)$	$x_1+x_2+x_1\times x_2$
3	$n-1$	$n+1$	$(n-1)+(n+1)$	$(n-1)\times(n+1)$	$(n-1)+(n+1)$ $+(n-1)\times(n+1)$

Table 2.2 - Tabulation of the network N in operation on a fixed input.

We regard the value held by m_5 at a time t to be the network's output at time t (since m_5 is the only module connected to a sink). Clearly, network output is dependent on the time t, the input to the network n, and the vector of initial internal values $x=(x_1,...,x_5)$. We can formally specify the network's output as a function of time and data in the following way: let $f_N : T\times\mathbb{N}\times\mathbb{N}^5\longrightarrow\mathbb{N}$ be defined by

$$f_N(t,n,x)=\begin{cases} x_5 & \text{if } t=0\\ x_3+x_4 & \text{if } t=1\\ x_1+x_2+x_1\times x_2 & \text{if } t=2\\ f(n) & \text{if } t\geq 3\end{cases} \qquad (1)$$

for each $t\in T$, $n\in\mathbb{N}$, and $x=(x_1,...,x_5)\in\mathbb{N}^5$.

The network achieves its purpose in the sense that if n is supplied to the network as input, then $f(n)$ eventually emerges as output. To be more precise, $f(n)$ emerges after three steps; that is, *with respect to the virtual clock defined by* N, the computation takes three time units.

It is also important to realise that because of the network's synchronous operation, the value held by a module at a time $t+1$ is always completely determined by the module's specification and the values held by neighbouring modules at time t. □

The preceding example serves to illustrate the important idea that synchronous networks can be formalised via functions of time, input data, and initial values; indeed this idea is basic to this thesis. In Section 2.4 we will explain how to define f_N (along with other functional specifications) for an arbitrary network N: if N is an n-source, k-module, m-sink, synchronous network over A, then this f_N will have functionality $f_N : T\times A^n\times A^k\longrightarrow A^m$; for each n-tuple of (constantly supplied) input data a and k-tuple initial values x, '$f_N(t,a,x)$' will denote the output of N at time t.

Unspecified Values. Observe that in the preceding example, for the first three time cycles ($t=0,1,2$) the network does not produce any meaningful output; during this time the initial internal values are being flushed out of the network. Said differently, the network has an *initialisation phase* (of three cycles). Intuitively, as users of the network, we are not interested in the network's output during the initialisation phase; having entered an input n we only want to know if and when $f(n)$ appears as output. Furthermore, specification of the network's output during the initialisation phase clutters the formal specification of the network's output (1) with irrelevant detail making it harder to read and understand.

We will introduce the symbol 'u' to denote a value in which we are not interested. Using this symbol the network specification (1) can be rephrased more succinctly in the following way:

$$f_N(t,n,x) = \begin{cases} u & \text{if } 0 \le t < 3 \\ f(n) & \text{if } t \ge 3 \end{cases} \qquad (2)$$

Introduction of the symbol u reflects *methodological* concerns: we read 'u' as '*an unspecified value*' and this makes a network specification easier to read and thus the intended purpose of the network is more readily determined. For example, (2) reads: 'for each time t, input n, and vector of initial values x, the output of the network is unspecified for the first three cycles, $f(n)$ otherwise'; clearly, this is more intelligible than the previous specification (1). For the same reasons we will also use u in tabulations of network behaviour; for example, Table 2.3 is Table 2.2 but with u replacing any entry in the table in which we are not interested; again this clarifies the behaviour of the network considerably.

time	module				
	1	2	3	4	5
0	u	u	u	u	u
1	$n-1$	$n+1$	u	u	u
2	$n-1$	$n+1$	$(n-1)+(n+1)$	$(n-1)\times(n+1)$	u
3	$n-1$	$n+1$	$(n-1)+(n+1)$	$(n-1)\times(n+1)$	$(n-1)+(n+1)$ $+(n-1)\times(n+1)$

Table 2.3 - Retabulation of the network N using the symbol u.

Note that since we are ultimately concerned with the *formal* specification of synchronous algorithms, the formal status of 'u' is at issue: what exactly *is* u? We will return to this question at the end of the next section. (For the time being, the reader should regard u as abbreviating 'some function of time and initial internal values, but not input data'; the conscientious reader may care to verify that every occurrence of u has this property.)

2.2 STREAM PROCESSING AND CORRECTNESS SPECIFICATIONS.

So far we have only considered synchronous computation on a single input which the sources continuously supplied to the network for $t = 0,1,2,\ldots$. What happens when the data supplied by the sources varies with time?

We have seen how a synchronous algorithm defines a clock $T = \{0,1,2,\ldots\}$ where each $t \in T$ is a time-point at which the modules of the underlying network are synchronised. In particular, we imagine a module connected to a source to require an input at these times (and only these). Thus, if we can contrive for the source to supply data to the network at times $t \in T$ (only) then the action of reading in input will be synchronised with the synchronised computations of the network. Moreover, it is intuitively clear from synchronous operation of a network that each time $t \in T$ is a time at which (new) output is available at the sinks.

To be more precise, let N be an n-source synchronous network over data set A, and let $T = T_N$. If we now define an (n-ary) *stream* to be a map $\underline{a} : T \longrightarrow A^n$, then for each $t \in T$, $\underline{a}(t) \in A^n$ is an appropriate input for the network; specifically, $\underline{a}(t)$ is, by definition, *scheduled* to arrive at an appropriate time.

These remarks suggest that when executed on a *stream* of input data, the output of a general (n-source, k-module, m-sink) network can be specified by a function F_N of functionality $F_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^m$; of course, here the idea is that for each input stream $\underline{a} : T \longrightarrow A^n$ and vector $x \in A^k$ of initial values, '$F_N(t, \underline{a}, x)$' is to denote the output of N at time t. We will explain how to define F_N for a general network N in Section 2.4; for the time being here is an example:

Example. If we wish to execute (the algorithm depicted by) N of Figure 2.1 on a sequence of inputs $n_0, n_1, \ldots,$, it is clear we need not wait for one computation to finish before entering more data: from the structure of N, and from its synchronous behaviour, we notice that computations may be *pipelined* with several (exactly: three) inputs being executed upon simultaneously. Moreover (again from the structure of N and its synchronous operation), it is clear that we can enter new data at every tick of the clock without disturbing any computations on previously entered data. Generally, the minimum time interval between the times at which successive new data may be loaded into a synchronous algorithm is called the *period* of the algorithm.

Of course, N has period 1. Thus if we wish to execute N on the sequence $n_0, n_1, \ldots,$ we can supply the sequence as a stream $\underline{n} : T \longrightarrow \mathbb{N}$ such that $n(t) = n_t$ for $t = 0, 1, 2, \ldots$. In this situation the value held by the ith module at time t for $i = 1, \ldots, 5$, for $t = 0, \ldots, 3$, and for the general step $t \geq 3$, is as tabulated in Table 2.4.

time	module				
	1	2	3	4	5
0	u	u	u	u	u
1	$\underline{n}(0)-1$	$\underline{n}(0)+1$	u	u	u
2	$\underline{n}(1)-1$	$\underline{n}(1)+1$	$(\underline{n}(0)-1)+(\underline{n}(0)+1)$	$(\underline{n}(0)-1)\times(\underline{n}(0)+1)$	u
3	$\underline{n}(2)-1$	$\underline{n}(2)+1$	$(\underline{n}(1)-1)+(\underline{n}(1)+1)$	$(\underline{n}(1)-1)\times(\underline{n}(1)+1)$	$(\underline{n}(0)-1)+(\underline{n}(0)+1)$ $+(\underline{n}(0)-1)\times(\underline{n}(0)+1)$
. . .	. . .	. . .	. . .	. . .	. . .
t	$\underline{n}(t-1)-1$	$\underline{n}(t-1)+1$	$(\underline{n}(t-2)-1)+(\underline{n}(t-2)+1)$	$(\underline{n}(t-2)-1)\times(\underline{n}(t-2)+1)$	$(\underline{n}(t-3)-1)+(\underline{n}(t-3)+1)$ $+(\underline{n}(t-3)-1)\times(\underline{n}(t-3)+1)$

Table 2.4 - Tabulation of the network N in operation on a stream.

Clearly, from the final column of Table 2.4, the output of N is given by the map $F_N : T \times [T \longrightarrow \mathbb{N}] \times \mathbb{N}^5 \longrightarrow \mathbb{N}$ defined by

$$F_N(t,\underline{n},x) = \begin{cases} u & \text{if } 0 \le t < 3 \\ f(\underline{n}(t-3)) & \text{if } t \ge 3 \end{cases} \qquad (3)$$

for each $t \in T$, $\underline{n} : T \longrightarrow \mathbb{N}$, and $x \in \mathbb{N}^5$. □

2.2.1 Stream Processing.

We have suggested that the output of a general network N can be defined by a map

$$F_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^m$$

An alternative idea is to use a function

$$G_N : [T \longrightarrow A^n] \times A^n \longrightarrow [T \longrightarrow A^m]$$

with the intention that for each $\underline{a} : T \longrightarrow A^n$ and $x \in A^k$, $G_N(\underline{a},x)$ is the stream of output data generated by $\underline{a}$ and x. Although we have yet to say what 'F_N' actually is for a given network N, it should be clear that given such a function we can define G_N by

$$G_N(\underline{a},x)(t) = F_N(t,\underline{a},x) \qquad (4)$$

for each $\underline{a} : T \longrightarrow A^n$, $x \in A^k$, and $t \in T$.

We are suggesting that instead of '$F_N(t,\underline{a},x)$' we write '$G_N(\underline{a},x)(t)$' for the output at time t given input $\underline{a}$ and initial values x. Mathematically, the functions F_N and G_N are very similar: essentially, the only difference is the order in which the arguments are supplied. (G_N is actually called the *curried form* of F_N.) From a methodological viewpoint however, G_N, as a mapping from streams (and initial values) into streams, is a more pleasing form of specification since it is natural to think of a synchronous algorithm as a 'black box' that maps input streams to output streams. Indeed, it is natural to specify the *correctness* of a synchronous algorithm in terms of functions from streams into streams as we will now explain.

2.2.2 User Specifications.

Typically, the purpose of an (n-source, m-sink) synchronous network is to evaluate some $f : A^n \longrightarrow A^m$ on each element of a sequence $a_0, a_1, a_2, \ldots$ of input data; that is, to compute the sequence $f(a_0), f(a_1), f(a_2), \ldots$. It is natural then to represent sequences as streams and to specify the task at hand as a stream transformation $\Phi_N : [T \longrightarrow A^n] \longrightarrow [T \longrightarrow A^m]$.

For example, given integers $\lambda_1, \lambda_2 \ge 1$ with $\lambda_1 \ge \lambda_2$, consider the stream transformation $\Phi_N : [T \longrightarrow A^n] \longrightarrow [T \longrightarrow A^m]$ where for each $\underline{a} : T \longrightarrow A^n$, $\Phi_N(\underline{a})$ is the stream defined by

$$\Phi_N(\underline{a})(t) = \begin{cases} u & \text{if } 0 \le t < \lambda_1 \\ f(\underline{a}(t-\lambda_2)) & \text{if } t \ge \lambda_1 \end{cases}$$

for each $t \in T$. Clearly, Φ_N asks for an n-source, m-sink synchronous network over A that has initialisation time λ_1, has length of computation λ_2, computes the function f, and has period 1.

Alternatively, for some $\lambda, \pi \ge 1$, suppose $\Phi_N(\underline{a})(t)$ satisfies the following equation for each $\underline{a} : T \longrightarrow A^n$ and $t \in T$:

$$\Phi_N(\underline{a})(t) = \begin{cases} u & \text{if } 0 \le t < \lambda+\pi \text{ or } (t-\lambda) \bmod \pi \ne 0 \\ f(\underline{a}(t-\pi)) & \text{if } t \ge \lambda+\pi \text{ and } (t-\lambda) \bmod \pi = 0 \end{cases}$$

Then this Φ_N asks for a network that has initialisation time λ, has length of computation π, computes the

function f, and has period π.

Both of the previous specifications are instances of the following more general specification:

2.2.3 Definition.

Let N be an n-source, m-sink, synchronous network over data set A with clock $T=T_N$. Also let $R:T\longrightarrow\mathbb{B}$ be any predicate, and let $\delta:T\longrightarrow T$ and $f:A^n\longrightarrow A^m$ be any functions. We say $\Phi_N:[T\longrightarrow A^n]\longrightarrow[T\longrightarrow A^m]$ is a *user specification* of N if for each $\underline{a}:T\longrightarrow A^n$ and each $t\in T$ $\Phi_N(\underline{a})(t)$ satisfies the following equation:

$$\Phi_N(\underline{a})(t)=\begin{cases}u & \text{if } \sim R(t)\\ f(\underline{a}(\delta(t))) & \text{if } R(t)\end{cases} \tag{5}$$

□

A user specification of a network is so called because it is often easy for a user of the network to read off what input/output characteristics the network has, irrespective of internal structure. In the general form (5) the predicate R (the *ready condition*) tells the user when output is ready, and when this is so, the function δ (the *input schedule*) tells the user which input gave rise to the current output.

(We expect R to have the property that there is at least one $t\in T$ such that $R(t)$ holds; otherwise N never produces any (meaningful) output. Similarly, we expect δ to have the property that whenever $t\in T$ is such that $R(t)$ holds, $\delta(t)<t$: the case $\delta(t)=t$ is unrealistic since if this is true for a given $t\in T$, then the specification tells us that $f(\underline{a}(\delta(t)))=f(\underline{a}(t))$ is computed in zero time. The case $\delta(t)>t$ is equally counterintuitive: if $\delta(t)>t$ then the specification says that the network's output at time t is dependent on *future* input! Whilst we will not rule out such 'unreasonable' properties of R and δ, they will never arise in this thesis.)

There are two other points about Definition 2.2.3 that are worth noting. First, whilst the intention and intuition behind (5) are quite clear, the above definition is not entirely mathematically rigorous because of the unspecified value symbol 'u'. To circumvent this problem, whenever a user specification is encountered in a context where mathematical rigour is an issue, we will regard the definition of a user specification to be as given above with exception that (5) is replaced with the following condition:

$$R(t)\Longrightarrow\Phi_N(\underline{a})(t)=f(\underline{a}(\delta(t)))$$

This makes a user specification totally rigorous since there is no mention of unspecified values.

Second, notice that a user specification Φ_N is properly subscripted by a network symbol 'N' since a user specification is defined over streams over the clock $T=T_N$ of a synchronous network N.

2.2.4 Network Correctness.

We have intimated above that a user specification serves as a correctness specification. How do we formalise the idea that a network meets a specification?

Suppose N is a synchronous network that is supposed to meet the specification $\Phi_N:[T\longrightarrow A^n]\longrightarrow[T\longrightarrow A^m]$. Then given an input stream $\underline{a}:T\longrightarrow A^n$, the output of N at time t must be $\Phi_N(\underline{a})(t)$. However, we have said above that the output of a synchronous network N can be defined

by a function $G_N : [T \longrightarrow A^n] \times A^k \longrightarrow [T \longrightarrow A^m]$ with $G_N(\underline{a},x)(t)$ denoting the output at time t. Note that G_N depends on initial values whereas Φ_N does not. Clearly, one way of expressing the correctness of N is to say that N meets the specification Φ_N if there is some vector $x \in A^k$ of initial values such that $G_N(\underline{a},x) = \Phi_N(\underline{a})$ for all input streams $\underline{a} : T \longrightarrow A^n$; that is if

$$(\exists x \in A^k)(\forall \underline{a} : T \longrightarrow A^n)(\forall t \in T) \ \ (G_N(\underline{a},x)(t) = \Phi_N(\underline{a})(t))$$

However, from a methodological perspective, it is a poor idea for an algorithm's correctness to be dependent on specific initial values, since in fabricating or implementing the algorithm it may be difficult to guarantee that initial values will always be what the algorithm needs to function correctly. For this reason we will say that N meets Φ_N if G_N satisfies the stronger condition:

$$(\forall x \in A^k)(\forall \underline{a} : T \longrightarrow A^n)(\forall t \in T) \ \ (G_N(\underline{a},x)(t) = \Phi_N(\underline{a})(t))$$

or, given our remarks concerning unspecified values above, if

$$(\forall x \in A^k)(\forall \underline{a} : T \longrightarrow A^n)(\forall t \in T) \ \ (R(t) \Longrightarrow G_N(\underline{a},x)(t) = \Phi_N(\underline{a})(t))$$

where R is Φ_N's ready condition.

We note that the above correctness conditions could be phrased solely in terms of the function F_N, given that G_N is defined in terms of F_N (by (4)).

2.3 SORTING AS A CASE STUDY.

In this section we shall introduce two synchronous algorithms that sort n elements from some set D that is linearly ordered by some relation $\leq_D$. These algorithms, as synchronous networks, meet user specifications of the general form

$$\Phi_N : [T \longrightarrow D^n] \longrightarrow [T \longrightarrow D^n]$$

where for each $\underline{a} : T \longrightarrow D^n$,

$$\Phi_N(\underline{a})(t) = \begin{cases} u & \text{if } \sim R(t) \\ sort(\underline{a}(\delta(t))) & \text{if } R(t) \end{cases}$$

Here $sort : D^n \longrightarrow D^n$ is the map defined by

$$sort(x_1,\ldots,x_n) = (x_{\pi(1)},\ldots,x_{\pi(n)}) \Longleftrightarrow x_{\pi(1)} \leq_D \cdots \leq_D x_{\pi(n)}$$

where π is a permutation of $\{ 1,\ldots,n \}$.

We will show by means of examples that both of our algorithms *are* sorters, and, indeed, that with respect to the clocks defined by these algorithms, both sort n elements in (less than) $n+1$ time units.

We will first describe the algorithms informally, and then show how the algorithms are synchronous networks in the sense of Section 2.1; to do this we must show how the algorithm's modules can be formally specified (by single valued functions on D). Also note that for simplicity we will describe the operation of the algorithms on a single fixed input; we will explain how to modify the algorithms so that they sort on streams of data later.

2.3.1 The OE Sorter.

The first sorting algorithm is called *Odd-Even Transposition Sort*, or 'OE' for short.

For sorting n elements the OE sorter is as depicted in Figure 2.5; it comprises an array of n modules $m_1,\ldots,m_n$, where each module m_i may hold one datum from D, and may communicate only

with its neighbouring modules as indicated by the connecting channels in the figure; in addition OE involves a collection $\{In_1,....,In_n\}$ of n sources which supply the input to the array, and a collection $\{Out_1,....,Out_n\}$ of n sinks which receive output from the array.

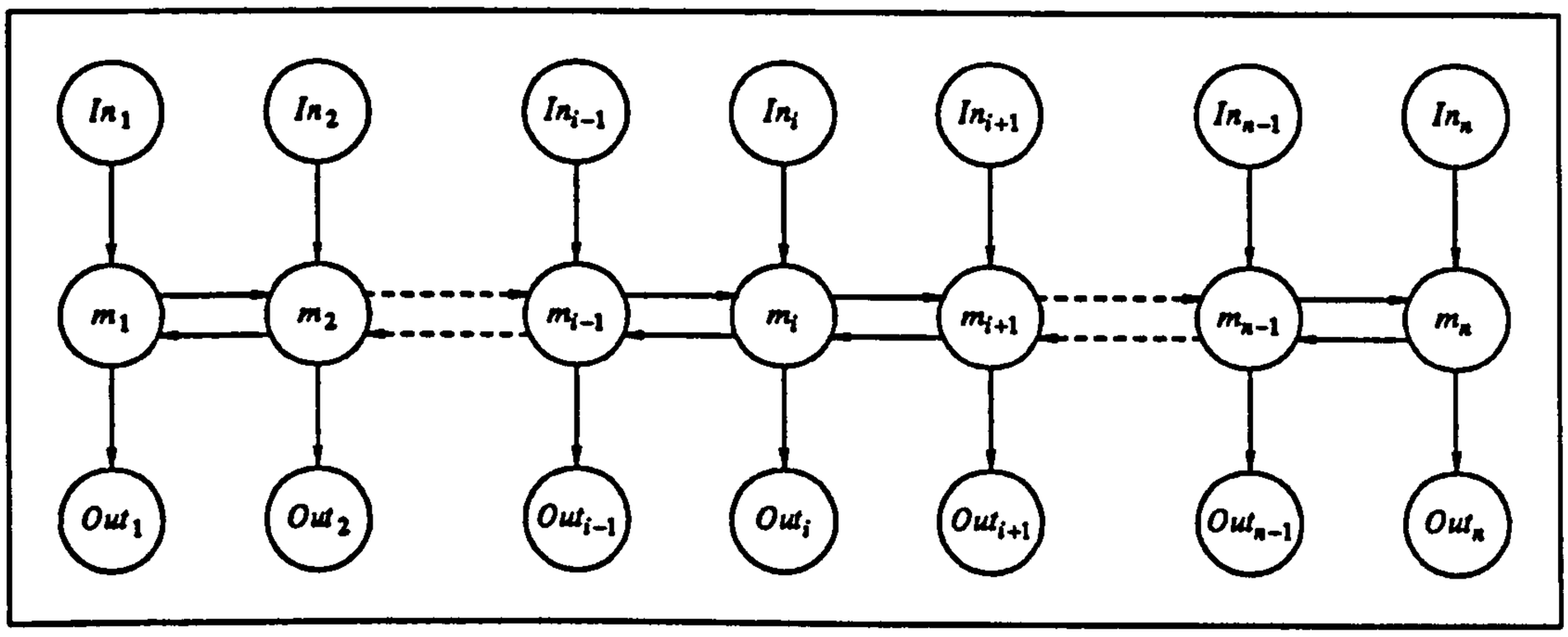


Figure 2.5 - The OE sorter.

Initially, that is, at $t=0$, we assume each module m_i to hold some $x_i \in D$ for $i=1,....,n$. Thereafter, the operation of OE proceeds in steps which determine OE's clock measuring discrete time $t=0,1,2,...$:

Loading (beginning at $t=0$, completed by $t=1$): Each module m_i reads a datum (namely a_i when the input to the array is $a=(a_1,...,a_n)\in D^n$) from source In_i in parallel for $i=1,...,n$.

The operation of each module now depends on time, its action alternating between even and odd time cycles; without loss of generality we will assume that n is even.

Odd steps (beginning at $t=2k+1$, completed by $t=2k+2$ for $k=0,1,2,...$): Each of the constituent modules in the module pairs (m_i,m_{i+1}) for $i=1,3,5,...,n-1$ exchanges (in parallel) values with the other member of the pair (if necessary) such that on completion of the step m_i holds the minimum of the two values and m_{i+1} holds the maximum.

Even steps (beginning at $t=2k+2$, completed by $t=2k+3$ for $k=0,1,2,...$): This step is similar to an odd-step, except that the module pairs are (m_i,m_{i+1}) for $i=2,4,6,...n-2$. During this step m_1 and m_n do nothing; they retain their current values for the duration of the step.

Example. Let $D=\mathbb{N}$. Suppose that $n=6$ and that the n sinks collectively supply the vector (6,5,4,3,2,1) to the array for sorting. We have tabulated the value held by module m_i at time t for $i=1,...,n$ and $t=0,....,7$ in Table 2.6. Notice the data is sorted in $n+1$ steps as claimed.

time	module					
	1	2	3	4	5	6
0	x_1	x_2	x_3	x_4	x_5	x_6
1	6	5	4	3	2	1
2	5	6	3	4	1	2
3	5	3	6	1	4	2
4	3	5	1	6	2	4
5	3	1	5	2	6	4
6	1	3	2	5	4	6
7	1	2	3	4	5	6

Table 2.6 - Tabulation of the OE network in operation;
illustrated for $n = 6$ and input (6,5,4,3,2,1).

□

It should be clear from our informal description that OE satisfies the structural constraints of a synchronous network except for one minor detail:

As we have depicted them in Figure 2.5, OE's modules appear to have more than one output channel (which we have proscribed), but we have only so depicted them to make the figure plain. In fact, since each output channel always carries the value currently held by the module, these channels are conceptually identified.

2.3.2 Module Specification for OE.

To see OE as a synchronous network in the sense of Section 2.1, it remains for us to show how OE's modules may be independently functionally specified, that is, *in isolation from the network*: what are the module specifications $f_i = f_{m_i}$ for $i = 1,\ldots,n$?

For simplicity, let us analyse module m_i for i even, $i \neq n$. (Analysis of the other cases follows easily from this.)

From our informal description of OE, it is clear that in general the value held by m_i depends upon the time t, and the following *four* values from the set D: a, say, the value supplied from above; l and r, say, the values supplied from the left and the right respectively, and v, say, the value currently held by the module. Consider Figure 2.7 which illustrates this situation: we see m_i holding value v and about to receive a, l, and r. Note that since we regard the module's three output channels as identified, each of these channels is shown supplying the same value v (to neighbouring modules). Since m_i depends on time and four data inputs it is appropriate for f_i to have the following functionality:

$$f_i : T \times D^4 \longrightarrow D$$

Thus m_i is a nonautonomous module and the expression '$f_i(t,a,l,v,r)$' denotes the value held by m_i at time $t+1$.

By examining the informal description of the behaviour of OE, we can write down the value held by m_i at time $t+1$ by considering the three cases: $t+1=1$ (or $t=0$); $t+1$ even (or t odd), and $t+1$ odd, $\neq 1$ (or t even, $\neq 0$).

If $t=0$, then m_i loads in the value supplied to it (from above) by In_i; in the expression '$f_i(0,a,l,v,r)$' (which denotes the value held by m_i at time $t+1$), this value is 'a' and so it is appropriate to define

$$f_i(0,a,l,v,r) = a$$

If t is odd, then m_i compares the value held by the module on its left (m_{i-1}) at time t with its own value and retains the maximum; in the expression '$f_i(t,a,l,v,r)$' these values are denoted by 'l' and 'v' respectively, and so we define

$$f_i(t,a,l,v,r) = \max\{ l,v \}$$

Similarly, if t is even (but nonzero) then m_i compares the value held by the module on its right (m_{i+1}) at time t with its own value and retains the minimum; in the expression '$f_i(t,a,l,v,r)$' these values are denoted by 'r' and 'v' respectively, and so we define

$$f_i(t,a,l,v,r) = \min\{ v,r \}$$

Putting these three cases together we obtain:

$$f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t=0 \\ \max\{ l,v \} & \text{if } t \textit{ odd} \\ \min\{ v,r \} & \text{if } t \textit{ even}, \neq 0 \end{cases}$$

for each $t \in T$ and $a,l,v,r \in D$.

In a similar way we can derive the module specifications f_i for $i = 1,\ldots,n$. These are:

$$f_1 : T \times D^3 \longrightarrow D$$

where for each $t \in T$, and $a,v,r \in D$,

$$f_1(t,a,v,r) = \begin{cases} a & \text{if } t=0 \\ \min\{ v,r \} & \text{if } t \textit{ odd} \\ v & \text{if } t \textit{ even}, \neq 0 \end{cases}$$

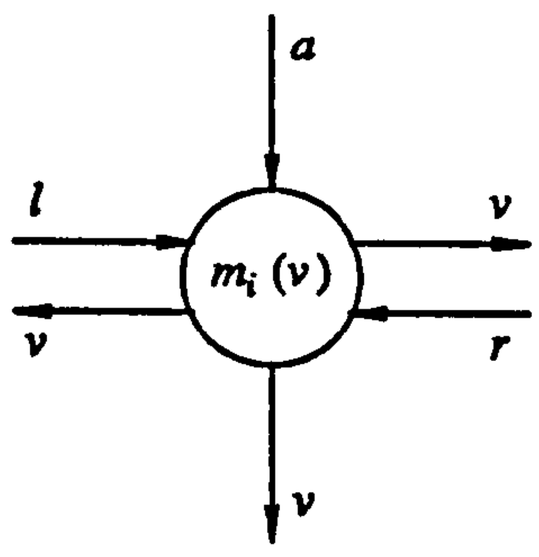


Figure 2.7 - A non-terminal module from the OE network.

For i even, $i \neq n$,

$$f_i : T \times D^4 \longrightarrow D$$

where for each $t \in T$, and $a,l,v,r \in D$,

$$f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t = 0 \\ \max\{l,v\} & \text{if } t \text{ odd} \\ \min\{v,r\} & \text{if } t \text{ even}, \neq 0 \end{cases}$$

For i odd, $i \neq 1$,

$$f_i : T \times D^4 \longrightarrow D$$

where for each $t \in T$, and $a,l,v,r \in D$,

$$f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t = 0 \\ \min\{v,r\} & \text{if } t \text{ odd} \\ \max\{l,v\} & \text{if } t \text{ even}, \neq 0 \end{cases}$$

and,

$$f_n : T \times D^3 \longrightarrow D$$

where for each $t \in T$, and $a,l,v \in D$,

$$f_n(t,a,l,v) = \begin{cases} a & \text{if } t = 0 \\ \max\{l,v\} & \text{if } t \text{ odd} \\ v & \text{if } t \text{ even}, \neq 0 \end{cases}$$

We begin to formalise the OE sorting algorithm by functionally specifying the algorithm's modules. Soon we will formalise the complete algorithm by using these module specifications to obtain a functional specification of the underlying network.

2.3.3 The EOE Sorter.

The second of our two sorters is called *Expanded Odd-Even Sort*, or 'EOE' for short.

For sorting n elements the EOE sorter is as depicted in Figure 2.8; it comprises an array of $n+1$ columns of n modules $m_{i,j}$ for $i = 1,...,n$ and $j = 0,...,n$, wherein $m_{i,j}$ denotes the ith module on the jth column. Similar to OE, each module holds a single datum and communicates only with neighbouring modules as indicated in the figure. Additionally, EOE involves n sources and n sinks whose purpose is identical to those of OE.

Initially, that is, at $t = 0$, we imagine each module $m_{i,j}$ to hold some $x_{i,j} \in D$ for $i = 1,...,n$ and for $j = 0,...,n$. Thereafter, operation of EOE proceeds in steps determining EOE's clock:

Every $m_{i,j}$ now performs the following general step in parallel for $i = 1,...,n$ and for $j = 0,...,n$:

General Step: The action that a module $m_{i,j}$ takes is determined according to whether j is zero, odd, or even.

Case (i): $j = 0$. Each module in column 0 (viz. $m_{i,0}$, for $i = 1,...,n$) reads a datum from its source in parallel.

Case (ii): j odd. The module $m_{i,j}$ first reads the value held by $m_{i,j-1}$. Then, if i is even, $m_{i,j}$ exchanges this value for the value held by $m_{i-1,j-1}$ (if necessary), such that on completion of the step $m_{i,j}$ holds the maximum of these two values. If i is odd then $m_{i,j}$ exchanges its value for the value held by $m_{i+1,j-1}$ (again, if necessary), such that on completion of the step $m_{i,j}$ holds the minimum of the two values.

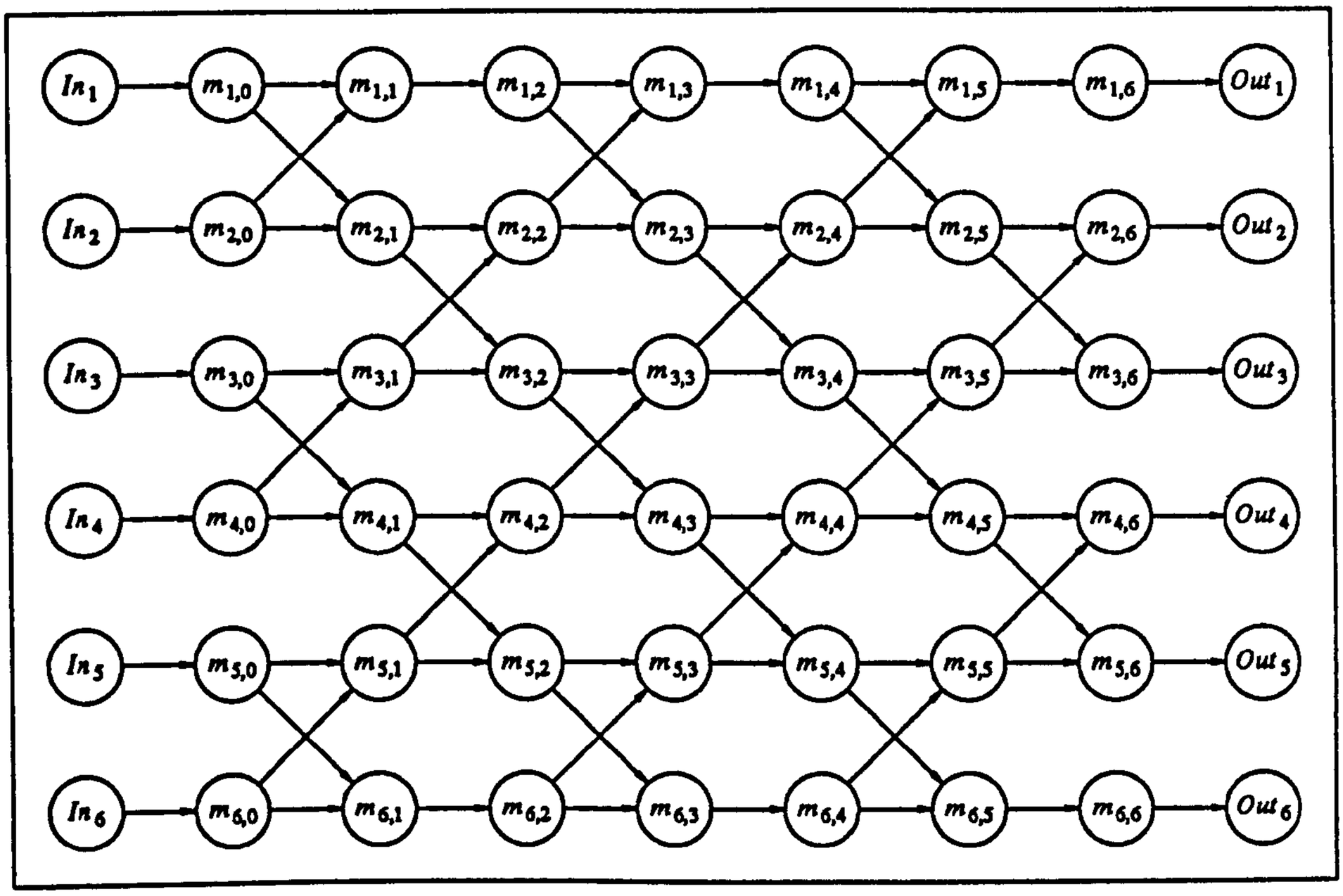


Figure 2.8 - The EOE sorter illustrated for $n = 6$.

Case (iii): j even. In this case each $m_{i,j}$ for $1 < i < n$ behaves as in the preceding case (dependent on whether $m_{i,j}$ is connected to $m_{i-1,j-1}$ or $m_{i+1,j-1}$). During this step $m_{1,j}$ and $m_{n,j}$ do nothing.

If we operate a similar (to the case of OE) convention concerning the number of output channels that EOE's modules have, then it is clear that EOE satisfies the structural constraints of a synchronous network.

Example. Similar to the OE example, let $D = \mathbb{N}$ and take $n = 6$, and suppose the n sinks collectively and constantly supply the vector (6,5,4,3,2,1). Then the value held by $m_{i,j}$ at time t is as tabulated in Table 2.9 for $i = 1,\ldots,n$, $j = 0,\ldots,n$ and for $t = 0,\ldots,7$. □

2.3.4 Module Specification for EOE.

Similar to the OE network, we can find the module specifications $f_{i,j}$ of EOE's modules for $i = 1,\ldots,n$ and $j = 0,\ldots,n$ by considering what inputs a given module has.

Similar to OE we will introduce names for a module's (possible) inputs. Let a denote the value supplied from the left and above, let l denote the value supplied from the left, and let b denote the value supplied from the left and below. By examining EOE's informal description we arrive at the following specifications for $i = 1,\ldots,n$ and for $j = 0,\ldots,n$ (note that $i+j$ is odd exactly when i is odd and j is even or vice versa, and that $i+j$ is even exactly when both i and j are both even or both odd):

time														
	$t=0$							$t=1$						
row	column							column						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
1	u	u	u	u	u	u	u	6	u	u	u	u	u	u
2	u	u	u	u	u	u	u	5	u	u	u	u	u	u
3	u	u	u	u	u	u	u	4	u	u	u	u	u	u
4	u	u	u	u	u	u	u	3	u	u	u	u	u	u
5	u	u	u	u	u	u	u	2	u	u	u	u	u	u
6	u	u	u	u	u	u	u	1	u	u	u	u	u	u

time														
	$t=2$							$t=3$						
row	column							column						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
1	6	5	u	u	u	u	u	6	5	5	u	u	u	u
2	5	6	u	u	u	u	u	5	6	3	u	u	u	u
3	4	3	u	u	u	u	u	4	3	6	u	u	u	u
4	3	4	u	u	u	u	u	3	4	1	u	u	u	u
5	2	1	u	u	u	u	u	2	1	4	u	u	u	u
6	1	2	u	u	u	u	u	1	2	2	u	u	u	u

time														
	$t=4$							$t=5$						
row	column							column						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
1	6	5	5	3	u	u	u	6	5	5	3	3	u	u
2	5	6	3	5	u	u	u	5	6	3	5	1	u	u
3	4	3	6	1	u	u	u	4	3	6	1	5	u	u
4	3	4	1	6	u	u	u	3	4	1	6	2	u	u
5	2	1	4	2	u	u	u	2	1	4	2	6	u	u
6	1	2	2	4	u	u	u	1	2	2	4	4	u	u

time														
	$t=6$							$t=7$						
row	column							column						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
1	6	5	5	3	3	1	u	6	5	5	3	3	1	1
2	5	6	3	5	1	3	u	5	6	3	5	1	3	2
3	4	3	6	1	5	2	u	4	3	6	1	5	2	3
4	3	4	1	6	2	5	u	3	4	1	6	2	5	4
5	2	1	4	2	6	4	u	2	1	4	2	6	4	5
6	1	2	2	4	4	6	u	1	2	2	4	4	6	6

Table 2.9 - Tabulation of the EOE network in operation;
illustrated for $n=6$ and input (6,5,4,3,2,1).

For $i=1,\ldots,n$,

$$f_{i,0}: D \longrightarrow D$$

where for each $l \in D$,

$$f_{i,0}(l) = l$$

For $i+j$ odd, $i \neq 1$, $j \neq 0$,

$$f_{i,j} : D^2 \longrightarrow D$$

where for each $a,l \in D$,

$$f_{i,j}(a,l) = \max\{ a,l \}$$

For $i+j$ even, $i \neq n$, $j \neq 0$,

$$f_{i,j} : D^2 \longrightarrow D$$

where for each $l,b \in D$,

$$f_{i,j}(l,b) = \min\{ l,b \},$$

and for $i = 1$ with j even ($\neq 0$), and $i = n$ with j odd,

$$f_{i,j} : D \longrightarrow D$$

where for each $l \in D$,

$$f_{i,j}(l) = l$$

2.3.5 Sorting on Streams.

Given a sequence $a_0,a_1,a_2,...$ of vectors from D^n, let us explore how OE and EOE can be made to sort $a_0,a_1,a_2,...$ in turn.

In the case of OE it should be clear from the informal description of the algorithm there must be a delay of at least $n+1$ time cycles between the times at which we load successive input vectors: loading data any faster than this will overwrite the partially sorted previous input vector. Indeed, in our informal description of OE, we said that the modules only ever load in data once to circumvent this problem; this design decision is reflected in the formal module specifications $f_1,...,f_n$: for $i = 1,...,n$, $f_i(t,a,...) = a$ only when $t = 0$.

In the case of EOE however, it is clear that we can load in new data with every tick of the clock without disturbing the computations on previously entered data. To be more precise, let $\underline{a} = (\underline{a}_1,...,\underline{a}_n) : T \longrightarrow D^n$ with the intention that for $i = 1,...,n$ $\underline{a}_i : T \longrightarrow D$ is the input stream supplied to EOE by the ith source. It is intuitively clear that EOE meets the user specification $\Phi_{EOE} : [T \longrightarrow D^n] \longrightarrow [T \longrightarrow D^n]$ where

$$\Phi_{EOE}(\underline{a})(t) = \begin{cases} u & \text{if } 0 \leq t < n+1 \\ sort(\underline{a}(t-n-1)) & \text{if } t \geq n+1 \end{cases} \qquad (6)$$

for each $\underline{a} : T \longrightarrow D^n$ and $t \in T$. Thus, as t increases from $t = n+1$, the output from EOE is $sort(\underline{a}(0))$, $sort(\underline{a}(1))$, $sort(\underline{a}(2))$,...; that is, EOE sorts every input supplied to it.

Now let us return to OE. Of course, it is impossible for OE to sort every input supplied as a stream $\underline{a} : T \longrightarrow D^n$ since at best we can only execute OE on every $(n+1)$th input supplied by the sources: intuitively, we load $\underline{a}(0)$ at time $t = 0$ (so that it is held by OE's modules by time $t = 1$), and by time $t = n+1$ $\underline{a}(0)$ has been sorted so we can begin to load the next input vector at this time (so that it is held by OE's modules by time $t = n+2$). However, at $t = n+1$ the input to OE is $\underline{a}(n+1)$; the inputs $\underline{a}(1),...,\underline{a}(n)$ have been lost or disregarded. Similarly, since $\underline{a}(n+1)$ has been loaded by time $t = n+2$ we anticipate that $\underline{a}(n+1)$ will be sorted by time $t = 2n+2$ (since $2n+2 = (n+1)+(n+1)$), but again $\underline{a}(n+2),...,\underline{a}(2n+1)$ will have been disregarded.

Instead of trying to sort every input, let us focus on the problem of redefining OE so that it sorts every $(n+1)$th input; that is, the modified OE algorithm is to implement the user specification

$\Phi_{OE} : [T \longrightarrow D^n] \longrightarrow [T \longrightarrow D^n]$ where for each $\underline{a} : T \longrightarrow D^n$ and each $t \in T$,

$$\Phi_{OE}(\underline{a})(t) = \begin{cases} u & \text{if } t = 0 \text{ or } t \ mod\ (n+1) \neq 0 \\ sort(\underline{a}(t-n-1)) & \text{if } t \neq 0 \text{ and } t \ mod\ (n+1) = 0 \end{cases} \quad (7)$$

Let us call OE 'ready at time t' if t is such that the next $n+1$ steps to be executed by OE are, in order, a loading step, and then, an odd-step followed by an even-step $n/2$ times. Now, whilst we have yet to prove that OE actually sorts any given input, let us agree that OE has the following property: if OE is ready at any time t (when the input to OE is $\underline{a}(t)$ of course), then OE's modules collectively hold $sort(\underline{a}(t))$ at time $t+n+1$. Thus without modification OE is ready only once, at $t=0$, and by hypothesis OE holds $sort(\underline{a}(0))$ at $t=n+1$.

Our strategy for making OE sort every $(n+1)$th input is to modify the algorithm in such a way that it is ready at time t iff t is of the form $t=m(n+1)$ for some integer $m \geq 0$; if we can do this, then for t of the form $m(n+1)$ we have by the agreed hypothesis that OE holds $sort(\underline{a}(m(n+1)))$ at $t=m(n+1)+(n+1)=(m+1)(n+1)$ at which time OE is again ready since $(m+1)(n+1)\ mod\ (n+1)=0$. Furthermore, if $t=(m+1)(n+1)$ for some $m \geq 0$, then $t \neq 0$ and $t\ mod\ (n+1)=0$. In other words OE will meet the specification Φ_{OE} as required.

Consider what happens if we replace the loading clause in OE's informal description (Section 2.3.1) with the following new clause (but keeping the odd- and even-step clauses the same):

Loading (beginning at $t=m(n+1)$ and completed by time $t=m(n+1)$ for $m=0,1,2,...$): Each module m_i reads a datum (namely a_i when the input to the array is $a=(a_1,...,a_n) \in D^n$) from source In_i in parallel for $i=1,...,n$.

With the modified loading clause, OE now fulfills the first requirement of being ready when t is of the form $t=m(n+1)$ since the next step will be a loading step. However, the new loading clause is not enough to guarantee that OE will be ready at times t of the form $t=m(n+1)$: we are assuming that n is even, and so for m odd, $m(n+1)$ is also odd, and thus the first step executed after the loading step begins at $t=m(n+1)+1$ which is *even*; thus OE will perform an even-step after the loading step, and so OE is not ready at time $m(n+1)$; presumably OE will not sort the input loaded at time $t=m(n+1)$, namely $\underline{a}(m(n+1))$. However, observe that when t is of the form $t=m(n+1)+2k+1$ for $k=0,...,n/2-1$ (which is when we want OE to begin performing odd-steps), t may not be odd but t' defined by $t'=t\ mod\ (n+1)$ certainly is. Similarly, when when t is of the form $t=m(n+1)+2k$ for $k=1,...,n/2$ (which is when we want OE to begin performing even-steps), whilst t may not even, t' again defined by $t'=t\ mod\ (n+1)$ certainly is. Thus, if we stipulate that an odd-step begins when $t\ mod\ (n+1)$ is odd, and that even-steps begin when $t\ mod\ (n+1)$ is even, then OE will be ready precisely when $t\ mod\ (n+1)=0$, that is when t is of the form $t=m(n+1)$ for some $m \geq 0$, and thus OE will meet Φ_{OE} as agreed above.

The full revised OE algorithm is as follows:

Loading (beginning whenever $t\ mod\ (n+1)$ is odd and completed by $t=t+1$): Each module m_i reads a datum (namely a_i when the input to the array is $a=(a_1,...,a_n) \in D^n$) from source In_i in parallel for $i=1,...,n$.

Odd steps (beginning whenever t *mod* $(n+1)$ is odd and completed by time $t+1$): Each of the constituent modules in the module pairs (m_i, m_{i+1}) for $i = 1,3,5,\ldots,n-1$ exchanges (in parallel) values with the other member of the pair (if necessary) such that on completion of the step m_i holds the minimum of the two values and m_{i+1} holds the maximum.

Even steps (beginning whenever t *mod* $(n+1)$ is even but nonzero and completed by time $t+1$): This step is similar to the Odd-step, except that the module pairs are (m_i, m_{i+1}) for $i = 2,4,6,\ldots n-2$. During this step m_1 and m_n do nothing; they retain their current values for the duration of the step.

2.3.6 Respecification of OE.

On examining the revised OE algorithm above, it is not difficult to see that by using the same principal of module specification that we have previously used, we obtain the following revised module specifications $f_1,\ldots,f_n$:

$$f_1 : T \times D^3 \longrightarrow D$$

where for each $t \in T$, and $a,v,r \in D$,

$$f_1(t,a,v,r) = \begin{cases} a & \text{if } t \textit{ mod } (n+1) = 0 \\ \min\{v,r\} & \text{if } t \textit{ mod } (n+1) \textit{ odd} \\ v & \text{if } t \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

For i even, $i \neq n$,

$$f_i : T \times D^4 \longrightarrow D$$

where for each $t \in T$, and $a,l,v,r \in D$,

$$f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t \textit{ mod } (n+1) = 0 \\ \max\{l,v\} & \text{if } t \textit{ mod } (n+1) \textit{ odd} \\ \min\{v,r\} & \text{if } t \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

For i odd, $i \neq 1$,

$$f_i : T \times D^4 \longrightarrow D$$

where for each $t \in T$, and $a,l,v,r \in D$,

$$f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t \textit{ mod } (n+1) = 0 \\ \min\{v,r\} & \text{if } t \textit{ mod } (n+1) \textit{ odd} \\ \max\{l,v\} & \text{if } t \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

and,

$$f_n : T \times D^3 \longrightarrow D$$

where for each $t \in T$, and $a,l,v \in D$,

$$f_n(t,a,l,v) = \begin{cases} a & \text{if } t \textit{ mod } (n+1) = 0 \\ \max\{l,v\} & \text{if } t \textit{ mod } (n+1) \textit{ odd} \\ v & \text{if } t \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

2.3.7 Comparison of the Algorithms.

How do OE and EOE compare when viewed as competing designs?

We have seen that in terms of the clocks naturally defined by our two algorithms, both OE and EOE sort in $n+1$ steps. This observation is based on the hypothesis that each module operates in unit time. Furthermore, if we assume that each module requires unit *area* then OE and EOE occupy area (proportional to) n and n^2 respectively.

However, it is clear from contrasting the functional specifications of typical OE- and EOE-modules, that an OE-module is algorithmically more complex than an EOE-module (OE's modules are

nonautonomous), and thus unit costs are intuitively *not* equal (OE's modules being the more expensive). Consequently OE and EOE are not directly comparable in terms of length of computation or area. The difficulty here is that unit costs differ with respect to implicit *external* time and area metrics.

We can make the difference between OE and EOE explicit by contrasting the user specifications that (we claim) they meet. The clock in terms of which we write down a user specification of a synchronous network is parameterised by the network; in the case of OE and EOE, these algorithms determine clocks T_{OE} and T_{EOE} respectively. With respect to these clocks the user specifications (7) and (6) are properly

$$\Phi_{OE} : [T_{OE} \longrightarrow D^n] \longrightarrow [T_{OE} \longrightarrow D^n]$$

where for each $\underline{a} : T_{OE} \longrightarrow D^n$ and $t \in T_{OE}$,

$$\Phi_{OE}(\underline{a})(t) = \begin{cases} u & \text{if } t = 0 \text{ or } t \bmod (n+1) \neq 0 \\ sort(\underline{a}(t-n-1)) & \text{if } t \neq 0 \text{ and } t \bmod (n+1) = 0 \end{cases}$$

and

$$\Phi_{EOE} : [T_{EOE} \longrightarrow D^n] \longrightarrow [T_{EOE} \longrightarrow D^n]$$

where for each $\underline{a} : T_{EOE} \longrightarrow D^n$ and $t \in T_{EOE}$,

$$\Phi_{EOE}(\underline{a})(t) = \begin{cases} u & \text{if } 0 \leq t < n+1 \\ sort(\underline{a}(t-n-1)) & \text{if } t \geq n+1 \end{cases}$$

respectively.

The algorithms can now be more carefully contrasted with respect to these specifications: it is apparent from the algorithms' user specifications that OE and EOE differ as they have different ready conditions and input schedules. Moreover, whilst we can certainly read off, for example, that both algorithms have length of computation $n+1$, here we are informed that these times are not comparable since the user specifications measure these times with explicit reference to different clocks.

2.4 SPECIFYING NETWORK BEHAVIOUR.

We have introduced the idea of a synchronous algorithm in general and we have seen three examples. We have also seen that the output of a synchronous network can be functionally specified in a variety of ways: recall the functions f_N, F_N, and G_N, and the differences between them. In this section we will show how from a network's communication structure, and from specifications of the network's modules, we can systematically obtain a function V_N that formalises the way in which we build up a table of a network N in operation. This V_N is the most important functional specification of network behaviour for it is from V_N that we will *derive* other functional specifications including f_N, F_N, and G_N.

2.4.1 Value Functions.

In order to specify a network's behaviour over time, first notice that the network will be completely specified if we can write down the value held by every module at any time t. Also observe that the value held by any module at any time can always be determined from the input to the network, the values initially held by the network's modules, and the current time t.

If N has n sources then the input to N is formalised by a stream $\underline{a} = (\underline{a}_1,...,\underline{a}_n) : T \longrightarrow A^n$ (with the intention that the ith source supplies $\underline{a}_i(t)$ to the network at time t). Now suppose that N has k modules $m_1,...,m_k$. Then any vector $x = (x_1,...,x_k) \in A^k$ will serve to specify the network's initial internal values (with the intention that m_i initially holds x_i). Now, the value held by each m_i at a time t can always be determined from t, $\underline{a}$, and x, so let us introduce a function $V_i : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A$ for $i = 1,...,k$; these functions we call the network's *value functions*. Intentionally, '$V_i(t,\underline{a},x)$' is read as: 'the value in the i th module at time t given input $\underline{a}$ and initial values x'.

We can put $V_1,...,V_k$ together as the coordinates of a function

$$V_N = (V_1,...,V_k) : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^k$$

which we refer to as *the* value function for network N. For each $t \in T$, $\underline{a} : T \longrightarrow A^n$, and $x \in A^k$, $V_N(t,\underline{a},x)$ is the vector $V_N(t,\underline{a},x) = (V_1(t,\underline{a},x),...,V_k(t,\underline{a},x)) \in A^k$ that tells us the values held by all the network's modules at time t.

We define V_N by exploiting the single most important consequence of a network's synchronous behaviour: *because the network is synchronous, every value in the network at every time is either specified initially, or is specified in terms of the values held at the previous time step.*

Here is an informal two-stage algorithm which describes how to define V_N:

The Synchronous Network Specification Algorithm. Let N have $n > 0$ sources $In_1,...,In_n$, and $k > 0$ modules $m_1,...,m_k$, and for $i = 1,...,k$ let $n_i > 0$ denote the number of input channels that m_i has from adjacent modules and sources, and let m_i be specified by $f_i = f_{m_i}$. Also let $\underline{a} = (\underline{a}_1,...,\underline{a}_n) : T \longrightarrow A^n$ and $x = (x_1,...,x_k) \in A^k$.

Stage 1: define $V_N(0,\underline{a},x)$. To define $V_N(0,\underline{a},x)$ we must define $V_i(0,\underline{a},x)$ for $i = 1,...,k$. Since x_i is intentionally the value held by m_i at time $t = 0$, it is appropriate to define

$$V_i(0,\underline{a},x) = x_i$$

for $i = 1,...,k$.

Stage 2: define $V_N(t+1,\underline{a},x)$. To define $V_N(t+1,\underline{a},x)$ we must define $V_i(t+1,\underline{a},x)$ for $i = 1,...,k$. There are two cases to consider:

Case 1: m_i is autonomous. If m_i is autonomous, then $f_i : A^{n_i} \longrightarrow A$. Now notice that the value m_i holds at time $t+1$ is already specified by f_i in the sense that if $b_1,...,b_{n_i}$ are the values supplied to m_i on its input channels at time t then $f_i(b_1,...,b_{n_i})$ is the value held at time $t+1$. However, for $j = 1,...,n_i$, each b_j is either the value supplied by some source, in which case $b_j = \underline{a}_p(t)$ for some $p \in [1,n]$, or, b_j is the value supplied by another module (possibly itself), in which case $b_j = V_q(t,\underline{a},x)$ for some $q \in [1,k]$. Accordingly we define $V_i(t+1,\underline{a},x)$ by

$$V_i(t+1,\underline{a},x) = f_i(b_1,...,b_{n_i})$$

where for $j = 1,...,n_i$,

$$b_j = \begin{cases} \underline{a}_p(t) & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ V_q(t,\underline{a},x) & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

Case 2: m_i is nonautonomous. If m_i is nonautonomous, then $f_i : T \times A^{n_i} \longrightarrow A$, and similar to Case 1 it is appropriate to define $V_i(t+1,\underline{a},x)$ by

$$V_i(t+1,\underline{a},x) = f_i(t,b_1,\ldots,b_{n_i})$$

where $b_1,\ldots,b_{n_i}$ are as described above. □

There are a number of points arising form the above definition of a network's value function(s) that warrant discussion. Before we do so however, we give some examples to familiarise the reader with the new notation.

2.4.2 Examples.

(1) Let us apply the specification algorithm to the network N of Section 2.1. N has one source and five modules whose functional specifications $f_1,\ldots,f_5$ have the following functionalities:

$$f_1, f_2 : \mathbb{N} \longrightarrow \mathbb{N}$$

and

$$f_3, f_4, f_5 : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

Using the communication structure of N (see Figure 2.1) together with specification algorithm we obtain a value function $V_N = (V_1,\ldots,V_5) : T \times [T \longrightarrow \mathbb{N}] \times \mathbb{N}^5 \longrightarrow \mathbb{N}$ where for each $\underline{a} : T \longrightarrow \mathbb{N}$ and $x = (x_1,\ldots,x_5) \in \mathbb{N}^5$, $V_i(0,\underline{a},x)$ is defined by

$$V_i(0,\underline{a},x) = x_i$$

for $i = 1,\ldots,5$, and

$$\begin{aligned} V_1(t+1,\underline{a},x) &= f_1(\underline{a}(t)) \\ V_2(t+1,\underline{a},x) &= f_2(\underline{a}(t)) \\ V_3(t+1,\underline{a},x) &= f_3(V_1(t,\underline{a},x), V_2(t,\underline{a},x)) \\ V_4(t+1,\underline{a},x) &= f_4(V_1(t,\underline{a},x), V_2(t,\underline{a},x)) \\ V_5(t+1,\underline{a},x) &= f_5(V_3(t,\underline{a},x), V_4(t,\underline{a},x)) \end{aligned}$$

In fact, $f_1,\ldots,f_5$ were the following functions:

$$\begin{aligned} f_1(n) &= n-1 \\ f_2(n) &= n+1 \\ f_3(n,m) &= n+m \\ f_4(n,m) &= n \times m \\ f_5(n,m) &= n+m \end{aligned}$$

(Here n and m are natural numbers of course.) Using these definitions the definition of $V_i(t+1,\underline{a},x)$ for $i = 1,\ldots,5$ can be written out as follows:

$$\begin{aligned} V_1(t+1,\underline{a},x) &= \underline{a}(t) - 1 \\ V_2(t+1,\underline{a},x) &= \underline{a}(t) + 1 \\ V_3(t+1,\underline{a},x) &= V_1(t,\underline{a},x) + V_2(t,\underline{a},x) \end{aligned}$$

$$V_4(t+1,\underline{a},x) = V_1(t,\underline{a},x) \times V_2(t,\underline{a},x)$$
$$V_5(t+1,\underline{a},x) = V_3(t,\underline{a},x) + V_4(t,\underline{a},x)$$

(2) Let us apply the specification algorithm to the OE network. OE has n sources and n modules whose functional specifications $f_1,\ldots,f_n$ have the following functionalities (see Section 2.3.6):

$$f_1, f_n : T \times D^3 \longrightarrow D$$

and

$$f_2,\ldots, f_{n-1} : T \times D^4 \longrightarrow D$$

Using the communication structure of OE (see Figure 2.5) together with the specification algorithm we obtain a value function $V_{OE} = (V_1,\ldots,V_n) : T \times [T \longrightarrow D^n] \times D^n \longrightarrow D^n$ where for each $\underline{a} = (\underline{a}_1,\ldots,\underline{a}_n) : T \longrightarrow D^n$ and $x = (x_1,\ldots,x_n) \in D^n$, $V_i(0,\underline{a},x)$ is defined by

$$V_i(0,\underline{a},x) = x_i$$

for $i = 1,\ldots,n$, and

$$V_i(t+1,\underline{a},x) = \begin{cases} f_i(t,\underline{a}_i(t),V_i(t,\underline{a},x),V_{i+1}(t,\underline{a},x)) & \text{if } i=1 \\ f_i(t,\underline{a}_i(t),V_{i-1}(t,\underline{a},x),V_i(t,\underline{a},x),V_{i+1}(t,x,a)) & \text{if } 1<i<n \\ f_i(t,\underline{a}_i(t),V_{i-1}(t,\underline{a},x),V_i(t,\underline{a},x)) & \text{if } i=n \end{cases}$$

for $i = 1,\ldots,n$.

Again, we can use the definitions of $f_1,\ldots,f_n$ to write down the definitions of $V_i(t+1,\underline{a},x)$ in full:

$$V_1(t+1,\underline{a},x) = \begin{cases} \underline{a}_1(t) & \text{if } t \bmod (n+1) = 0 \\ \min\{V_1(t,\underline{a},x),V_2(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ odd} \\ V_1(t,\underline{a},x) & \text{if } t \bmod (n+1) \text{ even}, \neq 0 \end{cases}$$

For i even, $i \neq n$,

$$V_i(t+1,\underline{a},x) = \begin{cases} \underline{a}_i(t) & \text{if } t \bmod (n+1) = 0 \\ \max\{V_{i-1}(t,\underline{a},x),V_i(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ odd} \\ \min\{V_i(t,\underline{a},x),V_{i+1}(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ even}, \neq 0 \end{cases}$$

For i odd, $i \neq 1$,

$$V_i(t+1,\underline{a},x) = \begin{cases} \underline{a}_i(t) & \text{if } t \bmod (n+1) = 0 \\ \min\{V_i(t,\underline{a},x),V_{i+1}(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ odd} \\ \max\{V_{i-1}(t,\underline{a},x),V_i(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ even}, \neq 0 \end{cases}$$

and,

$$V_n(t+1,\underline{a},x) = \begin{cases} \underline{a}_n(t) & \text{if } t \bmod (n+1) = 0 \\ \max\{V_{n-1}(t,\underline{a},x),V_n(t,\underline{a},x)\} & \text{if } t \bmod (n+1) \text{ odd} \\ V_n(t,\underline{a},x) & \text{if } t \bmod (n+1) \text{ even}, \neq 0 \end{cases}$$

(3) Let us apply the specification algorithm to the EOE network. EOE has n sources and $n+1$ columns of n modules (making a total of $n(n+1)$ modules). Each module m_{ij} is specified by a function f_{ij} whose functionality is as given below for $i = 1,\ldots,n$ and for $j = 0,\ldots,n$:

For $i+j$ odd, $i \neq 1$, $j \neq 0$, and for $i+j$ even, $i \neq n$, $j \neq 0$,

$$f_{i,j} : D \longrightarrow D$$

For other i and j in the ranges $1 \leq i \leq n$ and $0 \leq j \leq n$ respectively,

$$f_{i,j} : D^2 \longrightarrow D$$

Now, according to the specification algorithm, V_{EOE} is a map

$$V_{EOE} : T \times [T \longrightarrow D^n] \times D^{n(n+1)} \longrightarrow D^{n(n+1)}$$

Since we denote a typical module of EOE by $m_{i,j}$ rather than using the notation m_i with i in the range $1 \leq i \leq n(n+1)$, it will be helpful if we carry the double indexing to typical elements x of $D^{n(n+1)}$: we will write

$$x = (x_{1,0},\ldots,x_{n,0},\cdots,x_{1,j},\ldots,x_{n,j},\cdots,x_{1,n},\ldots,x_{n,n})$$

rather than $x = (x_1,\ldots,x_{n(n+1)})$, and we denote a typical element of x by $x_{i,j}$. Similarly, the codomain of V_{EOE} is $D^{n(n+1)}$ and so V_{EOE} has $n(n+1)$ coordinate functions: these we will denote $V_{i,j}$ for $i = 1,\ldots,n$ and $j = 0,\ldots,n$; thus we will write:

$$V_{EOE} = (V_{1,0},\ldots,V_{n,0},\cdots,V_{1,j},\ldots,V_{n,j},\cdots,V_{1,n},\ldots,V_{n,n}) : T \times [T \longrightarrow D^n] \times D^{n(n+1)} \longrightarrow D^{n(n+1)}$$

(Notice the last n coordinates of V_{EOE} correspond to the rightmost column of the EOE network.)

Using the communication structure of EOE (see Figure 2.8) together with the specification algorithm, for each $\underline{a} : T \longrightarrow D^n$ and $x \in D^{n(n+1)}$ we obtain the following definitions of $V_{i,j}(t,\underline{a},x)$ for $i = 1,\ldots,n$ and $j = 0,\ldots,n$:

$$V_{i,j}(0,\underline{a},x) = x_{i,j}$$

and for $i = 1,\ldots,n$ and $j = 0,\ldots,n$,

$$V_{i,j}(t+1,\underline{a},x) = \begin{cases} f_{i,j}(\underline{a}_i(t)) & \text{if } j = 0 \\ f_{i,j}(V_{i-1,j-1}(t,\underline{a},x), V_{i,j-1}(t,\underline{a},x)) & \text{if } i+j \text{ odd, } i \neq 1,\ j \neq 0 \\ f_{i,j}(V_{i,j-1}(t,\underline{a},x), V_{i+1,j-1}(t,\underline{a},x)) & \text{if } i+j \text{ even, } i \neq n,\ j \neq 0 \\ f_{i,j}(V_{i,j-1}(t,\underline{a},x)) & \text{otherwise} \end{cases}$$

Discussion. Notice how specification 'by value functions' leads to compact and yet comprehensible formal definitions of network behaviour.

Also notice that we can *calculate* with value functions and so *prove* facts about synchronous algorithms. For example, using the defining equations for V_N in the first example above, we can calculate the output of N, $V_5(t,\underline{a},x)$, for any $t \geq 3$, $\underline{a} : T \longrightarrow \mathbb{N}$, and $x \in \mathbb{N}^5$, as follows:

$$\begin{aligned} V_5(t,\underline{a},x) &= V_3(t-1,\underline{a},x) + V_4(t-1,\underline{a},x) \\ &= V_1(t-2,\underline{a},x) + V_2(t-2,\underline{a},x) + V_1(t-2,\underline{a},x) \times V_2(t-2,\underline{a},x) \\ &= (\underline{a}(t-3)-1) + (\underline{a}(t-3)+1) + (\underline{a}(t-3)-1) \times (\underline{a}(t-3)+1) \\ &= f(\underline{a}(t-3)) \end{aligned}$$

Clearly, this calculation is the nub of what must be proved to verify N; see the exercise below.

We will formalise our specification technique and develop a theory of synchronous algorithms by mathematically classifying value functions. Note the *form* of the equations that define the coordinates $V_1,\ldots,V_k$ of a value function V_N. Recall the *simultaneous primitive recursive* definition of $f_1,\ldots,f_k$ in

Section 1.1 (under 'Chapter 3'). The defining equations for $f_1,\ldots,f_k$ are very similar to those for $V_1,\ldots,V_k$. With a little work the definition of $V_1,\ldots,V_k$ can be easily seen to be an *instance* of the definition of $f_1,\ldots,f_k$ and so $V_1,\ldots,V_k$ are simultaneous primitive recursive functions made from module specifications. This point is the subject of Section 3.4.

2.4.3 Output Specifications.

In this section we will explain how to define the output specifications F_N and G_N from V_N; it is in terms of G_N (or F_N) that network correctness is phrased of course (see Section 2.2.4).

Let N be an n-source, k-module synchronous network over data set A; then

$$V_N=(V_1,\ldots,V_k):T\times[T\longrightarrow A^n]\times A^k\longrightarrow A^k$$

Notice that the value function V_N tells us the value held by every module in the network, whereas ultimately we are only interested in the values held by those modules that are connected to sinks (since this is where network output appears). We can restrict our attention to just these modules as follows. Let N have m sinks, and for $j=1,\ldots,m$, let i_j be the unique index of the module whose output channel supplies the jth sink of N. Now let $\pi=\pi_N:A^k\longrightarrow A^m$ be defined by

$$\pi(a)=(a_{i_1},\ldots,a_{i_m})$$

for each $a=(a_1,\ldots,a_k)\in A^k$. Intuitively, when applied to $V_N(t,\underline{a},x)$ this π 'picks out' the values that are sent to the sinks at time t, and so it is appropriate to define

$$F_N=(F_1,\ldots,F_m):T\times[T\longrightarrow A^n]\times A^k\longrightarrow A^m$$

by

$$F_N(t,\underline{a},x)=\pi_N(V_N(t,\underline{a},x)) \tag{8}$$

for each $t\in T$, $\underline{a}:T\longrightarrow A^n$ and $x\in A^k$. Notice that for any arguments t, $\underline{a}$, and x, we now have

$$F_N(t,\underline{a},x)=\pi_N(V_N(t,\underline{a},x))=\pi_N(V_1(t,\underline{a},x),\ldots,V_k(t,\underline{a},x))=(V_{i_1}(t,\underline{a},x),\ldots,V_{i_m}(t,\underline{a},x))$$

and so for $j=1,\ldots,m$ we have

$$F_j(t,\underline{a},x)=V_{i_j}(t,\underline{a},x)$$

That is, the jth coordinate of F_N tells us the output at the jth sink as a function of time, input data, and initial values. (Intuitively, this definition says that there is zero propagation delay from m_{i_j} to the jth sink, or, that the value held by m_{i_j} can be accessed or read from outside the network.)

As mentioned in Section 2.2.1, the alternative (stream transformation) version of F_N, namely $G_N:[T\longrightarrow A^n]\times A^k\longrightarrow[T\longrightarrow A^m]$, is readily definable from F_N by defining, for each $\underline{a}:T\longrightarrow A^n$ and $x\in A^k$, $G_N(\underline{a},x)$ to be the stream defined by

$$G_N(\underline{a},x)(t)=F_N(t,\underline{a},x)$$

for each $t\in T$.

Exercise. Let $\Phi_N:[T\longrightarrow\mathbb{N}]\longrightarrow[T\longrightarrow\mathbb{N}]$ be defined by

$$\Phi_N(\underline{n})(t)=\begin{cases}u & \text{if } 0\le t<3\\ f(\underline{n}(t-3)) & \text{if } t\ge 3\end{cases}$$

for each $\underline{n}:T\longrightarrow\mathbb{N}$ and $t\in T$, where $f(n)=n^2+2n-1$ for each $n\in\mathbb{N}$. Prove that the network N of Figure 2.1 meets this specification in the sense of Section 2.2.4.

2.4.4 Static Specifications.

The functional specifications that we have defined so far (viz V_N, F_N, and G_N) are *dynamic* specifications in the sense that they specify the behaviour of a network as the input at the sources changes with the time t. Later we will need network specifications that define the behaviour of a network when the input at the sources is held fixed or *static* throughout the entire execution of the network. Actually, we have already mentioned one such 'static specification', namely the function f_N of Section 2.1: recall that whereas

$$F_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^m$$

with '$F_N(t,\underline{a},x)$' denoting the output of a network N at time t given input *stream* $\underline{a}$ and initial values x, we had

$$f_N : T \times A^n \times A^k \longrightarrow A^m$$

with '$f_N(t,a,x)$' denoting the output of a network N at time t given *constant* input a and initial values x. The function F_N was defined from V_N by composing V_N with the function π_N that picked out output values. In this section we will define f_N by composing v_N, a 'static version' of V_N, with π_N. Whereas

$$V_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^k$$

with '$V_N(t,\underline{a},x)$' denoting the values held in network N at time t given input stream $\underline{a}$ and initial values x, this v_N will have functionality

$$v_N : T \times A^n \times A^k \longrightarrow A^k$$

with '$v_N(t,a,x)$' denoting the values held in a network N at time t given constant input a and initial values x.

Static Value Functions. As usual, let N be a n-source, k-module synchronous network over A. The function v_N, as a vector-valued function, will have coordinate functions $v_1,\ldots,v_k : T \times A^n \times A^k \longrightarrow A$, and for $i = 1,\ldots,k$ we want '$v_i(t,a,x)$' to denote the value held by the ith module at time t given static input a and initial values x. Clearly, the appropriate defining equations are (for $i = 1,\ldots,k$):

$$v_i(0,a,x) = x_i$$

and, if m_i is autonomous with respect to $T = T_N$, then

$$v_i(t+1,a,x) = f_i(b_1,\ldots,b_{n_i})$$

where f_i is the functional specification of m_i, n_i is the number of inputs to m_i from adjacent modules and sources, and for $j = 1,\ldots,n_i$,

$$b_j = \begin{cases} a_p & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ v_q(t,a,x) & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

and, if m_i is nonautonomous with respect to T, then

$$v_i(t+1,\underline{a},x) = f_i(t,b_1,\ldots,b_{n_i})$$

where $b_1,\ldots,b_{n_i}$ are as described above.

We call v_N N's *static value function* or its *static specification.*

Finally, we define f_N as promised earlier. It should be obvious now that the appropriate definition is

$$f_N(t,a,x) = \pi_N(v_N(t,a,x))$$

for each $t \in T$, $a \in A^n$, and $x \in A^k$.

Discussion. What is the relationship between dynamic and static specifications? Since the function V_N defines the behaviour of N on all possible input streams $\underline{a} : T \longrightarrow A^n$, we can define the behaviour of N on a constant input $a \in A^n$ by taking $\underline{a}$ to be a stream such that $\underline{a}(t) = a$ for all $t \in T$.

For each $a \in A^n$, let $fix(a) : T \longrightarrow A^n$ be the stream defined by $fix(a)(t) = a$ for each $t \in T$. Now consider the function $w_N : T \times A^n \times A^k \longrightarrow A^k$ defined by

$$w_N(t,\underline{a},x) = V_N(t,fix(a),x)$$

for each $t \in T$, $a \in A^n$, and $x \in A^k$. It is straightforward to prove (exercise!) that $w_N(t,a,x) = v_N(t,a,x)$ for all arguments t, a, and x, and so it does not seem to matter which way we define a static version of V_N. However, there is a subtle property associated with the function $fix : A^n \longrightarrow [T \longrightarrow A^n]$ that we may wish to avoid in certain circumstances. We will return to this matter in Section 3.4.

2.4.5 Exercises.

Here are some exercises concerning OE and EOE over $D = \mathbb{N}$.

(1) Take $n = 4$ and write out (in the style of Section 2.4.4) the defining equations for the coordinates of v_{OE} and v_{EOE}. Write down the definitions of π_{OE} and π_{EOE}.

(2) Use the definitions from part (1) above to show that

$$f_{OE}(5,a,x) = f_{EOE}(5,a,y) = sort(a)$$

where $a = (71,21,9,3)$, $x = (0,0,0,0)$, and $y = (0,\ldots,0)$ (twenty times zero).

(3) By using a computer programming language that supports user-defined functions or procedures (such as PASCAL), program OE's module specifications $f_1,\ldots,f_n$ (as defined in Section 2.3.2) as user-defined functions. Incorporate these functions in a program which evaluates $v_{OE}(t,a,x)$ on any given arguments $t \in T$, $a,x \in D^n$, and hence *simulate* the OE sorter on a fixed input for a finite amount of time. *Is it possible to write this program without using either arrays or recursion?*

2.5 OBJECTIVES.

In this chapter we have identified general concepts that are central to synchronous computation. In this section we will outline the objectives that a theory of synchronous systems should achieve.

2.5.1 Formalisation.

Our account of specifying synchronous algorithms has been a semantic one: we have assumed that the reader understands the principal concepts underlying our specification technique, namely those of 'data', 'function', and 'definition by means of equations'.

In Chapter 3 we will introduce our system PR which can be thought of as a formal language for defining functions by simultaneous primitive recursion. We will show that the specification algorithm of Section 2.4.1 always specifies a network as a (value) function whose definition is a term in this language. In this way we formalise our specification technique and hence formalise our account of synchronous algorithms.

2.5.2 Verification.

Generally, the most important point about being able to formally specify an algorithm's behaviour is that we can write down and ultimately (attempt to) prove assertions about that behaviour. In particular we can write down formal correctness specifications *CSpec* for the algorithm. For example, in the case of OE and EOE, possible correctness specifications are (in the style of Section 2.2.4)

$$CSpec_{OE} \iff (\forall x \in D^n)(\forall t \in T)(\forall \underline{a} : T \longrightarrow D^n)\ (\ G_{OE}(\underline{a},x)(t) = \Phi_{OE}(\underline{a})(t)\ )$$

and

$$CSpec_{EOE} \iff (\forall x \in D^{n(n+1)})(\forall t \in T)(\forall \underline{a} : T \longrightarrow D^n))\ (\ G_{EOE}(\underline{a},x)(t) = \Phi_{EOE}(\underline{a})(t)\ )$$

In Chapter 4 we will formally verify OE and EOE by proving $CSpec_{OE}$ and $CSpec_{EOE}$, and in Chapter 5 we will establish the correctness of other synchronous algorithms. Another important kind of formal specification of a (synchronous) algorithm is a *performance* specification *PSpec*. In Chapter 3 we consider performance in a general setting, and we will be able to write down performance specifications for OE and EOE.

2.5.3 Simulation.

Generally, having devised an algorithm for a given task, the first step towards verifying the algorithm is to *simulate* the algorithm, either by hand, or by coding the algorithm in an executable form. In the case of synchronous algorithms however, the algorithm need not be large to be complex, and this makes hand-simulation tedious and therefore error-prone, if not impossible in practice. Clearly, simulation by means of a computer program is generally preferable; in other words, a designer of synchronous systems needs *software tools* to assist in the design activity.

In Chapter 6 we will introduce the imperative programming language FPIT which is tailored to the evaluation of the functions defined in Chapter 3; in this way we can simulate our networks.

2.5.4 Hierarchical Design.

In algorithm design generally, the advantages of *structured* design are well-known. Specifically, an algorithm which comprises a small number of atomic steps is easy to understand and thereby more amenable to formal verification. In the case of a large algorithm, the algorithm is decomposed into a small number of procedures which are *regarded* as atomic steps so as to bound the conceptual complexity of the algorithm, and hence its verification *relative to the assumed correctness of the procedures*; if the procedures themselves are large, then they too are decomposed into sub-procedures, and these into sub-sub-procedures, until a level is reached at which the lowest level procedures *are* atomic steps.

This top-down approach to the design and verification of algorithms is one which is germane to synchronous algorithms: contemporary synchronous (VLSI) designs may incorporate many thousands of atomic processing elements (modules); such a design will be incomprehensible unless it has a hierarchical structure. We will consider the hierarchical design of synchronous algorithms in Chapter 8.

2.6 SOURCES.

The isolation of a general concept of a synchronous concurrent algorithm processing streams of data is the result of joint work with J. V. Tucker and myself, as is the representation of synchronous algorithms by means of value functions (see Thompson and Tucker[1985]). The sources that led our concept of a synchronous algorithm were mentioned in Section 1.2: these were the neurone nets of W. S. McCulloch and W. Pitts and the systolic algorithms of H. T. Kung. The attempt at a general theory is new. We note that streams are mentioned in Melhem and Rheinboldt[1984] (work on systolic algorithms), Sheeran[1983] (work on design transformations), and Johnson[1984] (work on synthesis of digital circuits).

Parallel sorting networks have received much attention in the literature. Batcher[1968] is generally acknowledged as being the first work on the subject; Bitton et al[1984] and Akl[1985] are useful surveys. The EOE sorter is due to D. E. Knuth: Section 5.3.4 of Knuth[1973] considers general networks of 'compare-and-exchange' modules, of which EOE is an instance (see Exercise 37, p 241, in Knuth[1973]). The OE sorter is accredited to Knuth also; however, (for historical reasons) we call 'EOE' what Knuth calls 'OE', and we have been unable to find (what we call) OE in Knuth[1973] despite the accreditations of Bitton et al[1984] (p. 291) and Akl[1985] (p. 58). Sorting on streams of data (in the sense of processing more than one vector of input data) has not been studied as far as we are aware.

Finally, our definition of a user specification (Definition 2.2.3) is a special case of the considerably more general kind of specification found in Harman and Tucker[1987].

CHAPTER 3
FORMAL SPECIFICATION

In this chapter we begin the theory of synchronous systems by introducing *PR* which we use to formalise the specification technique for synchronous algorithms introduced in Section 2.4.

In Section 3.1 we establish a formal account of the data set(s) and operations over which a synchronous algorithm is defined. We use the theory of *abstract data types* as developed by the ADJ group (J. A. Goguen, J. W. Thatcher, E. A. Wagner, and J. B. Wright) in their work Goguen, Thatcher, and Wagner[1978] and Meseguer and Goguen[1985]. This theory supports the hierarchical analysis of synchronous algorithms. In particular, we will see how the concept of an *augmentation* of a data type helps us to formalise some aspects of top-down design.

In Section 3.2 we begin the formal theory of algorithm *performance* or *complexity*. We define a *performance measure* which is an abstraction of the time taken to evaluate data and operations involved in a data type. This abstraction is the basis of a hierarchical treatment of algorithm performance.

In Section 3.3 we define the system PR. In essence, PR comprises a formal notation for the *simultaneous primitive recursive functions over an abstract data type*. These functions, first defined and studied in Tucker and Zucker[1987] (work of 1979), are the mathematical setting for the study of value functions. PR additionally involves an account of the complexity of evaluating a function defined by simultaneous primitive recursion that is based on a performance measure for the underlying data type.

In Section 3.4 we formalise the specification of synchronous algorithms by establishing the simultaneous primitive recursiveness of value functions.

In Section 3.5 we begin the theory of PR with an assortment of facts concerning the complexity of computation and top-down design in PR.

3.1 ABSTRACT DATA TYPES.

In Section 2.1 synchronous algorithms were described in the context of two sets, T and A, and a collection of operations on and between these sets. We will now make a modest generalisation by assuming that a synchronous algorithm involves operations on and between a (finite) number of data sets $A_1,\ldots,A_n$, say.

In order for us to provide a formal account of these data sets and operations we will adopt the algebraic approach of the ADJ group. We will first present basic ideas and definitions, and then we will extend these ideas to cater for synchronous computation (in Section 3.1.8). The knowledgeable reader will find much of our notation and terminology standard, and with the exception of Section 3.1.5, the preliminary sections below may be quickly scanned for consistency with Wagner[1981] (for example).

3.1.1 Syntax.

The first step towards formalisation is to introduce *names* for the data sets of interest:

Definition. A *sort set* S is a finite, nonempty set (of *sorts* or *sort symbols*). □

A sort $s \in S$ is merely a piece of syntax which we can use to index or label data sets in a consistent manner. For example, in the case of a general synchronous algorithm, a typical data set will be denoted by A_s indicating that this set contains data of sort or kind 's'. In the case of a specific algorithm, we choose sort names that are mnemonic of the kind of data in question: for example, the OE network of Section 2.3.1 involved a clock T as well as data from the set D; thus the modules of OE can be formally specified in the context of the sort set $S_0 = \{\mathrm{T,D}\}$ say, with A_T and A_D denoting the sets T and D respectively.

Here are some definitions and notations relating to sorts that we will find useful throughout the remainder of the thesis:

Definitions. Let S be a sort set.

(i) We let S^+ denote the set of all finite *words* or *strings* over S; that is, $w \in S^+$ iff for some $n \geq 1$ and some $s_1,\ldots,s_n \in S$, w is the concatenation or juxtaposition of symbols $w = s_1 \cdots s_n$. In this situation w is defined to have *length* n, in symbols: $|w| = n$.

(ii) We define S^* by $S^* = S^+ \cup \{\lambda\}$ where λ is the *empty word* satisfying $w\lambda = \lambda w = w$ for all words $w \in S^+$. Accordingly, we define $|\lambda| = 0$.

(iii) For each $n \geq 1$ we define $S^n = \{w \in S^+ : |w| = n\}$. (Notice that we do not define S^0; if $w \in S^n$, then $|w| = n \neq 0$.)

(iv) When $w \in S^n$ for any $n \geq 1$, we write w_i for the ith sort comprising w for $i = 1,\ldots,n$. That is, if $w = s_1 \cdots s_n$, then $w_i = s_i$ for $i = 1,\ldots,n$; alternatively, for all words $w \in S^+$, $w = w_1 \cdots w_{|w|}$. □

The intention behind a word over a sort set is that each word w names some Cartesian product of data sets. For example $w = \mathrm{TDDD} \in S_0^4$ names the domain of the function $f_1 : T \times D \times D \times D \longrightarrow D$ that specified module m_1 of the OE network (see Section 2.3.6). We use this idea to keep track of the domains and codomains involved in a collection of operations in the following way:

Definition. Let S be a sort set. An *S-sorted signature* Σ is an $S^* \times S$-indexed family

$$\Sigma = \langle \Sigma_{w,s} : w \in S^*, s \in S \rangle$$

of (disjoint) sets (of *operation names* or *operation symbols*). □

A signature formalises the idea of a (strongly typed) collection of functions available for use (in specifying synchronous algorithms); for $w \in S^n$, a symbol $\sigma \in \Sigma_{w,s}$ names some operation which maps each vector $a \in A_{w_1} \times \cdots \times A_{w_n}$ to an element of A_s. Conventionally, when $\sigma \in \Sigma_{\lambda,s}$, we often use the symbol 'c' rather than 'σ' since such a symbol names a constant (a '0-ary' operation).

Notation.

(i) We write '$\sigma \in \Sigma$', to abbreviate '$\sigma \in \Sigma_{w,s}$ for some $w \in S^*$ and $s \in S$'.

(ii) When Σ is a signature with n constants and operations $\sigma_1,\ldots,\sigma_n$, we write $\Sigma=(\sigma_1,\ldots,\sigma_n)$ when it is understood (or of no interest) which $\Sigma_{w,s}$ each σ_i belongs to.

3.1.2 Semantics.

The concept of a 'data type', that is, 'a collection of data sets and operations' can now be formalised as an *algebra*:

Definition. Let Σ be an S-sorted signature. A *Σ-algebra* or *Σ-structure* A comprises an S-indexed family

$$<A_s : s \in S>$$

of sets (call A_s the *carrier* of sort s), together with, for each $w \in S^n$ and each $s \in S$, a mapping

$$\sigma^A : A_{w_1} \times \cdots \times A_{w_n} \longrightarrow A_s$$

for each $\sigma \in \Sigma_{w,s}$. When $\sigma \in \Sigma_{\lambda,s}$, $\sigma^A \in A_s$ (that is, $c^A \in A_s$). We call σ^A the *interpretation of σ in A*. Additionally, if S is a singleton $S=\{s\}$ then we say A is *single-sorted*, otherwise A is *many-sorted*.

3.1.3 Examples.

(1) We can formalise the notion of discrete time as a single-sorted algebra C in the following way. First let $S=\{\mathrm{T}\}$ and let Γ be the $S^*\times S$-family

$$\Gamma = <\Gamma_{w,s} : w \in S^*, s \in S>$$

where

$$\Gamma_{\lambda,\mathrm{T}} = \{\mathrm{zero}\}, \qquad \Gamma_{\mathrm{T},\mathrm{T}} = \{\mathrm{succ}\},$$
$$\Gamma_{w,s} = \varnothing \text{ for other } w,s$$

Then it is easy to check that Γ is a S-sorted signature. Now let C comprise the single carrier $C_{\mathrm{T}}=\{0,1,2,\ldots\}$, the constant $\mathrm{zero}^C=0$, and the operation $\mathrm{succ}^C : \{0,1,2,\ldots\} \longrightarrow \{0,1,2,\ldots\}$ where $\mathrm{succ}^C(t)=t+1$ for each $t \in \{0,1,2,\ldots\}$. Then $\mathrm{zero}^C \in C_T$ and $\mathrm{succ}^C : C_T \longrightarrow C_T$; that is, C is a Γ-algebra.

(2) We can formalise the data sets and operations involved in the OE sorter of Section 2.3 as a many-sorted algebra A_{OE} in the following way:

From Section 2.3.6, OE's n modules were specified by the functions $f_1,\ldots,f_n$, each of which was a function of time T and data D; thus it is appropriate to begin with the sort set $S=\{\mathrm{T,D}\}$. Since T is intentionally a clock, we need a signature Σ with symbols $\mathrm{zero} \in \Sigma_{\lambda,\mathrm{T}}$ and $\mathrm{succ} \in \Sigma_{\mathrm{T},\mathrm{T}}$. Furthermore, the functions $f_1,\ldots,f_n$ had the following functionalities: $f_1 f_n : T \times D^3 \longrightarrow D$, and $f_2,\ldots,f_{n-1} : T \times D^4 \longrightarrow D$; thus if Σ has a symbol σ_i to name f_i for $i=1,\ldots,n$, then we must have $\sigma_1,\sigma_n \in \Sigma_{\mathrm{TDD,D}}$ and $\sigma_2,\ldots,\sigma_{n-1} \in \Sigma_{\mathrm{TDDD,D}}$. Now let $A=A_{OE}$ comprise the carriers $A_{\mathrm{T}}=\{0,1,2,\ldots\}$ and $A_{\mathrm{D}}=D$, and interpret the symbols of Σ as follows: $\mathrm{zero}^A=0$; $succ^A(t)=t+1$ for each $t \in A_{\mathrm{T}}$, and $\sigma_i^A=f_i$ for $i=1,\ldots,n$. Then A is a Σ-algebra. □

Here are some simple definitions and notations which relate to Σ-algebras A, and to S-indexed sets in general.

3.1.4 Definitions.

(i) For each $w \in S^*$ we define $A^w = A_{w_1} \times \cdots \times A_{w_n}$.

(ii) As a short-hand notation for a structure A with family of carriers $<A_s : s \in S>$ and n constants and operations, we write $A = (<A_s : s \in S>; \sigma_1^A, \ldots, \sigma_n^A)$. Indeed, we will further abbreviate by letting A denote the algebra's family of carriers; we will write $A = (A; \sigma_1^A, \ldots, \sigma_n^A)$.

(iii) Let A and B be S-indexed families of sets A_s and B_s respectively. An *S-indexed* mapping $f : A \longrightarrow B$ is a family

$$f = <f_s : s \in S>$$

of mappings $f_s : A_s \longrightarrow B_s$. If $f : A \longrightarrow B$ is an S-indexed mapping, then for each $w \in S^*$, we define $f_w : A^w \longrightarrow B^w$ by

$$(\forall a = (a_1, \ldots, a_n) \in A^w)\ (f_w(a) = (f_{w_1}(a_1), \ldots, f_{w_n}(a_n)))$$

(iv) If f is a function of the form $f : A^u \longrightarrow A^v$, we will sometimes write 'f_A' rather than just 'f'; this use of an algebra as a subscript is purely to emphasise that f is a function on the carriers of A.

(v) Let A be a Σ-algebra for S-sorted signature Σ, and let Ω be another S-sorted signature with $\Omega \subseteq \Sigma$ (that is, let $\Omega_{w,s} \subseteq \Sigma_{w,s}$ for each $w \in S^*$ and $s \in S$). Now let B be the Ω-algebra whose family of carriers $<B_s : s \in S>$ is defined by $B_s = A_s$ for each $s \in S$, and whose operations σ^B are defined by $\sigma^B = \sigma^A$ for each $\sigma \in \Omega$. Then we say B is the *restriction of A to Ω*, in symbols: $B = A|_\Omega$. (Intuitively, B is formed from A by 'forgetting about' some of A's operations.)

(vi) When $f : A^u \longrightarrow A^v$ for some $u, v \in S^+$ we say f has *functionality* or *arity* (u, v).

3.1.5 Augmentation.

We have seen how a Σ-algebra formalises the idea of a 'data type'. In this section we will define a construction which allows us to build new data types from old. This construction, which we call an *augmentation*, is the basis of a formal transformation between different levels of data abstraction, and thus, here we begin to explain the hierarchical aspects of the theory of data types.

Definitions. Let A be a Σ-algebra for S-sorted signature Σ.

(i) Let ϕ be some new symbol not occurring in Σ, and let $w_0 \in S^+$ and $s_0 \in S$. Then the *(w_0, s_0)-extension of Σ obtained by adding ϕ*, is the S-sorted signature Ω defined by

$$\Omega = <\Omega_{w,s} : w \in S^*, s \in S>$$

where for each $w \in S^*$ and $s \in S$,

$$\Omega_{w,s} = \begin{cases} \Sigma_{w,s} \cup \{\phi\} & \text{if } w = w_0 \text{ and } s = s_0 \\ \Sigma_{w,s} & \text{otherwise} \end{cases}$$

In symbols we write $\Omega = (\Sigma, \phi; w_0, s_0)$ or $\Omega = (\Sigma, \phi)$ if w_0 and s_0 are understood.

(ii) Let $\Omega = (\Sigma, \phi; w_0, s_0)$ be as above, and let $f_A : A^{w_0} \longrightarrow A_{s_0}$. Then the *augmentation of A obtained by adding f* is the Ω-algebra B whose family of carriers $<B_s : s \in S>$ is defined by $B_s = A_s$ for

each $s \in S$, and whose operations σ^B are defined by

$$\sigma^B = \begin{cases} f_A & \text{if } \sigma = \phi \\ \sigma^A & \text{if } \sigma \neq \phi \end{cases}$$

for each $\sigma \in \Omega$. In symbols we write $B = (A, f_A)$. □

The natural generalisation of the above definition(s) is to augment an algebra with a collection of functions $\{f_1,...,f_m\}$. We only need a special case of such a construction however: our ultimate intention is to augment Σ-algebras with vector-valued functions $f_A : A^u \longrightarrow A^v$. As we cannot do this directly (since the operations of an algebra must be single-valued), we add the coordinate functions of f_A instead:

Definition. Let A be a Σ-algebra for S-sorted signature Σ.

(i) Let $\Phi = (\phi_1,...,\phi_m)$ be a vector (ordered list) of new function identifiers $\phi_1,...,\phi_m$ not occurring in Σ. Also let $u, v \in S^+$ with $|v| = m$. Then the (u,v)-*extension of* Σ *obtained by adding* Φ is the S-sorted signature Ω defined by $\Omega = \Sigma^{(m)}$ where $\Sigma^{(0)} = \Sigma$, and for $k = 0,...,m-1$,

$$\Sigma^{(k+1)} = (\Sigma^{(k)}, \phi_{k+1}; u, v_{k+1})$$

In symbols we write $\Omega = (\Sigma, \Phi; u, v)$ or $\Omega = (\Sigma, \Phi)$ if u and v are understood.

(ii) Let $\Omega = (\Sigma, \Phi; u, v)$ be as above, and let $f : A^u \longrightarrow A^v$ be any function with coordinates $f_1,...,f_m$ (so $|v| = m$ and $f_i : A^u \longrightarrow A_{v_i}$ for $i = 1,...,m$). Then the *augmentation of* A *obtained by adding* f is the Ω-algebra B defined by $B = A^{(m)}$ where for $k = 1,...,m$ $A^{(k)}$ is the $\Sigma^{(k)}$-algebra defined by $A_s^{(0)} = A$ and for $k = 0,...,m-1$,

$$A^{(k+1)} = (A^{(k)}, f_{k+1})$$

In symbols we write $B = (A, f)$.

Discussion. Imagine a programmer whose task T, is to resolve by means of a program P, some problem involving computations over Σ-algebra A. Of course, P will be based on, or written *over* the signature Σ; typically P will involve expressions or terms t of the form $\sigma(t_1,...,t_n)$ where $\sigma \in \Sigma$ and $t_1,...,t_n$ are terms built up from Σ (together with some variables perhaps).

Suppose the task T is a complex one, complex enough that the programmer first assumes that some subtask T_f, which is to compute a function f_A defined over A, has already been accomplished by means of a subprogram or procedure P_f. Typically P_f will introduce some new identifier ϕ say, which the programmer can now use in expressions $\phi(t_1,...,t_n)$ in constructing the required program P.

Importantly, the programmer will now regard f_A as having status equal to that of any other operation σ^A, since like σ^A, f_A is a named operation which is conceptually indivisible.

Our hypothetical programmer is of course developing P 'top-down'. Initially T is regarded as a problem involving computations over the high-level structure $B = (A, f_A)$, and the programmer first solves T by means of a program P' say, which is written over the signature $\Omega = (\Sigma, \phi)$ (and thus $\phi^B = f_A$). In order to show that T can be solved by using the operations of (lower-level) A only, the programmer *implements* f_A over A; that is, P_f is written over the symbols of Σ only.

We begin to see the how the hierarchical design and specification of algorithms is intimately related to the hierarchical specification of data types. This relationship will be further explored in the second part of Section 3.5, and in Section 6.2.18.

3.1.6 Minimality.

Consider the $\{\mathrm{T}\}$-sorted signature $\Gamma=(\mathrm{zero},\mathrm{succ})$. Intuitively, Γ is the signature of a clock as an algebra $A=(A_{\mathrm{T}};\mathrm{zero}^A,\mathrm{succ}^A)$. Now suppose $\mathrm{zero}^A=0$, and $\mathrm{succ}^A(t)=t+1$ for each $t\in A_{\mathrm{T}}$, but $A_{\mathrm{T}}=\{-1,0,1,2,\ldots\}$. Whilst A is certainly a Γ-algebra, the inclusion of $-1\in A_{\mathrm{T}}$ is redundant in the sense that no expression over Γ will ever evaluate to -1.

Generally, we can isolate algebras with such values by means of the following

Definitions. Let A be a Σ-algebra.

(i) Let $a\in A$ (that is, let $a\in A_s$ for some carrier A_s). We say a is *finitely generated* if a can be obtained from finitely many applications of the operations of A to the constants of A.

(ii) We say A is *minimal* if every element of every carrier of A is finitely generated.

Example. Let A be the Γ-algebra defined above. It is obvious that A is not minimal, since $\mathrm{zero}^A=0\neq-1$ and $\mathrm{succ}^A(t)\neq-1$ for any $t\in\{-1,0,1,2,\cdots\}$, and thus -1 is not a finitely generated element. □

3.1.7 Abstract Data Types.

It is possible for A and B to both be Σ-algebras, and yet to have $A\neq B$. Under what conditions is a Σ-algebra *uniquely* defined?

Consider the Γ-algebras $A=(\{0,1,\ldots\};0,\mathrm{succ}^A)$ and $B=(\{-1,0,1,\ldots\};-1,\mathrm{succ}^B)$ where succ^A and succ^B are successor functions on the carriers of A and B respectively. Notice that both A and B *are* Γ-algebras (minimal ones in fact), but $A\neq B$ since $\mathrm{zero}^A=0\neq-1=\mathrm{zero}^B$.

Both A and B are 'counting structures': each comprises a 'zero' and a 'next number' function. *Abstractly*, we do not wish to distinguish between A and B: A and B are the same in the sense that we can count with either algebra; the only difference between A and B is that in A 'zero' is represented by 0, whereas in B it is represented by -1.

We can formalise this sense in which Σ-algebras are 'essentially the same' as follows.

Definitions. Let A and B be S-sorted Σ-algebras.

(i) An S-indexed mapping $h:A\longrightarrow B$ is said to be a Σ-*homomorphism* if h satisfies the following two *homomorphism conditions*:

(a) For each $c\in\Sigma_{\lambda,s}$,

$$h(c^A)=c^B$$

(b) For each $\sigma\in\Sigma_{w,s}$ with $w\in S^n$,

$$(\forall(a_1,\ldots,a_n)\in A^w)\ (\ h_s(\sigma^A(a_1,\ldots,a_n))=\sigma^B(h_{w_1}(a_1),\ldots,h_{w_n}(a_n))\ )$$

(ii) A Σ-homomorphism h is said to be a Σ-*isomorphism* if each component function h_s is a bijection.

(iii) We define A to be *Σ-isomorphic to B* if there is a Σ-isomorphism $h : A \longrightarrow B$. In symbols we write $A \cong B$ if there is some Σ-isomorphism $h : A \longrightarrow B$, and if we wish to name the Σ-isomorphism explicitly then we write $A \cong_h B$ (read: A is isomorphic to B *via* Σ-isomorphism h).

Lemma. *Let A and B be Σ-algebras. The following statements are equivalent:*

(i) $A \cong_h B$

(ii) *There exist Σ-homomorphisms $h_1 : A \longrightarrow B$ and $h_2 : B \longrightarrow A$.*

Proof. Omitted. □

Exercise. Let A and B be the Γ-algebras above. Define $h : A \longrightarrow B$ by

$$(\forall t \in A_\tau)\ (\ h(t) = t-1\ )$$

Show that h is a Γ-isomorphism, and hence that $A \cong_h B$. □

A Σ-algebra is regarded as uniquely specified in the sense that it is unique *up to isomorphism*. Now notice that the relationship of isomorphism is an *equivalence relation* on the class of all Σ-algebras: it is not difficult to prove for any Σ-algebras A, B, and C, that $A \cong A$, if $A \cong B$ then $B \cong A$, and if $A \cong B$ and $B \cong C$, then $A \cong C$.

Definitions. Let A be a Σ-algebra.

(i) The *isomorphism class* ISO(A) of A is the equivalence class of all Σ-algebras isomorphic to A. That is,

$$\mathrm{ISO}(A) = \langle B : B \cong A \rangle$$

(ii) An *abstract data type* is an isomorphism class of Σ-algebras, for some signature Σ, called the signature of the abstract data type.

Definition. Let P be the single-sorted algebra comprising carrier { 0,1,2,... }, constant 0, and operation t+1. A *clock* is an element of ISO(P). (Thus all clocks are isomorphic.)

3.1.8 Algebras for Synchronous Computation.

Since our methodology for specifying a synchronous algorithm is based on specifying values held at a given time t, a formal specification of an algorithm must be based on a Σ-algebra which has a clock as a substructure. In fact, since (synchronous) algorithms usually involve *tests* of some kind, we will assume for convenience that our Σ-algebras always include a Boolean component as well as a clock; such algebras we call *standard*:

Definitions.

(i) A sort set S is *standard* if $S \supseteq \{ \mathrm{T}, \mathrm{B} \}$ for the distinguished sort symbols T and B.

(ii) An S-sorted signature Σ is said to be *standard* if S is standard, and the following six conditions hold:

(a) $\mathrm{zero} \in \Sigma_{\lambda,\mathrm{T}}$

(b) $\mathrm{succ} \in \Sigma_{\mathrm{T},\mathrm{T}}$

(c) $\text{true},\text{false} \in \Sigma_{\lambda,\mathbb{B}}$

(d) $\text{not} \in \Sigma_{\mathbb{B},\mathbb{B}}$

(e) $\text{or},\text{and} \in \Sigma_{\mathbb{BB},\mathbb{B}}$

(f) for each $s \in S$, $=_s \in \Sigma_{ss,\mathbb{B}}$

(iii) A Σ-structure A is said to be *standard* if Σ is standard, and the following three conditions hold:

(a) $A_{\mathbb{T}}$ together with its constant and basic operation is a clock.

(b) $A_{\mathbb{B}}$ together with its basic operations is (isomorphic to) the Boolean algebra $\mathbb{B}=(\{tt,ff\};tt,ff,\sim,\vee,\wedge)$. (Here tt and ff stand for 'true' and 'false' respectively, and $\sim$, $\vee$, and $\wedge$, are logical negation, disjunction, and conjunction respectively.)

(c) For each $s \in S$, the operation in A named by $=_s$ is equality on A_s.

(iv) Given a standard algebra A, we call $A_{\mathbb{T}}=\{0,1,2,\ldots\}$ and $A_{\mathbb{B}}=\{tt,ff\}$ the *standard domains*; $0 \in T$ and $tt,ff \in \mathbb{B}$ the *standard constants*; and succ^A, $\sim$, $\vee$, and $\wedge$, the *standard operations.*

Notation. We write T for $A_{\mathbb{T}}$, $\mathbb{B}$ for $A_{\mathbb{B}}$, and $=$ for $=_s^A$.

Stream Algebras. Since synchronous algorithms process streams of data, we must extend our algebraic formalisation of data types to encompass streams. The way that we will do this is essentially as follows:

Given a standard algebra A, we first adjoin $[T \longrightarrow A_s]$ as a new carrier set for each $s \in S-\{\mathbb{T},\mathbb{B}\}$ (we do not need streams over T or $\mathbb{B}$). Secondly, in order to access elements of a stream we also adjoin as new basic operations, *evaluation functions* of the form $eval_A^s : T \times [T \longrightarrow A_s] \longrightarrow A_s$ where for each $s \in S$, $eval_A^s(t,\underline{a})=\underline{a}(t)$ for each $t \in T$ and $\underline{a} : T \longrightarrow A_s$.

These additions determine a new algebra $\underline{A}$ which appropriately formalises the data and operations involved in a synchronous algorithm over A. However, to formally analyse computation over $\underline{A}$ we must introduce formal names for the new carriers and operations. We do this as follows:

Definitions. Let S be a standard sort set, let Σ be a standard signature, let A be a standard Σ-algebra, and let $\mathbf{S}=\{\underline{s} : s \in S-\{\mathbb{T},\mathbb{B}\}\}$.

(i) We define $\underline{S}$ by $\underline{S}=S \cup \mathbf{S}$. As additional notation, for each $w \in S^n$ we define $\underline{w} \in \underline{S}^n$ by $\underline{w}=\underline{w}_1 \cdots \underline{w}_n$.

(ii) We define $\underline{\Sigma}$ to be the $\underline{S}$-sorted signature defined by

$$\underline{\Sigma} = <\underline{\Sigma}_{w,s} : w \in \underline{S}^*, s \in \underline{S}>$$

where for each $w \in S^*$ and $s \in S$,

$$\underline{\Sigma}_{w,s} = \Sigma_{w,s}$$

and for each $s \in S$,

$$\underline{\Sigma}_{\mathbb{T}\underline{s},s} = \{\text{eval}_s\}$$

and for other $w \in \underline{S}^*$ and $s \in \underline{S}$,

$$\underline{\Sigma}_{w,s} = \emptyset$$

(iii) We define the *stream algebra* $\underline{A}$ to be the $\underline{\Sigma}$-algebra whose family of carriers $<\underline{A}_s : s \in \underline{S}>$ is defined by

$$\underline{A}_s = \begin{cases} A_s & \text{if } s \in S \\ [T \longrightarrow A_s] & \text{otherwise} \end{cases}$$

for each $s \in \underline{S}$, and whose operations $\sigma^{\underline{A}}$ are defined by

$$\sigma^{\underline{A}} = \begin{cases} \sigma^A & \text{if } \sigma \in \Sigma \\ eval^s_A & \text{if } \sigma = \text{eval}_s \end{cases}$$

for each $\sigma \in \underline{\Sigma}$.

Example. Let $A = A_{OE}$ be as defined in Example 3.1.3(2); then $\underline{A} = \underline{A_{OE}}$ is constructed as follows:

We begin with the sort set $S = \{\text{T,D}\}$ underlying A. (Actually, this S and A are not officially standard since they do not have a Boolean component, but this need not concern us here: we only need A to have a clock in order for us to have streams.) From the preceding definition we have $\underline{S} = \{\text{T,D,}\underline{\text{D}}\}$. Also, if the signature of A is Σ, then $\underline{\Sigma}$ is defined by $\underline{\Sigma} = (\Sigma, \text{eval}_{\text{D}}; \text{T}\underline{\text{D}}, \text{D})$. Finally, $\underline{A}$ is the $\underline{\Sigma}$-algebra comprising the carriers of A together with the new carrier $\underline{A}_{\underline{\text{D}}} = [T \longrightarrow D]$, and whose operations are the operations of A with the exception of $\text{eval}_{\text{D}}^{\underline{A}}$ which is is the function $eval^{\text{D}}_A : T \times [T \longrightarrow D] \longrightarrow D$.

3.1.9 Notation.

Let $T = \{0,1,2,\ldots\}$ and let X_1 and X_2 be any sets. Now let S_1 and S_2 be defined by

$$S_1 = [T \longrightarrow X_1] \times [T \longrightarrow X_2]$$

and

$$S_2 = [T \longrightarrow X_1 \times X_2]$$

respectively. The sets S_1 and S_2 are isomorphic (in the sense that one can define a bijection between them) and so these sets may be conceptually identified. However, there is a subtle distinction between S_1 and S_2 which will be lost when we identify them with each other: an element of S_1 is a pair comprising two single-valued streams, whereas an element of S_2 is a single pair-valued stream; that is, a stream which supplies a pair of values at each time $t \in T$. Since the difference between a Cartesian product of sets of streams and the corresponding set of streams over Cartesian products is mathematically slight, we will choose to use the latter as it is notationally simpler. For example, to represent the set of all streams of pairs inputs we use S_2 above, although we may speak of S_2 as 'a collection of pairs of streams'.

More generally, if A is a S-sorted algebra and $w \in S^n$ then we identify the sets

$$[T \longrightarrow A_{w_1}] \times \cdots \times [T \longrightarrow A_{w_n}]$$

and

$$[T \longrightarrow A_{w_1} \times \cdots \times A_{w_n}]$$

In other words we make the following identification:

$$[T \longrightarrow A_{w_1}] \times \cdots \times [T \longrightarrow A_{w_n}] = [T \longrightarrow A^w]$$

Notice that further notational simplification is possible: recalling the definitions associated with $\underline{A}$, we have

$$\underline{A}^{\underline{w}} = \underline{A}_{\underline{w}_1} \times \cdots \times \underline{A}_{\underline{w}_n} = [T \longrightarrow A_{w_1}] \times \cdots \times [T \longrightarrow A_{w_n}] = [T \longrightarrow A^w]$$

Generally we will use the notation $[T \longrightarrow A^w]$ rather than $\underline{A}^{\underline{w}}$ since the former is easier to comprehend: the latter notation appears in formal calculations only.

3.2 PERFORMANCE MEASURES.

The formal theory of the complexity of a synchronous algorithm begins with an account of the complexity of the data and operations of the underlying algebra. Our analysis of algorithm complexity only extends to algorithm performance or execution time, and does not attempt to account for other aspects of algorithm complexity such as area or power consumption. (See Thompson[1980] for a model of circuit computation which encompasses these.) However, in the forthcoming definitions of performance estimations, the reader is invited to consider replacing references to a time metric by an area metric (say).

Definition. Let A be an S-sorted Σ-algebra. Also let C be a clock, and let $C^+ = C - \{0\}$. A *performance measure* for A is a family

$$P = \langle \sigma^P : \sigma \in \Sigma \rangle$$

of mappings σ^P such that $\sigma^P : A^w \longrightarrow C^+$ when $\sigma \in \Sigma_{w,s}$ for some $s \in S$, and $\sigma^P \in C^+$ for $\sigma \in \Sigma_{\lambda,s}$. Here we say P is *based on* the clock C. Additionally, we say P is *standard* if Σ is standard and $succ^P(t) = 1$ for every $t \in T$. □

In this way we count the cost of evaluating an operation on an argument: if $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and $s \in S$, then for each $a \in A^w$, the time taken with respect to C to evaluate σ^A on an argument $a \in A^w$ is the number $\sigma^P(a)$. If $c \in \Sigma_{\lambda,s}$ for some $s \in S$, there is no data for the evaluation of c^A to depend upon other than c^A itself: for this reason c^P is always a (nonzero) constant. Notice that a performance measure is very general: we do not impose any structure on the performance functions σ^P other than that they have an appropriate domain (and codomain: notice that we do not consider any constant or operation to have zero execution time).

3.2.1 A-time.

A-time is an important example of a specific performance measure for an algebra A: A-time is the formalisation as a performance measure P, of the *uniform cost criterion* of Asveld and Tucker[1982]. According to A-time, every operation and constant of an Σ-algebra A has unit computation time; that is, for each $\sigma \in \Sigma_{w,s}$, with $w \neq \lambda$, $\sigma^P(a) = 1$ for each $a \in A^w$, and for each $c \in \Sigma_{\lambda,s}$, $c^P = 1$.

A-time is an appropriate performance abstraction for A when we consider A as the starting point for an algorithm over A; in this situation, the operations of A are intuitively indivisible, and it is from these 'atoms' that we construct more complex operations. It is natural then, to consider the operations of A as determining an indivisible time metric with respect to which we can measure the length of computation of larger designs.

However, when the complexity of an algorithm over A is considered in a situation where it is either known or intended that the operations of A are to be implemented over another (lower-level) algebra B, say, then the computation time of such an operation will be determined by the complexity of its implementation over B, and thus unit costs are no longer appropriate; it is the generality of a performance measure that allows us to express and analyse algorithm performance from these different perspectives.

Example. Let P_1 and P_2 be performance measures for some algebra A which involves addition (+) on the natural numbers. Now contrast the following two performance estimations:

$$(\forall n,m \in \mathbb{N})\ (+^{P_1}(n,m) = 1)$$
$$(\forall n,m \in \mathbb{N})\ (+^{P_2}(n,m) = kn+c)$$

for some constants $k,c \geq 1$. Clearly, P_1 measures A-time, whereas P_2 counts the cost of addition as if, perhaps, it were known or intended that to add n to m, we increment m by one n times.

3.2.2 Extensions of Performance Measures.

The idea behind the way a performance measure P was defined for an Σ-algebra A was that each named operation σ^A had its own performance estimation σ^P. Consequently, when we augment A with a new operation we need to extend P with a new estimation function.

Definition. Let P be a performance measure for Σ-algebra A. Also let $f : A^w \longrightarrow A_s$ for some $w \in S^+$ and $s \in S$, and let $\lambda : A^w \longrightarrow C^+$ where C is the clock on which P is based. Now let (A,f) be the (Σ,ϕ)-algebra defined in Section 3.1.5. We define the *extension of P obtained by adding λ*, to be the performance measure Q where

$$\sigma^Q = \begin{cases} \lambda & \text{if } \sigma = \phi \\ \sigma^P & \text{otherwise} \end{cases}$$

for each $\sigma \in (\Sigma,\phi)$. In symbols we write $Q = (P,\lambda)$. Note that Q is a performance measure for (A,f) which is based on clock C. □

We also need to define a performance measure for (A,f) when f is a vector-valued function. We do this as follows:

Definition. Let P be a performance measure for Σ-algebra A. Also let $f : A^u \longrightarrow A^v$ have coordinates $f_1,\ldots,f_m$ (so $m = |v|$), and let $\lambda : A^u \longrightarrow C^+$ where C is the clock on which P is based. Now let (A,f) be a (Σ,Φ)-algebra as defined in Section 3.1.5. We define the *extension of P obtained by adding λ*, to be the performance measure Q where

$$\sigma^Q = \begin{cases} \lambda & \text{if } \sigma = \phi_i \in \Phi \\ \sigma^P & \text{otherwise} \end{cases}$$

for each $\sigma \in (\Sigma,\Phi)$. In symbols we write $Q = (P,\lambda)$. Note that Q is a performance measure for (A,f) which is based on clock C. □

Notice that according to the last definition, we charge the same cost (λ) for all the coordinates of f. More generally we might expect f_i to have its own performance estimation λ_i for $i = 1,\ldots,m$. However, whilst it is easy to frame such a definition, the given definition is adequate for our purposes.

3.2.3 Order Notation.

Given an S-sorted algebra A, we will often want to compare the performance of two (rivaling) algorithms which solve the same task: if α_1 and α_2 are two algorithms with common input domain A^u for some $u \in S^+$, then the time complexities of α_1 and α_2 will be expressed as functions $\lambda_1, \lambda_2 : A^u \longrightarrow C^+$ where C is a clock and $\lambda_i(a)$ is the time taken to execute α_i on input a for $i = 1,2$.

The simplest case is when $\lambda_1 = \lambda_2$, that is when $\lambda_1(a) = \lambda_2(a)$ for each $a \in A^u$, expressing the fact that α_1 and α_2 have identical performance. However, more generally we wish to express the fact that whilst two time complexities may not be equal, they might be equivalent in the sense that they are the same to within a constant factor:

Definition. Let X be any set, let $C = \{0,1,2,\ldots\}$, and let $\lambda_1, \lambda_2 : X \longrightarrow C$ be any functions. Then we say λ_1 is *order* λ_2 if there exists a constant $c \geq 1$ such that for every $x \in X$,

$$\lambda_1(x) \leq c \,.\, \lambda_2(x)$$

In symbols we write $\lambda_1 = O(\lambda_2)$.

Additionally, if $\lambda_1 = O(\lambda_2)$ and $\lambda_2 = O(\lambda_1)$ then we write $\lambda_1 \simeq \lambda_2$ and we say λ_1 and λ_2 are of *equivalent order* (or simply: *equivalent*).

3.3 THE SPECIFICATION SYSTEM PR.

We can now define the system PR. In Section 3.3.1 we will define a collection of syntactic terms built up from Σ which we call *PR schema*. In Section 3.3.2 we define a semantic evaluation mapping (or 'meaning' function) which interprets each scheme as a function on A, and in Section 3.3.3 we conclude the definition of PR by defining a length of computation function for PR schema. In the next section we will show how synchronous computation can be formalised using PR.

Note: throughout the remainder of this chapter, sort sets, signatures and Σ-algebras are always assumed to be standard.

3.3.1 Syntax.

Let Σ be an S-sorted signature. We define $PR(\Sigma)$ to be the $S^+ \times S^+$-indexed family

$$PR(\Sigma) = \langle PR(\Sigma)_{u,v} : u, v \in S^+ \rangle$$

of sets $PR(\Sigma)_{u,v}$ of *PR schema*. Each set $PR(\Sigma)_{u,v}$ is defined uniformly in u and v by induction as follows:

Basis Schema.

(i) *Constant Functions.* Let $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$. Then $\alpha \in PR(\Sigma)_{w,s}$.

(ii) *Algebraic Operations.* Let $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$. Then $\alpha \in PR(\Sigma)_{w,s}$.

(iii) *Projection Functions.* Let $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \leq i \leq |w|$. Then $\alpha \in PR(\Sigma)_{w,w_i}$.

Induction: Function Building Tools.

(iv) *Definition-by-Cases.* Suppose $\alpha = DC(\beta,\alpha_1,\alpha_2)$ where for some $u,v \in S^+$ $\beta \in PR(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$. Then $\alpha \in PR(\Sigma)_{u,v}$.

(v) *Vectorisation.* Suppose for some $m \geq 1$, $\alpha = <\alpha_1,...,\alpha_m>$ where for some $u \in S^+$ and $v \in S^m$, $\alpha_i \in PR(\Sigma)_{u,v_i}$ for $i = 1,...,m$. Then $\alpha \in PR(\Sigma)_{u,v}$. (Here $\alpha_1,...,\alpha_m$ are called the *coordinates* of α.)

(vi) *Composition.* Suppose $\alpha = \alpha_2 \circ \alpha_1$ where for some $u,v,w \in S^+$, $\alpha_1 \in PR(\Sigma)_{u,w}$ and $\alpha_2 \in PR(\Sigma)_{w,v}$. Then $\alpha \in PR(\Sigma)_{u,v}$.

(vii) *Primitive Recursion.* Suppose $\alpha = *(\alpha_1,\alpha_2)$ where for some $u,v \in S^+$, $\alpha_1 \in PR(\Sigma)_{u,v}$ and $\alpha_2 \in PR(\Sigma)_{\tau u\, v,v}$. Then $\alpha \in PR(\Sigma)_{\tau u,v}$.

Notation.

(i) We write '$\alpha \in PR(\Sigma)$' to abbreviate '$\alpha \in PR(\Sigma)_{u,v}$ for some $u,v \in S^+$'.

(ii) When $\alpha \in PR(\Sigma)_{u,v}$ for some $u,v \in S^+$, α is intended to denote or define some function on A with domain A^u and codomain A^v; for this reason we refer to the pair (u,v) as the *functionality* or *arity* of α. (Sometimes we will refer to u and v as the *domain* and *codomain* of α meaning that u and v are the 'names' of the domain and codomain of the function defined by α). □

Sometimes it will be useful to consider PR schema which do not involve instances of '$*(,)$':

Definition. A scheme $\alpha \in PR(\Sigma)$ is said to be a *polynomial scheme* if α does not involve any subscheme of the form $*(\alpha_1,\alpha_2)$. More precisely, we define the $S^+ \times S^+$-indexed family

$$POL(\Sigma) = <POL(\Sigma)_{u,v} : u,v \in S^+>$$

of all polynomial schema where for each $u,v \in S^+$, $POL(\Sigma)_{u,v} \subseteq PR(\Sigma)_{u,v}$ comprises exactly those schema definable by clauses (i)-(vi) of the previous definition.

3.3.2 Semantics.

Let A be an S-sorted Σ-algebra. For each $\alpha \in PR(\Sigma)$, the *meaning of* α *over* A is denoted by $[\![\alpha]\!]_A$ where $[\![.]\!]_A$ is the $S^+ \times S^+$-indexed family

$$[\![.]\!]_A = <[\![.]\!]_A^{u,v} : u,v \in S^+>$$

of mappings $[\![.]\!]_A^{u,v} : PR(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow A^v]$. Each mapping $[\![.]\!]_A^{u,v}$ (ambiguously denoted $[\![.]\!]_A$) is defined uniformly in u and v by the induction on the structure of a scheme $\alpha \in PR(\Sigma)_{u,v}$ as follows:

Basis Schema.

(i) *Constant Functions.* If $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$, then

$$[\![\alpha]\!]_A : A^w \longrightarrow A_s$$

is defined by

$$(\forall a \in A^w)\ ([\![\alpha]\!]_A(a) = c^A)$$

(ii) *Algebraic Operations.* If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$, then

$$[\![\alpha]\!]_A : A^w \longrightarrow A,$$

is defined by

$$(\forall a \in A^w)\ \ ([\![\alpha]\!]_A(a) = \sigma^A(a))$$

(Notice that since $\alpha \in \mathrm{PR}(\Sigma)$, $w \neq \lambda$.)

(iii) *Projection Functions.* If $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \leq i \leq n = |w|$, then

$$[\![\alpha]\!]_A : A^w \longrightarrow A_{w_i}$$

is defined by

$$(\forall a = (a_1,\ldots,a_n) \in A^w)\ \ ([\![\alpha]\!]_A(a) = a_i)$$

Induction: Function Building Tools.

(iv) *Definition-by-Cases.* Suppose $\alpha = \mathrm{DC}(\beta,\alpha_1,\alpha_2)$ where for some $u,v \in S^+$ $\beta \in \mathrm{PR}(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in \mathrm{PR}(\Sigma)_{u,v}$. Then

$$[\![\alpha]\!]_A : A^u \longrightarrow A^v$$

is defined by

$$(\forall a \in A^u)\ \ [\![\alpha]\!]_A(a) = \begin{cases} [\![\alpha_1]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ [\![\alpha_2]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

(v) *Vectorisation.* Suppose for some $m \geq 1$, $\alpha = <\alpha_1,\ldots,\alpha_m>$ where for some $u \in S^+$ and $v \in S^m$, $\alpha_i \in \mathrm{PR}(\Sigma)_{u,v_i}$ for $i = 1,\ldots,m$. Then

$$[\![\alpha]\!]_A : A^u \longrightarrow A^v$$

is defined by

$$(\forall a \in A^u)\ \ ([\![\alpha]\!]_A(a) = ([\![\alpha_1]\!]_A(a),\ldots,[\![\alpha_m]\!]_A(a)))$$

(vi) *Composition.* Suppose $\alpha = \alpha_2 \circ \alpha_1$ where for some $u,v,w \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$. Then

$$[\![\alpha]\!]_A : A^u \longrightarrow A^v$$

is defined by

$$(\forall a \in A^u)\ \ ([\![\alpha]\!]_A(a) = [\![\alpha_2]\!]_A([\![\alpha_1]\!]_A(a)))$$

(vii) *Primitive Recursion.* Suppose $\alpha = *(\alpha_1,\alpha_2)$ where for some $u,v \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\tau u\, v,v}$. Then

$$[\![\alpha]\!]_A : T \times A^u \longrightarrow A^v$$

is defined by

$$(\forall a \in A^u)\ \ ([\![\alpha]\!]_A(0,a) = [\![\alpha_1]\!]_A(a))$$

and

$$(\forall t \in T)(\forall a \in A^u)\ \ ([\![\alpha]\!]_A(t+1,a) = [\![\alpha_2]\!]_A(t,a,[\![\alpha]\!]_A(t,a)))$$

Definitions. Let $f_A : A^u \longrightarrow A^v$, and suppose there exists some $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ such that $[\![\alpha]\!]_A = f_A$. Then depending on the context, we say f_A is *(simultaneous) primitive recursive (over A)*, f_A is *PR-computable (over A via α)*, and/or f_A is *defined by (simultaneous) primitive recursion (over A)*. The collection of all (simultaneous) primitive recursive functions over A is denoted PR(A); that is,

$$\mathrm{PR}(A) = \{ [\![\alpha]\!]_A : \alpha \in \mathrm{PR}(\Sigma) \}$$

When $f_A \in \mathrm{PR}(A)$, 'α_f' usually denotes that scheme such that $[\![\alpha_f]\!]_A = f_A$; here we say α_f *computes* f_A *(over* A*)*.

Discussion. There are three points about the preceding definitions worth mentioning. First, note that the use of 'A' as a subscript in '$[\![\alpha]\!]_A$' means more than $[\![\alpha]\!]_A$ being a function on the carriers of A; here it means that A is an algebra in which all the operation symbols involved in α have interpretations.

Second, it is important to realise that the function $[\![\,.\,]\!]_A$ is defined for *any standard* Σ-algebra A; in particular, if A is standard and $B \cong A$ or $B = \underline{A}$, then B is also standard and so $[\![\,.\,]\!]_B$ is a well-defined quantity.

Finally, although we call $\mathrm{PR}(A)$ the class of 'simultaneous' primitive recursive functions over A, there does not seem to be much that is simultaneous about the functions defined by Definition 3.3.2, in particular, those defined by clause (vii). However, let us consider the interpretation of a scheme $\alpha \in \mathrm{PR}(\Sigma)$ which is of the form $\alpha = *(\alpha_1, \alpha_2)$. Suppose $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{Tuv,v}$ for some $u, v \in S^+$; then $\alpha \in \mathrm{PR}(\Sigma)_{Tu,v}$ (by clause (vii) of definition 3.3.1). By Definition 3.3.2 we have

$$[\![\alpha_1]\!]_A : A^u \longrightarrow A^v,$$
$$[\![\alpha_2]\!]_A : T \times A^u \times A^v \longrightarrow A^v,$$

and so

$$[\![\alpha]\!]_A : T \times A^u \longrightarrow A^v$$

Since $[\![\alpha]\!]_A$ is just a function on A (for a general scheme $\alpha \in \mathrm{PR}(\Sigma)$), let us further simplify the notation by letting f, g, and h, abbreviate $[\![\alpha]\!]_A$, $[\![\alpha_1]\!]_A$, and $[\![\alpha_2]\!]_A$, respectively. Then from clause (vii) of Definition 3.3.2 f is defined by:

$$\begin{aligned} f(0,a) &= g(a) \\ f(t+1,a) &= h(t,a,f(t,a)) \end{aligned} \tag{1}$$

for each $t \in T$ and $a \in A^u$. We can see that this definition *is* simultaneous by focusing on how vector-valued functions are defined. Just for the moment, let f and g be any functions with $f, g : A^u \longrightarrow A^v$ say, and suppose f and g have coordinate functions $f_1, \ldots, f_m$ and $g_1, \ldots, g_m$ (so $|v| = m$). If f is defined by

$$(\forall a \in A^u)\ (f(a) = g(a))$$

When expanded in terms of coordinate functions, this definition becomes

$$(\forall a \in A^u)\ ((f_1(a), \ldots, f_m(a)) = (g_1(a), \ldots, g_m(a)))$$

which, when regarded coordinate-wise, is an obvious short-hand for defining $f_1, \ldots, f_m$ by

$$\begin{aligned} f_1(a) &= g_1(a) \\ &\vdots \quad \vdots \quad \vdots \\ f_m(a) &= g_m(a) \end{aligned}$$

for each $a \in A^u$.

Returning to the function f as defined by (1) above, we see that (1) *is* simultaneous when we 'unfold' the definition of f to obtain a definition of the coordinates $f_1, \ldots, f_k$ of f in terms of the coordinates $g_1, \ldots, g_k$ of g and the coordinates $h_1, \ldots, h_k$ of h:

$$f_1(0,a) = g_1(a)$$
$$\vdots \quad \vdots \quad \vdots$$
$$f_k(0,a) = g_k(a)$$

and

$$f_1(t+1,a) = h_1(t,a,f_1(t,a),\ldots,f_k(t,a))$$
$$\vdots \quad \vdots \quad \vdots$$
$$f_k(t+1,a) = h_k(t,a,f_1(t,a),\ldots,f_k(t,a))$$ □

Before we consider an example, let us complete the definition of PR by defining a length of computation function for PR-schema.

3.3.3 Performance.

Let P be a performance measure for Σ-algebra A. For each $\alpha \in PR(\Sigma)$, the *length of computation function* for α *with respect to* P is denoted by $\lambda_P(\alpha)$ where λ_P is the S^+-indexed family

$$\lambda_P = <\lambda_P^u : u \in S^+>$$

of mappings $\lambda_P^u : \bigcup_{v \in S^+} PR(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow C^+]$. Each mapping λ_P^u (ambiguously denoted λ_P) is defined uniformly in u by induction on the structural complexity of a scheme $\alpha \in PR(\Sigma)_{u,v}$ as follows:

Basis Schema.

(i) *Constant Functions.* If $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$, then

$$\lambda_P(\alpha) : A^w \longrightarrow C^+$$

is defined by

$$(\forall a \in A^w)\ (\lambda_P(\alpha)(a) = c^P)$$

(ii) *Algebraic Operations.* If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$, then

$$\lambda_P(\alpha) : A^w \longrightarrow C^+$$

is defined by

$$(\forall a \in A^w)\ (\lambda_P(\alpha)(a) = \sigma^P(a))$$

(iii) *Projection Functions.* If $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \le i \le n = |w|$, then

$$\lambda_P(\alpha) : A^w \longrightarrow C^+$$

is defined by

$$(\forall a \in A^w)\ (\lambda_P(\alpha)(a) = 1)$$

Induction: Function Building Tools.

(iv) *Definition-by-Cases.* Suppose $\alpha = DC(\beta,\alpha_1,\alpha_2)$ where for some $u,v \in S^+$ $\beta \in PR(\Sigma)_{u,s}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$. Then

$$\lambda_P(\alpha) : A^u \longrightarrow C^+$$

is defined by

$$(\forall a \in A^u)\ \lambda_P(\alpha)(a) = \lambda_P(\beta)(a) + \begin{cases} \lambda_P(\alpha_1)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P(\alpha_2)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

(v) *Vectorisation.* Suppose for some $m \geq 1$, $\alpha = <\alpha_1,...,\alpha_m>$ where for some $u \in S^+$ and $v \in S^m$, $\alpha_i \in \mathrm{PR}(\Sigma)_{u,v_i}$ for $i = 1,...,m$. Then

$$\lambda_P(\alpha) : A^u \longrightarrow C^+$$

is defined by

$$(\forall a \in A^u)\ (\lambda_P(\alpha)(a) = \max\{\lambda_P(\alpha_1)(a),...,\lambda_P(\alpha_m)(a)\})$$

(vi) *Composition.* Suppose $\alpha = \alpha_2 \circ \alpha_1$ where for some $u,v,w \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$. Then $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$.

$$\lambda_P(\alpha) : A^u \longrightarrow C^+$$

is defined by

$$(\forall a \in A^u)\ (\lambda_P(\alpha)(a) = \lambda_P(\alpha_1)(a) + \lambda_P(\alpha_2)([\![\alpha_1]\!]_A(a)))$$

(vii) *Primitive Recursion.* Suppose $\alpha = *(\alpha_1,\alpha_2)$ where for some $u,v \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\tau u\, v,v}$. Then

$$\lambda_P(\alpha) : T \times A^u \longrightarrow C^+$$

is defined by

$$(\forall a \in A^u)\ (\lambda_P(\alpha)(0,a) = \lambda_P(\alpha_1)(a))$$

and

$$(\forall t \in T)(\forall a \in A^u)\ (\lambda_P(\alpha)(t+1,a) = \lambda_P(\alpha)(t,a) + \lambda_P(\alpha_2)(t,a,[\![\alpha]\!]_A(t,a)))$$

Discussion. In this way we calculate the cost of evaluating a scheme over A. For operations of the underlying algebra, the cost of evaluation is just the cost as determined by P. Projection functions always access a single datum and so we charge one unit for evaluation. For the vectorisation of functions $\alpha = <\alpha_1,...,\alpha_m>$ we imagine $\alpha_1,...,\alpha_m$ to be executed in parallel, and the time taken to evaluate α on an argument a is the maximum time taken to evaluate any coordinate scheme on a. For the composition of functions $\alpha = \alpha_2 \circ \alpha_1$, we imagine each α_i to be executed sequentially, and thus we take as the time required to evaluate α to be the sum of evaluation α_1 and α_2 on their respective arguments. The costs of $\mathrm{DC}(\beta,\alpha_1,\alpha_2)$ and $*(\alpha_1,\alpha_2)$ are arrived at along similar principles.

Example. As a simple example of how to calculate with PR schema, let A be an S-sorted Σ-algebra where $\mathrm{D} \in S$, and Σ is defined by $\Sigma_{\mathrm{DD},\mathrm{B}} = \{\leq\}$ and $\Sigma_{w,s} = \emptyset$ for other w,s. If $\leq^A$ is a linear ordering $\leq_D$ on $A_{\mathrm{D}} = D$ then we can show the maximum function on D is PR-computable over A as follows:

Semantically, $\max_D : D \times D \longrightarrow D$ is defined by

$$\max_D\{a,b\} = \begin{cases} a & \text{if } b \leq_D a \\ b & \text{otherwise} \end{cases}$$

for each $a,b \in D$.

To show $\max_D \in \mathrm{PR}(A)$, we must find some $\alpha_{\max} \in \mathrm{PR}(\Sigma)_{\mathrm{DD},\mathrm{D}}$ such that $\alpha_{\max}$ computes the maximum function. To do this, we let the semantic definition of '$\max_D$' guide us in synthesising such a scheme.

We start by considering an element $(a,b) \in D \times D$. We must first access a and b by projecting them out of the vector (a,b). This is achieved by projection functions of course: notice from clause (iii)

of Definition 3.3.2 we have

$$[\![U_1^{DD}]\!]_A(a,b) = a \quad \text{and} \quad [\![U_2^{DD}]\!]_A(a,b) = b \tag{2}$$

Now, to compute '$b \leq_D a$' we must apply '$\leq_D$' to the pair (b,a) (and not (a,b) of course). Notice $U_i^{DD} \in PR(\Sigma)_{DD,D}$ for $i = 1,2$ (by clause (iii) of Definition 3.3.1), and so if we define α by

$$\alpha = \langle U_2^{DD}, U_1^{DD} \rangle$$

then $\alpha \in PR(\Sigma)_{DD,DD}$ (by clause (v) of Definition 3.3.1). Furthermore for each $(a,b) \in D \times D$ we have

$$\begin{aligned} [\![\alpha]\!]_A(a,b) &= [\![\langle U_2^{DD}, U_1^{DD} \rangle]\!]_A(a,b) \\ &= ([\![U_2^{DD}]\!]_A(a,b), [\![U_1^{DD}]\!]_A(a,b)) \end{aligned}$$

(by clause (v) of Definition 3.3.2)

$$= (b,a) \tag{3}$$

from (2). Now, since the codomain of α is the same as the domain of $\leq_D$, we have that β defined by $\beta = \leq \circ \alpha$ is a well-defined member of $PR(\Sigma)_{DD,B}$ (by clause (vi) of Definition 3.3.1, via clause (ii)). We now calculate as follows for each $(a,b) \in D \times D$:

$$\begin{aligned} [\![\beta]\!]_A(a,b) &= [\![\leq \circ \alpha]\!]_A(a,b) \\ &= [\![\leq]\!]_A([\![\alpha]\!]_A(a,b)) \end{aligned}$$

(by clause (v) of Definition 3.3.2)

$$= \leq^A([\![\alpha]\!]_A(a,b))$$

(by clause (ii) of Definition 3.3.2)

$$= \leq^A(b,a)$$

(from (3))

$$= \leq_D(b,a) \tag{4}$$

The three schemes β, U_1^{DD}, and U_2^{DD}, are of the appropriate arities to combine with the 'DC' operator: let

$$\alpha_{max} = DC(\beta, U_1^{DD}, U_2^{DD})$$

Then $\alpha \in PR(\Sigma)_{DD,D}$ (by clause (v) of Definition 3.3.1). Furthermore, for each $(a,b) \in D \times D$, we calculate

$$\begin{aligned} [\![\alpha_{max}]\!]_A(a,b) &= [\![DC(\beta, U_1^{DD}, U_2^{DD})]\!]_A(a,b) \\ &= \begin{cases} [\![U_1^{DD}]\!]_A(a,b) & \text{if } [\![\beta]\!]_A(a,b) \\ [\![U_2^{DD}]\!]_A(a,b) & \text{otherwise} \end{cases} \end{aligned}$$

(by clause (iv) of Definition 3.3.2)

$$= \begin{cases} a & \text{if } \leq_D(b,a) \\ b & \text{otherwise} \end{cases}$$

(by (4) and (2))

$$= \max_D\{a,b\}$$

Thus $\max_D$ is PR-computable over A.

We can calculate the complexity of evaluating α_{max} on an argument $(a,b) \in D \times D$ according to A-time as follows:

$$\lambda_P(\alpha)(a,b) = \lambda_P(\mathrm{DC}(\beta,U_1^{DD},U_2^{DD}))(a,b)$$

$$= \lambda_P(\beta)(a,b) + \begin{cases} \lambda_P(U_1^{DD})(a,b) & \text{if } [\![\beta]\!]_A(a,b) = tt \\ \lambda_P(U_2^{DD})(a,b) & \text{otherwise} \end{cases}$$

(by clause (iv) of Definition 3.3.3)

$$= \lambda_P(\beta)(a,b) + \begin{cases} \lambda_P(U_1^{DD})(a,b) & \text{if } \leq_D(b,a) \\ \lambda_P(U_2^{DD})(a,b) & \text{otherwise} \end{cases}$$

(by (4))

$$= \lambda_P(\beta)(a,b) + \begin{cases} 1 & \text{if } \leq_D(b,a) \\ 1 & \text{otherwise} \end{cases}$$

(by clause (iii) of Definition 3.3.3).

$$= \lambda_P(\beta)(a,b) + 1$$

By definition of β we now have

$$\lambda_P(\alpha)(a,b) = \lambda_P(\alpha \circ \leq)(a,b) + 1$$

$$= \lambda_P(\alpha)(a,b) + \lambda_P(\leq)([\![\alpha]\!]_A(a,b)) + 1$$

(by clause (vi) of Definition 3.3.3)

$$= \lambda_P(\alpha)(a,b) + \leq^P([\![\alpha]\!]_A(a,b)) + 1$$

(by clause (ii) of Definition 3.3.3)

$$= \lambda_P(\alpha)(a,b) + 2$$

(since P measures A-time)

$$= \lambda_P(<U_1^{DD},U_2^{DD}>)(a,b) + 2$$

(by definition of α)

$$= \max\{\lambda_P(U_1^{DD})(a,b), \lambda_P(U_2^{DD})(a,b)\} + 2$$

(by clause (v) of Definition 3.3.3)

$$= \max\{1,1\} + 2$$

(by clause (iii) of Definition 3.3.3)

$$= 3$$

□

Discussion. We think of a PR scheme in two ways. First and foremost, we think of a scheme as a function definition; for example, $<U_2^{ss},U_1^{ss}>$ defines a function which swaps its two arguments. Secondly, we think of a PR scheme as a *program*: there is an algorithmic procedure associated with any PR function definition, namely the procedure which begins with a scheme α and an argument a, and comprises the step-wise process of calculating $[\![\alpha]\!]_A(a)$ according to the rules given by Definition 3.3.2. Note that this procedure is *effective* in the sense that there are *always* finitely many steps in calculating $[\![\alpha]\!]_A(a)$ for any α and a.

3.3.4 Derived Schema and Function Building Tools.

As further examples of how we use PR, we introduce notations for expressing identity functions and the *iteration* of a function.

Identity Schema. For any $w \in S^+$ we define

$$Id^w = \langle U_1^w, \ldots, U_n^w \rangle$$

where $n = |w|$.

Thus $Id^w \in PR(\Sigma)_{w,w}$, and for any $w \in S^+$, $[\![Id^w]\!]_A$ is the identity function on A^w; that is,

$$(\forall a \in A^w)\ ([\![Id^w]\!]_A(a) = a)$$

Iteration. Given a function $f : A^w \longrightarrow A^w$ for some $w \in S^+$, we can define a new function $g : T \times A^w \longrightarrow A^w$ where for each $t \in T$ and $a \in A^w$, $g(t,a)$ is the result of applying f to a t times. Traditionally $g(t,a)$ is denoted by $f^{(t)}(a)$, and g is called the *iteration* of f. It will be useful to include iteration as a new function building tool; we do this as follows:

Let $\alpha \in PR(\Sigma)_{w,w}$ for any $w \in S^+$. We define the *iteration* of α, $it(\alpha)$ by

$$it(\alpha) = *(Id^w, \alpha \circ \langle U_{2+n}^{Tww}, \ldots, U_{1+2n}^{Tww} \rangle)$$

where $n = |w|$. (Notice $it(\alpha) \in PR(\Sigma)_{Tw,w}$.) □

The following lemma establishes that $[\![it(\alpha)]\!]_A(t,a) = [\![\alpha]\!]_A^{(t)}(a)$:

3.3.5 Lemma. *Let $\alpha \in PR(\Sigma)_{w,w}$ for any $w \in S^+$. Then for any $a \in A^w$, $[\![it(\alpha)]\!]_A$ satisfies:*

$$[\![it(\alpha)]\!]_A(0,a) = a \tag{5}$$

$$[\![it(\alpha)]\!]_A(t+1,a) = [\![\alpha]\!]_A([\![it(\alpha)]\!]_A(t,a))$$

for each $t \in T$.

Proof. Let $n = |w|$. Then

$$it(\alpha) = *(Id^w, \beta)$$

where

$$\beta = \alpha \circ \langle U_{2+n}^{Tww}, \ldots, U_{1+2n}^{Tww} \rangle$$

Choose $a \in A^w$. We must show that for every $t \in T$, the value of $[\![it(\alpha)]\!]_A(t,a)$ is as predicted by (5).

First we consider $[\![it(\alpha)]\!]_A(0,a)$:

$$\begin{aligned} [\![it(\alpha)]\!]_A(0,a) &= [\![*(Id^w, \beta)]\!]_A(0,a) \\ &= [\![Id^w]\!]_A(a) \\ &= a \end{aligned}$$

since Id^w is an identity scheme. Thus the value of $[\![it(\alpha)]\!]_A(0,a)$ is as predicted by (5).

Now notice that since $|w| = n$, for any vector $x \in A^{Tww}$ the last n coordinates of x are x_i for $i = 2+n, \ldots, 1+2n$, and thus if $x = (t,a,b)$ for some $t \in T$ and some $a,b \in A^w$, then $x_{1+n+i} = b_i$ for $i = 1, \ldots, n$. Hence for each $t \in T$ and $a,b \in \in A^w$,

$$\begin{aligned} [\![\beta]\!]_A(t,a,b) &= [\![\alpha]\!]_A([\![\langle U_{2+n}^{Tww}, \ldots, U_{1+2n}^{Tww} \rangle]\!]_A(t,a,b)) \\ &= [\![\alpha]\!]_A(b) \end{aligned} \tag{6}$$

Now choose $t \in T$ and $a \in A^w$. Then,

$$\begin{aligned}[it(\alpha)]_A(t+1,a) &= [\![*(Id^w,\beta)]\!]_A(t+1,a)\\ &= [\![\beta]\!]_A(t,a,[\![it(\alpha)]\!]_A(t,a))\\ &= [\![\alpha]\!]_A([\![it(\alpha)]\!]_A(t,a))\end{aligned}$$

by (6). Thus the value of $[it(\alpha)]_A(t+1,a)$ is also as predicted by (5). □

3.4 SPECIFYING SYNCHRONOUS ALGORITHMS.

In Section 2.4 we saw how the behaviour of a synchronous algorithm or network N could be mathematically described via the concept of a value function V_N, and we noted that this V_N was intuitively defined by simultaneous primitive recursion over the network's module specifications. In this chapter we have formalised simultaneous primitive recursion over a general collection of functions, and so it is now time to make explicit the connection between the account of synchronous algorithms given in Chapter 2 and the formal system PR; in doing so we provide the necessary formalisation promised in Section 2.5.1.

Our principal objective is to show that $V_N \in \mathrm{PR}(A)$ for some algebra A. Before we do so however, we need to introduce some notation for describing synchronous networks in the context of a many-sorted algebra. (Recall that in Chapter 2 synchronous networks were described as processing data taken from a *single*-sorted algebra A.) In Section 3.4.1 we introduce many-sorted algebra notation for describing the data sets and operations involved in a synchronous network, and in Section 3.4.2 we generalise network specifications to the many-sorted case.

3.4.1 Data Types.

The formal specification of a synchronous algorithm begins with a formal specification of the data sets and operations involved in that algorithm. In this section we will explain how a given synchronous network *determines* an algebra that is the basis for the formal specification of the network.

Since every synchronous algorithm involves a concept of time, and for convenience we regard Boolean values and operations as always available, it is appropriate to begin formal specifications with the smallest possible standard sort set S and signature Σ: thus $S = \{\mathrm{T}, \mathrm{B}\}$ and $\Sigma = (\mathrm{zero}, \mathrm{succ}, \mathrm{true}, \mathrm{false}, \mathrm{not}, \mathrm{or}, \mathrm{and})$ (see Section 3.1.8). Now let N be a k-module synchronous network. We adjoin new sorts to S and new symbols to Σ in the following way. Suppose we intend the ith module m_i of N to be specified by a function $f_i : A^{w(i)} \longrightarrow A_{s(i)}$ or $f_i : T \times A^{w(i)} \longrightarrow A_{s(i)}$ for $i = 1,\ldots,k$; then we extend S by adding the sort symbol $s(i)$ and all the sort symbols comprising the word $w(i)$ for $i = 1,\ldots,k$. Also for $i = 1,\ldots,k$ we provide a name for f_i by adjoining a new symbol σ_i to $\Sigma_{w(i),s(i)}$ if m_i is intended to be autonomous, or to $\Sigma_{Tw(i),s(i)}$ otherwise.

(Note that a sort symbol for the kind of data supplied by each source should also be adjoined to S. However, since we will always assume that the output channel of every source is an input channel to some module, the sort associated with the source is included in S when we consider the domain of the function that specifies the module in question.)

A picture of N together with the sort set $S = S_N$ and signature $\Sigma = \Sigma_N$ described above are sufficient to describe the syntactic aspects of N; of course, to specify the actual data sets and operations involved in N we give a Σ-algebra $A = A_N$. Finally, since network input varies with time in the general case, we must introduce streams of data; this is accomplished by moving from A to the stream algebra $\underline{A} = \underline{A_N}$ (see Section 3.1.8).

3.4.2 Generalised Functional Specifications of Network Behaviour.

In Section 2.4 we introduced six functions for specifying the behaviour of an n-source, k-module, m-sink synchronous network over data set A. In summary, these functions were (in order of appearance):

$$V_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^k$$
$$F_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^m$$
$$G_N : [T \longrightarrow A^n] \times A^k \longrightarrow [T \longrightarrow A^m]$$
$$v_N, w_N : T \times A^n \times A^k \longrightarrow A^k$$
$$f_N : T \times A^n \times A^k \longrightarrow A^m$$

At this point the reader should recall the definition and interpretation of each of these functions. Below we generalise each function to the more general case where different modules can hold different kinds of data and different sources can supply streams over different data sets.

Consider the occurrence of 'A^k' in the domains of the above functions: this is the set of all vectors of data that can be collectively held by the modules of N at any given time t in the single-sorted case. Now, in the many-sorted situation described just above, the ith module holds data from $A_{s(i)}$ for $i = 1,\ldots,k$, and so at each time $t \in T$ the k modules will collectively hold some vector from A^w where $w = s(1) \cdots s(k)$. Intuitively then, this A^w is the appropriate generalisation of A^k to the many-sorted case.

Now consider the occurrence of 'A^n' in the domains of our functional specifications: this is the set of all vectors of input data that can be collectively supplied by the sources of N at any given time t in the single-sorted case. In the many-sorted case the ith source of N can supply data from $A_{s'(i)}$ for some sort symbol $s'(i) \in S = S_N$ for $i = 1,\ldots,n$, and so the appropriate many-sorted generalisation of A^n is A^u where $u = s'(1) \cdots s'(k)$: at each time t the n sources of N collectively supply a vector from A^u.

What is the many-sorted generalisation of 'A^m'? For $j = 1,\ldots,m$, let i_j be the (unique) index of that module of N that supplies the jth sink with output data; then since that module holds data of sort $s(i_j)$ (in the many-sorted case described above) the jth sink receives data also of sort $s(i_j)$. Clearly, the many-sorted generalisation of A^m is A^v where $v = s(i_1) \cdots s(i_m)$.

It should be clear that under these hypotheses on the sorts of data associated with the sources and modules of a synchronous network, the specification methodology of Section 2.4 extends uniformly to the many-sorted case to yield generalised specifications of the form:

$$V_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^w$$

$$F_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^v$$
$$G_N : [T \longrightarrow A^u] \times A^w \longrightarrow [T \longrightarrow A^v]$$
$$v_N, w_N : T \times A^u \times A^w \longrightarrow A^w$$
$$f_N : T \times A^u \times A^w \longrightarrow A^v$$

As far as possible we will use the notation introduced above in a consistent manner. For example, we will continue to use n, k, and m, for the number of sources, modules, and sinks, respectively, of a synchronous network, and u, w, and v, for the 'types' of the vectors collectively held at a network's sources, modules, and sinks, respectively. □

We now formalise the specification of synchronous algorithms as promised earlier. In Theorem 3.4.3 we prove V_N and F_N are members of PR($\underline{A}$), and in Theorem 3.4.4 we prove v_N and f_N are members of PR(A) (note that we *do* mean 'A' here and not '$\underline{A}$': static specifications do not involve streams of data).

3.4.3 Theorem. *If N is an n-source, k-module, m-sink synchronous network over S-sorted Σ-algebra $A = A_N$, then $V_N \in \mathrm{PR}(\underline{A})$ and $F_N \in \mathrm{PR}(\underline{A})$.*

Proof. To prove the theorem we must find some $\alpha_N, \beta_N \in \mathrm{PR}(\underline{\Sigma})$ such that $[\![\alpha_N]\!]_{\underline{A}} = V_N$ and $[\![\beta_N]\!]_{\underline{A}} = F_N$.

Let us consider V_N first. Notice that for us to have $[\![\alpha_N]\!]_{\underline{A}} = V_N$, $[\![\alpha_N]\!]_{\underline{A}}$ must have the same functionality as V_N. Now, under the given hypotheses on N, V_N must have functionality

$$V_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^w$$

(for some $u \in S^n$ and $w \in S^k$) and so α_N must have arity $(\mathrm{T}\underline{uw}, w)$; that is, α_N must be a member of $\mathrm{PR}(\underline{\Sigma})_{\mathrm{T}\underline{uw}, w}$, because then we will have

$$[\![\alpha]\!]_{\underline{A}} : \underline{A}^{\mathrm{T}\underline{uw}} \longrightarrow \underline{A}^w$$

that is,

$$[\![\alpha]\!]_{\underline{A}} : T \times \underline{A}^{\underline{u}} \times \underline{A}^w \longrightarrow \underline{A}^w$$

But by definition of $\underline{A}$ for a general algebra A (see Section 3.1.8) $\underline{A}^{\underline{u}} = [T \longrightarrow A^u]$ and $\underline{A}^w = A^w$, and so

$$[\![\alpha_N]\!]_{\underline{A}} : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^w$$

as required (cf. Notation 3.1.9).

Define α_N by $\alpha_N = *(\alpha_1, \alpha_2)$ where

$$\alpha_1 = \langle \mathrm{U}^{\underline{uw}}_{n+1}, \ldots, \mathrm{U}^{\underline{uw}}_{n+k} \rangle$$

and

$$\alpha_2 = \langle \alpha_{2,1}, \ldots, \alpha_{2,k} \rangle$$

where, if the ith module m_i of N is autonomous, then for $i = 1, \ldots, k$,

$$\alpha_{2,i} = \sigma_i \circ \langle \eta_1, \ldots, \eta_{n_i} \rangle$$

where n_i is the number of inputs to m_i, and for $j = 1, \ldots, n_i$,

$$\eta_j = \begin{cases} \mathrm{eval}_{u_p} \circ \langle U_1^{\tau \underline{u} w}, U_{1+p}^{\tau \underline{u} w} \rangle & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ U_{1+n+k+q}^{\tau \underline{u} w} & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

and if m_i is nonautonomous, then for $i = 1,\ldots,k$,

$$\alpha_{2,i} = \sigma_i \circ \langle U_1^{\tau \underline{u} w}, \eta_1, \ldots, \eta_{n_i} \rangle$$

where n_i and $\eta_1,\ldots,\eta_{n_i}$ are as above.

It is not difficult to check that $\alpha_N \in \mathrm{PR}(\underline{\Sigma})_{\tau \underline{u} w, w}$ as required. Also, a routine inductive proof on t (which we also leave as an exercise) yields

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w) \ \ ([\![\alpha_N]\!]_{\underline{A}}(t, \underline{a}, x) = V_N(t, \underline{a}, x))$$

That is, $V_N = [\![\alpha_N]\!]_{\underline{A}}$ and so we conclude $V_N \in \mathrm{PR}(\underline{A})$.

Let us now consider F_N. Since N has m sinks we must have

$$F_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^v$$

for some $v \in S^m$, and so we seek a scheme $\beta_N \in \mathrm{PR}(\underline{\Sigma})$ of arity $(\tau \underline{u} w, v)$.

Recall that F_N is defined from V_N by

$$F_N(t, \underline{a}, x) = \pi_N(V_N(t, \underline{a}, x))$$

for each $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x \in A^w$, where π_N (in the many-sorted case) is the function $\pi_N : A^w \longrightarrow A^v$ defined by

$$\pi_N(a_1,\ldots,a_k) = (a_{i_1},\ldots,a_{i_m})$$

(Here, as in Sections 2.4.3 and 3.4.2, for $j = 1,\ldots,m$, i_j is the unique index of the module whose output channel supplies the jth sink of N.)

Now define $\alpha_\pi \in \mathrm{PR}(\underline{\Sigma})_{w,v}$ by

$$\alpha_\pi = \langle U_{i_1}^w, \ldots, U_{i_m}^w \rangle$$

Then $[\![\alpha_\pi]\!]_{\underline{A}} = \pi_N$. It should now be clear that if we define β_N by

$$\beta_N = \alpha_\pi \circ \alpha_N$$

then $\beta_N \in \mathrm{PR}(\underline{\Sigma})_{\tau \underline{u} w, v}$ as required, and $[\![\beta_N]\!]_{\underline{A}}(t, \underline{a}, x) = F_N(t, \underline{a}, x)$ for each $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x \in A^w$. (Again we leave the reader to fill in the details.) Thus $F_N \in \mathrm{PR}(\underline{A})$ as claimed.

3.4.4 Theorem. *If N is an n-source, k-module, m-sink synchronous network over S-sorted Σ-algebra $A = A_N$, then $v_N \in \mathrm{PR}(A)$ and $f_N \in \mathrm{PR}(A)$.*

Proof. To prove the theorem we must find some $\gamma_N, \delta_N \in \mathrm{PR}(\Sigma)$ such that $[\![\gamma_N]\!]_A = v_N$ and $[\![\delta_N]\!]_A = f_N$.

Let us consider v_N first. Recall how the definition of v_N (Section 2.4.4) was almost identical to the definition of V_N (Section 2.4.1). It should not be too surprising then that the required scheme γ_N is almost identical to α_N: we define $\gamma_N = *(\gamma_1, \gamma_2)$ where

$$\gamma_1 = \langle U_{n+1}^{kw}, \ldots, U_{n+k}^{kw} \rangle$$

and

$$\gamma_2 = \langle \gamma_{2,1}, \ldots, \gamma_{2,k} \rangle$$

where, if the ith module m_i of N is autonomous, then for $i = 1,\ldots,k$,

$$\gamma_{2,i} = \sigma_i \circ \langle \eta_1, \ldots, \eta_{n_i} \rangle$$

where n_i is the number of inputs to m_i, and for $j = 1, \ldots, n_i$,

$$\eta_j = \begin{cases} U_{1+p}^{\tau+w+w} & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ U_{1+n+k+q}^{\tau+w+w} & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

and if m_i is nonautonomous, then for $i = 1, \ldots, k$,

$$\gamma_{2,i} = \sigma_i \circ \langle U_1^{\tau+w+w}, \eta_1, \ldots, \eta_{n_i} \rangle$$

where n_i and $\eta_1, \ldots, \eta_{n_i}$ are as above.

We leave the reader to check that $\gamma_N \in \mathrm{PR}(\Sigma)_{\tau+w+w,w}$ (so that $[\![\gamma]\!]_A$ has the same functionality as $f_N : T \times A^u \times A^w \longrightarrow A^w$), and that

$$(\forall t \in T)(\forall a \in A^u)(\forall x \in A^w) \ \ ([\![\gamma_N]\!]_A(t,a,x) = \nu_N(t,a,x))$$

That is, $\nu_N = [\![\gamma_N]\!]_A$ and so we conclude $\nu_N \in \mathrm{PR}(A)$.

Let us now consider f_N. Since N has m sinks we must have

$$f_N : T \times A^u \times A^w \longrightarrow A^v$$

for some $v \in S^m$, and so we seek a scheme $\delta_N \in \mathrm{PR}(\Sigma)$ of arity $(\tau+w+w, v)$.

Recall that f_N is defined from ν_N by

$$f_N(t, \underline{a}, x) = \pi_N(\nu_N(t, \underline{a}, x))$$

for each $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x \in A^w$ (where π_N is as in the proof of the previous theorem).

Clearly, the required scheme δ_N is

$$\delta_N = \alpha_\pi \circ \gamma_N$$

(where $\alpha_\pi \in \mathrm{PR}(\underline{\Sigma})_{w,v}$ is as in the proof of the previous theorem).

3.4.5 Notation.

Henceforth we will always use α_N, β_N, γ_N, and δ_N, for the schema that formalise V_N, F_N, ν_N, and f_N, respectively.

Discussion. We have now completed the task of formalising the specification of synchronous algorithms as promised in Chapter 2. There are two outstanding points to be covered however.

First, notice that since PR comes equipped with a complexity theory, we have a formal account of the *performance* of synchronous algorithms for free: the time complexity of a synchronous network from which we derive performance specifications *PSpec* (cf. Section 2.5.2), is simply $\lambda_P(\alpha_N)$ where P is any chosen account of the complexity of the data and operations of $\underline{A}$. The case that P is $\underline{A}$-time is of particular interest to us, and we will consider this in the next section.

Second, it is interesting to note that G_N, the stream transformation version of F_N (see Sections 2.2.4 and 2.4.3) is *not* simultaneous primitive recursive over $\underline{A}$, that is, $G_N \notin \mathrm{PR}(\underline{A})$, despite the facts that $F_N \in \mathrm{PR}(\underline{A})$ and G_N is 'almost the same as' F_N: in the many-sorted case F_N and G_N are maps of the form

$$F_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^v$$

and

$$G_N : [T \longrightarrow A^u] \times A^w \longrightarrow [T \longrightarrow A^v]$$

respectively, and (as in the many-sorted case) G_N is defined from F_N by

$$G_N(\underline{a},x)(t) = F_N(t,\underline{a},x)$$

for each $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x \in A^w$.

Technically, the non-primitive recursiveness of G_N means that unlike V_N, F_N, v_N, and f_N, the function G_N cannot have any formal status within our theory of of synchronous systems. One way of resolving this problem is to extend PR by adding *functional abstraction* as a new function building tool: given a map $f : X \times Y \longrightarrow Z$ where X, Y, and Z, are any sets, the map $g = abs(f) : Y \longrightarrow [X \longrightarrow Z]$ is said to be defined from f by functional abstraction if for each $y \in Y$ $g(y) : X \longrightarrow Z$ is the function defined by $g(y)(x) = f(x,y)$ for each $x \in X$. (Note that $G_N = abs(F_N)$.) We will not do this because we are mainly interested in value functions for which abstraction is not necessary, and adding abstraction complicates the class of functions.

A similar problem arises with the function w_N, the alternative version of v_N that was defined from V_N via the operator '*fix*' in Section 2.4.4. We noted that $w_N = v_N$ and v_N is simultaneous primitive recursive over A (Theorem 3.4.4), so $w_N \in \mathrm{PR}(A)$. However, the function *fix* is *not* simultaneous primitive recursive over $\underline{A}$ and so we read 'v_N' for 'w_N' whenever the formal status of functional specifications is an issue.

3.4.6 Exercise.

Prove $fix \notin \mathrm{PR}(\underline{A})$ and $G_N \notin \mathrm{PR}(\underline{A})$. (Hint: recall the definition of $\underline{A}$ for general A from Section 3.1.8. Notice that $\underline{A}$ contains no constants of sort 'stream', and no operations whose codomain is a set of streams: that is, for $s \in S$, $\underline{\Sigma}_{w,s} = \emptyset$ for every $w \in \underline{S}^+$.)

3.5 THEORETICAL RESULTS FOR PR.

In this section we gather together an assortment of facts about PR. The section has three parts: in the first part we derive formulae that predict the complexity of simple kinds of PR schema; in particular, we give a formula which is useful for calculating the complexity of a synchronous algorithm over A with respect to $\underline{A}$-time. In the second part we lay the theoretical foundations of PR as a *hierarchical* specification language; in particular, we make more substantial our informal remarks concerning augmentation of data types made in Section 3.1.5. In the final part we prove that PR-computability is an *isomorphism invariant.*

Performance Lemmas. Here are two simple facts whose proofs we leave as exercises:

3.5.1 Lemma. *Let A be an S-sorted Σ-algebra and let P be a performance measure for A. Suppose $\alpha = *(\alpha_1,\alpha_2) \in \mathrm{PR}(\Sigma)_{\tau u,v}$ for some $u,v \in S^+$ and some $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\tau uv,v}$. Further suppose there exist $c,k \geq 1$ such that*

$$(\forall a \in A^u)\ (\lambda_P(\alpha_1)(a) = c)$$

and

$$(\forall t \in T)(\forall a \in A^u)(\forall b \in A^v)\ (\lambda_P(\alpha_2)(t,a,b) = k)$$

Then

$$(\forall t \in T)(\forall a \in A^u)\ (\lambda_P(\alpha)(t,a) = c + kt)$$

3.5.2 Lemma. *Let A be an S-sorted Σ-algebra and let P be a performance measure for A. Also let $\alpha \in \mathrm{POL}(\Sigma)_{u,v}$ for some $u,v \in S^+$. If P measures A-time then there exists $k_\alpha \geq 1$ such that*

$$(\forall a \in A^u)\ (\lambda_P(\alpha)(a) = k_\alpha)$$ □

Putting Lemmas 3.5.1 and 3.5.2 together yields:

3.5.3 Lemma. *Let A be an S-sorted Σ-algebra and let P measure A-time. Suppose $\alpha = *(\alpha_1,\alpha_2) \in \mathrm{PR}(\Sigma)_{\tau u,v}$ for some $u,v \in S^+$ and some $\alpha_1 \in \mathrm{POL}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{POL}(\Sigma)_{\tau uv,v}$. Then there exist $c,k \geq 1$ such that*

$$(\forall t \in T)(\forall a \in A^u)\ (\lambda_P(\alpha)(t,a) = c + kt)$$

Proof. Immediate from Lemmas 3.5.1 and 3.5.2: in fact, $\lambda_P(\alpha)(t,a) = k_{\alpha_1} + k_{\alpha_2} t$ for each $t \in T$ and $a \in A^u$. □

The significance of these lemmas (especially Lemma 3.5.3) is that they hold for *all* (standard) Σ and A, in particular $\underline{\Sigma}$ and $\underline{A}$:

3.5.4 Lemma. *Let N be a synchronous network over $A = A_N$, and let $\alpha_N \in \mathrm{PR}(\underline{\Sigma})_{\tau \underline{u}w,w}$ and $\beta_N \in \mathrm{PR}(\underline{\Sigma})_{\tau \underline{u}w,v}$ be as defined in Theorem 3.4.3. If P measures $\underline{A}$-time then*

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w)\ (\lambda_P(\alpha_N)(t,\underline{a},x) = 1 + 3t) \tag{7}$$

and

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w)\ (\lambda_P(\beta_N)(t,\underline{a},x) = 2 + 3t) \tag{8}$$

Proof. To prove (7) first recall that α_N is defined by $\alpha_N = *(\alpha_1,\alpha_2)$ where

$$\alpha_1 = \langle U^{uw}_{n+1}, \ldots, U^{uw}_{n+k} \rangle$$

and

$$\alpha_2 = \langle \alpha_{2,1}, \ldots, \alpha_{2,k} \rangle$$

where $\alpha_{2,i} \in \mathrm{POL}(\underline{\Sigma})$; thus by (the proof of) Lemma 3.5.3, we have

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w)\ (\lambda_P(\alpha)(t,\underline{a},x) = k_{\alpha_1} + k_{\alpha_2} t)$$

Thus it remains to show that $k_{\alpha_1} = 1$ and $k_{\alpha_2} = 3$; that is, that

$$(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w)\ (\lambda_P(\alpha_1)(t,\underline{a},x) = 1) \tag{9}$$

and

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow A^u)(\forall x \in A^w)\ \ (\lambda_P(\alpha_2)(t,\underline{a},x) = 3) \tag{10}$$

The claim (9) is obviously true; let us prove (10).

Using the relevant scheme definitions and the definition of λ_P as given by Definition 3.3.3, for each $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x,y \in A^w$ we calculate as follows:

$$\begin{aligned}\lambda_P(\alpha_2)(t,\underline{a},x,y) &= \lambda_P(<\alpha_{2,1},\ldots,\alpha_{2,k}>)(t,\underline{a},x,y)\\ &= \max_{1\le i\le k}\{\lambda_P(\alpha_{2,i})(t,\underline{a},x,y)\}\end{aligned} \tag{11}$$

Now, for any $i \in [1,k]$ with m_i autonomous we have

$$\begin{aligned}\lambda_P(\alpha_{2,i})(t,\underline{a},x,y) &= \lambda_P(\sigma_i \circ <\beta_1,\ldots,\beta_{n_i}>)(t,\underline{a},x,y)\\ &= \lambda_P(<\beta_1,\ldots,\beta_{n_i}>)(t,\underline{a},x,y) + \sigma_i^P([\![<\beta_1,\ldots,\beta_{n_i}>]\!]_{\underline{A}})\\ &= \lambda_P(<\beta_1,\ldots,\beta_{n_i}>)(t,\underline{a},x,y) + 1\end{aligned}$$

(since P measures $\underline{A}$-time)

$$= \max_{1\le j\le n_i}\{\lambda_P(\beta_j)(t,\underline{a},x,y)\} + 1 \tag{12}$$

Now, for $j = 1,\ldots,n_i$ (and for $i = 1,\ldots,k$), if we let $R(i,j,p)$ abbreviate 'the jth input to module i is from source In_p' and $R'(i,j,q)$ abbreviate 'the jth input to module i is from module m_q' then

$$\begin{aligned}\lambda_P(\beta_j)(t,\underline{a},x,y) &= \begin{cases}\lambda_P(eval_{u_i} \circ <U_1^{t+w+w}, U_{1+p}^{t+w+w}>)(t,\underline{a},x,y) & \text{if } R(i,j,p)\\ \lambda_P(U_{1+n+k+q}^{t+w+w})(t,\underline{a},x,y) & \text{if } R'(i,j,q)\end{cases}\\ &= \begin{cases}1 + eval_{u_i}^P([\![<U_1^{t+w+w}, U_{1+p}^{t+w+w}>]\!]_{\underline{A}}(t,\underline{a},x,y)) & \text{if } R(i,j,p)\\ 1 & \text{if } R'(i,j,q)\end{cases}\\ &= \begin{cases}1+1 & \text{if } R(i,j,p)\\ 1 & \text{if } R'(i,j,q)\end{cases}\end{aligned}$$

(since P measures $\underline{A}$-time)

$$= \begin{cases}2 & \text{if } R(i,j,p)\\ 1 & \text{if } R'(i,j,q)\end{cases} \tag{13}$$

It is equally easy to show that (13) also holds when m_i is nonautonomous; this we leave as an exercise. Now let us call a module *interior* if none of its inputs comes from any source. Then from (13) we have

$$\max_{1\le j\le n_i}\{\lambda_P(\beta_j)(t,\underline{a},x,y)\} = \begin{cases}1 & \text{if } interior(i)\\ 2 & \text{otherwise}\end{cases} \tag{14}$$

Thus using (14) in (12) we get

$$\lambda_P(\alpha_{2,i})(t,\underline{a},x,y) = 1 + \begin{cases}1 & \text{if } interior(i)\\ 2 & \text{otherwise}\end{cases} \tag{15}$$

and using (15) in (11) we have

$$\lambda_P(\alpha_2)(t,\underline{a},x,y) = 3 \tag{16}$$

(since there must be at least one noninterior module).

Since (16) holds for all $t \in T$, $\underline{a} : T \longrightarrow A^u$, and $x,y \in A^w$, we have $k_{\alpha_2} = 3$ as required.

We now prove (8). First recall that β_N is defined by $\beta_N = \alpha_\pi \circ \alpha_N$ where α_π is as in Theorems 3.4.3 and 3.4.4. Now, since α_π is a vectorisation of projection functions, we have $\lambda_P(\alpha_\pi)(a) = 1$ for each $a \in A^w$. Thus, for each $t \in T$, $\underline{a} : T \longrightarrow A^w$, and $x \in A^w$, we have

$$\begin{aligned}\lambda_P(\beta_N)(t,\underline{a},x) &= \lambda_P(\alpha_\pi \circ \alpha_N)(t,\underline{a},x)\\ &= \lambda_P(\alpha_N)(t,\underline{a},x) + \lambda_P(\alpha_\pi)((\llbracket \alpha_N \rrbracket_{\underline{A}})(t,\underline{a},x))\\ &= 1+3t+1 = 2+3t.\end{aligned}$$

Discussion. The preceding lemma informs us that when we specify a synchronous algorithm in PR, we specify the algorithm in terms of concepts that are lower level than that of an algorithmic clock cycle and thus α_N actually specifies an *implementation* of N. Said differently, the lemma reveals that $\mathrm{PR}(\underline{A})$ is a *model* of synchronous computation: a model in which a 'step' of an algorithm is modelled by three actions: reading the clock; reading in input, and evaluating a function on the input; if we charge one time unit for each action ($\underline{A}$-time) then of course n steps of the algorithm requires time $3n$. In contrast, if we interpret a step as a time metric then the time required to execute a synchronous algorithm for n steps is n (by definition: it takes n steps to execute an algorithm for n steps!).

For completeness we conclude this part of the current section with the static counterpart of Lemma 3.5.4:

3.5.5 Lemma. *Let N be a synchronous network over $A = A_N$, and let $\gamma_N \in \mathrm{PR}(\Sigma)_{\tau_{N,w}}$ and $\delta_N \in \mathrm{PR}(\Sigma)_{\tau_{N,v}}$ be as defined in Theorem 3.4.4. If P measures $\underline{A}$-time then*

$$(\forall t \in T)(\forall a \in A^u)(\forall x \in A^w)\ \ (\lambda_P(\gamma_N)(t,\underline{a},x) = 1+2t) \tag{17}$$

and

$$(\forall t \in T)(\forall a \in A^u)(\forall x \in A^w)\ \ (\lambda_P(\delta_N)(t,\underline{a},x) = 2+2t) \tag{18}$$

Proof. Exercise.

Augmentation Lemmas. In this second part of the current section we will state and prove a lemma concerning PR schema defined over an augmented data-type. Ultimately this lemma is not used until Chapter 7 where it appears as a preliminary technical result; however since the lemma is of interest in its own right, we will explore its implications here at a more leisurely pace.

Given a (standard) algebra A, $\mathrm{PR}(A)$ is the set of all functions definable by primitive recursion over A. Alternatively, in an informal way we can think of $\mathrm{PR}(A)$ as the collection of all tasks which are solvable over A by means of a primitive recursive program. Now suppose that f_A is some single-valued function on A; for definiteness, suppose $f_A : A^w \longrightarrow A_s$ where w and s are drawn from the underlying sort set S. We can augment A with f_A to form the algebra (A, f_A), and intuitively, since (A, f_A) has more basic operations than A we expect to be able to solve more tasks over (A, f_A) than we could over A; mathematically, we anticipate $\mathrm{PR}(A, f_A) \supseteq \mathrm{PR}(A)$. Now consider the following statement:

$$f_A \in \mathrm{PR}(A) \Longrightarrow \mathrm{PR}(A, f_A) \subseteq \mathrm{PR}(A) \tag{19}$$

This statement (or assertion: (19) is the essence of the lemma we will eventually prove) can be regarded as a *computability* result which has implications for the theory of hierarchical design of algorithms

(ultimately *synchronous* algorithms).

Generally, computability results express bounds on what can be computed or solved with a particular model of computation. Of course, such results are foundational for computer science (there being no point in trying to solve a given task if it is already known to be impossible). Again generally, if L_1 and L_2 are programming languages and $T(L_1)$ and $T(L_2)$ denote the tasks solvable by L_1- and L_2-programs respectively, then the computability theorist is interested in if and when $T(L_1) \subseteq T(L_2)$, $T(L_2) \subseteq T(L_1)$, or $T(L_1) = T(L_2)$, expressing that what can be solved with L_1 is less than, more than, or just the same as what can be solved with L_2 respectively.

Now, we have argued above that (as sets of solvable tasks) $\mathrm{PR}(A)$ and $\mathrm{PR}(A, f_A)$ satisfy $\mathrm{PR}(A) \subseteq \mathrm{PR}(A, f_A)$ and (we claim) $f_A \in \mathrm{PR}(A) \Rightarrow \mathrm{PR}(A, f_A) \subseteq \mathrm{PR}(A)$. For general sets X and Y, $X \subseteq Y$ and $X \supseteq Y$ together imply $X = Y$; furthermore, $\mathrm{PR}(A) \subseteq \mathrm{PR}(A, f_A)$ independent of whether $f_A \in \mathrm{PR}(A)$, and thus

$$f_A \in \mathrm{PR}(A) \Rightarrow \mathrm{PR}(A, f_A) = \mathrm{PR}(A)$$

In fact, we can do better than this: primitive recursiveness of f_A is not only sufficient for $\mathrm{PR}(A, f_A)$ to be equal to $\mathrm{PR}(A)$, it is also *necessary*; that is,

$$f_A \in \mathrm{PR}(A) \Leftrightarrow \mathrm{PR}(A, f_A) = \mathrm{PR}(A)$$

To see why the reverse implication holds, first notice that any operation σ^A of an algebra A is always primitive recursive over A (trivially: if A has signature Σ, then $\alpha = \sigma \in \mathrm{PR}(\Sigma)$ by clause (ii) of Definition 3.3.2, and moreover $[\![\alpha]\!]_A$ is defined to be σ^A). Since f_A is an operation of (A, f_A), we have $f_A \in \mathrm{PR}(A, f_A)$ and thus $f \in \mathrm{PR}(A)$ (since $\mathrm{PR}(A, f_A)$ and $\mathrm{PR}(A)$ have the same members by hypothesis).

Now let us return to what (19) has to say about hierarchical design. Quite simply, it is this: to solve a task by means of a primitive recursive program over A, it is sufficient to find a primitive recursive program which solves the task over some other algebra (A, f_A) and then to establish that f_A is $\mathrm{PR}(A)$-computable (for then it follows that we can solve the task over A). In other words, bearing the discussion at the end of Section 3.1.5 in mind, the statement (19) is a theoretical justification of top-down design.

We can elaborate on this last remark by exploring (19) at the level of PR schema. By definition of $\mathrm{PR}(A)$, $f_A \in \mathrm{PR}(A)$ iff there exists some $\alpha_f \in \mathrm{PR}(\Sigma)_{w,s}$ (when $f_A : A^w \longrightarrow A_s$) such that $[\![\alpha_f]\!]_A = f_A$. Now suppose $\alpha_T \in \mathrm{PR}(\Sigma, \phi)$ is some scheme which solves a task T, but in doing so involves the symbol ϕ as a name for the function f_A. Since $[\![\alpha_T]\!]_{(A, f_A)} \in \mathrm{PR}(A, f_A)$ by definition, (19) asserts that $[\![\alpha_T]\!]_{(A, f_A)} \in \mathrm{PR}(A)$; to be more precise it asserts that there is some other scheme $\alpha'_T \in \mathrm{PR}(\Sigma)$ such that $[\![\alpha'_T]\!]_A = [\![\alpha_T]\!]_{(A, f_A)}$. This means that not only is there a program over (Σ, ϕ) which solves T, but also one over Σ. In other words, (19) implicitly asserts that there is a *correctness preserving transformation* form programs over (Σ, ϕ) to programs over Σ.

Let us consider an example to see where this transformation comes from.

Example. Consider the task T of computing the maximum function on a set D that is linearly ordered by a relation $\leq_D$. We suppose that T is to be solved by means of a PR program over some S-sorted Σ-algebra A where $\leq_D$ is named by $\leq \in \Sigma_{DD,B}$, $\leq^A = \leq_D : D \times D \longrightarrow \mathbb{B}$, $D = A_D$ and $\mathbb{B} = A_B$ for some symbols $D,B \in S$. Thus the required solution to T is a scheme $\alpha_T \in \mathrm{PR}(\Sigma)_{DD,D}$ such that

$$(\forall a,b \in D)\ (\ [\![\alpha_T]\!]_A(a,b) = \max_D \{\, a,b \,\}\ )$$

One natural algorithm for solving T is expressed in terms of $\geq_D$ (rather than $\leq_D$): letting '$\geq$' be a symbol to name $\geq_D$, take α_T to be the scheme

$$\alpha_T = \mathrm{DC}(\geq, U_1^{DD}, U_2^{DD})$$

Then $\alpha_D \in \mathrm{PR}(\Sigma,\geq; DD,D)_{DD,D}$ and if $B = (A, \geq_D)$ then for each $a,b \in D$ we have

$$[\![\alpha_T]\!]_B(a,b) = [\![\mathrm{DC}(\geq, U_1^{DD}, U_2^{DD})]\!]_B(a,b)$$

$$= \begin{cases} [\![U_1^{DD}]\!]_B(a,b) & \text{if } [\![\geq]\!]_B(a,b) \\ [\![U_2^{DD}]\!]_B(a,b) & \text{otherwise} \end{cases}$$

$$= \begin{cases} a & \text{if } \geq^B(a,b) \\ b & \text{otherwise} \end{cases}$$

$$= \begin{cases} a & \text{if } \geq_D(a,b) \\ b & \text{otherwise} \end{cases}$$

$$= \max_D \{\, a,b \,\}$$

Thus α_T solves T but over $B = (A, \geq_D)$.

It is easy to see that $\geq_D \in \mathrm{PR}(A)$ however: let $\alpha_{\geq} = \leq_\circ < U_2^{DD}, U_1^{DD} >$, then $\alpha_{\geq} \in \mathrm{PR}(\Sigma)_{DD,B}$ and it is easy to check that $[\![\alpha_{\geq}]\!]_A(a,b) = \geq_D(a,b)$ for each $a,b \in D$. According to (19) we should now have

$$\geq_D \in \mathrm{PR}(A) \implies \mathrm{PR}(A, \geq_D) \subseteq \mathrm{PR}(A)$$

so there must be some $\alpha_T' \in \mathrm{PR}(\Sigma)$ such that $[\![\alpha_T']\!]_A = [\![\alpha_T]\!]_B$ (recall $B = (A, \geq_D)$); such a scheme α_T' is the required solution to T over A of course since α_T' uses only the operations of A and $[\![\alpha_T']\!]_A = [\![\alpha_T]\!]_B = \max_D$.

It is not too difficult to see that α_T' can be obtained from α_T by substituting $\alpha_{\geq}$ for the symbol $\geq$ in α_T. Let us denote this substitution by $\alpha_T' = \alpha_T\{\alpha_{\geq}/\geq\}$ (generally the notation '$\{\beta/\phi\}$' is to be read as 'except that β is substituted for ϕ'). It is easy to check that α_T' is the scheme α_{max} of the previous example, and hence $[\![\alpha_T']\!]_A = [\![\alpha_{max}]\!]_A = \max_D$. Notice that we do not need to prove that α_T' solves T: correctness of α_T' is already guaranteed by what we have said about (19). □

Returning to a more general setting, we let $\{\beta/\phi\}$ be the transformation

$$\{\beta/\phi\} : \mathrm{PR}(\Sigma,\phi) \longrightarrow \mathrm{PR}(\Sigma)$$

where for each $\alpha \in \mathrm{PR}(\Sigma,\phi)$, $\alpha\{\beta/\phi\}$ is obtained by substituting β for *every* occurrence of ϕ in α. It is intuitively clear that (and this is supported by the example) if $w \in S^+$ and $s \in S$ are such that $\phi \in (\Sigma,\phi)_{w,s}$ then β must have arity (w,s) in order that the transformation $\{\beta/\phi\}$ preserves the arity of any scheme to which it is applied. Moreover, is again intuitively clear that if B is a (Σ,ϕ)-algebra in which ϕ is interpreted as $[\![\beta]\!]_A$, then $\{\beta/\phi\}$ will not only preserve the arity of a scheme to which it is applied but also

its meaning, and hence its correctness.

Analogously, if P and Q are performance measures for A and B respectively then the complexity of evaluating any scheme over (Σ,ϕ) will involve the complexity of evaluating ϕ^B, namely ϕ^Q; if ϕ^Q is identically the cost of evaluating β, namely $\lambda_P(\beta)$, then we expect $\{\beta/\phi\}$ to preserve the complexity of each $\alpha \in PR(\Sigma,\phi)$, that is, if $\alpha \in PR(\Sigma,\phi)_{u,v}$ for some $u,v \in S^+$, then we expect

$$Q = (P,\lambda_P(\beta)) \Rightarrow (\forall a \in A^u)(\lambda_P(\alpha\{\beta/\phi\})(a) = \lambda_Q(\alpha)(a)) \qquad (20)$$

Importantly (20) *predicts* the complexity of the transformed scheme without us having to calculate it at the lower level (that is, with respect to P: we only need to calculate the complexity of β with respect to higher level Q and the complexity of α with respect to P).

Exercise. In the context of the previous example, let P be any performance measure for A and let $Q = (P,\lambda_P(\alpha_2))$. Now prove

$$(\forall a,b \in D)\ (\lambda_P(\alpha_T')(a,b) = \lambda_Q(\alpha_T)(a,b))$$

Further Observations. We have seen how PR may be used to make theoretical observations about the hierarchical design of algorithms: we have seen a transformation between collections of high- and low-level PR schema that is correctness and performance preserving. We think of $PR(\Sigma)$ as a programming language and thus the principle content of (19) and (20) is that they assert the existence of a correctness and performance preserving *compiler* $c : PR(\Sigma,\phi) \longrightarrow PR(\Sigma)$; existence of this compiler is of paramount importance for it underwrites top-down design: we design at a high-level where an algorithm can be phrased in terms of abstract primitive operations making the algorithm amenable to formal verification; by using a correct and performance preserving compiler, we do not need to verify the algorithm at the level of its implementation where the algorithm may be so complex as to be impossible to verify in practice.

We will return to consider compilers in later chapters. Now we must prove (19) and (20).

Actually, we will prove slightly more general results: we prove that (variants of) (19) and (20) hold for vector-valued functions. By way of preliminary explanation, recall that when A is augmented with a vector-valued function $f_A = (f_1,\ldots,f_m)$ we augment A with the coordinates $f_1,\ldots,f_m$ of f_A, and the signature Σ of A is extended with a symbol ϕ_i for each coordinate f_i. Now, we want to show that if $f_A \in PR(A)$ then $PR(A,f_A) \subseteq PR(A)$. However, if $f_A \in PR(A)$ then for some $\beta \in PR(\Sigma)$ we have $[\![\beta]\!]_A = f_A$. Given a scheme $\alpha \in PR(\Sigma,\Phi)$ where $\Phi = (\phi_1,\ldots,\phi_m)$, to show that there exists some $\alpha' \in PR(\Sigma)$ with $[\![\alpha']\!]_A = [\![\alpha]\!]_{(A,f_A)}$, it follows that since each ϕ_i names the ith coordinate of f_A (viz f_i), we must replace ϕ_i by some $\beta_i \in PR(\Sigma)$ such that $[\![\beta_i]\!]_A = f_i$ for $i = 1,\ldots,m$; notice that $\beta_i = U_i^v \circ \beta$ is such a scheme when the codomain of β (or f_A) is v (or A^v).

Below we define a map $c = c(\beta,\Phi)$ which realises this substitution; we will then establish that given $\alpha \in PR(\Sigma,\Phi)$ $c(\alpha)$ has the same arity as α. Finally we prove that c is both meaning and performance preserving thus establishing (19) and (20) (as special cases).

3.5.6 Definition.

Let Σ be an S-sorted signature for some sort set S. Also let $\beta \in PR(\Sigma)_{u',v'}$ for some $u',v' \in S^+$, and let $\Phi = (\phi_1,...,\phi_{m'})$ where $m' = |v'|$ and $\{\phi_1,...,\phi_{m'}\} \cap \Sigma = \varnothing$. Now let $(\Sigma,\Phi) = (\Sigma,\Phi;u',v')$. We define

$$c = c(\beta,\Phi): PR(\Sigma,\Phi) \longrightarrow PR(\Sigma)$$

by induction on the structural complexity of arguments $\alpha \in PR(\Sigma,\Phi)$ as follows:

Basis Cases.

(i) *Constant Functions.* If $\alpha = c^w$ for some $c \in (\Sigma,\Phi)_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$, then $c(\alpha)$ is defined by

$$c(\alpha) = \alpha$$

(ii) *Algebraic Operations.* If $\alpha = \sigma$ for some $\sigma \in (\Sigma,\Phi)_{w,s}$ for some $w \in S^+$ and $s \in S$, then $c(\alpha)$ is defined by

$$c(\alpha) = \begin{cases} \alpha & \text{if } \sigma \in \Sigma \\ U_i^{v'} \circ \beta & \text{if } \sigma = \phi_i \in \Phi \end{cases}$$

(iii) *Projection Functions.* If $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \le i \le n = |w|$, then $c(\alpha)$ is defined by

$$c(\alpha) = \alpha$$

Induction.

(iv) *Definition-by-Cases.* Suppose $\alpha = DC(\beta',\alpha_1,\alpha_2)$ where for some $u,v \in S^+$ $\beta' \in PR(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$. Then $c(\alpha)$ is defined by

$$c(\alpha) = DC(c(\beta'),c(\alpha_1),c(\alpha_2))$$

(v) *Vectorisation.* Suppose for some $m \ge 1$, $\alpha = <\alpha_1,...,\alpha_m>$ where for some $u \in S^+$ and $v \in S^m$, $\alpha_i \in PR(\Sigma)_{u,v_i}$ for $i = 1,...,m$. Then $c(\alpha)$ is defined by

$$c(\alpha) = <c(\alpha_1),...,c(\alpha_m)>$$

(vi) *Composition.* Suppose $\alpha = \alpha_2 \circ \alpha_1$ where for some $u,v,w \in S^+$, $\alpha_1 \in PR(\Sigma)_{u,w}$ and $\alpha_2 \in PR(\Sigma)_{w,v}$. Then $c(\alpha)$ is defined by

$$c(\alpha) = c(\alpha_2) \circ c(\alpha_1)$$

(vii) *Primitive Recursion.* Suppose $\alpha = *(\alpha_1,\alpha_2)$ where for some $u,v \in S^+$, $\alpha_1 \in PR(\Sigma)_{u,v}$ and $\alpha_2 \in PR(\Sigma)_{\tau u\, v,v}$. Then $c(\alpha)$ is defined by

$$c(\alpha) = *(c(\alpha_1),c(\alpha_2))$$ □

We now show c is well-defined in the following sense:

3.5.7 Lemma. *Let Σ, β, Φ, and $c = c(\beta,\Phi)$ be as above. For every $u,v \in S^+$,*

$$c: PR(\Sigma,\Phi)_{u,v} \longrightarrow PR(\Sigma)_{u,v}$$

That is, for every $\alpha \in PR(\Sigma,\Phi)_{u,v}$, $c(\alpha) \in PR(\Sigma)_{u,v}$.

Proof. Choose $\alpha \in PR(\Sigma,\Phi)_{u,v}$ for any $u,v \in S^+$. We show $c(\alpha) \in PR(\Sigma)_{u,v}$ by induction on the structural complexity of α as follows:

Basis Cases.

(i) *Constant Functions.* Suppose $\alpha = c^w$ for some $c \in (\Sigma,\Phi)_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$.

In this case $u = w$ and $v = s$, and we must show $c(\alpha) \in PR(\Sigma)_{w,s}$. However, $c(\alpha) = \alpha$ here, and since $c \in (\Sigma,\Phi)$, we have $c \in \Sigma$ since $c \neq \phi_i$ for any $i \in [1,m']$ (by definition of (Σ,Φ)). Thus $c(\alpha) = \alpha = c^w \in PR(\Sigma)_{w,s}$ as required.

(ii) *Algebraic Operations.* Suppose $\alpha = \sigma$ for some $\sigma \in (\Sigma,\Phi)_{w,s}$ for some $w \in S^+$ and $s \in S$.

In this case $u = w$ and $v = s$, and we must show $c(\alpha) \in PR(\Sigma)_{w,s}$. First suppose $\sigma \in \Sigma$. Then $\alpha = \sigma \in PR(\Sigma)_{w,s}$, but $c(\alpha) = \alpha$; thus $c(\alpha) \in PR(\Sigma)_{w,s}$ as required. Now suppose $\sigma = \phi_i$ for some $i \in [1,m']$. Since ϕ_i is adjoined to Σ_{u',ϕ_i} in forming $(\Sigma,\Phi) = (\Sigma,\Phi;u',v')$, we have $w = u'$ and $s = v'_i$; so we must show $c(\alpha) \in PR(\Sigma)_{w,s}$ where $w = u'$ and $s = v'_i$. However, $c(\alpha) = U_i^{v'} \circ \beta$, and $\beta \in PR(\Sigma)_{u',v'}$ by hypothesis; since $U_i^{v'} \in PR(\Sigma)$ with arity (v',v'_i) we have $c(\alpha) \in PR(\Sigma)$ with arity (u',v'_i) by clause (vi) of Definition 3.3.1.

(iii) *Projection Functions.* Suppose $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \leq i \leq n = |w|$.

In this case we have $u = w$ and $v = w_i$, and we must show $c(\alpha) \in PR(\Sigma)_{w,w_i}$. However $\alpha \in PR(\Sigma)_{w,w_i}$ by clause (iii) of Definition 3.3.1 and $c(\alpha) = \alpha$; thus $c(\alpha) \in PR(\Sigma)_{w,w_i}$ as required.

Induction Cases. Let $\alpha \in PR(\Sigma,\Phi)$ be some fixed (non-basis) scheme with the property that for every $\alpha_0 \in PR(\Sigma,\Phi)$ of less structural complexity than α, if α_0 is of arity (u_0,v_0) for some $u_0,v_0 \in S^+$, then $c(\alpha_0) \in PR(\Sigma)_{u_0,v_0}$.

We now show that $c(\alpha) \in PR(\Sigma)_{u,v}$ when $\alpha \in PR(\Sigma,\Phi)_{u,v}$ according to the four possible following cases:

(iv) *Definition-by-Cases.* Suppose $\alpha = DC(\beta',\alpha_1,\alpha_2)$. Then $\beta' \in PR(\Sigma)_{u,s}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$.

Since β',α_1, and α_2 are of less structural complexity than α, by the induction hypothesis applied to β',α_1, and α_2, we have, respectively: $c(\beta') \in PR(\Sigma)_{u,s}$; $c(\alpha_1) \in PR(\Sigma)_{u,v}$, and $c(\alpha_2) \in PR(\Sigma)_{u,v}$. Thus $c(\beta')$, $c(\alpha_1)$, and $c(\alpha_2)$ are of the appropriate arities to combine with the DC operator yielding

$$c(\alpha) = DC(c(\beta'),c(\alpha_1),c(\alpha_2)) \in PR(\Sigma)_{u,v}$$

as required.

(v) *Vectorisation.* Suppose $\alpha = <\alpha_1,\ldots,\alpha_m>$. Then $\alpha_i \in PR(\Sigma)_{u,v_i}$ for $i = 1,\ldots,m = |v|$.

Since α_i is of less structural complexity than α for $i = 1,\ldots,m$, by the induction hypothesis applied to α_i we have $c(\alpha_i) \in PR(\Sigma)_{u,v_i}$ for $i = 1,\ldots,m$. Thus $c(\alpha_1),\ldots,c(\alpha_m)$ are of the appropriate arities to combine in an instance of vectorisation yielding

$$c(\alpha) = <c(\alpha_1),\ldots,c(\alpha_m)> \in PR(\Sigma)_{u,v}$$

Cases (vi) *Composition* and (vii) *Primitive Recursion* are no more difficult than cases (iv) and (v) and we leave them as exercises. □

3.5.8 Lemma. *Let Σ be an S-sorted signature, and let (Σ,Φ) be a (u',v')-extension of Σ for some $u',v' \in S^+$. Also let A be a Σ-algebra, and let P be a performance measure for A which is based on clock C. Also let $\beta \in PR(\Sigma)_{u',v'}$ and let $c = c(\beta,\Phi)$ be the transformation defined above. If $B = (A, [\![\beta]\!]_A)$ then for every $u,v \in S^+$ and for every $\alpha \in PR(\Sigma)_{u,v}$,*

$$[\![c(\alpha)]\!]_A = [\![\alpha]\!]_B \tag{21}$$

Furthermore, if $Q = (P,\lambda)$ where $\lambda : A^{u'} \longrightarrow C^+$ is any function with $\lambda \approx \lambda_P(\beta))$, then

$$\lambda_P(c(\alpha)) \approx \lambda_Q(\alpha) \tag{22}$$

Proof. Choose $\alpha \in PR(\Sigma,\Phi)_{u,v}$ for any $u,v \in S^+$. To prove the lemma we must show that (21) and (22) hold for α. To show that (21) holds we must show

$$(\forall a \in A^u)\ (\ [\![c(\alpha)]\!]_A(a) = [\![\alpha]\!]_B(a)\ ) \tag{23}$$

Also, to show that (22) holds we must show that there exists constants $k_1,k_2 \geq 1$ such that

$$(\forall a \in A^u)\ (\ \lambda_P(c(\alpha))(a) \leq k_1 . \lambda_Q(\alpha)(a)\ ) \tag{24}$$

thus establishing $\lambda_P(c(\alpha)) = O(\lambda_Q(\alpha))$, and

$$(\forall a \in A^u)\ (\ \lambda_Q(\alpha)(a) \leq k_2 . \lambda_P(c(\alpha))(a)\ ) \tag{25}$$

establishing $\lambda_Q(\alpha) = O(\lambda_P(c(\alpha)))$.

Notice that since $\lambda \approx \lambda_P(\beta)$ by hypothesis, there are constants k'_1,k'_2 such that

$$(\forall a' \in A^{u'})\ (\ \lambda_P(\beta)(a') \leq k'_1 . \lambda(a')\ ) \tag{26}$$

and

$$(\forall a' \in A^{u'})\ (\ \lambda(a') \leq k'_2 . \lambda_P(\beta)(a')\ ) \tag{27}$$

We now prove (21) and (22) by induction on the structural complexity of α as follows:

Basis Cases.

(i) *Constant Functions.* Suppose $\alpha = c^w$ for some $c \in (\Sigma,\Phi)_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$.

To see that (21) holds for this α, we choose $a \in A^w$ and calculate as follows:

$$\begin{aligned}
[\![c(\alpha)]\!]_A(a) &= [\![c(c^w)]\!]_A(a) \\
&= [\![c^w]\!]_A(a) \\
&= c^A \\
&= c^{(A,f_A)} \\
&= [\![c^w]\!]_B(a) \\
&\text{(since } B = (A,f_A)) \\
&= [\![\alpha]\!]_B(a)
\end{aligned}$$

To see that (22) holds for $\alpha = c^w$ choose $a \in A^w$ and calculate as follows:

$$\begin{aligned}
\lambda_P(c(\alpha))(a) &= \lambda_P(c^w)(a) \\
&= c^P
\end{aligned}$$

$$= c^{(P,\lambda)}$$

(since $c \notin \Phi$)

$$= c^{Q}$$
$$= \lambda_Q(c^w)(a)$$
$$= \lambda_Q(\alpha)(a)$$

Since $\lambda_P(c(\alpha)) = \lambda_Q(\alpha)$, we immediately conclude $\lambda_P(c(\alpha)) \approx \lambda_Q(\alpha)$.

(ii) *Algebraic Operations.* Suppose $\alpha = \sigma$ for some $\sigma \in (\Sigma,\Phi)_{w,s}$ for some $w \in S^+$ and $s \in S$.

First notice that if $\sigma \notin \Phi$, that is, if $\sigma \in \Sigma$, then $c(\alpha) = \alpha$, and similar to case (i) above it is easy to show that $[\![c(\alpha)]\!]_A = [\![\alpha]\!]_B$ (establishing (21)) and that $\lambda_P(c(\alpha)) = \lambda_Q(\alpha)$ (establishing (22)).

Now suppose $\sigma = \phi_i$ for some $i \in [1,m']$. To show that (21) holds in this case, first observe that the functions $f_1,\ldots,f_{m'}$ defined by $f_i(a) = [\![c(\phi_i)]\!]_A(a)$ for each $a \in A^{u'}$ are the coordinates of $[\![\beta]\!]_A$. To see this explicitly, we must show that for every $a \in A^{u'}$,

$$[\![\beta]\!]_A(a) = (f_1(a),\ldots,f_{m'}(a))$$

Choose $a \in A^{u'}$. Then $[\![\beta]\!]_A(a)$ is a vector $b \in A^{v'}$ with $b = (b_1,\ldots,b_{m'})$; that is

$$[\![\beta]\!]_A(a) = (b_1,\ldots,b_{m'}) \tag{28}$$

when $[\![\beta]\!]_A(a) = (b_1,\ldots,b_m)$. Now, for $i = 1,\ldots,m'$, we have

$$f_i(a) = [\![c(\phi_i)]\!]_A(a)$$
$$= [\![U_i^{v'} \circ \beta]\!]_A(a)$$
$$= [\![U_i^{v'}]\!]_A([\![\beta]\!]_A(a))$$
$$= b_i$$

(using (28)).

Thus,

$$(f_1(a),\ldots,f_{m'}(a)) = (b_1,\ldots,b_{m'})$$
$$= [\![\beta]\!]_A(a)$$

That is, $f_1,\ldots,f_{m'}$ are the coordinates of $[\![\beta]\!]_A$ as claimed. Since ϕ_i^B is defined to be the ith coordinate of $[\![\beta]\!]_A$ we now have for each $a \in A^{u'}$,

$$[\![c(\alpha)]\!]_A(a) = [\![c(\phi_i)]\!]_A(a)$$
$$= f_i(a)$$
$$= \phi_i^B(a)$$
$$= [\![\phi_i]\!]_B(a)$$
$$= [\![\alpha]\!]_B(a)$$

To show that (22) holds for $\alpha = \phi_i$, we show (24) and (25) hold for α for some choice of k_1 and k_2. For (24) take with $k_1 = 2k_1'$. Then for any $a \in A^{u'}$,

$$\lambda_P(c(\alpha))(a) = \lambda_P(c(\phi_i))(a)$$
$$= \lambda_P(U_i^{v'} \circ \beta)(a)$$
$$= \lambda_P(U_i^{v'})([\![\beta]\!]_A(a)) + \lambda_P(\beta)(a)$$

$$= 1 + \lambda_P(\beta)(a)$$
$$\leq 1 + k_1' . \lambda(a)$$

(by (26))

$$\leq 2 . k_1' . \lambda(a)$$

(since the codomain of λ is $C^+ = \{ 1,2,3,... \}$)

$$= k_1 . \phi_i^Q(a)$$
$$= k_1 . \lambda_Q(\phi_i)(a)$$
$$= k_1 . \lambda_Q(\alpha)(a)$$

It is equally easy to prove that (25) holds for the choice $k_2 = 2k_2'$; this we leave as an exercise.

(iii) *Projection Functions.* Suppose $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \leq i \leq n = |w|$.

Since $c(\alpha) = \alpha$ in this case and α involves no symbols of Φ it is easy to see that we will have $[\![c(\alpha)]\!]_A = [\![\alpha]\!]_B$ and $\lambda_P(c(\alpha)) = \lambda_Q(\alpha)$; the details we leave as exercises.

Induction. Let $\alpha \in PR(\Sigma,\Phi)$ be some fixed (non-basis) scheme with the property that for every $\alpha_0 \in PR(\Sigma,\Phi)$ of less structural complexity than α, if α_0 is of arity (u_0,v_0) for some $u_0,v_0 \in S^+$, then

$$[\![c(\alpha)]\!]_A(a) = [\![\alpha]\!]_B \tag{29}$$

and

$$\lambda_P(c(\alpha)) \approx \lambda_Q(\alpha) \tag{30}$$

We must now show that (21) and (22) hold in the four induction cases namely (iv) Definition-by-Cases, (v) Vectorisation, (vi) Composition, and (vii) Primitive Recursion. We will do case (vi) only. The remaining cases are no more difficult than this and we leave them as exercises.

(vi) *Composition.* Suppose $\alpha = \alpha_2 \circ \alpha_1$. Then for some $w \in S^+$, $\alpha_1 \in PR(\Sigma)_{u,w}$ and $\alpha_2 \in PR(\Sigma)_{w,v}$ since $\alpha \in PR(\Sigma)_{u,v}$ by hypothesis.

To see that (21) holds for α, first observe that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (29) applied to α_i for $i = 1,2$ we have

$$[\![c(\alpha_i)]\!]_A = [\![\alpha_i]\!]_B$$

That is,

$$(\forall a \in A^u)\ (\ [\![c(\alpha_1)]\!]_A(a) = [\![\alpha_1]\!]_B(a)\ ) \tag{31}$$

and

$$(\forall a \in A^w)\ (\ [\![c(\alpha_2)]\!]_A(a) = [\![\alpha_2]\!]_B(a)\ ) \tag{32}$$

Now choose $a \in A^u$ and calculate as follows:

$$[\![c(\alpha)]\!]_A(a) = [\![c(\alpha_2 \circ \alpha_1)]\!]_A(a)$$
$$= [\![c(\alpha_2) \circ c(\alpha_1)]\!]_A(a)$$
$$= [\![c(\alpha_2)]\!]_A([\![c(\alpha_1)]\!]_A(a))$$
$$= [\![c(\alpha_2)]\!]_A([\![\alpha_1]\!]_B(a))$$

(by (31))

$$= [\![\alpha_2]\!]_B([\![\alpha_1]\!]_B(a))$$

(by (32))

$$= [\![\alpha_2 \circ \alpha_1]\!]_B(a)$$
$$= [\![\alpha]\!]_B(a)$$

Thus (21) holds for α as claimed.

To see that (22) holds for α, first notice that again since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (30) applied to α_i for $i = 1,2$ we have

$$\lambda_P(c(\alpha_i)) \approx \lambda_Q(\alpha_i) \tag{33}$$

Thus for some constants $k_{\alpha_1}, k_{\alpha_2} \geq 1$ we have

$$(\forall a \in A^u)\ (\lambda_P(c(\alpha_1))(a) \leq k_{\alpha_1} . \lambda_Q(\alpha_1)(a)) \tag{34}$$

and

$$(\forall a \in A^w)\ (\lambda_P(c(\alpha_2))(a) \leq k_{\alpha_2} . \lambda_Q(\alpha_2)(a)) \tag{35}$$

To see that (25) holds for some choice of k_1, take $k_1 = \max\{k_{\alpha_1}, k_{\alpha_2}\}$. Now choose $a \in A^u$ and calculate as follows:

$$\lambda_P(c(\alpha))(a) = \lambda_P(c(\alpha_2 \circ \alpha_1))(a)$$
$$= \lambda_P(c(\alpha_2) \circ c(\alpha_1))(a)$$
$$= \lambda_P(c(\alpha_1))(a) + \lambda_P(c(\alpha_2))([\![c(\alpha_1)]\!]_A(a))$$
$$= \lambda_P(c(\alpha_1))(a) + \lambda_P(c(\alpha_1))([\![\alpha_1]\!]_B(a))$$

(by (31))

$$\leq k_{\alpha_1} . \lambda_Q(\alpha_1)(a) + \lambda_P(c(\alpha_2))([\![\alpha_1]\!]_B(a))$$

(by (34))

$$\leq k_{\alpha_1} . \lambda_Q(\alpha_1)(a) + k_{\alpha_2} . \lambda_Q(\alpha_2)([\![\alpha_1]\!]_B(a))$$

(by (35))

$$\leq \max\{k_{\alpha_1}, k_{\alpha_2}\} . (\lambda_Q(\alpha_1)(a) + \lambda_Q(\alpha_2)([\![\alpha_1]\!]_B(a)))$$

(since $m . p + n . q \leq \max\{m,n\} . (p+q)$ for any numbers n, m, p, and q)

$$= k_1 . \lambda_Q(\alpha_2 \circ \alpha_1)(a)$$
$$= k_1 . \lambda_Q(\alpha)(a)$$

Thus (25) holds for α and this choice of k_1 as claimed.

It is equally easy to show from (33) that (26) holds for α and some choice of k_2, and thus (22) holds for α. □

Isomorphism Invariance. In this final part of the current section we will prove that primitive recursive computability is an isomorphism invariant; that is, given two isomorphic algebras A and B, what you can compute with one is no more than what you can compute with the other (up to isomorphism).

3.5.9 Lemma. *Let A and B be S-sorted Σ-algebras and let $h : A \longrightarrow B$ be a Σ-homomorphism. For each $u, v \in S^+$, and for every $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, the following diagram commutes for every $a \in A^u$:*

$$\begin{array}{ccc} A^u & \xrightarrow{[\![\alpha]\!]_A} & A^v \\ \downarrow{\scriptstyle h_u} & & \downarrow{\scriptstyle h_v} \\ B^u & \xrightarrow[[\![\alpha]\!]_B]{} & B^v \end{array}$$

That is,

$$(\forall a \in A^u)\ \ (\, h_v([\![\alpha]\!]_A(a)) = [\![\alpha]\!]_B(h_u(a))\,) \tag{36}$$

Proof. Choose $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ for some $u, v \in S^+$. We prove (36) uniformly in u and v by induction on the structural complexity of α as follows:

Basis Schema.

(i) *Constant Functions.* Suppose $\alpha = c^w$ for some $w \in S^+$ and for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$.

Choose $a \in A^w$. Then by the definition of $[\![c^w]\!]_A$ for any Σ-algebra A, and by the the homomorphism property of h, we calculate

$$\begin{aligned} h_s([\![\alpha]\!]_A(a)) &= h_s([\![c^w]\!]_A(a)) \\ &= h_s(c^A) \\ &= c^B \\ &= [\![c^w]\!]_B(b) \end{aligned}$$

for any $b \in B^w$. To show (36) holds in this case, take $b = h_w(a)$. Then,

$$\begin{aligned} h_s([\![\alpha]\!]_A(a)) &= [\![c^w]\!]_B(h_w(a)) \\ &= [\![\alpha]\!]_B(h_w(a)) \end{aligned}$$

Thus (36) holds for $\alpha = c^w$.

(ii) *Algebraic Operations.* Suppose $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and $s \in S$. Then,

Choose $a \in A^w$. Then by the definition of $[\![\sigma]\!]_A$ for any Σ-algebra A, and by the the homomorphism property of h, we calculate

$$\begin{aligned} h_s([\![\alpha]\!]_A(a)) &= h_s([\![\sigma]\!]_A(a)) \\ &= h_s(\sigma^A(a)) \\ &= \sigma^B(h_w(a)) \\ &= [\![\sigma]\!]_B(h_w(a)) \\ &= [\![\alpha]\!]_B(h_w(a)) \end{aligned}$$

Thus (36) holds for $\alpha = \sigma$.

(iii) *Projection Functions.* Suppose $\alpha = \mathrm{U}_i^w$ for some $w \in S^+$ and some i with $1 \le i \le n = |w|$.

Choose $a=(a_1,\ldots,a_n)\in A^w$. Then by the definition of $[\![U_i^w]\!]_A$ for any Σ-algebra A, we calculate

$$\begin{aligned} h_{w_i}([\![\alpha]\!]_A(a)) &= h_{w_i}([\![U_i^w]\!]_A(a)) \\ &= h_{w_i}(a_i) \end{aligned}$$

Now, $h_{w_i}(a_i)$ is just the ith coordinate of the vector $(h_{w_1}(a_1),\ldots,h_{w_n}(a_n))=h_w(a_1,\ldots,a_n)\in B^w$. Thus

$$\begin{aligned} h_{w_i}([\![\alpha]\!]_A(a)) &= [\![U_i^w]\!]_B(h_{w_1}(a_1),\ldots,h_{w_n}(a_n)) \\ &= [\![U_i^w]\!]_B(h_w(a_1,\ldots,a_n)) \\ &= [\![\alpha]\!]_B(h_w(a)) \end{aligned}$$

Thus (36) holds for $\alpha=U_i^w$.

Induction Cases. Let $\alpha\in \mathrm{PR}(\Sigma)_{u,v}$ for some $u,v\in S^+$ be some fixed non-basis scheme with the property that for each $u_0,v_0\in S^+$, if $\alpha_0\in \mathrm{PR}(\Sigma)_{u_0,v_0}$ is any scheme of less structural complexity than α, then

$$(\forall a\in A^{u_0})\ (\ h_{v_0}([\![\alpha]\!]_A(a))=[\![\alpha]\!]_B(h_{u_0}(a))\ ) \tag{37}$$

We now show (36) holds for α according to the four following possible cases:

(iv) *Definition-by-Cases.* Suppose $\alpha=\mathrm{DC}(\beta,\alpha_1,\alpha_2)$. Then $\beta\in \mathrm{PR}(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2\in \mathrm{PR}(\Sigma)_{u,v}$ since $\alpha\in \mathrm{PR}(\Sigma)_{u,v}$ by hypothesis.

Since β, α_1, and α_2 are all of less structural complexity than α, by the induction hypothesis (37) applied to β, α_1, and α_2 respectively, we have:

$$(\forall a\in A^u)\ (\ h_B([\![\beta]\!]_A(a))=[\![\beta]\!]_B(h_u(a))\ ) \tag{38}$$

$$(\forall a\in A^u)\ (\ h_v([\![\alpha_1]\!]_A(a))=[\![\alpha_1]\!]_B(h_u(a))\ ) \tag{39}$$

$$(\forall a\in A^u)\ (\ h_v([\![\alpha_2]\!]_A(a))=[\![\alpha_2]\!]_B(h_u(a))\ ) \tag{40}$$

Now notice that if $[\![\beta]\!]_A(a)=tt=tt^A$, then $h_B([\![\beta]\!]_A(a))=h_B(tt^A)=tt^B$ since h is a homomorphism. However, from (38) we have $h_B([\![\beta]\!]_A(a))=[\![\beta]\!]_B(h_u(a))$, and thus

$$[\![\beta]\!]_A(a)=tt^A \implies [\![\beta]\!]_B(h_u(a))=tt^B \tag{41}$$

Similarly, we can show

$$[\![\beta]\!]_A(a)=ff^A \implies [\![\beta]\!]_B(h_u(a))=ff^B \tag{42}$$

To see that (36) holds for α, choose $a\in A^u$. Then by the definition of $[\![\alpha]\!]_A$ for any Σ-algebra A, we calculate

$$\begin{aligned} h_v([\![\alpha]\!]_A(a)) &= h_v([\![\mathrm{DC}(\beta,\alpha_1,\alpha_2)]\!]_A(a)) \\ &= \begin{cases} h_v([\![\alpha_1]\!]_A(a)) & \text{if } [\![\beta]\!]_A(a)=tt^A \\ h_v([\![\alpha_2]\!]_A(a)) & \text{if } [\![\beta]\!]_A(a)=ff^A \end{cases} \\ &= \begin{cases} [\![\alpha_1]\!]_B(h_u(a)) & \text{if } [\![\beta]\!]_A(a)=tt^A \\ [\![\alpha_2]\!]_B(h_u(a)) & \text{if } [\![\beta]\!]_A(a)=ff^A \end{cases} \end{aligned}$$

(by (39) and (40))

$$= \begin{cases} [\![\alpha_1]\!]_B(h_u(a)) & \text{if } [\![\beta]\!]_B(h_u(a))=tt^B \\ [\![\alpha_2]\!]_B(h_u(a)) & \text{if } [\![\beta]\!]_B(h_u(a))=ff^B \end{cases}$$

(by (41) and (42))

$$= [\![DC(\beta,\alpha_1,\alpha_2)]\!]_B(h_u(a))$$

$$= [\![\alpha]\!]_B(h_u(a))$$

Thus (36) holds for $\alpha = DC(\beta,\alpha_1,\alpha_2)$.

The remaining cases are no more difficult than this, and we leave them as an exercises. □

Notice that the premise of the preceding lemma is that there is a homomorphism h from A to B, and the lemma concludes that for each $f_A \in PR(A)$ there exists some $f_B \in PR(B)$ such that $h \circ f_A = f_B \circ h$. By symmetry, we can argue that if $h' : B \longrightarrow A$ is also a Σ-homomorphism then for every $f_B \in PR(B)$ there exists $f_A \in PR(A)$ such that $h' \circ f_B = f_A \circ h'$. Since the existence of homomorphisms h and h' together implies the existence of a Σ-isomorphism ϕ (see the lemma in Section 3.1.7), we conclude that primitive recursive computability (or definability) is an isomorphism invariant as previously claimed.

3.6 SOURCES.

Our treatment of data types as algebras (in Section 3.1) is based on the work of ADJ: our notation and development of the subject closely follows Goguen, Thatcher, and Wagner[1978], Wagner[1981], or Meseguer and Goguen[1985]; the concept of an abstract data type as presented here (Section 3.1.7) is from Meseguer and Goguen[1985]. Other useful references are the bibliography Kutzler and Lichtenberger[1983] and the text-book Ehrig and Mahr[1985]. The concept of a clock as an abstract data type is taken from Harman and Tucker[1987]. The idea of including streams in an algebra is due to joint work between N. A. Harman, J. V. Tucker, and myself.

The formal semantics of performance of computation on abstract data types is first discussed in Asveld and Tucker[1982] (using norms on data). A more general treatment of the cost of computation on abstract structures, equivalent to our performance measures (Section 3.2), can be found in Nielson[1984] where this is used to study Hoare-style proof systems for performance. The concept of a performance measure was obtained independently by myself.

Primitive recursive functions first appear defined over the natural numbers in Dedekind[1888]. Simultaneous primitive recursive functions (over the natural numbers) were first studied by R. Péter: see Péter[1967]. A form of (non-simultaneous) primitive recursion over a single-sorted abstract structure first appears in Engler[1968]. Simultaneous primitive recursive functions over an abstract structure are due to Tucker and Zucker[1987] (work of 1979).

The revised form (syntax, semantics, performance) of simultaneous primitive recursive functions given here (Section 3.3), and their application to the definition of synchronous concurrent algorithms is new and the result of joint work by J. V. Tucker and myself, as are the results in Section 3.5; the precise formulation and proof of these results is due to myself however.

3.6.1 Excursus on Computability in an Abstract Setting.

The simultaneous primitive recursive functions may be extended to other interesting classes of computable functions on abstract structures. For example, Kleene's *least number operator* may be added to complete the obvious generalisation of Kleene's definition of the partial recursive functions on $\mathbb{N}$. More interestingly, simultaneous primitive recursion can be replaced with simultaneous *course-of-values* recursion which together with the least number operator defines a stronger generalisation. Both of these generalisations are made in Tucker and Zucker[1987] and it is shown that the first (called the *inductively definable functions*) is equivalent to while-programs and the second (called the *inductively definable functions*) is equivalent to while-programs with arrays on a standard structure. This means that simultaneous course-of-values recursion is stronger than simultaneous primitive recursion and provides the 'correct' generalisation of partial recursive functions from $\mathbb{N}$ to an abstract structure A: many of the basic results of recursive function theory (such as the existence of universal functions, the halting problem etc.) generalise to this class of functions. In Tucker and Zucker[1987] there is a thorough survey of disparate methods of defining computable functions on an abstract data type including the *A-register machines* of Friedman, the *axiomatic techniques* of Moschovakis-Fenstad, Platek's *inductive definability*, Normann's *set recursion*, and Herbrand-Gödel-Kleene *equational definability*. They also discuss connections with the literature on *program schemes*. See also Greibach[1975] and Shepherdson[1986].

CHAPTER 4
FORMAL VERIFICATION

In this chapter we will verify the OE and EOE sorters of Chapter 2. The verifications are built on PR-specifications of algorithms, and thus with this chapter we begin to test out the PR formalism.

4.1 CORRECTNESS THEOREMS.

We verify OE and EOE by proving the following two theorems:

4.1.1 Theorem. *(OE sorts on streams.) For each* $\underline{a} : T \longrightarrow D^n$ *and* $x \in D^n$, *and for each* $t \in T$,

$$G_{OE}(\underline{a},x)(t) = \begin{cases} u & \text{if } \neg R(t) \\ sort(\underline{a}(\delta(t))) & \text{if } R(t) \end{cases}$$

where $R(t) \Leftrightarrow t \neq 0 \wedge t \bmod (n+1) = 0$, *and* $\delta : T \longrightarrow T$ *is defined by* $\delta(t) = t-n-1$ *for each* $t \in T$.

4.1.2 Theorem. *(EOE sorts on streams.) For each* $\underline{a} : T \longrightarrow D^n$ *and* $y \in D^{n(n+1)}$, *and for each* $t \in T$,

$$G_{EOE}(\underline{a},y)(t) = \begin{cases} u & \text{if } \neg R(t) \\ sort(\underline{a}(\delta(t))) & \text{if } R(t) \end{cases}$$

where $R(t) \Leftrightarrow t \geq n+1$, *and* $\delta : T \longrightarrow T$ *is defined by* $\delta(t) = t-n-1$ *for each* $t \in T$.

Our strategy for proving Theorem 4.1.1 is based on the following two facts:

(i) OE sorts every $a \in D^n$ when it is presented to the OE network as a fixed input.

(ii) The behaviour of OE on streams can be related to its behaviour on a fixed input.

In a similar way Theorem 4.1.2 follows from (i) and the following fact:

(iii) The behaviour of EOE on streams can be related to the behaviour of OE on a fixed input.

Below, after some preliminary notes, we will prove Theorems 4.1.1 and 4.1.2 using Theorems 4.1.3, 4.1.4, and 4.1.5, that make precise the statements (i), (ii), and (iii) respectively. The remainder of the chapter is devoted to proving these latter theorems.

Preliminary Notes. Throughout the proofs we take V_{OE} and V_{EOE} to be as defined in Examples 2.4.2. Also, unless otherwise stated, we use 'V_i' and '$V_{i,j}$' to denote typical coordinates of V_{OE} and V_{EOE} respectively.

Recall from Section 2.4.3 that for any synchronous network N (in particular OE and EOE), the functions F_N and G_N are *automatically* defined once we have defined the network's value function V_N. In particular, recall how F_N is defined by vectorising those coordinates of V_N that describe the output sent to the network's sinks.

We have said that the verification of OE and EOE on streams is a consequence of the fact that OE sorts a fixed input. Throughout this chapter '*fix*' denotes the function $fix : D^n \longrightarrow [T \longrightarrow D^n]$ defined by $fix(a)(t) = a$ for each $a \in D^n$ and $t \in T$.

Proof of Theorem 4.1.1. Theorem 4.1.1 is an easy consequence of the following two theorems:

4.1.3 Theorem. *(OE sorts any fixed input.) For each $a \in D^n$ and $x \in D^n$,*

$$V_{OE}(n+1, fix(a), x) = sort(a) \tag{1}$$

4.1.4 Theorem. *(The behaviour of OE on streams can be related to its behaviour on a fixed input.) For each $t \in T$, $\underline{a} : T \longrightarrow D^n$, and $x \in D^n$,*

$$V_{OE}(t, \underline{a}, x) = \begin{cases} x & \text{if } t = 0 \\ V_{OE}(t', \theta(t, \underline{a}), x) & \text{if } t > 0 \end{cases} \tag{2}$$

where $t' = (t-1) \bmod (n+1)+1$ and $\theta : T \times [T \longrightarrow D^n] \longrightarrow [T \longrightarrow D^n]$ is defined by $\theta(t, \underline{a}) = fix(\underline{a}(t-t'))$ for each $t \in T$ and $\underline{a} : T \longrightarrow D^n$. □

We will return to the proofs of Theorems 4.1.3 and 4.1.4 later: let us prove Theorem 4.1.1:

Given our remarks following Definition 2.2.3 concerning unspecified values, to prove Theorem 4.1.1 we must show that for each $\underline{a} : T \longrightarrow D^n$, $x \in D^n$, and $t \in T$,

$$R(t) \implies G_{OE}(\underline{a}, x)(t) = sort(\underline{a}(\delta(t))) \tag{3}$$

where R and δ are as given in the statement of Theorem 4.1.1. Choose $\underline{a} : T \longrightarrow D^n$, $x \in D^n$, and $t \in T$, and suppose $R(t)$ holds. Then by definition of R we can assume that t is of the form $t = (k+1)(n+1)$ for some $k \in \mathbb{N}$, and in this case it is easy to show $t' = n+1$ and $t - t' = k(n+1)$ and so

$$\begin{aligned} R(t) \implies \quad V_{OE}(t, \underline{a}, x) &= V_{OE}((k+1)(n+1), \underline{a}, x) \\ &= V_{OE}(n+1, \theta((k+1)(n+1), \underline{a}), x) \end{aligned}$$

(by Theorem 4.1.4)

$$= V_{OE}(n+1, fix(\underline{a}(k(n+1))), x)$$

(by definition of θ)

$$= sort(\underline{a}(k(n+1)))$$

(by Theorem 4.1.3)

$$= sort(\underline{a}(\delta(t)))$$

(by definition of δ).

We have shown that $R(t) \implies V_{OE}(t, \underline{a}, x) = sort(\underline{a}(\delta(t)))$; it is also easy to check that $F_{OE} = V_{OE}$ (since the i th module of OE is connected to the i th source for $i = 1, \ldots, n$), and thus

$$R(t) \implies F_{OE}(t, \underline{a}, x) = sort(\underline{a}(\delta(t))) \tag{4}$$

But G_{OE} is defined by $G_{OE}(\underline{a}, x)(t) = F_{OE}(t, \underline{a}, x)$ and so from (4) we have

$$R(t) \implies \quad G_{OE}(\underline{a}, x)(t) = F_{OE}(t, \underline{a}, x) = sort(\underline{a}(\delta(t)))$$

Thus (3) holds, proving Theorem 4.1.1. □

Proof of Theorem 4.1.2. Theorem 4.1.2 is an easy consequence of Theorem 4.1.3 and the following result:

4.1.5 Theorem. *(The behaviour of EOE on streams can be described in terms of the behaviour of OE on a fixed input.) For each* $t \in T$, $\underline{a} : T \longrightarrow D^n$, $x \in D^n$ *and* $y \in D^{n(n+1)}$, *and for* $i = 1,\ldots,n$ *and for* $j = 0,\ldots,n$,

$$t \geq j+1 \implies V_{i,j}(t,\underline{a},y) = V_i(j+1,fix(\underline{a}(t-j-1)),x) \tag{5}$$

We will return to the proof of Theorem 4.1.5 later: let us prove Theorem 4.1.2:

Again given our remarks following Definition 2.2.3 concerning unspecified values, to prove Theorem 4.1.2 we must show that for each $\underline{a} : T \longrightarrow D^n$, $y \in D^{n(n+1)}$, and $t \in T$,

$$R(t) \implies G_{OE}(\underline{a},y)(t) = sort(\underline{a}(\delta(t))) \tag{6}$$

where R and δ are as given in the statement of Theorem 4.1.2. Now choose $\underline{a} : T \longrightarrow D^n$, $y \in D^{n(n+1)}$, and $t \in T$ with $t \geq n+1$. Then $R(t)$ holds and from Theorem 4.1.5 we have for $i = 1,\ldots,n$,

$$\begin{aligned} V_{i,n}(t,\underline{a},y) &= V_i(n+1,fix(\underline{a}(t-n-1)),x) \\ &= V_i(n+1,fix(\underline{a}(\delta(t))),x) \end{aligned} \tag{7}$$

Thus, by definition of F_{EOE} we have

$$\begin{aligned} R(t) \implies \quad F_{EOE}(t,\underline{a},y) &= (V_{1,n}(n+1,\underline{a},y),\ldots,V_{n,n}(n+1,\underline{a},y)) \\ &= (V_1(n+1,fix(\underline{a}(\delta(t))),x),\ldots,V_n(n+1,fix(\underline{a}(\delta(t))),x)) \end{aligned}$$

(by (7) with $i = 1,\ldots,n$)

$$= V_{OE}(n+1,fix(\underline{a}(\delta(t))),x)$$

(by definition of V_{EOE})

$$= sort(\underline{a}(\delta(t)))$$

(by Theorem 4.1.3).

But G_{EOE} is defined by $G_{EOE}(\underline{a},y)(t) = F_{EOE}(t,\underline{a},y)$ and so

$$R(t) \implies \quad G_{EOE}(\underline{a},y)(t) = F_{EOE}(t,\underline{a},y) = sort(\underline{a}(\delta(t)))$$

Thus (6) holds, proving Theorem 4.1.2. □

It remains to prove Theorems 4.1.3, 4.1.4, and 4.1.5.

Proof of Theorem 4.1.3. Theorem 4.1.3 is a direct consequence of Theorem 4.1.5 and the following:

4.1.6 Theorem. *(EOE sorts any fixed input.) For each* $a \in D^n$ *and* $y \in D^{n(n+1)}$,

$$F_{EOE}(n+1,fix(a),y) = sort(a)$$

We will return to the proof of Theorem 4.1.6 later: let us prove Theorem 4.1.3:

To begin with, consider (5) with $t = n+1$, $j = n$, and $\underline{a} = fix(a)$: Theorem 4.1.5 yields

$$V_{i,n}(n+1,fix(a),y) = V_i(n+1,fix(fix(a)(0)),x)$$

for $i = 1,\ldots,n$, for any $x \in D^n$ and $y \in D^{n(n+1)}$. However, $fix(a)(0) = a$, and so for $i = 1,\ldots,n$ we have

$$V_{i,n}(n+1,fix(a),y) = V_i(n+1,fix(a),x) \tag{8}$$

for any $x \in D^n$ and $y \in D^{n(n+1)}$.

Now, by definition of V_{OE} we have

$$V_{OE}(n+1,fix(a),x) = (V_1(n+1,fix(a),x),\ldots,V_n(n+1,fix(a),x))$$
$$= (V_{1,n}(n+1,fix(a),y),\ldots,V_{n,n}(n+1,fix(a),y))$$

(by (8) with $i = 1,\ldots,n$)

$$= F_{EOE}(n+1,fix(a),y)$$

(by definition of F_{EOE})

$$= sort(a)$$

(by Theorem 4.1.6).

Thus (1) holds, proving Theorem 4.1.3. □

It remains to prove Theorems 4.1.4, 4.1.5, and 4.1.6.

Proof of Theorem 4.1.4. Choose $t \in T$, $\underline{a} : T \longrightarrow D^n$, and $x = (x_1,\ldots,x_n) \in D^n$. We show (2) holds by induction on t as follows:

Basis. If $t = 0$ then by definition of V_{OE} we have

$$V_{OE}(0,\underline{a},x) = (V_1(0,\underline{a},x),\ldots,V_n(0,\underline{a},x))$$
$$= (x_1,\ldots,x_n)$$
$$= x$$

Thus (2) holds for $t = 0$.

Induction. Suppose that for some fixed $l \in \mathbf{N}$ that we have shown for $t = 0,\ldots,l$,

$$V_{OE}(t,\underline{a},x) = \begin{cases} x & \text{if } t = 0 \\ V_{OE}(t',\theta(t,\underline{a}),x) & \text{if } t > 0 \end{cases} \tag{9}$$

Then taking the equality expressed in (9) coordinatewise with $t = l$, we have

$$V_i(l,\underline{a},x) = \begin{cases} x_i & \text{if } l = 0 \\ V_i(l',\theta(l,\underline{a}),x) & \text{if } l > 0 \end{cases} \tag{10}$$

for $i = 1,\ldots,n$. We will now show that (2) holds for $t = l+1$ according to the two cases:

Case 1: $l = k(n+1)$ for some $k \in \mathbb{N}$, and
Case 2: $l \neq k(n+1)$ for any $k \in \mathbb{N}$.

Case 1. Suppose $l = k(n+1)$ for some $k \in \mathbb{N}$. Then it is easy to show that $l' = n+1$ and $(l+1)' = 1$, and so $(l+1)-(l+1)' = l$. Thus

$$\theta(l+1,\underline{a}) = fix(\underline{a}(l+1-(l+1)'))$$
$$= fix(\underline{a}(l))$$

Thus for any $i \in [1,n]$ we have

$$V_i((l+1)',\theta(l+1,\underline{a}),x) = V_i(1,fix(\underline{a}(l)),x)$$
$$= (fix(\underline{a}(l)))_i(0)$$

(by definition of V_i)

$$= (fix(\underline{a}(l))(0))_i$$
$$= \underline{a}_i(l)$$
$$= V_i(l+1,\underline{a},x)$$

(by definition of V_i, since $l \ mod \ (n+1) = 0$ by hypothesis).

Thus (2) holds for $t = l+1$ when l is of the form $l = k(n+1)$.

Case 2. Suppose $l \neq k(n+1)$ for any $k \in \mathbf{N}$. Then it must be that l is of the form $l = k(n+1)+r$ for some $k \in \mathbf{N}$ and some $r \in \mathbf{N}$ with $0 < r < n+1$, and in this case it is easy to show that $l' = r$ and $(l+1)' = r+1$, and so $(l+1)-(l+1)' = l-r$. Thus for any $i \in [1,n]$ we have

$$V_i((l+1)', \theta(l+1,\underline{a}), x) = V_i(r+1, \theta(l+1,\underline{a}), x)$$

However, it is easy to prove $\theta(l+1,\underline{a}) = \theta(l,\underline{a})$ and so

$$V_i((l+1)', \theta(l+1,\underline{a}), x) = V_i(r+1, \theta(l,\underline{a}), x) \tag{11}$$

There are now four subcases to consider (which originate from the four possibilities for the value $V_i(l+1,\underline{a},x)$ in the definition of V_{OE}); these are:

Case 2.1: $i = 1$;
Case 2.2: i odd, $\neq 1$;
Case 2.3: i even, $\neq n$, and
Case 2.4: $i = n$.

We will consider Case 2.2 only.

Case 2.2. Suppose i is odd, $\neq 1$. Then from (11) and the definition of V_i we have

$$V_i((l+1)', \theta(l+1,\underline{a}), x) = V_i(r+1, \theta(l,\underline{a}), x)$$
$$= \begin{cases} \min\{ V_i(r, \theta(l,\underline{a}), x), V_{i+1}(r, \theta(l,\underline{a}), x) \} & \text{if } r \textit{ odd} \\ \max\{ V_{i-1}(r, \theta(l,\underline{a}), x), V_i(r, \theta(l,\underline{a}), x) \} & \text{if } r \textit{ even} \end{cases} \tag{12}$$

(since $0 < r \leq n$ so $r \textit{ mod } (n+1) = r \neq 0$).

However, also by definition of V_i we have

$$V_i(l+1,\underline{a},x) = \begin{cases} \underline{a}_i(l) & \text{if } l \textit{ mod } (n+1) = 0 \\ \min\{ V_i(l,\underline{a},x), V_{i+1}(l,\underline{a},x) \} & \text{if } l \textit{ mod } (n+1) \textit{ odd} \\ \max\{ V_{i-1}(l,\underline{a},x), V_i(l,\underline{a},x) \} & \text{if } l \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

$$= \begin{cases} \min\{ V_i(l,\underline{a},x), V_{i+1}(l,\underline{a},x) \} & \text{if } r \textit{ odd} \\ \max\{ V_{i-1}(l,\underline{a},x), V_i(l,\underline{a},x) \} & \text{if } r \textit{ even} \end{cases}$$

(since $l \textit{ mod } (n+1) = r \neq 0$)

$$= \begin{cases} \min\{ V_i(l', \theta(l,\underline{a}), x), V_{i+1}(l', \theta(l,\underline{a}), x) \} & \text{if } r \textit{ odd} \\ \max\{ V_{i-1}(l', \theta(l,\underline{a}), x), V_i(l', \theta(l,\underline{a}), x) \} & \text{if } r \textit{ even} \end{cases}$$

(by the induction hypothesis (10), and the fact that $l \neq 0$)

$$= \begin{cases} \min\{ V_i(r, \theta(l,\underline{a}), x), V_{i+1}(r, \theta(l,\underline{a}), x) \} & \text{if } r \textit{ odd} \\ \max\{ V_{i-1}(r, \theta(l,\underline{a}), x), V_i(r, \theta(l,\underline{a}), x) \} & \text{if } r \textit{ even} \end{cases}$$

(since $l' = r$)

$$= V_i((l+1)',\theta(l+1,\underline{a}),x)$$

(from (12)).

We have now shown that

$$V_i(l+1,\underline{a},x) = V_i((l+1)',\theta(l+1,\underline{a}),x) \tag{13}$$

in the case that i is odd and $\neq 1$. It is equally easy to show that (13) holds in the three remaining cases (namely Cases 2.1, 2.3, and 2.4 which we leave as exercises), and so we may conclude that (13) holds for $i = 1,\ldots,n$. Thus

$$\begin{aligned} V_{OE}(l+1,\underline{a},x) &= (V_1(l+1,\underline{a},x),\ldots,V_n(l+1,\underline{a},x)) \\ &= (V_1((l+1)',\theta(l+1,\underline{a}),x),\ldots,V_n((l+1)',\theta(l+1,\underline{a}),x)) \end{aligned}$$

(by (13) with $i = 1,\ldots,n$)

$$= \begin{cases} x & \text{if } l+1=0 \\ V_{OE}((l+1)',\theta(l+1,\underline{a}),x) & \text{if } l+1>0 \end{cases}$$

$$= V_{OE}((l+1)',\theta(l+1,\underline{a}),x)$$

(since $l+1 \neq 0$).

Thus (2) holds for $t = l+1$ and so by the principle of mathematical induction (2) holds for all $t \in T$ proving Theorem 4.1.4. □

Proof of Theorem 4.1.5. For convenience, let us restate what we must prove: for each $t \in T$, $\underline{a} : T \longrightarrow D^n$, $x \in D^n$ and $y \in D^{n(n+1)}$, and for $i = 1,\ldots,n$ and for $j = 0,\ldots,n$, we claim

$$t \geq j+1 \implies V_{i,j}(t,\underline{a},y) = V_i(j+1,fix(\underline{a}(t-j-1)),x) \tag{5}$$

Choose $t \in T$, $\underline{a} : T \longrightarrow D^n$, $x \in D^n$ and $y \in D^{n(n+1)}$. Also fix $i \in [1,n]$ and $j \in [0,n]$. We will show (5) holds uniformly in i and j and by induction on t as follows:

Basis. If $t = 0$ then t is strictly less than every $j+1$, and so (5) is vacuously true.

Induction. Suppose for some fixed $l \in \mathbf{N}$ that for $t = 0,\ldots,l$ we have shown

$$t \geq j+1 \implies V_{i,j}(t,\underline{a},y) = V_i(j+1,fix(\underline{a}(t-j-1),x) \tag{14}$$

for $i = 1,\ldots,n$ and for $j = 0,\ldots,n$.

We must now prove that (5) holds for $t = l+1$; that is we must show

$$l+1 \geq j+1 \implies V_{i,j}(l+1,\underline{a},y) = V_i(j+1,fix(\underline{a}(l+1-j-1),x)$$

that is,

$$l \geq j \implies V_{i,j}(l+1,\underline{a},y) = V_i(j+1,fix(\underline{a}(l-j),x) \tag{15}$$

We show (15) holds according to the four cases (which originate in the four possible cases for the value $V_{i,j}(t+1,\underline{a},x)$ in the definition of V_{EOE}):

Case 1: $j = 0$;
Case 2: $i+j$ is odd with $i \neq 1$ and $j \neq 0$;
Case 3: $i+j$ is even with $i \neq n$ and $j \neq 0$, and
Case 4: $i+j$ is odd with $i = 1$ and $j \neq 0$, or $i+j$ is even with $i = n$ and $j \neq 0$.

Case 1. Suppose $j=0$. Then $l \geq j \implies l \geq 0$. Also, by definition of $V_{i,0}$ for $i=1,\ldots,n$ we have

$$\begin{aligned} V_{i,0}(l+1,\underline{a},y) &= \underline{a}_i(l) \\ &= (fix(\underline{a}(l))(0))_i \\ &= V_i(1,fix(\underline{a}(l)),x) \end{aligned}$$

(for any $x \in D^n$ by definition of V_i)

$$= V_i(j+1,fix(\underline{a}(l-j)),x)$$

(since $j=0$).

Thus (15) holds for $j=0$.

Case 2. Suppose $i+j$ is odd with $i \neq 1$ and $j \neq 0$. Then by definition of $V_{i,j}$ we have

$$V_{i,j}(l+1,\underline{a},y) = \max\{V_{i-1,j-1}(l,\underline{a},y),V_{i,j-1}(l,\underline{a},y)\} \tag{16}$$

Now, $l \geq j$ and so $l \geq (j-1)+1$. Thus by the induction hypothesis (14) with $t=l$ we have

$$V_{i-1,j-1}(l,\underline{a},y) = V_{i-1}(j,fix(\underline{a}(l-j)),x) \tag{17}$$

and

$$V_{i,j-1}(l,\underline{a},y) = V_i(j,fix(\underline{a}(l-j)),x) \tag{18}$$

Thus from (16), (17), and (18) we have

$$V_{i,j}(l+1,\underline{a},y) = \max\{V_{i-1}(l,fix(\underline{a}(l-j)),x),V_i(l,fix(\underline{a}(l-j)),x)\} \tag{19}$$

There are now two subcases to consider:

Case 2.1: j odd, and
Case 2.2: j even, $\neq 0$.

Case 2.1. If j is odd, then i must be even (since $i+j$ is odd). Also, since $0<j\leq n$ (j is the column index) and so $j \bmod (n+1)=j \neq 0$; thus $j \bmod (n+1)$ is also odd. Thus by definition of $V_{i,j}$ we have

$$\begin{aligned} V_i(j+1,fix(\underline{a}(l-j)),x) &= \max\{V_{i-1}(j,fix(\underline{a}(l-j),x),V_i(j,fix(\underline{a}(l-j)),x)\} \\ &= V_{i,j}(l+1,\underline{a},y) \end{aligned}$$

(from (19)).

Thus (15) holds in this subcase.

Case 2.2. Suppose j is even but nonzero. It is easy to see that $j \bmod (n+1)=j$ and thus $j \bmod (n+1)$ is also even and nonzero. Also, since i must be odd here (but $i \neq 1$), we have

$$V_i(j+1,fix(\underline{a}(l-j)),x) = \max\{V_{i-1}(j,fix(\underline{a}(l-j),x),V_i(j,fix(\underline{a}(l-j)),x)\}$$

(irrespective of whether $i=n$ or not)

$$= V_{i,j}(l+1,\underline{a},y)$$

(from (19)).

Thus (15) holds in the second subcase, and so it holds in the case that $i+j$ is odd with j nonzero; that is, in Case 2.

It is equally easy to show that (15) holds in the remaining two cases (Case 3 and Case 4 which we leave as exercises), and so we may conclude that (15) holds for all $i \in [1,n]$ and all $j \in [0,n]$, and thus (5) holds for $t=l+1$; thus (5) holds for all $t \in T$ by the principle of mathematical induction, proving Theorem 4.1.5.

□

It remains to prove Theorem 4.1.6. As we shall see, this theorem is a direct consequence of two facts: Lemma 4.1.8 concerning a class of synchronous networks which we call *parallel comparison algorithms*, of which EOE is an instance; and Lemma 4.1.9 which concerns OE.

4.1.7 Parallel Comparison Algorithms.

A parallel comparison algorithm, or PCA, is a synchronous algorithm comprising a rectangular array of modules $m_{i,j}$ (see Figures 2.8, 4.1, and 4.2). The modules of a PCA are like those of the EOE network: for a linearly ordered set D, each $m_{i,j}$ computes (depending on i and j, and in one time step), either the maximum function (on D), or the minimum function, or the identity function. The operation of an arbitrary PCA will officially be defined using PR. However, a PCA can be informally described as follows: a PCA P, comprises $\beta \geq 1$ columns of $n \geq 1$ modules, together with n sources, n sinks, and a column of n dummy modules $m_{1,0},\ldots,m_{n,0}$, as illustrated in Figure 4.1 (where $\beta = 5$ and $n = 6$). For $j = 1,\ldots,\beta$, the jth column of P has a number of pairs $(m_{l,j},m_{l+1,j})$ of modules where each module in the pair has the same two inputs (supplied by $m_{l,j-1}$ and $m_{l+1,j-1}$); the intention is that $m_{l,j}$ computes the minimum of the two inputs, and $m_{l+1,j}$ the maximum; we call these pairs of modules *interchange* pairs. Other modules, which are not part of such module pairs, are to be regarded as more dummy modules, that is, modules that compute the identity function in one time step.

We need a notation for expressing the communication structure of an PCA. We do this as follows:

Definition. A *parallel comparison algorithm* P for n inputs is denoted by a vector (of vectors) of numbers, $P = (r_1,\ldots,r_\beta)$ where for $j = 1,\ldots,\beta$:

(i) $r_j = (r_{1,j},\ldots,r_{k_j,j})$ for some $k_j \geq 1$;

(ii) $1 \leq r_{i,j} < n$ for $i = 1,\ldots,k_j$, and

(iii) $|r_{i,j} - r_{i+1,j}| > 1$ for $i = 1,\ldots,k_j$.

We denote the collection of all such PCAs by $PCA(n)$, and if $P = (r_1,\ldots,r_\beta)$ then we say P has *length* β, in symbols: $|P| = \beta$. □

The notation for PCAs is interpreted as follows. Suppose $P = (r_1,\ldots,r_\beta) \in PCA(n)$ where for $j = 1,\ldots,\beta$, $r_j = (r_{1,j},\ldots,r_{k_j,j})$. Of course, to say $|P| = \beta$ is to say that P comprises β columns of modules (or $\beta+1$ including column zero). Each r_j describes the jth column as having k_j interchange pairs. If $r_{i,j} = l$ then the ith interchange pair has its uppermost module on the l^{th} row; that is, the ith interchange pair is $(m_{l,j},m_{l+1,j})$. Condition (ii) above tells us that for a network with n rows $r_{i,j}$ is always at least 1, and that $r_{i,j}+1$ is at most n. Condition (iii) states that any two interchange pairs on the same column never have a module in common.

Example. Consider the PCA P of Figure 4.1. It has description $P=(r_1,...,r_5)$ where $r_1=(1,3)$, $r_2=(2,5)$, $r_3=(5)$, $r_4=(1,3,5)$, and $r_5=(2)$.

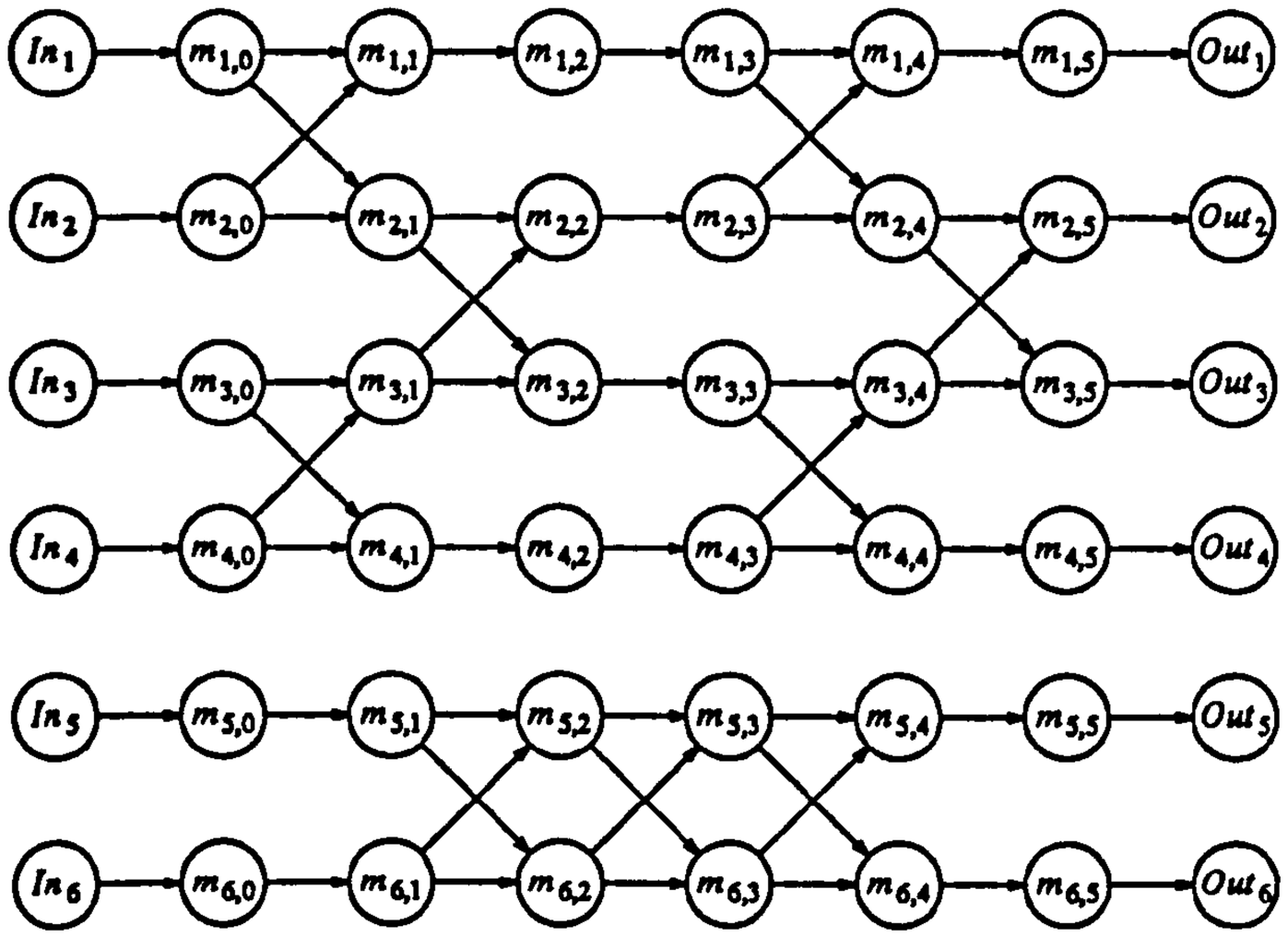


Figure 4.1 - The PCA P.

□

A PCA is a synchronous network in the sense that all its modules operate synchronously and in one time step and so we can use the network specification algorithm of Section 2.4 to construct a value function $V_P \in \mathrm{PR}(\underline{A})$ of a PCA in PR (for the appropriate choice of $\underline{A}$): if $P \in PCA(n)$ has length β then P comprises $n(\beta+1)$ modules, each of which holds a value from D, and so we will have $V_P : T \times [T \longrightarrow D^n] \times D^{n(\beta+1)} \longrightarrow D^{n(\beta+1)}$.

It will be helpful later to have an explicit definition of V_P:

Definition. Let $P \in PCA(n)$ have length β. We say $V_P : T \times [T \longrightarrow D^n] \times D^{n(\beta+1)} \longrightarrow D^{n(\beta+1)}$ is the *PR-characterisation* of P if the coordinate functions $V_{i,j} : T \times [T \longrightarrow D^n] \times D^{n(\beta+1)} \longrightarrow D$ of V^P are defined thus for $i=1,...,n$ and for $j=0,...,\beta$:

$$V_{i,j}(0,\underline{a},x) = x_{i,j}$$

and

$$V_{i,j}(t+1,\underline{a},x) = \begin{cases} \underline{a}_i(t) & \text{if } j=0 \\ \min\{V_{i,j-1}(t,\underline{a},x), V_{i+1,j-1}(t,\underline{a},x)\} & \text{if } i=r_{l,j} \text{ for some } l \in [1,k_j] \\ \max\{V_{i,j-1}(t,\underline{a},x), V_{i-1,j-1}(t,\underline{a},x)\} & \text{if } i=r_{l,j}+1 \text{ for some } l \in [1,k_j] \\ V_{i,j-1}(t,\underline{a},x) & \text{if } i \neq r_{l,j} \text{ and } i \neq r_{l,j}+1 \text{ for any } l \in [1,k_j] \end{cases}$$

□

Exercise. Describe the EOE network (Figure 2.8) as a PCA P_{EOE} and deduce that the PR-characterisation of P_{EOE} coincides with the value function V_{EOE} as defined in Examples 2.4.2(3).

Note. Given $P \in PCA(n)$ with $|P| = \beta$, notice that the function $F_P : T \times [T \longrightarrow D^n] \times D^{n(\beta+1)} \longrightarrow D^n$ (which describes P's output over time) is already defined (see the preliminary notes at the beginning of the chapter). Again, it will be helpful later if we see this definition written out: for each $t \in T$, $\underline{a} : T \longrightarrow D^n$, and $y \in D^{n(\beta+1)}$ we have

$$F_P(t, \underline{a}, y) = (V_{1,\beta}(t, \underline{a}, y), \ldots, V_{n,\beta}(t, \underline{a}, y))$$

where $V_{1,\beta}, \ldots, V_{n,\beta}$ are (the last n) coordinates of V_P. □

After a preliminary definition, we can now state the basic fact upon which Theorem 4.1.6 depends.

Definition. Let D be linearly ordered by $\leq$. We say $a_R = (a_1, \ldots, a_n) \in D^n$ is a *reverse vector* if

$$a_1 > \cdots > a_n$$

(where $a > b \iff \sim(a \leq b)$). Notice that if $a = (a_1, \ldots, a_n)$ is a reverse vector, then for $i, j = 1, \ldots, n$, $i < j \iff a_i > a_j$.

4.1.8 Lemma. *(A PCA sorts a reverse vector if and only if it sorts every vector.) Let $P \in PCA(n)$ have length β and PR-characterisation V_P, and let $a_R \in D^n$ be a reverse vector. Then F_P satisfies*

$$(\forall y \in D^{n(\beta+1)})(\, F_P(\beta+1, fix(a_R), y) = sort(a_R)\,)$$
$$\iff (\forall a \in D^n)(\forall y' \in D^{n(\beta+1)})(\, F_P(\beta+1, fix(a), y') = sort(a)\,)$$

Proof. Postponed.

Proof of Theorem 4.1.6. To prove Theorem 4.1.6, from Lemma 4.1.8 we only need to show that EOE as a PCA P_{EOE}, does indeed sort some reverse vector to show that EOE sorts every vector $a \in D^n$ as required. In fact, we have

4.1.9 Lemma. *(OE sorts any reverse vector.) For any reverse vector $a_R \in D^n$ and any $x \in D^n$,*

$$V_{OE}(n+1, fix(a_R), x) = sort(a_R)$$

Proof. Postponed.

To conclude the proof of Theorem 4.1.6, let $\underline{a} = fix(a_R)$ for any reverse vector $a_R \in D^n$. Then from Theorem 4.1.5 we have

$$V_{i,n}(n+1, \underline{a}, y) = V_i(n+1, fix(\underline{a}(0)), x)$$

for $i = 1, \ldots, n$, for each $\underline{a} : T \longrightarrow D^n$, $x \in D^n$, and $y \in D^{n(n+1)}$. However, $\underline{a}(0) = a_R$ and so

$$V_{i,n}(n+1, \underline{a}, y) = V_i(n+1, fix(a_R), x) \tag{20}$$

Thus for any $y \in D^{n(n+1)}$ we have

$$F_{EOE}(n+1, fix(a_R), y) = F_{EOE}(n+1, \underline{a}, y)$$
$$= (V_{1,n}(n+1, \underline{a}, y), \ldots, V_{n,n}(n+1, \underline{a}, y))$$

(by definition of F_{EOE})

$$= (V_1(n+1, fix(a_R), x), \ldots, V_n(n+1, fix(a_R), x))$$

(by (20) with $i = 1, \ldots, n$)

$$= V_{OE}(n+1, fix(a_R), x)$$
$$= sort(a_R)$$

(by Lemma 4.1.9). Thus EOE sorts any reverse vector and so by Lemma 4.1.8 we have

$$(\forall a \in D^n)(\forall y \in D^{n(n+1)})(\, F_{EOE}(n+1, fix(a), y) = sort(a)\,)$$

which of course proves Theorem 4.1.6.

Postponed Proofs. We will now prove Lemmas 4.1.8 and 4.1.9.

Proof of Lemma 4.1.8. In order to prove Lemma 4.1.8 for an arbitrary PCA, we first transform each PCA P, to a simpler form $Seq(P)$, which we refer to as the *sequentialisation* of P. We then prove Lemma 4.1.8 for such sequentialisations and then invoke an equivalence result to prove Lemma 4.1.8.

Sequentialisation. Consider an arbitrary PCA $P = (r_1, \ldots, r_\beta)$. We can transform P into a new PCA $Seq(P)$ by spreading each column r_j of P with k_j interchange pairs into k_j distinct columns with only one interchange pair per column. If we perform this transformation in a (columnwise) consistent manner, it should be clear that P and $Seq(P)$ will produce the same output for the same input, but P will take time $1+\beta$ to produce output, whereas $Seq(P)$ will take time $1+\beta'$ where $\beta' = k_1 + \cdots + k_\beta$ ($1+\beta'$ being the length of $Seq(P)$). (Compare Figure 4.1 depicting a PCA P with Figure 4.2 depicting $C = Seq(P)$.)

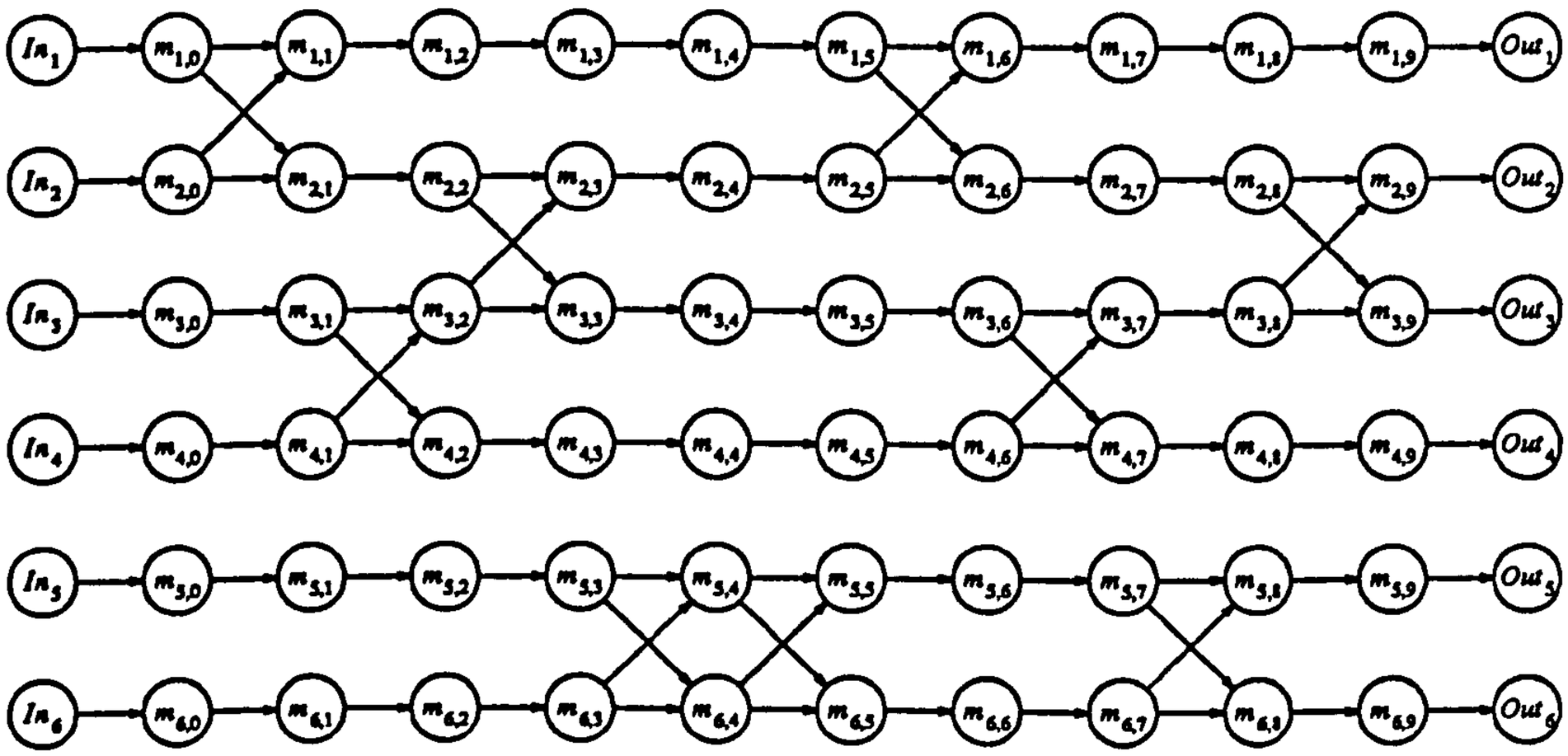


Figure 4.2 - The PCA $C = Seq(P)$

Definition. Let $P = (r_1, \ldots, r_\beta) \in PCA(n)$ where for $j = 1, \ldots, \beta$, $r_j = (r_{1,j}, \ldots, r_{k_j,j})$. Then the PCA $C = Seq(P)$ is defined by

$$C = (s_{1,1}, \ldots, s_{k_1,1}, \ldots, \cdots, s_{1,j}, \ldots, s_{k_j,j}, \cdots, s_{1,\beta}, \ldots, s_{k_\beta,\beta})$$

where $s_{i,j}$ is the singleton $(r_{i,j})$ for $j = 1, \ldots, \beta$ and for $i = 1, \ldots, k_j$. (Notice $Seq : PCA(n) \longrightarrow PCA(n)$; that is, for every PCA $P \in PCA(n)$, $C = Seq(P)$ is also a PCA with n rows.)

A routine inductive proof (which we leave as an exercise) on the length of a PCA yields the following:

4.1.10 Lemma. *(The behaviour of a PCA is invariant under sequentialisation.) Let* $P \in PCA(n)$ *and* $C = Seq(P)$ *have respective PR-characterisations*

$$V_P : T \times [T \longrightarrow D^n] \times D^{n(\beta+1)} \longrightarrow D^{n(\beta+1)}$$

and

$$V_C : T \times [T \longrightarrow D^n] \times D^{n(\alpha+1)} \longrightarrow D^{n(\alpha+1)}$$

where $\alpha = |C|$ *and* $\beta = |P|$ *respectively. Then the output functions* F_C *and* F_P *satisfy:*

$$(\forall a \in D^n)(\forall x \in D^{n(\beta+1)})(\forall y \in D^{n(\alpha+1)}) \ \ (F_P(\beta+1, fix(a), x) = F_C(\alpha+1, fix(a), y))$$

4.1.11 Lemma. *(Every sequentialised PCA sorts a reverse vector if and only if it sorts every vector.) Let* $C = Seq(P)$ *for some PCA* $P \in PCA(n)$*, and let* a_R *be a reverse vector in* D^n*. If* C *has length* α *then*

$$(\forall y \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(a_R), y) = sort(a_R))$$
$$\iff (\forall a \in D^n)(\forall y' \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(a), y') = sort(a))$$

Before proving Lemma 4.1.11 notice that Lemma 4.1.8 is an easy consequence of Lemmas 4.1.10 and 4.1.11; thus it remains to prove Lemma 4.1.11 (and also Lemma 4.1.9 of course).

Proof of Lemma 4.1.11. First notice that if $C = Seq(P)$ sorts every vector then of course it sorts any reverse vector. Thus we only need to prove that if C sorts a reverse vector then it sorts every vector; that is, we must show for every $P \in PCA(n)$ that

$$(\exists b \in D^n)(rev(b) \wedge (\forall y' \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(b), y') = sort(b)))$$
$$\Rightarrow (\forall a \in D^n)(\forall y \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(a), y) = sort(a))$$

where $C = seq(P)$, $\alpha = |C|$, and $rev(b) \iff b$ is a reverse vector. In fact, our strategy for establishing this result is to prove the contrapositive; that is, we prove that if C does *not* sort every vector then, in particular, it does not sort *any* reverse vector; formally, we make the following

Claim. For every $P \in PCA(n)$, if $C = Seq(P)$ with $|C| = \alpha$ then

$$(\exists a \in D^n)(\exists y \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(a), y) \neq sort(a))$$
$$\Rightarrow (\forall b \in D^n)(\neg rev(b) \vee (\exists y' \in D^{n(\alpha+1)})(F_C(\alpha+1, fix(b), y') \neq sort(b)))$$

Of course, Lemma 4.1.11 follows easily from the Claim.

Proof of Claim. First notice that if $a = (a_1, \ldots, a_n) \in D^n$ and $F_C(\alpha+1, fix(a), y) = b \in D^n$, then $b \neq sort(a)$ iff for some i and j with $1 \leq i < j \leq n$ we have $b_i > b_j$ when $b = (b_1, \ldots, b_n)$. Now, for $i = 1, \ldots, n$ b_i is the ith coordinate of $F_C(\alpha+1, fix(a), y)$ which is $V_{i,\alpha}(\alpha+1, fix(a), y)$ when $V_{1,\alpha}, \ldots, V_{n,\alpha}$ are (the last n) coordinates of V_C, the PR-characterisation of C. Thus

$$F_C(\alpha+1, fix(a), y) \neq sort(a) \tag{21}$$
$$\iff (\exists i)(\exists j)(1 \leq i < j \leq n \wedge V_{i,\alpha}(\alpha+1, fix(a), y) > V_{j,\alpha}(\alpha+1, fix(a), y))$$

If $a \in D^n$ and $y \in D^{n(\alpha+1)}$ are such that $F_C(\alpha+1, fix(a), y) \neq sort(a)$ then (from (21)) it must be that for some i and j with $1 \leq i < j \leq n$ we have $V_{i,\alpha}(\alpha+1, fix(a), y) > V_{j,\alpha}(\alpha+1, fix(a), y)$. To prove the Claim then, we must show that for any reverse vector $b \in D^n$ there exists some $y' \in D^{n(\alpha+1)}$ and some i' and j'

with $1 \le i' < j' \le n$ such that $V_{i',\alpha}(\alpha+1, fix(b), y') > V_{j',\alpha}(\alpha+1, fix(b), y')$. We will show that $i' = i$ and $j' = j$ have this property for any $y' \in D^{n(\alpha+1)}$. Before we do so however, we introduce some alternative notation for sequentialised PCAs that will make the proof clearer:

First, if C is a sequentialised PCA with $|C| = \alpha$, then we write $C = (r_1, \ldots, r_\alpha)$; for $j = 1, \ldots, \alpha$, $l = r_j \in [1, n-1]$ indicates that $(m_{l,j}, m_{l+1,j})$ is the single interchange pair on the jth column of C. For example, the PCA $C = Seq(P)$ of Figure 4.2 has description (1,3,2,5,5,1,3,5,2).

Secondly, for an arbitrary sequentialised PCA $C = (r_1, \ldots, r_\alpha)$, we define the map $f_C : D^n \longrightarrow D^n$ by $f_C = f_\alpha \circ \cdots \circ f_1$ where for $j = 1, .., \alpha$,

$$f_j = (f_{1,j}, \ldots, f_{n,j}) : D^n \longrightarrow D^n$$

where for $i = 1, \ldots, n$ each $f_{i,j} : D^n \longrightarrow D$ is defined for each $a = (a_1, \ldots, a_n) \in D^n$ by

$$f_{i,j}(a) = \begin{cases} \min\{a_i, a_{i+1}\} & \text{if } i = r_j \\ \max\{a_i, a_{i+1}\} & \text{if } i = r_j + 1 \\ a_i & \text{otherwise} \end{cases} \tag{22}$$

Notice that for $j = 1, \ldots, n$, f_j describes the action of the jth column of C in isolation from the rest of the network. A routine inductive proof (which we leave as an exercise) on the length of C yields the following fact:

For $i = 1, \ldots, n$ and for each $a \in D^n$ and $y \in D^{n(\alpha+1)}$,

$$f_C(a)_i = V_{i,\alpha}(\alpha+1, fix(a), y) \tag{23}$$

where for $i = 1, \ldots, n$, $f_C(.)_i$ is the ith coordinate of f_C which satisfies

$$f_C(a)_i = \begin{cases} f_{i,1}(a) & \text{if } \alpha = 1 \\ f_{i,\alpha}((f_{\alpha-1} \circ \cdots \circ f_1)(a)) & \text{if } \alpha > 1 \end{cases} \tag{24}$$

for each $a \in D^n$.

If $a \in D^n$ and $y \in D^{n(\alpha+1)}$ are such that for some i and j with $1 \le i < j \le n$ we have $V_{i,\alpha}(\alpha+1, fix(a), y) > V_{j,\alpha}(\alpha+1, fix(a), y)$, then from (23) it must be that $f_C(a)_i > f_C(a)_j$. Now suppose we can show that $f_C(b)_i > f_C(b)_j$ for any reverse vector $b \in D^n$; then again from (23) it must be that $V_{i,\alpha}(\alpha+1, fix(b), y') > V_{j,\alpha}(\alpha+1, fix(b), y')$ for any $y' \in D^{n(\alpha+1)}$ proving the Claim.

It remains to show that any $C = Seq(P)$ has the property that for any i and j with $1 \le i < j \le n$

$$(\exists a \in D^n)(f_C(a)_i > f_C(a)_j) \implies (\forall b \in D^n)(rev(b) \implies (f_C(b)_i > f_C(b)_j)) \tag{25}$$

We now prove (25) by induction on $\alpha = |C|$ as follows:

Basis. If $\alpha = 1$, then C is the singleton $C = (r)$ where $r = r_1 = r_\alpha$. Furthermore, it is not difficult to check that for each $a = (a_1, \ldots, a_n) \in D^n$, $f_C(a)$ satisfies:

$$f_C(a)_k = \begin{cases} \min\{a_k, a_{k+1}\} & \text{if } k = r \\ \max\{a_k, a_{k+1}\} & \text{if } k = r+1 \\ a_k & \text{otherwise} \end{cases} \tag{26}$$

for $k = 1, \ldots, n$, and so for any reverse vector $b = (b_1, \ldots, b_n) \in D^n$ we have

$$f_C(b)_k = \begin{cases} b_{k+1} & \text{if } k = r \\ b_k & \text{if } k = r+1 \\ b_k & \text{otherwise} \end{cases} \qquad (27)$$

Now suppose the premise of (25) holds for some $a \in D^n$ and some i and j with $1 \le i < j \le n$; then we must show $f_C(b)_i > f_C(b)_j$ for any reverse vector $b \in D^n$. Superficially, there are nine cases on i and j to be considered; these are tabulated below:

	$i=r$	$i=r+1$	$r \neq i \neq r+1$
$j=r$	1	2	3
$j=r+1$	4	5	6
$r \neq j \neq r+1$	7	8	9

Notice it cannot be that $i=r$ and $j=r+1$ (Case 4) since if this were so then from (26) we would have

$$f_C(a)_i = f_C(a)_r = \min\{a_r, a_{r+1}\} \le \max\{a_r, a_{r+1}\} = f_C(a)_{r+1} = f_C(a)_j$$

contradicting the premise $f_C(a)_i > f_C(a)_j$. Also, trivially, Cases 1,2, and 5 do not arise, since $i<j$ by hypothesis. The five remaining cases are all simple; we shall consider Case 6 and Case 9 only.

Case 6. Suppose $r \neq i \neq r+1$ and $j=r+1$. Then from (27) we have $f_C(b)_i = b_i$ and $f_C(b)_j = b_{r+1} = b_j$. Thus $f_C(b)_i = b_i > b_j = f_C(b)_j$ since $i<j$ and b is a reverse vector.

Case 9. Suppose $r \neq i \neq r+1$ and $r \neq j \neq r+1$. Then from (27) we have $f_C(b)_i = b_i$ and $f_C(b)_j = b_j$. Thus $f_C(b)_i = b_i > b_j = f_C(b)_j$ since $i < j$ and b is a reverse vector.

Since (25) also holds in Cases 3,7, and 8 (we claim), we conclude (25) holds when $|C| = 1$.

Induction. Suppose for some fixed $\gamma \ge 1$, that for any sequentialised PCA C' with $|C'| = \gamma$, and for any i,j with $1 \le i < j \le n$, we have proved

$$(\exists a \in D^n)(f_{C'}(a)_i > f_{C'}(a)_j) \Rightarrow (\forall b \in D^n)(rev(b) \Rightarrow (f_{C'}(b)_i > f_{C'}(b)_j)) \qquad (28)$$

Now suppose $C = Seq(P)$ for some PCA $P \in PCA(n)$ such that $|C| = \gamma+1$. It is not difficult to see that if $C = (r_1,\ldots,r_\gamma,r)$ for some $r_1,\ldots,r_\gamma,r \in [1,n-1]$, say, then the PCA $C' = (r_1,\ldots,r_\gamma)$ satisfies $C' = Seq(P')$ for some $P' \in PCA(n)$.

Now, $f_C = f_{\gamma+1} \circ f_\gamma \circ \cdots \circ f_1$, where for $j = 1,\ldots,\gamma+1$, $f_j = (f_{1j},\ldots,f_{nj}) : D^n \longrightarrow D^n$ is as defined by (22), so $f_C = f_{\gamma+1} \circ f_{C'}$ (since the first γ columns of C and C' are identical). That is, for any $a \in D^n$, and for $k=1,\ldots,n$,

$$f_C(a)_k = f_{k,\gamma+1}(f_{C'}(a))$$

$$= \begin{cases} \min\{f_{C'}(a)_k, f_{C'}(a)_{k+1}\} & \text{if } k = r \\ \max\{f_{C'}(a)_k, f_{C'}(a)_{k+1}\} & \text{if } k = r+1 \\ f_{C'}(a)_k & \text{otherwise} \end{cases} \qquad (29)$$

(using (24)). We can now show (25) holds for C. There are five cases to consider:

Case 1: $r \neq i \neq r+1$ and $r \neq j \neq r+1$
Case 2: $i = r$
Case 3: $i = r+1$
Case 4: $j = r$

Case 5: $j = r+1$

We will show (25) holds in Cases 1 and 2 only; the remaining cases have proofs very similar to that of Case 2.

Case 1. Suppose $r \neq i \neq r+1$ and $r \neq j \neq r+1$. Then by (29) for any $a \in D^n$ we have

$$f_C(a)_i = f_{C'}(a)_i \tag{30}$$

and

$$f_C(a)_j = f_{C'}(a)_j \tag{31}$$

Thus from (30) and (31) we have

$$(\exists a \in D^n)(\, f_C(a)_i > f_C(a)_j \,) \Rightarrow (\exists a \in D^n)(\, f_{C'}(a)_i > f_{C'}(a)_j \,)$$
$$\Rightarrow (\forall b \in D^n)(\, rev(b) \Rightarrow (\, f_{C'}(b) > f_{C'}(b)_j \,)\,)$$

(by the induction hypothesis (28))

$$\Rightarrow (\forall b \in D^n)(\, rev(b) \Rightarrow (\, f_C(b) > f_C(b)_j \,)\,)$$

again by (30) and (31). Thus the Claim (25) holds in this case.

Case 2. Suppose $i = r$. Since $r = i < j$ we must have $j = r+1$ or $j > r+1$. In fact, it must be that $j > r+1$, for if $j = r+1$, then $m_{i,\gamma+1}$ and $m_{i+1,\gamma+1}$ are the interchange modules on column $\gamma+1$ of C, and it so for any $a \in D^n$ we have $f_C(a)_i \leq f_C(a)_j$ contradicting a premise of the Claim (25) (so there is nothing to prove).

Suppose $j > r+1$ then. Now, we cannot have either $f_{C'}(a)_i \leq f_{C'}(a)_j$ or $f_{C'}(a)_{i+1} \leq f_{C'}(a)_j$ since then

$$\min\{f_{C'}(a)_i, f_{C'}(a)_{i+1}\} \leq f_{C'}(a)_j \tag{32}$$

and so

$$f_C(a)_i = \min\{f_{C'}(a)_i, f_{C'}(a)_{i+1}\}$$

(from (29))

$$\leq f_{C'}(a)_j$$

(by (32))

$$= f_C(a)_j$$

again from (29). Thus $f_C(a)_i \leq f_C(a)_j$, contradicting the same premise of the Claim (25).

Thus, if there is anything to prove, it must be when $f_{C'}(a)_i$ and $f_{C'}(a)_{i+1}$ are both strictly greater than $f_{C'}(a)_j$. In this case, notice that since $r = i < j$ and $j > r+1$, we have $i < j$ and $i+1 < j$. Thus by the induction hypothesis (28) we have

$$f_{C'}(b)_i > f_{C'}(b)_j \tag{33}$$

and

$$f_{C'}(b)_{i+1} > f_{C'}(b)_j \tag{34}$$

for any reverse vector $b \in D^n$.

Thus,

$$f_C(b)_i = \min\{f_{C'}(b)_i, f_{C'}(b)_{i+1}\}$$

(by (29))

$$> f_C(b)_j$$

(by (33) and (34))

$$= f_C(b)_j$$

(by (29)).

Thus (25) holds in this case also.

Since (25) also holds in Cases 3,4, and 5 (we claim), we conclude that (25) holds when $|C| = \gamma+1$; thus by the principle of mathematical induction the Claim holds for every C. □

Proof of Lemma 4.1.9. Let us first define $Z = (Z_1,\ldots,Z_n) : [0,n] \times D^n \longrightarrow D^n$ coordinatewise for each $t \in [0,n]$ and for each $a = (a_1,\ldots,a_n) \in D^n$ by:

$$Z_i(t,a) = \begin{cases} a_{i+t} & \text{if } i+t \text{ even}, i \leq n-t \\ a_{2n-t-i+1} & \text{if } i+t \text{ even}, i > n-t \\ a_{t-i+1} & \text{if } i+t \text{ odd}, i \leq t-1 \\ a_{i-t} & \text{if } i+t \text{ odd}, i > t-1 \end{cases} \tag{35}$$

Notice that taking $t = n$ in (35) we obtain

$$Z_i(n,a) = a_{n-i+1}$$

for $i = 1,\ldots,n$, and so

$$Z(n,a_1,\ldots,a_n) = (a_n,\ldots,a_1) \tag{36}$$

We now make the following

Claim. For $i = 1,\ldots,n$:

$$(\forall t \in [0,n])(\forall a \in D^n)(\forall x \in D^n)\ (\, rev(a) \Longrightarrow (\, V_i(t+1,fix(a),x) = Z_i(t,a)\,)\,) \tag{37}$$

where $V_1,\ldots,V_n$ are the coordinates of the value function V_{OE}.

Notice that if a_R is a reverse vector, then for any $x \in D^n$ we have

$$V_{OE}(n+1,fix(a_R),x) = (V_1(n+1,fix(a_R),x),\ldots,V_n(n+1,fix(a_R),x))$$
$$= (Z_1(n,a_R),\ldots,Z_n(n,a_R))$$

(by the Claim (37))

$$= Z(n,a_R)$$
$$= sort(a_R)$$

(by (36)) which proves Lemma 4.1.9.

Proof of Claim. We prove (37) uniformly in i and by induction on t as follows:

Basis. If $t = 0$, then immediately by the definitions of V_i and Z_i for any $i \in [1,n]$, we have for any $a,x \in D^n$ that

$$V_i(1,fix(a),x) = fix(a)(0)_i$$
$$= a_i$$
$$= Z_i(0,a)$$

(here we have used the fact that i cannot be 0 or greater than n).

Thus (37) holds for $t=0$.

Induction. Suppose for some fixed $k \leq 0$ that for $t=0,\ldots,k$ and for $i=1,\ldots,n$ we have proved

$$(\forall a \in D^n)(\forall x \in D^n)\ \ (rev(a) \Longrightarrow V_i(k+1,fix(a),x) = Z_i(k,a)) \tag{38}$$

We now show that (37) holds for $t=k+1$; that is, we show for $i=1,\ldots,n$

$$(\forall a \in D^n)(\forall x \in D^n)\ \ (rev(a) \Longrightarrow V_i(k+2,fix(a),x) = Z_i(k+1,a)) \tag{39}$$

There are four cases to consider, these being:

Case 1: $i+(k+2)$ is even, $i \neq 1$,
Case 2: $i+(k+2)$ is even, $i=1$,
Case 3: $i+(k+2)$ is odd, $i \neq n$,
Case 4: $i+(k+2)$ is odd, $i=n$.

We will prove the Claim in Cases 1 and 2 only; Cases 3 and 4 have proofs almost identical to those of Cases 1 and 2 respectively.

First notice that since the statement of the Lemma only quantifies over $t \in [0,n]$, we may assume that $k+1 \leq n$: if $k+1>n$, then the induction step will hold vacuously. Also, it is not difficult to prove that for $t \in [1,n]$ the coordinates $V_1,\ldots,V_n$ of V_{OE} satisfy the following equation for any $\underline{a}: T \longrightarrow D^n$ and $x \in D^n$:

$$V_i(t+1,\underline{a},x) = \begin{cases} \min\{V_i(t,\underline{a},x), V_{i+1}(t,\underline{a},x)\} & \text{if } i+t \text{ even}, i \neq n \\ \max\{V_{i-1}(t,\underline{a},x), V_i(t,\underline{a},x)\} & \text{if } i+t \text{ odd}, i \neq 1 \\ V_i(t,\underline{a},x) & \text{otherwise} \end{cases} \tag{40}$$

Now let $a \in D^n$ be a reverse vector, let $x \in D^n$ be any vector, and fix $i \in [1,n]$; we will now show that $V_i(k+2,fix(a),x) = Z_i(k+1,a)$ (in Cases 1 and 2).

Case 1. Suppose $i+(k+2)$ is even and $i \neq 1$. Then $i+(k+1)$ is odd with $i \neq 1$; also $k+1 \leq n$ and so from (40) we have

$$\begin{aligned} V_i(k+2,fix(a),x) &= V_i((k+1)+1,fix(a),x) \\ &= \max\{V_{i-1}(k+1,fix(a),x), V_i(k+1,fix(a),x)\} \\ &= \max\{Z_{i-1}(k,a), Z_i(k,a)\} \end{aligned} \tag{41}$$

by the induction hypothesis (38) applied to V_{i-1} and V_i. There are now two subcases to consider, these being:

Case 1.1: $i-1 \leq k-1$, and
Case 1.2: $i-1 > k-1$.

Case 1.1. Suppose $i-1 \leq k-1$. Then since $i+(k+2)$ is even here, $i+(k+1)$ is odd and since $i-1 \leq k-1$ we have $i \leq k$, that is, $i \leq (k+1)-1$. Thus, from (35) we have

$$Z_i(k+1,a) = a_{k+1-i+1} = a_{k-i+2} \tag{42}$$

Also since $i+(k+2)$ is even, we have $(i-1)+k$ is odd and again $i-1 \leq k-1$; thus from (35) we have

$$Z_{i-1}(k,a) = a_{k-(i-1)+1} = a_{k-i+2}$$

and thus from (41) we have

$$V_i(k+2,fix(a),x) = \max\{a_{k-i+2}, Z_i(k,a)\} \tag{43}$$

There are now two subsubcases:

Case 1.1.1: $i \leq n-k$, and
Case 1.1.2: $i > n-k$.

Case 1.1.1. Suppose $i \leq n-k$. Then since $i+(k+2)$ is even, we have that $i+k$ is also even, and so from (35) we have

$$Z_i(k,a) = a_{i+k} \tag{44}$$

Substituting for $Z_i(k,a)$ from (44) in (43) yields

$$V_i(k+2,fix(a),x) = \max\{ a_{k-i+2}, a_{i+k} \} \tag{45}$$

Now, the greater of a_{k-i+2} and a_{i+k} is the one with the smaller index (since a is a reverse vector): notice that $k-i+2 \geq i+k$ iff $i=1$, but $i \neq 1$ by the Case 1 hypothesis; thus $k-i+2 < k+i$ and so from (45) we have

$$\begin{aligned} V_i(k+2,fix(a),x) &= \max\{ a_{k-i+2}, a_{i+k} \} \\ &= a_{k-i+2} \\ &= Z_i(k+1,a) \end{aligned}$$

from (42). Thus (39) holds in this subsubcase.

Case 1.1.2. Suppose $i > n-k$. Then again since $i+(k+2)$ is even, we have $i+k$ is even, and so from (35) we have

$$Z_i(k,a) = a_{2n-k-i+1} \tag{46}$$

Substituting for $Z_i(k,a)$ from (46) in (43) yields

$$V_i(k+2,fix(a),x) = \max\{ a_{k-i+2}, a_{2n-k-i+1} \} \tag{47}$$

Similar to Case 1.1.1 we must now determine which of $k-i+2$ and $2n-k-i+1$ is the smaller (in order to determine which of a_{k-i+2} and $a_{2n-k-i+1}$ is the larger). Since $k+1 \leq n$ we have $k < n$ and so $2k < 2n$ and so $2k+1 \leq 2n$; thus $k-i+1 \leq 2n-k-i$ (subtracting $k+i$ from both sides), and so $k-i+2 \leq 2n-k-i+1$. However, it cannot be that $k-i+2 = 2n-k-i+1$ (since this would imply that $2k+2 = 2n+1$; but $2k+2$ is even whereas $2n+1$ is odd) and so we conclude $k-i+2 < 2n-k-i+1$. Thus, from (47) we have

$$\begin{aligned} V_i(k+2,fix(a),x) &= \max\{ a_{k-i+2}, a_{2n-k-i+1} \} \\ &= a_{k-i+2} \\ &= Z_i(k+1,a) \end{aligned}$$

from (42). Thus (39) holds in this subsubcase also.

Case 1.2. Suppose $i-1 > k-1$. Then $i > k = (k+1)-1$. Also, again since $i+(k+2)$ is even, we have that $i+(k+1)$ is odd and again $i-1 > k-1$; thus from (35) we have

$$Z_i(k+1,a) = a_{i-(k+1)} = a_{i-1-k} \tag{48}$$

Also since $i+(k+2)$ is even, $(i-1)+k$ is odd, and so from (35) we have

$$Z_{i-1}(k,a) = a_{i-1-k}$$

and thus from (41) we have

$$V_i(k+2,fix(a),x) = \max\{ a_{i-1-k}, Z_i(k,a) \} \tag{49}$$

Again there are two subsubcases:

Case 1.2.1: $i \leq n-k$, and

Case 1.2.2: $i > n-k$.

Case 1.2.1. Suppose $i \leq n-k$. Then exactly as in Case 1.1.1 we can show that

$$Z_i(k,a) = a_{i+k} \tag{50}$$

Substituting for $Z_i(k,a)$ from (50) in (49) yields

$$V_i(k+2,fix(a),x) = \max\{ a_{i-1-k}, a_{i+k} \}$$
$$= a_{i-1-k}$$

(since $i-1-k < i+k$, and a is a reverse vector)

$$= Z_i(k+1,a)$$

from (48). Thus (39) holds in this subsubcase.

Case 1.2.2. Suppose $i > n-k$. Then exactly as in Case 1.1.2 we can show:

$$Z_i(k,a) = a_{2n-k-i+1} \tag{51}$$

Substituting for $Z_i(k,a)$ from (51) in (49) yields

$$V_i(k+2,fix(a),x) = \max\{ a_{i-1-k}, a_{2n-k-i+1} \} \tag{52}$$

However, it is easy to check that $i-1-k < 2n-k-i+1$ follows from the fact that $i < n+1$ and so from (52) we have

$$V_i(k+2,fix(a),x) = \max\{ a_{i-1-k}, a_{2n-k-i+1} \}$$
$$= a_{i-1-k}$$
$$= Z_i(k+1,a)$$

from (48). Thus (39) holds in this final subsubcase, and hence in Case 1.

Case 2. Suppose $i+(k+2)$ is even but $i = 1$. Then $i+(k+1)$ is odd and so from (40) we have

$$V_i(k+2,fix(a),x) = V_i(k+1,fix(a),x)$$
$$= Z_i(k,a) \tag{53}$$

(by the induction hypothesis (38) applied to V_i). Now, since $k < n$, we have $k \leq n-1$ and so $i = 1 \leq n-k$; thus from (35) we have $Z_i(k,a) = a_{i+k} = a_{1+k}$. Also $i+k$ is even since $i+(k+2)$ is even and so from (53) we have

$$V_i(k+2,fix(a),x) = a_{1+k}$$

Thus to show (39) holds in the current case, we must show that $Z_i(k+1,a) = a_{1+k}$. Since $k \geq 1$ we have $i = 1 \leq (k+1)-1$ and so from (35) we have

$$Z_i(k+1,a) = a_{k+1-i+1} = a_{1+k}$$

(since $i = 1$) as required.

Since (39) holds in the remaining cases (we claim), we conclude that the Claim (37) holds for $t = k+1$ and so by the principle of mathematical induction, the Claim holds for every $t \in T$. □

4.2 SOURCES.

The origins of the OE and EOE sorting algorithms were mentioned in Section 2.6. The proofs the algorithms as stream processors are my own work, although the basic strategy behind the proof of Lemma 4.1.11 (using 'reverse vectors') is due to R. W. Floyd: see Exercise 36(b) in Knuth[1973] (p. 241). The concept of a PCA is based on the parallel sorting networks found in Section 5.3.4 of

Knuth[1973]. Knuth and Floyd do not of course consider stream processing.

Satisfactory proofs of correctness of synchronous algorithms are extremely rare, even in the theoretical research literature on systolic algorithms and verification. In the general scientific literature such algorithms are not even clearly described. The careful treatment of the sorting networks given here is to be contrasted with that found in the text-book Akl[1985]. For example, the treatment of OE in Akl[1985] (p. 41) is at best cursory, and the proof of correctness given there (essentially two diagrams of the sorter in operation) is not a proof at all.

CHAPTER 5
FURTHER EXAMPLES

In this chapter we test out our formalism by using PR to specify and establish the correctness of a variety of synchronous algorithms. First, we consider an algorithm for *finite impulse response filtering* (or *convolution*) found in Brookes[1983]; this algorithm has parallel loading of data. Second, we adapt the first algorithm to make an alternative algorithm for convolution that has serial loading of data. Thirdly, we consider a new algorithm for recognising *palindromes*. Finally we consider the *matrix-vector multiplication* algorithm of Kung and Leiserson (taken from Chapter 8 of Mead and Conway[1980]).

The purpose of this chapter is to convince the reader by means of the number and variety of examples (recall we have already specified two sorting algorithms), that a large number of synchronous algorithms are amenable to our specification technique. Thus our interest lies not in a given algorithm *per se*, but rather in showing that it is specifiable by the methodology proposed in Section 2.4 and subsequently generalised and formalised in Section 3.4.

Each example is treated in the following way: first, we write down a user specification Φ_N of the problem at hand; second, we present and informally discuss a synchronous network N to solve the problem; thirdly, we formalise the network by writing down its value function V_N; finally, we establish the correctness of the algorithm.

5.1 FINITE IMPULSE RESPONSE FILTERING.

Let R be a Σ-algebra for some signature Σ which involves symbols for zero, addition, and multiplication; for example, let R be a *ring with 0*. Also, for some fixed constants (or *weights*) $w_1,...,w_n \in R$, let $fir : R^n \longrightarrow R$ be defined by

$$fir(a) = w_1 . a_n + \cdots + w_i . a_{n-i+1} + \cdots + w_n . a_1 \qquad (1)$$

for each $a = (a_1,...,a_n) \in R^n$.

The *finite impulse response filtering problem* over R is to devise a synchronous network N with clock $T = T_N$ which implements the following user specification:

$$\Phi_N : [T \longrightarrow R^n] \longrightarrow [T \longrightarrow R]$$

where for each $\underline{a} : T \longrightarrow R^n$ and for each $t \in T$,

$$\Phi_N(\underline{a})(t) = \begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}(t-n)) & \text{if } R(t) \end{cases} \qquad (2)$$

where $R(t) \Longleftrightarrow t \geq n$. (The ready condition R should not be confused with the ring R of course.) Thus the problem is to compute *fir* on every input $\underline{a}(0),\underline{a}(1),\underline{a}(2),....$

As preliminary notation, for $r = 1,...,n$, and for $c = 0,1,2,...$, let $y_{r,c} \in R$ be defined by

$$y_{r,c} = \sum_{k=1}^{k=r} w_k \cdot \underline{a}_{n-k+1}(c) \qquad (3)$$

Notice $y_{n,c} = fir(\underline{a}(c))$; generally, $y_{r,c}$ is sum of the first r terms of $fir(\underline{a}(c))$ as given by (1).

5.1.1 The FIR Algorithm.

A synchronous algorithm for implementing Φ_N is depicted in Figure 5.1. The network FIR of Figure 5.1 comprises n modules $m_1,...,m_n$, n sources $In_1,...,In_n$, and a single sink Out.

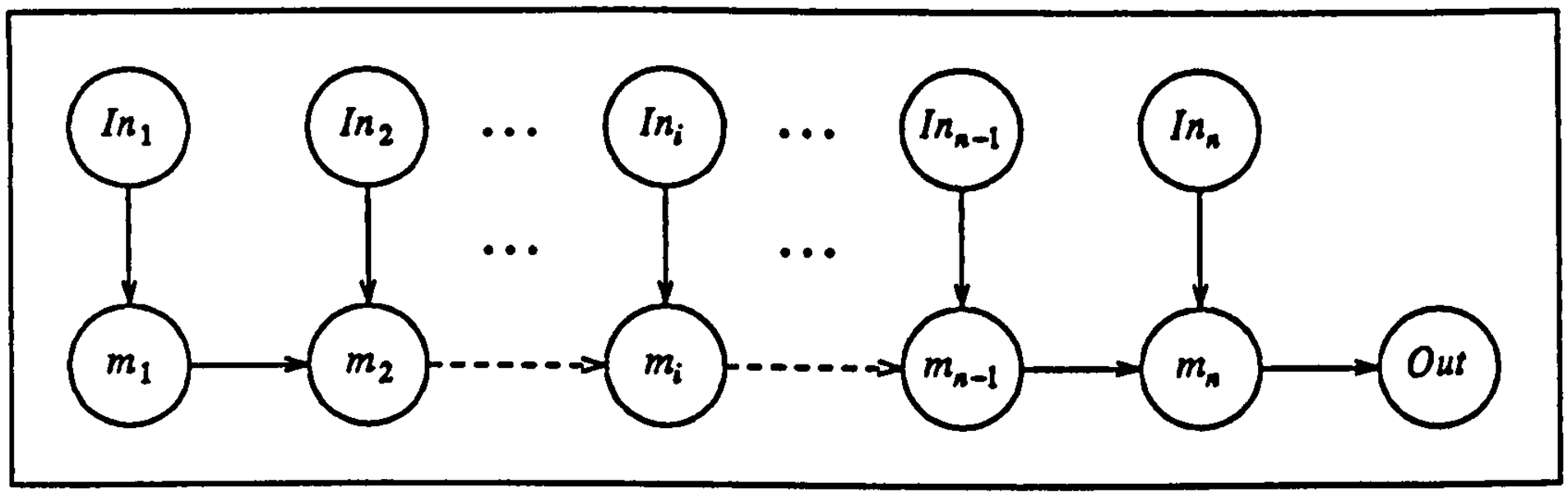


Figure 5.1 - The network FIR.

Each of FIR's n sources supplies elements of R, and so we represent the the input to FIR as a stream $\underline{a} : T \longrightarrow R^n$ (with the intention that for $i = 1,...,n$, the value supplied by In_i at time t is $\underline{a}_i(t)$).

The FIR network operates as follows. Initially, at $t = 0$, we imagine each module m_i of FIR to be holding some initial value $x_i \in R$, and to be about to read $\underline{a}_i(0)$ from source In_i for $i = 1,...,n$.

For $t = 0,1,2,...$, the value held by each m_i by time $t+1$ is calculated as follows for $i = 1,...,n$:

Case $i = 1$. Module m_1 first reads $\underline{a}_1(t)$ from In_1 and then multiplies this value by the weight w_1.

Case $i = 2,...,n$. Module m_i first reads $\underline{a}_i(t)$ from In_i and multiplies this value by the weight w_i. The result of this multiplication is then added to the value held by module m_{i-1} at time t. □

The operation of the FIR network is illustrated for the case $n = 5$ and for $t = 0,1,2,4,6$ in Figures 5.5 - 5.9 (which are to be found at the end of this chapter). The reader is recommended to scan through these figures before attempting to follow the detailed description below. The figures are quite detailed and warrant some preliminary explanation:

Consider a typical figure (Figure 5.7 say). In the figure we see the FIR network at a given time t (we have not drawn the sources). Above the network, we see the data which is to be supplied to the network; the column of data above each module is the stream to be supplied to that module. On the extreme left of the figure there is a column of numbers that we have included for reference purposes: the ith datum (from the left) in the row of data adjacent to a number k in the column is the datum to be supplied to m_i at time $t = k$. For example, considering Figure 5.7, $\underline{a}_3(0)$ is the input to m_3 at $t = 2$, and $\underline{a}_1(0)$ will be the input to m_5 at $t = 4$. A feature of the input data is that for $c = 0,1,2,...$, the coordinates

$\underline{a}_1(c),\ldots,\underline{a}_n(c)$ of each input vector $\underline{a}(c)$ do not form a row; the elements are supplied in reverse order (the ith coordinate is supplied to module m_{n-i+1} rather than to m_i) and with a unit delay between each successive element. The symbol '.' in the figures is used in two ways: below the bands of input data a '.' means 'don't care' (for definitiness the reader may think of these values as zeros). Above the input data and extending infinitely upwards, a '.' means 'whatever comes next': we have only shown the first three input vectors to emphasise the pattern of the input data; actually, the input streams extend upwards infinitely; for example, considering Figure 5.7 again, the values directly above $\underline{a}_3(2)$ will be $\underline{a}_3(3)$, $\underline{a}_3(4)$, $\underline{a}_3(5)$ etc. Finally, the value situated to the right of a module and slightly below the module's output channel is intended to be the value held by a module at the current time t; for example, at $t=2$ m_2 holds '$y_{2,0}$' and m_5 holds 'don't care'.

In tracing the behaviour of FIR (in the figures), it is helpful to notice from (3) above that

$$y_{r+1,c} = y_{r,c} + w_{r+1} \cdot \underline{a}_{n-r}(c) \tag{4}$$

for $r=1,\ldots,n-1$, and for $c=0,1,2,\ldots$. Thus for example, in the figures, where $n=5$, at $t=6$ the inputs to m_5 are $y_{4,2}$ and $\underline{a}_1(2)$ (see Figure 5.9), and so by time $t=7$, m_5 holds

$$y_{4,2}+w_5 \cdot \underline{a}_1(2) = y_{4,2}+w_{4+1} \cdot \underline{a}_{5-4}(2) = y_{4+1,2} = y_{5,2} = fir(\underline{a}(2))$$

Let us now consider in detail how $fir(\underline{a}(0))=y_{n,0}$ is computed by FIR with the given input configuration.

Since the input to m_1 at $t=0$ is $\underline{a}_n(0)$ (see Figure 5.5), m_1 holds $w_1 \cdot \underline{a}_n(0)=y_{1,0}$ at $t=1$ (see Figure 5.6). Since this value is the input to m_2 at time $t=1$ together with $\underline{a}_{n-1}(0)$, m_2 holds $w_1 \cdot \underline{a}_n(0)+w_2 \cdot \underline{a}_{n-1}(0)=y_{2,0}$ at time $t=2$ (see Figure 5.7). It is not difficult to see that by $t=n$, m_n will hold $y_{n,0}=fir(\underline{a}(0))$.

Of course, during these first n cycles FIR is simultaneously computing other partial sums $y_{r,c}$. For example, at $t=1$ the input to m_1 is $\underline{a}_n(1)$, and so by time $t=2$, m_1 holds $w_1 \cdot \underline{a}_n(1)=y_{1,1}$. In fact, it is not difficult to see that for the *particular* input configuration of Figure 5.5, m_i holds $y_{i,t-i}$ at time t for $t \geq i$, and hence m_n holds $y_{n,t-n}=fir(\underline{a}(t-n))$ at time t for $t \geq n$. Before we can prove this fact we must f FIR below; first we must formalise the operation of FIR on *arbitrary* input by defining its value function V_{FIR} and from this the stream transformation G_{FIR}.

5.1.2 Formal Specification of FIR.

We will now formalise the operation of FIR according to our specification technique: first we formalise each module by specifying its behaviour by means of a function on R, and then we specify the value held by each module at each time t by means of a value function.

Module Specification. For $i=1,\ldots,n$, the operation of the ith module m_i is specified by means of the function f_i defined as follows: first, we define

$$f_1 : R \longrightarrow R$$

by

$$(\forall a \in R)\ \ (f_1(a) = w_1 \cdot a\ )$$

For $i=2,\ldots,n$, we define

$$f_i : R \times R \longrightarrow R$$

by

$$(\forall a, l \in R)\ \ (f_i(a,l) = l + w_i \,.\, a\ )$$

(Following Chapter 2, the arguments 'a' and 'l' in the above definitions are intended to be mnemonic of 'the value from above' and 'the value from the left' respectively.)

Value Functions. Value functions for the modules of FIR can be immediately obtained from the network of Figure 5.1. Since FIR has n modules each of which holds a value from R, and n sources each supplying elements of R, the FIR network is formalised by the value function

$$V_{FIR} = (V_1, \ldots, V_n) : T \times [T \longrightarrow R^n] \times R^n \longrightarrow R^n$$

where for $i = 1, \ldots, n$, $V_i(t, \underline{a}, x)$ is defined for each $t \in T$, input stream $\underline{a} : T \longrightarrow R^n$, and initial values $x \in R^n$ as follows:

$$V_i(0, \underline{a}, x) = x_i,$$

and

$$V_i(t+1, \underline{a}, x) = \begin{cases} f_1(\underline{a}_1(t)) & \text{if } i = 1 \\ f_i(\underline{a}_i(t), V_{i-1}(t, \underline{a}, x)) & \text{otherwise} \end{cases}$$

that is,

$$V_i(t+1, \underline{a}, x) = \begin{cases} w_1 \,.\, \underline{a}_1(t) & \text{if } i = 1 \\ V_{i-1}(t, \underline{a}, x) + w_i \,.\, \underline{a}_i(t) & \text{otherwise} \end{cases} \qquad (5)$$

We note that $V_{FIR} \in \mathrm{PR}(\underline{A})$ where $A = A_{FIR}$ comprises $T = T_{FIR}$ and $\mathbb{B}$ with their standard operations, together with the set R (as a carrier) and $f_1, \ldots, f_n$ (as operations).

5.1.3 Correctness of FIR.

It is not difficult to prove from the definitions of FIR's value functions that V_n satisfies:

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow R^n)(\forall x \in R^n)\ \ (t \geq n \implies V_n(t, \underline{a}, x) = \sum_{k=1}^{k=n} w_k \,.\, \underline{a}_k(t-n-1+k)\ ) \qquad (6)$$

Note that this formula tells us the value held by m_n for *arbitrary* input streams $\underline{a}$. Now recall from Section 2.4.3 how the output function F_N and the stream transformation G_N were defined from V_N for a general network N. Since FIR has a single sink we have $F_{FIR} = V_n$, thus from (6) we have

$$R(t) \implies F_{FIR}(t, \underline{a}, x) = \sum_{k=1}^{k=n} w_k \,.\, \underline{a}_k(t-n-1+k)$$

for each $t \in T$, $\underline{a} : T \longrightarrow R^n$, and $x \in R^n$. Thus

$$R(t) \implies G_{FIR}(\underline{a}, x)(t) = \sum_{k=1}^{k=n} w_k \,.\, \underline{a}_k(t-n-1+k)$$

for each $\underline{a} : T \longrightarrow R^n$, $x \in R^n$, and $t \in T$, which we can write as

$$G_{FIR}(\underline{a}, x)(t) = \begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}_n(t-1), \ldots, \underline{a}_1(t-n)) & \text{if } R(t) \end{cases} \qquad (7)$$

Comparing (2) with (7) we note that G_{FIR} differs from Φ_N at those times t for which $R(t)$ holds, and so FIR does not meet the required specification (in the sense of Section 2.2.4). However, FIR was developed in the context of staggered input data, and so rather than consider $G_{FIR}(\underline{a}, x)(t)$ for general

$\underline{a} : T \longrightarrow R^n$, it is more appropriate to consider $G_{FIR}(\underline{b},x)(t)$ where $\underline{b}$ is $\underline{a}$ appropriately staggered.

With this last point in mind, let us define a 'staggering operator'

$$\Psi = (\Psi_1,\ldots,\Psi_n) : [T \longrightarrow R^n] \longrightarrow [T \longrightarrow R^n]$$

where for each $\underline{a} : T \longrightarrow R^n$ and $t \in T$,

$$\Psi_i(\underline{a})(t) = \begin{cases} 0 & \text{if } 0 \leq t < i-1 \\ \underline{a}_{n-i+1}(\lambda_i(t)) & \text{if } t \geq i-1 \end{cases}$$

wherein $\lambda_i : T \longrightarrow T$ is the function defined by $\lambda_i(t) = t-i+1$ for each $t \in T$.

For $i = 1,\ldots,n$, $\Psi_i(\underline{a})(t)$ is intended to be the value supplied to m_i at time t. As a quick check, take $n = 5$, $i = 5$, and $t = 6$. Then $t \geq i-1$ and so by definition of Ψ_i and λ_i we have

$$\Psi_i(\underline{a})(t) = \Psi_5(\underline{a})(6) = \underline{a}_{5-5+1}(\lambda_5(6)) = \underline{a}_1(6-5+1) = \underline{a}_1(2)$$

which is confirmed in the figures.

Verification. We now show that FIR meets Φ_N with respect to the staggering operator Ψ.

First notice that for $t \geq n$ we have $t-n \geq 0$ and so $t-n-1+k \geq k-1$ for any $k \in \mathbf{N}$. Thus by definition of Ψ_k for $k = 1,\ldots,n$ we have

$$\Psi_k(\underline{a})(t-n-1+k) = \underline{a}_{n-k+1}(\lambda_k(t-n-1+k)) = \underline{a}_{n-k+1}(t-n-1+k-k+1) = \underline{a}_{n-k+1}(t-n) \qquad (8)$$

Now choose $\underline{a} : T \longrightarrow R^n$, $x \in R^n$, and $t \in T$ with $t \geq n$. Then $R(t)$ holds and so from (7) we have

$$G_{FIR}(\Psi(\underline{a}),x)(t) = \sum_{k=1}^{k=n} w_k \cdot \Psi_k(\underline{a})(t-n-1+k)$$

$$= \sum_{k=1}^{k=n} w_k \cdot \underline{a}_{n-k+1}(t-n)$$

(from (8))

$$= fir(\underline{a}(t-n))$$

(by definition of *fir*).

We have shown that $R(t) \Rightarrow G_{FIR}(\Psi(\underline{a}),x)(t) = fir(\underline{a}(t))$; that is, for each $\underline{a} : T \longrightarrow R^n$ and $x \in R^n$, $G_{FIR}(\Psi(\underline{a}),x)$ is a stream such that for each $t \in T$,

$$G_{FIR}(\Psi(\underline{a}),x)(t) = \begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}(t-n)) & \text{if } R(t) \end{cases}$$

$$= \Phi_N(\underline{a})(t)$$

Since the expression '$\Psi(\underline{a})$' formalises the idea of staggered input data, and since $G_{FIR}(\Psi(\underline{a}),x) = \Phi_N(\underline{a})$ for each $\underline{a} : T \longrightarrow R^n$ and $x \in R^n$, we conclude that N = FIR is a correct implementation.

5.1.4 A Serial Loading Algorithm.

A second synchronous algorithm for solving the finite impulse response filtering problem is depicted in Figure 5.2. Like FIR the network FIR2 of Figure 5.2 comprises n modules $m_1,\ldots,m_n$ and a single sink *Out*, but unlike FIR, FIR2 only has a single source *In*.

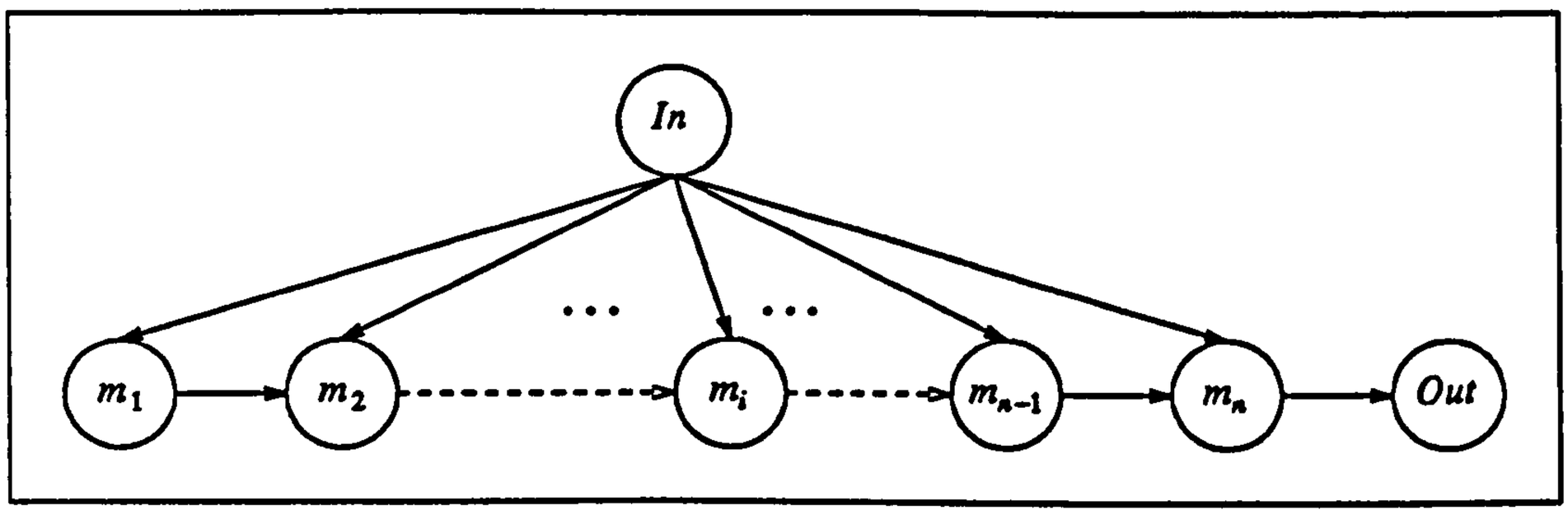


Figure 5.2 - The network FIR2.

Since FIR2 has a single source which supplies (broadcasts) elements of R to every module, we represent the the input to FIR2 as a stream $\underline{a} : T \longrightarrow R$ (with the intention that the value supplied by In at time t is $\underline{a}(t)$).

The FIR2 network operates as follows. Initially, at $t = 0$, we imagine each module m_i of FIR2 to be holding some initial value $x_i \in R$, and to be about to read $\underline{a}(0)$ from the source In.

For $t = 0,1,2,\ldots$, the value held by each m_i by time $t+1$ is calculated as follows for $i = 1,\ldots,n$:

Case $i = 1$. Module m_1 first reads $\underline{a}(t)$ from In and then multiplies this value by the weight w_1.

Case $i = 2,\ldots,n$. Module m_i first reads $\underline{a}(t)$ from In and multiplies this value by the weight w_i. The result of this multiplication is then added to the value held by module m_{i-1} at time t. □

The operation of FIR2 is illustrated for the case $n = 5$ and for $t = 0,1,2,8,19$ in Figures 5.10 - 5.14. The reader is again recommended to scan through these figures before attempting to follow the detailed discussion below. (In the figures, we have again used the notation $y_{r,c}$ for the sum of the first $r > 0$ terms of $fir(\underline{a}(c))$, and in tracing the operation of FIR2, the reader should recall the identity (4): $y_{r+1,c} = y_{r,c} + w_{r+1} \cdot \underline{a}_{n-r}(c)$.)

In order to make FIR correctly compute $fir(\underline{a}(c))$ for $c = 0,1,2,\ldots$, the coordinates $\underline{a}_1(c),\ldots,\underline{a}_n(c)$ of each $\underline{a}(c)$ are be supplied serially and in reverse order by the source; that is, $\underline{a}_n(c)$ enters first, then $\underline{a}_{n-1}(c)$, and so on, until $\underline{a}_1(c)$ enters. Thereafter, the pattern repeats with $\underline{a}(c+1)$: $\underline{a}_n(c+1)$ enters, then $\underline{a}_{n-1}(c+1)$, etc. This is illustrated in Figure 5.10 ($t = 0$) for the case $n = 5$ and $c = 0,\ldots,3$.

Let us now consider in detail how $fir(\underline{a}(0))$ is computed by FIR2 with the given input configuration.

Since the input to m_1 at $t = 0$ is $\underline{a}_n(0)$ (see Figure 5.10), m_1 holds $w_1 \cdot \underline{a}_n(0) = y_{1,0}$ at $t = 1$ (see Figure 5.11). Since this value is the input to m_2 at $t = 1$ together with $\underline{a}_{n-1}(0)$, m_2 holds $w_1 \cdot \underline{a}_n(0) + w_2 \cdot \underline{a}_{n-1}(0) = y_{2,0}$ at time $t = 2$ (see Figure 5.12). It is not difficult to see that by $t = n$, m_n will hold $y_{n,0} = fir(\underline{a}(0))$.

However, unlike FIR, during these first n cycles FIR2 is not simultaneously computing other partial sums $y_{r,c}$. Rather, we must wait until $t=n$ before the first coordinate $\underline{a}_n(1)$ of the next vector $\underline{a}(1)$ appears at the source. It is not difficult to see in this case, that for the particular input configuration of Figure 5.10, m_i holds $y_{i,(t-i)\,div\,n}$ when t is of the form $t=i+kn$ for some $k \in \mathbb{N}$, and hence m_n holds $y_{n,c}=fir(\underline{a}(c))$ at time $t=(c+1)n$.

5.1.5 Formal Specification of FIR2.

We will now formalise the operation of FIR2 according to our specification technique: first we formalise each module by specifying its behaviour by means of a function on R, and then we specify the value held by each module at each time t by means of a value function.

Module Specification. For $i=1,\ldots,n$, the operation of the ith module m_i is specified by means of exactly the same function f_i that served to specify module m_i of FIR (see Section 5.1.2).

Value Functions. Value functions for the modules of FIR2 can be immediately obtained from the network of Figure 5.2. Since FIR2 has n modules each of which holds a value from R, and a single source which suplies elements of R, the FIR2 network is formalised via the value function

$$V_{FIR2}=(V_1,\ldots,V_n): T\times[T\longrightarrow R]\times R^n \longrightarrow R^n$$

where for $i=1,\ldots,n$ $V_i(t,\underline{a},x)$ is defined for each $t\in T$, $\underline{a}: T\longrightarrow R$, and $x\in R^n$ as follows:

$$V_i(0,\underline{a},x)=x_i,$$

and

$$V_i(t+1,\underline{a},x)=\begin{cases} f_1(\underline{a}(t)) & \text{if } i=1 \\ f_i(\underline{a}(t),V_{i-1}(t,\underline{a},x)) & \text{otherwise} \end{cases}$$

that is,

$$V_i(t+1,\underline{a},x)=\begin{cases} w_1\cdot\underline{a}(t)) & \text{if } i=1 \\ V_{i-1}(t,\underline{a},x)+\underline{a}(t) & \text{otherwise} \end{cases} \qquad (9)$$

We note that $V_{FIR2}\in \mathrm{PR}(\underline{A})$ where $A=A_{FIR}$ comprises $T=T_{FIR2}=T_{FIR}$ and $\mathbb{B}$ with their standard operations, together with the set R (as a carrier) and $f_1,\ldots,f_n$ (as operations).

5.1.6 Correctness of FIR2.

What user specification does FIR2 meet? As we have argued above, it is apparent that $y_{n,c}=fir(\underline{a}(c))$ emerges from FIR2 at time $t=(c+1)n$ for $c=0,1,2,\ldots$ (assuming the input $\underline{a}$ is in the necessary serial form). We note that Φ_N asks for the network to have period 1, however it is apparent that FIR2 will have period n (since it takes n steps to supply the n coordinates of each input vector), and thus FIR2 will not meet Φ_N; let us consider another correctness specification instead:

Let $\Phi_N: [T\longrightarrow R^n]\longrightarrow[T\longrightarrow R]$ be defined by

$$\Phi_N(\underline{a})(t)=\begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}(t-n)) & \text{if } R(t) \end{cases} \qquad (10)$$

for each $\underline{a}: T\longrightarrow R^n$ and for each $t\in T$, where $R(t) \Longleftrightarrow t \bmod n=0$ and $t\neq 0$. Then Φ_N says that FIR2 must compute *fir* on every nth input.

It is not difficult to prove from the definitions of FIR2's value functions that V_n satisfies:

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow R)(\forall x \in R^n)\ (t \geq n \implies V_n(t,\underline{a},x) = \sum_{k=1}^{k=n} w_k \cdot \underline{a}(t-n-1+k)) \qquad (11)$$

Furthermore, since FIR2 has a single sink we have $F_{FIR2} = V_n$, and since $R(t) \Longrightarrow t \geq n$, from (11) we have

$$R(t) \implies F_{FIR2}(t,\underline{a},x) = \sum_{k=1}^{k=n} w_k \cdot \underline{a}(t-n-1+k)$$

for each $t \in T$, $\underline{a} : T \longrightarrow R$, and $x \in R^n$. Thus

$$R(t) \implies G_{FIR2}(\underline{a},x)(t) = \sum_{k=1}^{k=n} w_k \cdot \underline{a}(t-n-1+k)$$

which we can write as

$$G_{FIR2}(\underline{a},x)(t) = \begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}(t-1),\ldots,\underline{a}(t-n)) & \text{if } R(t) \end{cases} \qquad (12)$$

Similar to the FIR network, in order to make FIR2 meet Φ_N we must consider FIR2 executing on transformed input data. For a given input stream $\underline{a} : T \longrightarrow R^n$, we need to consider $G_{FIR}(\underline{b},x)(t)$ where $\underline{b}$ is $\underline{a}$ appropriately 'serialised'.

Let us define a 'serialising operator' $\Psi : [T \longrightarrow R^n] \longrightarrow [T \longrightarrow R]$ in the following way: for each $\underline{a} : T \longrightarrow R^n$ define $\Psi(\underline{a})$ by

$$\Psi(\underline{a})(t) = \underline{a}_{n-(t \bmod n)}(\lambda(t))$$

for each $t \in T$, wherein $\lambda : T \longrightarrow T$ is the function defined by $\lambda(t) = t \; div \; n$ for each $t \in T$.

$\Psi(\underline{a})(t)$ is intended to be the value supplied by FIR2's source at time t. As a quick check, take $n = 5$ and $t = kn+3$ for any $k \in \mathbb{N}$. Then by definition of Ψ and λ we have

$$\Psi(\underline{a})(t) = \Psi(\underline{a})(5k+3) = \underline{a}_{5-(5k+3 \bmod 5)}(\lambda(5k+3)) = \underline{a}_{5-3}((5k+3) \; div \; 5) = \underline{a}_2(k)$$

which is confirmed in Figure 5.13 ($t = 8$, $k = 1$).

Verification. We now show that FIR2 meets Φ_N with respect to the serialising operator Ψ.

First notice that if $t = (m+1)n$ for some $m \in \mathbb{N}$ then $t-n-1+k = mn-1+k$ for any $k \in \mathbb{N}$, and for $1 \leq k \leq n$ we see that $t-n-1+k \geq mn$ and $t-n-1+k < (m+1)n$; thus $(t-n-1+k) \; mod \; n = k-1$ and $(t-n-1+k) \; div \; n = m = t-n$. Thus, by definition of Ψ we have

$$\begin{aligned} \Psi(\underline{a})(t-n-1+k) &= \underline{a}_{n-(t-n-1+k) \bmod n}(\lambda(t-n-1+k)) \\ &= a_{n-(k-1)}((t-n-1+k) \; div \; n) \\ &= \underline{a}_{n-k+1}(t-n) \end{aligned} \qquad (13)$$

Now choose $\underline{a} : T \longrightarrow R^n$, $x \in R^n$, and $t \in T$ with $t = (m+1)n$ for some $m \in \mathbb{N}$. Then $R(t)$ holds and so from (12) we have

$$\begin{aligned} G_{FIR2}(\Psi(\underline{a}),x)(t) &= \sum_{k=1}^{k=n} w_k \cdot \Psi(\underline{a})(t-n-1+k) \\ &= \sum_{k=1}^{k=n} w_k \cdot \underline{a}_{n-k+1}(t-n) \end{aligned}$$

(from (13))

$$= fir(\underline{a}(t-n))$$

(by definition of *fir*).

We have now shown that $R(t) \Longrightarrow G_{FIR2}(\Psi(\underline{a}),x)(t) = fir(\underline{a}(t))$; that is, for each $\underline{a} : T \longrightarrow R^n$, $G_{FIR2}(\Psi(\underline{a}),x)$ is a stream such that for each $t \in T$,

$$G_{FIR2}(\Psi(\underline{a}),x)(t) = \begin{cases} u & \text{if } \sim R(t) \\ fir(\underline{a}(t-n)) & \text{if } R(t) \end{cases}$$

$$= \Phi_N(\underline{a})(t)$$

Since the expression '$\Psi(\underline{a})$' formalises the idea of serialised input data, and since $G_{FIR2}(\Psi(\underline{a}),x) = \Phi_N(\underline{a})$ for each $\underline{a} : T \longrightarrow R^n$ and $x \in R^n$, we conclude that N = FIR2 is a correct implementation.

5.2 PALINDROME RECOGNITION.

Let Γ be some finite *alphabet*. We say a vector $a = (a_1,...,a_n) \in \Gamma^n$ (that is, a *word of length n* over Γ) is a *palindrome* if

$$(a_1 = a_n) \wedge \cdots \wedge (a_i = a_{n-i+1}) \wedge \cdots \wedge (a_p = a_{n-p+1})$$

where $p = \left\lceil \frac{n}{2} \right\rceil$ and '$\wedge$' denotes logical conjunction, and '$\lceil k \rceil$' denotes the smallest integer greater than or equal to k.

Let n be even, $n = 2p$ say, for some $p > 0$. Now let $pal : \Gamma^n \longrightarrow \mathbb{B}$ be defined by

$$pal(a) = (a_1 = a_n) \wedge \cdots \wedge (a_i = a_{n-i+1}) \wedge \cdots \wedge (a_p = a_{p+1})$$

for each $a = (a_1,...,a_n) \in \Gamma^n$. Then equivalently, $a \in \Gamma^n$ is a palindrome iff $pal(a) = tt$.

The *palindrome recognition problem for words of length* $n = 2p$ over Γ is to devise a synchronous network N with clock $T = T_N$ that implements the following user specification:

$$\Phi_N : [T \longrightarrow \Gamma^n] \longrightarrow [T \longrightarrow \mathbb{B}]$$

where for each $\underline{a} : T \longrightarrow \Gamma^n$ and for each $t \in T$,

$$\Phi_N(\underline{a})(t) = \begin{cases} u & \text{if } \sim R(t) \\ pal(\underline{a}(t-p)) & \text{if } R(t) \end{cases}$$

wherein $R(t) \Longleftrightarrow t \ mod\ (p-1) = 1$ and $t \neq 1$. Thus the problem is to see if $\underline{a}(c(p-1))$ is a palindrome for each $c \in \mathbb{N}$.

Similar to the previous examples, let us introduce a notation for partially computed results: for $r = 1,...,p-1$ and $c = 0,1,2,...$ define $y_{r,c}$ by

$$y_{r,c} = \bigwedge_{k=p}^{k=p-r} b_{k,c} \qquad (14)$$

(note the reverse order of conjunction) wherein $b_{k,c}$ denotes the result of comparing the the kth and $(n-k+1)$th elements of c th input to be processed; that is,

$$b_{k,c} = (\underline{a}_k(c(p-1)) = \underline{a}_{n-k+1}(c(p-1)))$$

Thus $y_{r,c}$ denotes the conjunction of the *last* $r > 0$ terms of $pal(\underline{a}(c))$. Notice $y_{p-1,c} = pal(\underline{a}(c(p-1)))$

for each $c \in \mathbb{N}$.

5.2.1 The Algorithm.

A synchronous algorithm for implementing Φ_N is depicted in Figure 5.3. The network PAL of Figure 5.3 comprises $n-1$ modules $m_1,\ldots,m_{n-1}$, n sources $In_1,\ldots,In_n$, and a single sink Out.

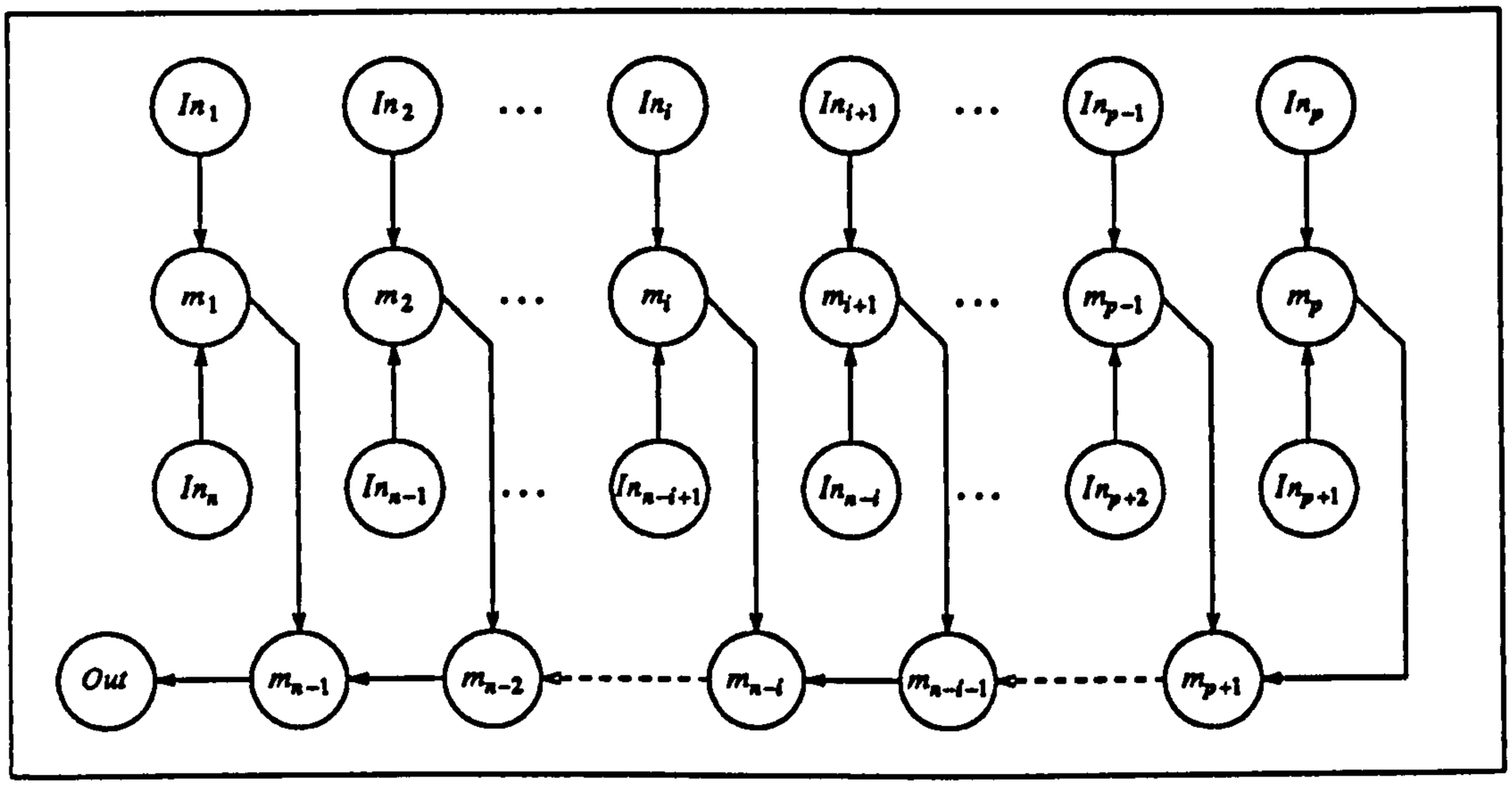


Figure 5.3 - The network PAL.

Each of PAL's n sources supplies elements of Γ, and so we represent the the input to PAL as a stream $\underline{a} : T \longrightarrow \Gamma^n$ (with the intention that for $i = 1,\ldots,n$ the value supplied by In_i at time t is $\underline{a}_i(t)$).

The PAL network operates as follows. Initially, at $t = 0$, we imagine each module m_i of PAL to be holding some value $x_i \in \mathbb{B}$, and to be about to read $\underline{a}_i(0)$ from source In_i for $i = 1,\ldots,n$.

For $t = 0,1,2,\ldots$, the value held by each m_i by time $t+1$ is calculated as follows for $i = 1,\ldots,n$:

Case $1 \leq i \leq p$. If the current time t is an integral multiple of $p-1$, that is if t satisfies $t \; mod \; (p-1) = 0$, then module m_i first reads $\underline{a}_i(t)$ and $\underline{a}_{n-i+1}(t)$ from In_i and In_{n-i+1} respectively. Then m_i compares these values, and by time $t+1$ the module holds either the truth-value *tt* if the values were the same, and the truth-value *ff* if the values were different. If the current time t does *not* satisfy $t \; mod \; (p-1) = 0$, then the module does nothing; that is, it retains its current value for one clock cycle.

Case $i = p+1$. The module m_{p+1} first reads the values held by m_{p-1} and m_p at time t. Then, by time $t+1$, m_{p+1} holds the result of '$\wedge$-ing' these values together.

Case $p+2 \leq i \leq n-1$. In this case module m_i behaves exactly as m_{p+1} above, except that in this case, m_i reads the values held by m_{n-i} and m_{i-1}. □

The operation of PAL is illustrated for the case $n = 10$ and for $t = 0,\ldots,5$ in Figures 5.15 - 5.20. Again the reader is invited to study these figures before attempting to follow the discussion below.

Similar to the FIR and FIR2 networks, in tracing through the figures, it is helpful to bear the following identity in mind (derived from (14)):

$$y_{r+1,c} = y_{r,c} \wedge b_{p-r-1,c} \tag{15}$$

Let us now consider in detail how $pal(\underline{a}(0))$ is computed by the PAL network.

For $i=1,...,p$, the inputs to m_i at $t=0$ are $\underline{a}_i(0)$ and $\underline{a}_{n-i+1}(0)$, and so by time $t=1$ m_i holds $(\underline{a}_i(0)=\underline{a}_{n-i+1}(0))=b_{i,0}$ (see Figure 5.16). Thus at time $t=2$, the inputs to m_{p+1} are $b_{p,0}$ and $b_{p-1,0}$, and hence m_{p+1} holds $b_{p-1,0} \wedge b_{p,0}=y_{1,0}$ by time $t=2$ (see Figure 5.17).

Now, according to the informal description of the modules above, for $i=1,...,p$ each module m_i retains its value ($b_{i,0}$) until $t=p-1$. Thus (in the figures, where $n=10$), at $t=2(<p-1)$ the inputs to m_{p+2} are $y_{1,0}$ and $b_{p-2,0}$, and so from (15), m_{p+2} will hold $y_{1,0} \wedge b_{p-2,0}=y_{2,0}$ by $t=3$ (see Figure 5.18). It is not difficult to see that at time $t=p-1$ the inputs to m_{n-1} will be $y_{p-2,0}$ and $b_{1,0}$ and thus by time $t=p$, m_{p-1} will hold $y_{p-2,0} \wedge b_{1,0}=y_{p-1,0}=pal(\underline{a}(0))$, again using (15). Thus it takes p cycles to determine whether the first input $\underline{a}(0)$ is a palindrome. However, for subsequent inputs $\underline{a}((c+1)(p-1))$, the computation time is only $p-1$ since the comparison of coordinates of $\underline{a}((c+1)(p-1))$ are overlapped with the computation of the last term of $pal(\underline{a}(c(p-1)))$. Thus, generally, $y_{p-1,c}=pal(\underline{a}(c(p-1)))$ is held by m_{n-1} at time $t=1+(c+1)(p-1)$. (See Figure 5.20 for the case $c=0$.)

5.2.2 Formal Specification of PAL.

We will now formalise the operation of PAL according our specification technique: first we formalise each module by specifying its behaviour by means of a function on Γ and $\mathbb{B}$, and then we specify the value held by each module at each time t by means of a value function.

Module Specification. For $i=1,...,n$, the operation of i th module m_i is specified by means of the function f_i defined as follows: first, for $i=1,...,p$, we define

$$f_i : T \times \Gamma \times \mathbb{B} \times \Gamma \longrightarrow \mathbb{B}$$

by

$$f_i(t,a,v,b) = \begin{cases} (a=b) & \text{if } t \ mod\ (p-1)=0 \\ v & \text{otherwise} \end{cases}$$

for each $t \in T$, $a,b \in \Gamma$, and $v \in \mathbb{B}$.

For $i=p+1,...,n-1$, we define

$$f_i : \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$$

by

$$f_i(a,r) = a \wedge r$$

for each $a,r \in \mathbb{B}$.

(Again following Chapter 2, the arguments 'a', 'b','v' and 'r' in the above definitions are intended to be mnemonic of 'the value from above', 'the value from below, 'the value held by the module', and 'the value from the right' respectively.)

Value Functions. Value functions for PAL can be immediately obtained from the network of Figure 5.3. Since PAL has $n-1$ modules each of which holds a Boolean value, and n sources each supplying elements of Γ, the PAL network is formalised via the value function

$$V_{PAL} = (V_1,\ldots,V_{n-1}) : T \times [T \longrightarrow \Gamma^n] \times \mathbb{B}^{n-1} \longrightarrow \mathbb{B}^{n-1}$$

where for $i = 1,\ldots,n-1$, $V_i(t,\underline{a},x)$ is defined for each $t \in T$, $\underline{a} : T \longrightarrow \Gamma^n$, and $x \in \mathbb{B}^{n-1}$ as follows:

$$V_i(0,\underline{a},x) = x_i,$$

and

$$V_i(t+1,\underline{a},x) = \begin{cases} f_i(\underline{a}_i(t), V_i(t,\underline{a},x), \underline{a}_{n-i+1}(t)) & \text{if } 1 \le i \le p \\ f_i(V_{n-i}(t,\underline{a},x), V_{i-1}(t,\underline{a},x)) & \text{if } p+1 \le i \le n-1 \end{cases}$$

Exercise. For what structure A is V_{PAL} simultaneous primitive recursive over $\underline{A}$?

5.2.3 Correctness of PAL.

It is not difficult to prove from the definitions of PAL's value functions that V_{n-1} satisfies:

$$(\forall t \in T)(\forall \underline{a} : T \longrightarrow \Gamma^n)(\forall x \in \mathbb{B}^{n-1}) \ (R(t) \implies V_{n-1}(t,\underline{a},x) = pal(\underline{a}(t-p))) \qquad (16)$$

Furthermore, since PAL has a single sink we have $F_{PAL} = V_{n-1}$, thus from (16) we have

$$R(t) \implies F_{PAL}(t,\underline{a},x) = pal(\underline{a}(t-p))$$

for each $t \in T$, $\underline{a} : T \longrightarrow \Gamma^n$, and $x \in \mathbb{B}^{n-1}$. Thus

$$R(t) \implies G_{PAL}(\underline{a},x)(t) = pal(\underline{a}(t-p))$$

for each $\underline{a} : T \longrightarrow \Gamma^n$, $x \in \mathbb{B}^{n-1}$, and $t \in T$, which we can write as

$$G_{PAL}(\underline{a},x)(t) = \begin{cases} u & \text{if } \neg R(t) \\ pal(\underline{a}(t-p)) & \text{if } R(t) \end{cases}$$

$$= \Phi_N(\underline{a})(t)$$

Since G_{PAL} and Φ_N agree on all inputs we conclude that $N = \text{PAL}$ is a correct implementation.

5.3 MATRIX-VECTOR MULTIPLICATION.

Let R be a ring with 0, and let $M(n,R)$ be the collection of all $n \times n$ matrices over R. Let $M \in M(n,R)$. Then we write $M_{p,q} \in R$ for the q th element on th p th row of M for $1 \le p,q \le n$.

Given $M \in M(n,R)$ and $a \in R^n$, the *matrix-vector product* $y = mult(M,a)$ is a vector $y = (y_1,\ldots,y_n) \in R^n$ such that for $i = 1,\ldots,n$,

$$y_i = \sum_{k=1}^{k=n} M_{i,k} \cdot a_k \qquad (17)$$

Now let A be the algebra comprising the algebras R and $M(n,R)$ together. The *matrix-vector multiplication problem* over A is to devise a synchronous network N with clock $T = T_N$ that implements the following user specification:

$$\Phi_N : [T \longrightarrow M(n,R)] \times [T \longrightarrow R^n] \longrightarrow [T \longrightarrow R^n]$$

where for each $\underline{M} : T \longrightarrow M(n,R)$, $\underline{a} : T \longrightarrow R^n$, and $t \in T$,

$$\Phi_N(\underline{M},\underline{a})(t) = \begin{cases} u & \text{if } \neg R(t) \\ mult(\underline{M}(\delta(t)),\underline{a}(\delta(t))) & \text{if } R(t) \end{cases}$$

where $R(t) \Leftrightarrow t \bmod 2n = 0$ and $t \neq 0$, and $\delta(t) = (t-2n) \operatorname{div} 2n$ for each $t \in T$. (The ready condition R should not be confused with the ring R of course.) Thus the problem is to compute *mult* on every pair $(\underline{M}(c),\underline{a}(c))$ for each $c \in \mathbf{N}$.

Similar to previous examples, let us introduce a notation for partial sums: for $1 \leq i,j \leq n$, and for $c = 0,1,2,\ldots$, let $y^c_{ij} \in R$ be defined by

$$y^c_{ij} = \sum_{k=1}^{k=j} \underline{M}_{i,k}(c) \cdot \underline{a}_k(c) \qquad (18)$$

Then y^c_{ij} is the sum of the first j terms in calculating the ith coordinate of $mult(\underline{M}(c),\underline{a}(c))$ since $y^c_i = y^c_{i,n}$ (cf. (17) above).

5.3.1 The Algorithm.

A synchronous algorithm for implementing Φ_N is depicted in Figure 5.4. The network MV of Figure 5.4 comprises $2n-1$ modules $m_{1-n},\ldots,m_0,\ldots,m_{n-1}$, $2n$ sources $In_{1-n},\ldots,In_0,\ldots,In_{n-1}$ and In_n, and n sinks $Out_1,\ldots,Out_n$. The network is the most complex we have seen so far, and so we will describe it in some detail:

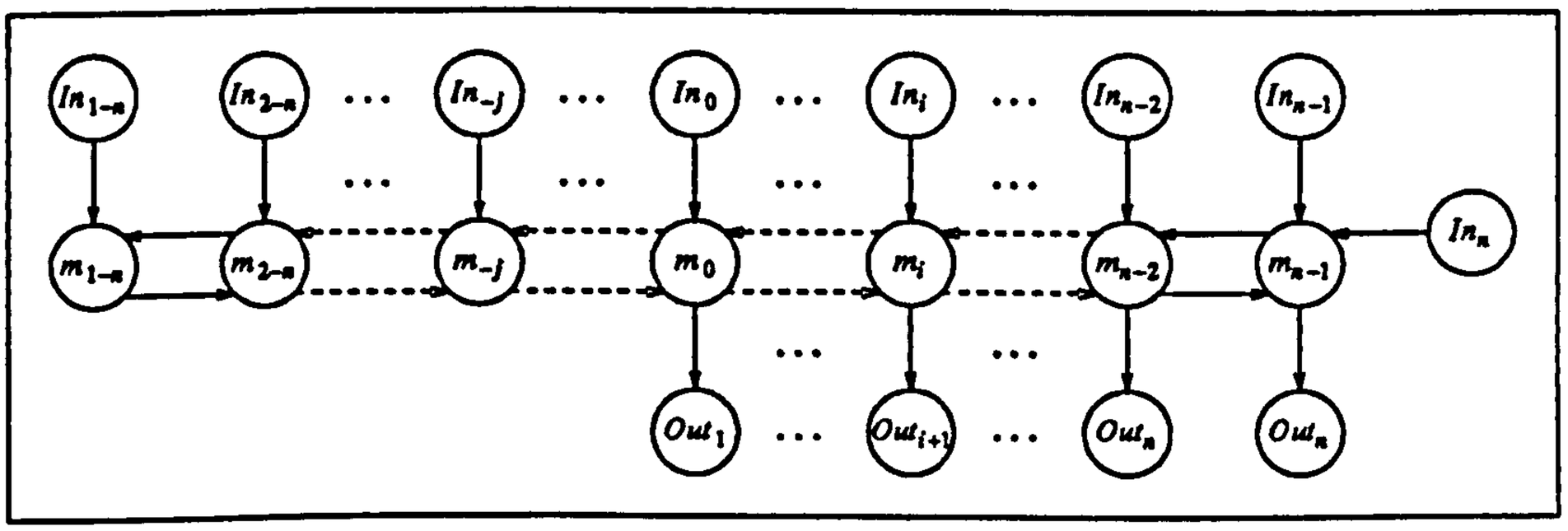


Figure 5.4 - The network MV.

For $i = 2-n,\ldots,n-2$, each module m_i of MV has three inputs: one from the source In_i; one from the module to its left (m_{i-1}), and one from the module to its right (m_{i+1}). Additionally, m_i has two outputs: one going left and one going right (to m_{i-1} and m_{i+1} respectively). (In Figure 5.4, it appears that for $i = 0,\ldots,n-1$, m_i has a third output to the sink Out_{i+1}, but in fact, this output channel is intended to be a branch of the right-going channel, and so these modules do only have two outputs.)

Module m_{1-n} has two inputs: one from the source In_{1-n}, and one from m_{2-n}. Additionally, the module has two outputs: one going right to m_{2-n}, and one going left. The latter output is simply lost from the network, and so we have not drawn a left-going channel from m_{1-n} in the figure.

Module m_{n-1} has three inputs: one from source In_{n-1}; one from In_n, and one from m_{n-2}. Additionally, the module has two outputs: one going left to m_{n-2}, and one going right. The right-going output from m_{n-1} is sent to Out_n.

Each module m_i thus holds *two* values, one value for each output. At first sight this would appear to contradict the idea that modules only hold a single datum However, this is not the case as we will explain later when we formalise the MV algorithm.

Let us consider how MV operates in the context of an example. In Figures 5.21 - 5.27, we have illustrated the case $n=6$ for $t=0,5,6,13,17,18,22$. (Throughout the figures '.' represents 'don't care'.)

Consider Figure 5.21. Here we see the initial input configuration for the input data $\underline{M}(0)$ and $\underline{a}(0)$. Generally, each matrix $\underline{M}(c)$ is supplied in the 'diamond' configuration illustrated in Figure 5.21 for $c=0$. Notice (in subsequent figures) how the next matrix $\underline{M}(c+1)$ follows behind $\underline{M}(c)$ in the same configuration. The coordinates of each input vector $\underline{a}(c)$ enter the network serially from the right, that is, like the input to the FIR2 network, they are supplied by a single source (In_n), and they enter in the order $\underline{a}_1(c),\underline{a}_2(c),....,\underline{a}_n(c)$, but in this case, the coordinates need a padding element in between each (to slow the stream by a factor of two). Notice how the coordinates of the next vector $\underline{a}(c+1)$ follow behind the last coordinate of $\underline{a}(c)$ with an additional padding element between $\underline{a}_n(c)$ and $\underline{a}_1(c+1)$.

We will now describe how the modules of MV operate: we will describe the values sent out on a module's left- and right-going channels separately; the left-going values are simplest to describe, and we consider them first:

For each module m_i, a value entering from the right is simply propagated left to m_{i-1} in one time step. Thus, as illustrated in the figures, the stream of vector data marches across the network until it is thrown away by m_{1-n}.

Now let us focus on the right-going values in the MV network: this is where the coordinates of the required matrix-vector products ultimately appear.

At each time t, the right-going value held by each module m_i by time $t+1$ is calculated as follows for $i=1-n,....,n-1$:

Case $i=1-n$. The right-going value held by module m_{1-n} is 's .x' where s is the the value supplied by source In_{1-n} at time t, and x is the value supplied by m_{2-n} at time t.

Case $2-n \le i \le n-2$. Generally, the right-going value held by module m_i is 's .$x+y$' where s is the the value supplied by source In_i at time t, x is the value supplied by m_{i+1} at time t, and y is the value supplied by m_{i-1} at time t. Exceptionally, if $i \le 0$ and $t=n-1-i+2nk$ for some $k \in \mathbb{N}$, then the right-going value held by m_i at time $t+1$ is 's .x' where s and x are as before. (The reason why m_i does not add in the left-coming value when t is of the form $n-1-i+2k$ will be explained below.)

Case $i=n-1$. The right-going value held by module m_{n-1} is 's .$x+y$' where s is the the value supplied by source In_{n-1} at time t, x is the value supplied by In_n at time t, and y is the value supplied by m_{n-2} at time t.

Let us now consider how MV begins to compute $\underline{M}(0) . \underline{a}(0)$ in detail.

First, there is an initialisation period where very little happens in the MV network: for the first $n-2$ cycles, the left-going values slowly begin to fill up with elements of the vector input and (intuitively), the matrix data gets closer to the network. Also, because the sources $In_{1-n},...,In_{n-1}$ are supplying 'junk' to the network for first $n-2$ cycles (that is, we don't care what is supplied), and because the network's initial values are also junk, all right-going values are also junk during this period.

At $t=n-1$ (see Figure 5.22), the network is initialised and ready to begin computing elements of the required matrix-vector product.

Notice that $\underline{M}_{1,1}(0)$ and $\underline{a}_1(0)$ are now available to m_0. We want m_0 to compute $\underline{M}_{1,1}(0) . \underline{a}_1(0) = y^0_{1,1}$ in the next time cycle as this is the first term of the first coordinate of $\underline{M}(0) . \underline{a}(0)$. Now recall the 'exceptional' clause in the informal specification of MV's modules above (within the case: $2-n \le i \le n-2$). If we had not included this exception, m_0 would compute $\underline{M}_{1,1}(0) . \underline{a}_1(0)$, and then attempt to add in the junk value coming from left, which would intuitively corrupt the result. Now notice here $i=0$ and $t=n-1$ and so $t=n-i-1+2k$ (for $k=0$). Thus the exception comes into play, and m_0 does not perform the unwanted addition; hence $y^0_{1,1}$ is correctly computed (see Figure 5.23).

During the next $n-1$ cycles, $\underline{a}_1(0)$, as it moves from module to module, meets $\underline{M}_{2,1}(0),..., \underline{M}_{n,1}(0)$ at modules $m_{-1},..., m_{1-n}$ respectively. To be precise, for $r=2,..., n$, $\underline{a}_1(0)$ and $\underline{M}_{r,1}(0)$ are simultaneous inputs to module m_{1-r} at time $t=n+r-2$, and these inputs will yield $\underline{M}_{r,1}(0) . \underline{a}_1(0) = y^0_{r,1}$ *as long as we do not add in the left-coming (junk) value*; each $y^0_{r,1}$ is the first term in the rth coordinate of $\underline{M}(0) . \underline{a}(0)$ of course (by definition of the notation $y^c_{i,j}$; see (18)). Exactly as we have said above for m_0, m_{1-r} must not add in the left-coming value at this time $(n+r-2)$ for fear of corrupting the required product. However, when $i=1-r$ and $t=n+r-2$, we have $t=n-1-i$, and so the exception clause comes into effect; thus the terms $y^0_{r,1}$ are indeed correctly computed.

In fact, a similar situation arises $2n$ time units later (see Figure 5.25 where $t=17=6-1+0+2.6$): for $i=1,..., n$, at time $t=n-i-1+2n$, m_{1-i} will be about to compute $\underline{M}_{i,1}(1) . \underline{a}_1(1) = y^1_{i,1}$, and again the module must be prevented from adding in the junk value that is supplied from the left. Generally, m_{1-i} is about to compute $y^c_{i,1}$ at time $t=n-i-1+2nc$, and since there will always be junk coming in from the left at this time, we must prohibit the module from performing the unwanted addition every $2n$ time units after $t=n-i-1$ (hence the formula '$t=n-i+1+2nk$').

At the time $y^0_{r,1}$ appears as a right-going value it is the input to the next module together with exactly the right $\underline{a}$- and $\underline{M}$- values for the next module to compute $y^0_{r,2}$, and at the next time-step $y^0_{r,2}$ meets the right values to compute $y^0_{r,3}$, and so on, until $y^0_{i,n} = y_i{}^0$ appears. Generally, a new value $y^c_{i,j+1}$ is computed from $y^c_{i,j}$ according to the the identity (derived from (18)):

$$y^c_{i,j+1} = y^c_{i,j} + \underline{M}_{i,j+1}(c) . \underline{a}_{j+1}(c)$$

(for $i=1,..., n$, $j=1,..., n-1$, and $c=0,1,2,...$).

For example, in the figures, where $n=6$, at $t=22$ the inputs to m_1 are $y^1_{3,3}$, $\underline{M}_{3,4}(1)$, and $\underline{a}_4(1)$, and so by time $t=23$, m_1 holds

$$y^1_{3,3} + \underline{M}_{3,4}(1) \cdot \underline{a}_4(1) = y^1_{3,3+1} = y^1_{3,4}$$

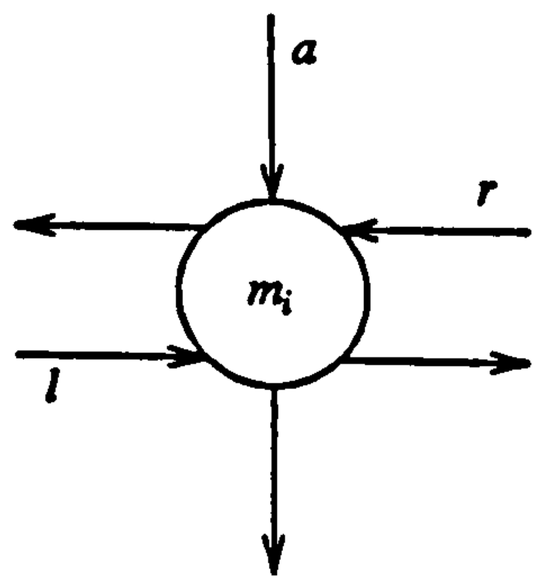
To be more precise, it is apparent (from the figures) that $y^0_{i,n}$ is the right-going value held by module m_{n-i} $n+i-1$ time units after the initialisation time $n-1$; that is, m_{n-i} holds $y_i^0 = y^0_{n,i}$ at $t = (n-1)+(n+i-1) = 2n+i-2$. Now, y_i^0 is the ith coordinate of the required product $\underline{M}(0) \cdot \underline{a}(0)$, and so we anticipate y_i^0 at sink Out_{n-i+1} at time $t = 2n+i-2$ (assuming zero propagation delay to the sinks). (Our placement of the sinks is a necessary one, for we cannot let the values y_i^0 propagate rightwards along the array and read them off at the end, since they will meet elements of the *next* vector $\underline{a}(1)$, and thereby be corrupted. For example, in the figures, at $t = 13$, $y^0_{3,6} = y^0_3$ meets $\underline{a}_1(1)$.)

Generally, for $c = 0,1,2,\ldots$, there is a delay of $2n$ time units between $\underline{M}_{1,1}(c)$ and $\underline{M}_{1,1}(c+1)$ entering the network, and also between $\underline{a}_1(c)$ and $\underline{a}_1(c+1)$, and thus it seems reasonable to speculate that the period of MV is $2n$; that is, y_i^c appears at m_{n-i} at time $2nc+2n+i-2 = 2n(c+1)+i-2$.

Formal Specification of MV. Formal specification of the MV network is not quite as straightforward as it was for previous networks; this is because modules of MV apparently hold a *pair* of values.

In order to not contradict the 'single-valuedness' condition of Chapter 2, we will admit 'pairs of R-data' as a new *sort* of data; that is, the underlying sort set involves some symbol P, say, to name the sort of data 'pair', and the Cartesian product $P = (R \times R)$ is adjoined to A as the carrier of sort P. In this way, a pair of values from R can be regarded as a single datum, that is, a datum from the set P.

Now, in isolation from the MV network, a nonterminal module m_i is pictured thus:



and so, following Chapter 2, we would like to say that m_i is specified by the function:

$$f_i : R \times R \times R \longrightarrow P \qquad (19)$$

defined by

$$f_i(a,l,r) = (r\ ,l+a\ .r\ )$$

for each $a,l,r \in R$. However, this causes the following problem when we try to write down a value function for m_i:

According to our specification technique the value function for m_i (V_i) must have the following functionality:

$$V_i : T \times [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \times P^{2n-1} \longrightarrow P$$

since there are $2n-1$ modules each of which holds a pair, and $2n-1+1$ sources supplying elements of R. Now, given input streams $\underline{m} : T \longrightarrow R^{2n-1}$ and $\underline{a} : T \longrightarrow R$, and a vector of initial (pair-) values $x \in P^{2n-1}$,

there is no difficulty specifying m_i at $t=0$, for this is simply

$$V_i(0,\underline{m},\underline{a},x) = x_i$$

Now consider what happens when we try to write down m_i's value at time $t+1$. We must have:

$$V_i(t+1,\underline{m},\underline{a},x) = f_i(b_1,b_2,b_3)$$

where b_1, b_2, and b_3 are the first, second, and third inputs to m_i at time t respectively. Again there is no difficulty with b_1, for the first input to m_i is from In_i, and so $b_1 = \underline{m}_i(t)$. However, the second input to m_2 must be 'the' value supplied by m_{i+1}, but this is a *pair* of data from R, whereas the second argument to f_i is supposed to be a single *datum* from R (see (19)). (Similar remarks apply to b_3 as well.)

To rectify this argument-type 'mismatch', we must think of m_i as actually depending on both the left- *and* right-going values from *each* neighbouring module, that is, f_i must have the following functionality:

$$f_i : R \times P \times P \longrightarrow P$$

Now, for any $p \in P$ let p^L and p^R denote the first and second coordinates of p respectively (thus $p^L, p^R \in R$, and of course, the superscripts 'L' and 'R' are intended to be mnemonic of 'left' and 'right' respectively). Now let $a \in R$, and let $l, r \in P$, that is, let l and r be *pairs* of R-data. Then we can define f_i by

$$f_i(a,l,r) = (r^R, l^L + a \,.\, r^R)$$

and we can now define V_i at time $t+1$ by

$$V_i(t+1,\underline{m},\underline{a},x) = f_i(\underline{m}_i(t), V_{i-1}(t,\underline{m},\underline{a},x), V_{i+1}(t,\underline{m},\underline{a},x))$$

The full formal specification of MV now follows:

Module Specification. For $i = 1-n,\ldots,n-1$, the operation of the ith module m_i is specified by the function f_i defined as follows: first, we define

$$f_{1-n} : R \times P \longrightarrow P$$

by

$$f_{1-n}(a,r) = (r^R, a \,.\, r^R)$$

for each $a \in R$ and $r \in P$.

For $i = 2-n,\ldots,n-2$, we define

$$f_i : T \times R \times P \times P \longrightarrow P$$

by

$$f_i(t,a,l,r) = \begin{cases} (r^R, l^L + a \,.\, r^R) & \text{if } i > 0 \text{ or } i \leq 0 \text{ and } (t-n+i+1) \bmod 2n \neq 0 \\ (r^R, a \,.\, r^R) & \text{if } i \leq 0 \text{ and } (t-n+i+1) \bmod 2n = 0 \end{cases}$$

for each $t \in T$, $a \in R$ and $l, r \in P$.

Finally, for $i = n-1$, we define

$$f_{n-1} : R \times P \times R \longrightarrow P$$

by:

$$(\forall a, r \in R)(\forall l \in P)\ \ (f_{n-1}(a,l,r) = (r, l^L + a \,.\, r))$$

(Notice that in the last case of m_{n-1}, the value from the right ('r') is supplied by In_n, which supplies data

from R and not pairs of data.)

Value Functions. Since MV has $2n-1$ modules each of which holds a value from P, and $2n-1+1$ sources each supplying elements of R, the MV network is formalised via the value function

$$V_{MV} = (V_{1-n},\ldots,V_{n-1}) : T \times [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \times P^{2n-1} \longrightarrow P^{2n-1}$$

where for $i = 1-n,\ldots,n-1$, $V_i(t,\underline{m},\underline{a},x)$ is defined for each $t \in T$, $\underline{m} : T \longrightarrow R^{2n-1}$, $\underline{a} : T \longrightarrow R$, and $x \in P^{2n-1}$ by

$$V_i(0,\underline{a},x) = x_i,$$

and for each $t \in T$,

$$V_{1-n}(t+1,\underline{a},x) = f_{1-n}(\underline{m}_{1-n}(t), V_{2-n}(t,\underline{m},\underline{a},x))$$

and for $i = 2-n,\ldots,n-2$,

$$V_i(t+1,\underline{m},\underline{a},x) = f_i(\underline{m}_i(t), V_{i-1}(t,\underline{m},\underline{a},x), V_{i+1}(t,\underline{m},\underline{a},x))$$

and

$$V_{n-1}(t+1,\underline{a},x) = f_{n-1}(\underline{m}_{n-1}(t), V_{n-2}(t,\underline{m},\underline{a},x), \underline{a}(t))$$

We note that $V_{MV} \in \mathrm{PR}(\underline{A})$ where $A = A_{MV}$ comprises $T = T_{MV}$ and $\mathbb{B}$ with their standard operations, together with the sets R and P (as carriers) and $f_{1-n},\ldots,f_{n-1}$ (as operations).

5.3.2 Correctness of MV.

For $j = 1,\ldots,n$ define R_j by $R_j(t) \Longleftrightarrow t \bmod 2n = j-2$ and $t \neq j-2$ for each $t \in T$. Also let '$V_i^R(\cdots)$' denote the second coordinate of $V_i(\cdots)$ for $i = 1-n,\ldots,n-1$. Then with some work, it is possible to show that for $j = 1,\ldots,n$, V_{n-j}^R satisfies:

$$R_j(t) \implies V_{n-j}^R(t,\underline{m},\underline{b},x) = \sum_{k=1}^{k=n} \underline{m}_{k-j}(t-n+k-1) \cdot \underline{b}(t-2n-j+2k) \qquad (20)$$

for each $t \in T$, $\underline{m} : T \longrightarrow R^{2n-1}$, $\underline{b} : T \longrightarrow R$, and $x \in P^{2n-1}$.

We will now use (20) to prove correctness properties of F_{MV} and G_{MV}. Let us first introduce functions

$$g_0,\ldots,g_{n-1} : T \times [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \longrightarrow R$$

where for $j = 1,\ldots,n$,

$$g_{n-j}(t,\underline{m},\underline{b}) = \sum_{k=1}^{k=n} \underline{m}_{k-j}(t-n+k-1) \cdot \underline{b}(t-2n-j+2k) \qquad (21)$$

for each $t \in T$, $\underline{m} : T \longrightarrow R^{2n-1}$, and $\underline{b} : T \longrightarrow R$. Then (20) can be more compactly expressed as

$$R_j(t) \implies V_j^R(t,\underline{m},\underline{b},x) = g_{n-j}(t,\underline{m},\underline{b}) \qquad (22)$$

for $j = 1,\ldots,n$.

Now let us consider the definition of F_{MV}. At first sight it would appear that F_{MV} should have functionality

$$F_{MV} : T \times [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \times P^{2n-1} \longrightarrow R^n$$

(since MV has n sinks) and should be defined by

$$F_{MV}(t,\underline{m},\underline{b},x) = (V_0^R(t,\underline{m},\underline{b},x),\ldots,V_{n-1}^R(t,\underline{m},\underline{b},x))$$

since for $j = 1,\ldots,n$ Out_j is the right-going output of m_{j-1}. However, this is officially incorrect: according to the methodology of Section 2.4.3 the jth coordinate of $F_{MV}(t,\underline{m},\underline{b},x)$ should be 'the' value supplied

by m_{j-1} which is pair of R-data, not a single datum. The correct definition of F_{MV} is

$$F_{MV} = (F_1,...,F_n): T \times [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \times P^{2n-1} \longrightarrow P^n$$

where for $j = 1,...,n$, $F_j(t,\underline{m},\underline{b},x) = V_{j-1}(t,\underline{m},\underline{b},x)$ for each $t \in T$, $\underline{m}: T \longrightarrow R^{2n-1}$, $\underline{b}: T \longrightarrow R$, and $x \in P^{2n-1}$.

Now, for $j = 1,...,n$, by definition of F_j we have

$$\begin{aligned} R_j(t) \implies F_{n-j+1}(t,\underline{m},\underline{b},x) &= V_{n-j}(t,\underline{m},\underline{b},x) \\ &= (V^L_{n-j}(t,\underline{m},\underline{b},x), V^R_{n-j}(t,\underline{m},\underline{b},x)) \\ &= (V^L_{n-j}(t,\underline{m},\underline{b},x), g_{n-j}(t,\underline{m},\underline{b})) \end{aligned} \tag{23}$$

(using (22)). Thus

$$G_{MV} = (G_1,...,G_n): [T \longrightarrow R^{2n-1}] \times [T \longrightarrow R] \longrightarrow [T \longrightarrow P^n]$$

satisfies

$$R_j(t) \implies G_{n-j+1}(\underline{m},\underline{b},x)(t) = (V^L_{n-j}(t,\underline{m},\underline{b},x), g_{n-j}(t,\underline{m},\underline{b})) \tag{24}$$

for $j = 1,...,n$ (for each $t \in T$, $\underline{m}: T \longrightarrow R^{2n-1}$, $\underline{b}: T \longrightarrow R$, and $x \in P^{2n-1}$).

We will eventually use (24) to establish the correctness of MV. However, like the FIR and FIR2 networks we must introduce operators which transform input matrix- and vector-streams into the configuration required by MV.

An operator $\Psi: [T \longrightarrow R^n] \longrightarrow [T \longrightarrow R]$ to transform a stream $\underline{a}: T \longrightarrow R^n$ into the necessary serial form can be defined as follows:

$$\Psi(\underline{a})(t) = \begin{cases} 0 & \text{if } t \text{ odd} \\ \underline{a}_i(\lambda(t)) & \text{if } t \text{ even} \end{cases} \tag{25}$$

for each $\underline{a}: T \longrightarrow R^n$ and $t \in T$, wherein $i = 1 + \dfrac{t \bmod 2n}{2}$ and $\lambda(t) = t \operatorname{div} 2n$ for each $t \in T$. As an example of Ψ, take $n = 6$ and $t = 18$, then t is even and

$$\Psi(\underline{a})(t) = \Psi(\underline{a})(18) = \underline{a}_{1+\frac{18 \bmod 2n}{2}}(\lambda(18)) = \underline{a}_4(1)$$

which is confirmed in Figure 5.26 for example.

Now let us define a transformation Θ to 'serialise' a stream of matrices. From this point onwards we will assume n is even; analysis for the case that n is odd must be done separately but is very similar to the case that n is even. Define

$$\Theta = (\Theta_{1-n},...,\Theta_{n-1}): [T \longrightarrow M(n,R)] \longrightarrow [T \longrightarrow R^{2n-1}]$$

by

$$\Theta_i(\underline{M})(t) = \begin{cases} 0 & \text{if } \sim S_i(t) \\ \underline{M}_{p,q}(\lambda'(t)) & \text{if } S_i(t) \end{cases} \tag{26}$$

for each $\underline{M}: C \longrightarrow M(n,R)$ and $t \in T$ for $i = 1-n,...,n-1$. Here S_i, p and q (for $i = 1-n,...,n-1$), and λ' are respectively defined by:

$$S_i(t) \iff i+t \text{ odd} \text{ and } |i| \le (t-n+1) \bmod 2n \le |i| + 2(n-|i|) - 1$$

for each $t \in T$, and

$$p = 1+\frac{-i+(t-n+1)\ mod\ 2n}{2}$$

and

$$q = 1+\frac{i+(t-n+1)\ mod\ 2n}{2}$$

and $\lambda'(t)=(t-n+1)\ div\ 2n$ for each $t \in T$. As an example, take $n=6$, $i=-3$, and $t=22$. Then $i+t$ is odd, and

$$|i| = |-3| = 3 \leq 5 = (22-6+1)\ mod\ 12 = (t-n+1)\ mod\ 2n$$

and

$$(t-n+1)\ mod\ 2n = 5 \leq 8 = |-3|+2(6-|3|)-1 = |i|+2(n-|i|)-1$$

and thus $S_{-3}(22)$ holds. Furthermore, we calculate:

$$p = 1+\frac{-i+(t-n+1)\ mod\ 2n}{2} = 1+\frac{3+(22-6+1)\ mod\ 12}{2} = 1+\frac{3+5}{2} = 5$$

and

$$q = 1+\frac{i+(t-n+1)\ mod\ 2n}{2} = 1+\frac{-3+(22-6+1)\ mod\ 12}{2} = 1+\frac{-3+5}{2} = 2$$

and

$$\lambda'(t) = (22-6+1)\ div\ 12 = 17\ div\ 12 = 1$$

thus $\Theta_{-3}(\underline{M})(22)=\underline{M}_{5,2}(1)$ which is confirmed in Figure 5.27.

Before we use Ψ and Θ in verifying MV, notice that Φ_N says that all the coordinates of each matrix-vector product are to emerge from the network at the same time (namely $\delta(t)$), whereas we know that MV produces the coordinates at different times. To be more precise, as we have argued above , the jth coordinate of $mult(\underline{M}(c),\underline{a}(c))$ namely $y_j^c=y_{j,n}^c$ emerges from m_{n-j} at time $t=2n(c+1)+j-2$ for each $c \in \mathbb{N}$. Thus y_1^c always emerges first, and then y_2^c one step later, and then y_3^c, and so on, until y_n^c emerges n steps after the first coordinate y_1^c. In order to make the coordinates appear to emerge simultaneously at $t=2n(c+1)$ we define

$$\Xi = (\Xi_1,\ldots,\Xi_n):[T\longrightarrow P^n]\longrightarrow[T\longrightarrow R^n]$$
$$\Xi_j(\underline{b})(t) = \underline{b}^R_{n-j+1}(t+j-2)$$

for each $\underline{b}=(\underline{b}^L,\underline{b}^R):T\longrightarrow P^n$ and $t \in T$ for $j=1,\ldots,n$. (Note that Ξ additionally projects out only 'right-going' values, and reverses the order of its inputs.)

We anticipate that Ξ will 'unstagger' the output of MV when executed on streams $\Theta(\underline{M})$ and $\Psi(\underline{a})$; that is, we expect

$$R(t) \Longrightarrow \Xi(G_{MV}(\Theta(\underline{M}),\Psi(\underline{a}),x))(t) = \Phi_N(\underline{M},\underline{a})(t) \qquad (27)$$

for each $\underline{M}:T\longrightarrow M(n,R)$, $\underline{a}:T\longrightarrow R^n$, $x \in P^{2n-1}$, and $t \in T$.

Verification. We will now formally establish that $N=$MV is a correct implementation by proving (27).

Choose $\underline{M}:T\longrightarrow M(n,R)$, $\underline{a}:T\longrightarrow R^n$, and $t \in T$. Suppose $R(t)$ holds. Then t must be of the form $t=2n(c+1)$ for some $c \in \mathbb{N}$, and so $t+j-2=2n(c+1)+j-2$ for any j, and thus $R_j(t+j-2)$ holds for $j=1,\ldots,n$. Thus from (24) we have

$$G^R_{n-j+1}(\Theta(\underline{M}),\Psi(\underline{a}),x)(t+j-2) = g_{n-j}(t+j-2,\Theta(\underline{M}),\Psi(\underline{a}))$$

$$= \sum_{k=1}^{k=n} \Theta_{k-j}(\underline{M})(t+j-2-n+k-1) \cdot \Psi(\underline{a})(t+j-2-2n-j+2k)$$

(from (21))

$$= \sum_{k=1}^{k=n} \Theta_{k-j}(\underline{M})(t+j-n+k-3) \cdot \Psi(\underline{a})(t-2-2n+2k)$$

$$= \sum_{k=1}^{k=n} \Theta_{k-j}(\underline{M})(2nc+j+n+k-3) \cdot \Psi(\underline{a})(2nc+2k-2) \qquad (28)$$

where $c = \delta(t)$. (Notice that $R(t) \Rightarrow t = 2n(c+1)$ for some $c \in \mathbb{N}$.)

Now consider $\Psi(\underline{a})(2nc+2k-2)$. The number $2nc+2k-2$ is obviously even, and furthermore, $(2nc+2k-2)\ mod\ 2n = 2k-2$ and $(2nc+2k-2)\ div\ 2n = c$ (since $1 \le k \le n$). Thus from (25) we have

$$\Psi(\underline{a})(2nc+2k-2) = \underline{a}_{1+\frac{2k-2}{2}}(c) = \underline{a}_k(c) \qquad (29)$$

Now consider $\Theta_{k-j}(\underline{M})(2nc+j+n+k-3)$. We claim that $S_{k-j}(2nc+j+n+k-3)$ holds: first notice that $(k-j)+(2nc+j+n+k-3) = 2nc+n+2k-3$ which is odd since n is even by hypothesis; thus the first part of the condition holds, the remaining parts also hold but we leave proof of this to the reader.

Since $S_{k-j}(2nc+j+n+k-3)$ holds, from (26) we have

$$\Theta_{k-j}(\underline{M})(2nc+j+n+k-3) = \underline{M}_{p,q}(\lambda'(2nc+j+n+k-3))$$

where

$$p = 1 + \frac{-(k-j)+((2nc+j+n+k-3)-(n-1))\ mod\ 2n}{2}$$

that is,

$$p = 1 + \frac{-(k-j)+(2nc+j+k-2)\ mod\ 2n}{2}$$

and

$$q = 1 + \frac{(k-j)+((2nc+j+n+k-3)-(n-1))\ mod\ 2n}{2}$$

that is,

$$q = 1 + \frac{(k-j)+(2nc+j+k-2)\ mod\ 2n}{2}$$

Now, since $1 \le j,k \le n$ we have

$$2nc \le 2nc+j+k-2 \le 2nc+2n-2$$

thus $(2nc+j+k-2)\ mod\ 2n = j+k-2$, and $(2nc+j+k-2)\ div\ 2n = c$. Thus $p = j$, $q = k$, and $\lambda'(2nc+j+n+k-3) = c$. Thus $\Theta_{k-j}(\underline{M})(2nc+j+n+k-3) = \underline{M}_{j,k}(c)$.

Substituting $\Theta_{k-j}(\underline{M})(2nc+j+n+k-3) = \underline{M}_{j,k}(c)$ and $\Psi(\underline{a})(2nc+2k-2) = \underline{a}_k(c)$ in (28) yields

$$G^R_{n-j+1}(\Theta(\underline{M}),\Psi(\underline{a}),x)(t+j-2) = \sum_{k=1}^{k=n} \underline{M}_{j,k}(c) \cdot \underline{a}_k(c)$$

$$= y_j^c \qquad (30)$$

Now, by the coordinatewise definition of Ξ we have

$$\Xi(G_{MV}(\Theta(\underline{M}),\Psi(\underline{a}),x))(t) = (\Xi_1(G_{MV}(\Theta(\underline{M}),\Psi(\underline{a}),x))(t),\ldots,\Xi_n(G_{MV}(\Theta(\underline{M}),\Psi(\underline{a}),x))(t))$$

$$= (G_n^R(\Theta(\underline{M}),\Psi(\underline{a}),x)(t-1),\ldots,G_1^R(\Theta(\underline{M}),\Psi(\underline{a}),x)(t-n))$$
$$= (y_1^c,\ldots,y_n^c)$$

(by (30) with $j=1,\ldots,n$)

$$= mult(\underline{M}(c),\underline{a}(c))$$

However, $c=\delta(t)$. Thus

$$R(t) \implies \Xi(G_{MV}(\Theta(\underline{M}),\Psi(\underline{a}),x))(t) = mult(\underline{M}(\delta(t)),\underline{a}(\delta(t))) = \Phi_N(\underline{M},\underline{a})(t)$$

Since (27) holds as claimed, we conclude that $N=$MV is a correct implementation.

5.4 SOURCES.

The source for the FIR algorithm is Brookes[1983] where it is in turn taken from Kung and Lin[1983]. Convolution figures in the basic popular article Kung[1982]. Brookes[1983] also considers palindromes, but the imprecise account of the algorithm presented there lead me to devise my own algorithm (viz PAL). The source for MV is Section 8.3.3 of Mead and Conway[1980] although the algorithm first appears in Kung and Leiserson[1979].

As with the analysis of the sorters in Chapter 4, our treatment of the algorithms in this chapter is a significant improvement on the (usually sketchy) accounts to be found in the general scientific literature: of course, it the use of value functions to formally define the behaviour of the algorithms that underwrites this improvement. Moreover, these definitions are of immediate practical use to anyone wishing to understand the algorithms in detail because it is easy to implement value functions on a computer and hence to *simulate* the algorithms (cf. Exercises 2.4.5(3)).

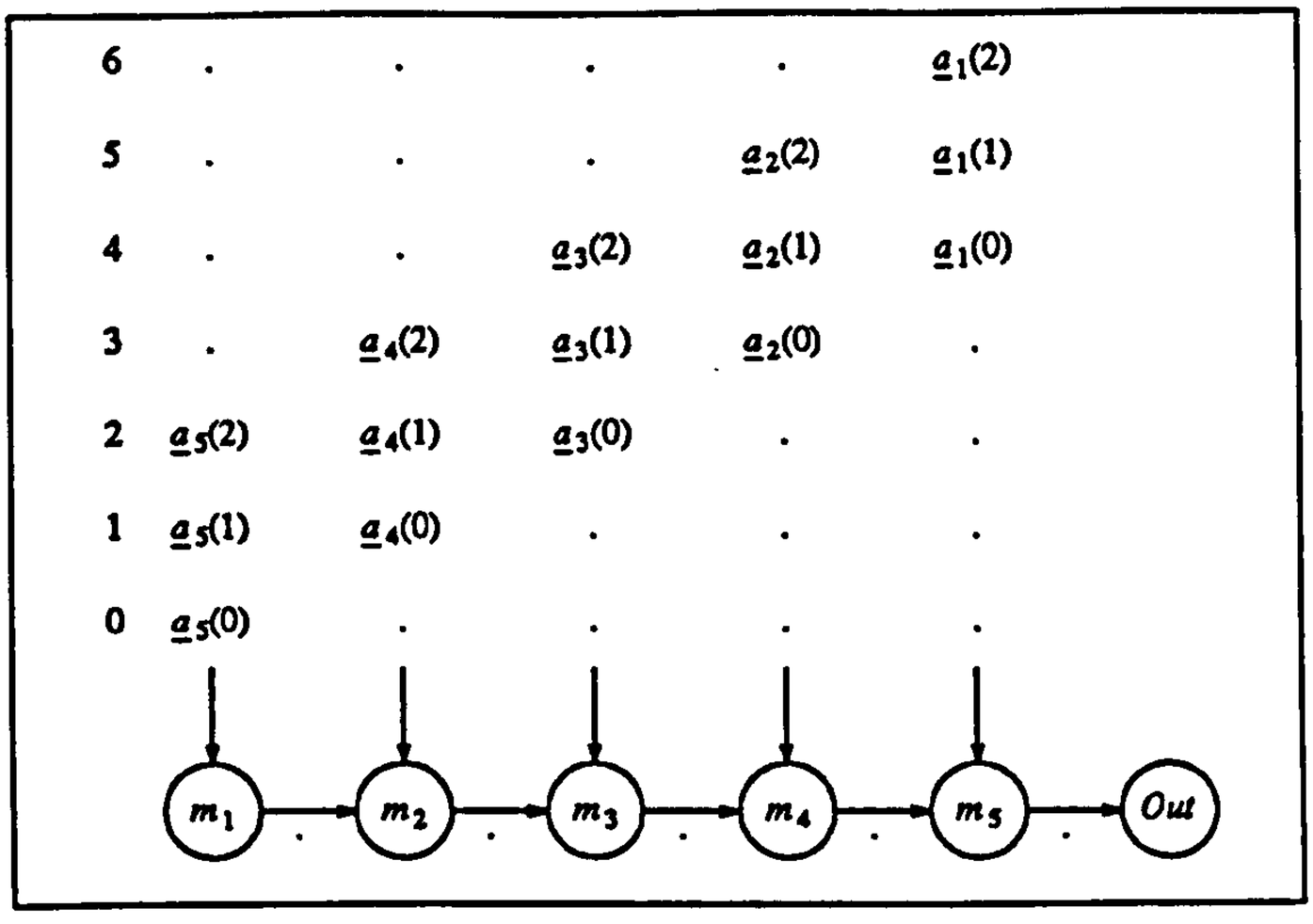


Figure 5.5 - FIR illustrated for $n = 5$ at $t = 0$.

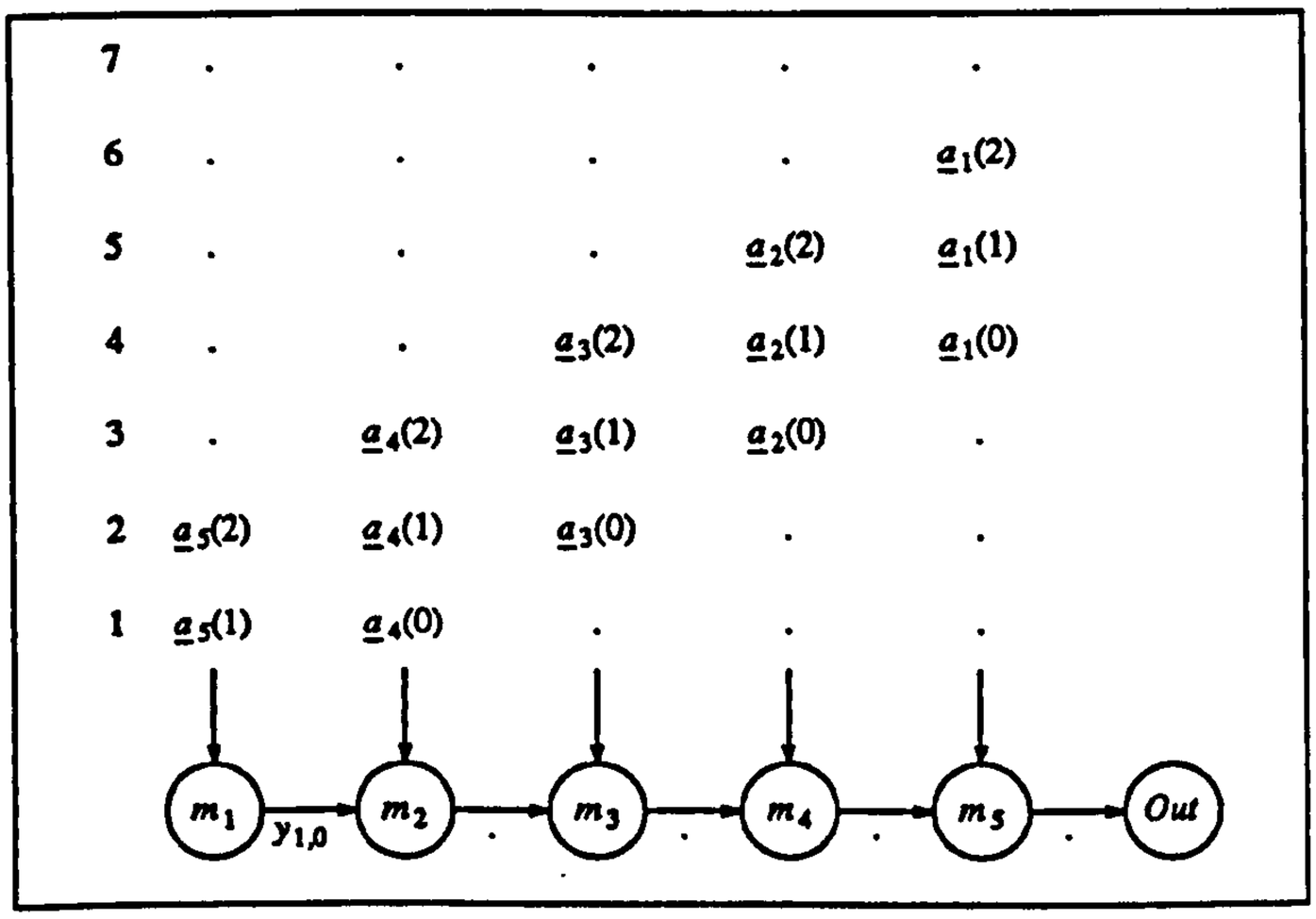


Figure 5.6 - FIR illustrated for $n = 5$ at $t = 1$.

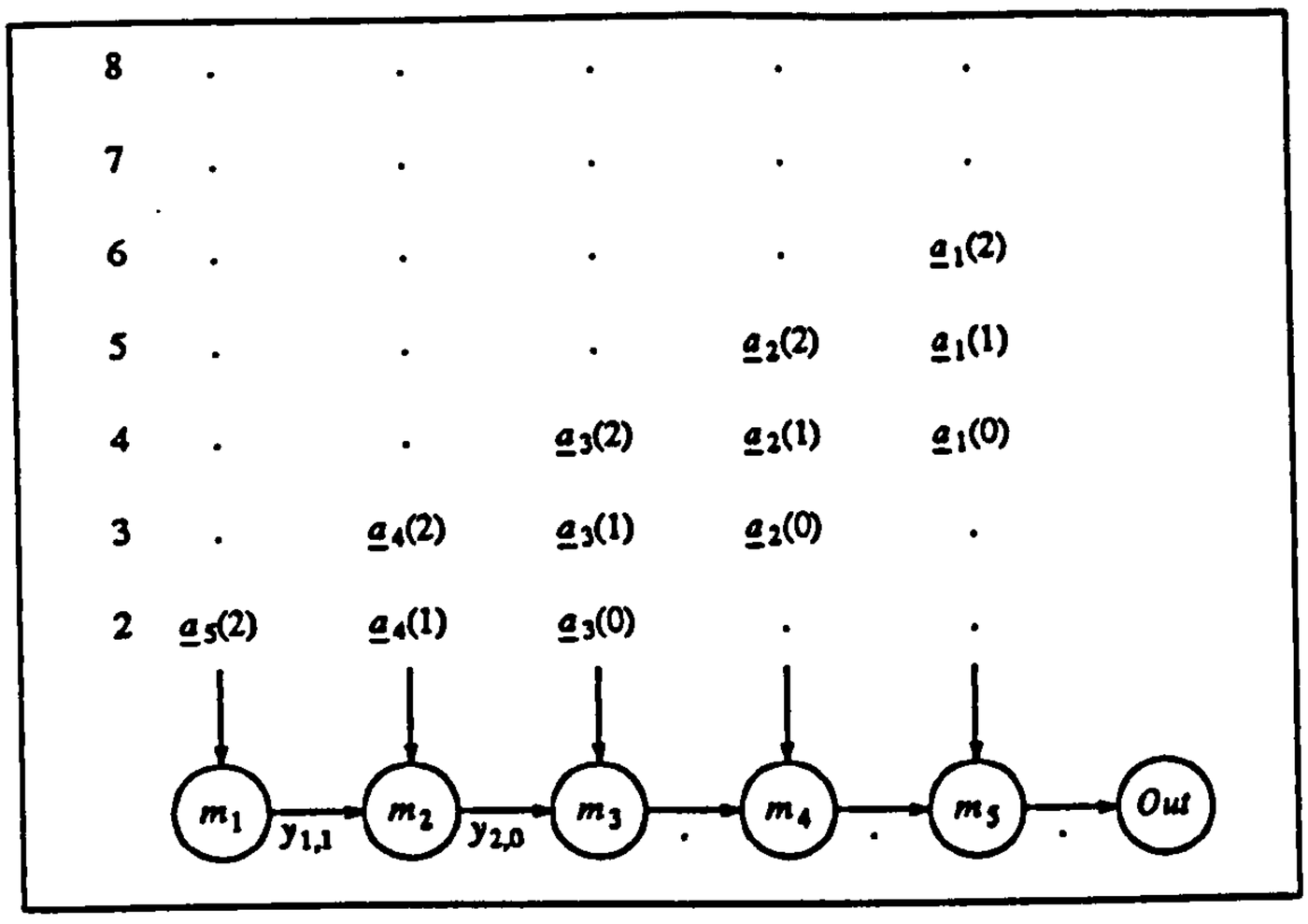


Figure 5.7 - FIR illustrated for $n = 5$ at $t = 2$.

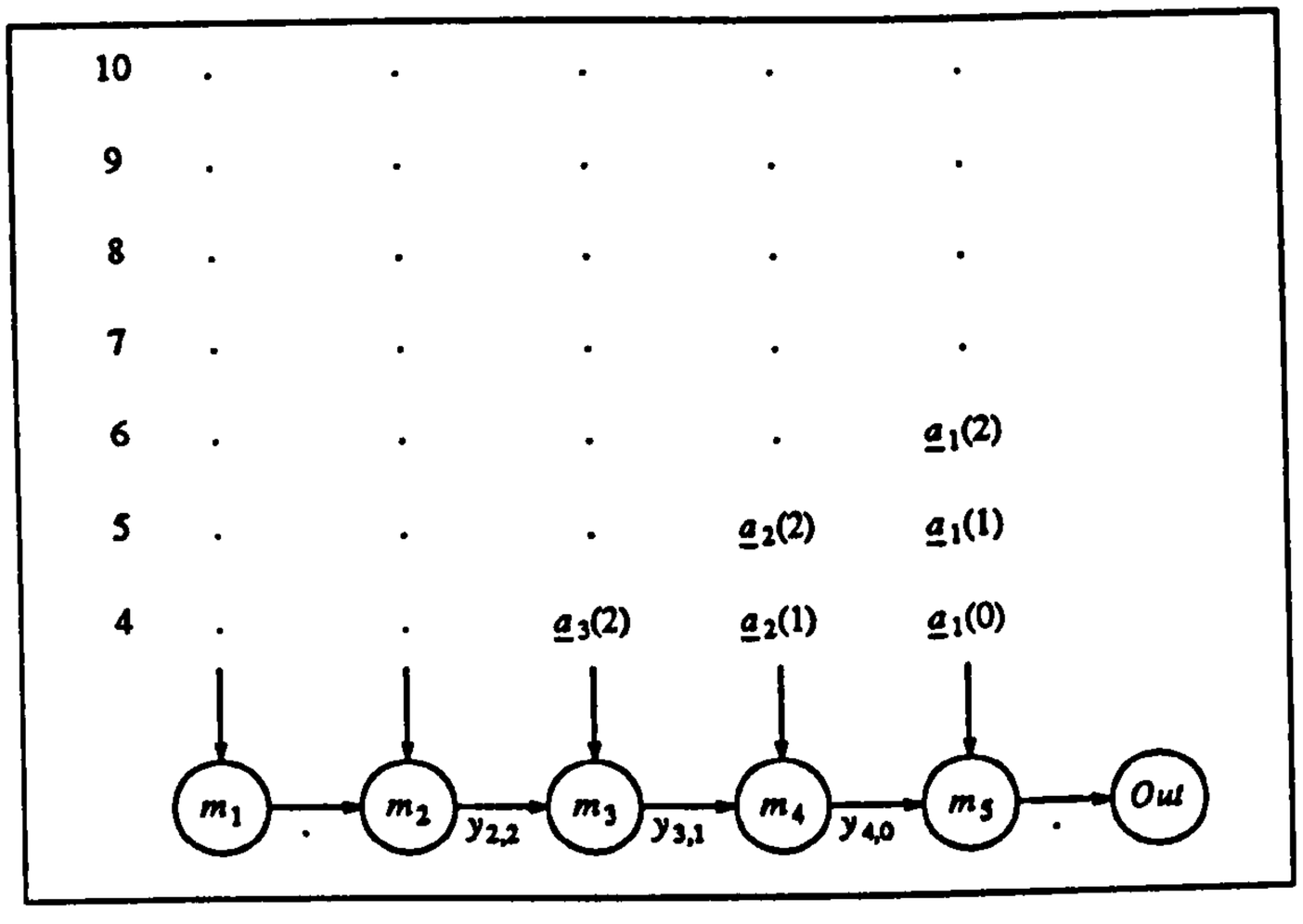


Figure 5.8 - FIR illustrated for $n = 5$ at $t = 4$.

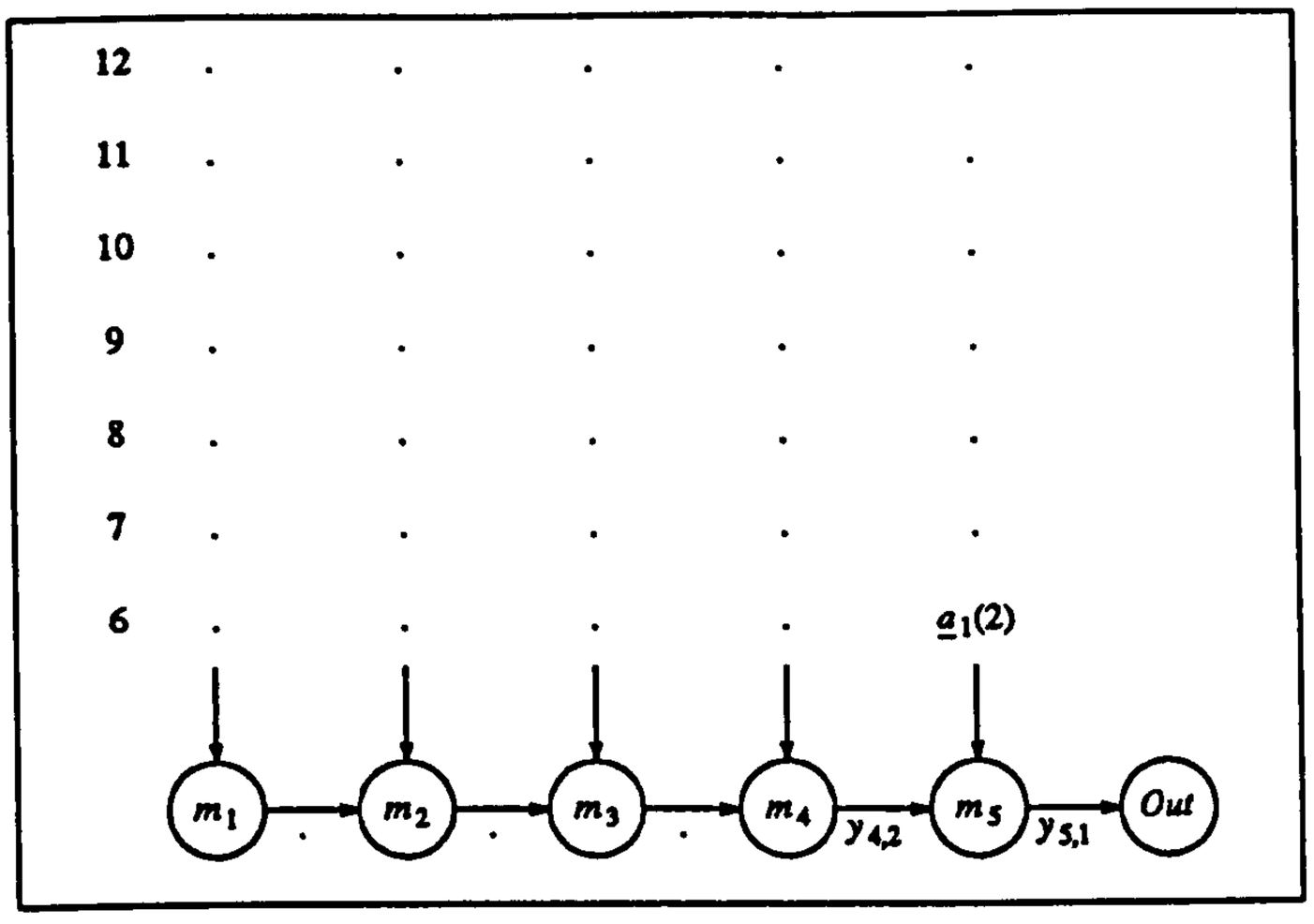


Figure 5.9 - FIR illustrated for $n = 5$ at $t = 6$.

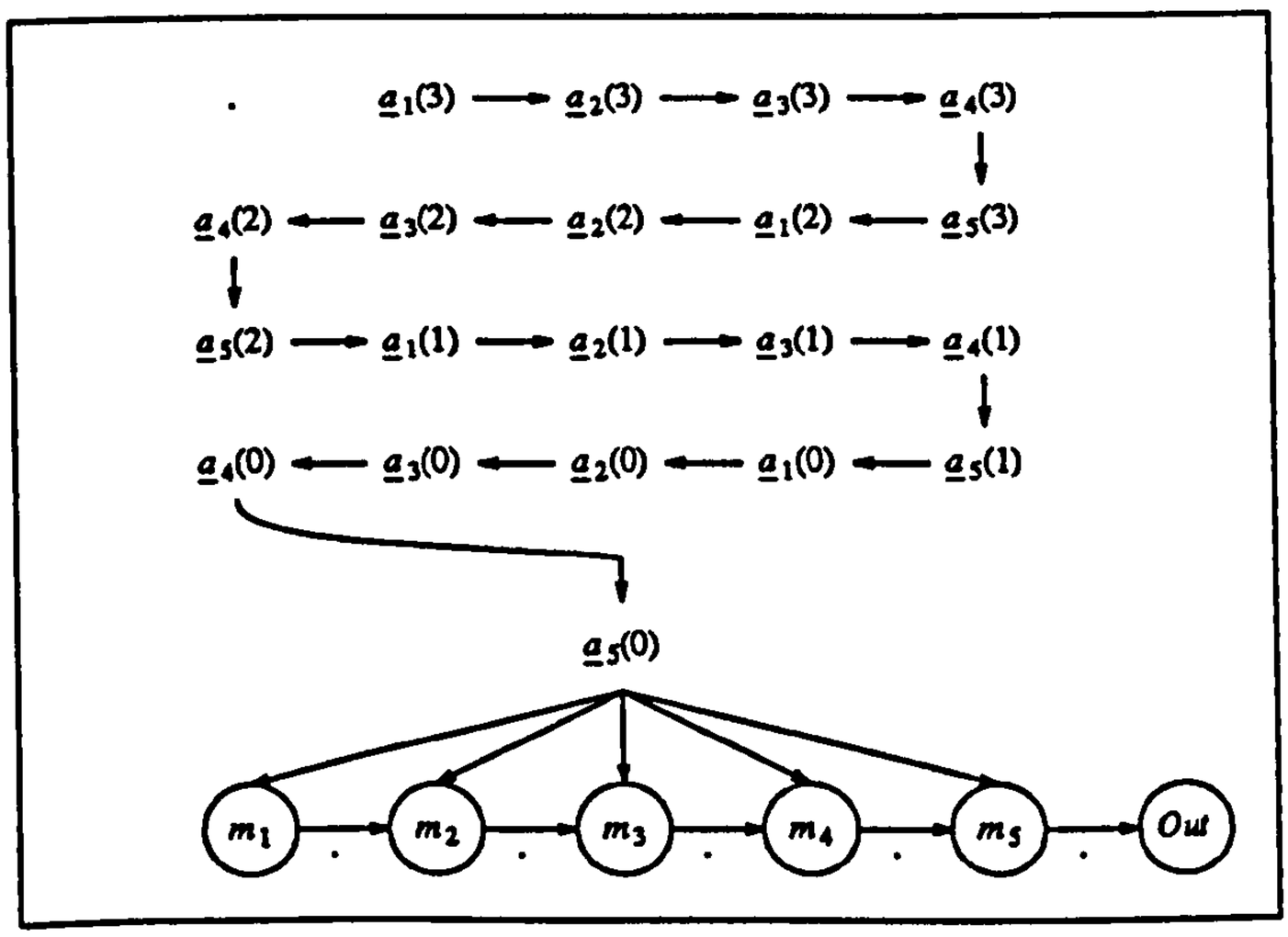


Figure 5.10 - FIR2 illustrated for $n = 5$ at $t = 0$.

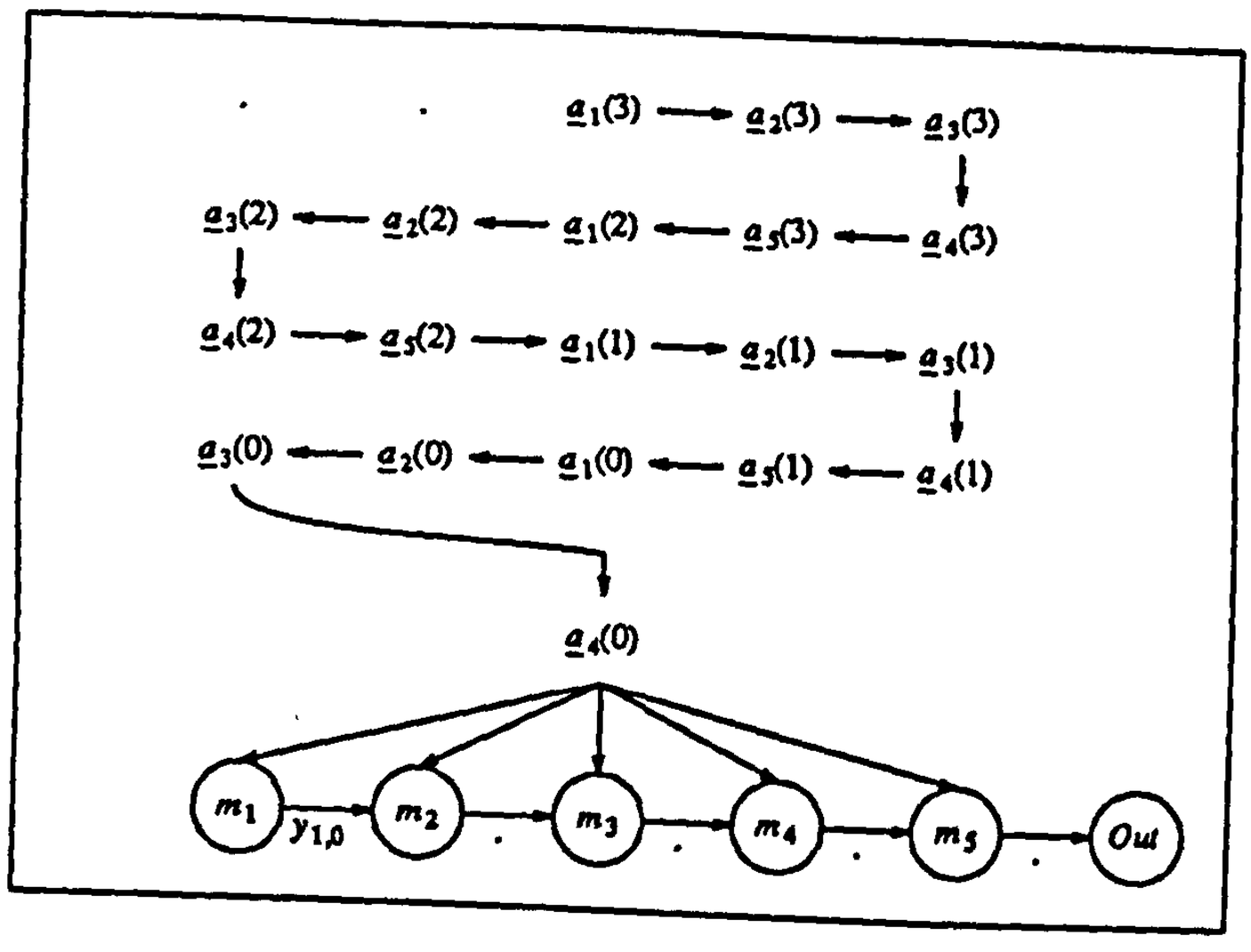


Figure 5.11 - FIR2 illustrated for $n = 5$ at $t = 1$.

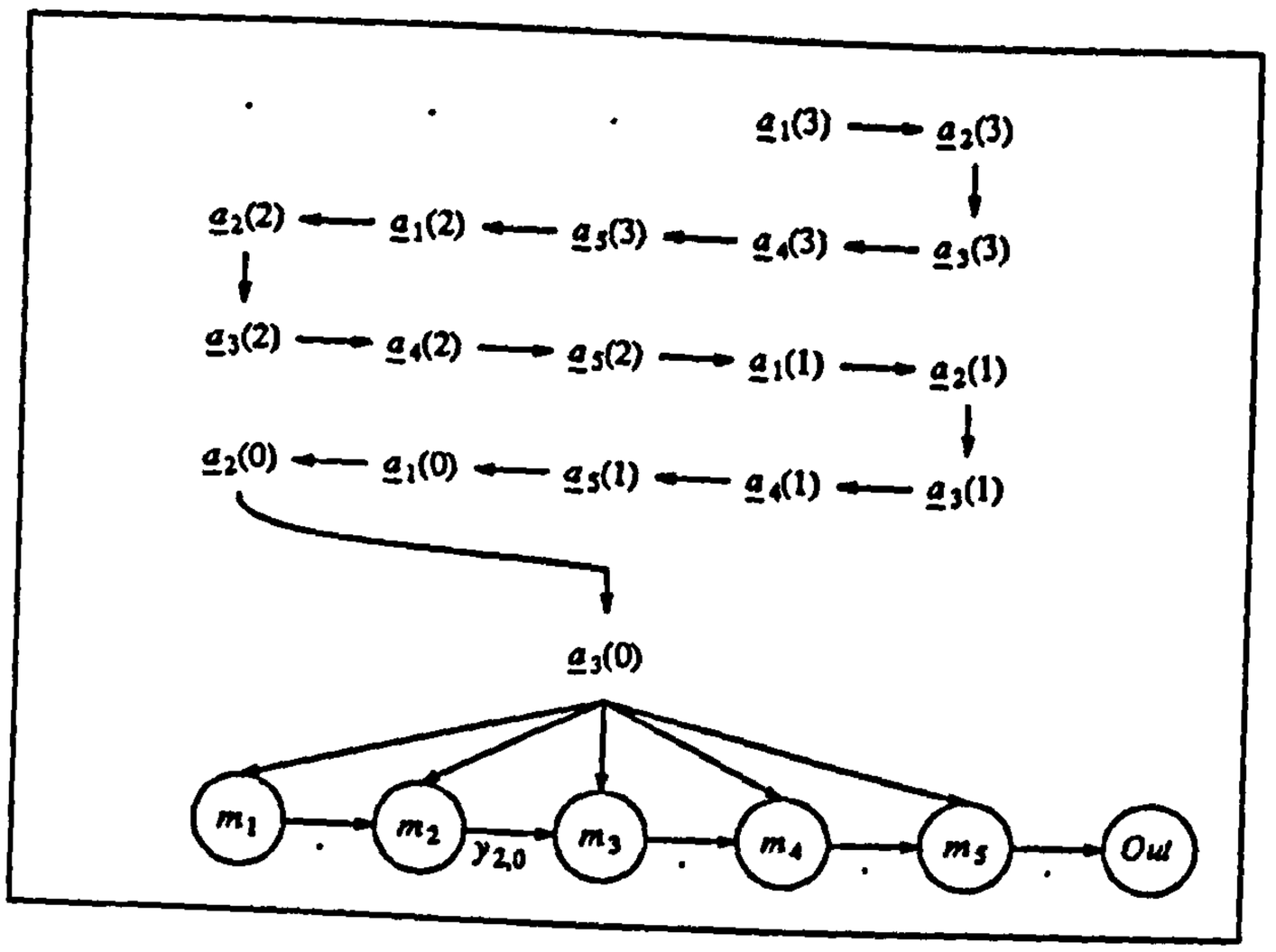


Figure 5.12 - FIR2 illustrated for $n = 5$ at $t = 2$.

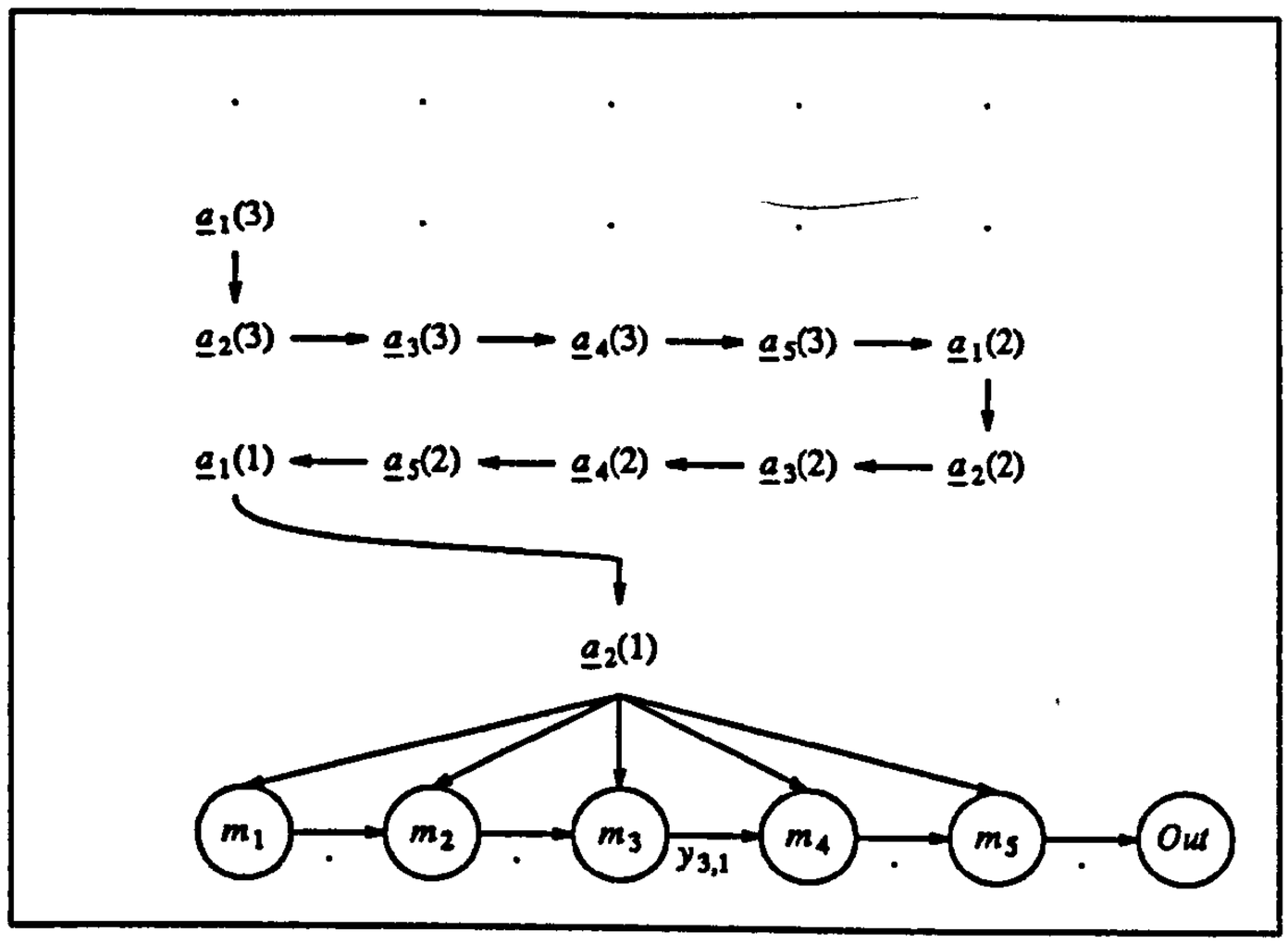


Figure 5.13 - FIR2 illustrated for $n = 5$ at $t = 8$.

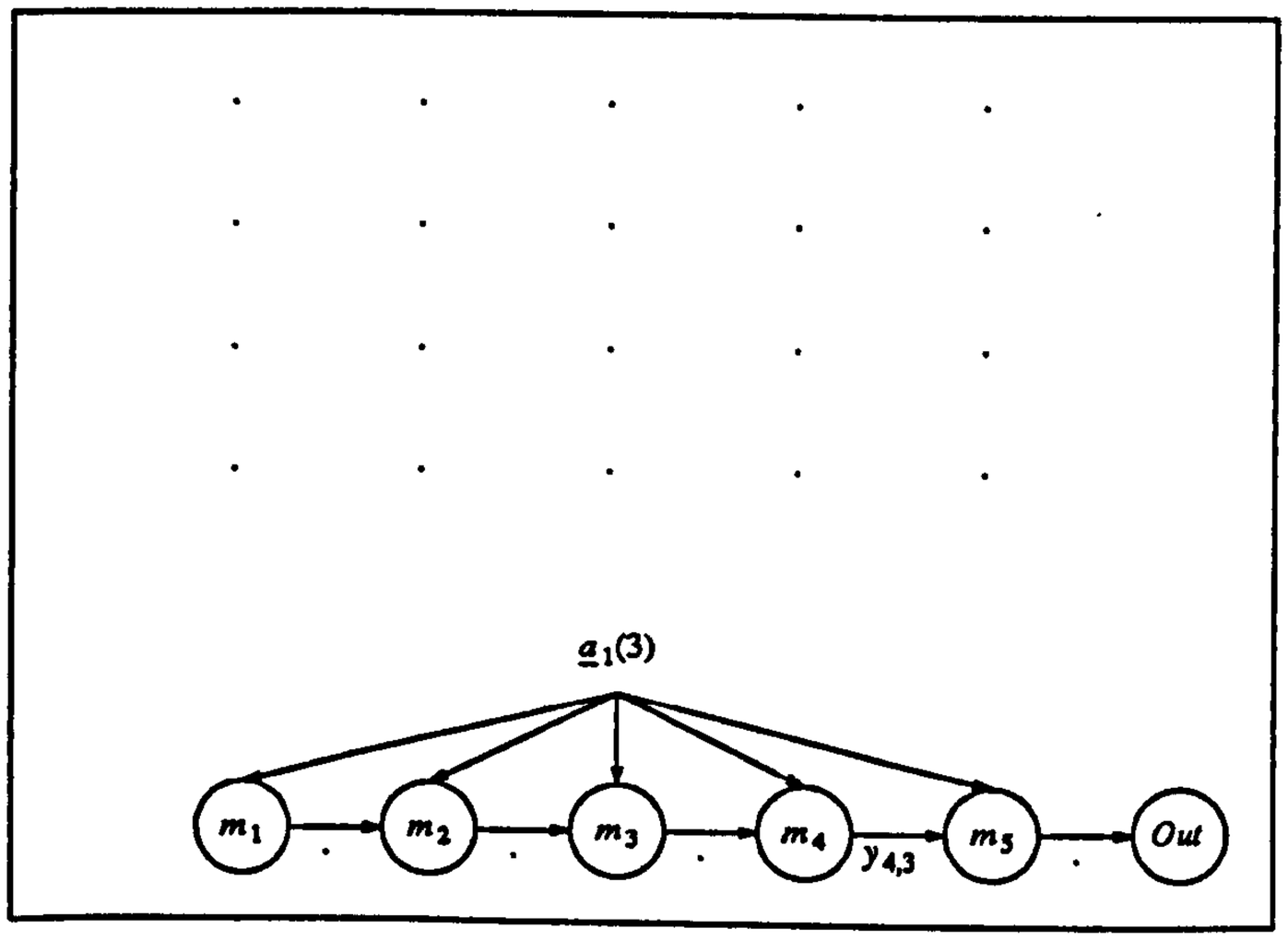


Figure 5.14 - FIR2 illustrated for $n = 5$ at $t = 19$.

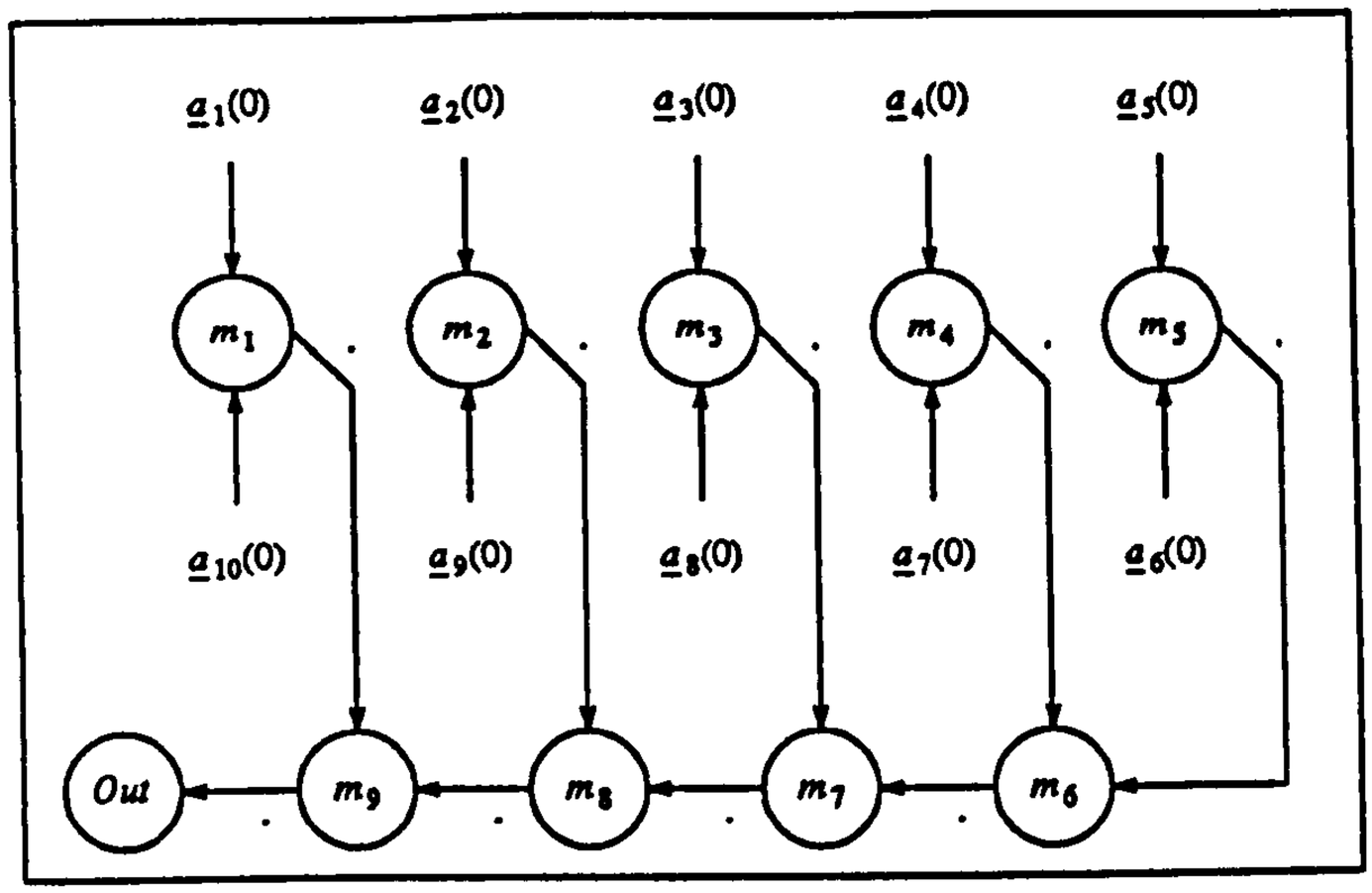


Figure 5.15 - PAL illustrated for $n = 10$ at $t = 0$.

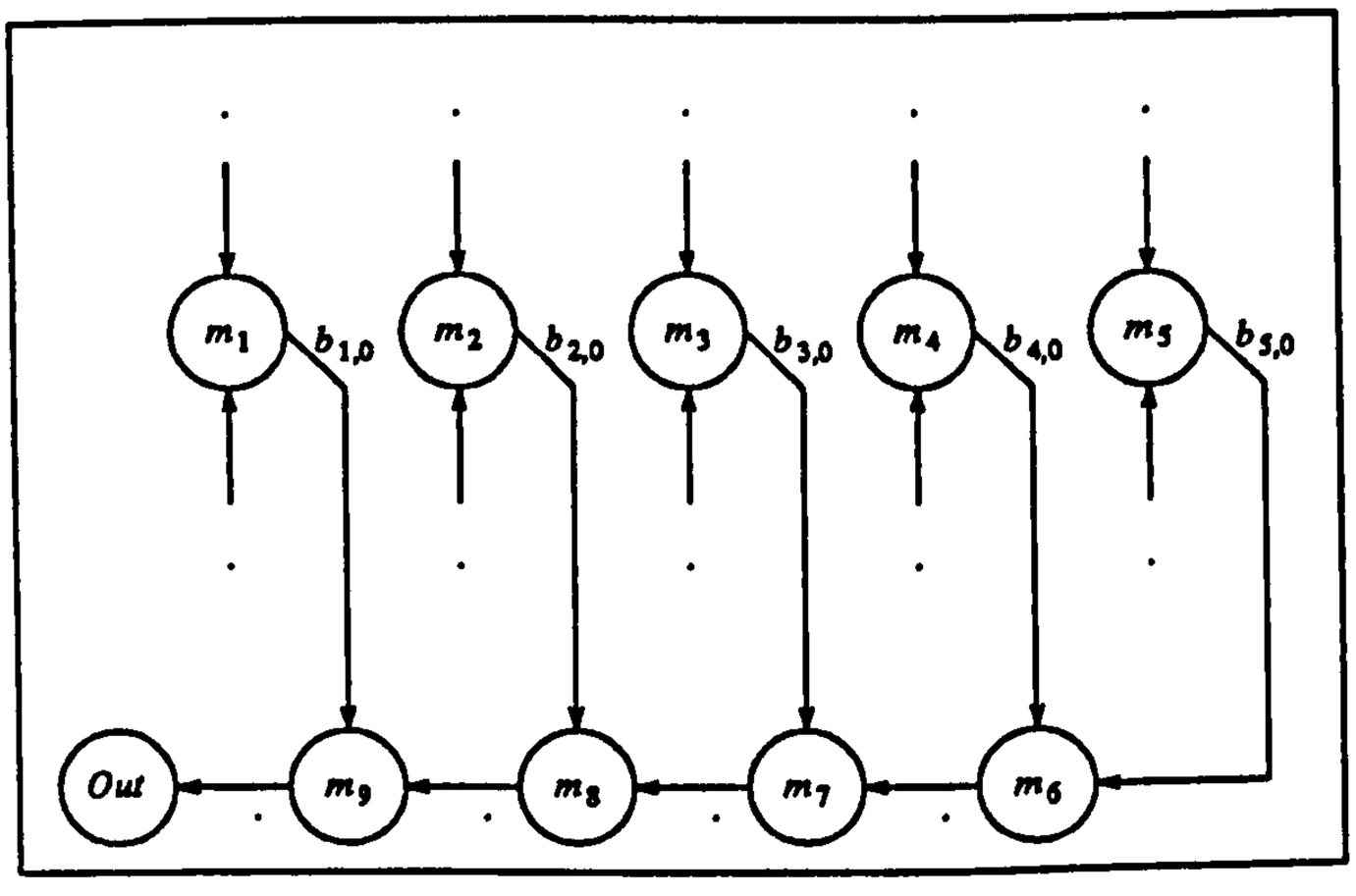


Figure 5.16 - PAL illustrated for $n = 10$ at $t = 1$.

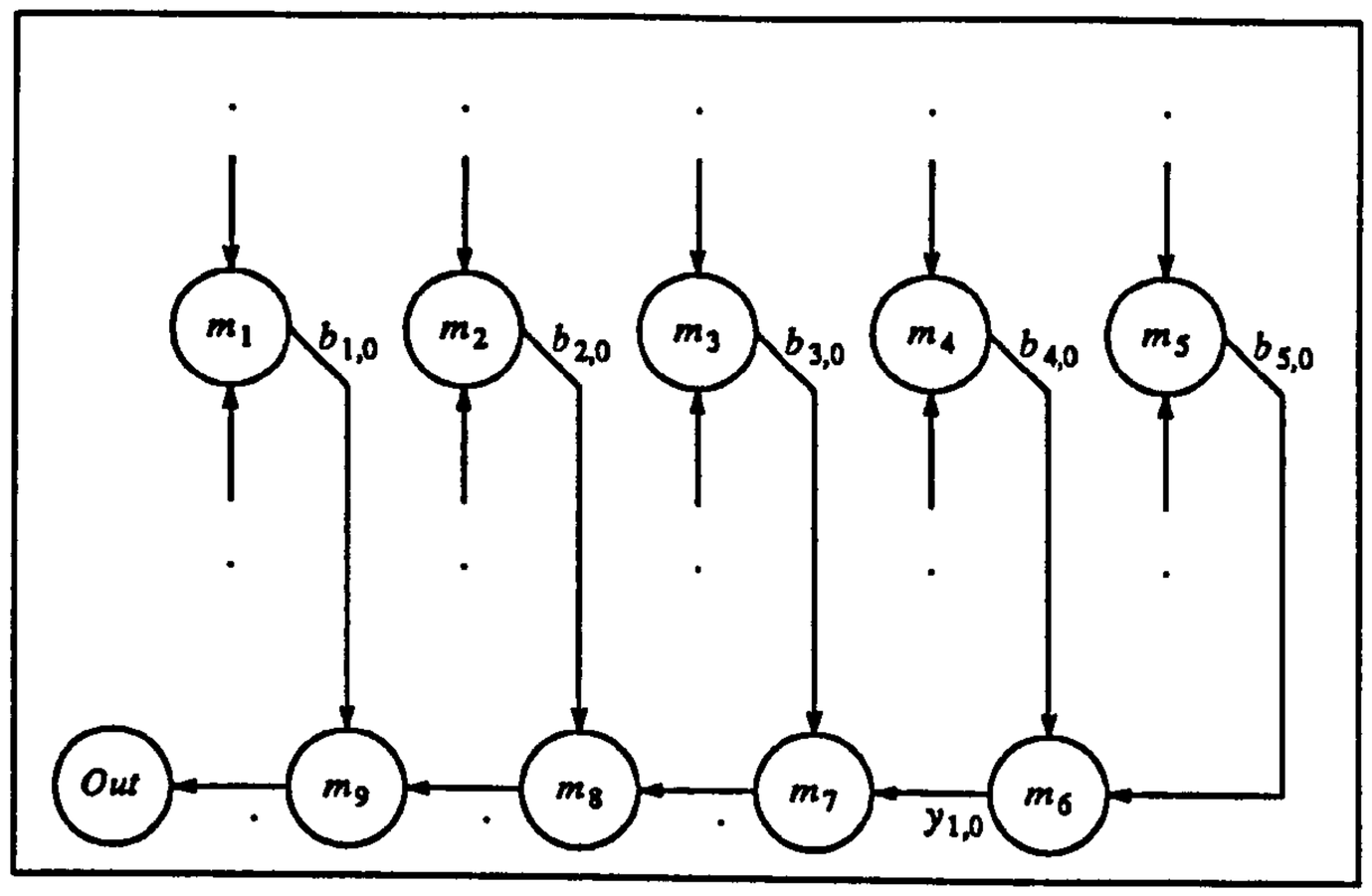


Figure 5.17 - PAL illustrated for $n = 10$ at $t = 2$.

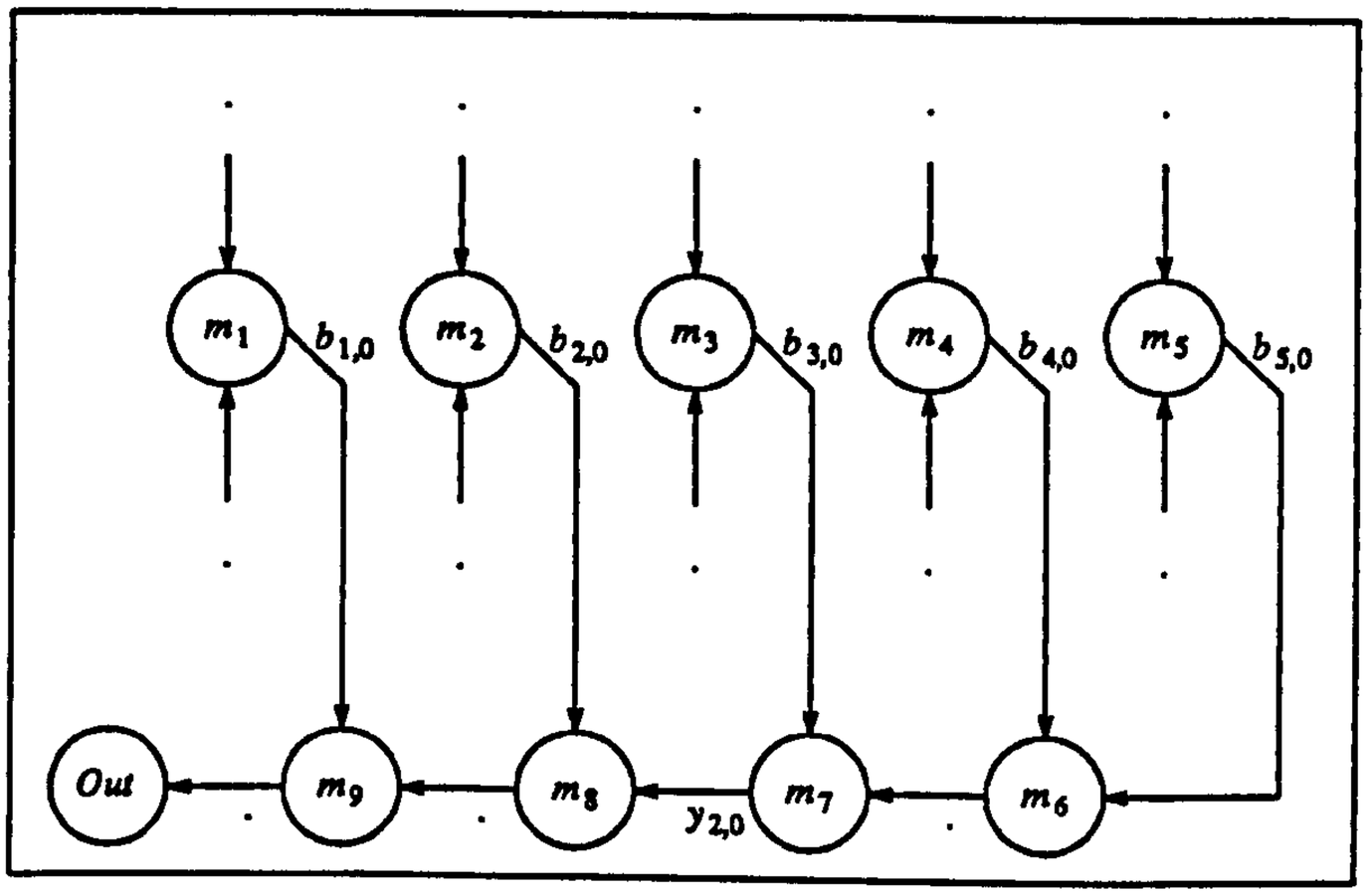


Figure 5.18 - PAL illustrated for $n = 10$ at $t = 3$.

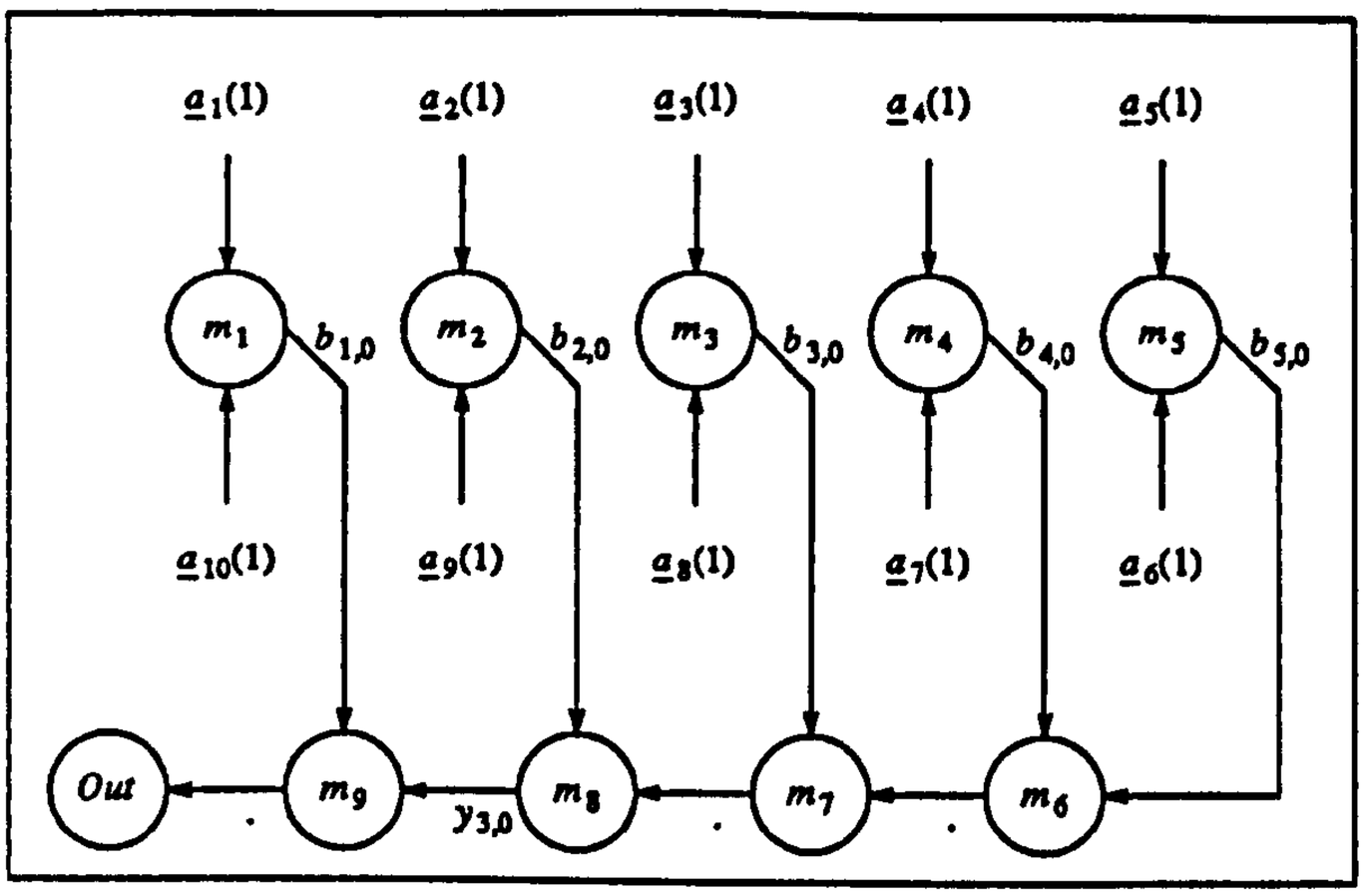


Figure 5.19 - PAL illustrated for $n = 10$ at $t = 4$.

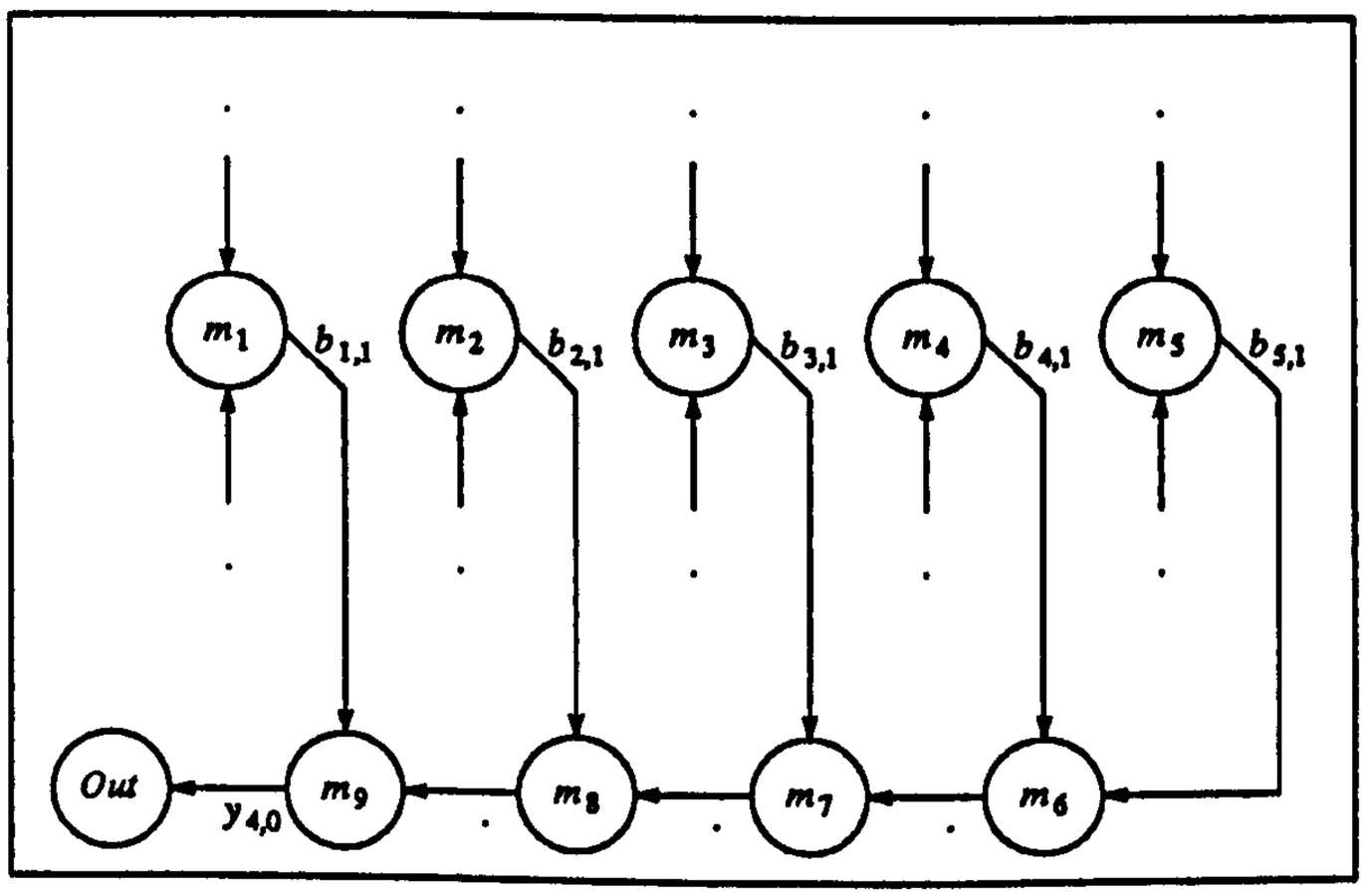


Figure 5.20 - PAL illustrated for $n = 10$ at $t = 5$.

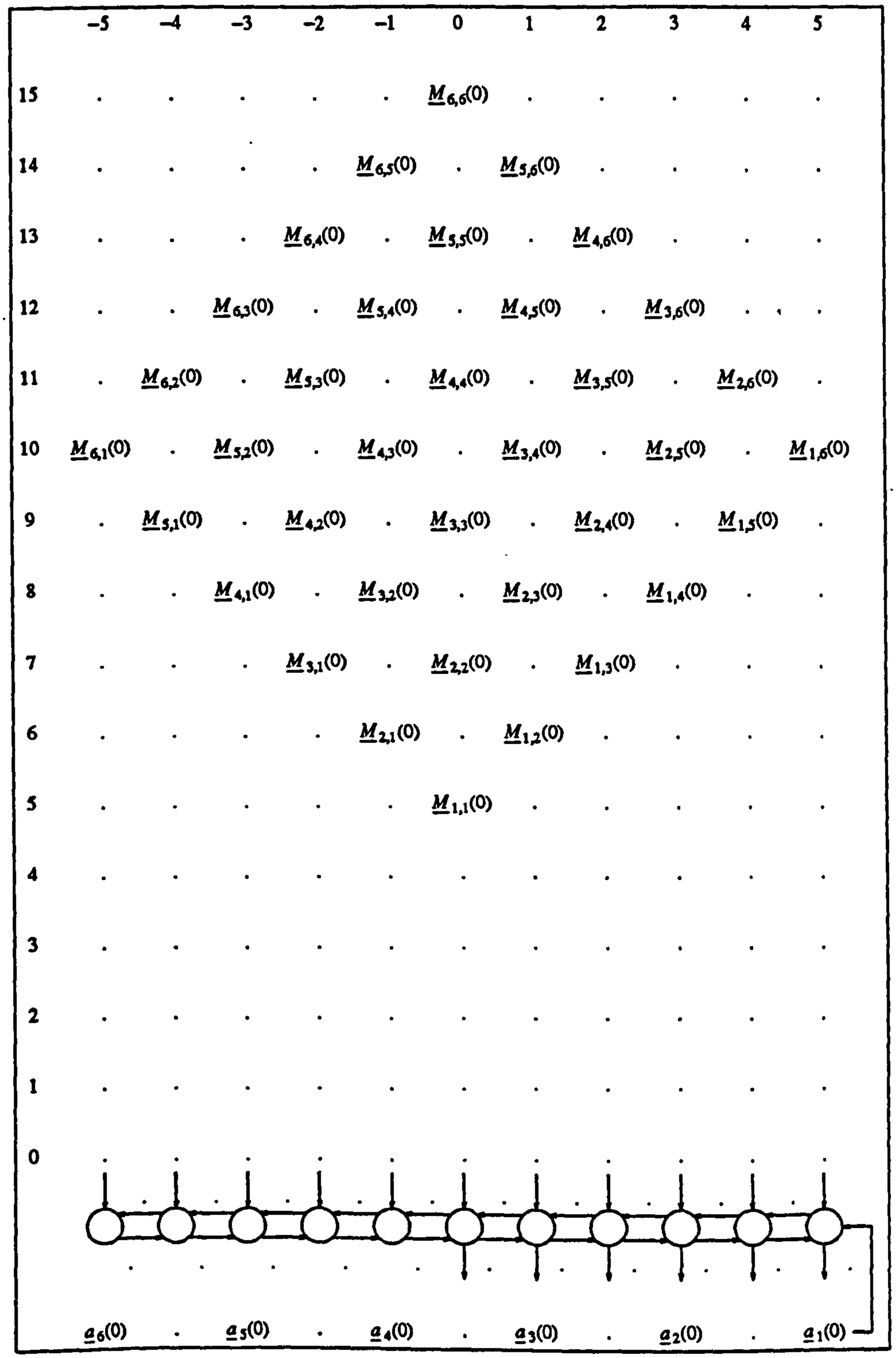


Figure 5.21 - MV illustrated for $n = 6$ at time $t = 0$.

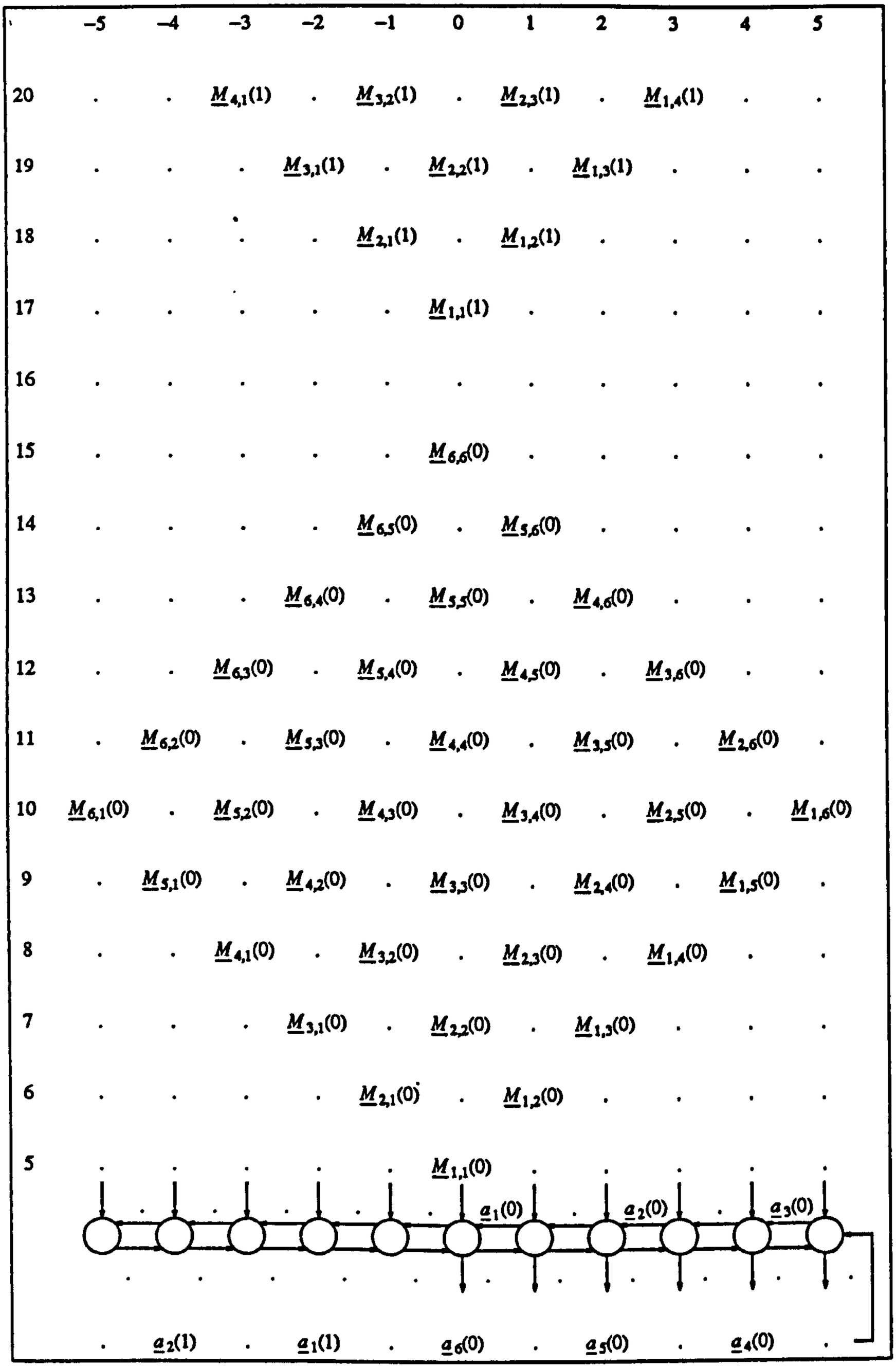


Figure 5.22 - MV illustrated for $n = 6$ at time $t = 5$.

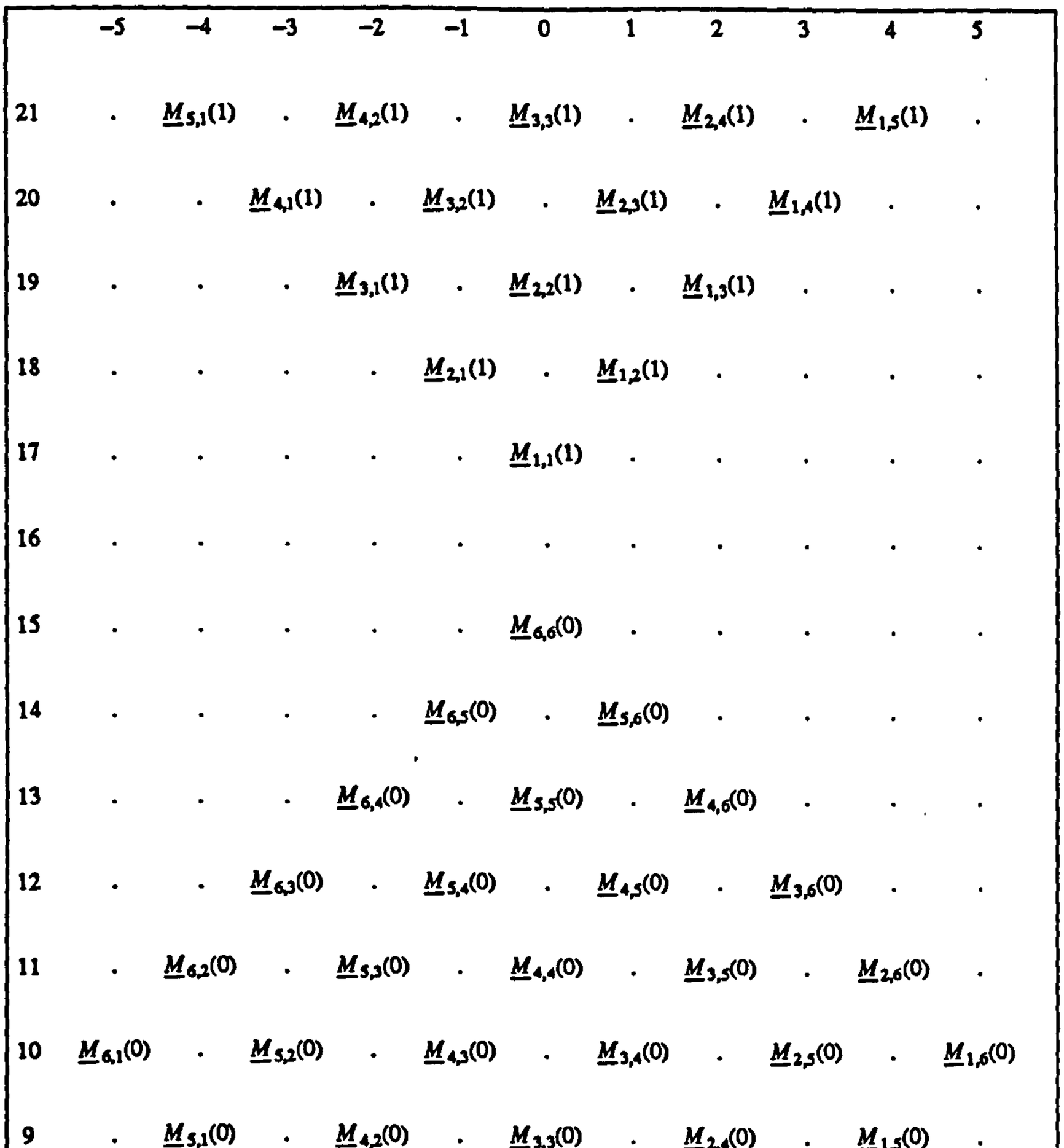


Figure 5.23 - MV illustrated for $n = 6$ at time $t = 6$.

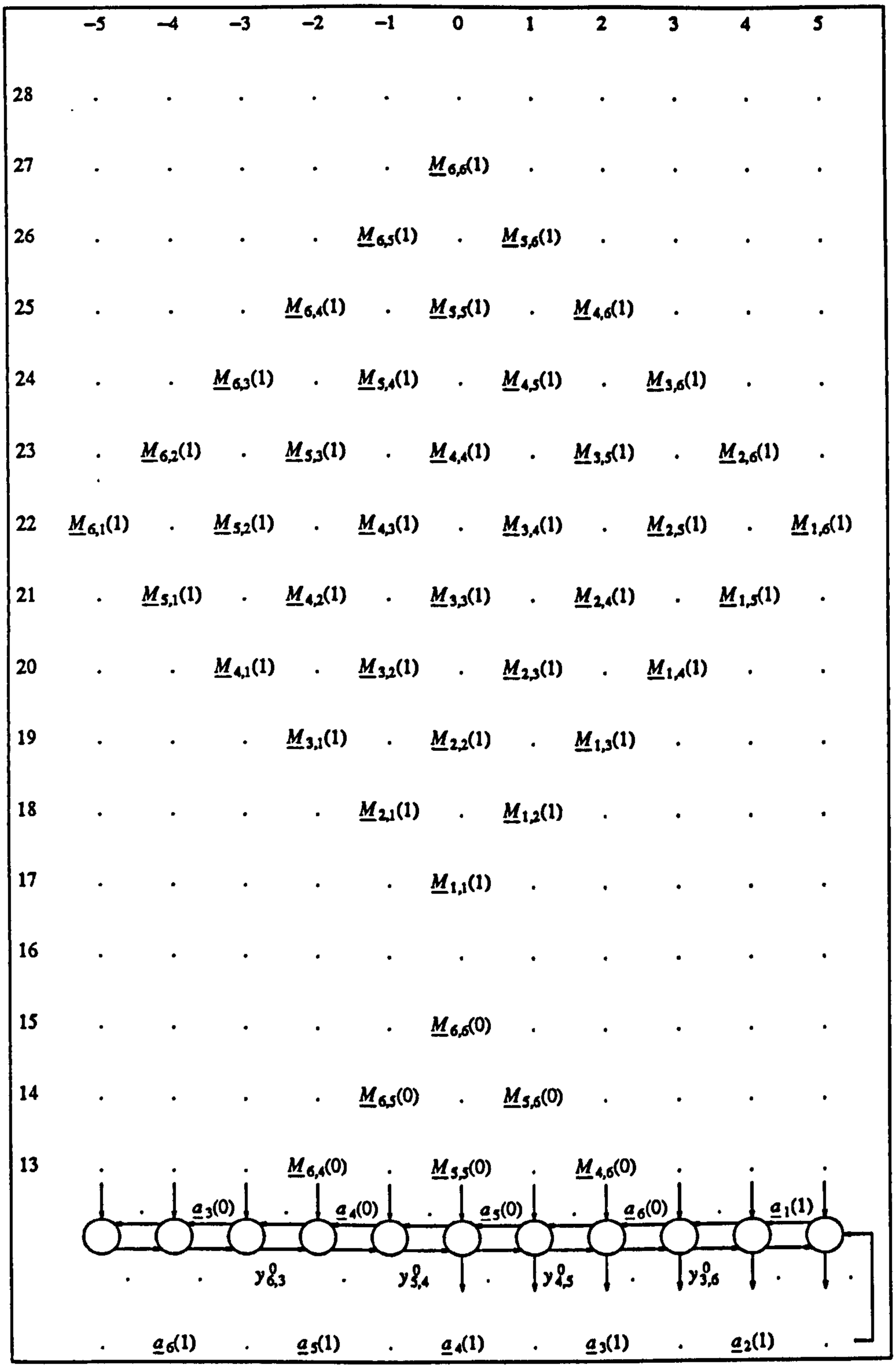


Figure 5.24 - MV illustrated for $n = 6$ at time $t = 13$.

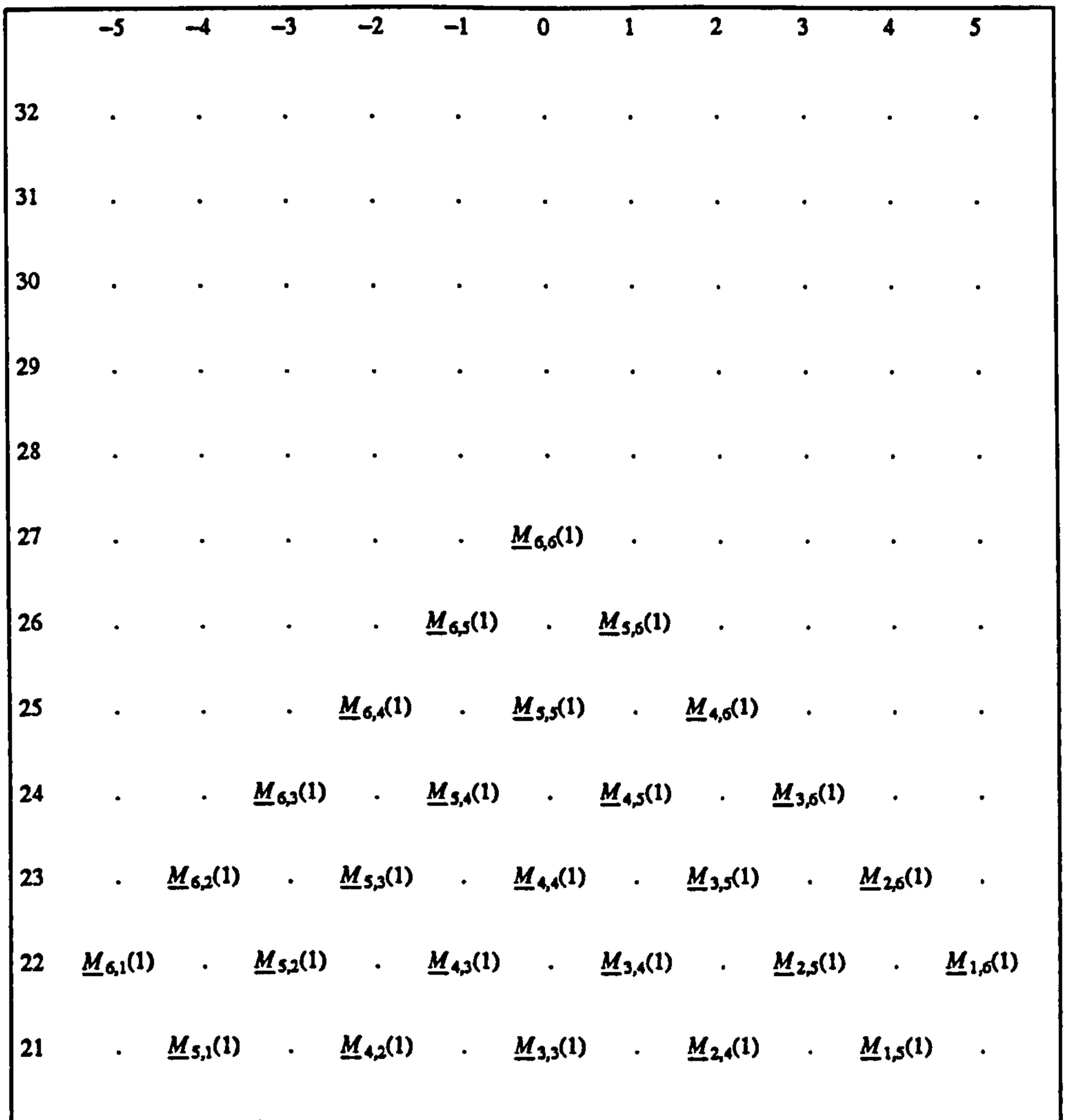


Figure 5.25 - MV illustrated for $n = 6$ at time $t = 17$.

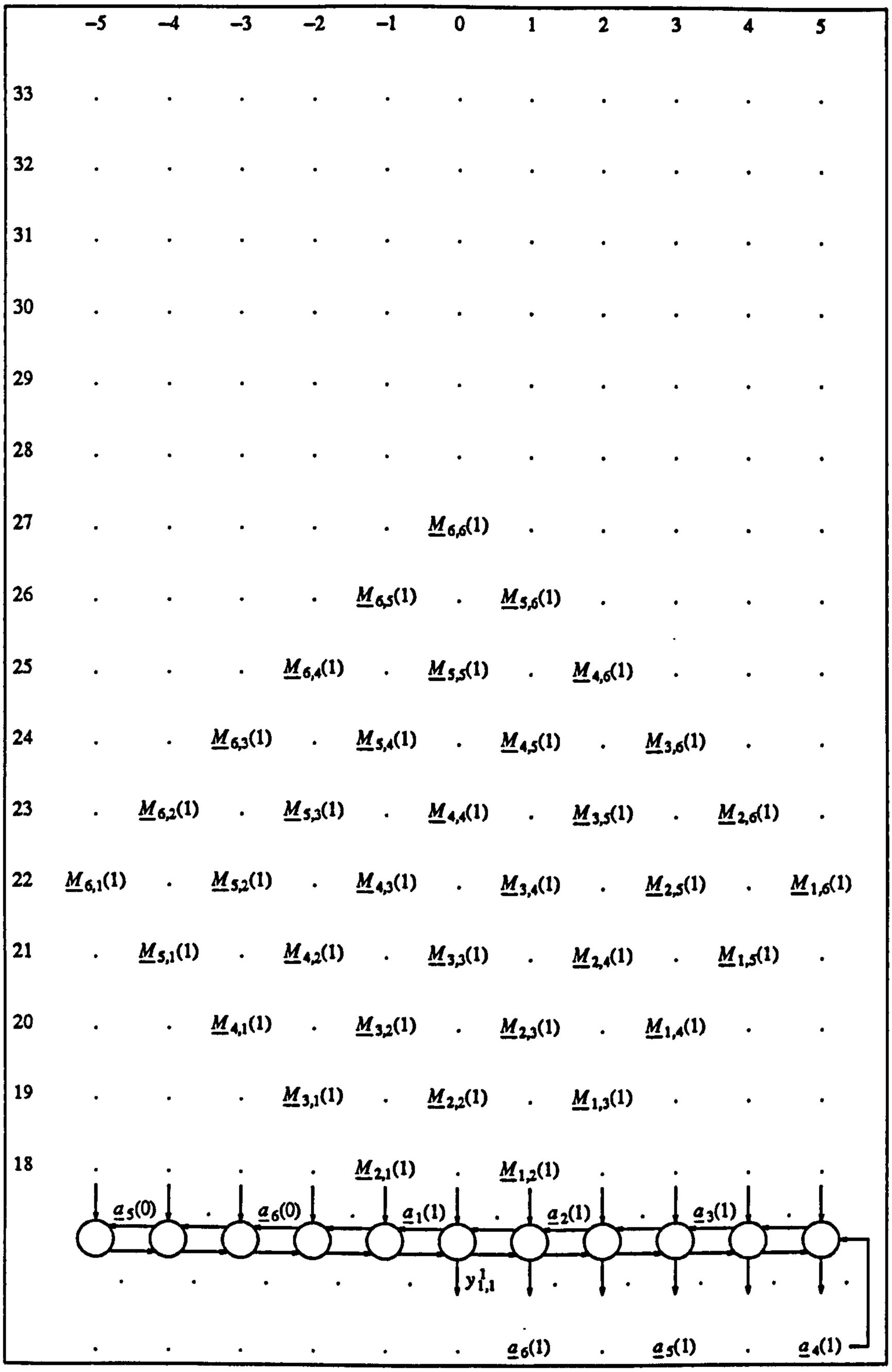


Figure 5.26 - MV illustrated for $n = 6$ at time $t = 18$.

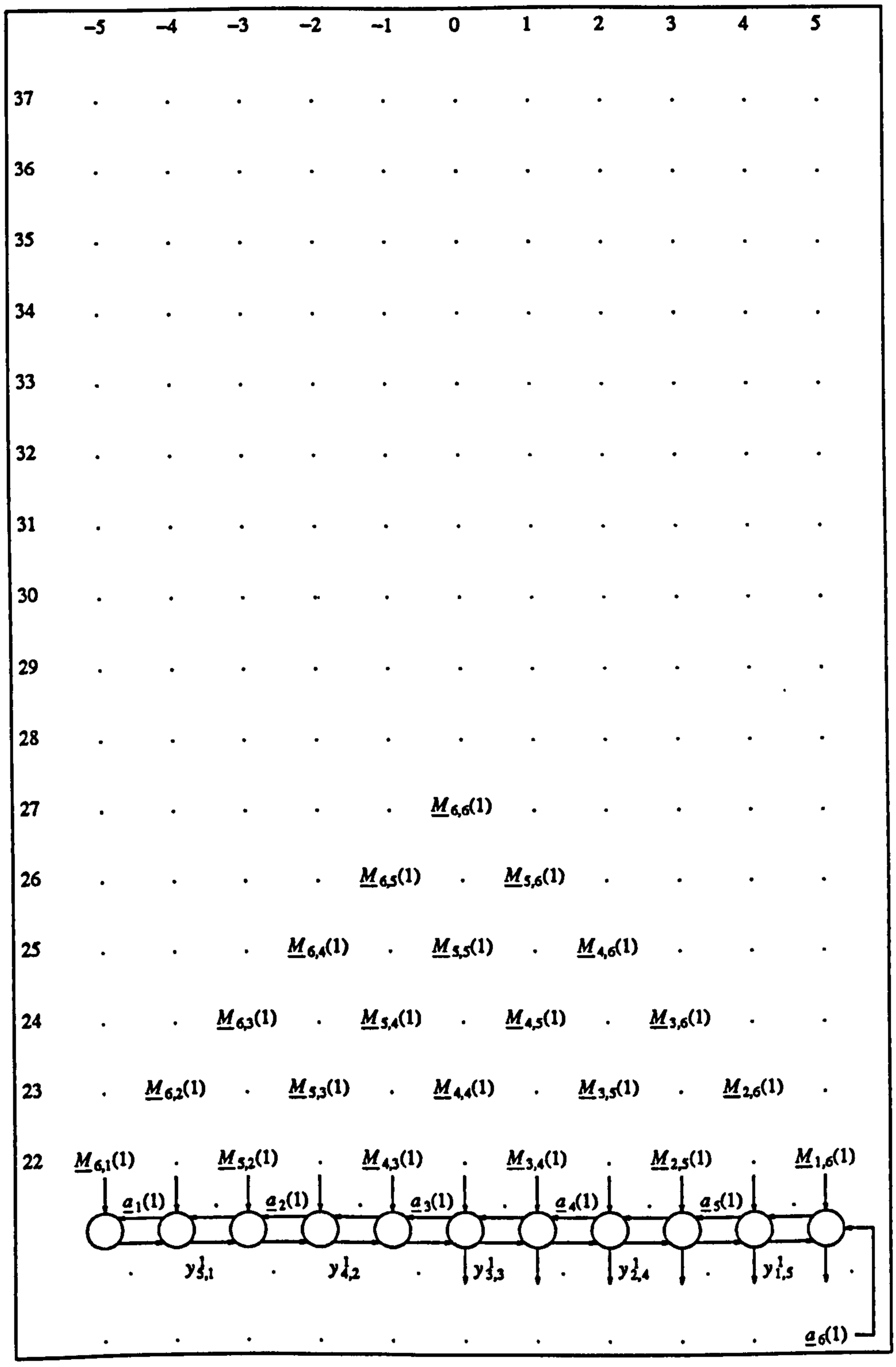


Figure 5.27 - MV illustrated for $n = 6$ at time $t = 22$.

CHAPTER 6
SIMULATION

We have seen how PR is a useful tool in formally defining and verifying synchronous algorithms. In this chapter we will consider how synchronous algorithms may be *executed* or *simulated*.

Whilst there is no theoretical substitute for formal verification, simulation is an important stage in the development of a synchronous algorithm, since it is by simulating the algorithm that we increase our confidence in the algorithm's correctness, or indeed, demonstrate the existence of errors. Also, in the case of algorithms that are difficult to understand, a simulation of the algorithm can reveal what needs to be proved about the algorithm. For example, in the case of the MV algorithm of Section 5.3, it is not initially obvious at what times and places the required outputs will appear; it is by scrutinising the trace or simulation of the algorithm in operation that we first *guess* the formulae that predict when and where outputs arrive, and that we may ultimately attempt to verify.

Strategy. The obvious way to proceed in providing simulation tools for synchronous algorithms is to *implement* PR. Since the specification of a synchronous algorithm's behaviour in PR (by means of value functions) tells us everything there is to know about that behaviour, we can simulate the algorithm by implementing its PR-specification in some executable computer programming language.

What language characteristics are suitable for implementing PR? We think of PR as an algorithmic language for defining functions over an abstract data type, and so it is natural to consider implementation languages which involve both abstract data types, and a facility for programming functions by means of *procedures*. Also, since PR enjoys some parallelism, it will be helpful if the implementation language does also.

Rather than employ an existing language with (some of) these facilities, such as ML (Gordon, Milner, and Wadsworth[1979]), HOPE (Burstall, MacQueen, and Sanella[1980]), MIRANDA (Turner[1982]), FP (Backus[1978]), or OCCAM (Inmos[1984]), we will invent a von Neumann language which is especially tailored to implementing PR. Specifically, we will employ an algorithmic language called *FPIT*. FPIT is an executable von Neumann language comprising an algorithmic notation of functional parallel iterative programs, together with a formally defined semantics and complexity theory. FPIT is defined over an abstract data type and involves *function procedures*. Additionally, FPIT programs involve the *concurrent* or *parallel assignment statement* wherein many assignments are executed simultaneously.

Programs in FPIT are built up from programs in another von Neumann language that we call *PIT* (for parallel iterative). Like all our languages, PIT comes equipped with a formally defined semantics and complexity theory. In Sections 6.1 and 6.2 we develop a theory of PIT programs which underpins the definition of FPIT.

After these preliminary ideas, in Section 6.3 we will define FPIT and then show how PR-schema can be implemented in FPIT. In Section 6.4 we address the general problem of implementing arbitrary PR-schema: we do this by defining a *compiler* from PR into FPIT. Importantly, the semantics of FPIT is defined independently of the semantics of PR, and so we may consider the *correctness* of this compiler. Indeed, we will ultimately prove (in Chapter 7) that this compiler is correct, and, moreover, we will prove that the compiler *preserves lengths of computation*; that we can relate the performance of a synchronous algorithm to that of its simulation is an important fact, since in order for a simulation of an algorithm to be practicable, it must have a 'reasonable' execution-time. That we can *predict* the complexity of the simulation from the complexity of the original algorithm is obviously a useful feature.

We will see in Examples 6.1.8 that synchronous algorithms may be directly simulated in FPIT without using PR as an intermediate stage, and thus FPIT can be viewed as an alternative means of specifying or defining the behaviour of a synchronous algorithm. Now, until this time, we have said that the behaviour of a synchronous algorithm is officially defined by means of a PR scheme and so we must ask: what is the relationship between FPIT and PR? Does a specification of a synchronous algorithm in one system tell us any more about the algorithm than its specification in the other? In Chapter 7 we will answer this question (in the negative) by establishing that FPIT and PR are *equivalent* specification languages.

6.1 THE LANGUAGE PIT.

PIT is an *imperative* programming language (that is, it involves assignment statements), and the language has *operational* or *state-transformer* semantics. An excellent introduction to semantics can be found in Bakker[1980] (this exposition we will subsequently refer to as simply 'de Bakker'). De Bakker provides an account of the mathematical theory of computation over the natural numbers; the reader who is familiar with the subject of operational semantics will appreciate that much of what follows is simply an extension of the definitions and ideas of de Bakker to computation over the more general structure of a many-sorted algebra along the lines of Tucker and Zucker[1987]

After some preliminary definitions, in Section 6.1.5 we define the syntactic part of the language, that is, we define *PIT programs*. In Section 6.1.6 we define the semantics of PIT programs, and in Section 6.1.7 we define the performance of PIT programs by means of length of computation functions. In Examples 6.1.8 we give some examples of PIT programs in operation.

In the same way that PR needs a clock T (which is a copy of the natural numbers) and the Booleans $\mathbb{B}$ to be able to define 'definition by primitive recursion' and 'definition-by-cases' respectively, PIT needs these sets to define the semantics of iteration and 'if-then-else' constructs. For this reason we will assume S to be standard sort set, Σ to be a standard S-sorted signature, and A to be a standard Σ-algebra, throughout the following technical sections (cf. Section 3.1.8).

6.1.1 Variables.

We begin with some many-sorted notation for variables.

Definition. We define Var to be the S-indexed family:

$$Var = \langle Var_s : s \in S \rangle$$

of collections $Var_s = \{ x^s, x_1^s, x_2^s, \ldots \}$ of *variables* or *identifiers of sort* s. (When the sort of some variable x^s is understood, we will omit the superscript s.)

6.1.2 States.

The idea behind a 'variable of sort s' is that x is something which can hold a value from A_s. When executing a program which involves a collection of variables, the values held by each variable are collected together into a *state*:

Definition. We define the collection $States(A)$ of *states over* A to be the collection of all *total* mappings

$$\rho : Var \longrightarrow \bigcup_{s \in S} A_s$$

such that $\rho \in States(A)$ only when for each sort s, and for each $x^s \in Var_s$, $\rho(x^s) \in A_s$; that is, $\rho \in States(A)$ only when the codomain of ρ restricted to Var_s is A_s. □

To say that $x \in Var_s$ is to *declare* that x is a variable of sort (or *type*) s; that is, the intention is that x should only ever hold a value taken from the set A_s.

A state is essentially just a mapping from variables onto (their) values: if x is a variable and ρ is a state, then $\rho(x)$ is the value of x *under* ρ; of course, the restriction '$\rho(x^s) \in A_s$' amounts to type-checking the value held by x^s.

Given a state $\rho \in States(A)$, $\rho(x)$ is just the value of x. Now suppose we change the value of x to be a rather than whatever it was under ρ. Then we can form a new state $\rho' \in States(A)$ under which x has the value a but all other variables have the same value under ρ' as they did under ρ. We formalise this state transformation as follows:

Definition. (*Variant* of a state.) Let $\rho \in States(A)$, and let $a \in A_s$ and $x \in Var_s$, for any $s \in S$. Then $\rho' = \rho\{ a / x \} \in States(A)$ is the state defined by

$$(\forall y \in Var)\ \ \rho'(y) = \begin{cases} \rho(y) & \text{if } y \neq x \\ a & \text{if } y = x \end{cases}$$

Definition. Let $\rho, \rho' \in States(A)$. Then ρ and ρ' are said to be *equal* if they agree on the values of all variables. Formally,

$$\rho = \rho' \iff (\forall x \in Var)\ \ (\ \rho(x) = \rho'(x)\ )$$

6.1.3 Lemma. *Let* $\rho \in States(A)$. *Also let* $x \in Var_s$ *and* $y \in Var_{s'}$ *for any* $s, s' \in S$. *Then for any* $a, b \in A_s$, *and for any* $c \in A_{s'}$,

$$\rho\{ a / x \}\{ c / y \}\{ b / x \} = \rho\{ c / y \}\{ b / x \}$$

Proof. Exercise

6.1.4 Expressions.

Given a collection of variables whose values are known via some state ρ, we can form new values by applying operations to combinations of the variables and perhaps also some constants; for example, given variables x and y of sort natural number, and a state ρ under which x and y have the values 3 and 4 respectively, '$x+y$' is an expression whose value under ρ is 7.

Below, we formalise the concept of an expression by defining a syntax for expressions of sort s (that is, expressions which evaluate to a value in A_s), together with a semantics for expressions which is a rule which tells us how to evaluate a given expression under a given state. Also, it is important that we have regard for how long it takes to evaluate a given expression (under a given state): this we formalise via a length of computation function for an expression.

Syntax. We define the S-indexed family $\mathrm{EXP}(\Sigma)$ of *expressions over* Σ by

$$\mathrm{EXP}(\Sigma) = \langle \mathrm{EXP}(\Sigma)_s : s \in S \rangle$$

where for each $s \in S$ the collection $\mathrm{EXP}(\Sigma)_s$ of *expressions of sort* s is defined uniformly in s by structural induction as follows:

Basis.

(i) *Constants.* Suppose $c \in \Sigma_{\lambda,s}$ for some $s \in S$. Then $c \in \mathrm{EXP}(\Sigma)_s$.

(ii) *Variables.* Suppose $x \in Var_s$ for some $s \in S$. Then $x \in \mathrm{EXP}(\Sigma)_s$.

Induction.

(iii) *Operations.* Suppose $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and some $s \in S$. If $e_i \in \mathrm{EXP}(\Sigma)_{w_i}$ for $i = 1,\ldots,n = |w|$, then $\sigma(e_1,\ldots,e_n) \in \mathrm{EXP}(\Sigma)_s$. □

Semantics. For each $e \in \mathrm{EXP}(\Sigma)$, the *value* of e under a given state $\rho \in States(A)$ is denoted by $E_A(e)(\rho)$ where E_A is the S-indexed family

$$E_A = \langle E_A^s : s \in S \rangle$$

of *evaluation mappings* $E_A^s : \mathrm{EXP}(\Sigma)_s \longrightarrow [States(A) \longrightarrow A_s]$. Each mapping E_A^s (ambiguously denoted E_A) is defined uniformly in s by induction on the structure of an expression $e \in \mathrm{EXP}(\Sigma)_s$ as follows:

Basis.

(i) *Constants.* If $e = c$ for some $c \in \Sigma_{\lambda,s}$, then

$$E_A(e) : States(A) \longrightarrow A_s$$

is defined by

$$(\forall \rho \in States(A))\ (\ E_A(e)(\rho) = c^A\ )$$

(ii) *Variables.* If $e = x$ for some $x \in Var_s$, then

$$E_A(e) : States(A) \longrightarrow A_s$$

is defined by

$$(\forall \rho \in States(A))\ \ (E_A(e)(\rho) = \rho(x))$$

Induction.

(iii) *Operations.* Suppose $e = \sigma(e_1,...,e_n)$ where for some $w \in S^+$ and some $s \in S$, $\sigma \in \Sigma_{w,s}$, and $e_i \in \mathrm{EXP}(\Sigma)_{w_i}$ for $i = 1,...,n = |w|$. Then

$$E_A(e): States(A) \longrightarrow A_s$$

is defined by

$$(\forall \rho \in States(A))\ \ (E_A(e)(\rho) = \sigma^A(E_A(e_1)(\rho),...,E_A(e_n)(\rho)))$$

Discussion. $E_A(e)(\rho)$ is the evaluation of e in A under ρ. When e is a constant (expression), evaluation of e is independent of the state ρ; the expression simply evaluates to the value of the constant in A. When e is a variable x, then the expression denotes the value held by x under ρ, viz $\rho(x)$. Finally, when e is of the form $\sigma(e_1,...,e_n)$, to evaluate e we first evaluate the subexpressions $e_1,...,e_n$ and then we supply the resulting values to σ^A as arguments.

Notice that our formal expressions are written in a functional style rather than in the usual infix notation. For example, if Σ is the signature of Peano Arithmetic $\mathbb{N}$, then a typical expression over Σ such as $0+x\times y$ is written as $+(0,\times(x,y))$. Actually, technically, '0' is a number and not a piece of syntax, and thus properly one writes $+(\mathrm{zero},\times(x,y))$ where $\mathrm{zero} \in \Sigma_{\lambda,N}$ is such that $\mathrm{zero}^{\mathbb{N}} = 0$.

Example. Let Σ be the signature of Peano Arithmetic $\mathbb{N}$. Suppose $\rho: Var_N \longrightarrow \mathbb{N}$ satisfies $\rho(x) = 3$ and $\rho(y) = 2$ for some $x, y \in Var_N$. Then the value of the expression $+(\mathrm{zero},\times(x,y)) \in \mathrm{EXP}(\Sigma)_N$ under ρ is calculated as follows:

$$\begin{aligned} E_{\mathbb{N}}(+(\mathrm{zero},\times(x,y)))(\rho) &= +^{\mathbb{N}}(E_{\mathbb{N}}(\mathrm{zero})(\rho),E_{\mathbb{N}}(\times(x,y))(\rho)) \\ &= +^{\mathbb{N}}(\mathrm{zero}^{\mathbb{N}},\times^{\mathbb{N}}(E_{\mathbb{N}}(x)(\rho),E_{\mathbb{N}}(y)(\rho))) \\ &= +^{\mathbb{N}}(0,\times^{\mathbb{N}}(\rho(x),\rho(y))) \\ &= +^{\mathbb{N}}(0,\times^{\mathbb{N}}(3,2)) \\ &= 6 \end{aligned}$$

Performance. Let P be a performance measure for A which is based on clock C. For each $e \in \mathrm{EXP}(\Sigma)$, the *length of evaluation function for e with respect to P* is denoted by $\lambda_P^{EXP}(e)$ where λ_P^{EXP} is the S-indexed family

$$\lambda_P^{EXP} = <\lambda_P^s : s \in S>$$

of mappings $\lambda_P^s : \mathrm{EXP}(\Sigma)_s \longrightarrow [States(A) \longrightarrow C^+]$. Each mapping λ_P^s (ambiguously denoted λ_P^{EXP}) is defined uniformly in s by induction on the structure of an expression $e \in \mathrm{EXP}(\Sigma)_s$ as follows:

Basis.

(i) *Constants.* If $e = c$ for some $c \in \Sigma_{\lambda,s}$, then

$$\lambda_P^{EXP}(e): States(A) \longrightarrow C^+$$

is defined by

$$(\forall \rho \in States(A))\ \ (\lambda_P^{EXP}(e)(\rho) = c^P)$$

(ii) *Variables.* If $e = x$ for some $x \in Var_s$, then

$$\lambda_P^{EXP}(e) : States(A) \longrightarrow C^+$$

is defined by

$$(\forall \rho \in States(A))\ (\ \lambda_P^{EXP}(e)(\rho) = 1\ )$$

Induction.

(iii) *Operations.* Suppose $e = \sigma(e_1,\ldots,e_n)$ where for some $w \in S^+$ and some $s \in S$, $\sigma \in \Sigma_{w,s}$, and $e_i \in \mathrm{EXP}(\Sigma)_{w_i}$ for $i = 1,\ldots,n = |w|$. Then

$$\lambda_P^{EXP}(e) : States(A) \longrightarrow C^+$$

is defined by

$$(\forall \rho \in States(A))\ (\ \lambda_P^{EXP}(e)(\rho) = \sigma^P(E_A(e_1)(\rho),\ldots,E_A(e_n)(\rho)) + \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_n)(\rho)\}\ )$$

Discussion. The complexity of evaluating a given expression is arrived at by building on the notion of a performance measure. The complexity of evaluating a constant (expression) is as determined by the performance measure. For the complexity of evaluating $\sigma(e_1,\ldots,e_n)$, there are two stages to the evaluation: first we evaluate the arguments $e_1,\ldots,e_n$, and then we apply the operation σ. Consequently, we take the complexity of evaluating $\sigma(e_1,\ldots,e_n)$ to be the sum of the complexities of these two stages: we imagine the subexpressions $e_1,\ldots,e_n$ to be evaluated in parallel, and so we take the complexity of this stage to be the maximum evaluation time over all the subexpressions. Of course, the complexity of evaluating σ applied to its arguments is just as prescribed by the performance measure P. Finally, as in the case of the projection functions in PR (see Section 3.3.3), to access the value held by a variable is to access a single datum, and so we charge a single unit for evaluation.

Example. Let P be a performance measure for $\mathbb{N}$. Then we can calculate the cost of evaluating the expression of the previous example as follows:

Let $\rho \in States(\mathbb{N})$ satisfy $\rho(x) = 3$ and $\rho(y) = 2$. Then,

$$\begin{aligned}\lambda_P^{EXP}(+(\mathrm{zero},\times(x,y)))(\rho) &= +^P(E_{\mathbb{N}}(\mathrm{zero})(\rho),E_{\mathbb{N}}(\times(x,y)(\rho)) \\ &\quad + \max\{\lambda_P^{EXP}(\mathrm{zero})(\rho),\lambda_P^{EXP}(\times(x,y))(\rho)\} \\ &= +^P(0,\times^{\mathbb{N}}(\rho(x),\rho(y))) \\ &\quad + \max\{\mathrm{zero}^P,\times^P(\rho(x),\rho(y))+\max\{\lambda_P^{EXP}(x)(\rho),\lambda_P^{EXP}(y)(\rho)\}\} \\ &= +^P(0,\times^{\mathbb{N}}(2,3)) \\ &\quad + \max\{\mathrm{zero}^P,\times^P(2,3)+\max\{1,1\}\} \\ &= +^P(0,6) \\ &\quad + \max\{\mathrm{zero}^P,\times^P(2,3)+1\}\end{aligned}$$

Thus, if for example $\mathrm{zero}^P = 1$ and $+^P(n,m) = 1+m$ and $\times^P(n,m) = 1+m^2$ for each $n,m \in \mathbb{N}$, then

$$\lambda_{\rho}^{EXP}(+(\text{zero},\times(x,y)))(\rho) = 1+6+\max\{\,1,1+9+1\,\}$$
$$= 7+11 = 18$$

6.1.5 The Syntax of PIT.

We define the collection PIT(Σ) of *PIT programs* or *PIT statements* by structural induction as follows:

Basis.

(i) *Multiple Assignment Statements*. Suppose for some $r > 0$, that S is the statement

$$S = x_1,\ldots,x_r := e_1,\ldots,e_r$$

where for $i = 1,\ldots,r$, $x_i \in Var_{s_i}$ and $e_i \in \text{EXP}(\Sigma)_{s_i}$ for some sorts $s_1,\ldots,s_r \in S$. If $x_1,\ldots,x_r$ are distinct variables, then $S \in \text{PIT}(\Sigma)$.

Induction.

(ii) *Sequencing*. Suppose S is the statement

$$S = S_1 \,;\, S_2$$

for some $S_1, S_2 \in \text{PIT}(\Sigma)$. Then $S \in \text{PIT}(\Sigma)$.

(iii) *Conditional*. Suppose S is the statement

$$S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

for some $b \in \text{EXP}(\Sigma)_B$ and for some $S_1, S_2 \in \text{PIT}(\Sigma)$. Then $S \in \text{PIT}(\Sigma)$.

(iv) *Bounded Iteration*. Suppose S is the statement

$$S = \textbf{do } e \textbf{ times } S_0 \textbf{ od}$$

for some $e \in \text{EXP}(\Sigma)_\tau$ and some $S_0 \in \text{PIT}(\Sigma)$. Then $S \in \text{PIT}(\Sigma)$. (Here we refer to S as a *loop*, and we call S_0 the *body* of the loop.)

6.1.6 The Semantics of PIT.

We will now define the semantics of PIT. Essentially, the idea is that given some *initial* state $\rho \in \textit{States}(A)$, the effect of executing a PIT program is to modify or transform ρ to some *final* state $\rho' \in \textit{States}(A)$. In more detail, ρ tells us what the initial value of every variable is prior to execution. During execution, values of the variables are modified by assignment statements, and upon termination of the program, the final state tells us what the final value of each variable is.

The state-transformer semantics of an arbitrary PIT program is defined as follows:

Definition. For each $S \in \text{PIT}(\Sigma)$, the *meaning* of S *over* A is denoted by $M_A(S)$ where

$$M_A : \text{PIT}(\Sigma) \longrightarrow [\textit{States}(A) \longrightarrow \textit{States}(A)]$$

is defined by induction on the structural complexity of arguments S as follows:

Basis.

(i) *Multiple Assignment Statements.* If S is the statement

$$S = x_1,\ldots,x_r := e_1,\ldots,e_r$$

for some $x_1,\ldots,x_r \in Var$ and some $e_1,\ldots,e_r \in \mathrm{EXP}(\Sigma)$, then $M_A(S)$ is defined by

$$(\forall \rho \in States(A))\ \ (M_A(S)(\rho) = \rho\{E_A(e_1)(\rho)/x_1\}\{\cdots\}\{E_A(e_r)(\rho)/x_r\})$$

Induction.

(ii) *Sequencing.* Suppose S is the statement

$$S = S_1 ; S_2$$

for some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$. Then we define $M_A(S)$ by

$$(\forall \rho \in States(A))\ \ (M_A(S)(\rho) = M_A(S_2)(M_A(S_1)(\rho)))$$

(iii) *Conditional.* Suppose S is the statement

$$S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

for some $b \in \mathrm{EXP}(\Sigma)_B$ and for some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$. Then we define $M_A(S)$ by

$$(\forall \rho \in States(A))\ \ M_A(S)(\rho) = \begin{cases} M_A(S_1)(\rho) & \text{if } E_A(b)(\rho) = tt \\ M_A(S_2)(\rho) & \text{if } E_A(b)(\rho) = ff \end{cases}$$

(iv) *Bounded Iteration.* Suppose S is the statement

$$S = \textbf{do } e \textbf{ times } S_0 \textbf{ od}$$

for some $e \in \mathrm{EXP}(\Sigma)_\tau$ and some $S_0 \in \mathrm{PIT}(\Sigma)$. Then $M_A(S)$ is defined by

$$(\forall \rho \in States(A))\ \ (M_A(S)(\rho) = \rho_n)$$

where $n = E_A(e)(\rho)$,

$$\rho_0 = \rho$$

and for any $k \geq 0$,

$$\rho_{k+1} = M_A(S_0)(\rho_k)$$

Discussion. There are two features of the above definition that require comment. First consider the defining line for $M_A(S)$ when S is a multiple assignment statement (see clause (i) above). At first sight it appears that the order in which expressions are evaluated is significant: as the defining line reads, it appears that e_1 is evaluated first and e_r last. In fact this is not the case: if a and b are any values, and if x and y are *different* variables, then the two states $\rho\{a/x\}\{b/y\}$ and $\rho\{b/y\}\{a/x\}$ are equal. Since $x_1,\ldots,x_r$ are assumed to be different variables, the order in which we write down the state modifiers $\{E_A(e_i)(\rho)/x_i\}$ is not significant. For example, we could have equivalently defined $M_A(S)(\rho)$ by

$$(\forall \rho \in States(A))\ \ (M_A(S)(\rho) = \rho\{E_A(e_r)(\rho)/x_r\}\{\cdots\}\{E_A(e_1)(\rho)/x_1\})$$

The second point to notice is that for any $S \in \mathrm{PIT}(\Sigma)$, the function $M_A(S)$ is always *total*; for any $\rho \in States(A)$ the state $M_A(S)(\rho)$ is always well-defined in the sense that it can be calculated in finitely many steps. We infer from this observation that every PIT program *always terminates* (when executed from any initial state).

6.1.7 Performance of PIT Programs.

Let P be a performance measure for A which is based on clock C.

For each $S \in \mathrm{PIT}(\Sigma)$, the *length of computation function* for S *with respect to* P is denoted by $\lambda_P^{PIT}(S)$ where

$$\lambda_P^{PIT}: \mathrm{PIT}(\Sigma) \longrightarrow [States(A) \longrightarrow C^+]$$

is defined by induction on the structural complexity of arguments S as follows:

Basis.

(i) *Multiple Assignment Statements.* If S is the statement

$$S = x_1,\ldots,x_r := e_1,\ldots,e_r$$

for some $x_1,\ldots,x_r \in Var$ and some $e_1,\ldots,e_r \in \mathrm{EXP}(\Sigma)$, then $\lambda_P^{PIT}(S)$ is defined by

$$(\forall \rho \in States(A))\ (\lambda_P^{PIT}(S)(\rho) = \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_r)(\rho)\})$$

Induction.

(ii) *Sequencing.* Suppose S is the statement

$$S = S_1 ; S_2$$

for some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$. Then we define $\lambda_P^{PIT}(S)$ by

$$(\forall \rho \in States(A))\ (\lambda_P^{PIT}(S)(\rho) = \lambda_P^{PIT}(S_1)(\rho) + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho)))$$

(iii) *Conditional.* Suppose S is the statement

$$S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

for some $b \in \mathrm{EXP}(\Sigma)_{\mathbf{B}}$ and for some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$. Then we define $\lambda_P^{PIT}(S)$ by

$$(\forall \rho \in States(A))\ \lambda_P^{PIT}(S)(\rho) = \lambda_P^{EXP}(b)(\rho) + \begin{cases} \lambda_P^{PIT}(S_1)(\rho) & \text{if } E_A(b)(\rho) = tt \\ \lambda_P^{PIT}(S_2)(\rho) & \text{if } E_A(b)(\rho) = ff \end{cases}$$

(iv) *Bounded Iteration.* Suppose S is the statement

$$S = \textbf{do } e \textbf{ times } S_0 \textbf{ od}$$

for some $e \in \mathrm{EXP}(\Sigma)_{\mathbf{T}}$ and some $S_0 \in \mathrm{PIT}(\Sigma)$. Then $\lambda_P^{PIT}(S)$ is defined by

$$(\forall \rho \in States(A))\ (\lambda_P^{PIT}(S)(\rho) = \lambda_P^{EXP}(e)(\rho) + \lambda_n)$$

where $n = E_A(e)(\rho)$,

$$\lambda_0 = 0,$$

and for any $k \geq 0$,

$$\lambda_{k+1} = \lambda_k + \lambda_P^{PIT}(S_0)(\rho_k)$$

where ρ_k is as defined in Section 6.1.6 clause (iv).

Discussion. The complexity or performance of a PIT program is arrived at by building on the complexity of expression evaluation in the following way:

We imagine each assignment in a multiple assignment statement to be executed in parallel, and so we take the complexity of execution to be the longest time required to execute any of the individual assignments. For sequencing $S_1;S_2$, we first execute S_1, and then S_2, and so we charge the sum of the costs of executing each component statement (from the appropriate initial state). For the conditional

statement **if** b **then** S_1 **else** S_2 **fi**, we must first evaluate the Boolean expression before executing either S_1 or S_2, and so we charge the sum of the costs of evaluating the Boolean expression and the cost of executing either S_1 or S_2 as appropriate. The cost of executing a loop **do** e **times** S_o **od** is arrived at along similar principles.

6.1.8 Examples.

Here are three examples of PIT programs. The first is a trivial example that illustrates how to work with state-transformer semantics. The second example illustrates the idea that a PIT program can be used to compute a function; this is the subject of the next section. The final example is a PIT simulation of the OE sorter; this serves to illustrate the important idea that synchronous algorithms can be simulated directly in PIT (that is, without using PR).

Swapping. Swapping the values held by two program variables x and y can be accomplished via the single PIT statement $S = x,y := y,x$. We can establish that executing S swaps the values held by x and y in the following way. Given an initial state ρ, the initial values of x and y are $\rho(x)$ and $\rho(y)$ respectively. If we now execute S from this initial state, then upon termination the final values of x and y are $\rho'(x)$ and $\rho'(y)$ respectively, where $\rho' = M_A(S)(\rho)$. Thus to show S swaps the values held by x and y we must show $\rho'(x) = \rho(y)$ and $\rho'(y) = \rho(x)$.

First, since S is a multiple assignment statement, we have for any initial state ρ,

$$\begin{aligned} M_A(S)(\rho) &= \rho\{ E_A(y)(\rho)/x \}\{ E_A(x)(\rho)/y \} \\ &= \rho\{ \rho(y)/x \}\{ \rho(x)/y \} \end{aligned}$$

Thus,

$$\begin{aligned} \rho'(x) &= M_A(S)(\rho)(x) \\ &= \rho\{ \rho(y)/x \}\{ \rho(x)/y \}(x) \\ &= \rho\{ \rho(y)/x \}(x) \end{aligned}$$

(using the definition of a variant of a state; see Section 6.1.2)

$$= \rho(y)$$

(again from the definition of a variant of a state.) Thus the final value of x is $\rho(y)$, that is, y's initial value. In a similar way it is easy to show that $\rho'(y) = \rho(x)$; that is, that y's final value is the initial value of x. Thus S does indeed swap the values of x and y as claimed.

Factorial. For each $n \in \mathbb{N}$ let $n!$ denote the *factorial* of n. That is, let $n!$ be the number defined by

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

Also let Σ be the signature of Peano Arithmetic $\mathbb{N}$. Then a PIT(Σ)-program to compute $n!$ for each $n \in \mathbb{N}$ is as follows:

$$S_{fact} = \begin{cases} f,m := 1,1; \\ \textbf{do } num \textbf{ times} \\ \quad f,m := \times(m,f),succ(m) \\ \textbf{od} \end{cases}$$

In the above program, we imagine num, f, and m to be (distinct) variables of sort natural number whose purposes are as follows: the variable num is intended to hold the value n for which we wish to compute $n!$; the variable f acts as an accumulator in which we build up the value of $n!$, and the variable m is the multiplier from which we build $n!$.

Let us see how S_{fact} above computes $n!$ in some simple cases: First, in all cases, the initial multiple assignment is executed, leaving both f and m with the value 1. If num holds the value 0, then the *body* of the loop is executed zero times, that is, it is not executed at all; thus S_{fact} will terminate with f holding the value 1, which is correct since $0!=1$ by definition. If num holds the value 1, then the body of the loop is executed once: f is multiplied by m, and m is incremented by 1; thus on termination of S_{fact}, f holds the value $1\times1=1$, which is again correct since $1!=1$. If num holds the value 2, then the body of the loop is executed twice: after the first execution f and m will hold the values 1 and 2 respectively, as in the case above (when the value of num is 1); on the second execution of the loop, f is multiplied by m leaving f with the value $2\times1=2$, and m is incremented by 1 to 3; thus on termination the value of f is $2=2!$, and m is left with the value 3 ready to compute $3\times2=3!$.

It is not difficult to see that, in general, if the value of num is $n>0$, then the loop is executed n times, and immediately prior to the nth execution of the body of the loop, f and m hold the values $(n-1)!$ and n respectively; thus upon completion of the last execution of the loop, f is left with the value $n\times(n-1)!=n!$.

We leave it to the interested reader to prove that S_{fact} computes the factorial function in the following sense:

$$(\forall \rho \in States(\mathbb{N}))\ \ (M_{\mathbb{N}}(S_{fact})(\rho)(f) = \rho(num)!\ )$$

Simulating OE. Let us write a PIT program that simulates the OE sorter of Chapter 2. Recall that the informal description of the OE sorter determined an algebra $A=A_{OE}$ comprising the three carrier sets $T=T_{OE}$, $\mathbb{B}$, and D, and operations $f_1,\ldots,f_n$ where for $i=1,\ldots,n$ f_i was the function that specified module m_i; see Section 2.3.6 for the definitions of $f_1,\ldots,f_n$. Now let $S=\{\mathrm{T,B,D}\}$ and let Σ be the S-sorted signature of A. Then Σ contains a symbol σ_i to name f_i for $i=1,\ldots,n$; more precisely, for $i=1,n$ we have $\sigma_i \in \Sigma_{\mathrm{TDD,D}}$ and for $i=2,\ldots,n-1$ we have $\sigma_i \in \Sigma_{\mathrm{TDDD,D}}$ with, for example,

$$\sigma_i^A(t,a,l,v,r) = f_i(t,a,l,v,r) = \begin{cases} a & \text{if } t \textit{ mod } (n+1)=0 \\ \min\{v,r\} & \text{if } t \textit{ mod } (n+1) \textit{ odd} \\ \max\{l,v\} & \text{if } t \textit{ mod } (n+1) \textit{ even}, \neq 0 \end{cases}$$

for each $t \in T$ and $a,l,v,r \in D$, in the case that i is odd, $\neq 1$. Now, according to the general definition of the sort set $\underline{S}$, the $\underline{S}$-sorted signature $\underline{\Sigma}$, and $\underline{\Sigma}$-algebra $\underline{A}$ given in Section 3.1.8, $\underline{S}$ contains a new symbol $\underline{\mathrm{D}}$ which is the formal name of $\underline{A}_{\underline{\mathrm{D}}}=[T\longrightarrow A_D]=[T\longrightarrow D]$, $\underline{\Sigma}$ contains a new symbol $\mathrm{eval} \in \underline{\Sigma}_{\mathrm{TD,D}}$, and in addition to the new carrier $[T\longrightarrow D]$, $\underline{A}$ has the new operation

$\text{eval}^{\underline{A}} = eval : T \times [T \longrightarrow D] \longrightarrow D$ defined by $eval(t, \underline{a}) = \underline{a}(t)$ for each $t \in T$ and $\underline{a} : T \longrightarrow D$.

Since $\underline{\Sigma}$ is a standard signature the collection $\text{PIT}(\underline{\Sigma})$ is well-defined, and since $\underline{A}$ is standard the function $M_{\underline{A}} : \text{PIT}(\underline{\Sigma}) \longrightarrow [States(\underline{A}) \longrightarrow States(\underline{A})]$ is also well-defined. We construct a simulation $S_{OE} \in \text{PIT}(\underline{\Sigma})$ as follows:

To simulate OE we first introduce a variable $v_i \in Var_{\underline{D}}$ to represent the value held by m_i for $i = 1,\ldots,n$. Now, the function f_i tells us the value held by m_i at time $t+1$ in terms of the values held by $m_1,\ldots,m_n$ and the value supplied by source In_i at time t. Since σ_i names f_i, the statement

$$v_i := \sigma_i(t, a, v_{i-1}, v_i, v_{i+1})$$

will simulate the t th step performed by m_i (in the case $2 \le i \le n$) when 'a' represents the value supplied by In_i at time t. (Note that here 't' is a program variable $t \in Var_{\tau}$; by 'the time t' we really mean 'the value of t under the current state'.)

We can simulate the loading of data into the OE network by means of the steam evaluation functions. Suppose in is a variable of sort $\underline{D}$, that is, suppose $in \in Var_{\underline{D}}$. Then for any $\rho \in States(\underline{A})$ we have $\rho(in) \in \underline{A}_{\underline{D}}$, that is, $\rho(in) : T \longrightarrow D$. To access an element of the stream denoted by in we must apply $eval$ to the time t and the stream. Consider the expression $e = \text{eval}(t, in)$; then $e \in \text{EXP}(\underline{\Sigma})_{\underline{D}}$, and by definition of $E_{\underline{A}}$ the value of e under any $\rho \in States(\underline{A})$ is calculated as follows:

$$\begin{aligned} E_{\underline{A}}(e)(\rho) &= E_{\underline{A}}(\text{eval}(t, in))(\rho) \\ &= \text{eval}^{\underline{A}}(E_{\underline{A}}(t)(\rho), E_{\underline{A}}(in)(\rho)) \\ &= eval(\rho(t), \rho(in)) \\ &= \rho(in)(\rho(t)) \end{aligned}$$

Thus if $\rho(in)$ is the stream $\underline{a} : T \longrightarrow D$ then as the value of t under ρ varies 0,1,2,... the expression $\text{eval}(t, in)$ evaluates to $\underline{a}(0), \underline{a}(1), \underline{a}(2), \ldots$.

In the case of the OE network, the input is a stream $\underline{a} = (\underline{a}_1, \ldots, \underline{a}_n) : T \longrightarrow D^n$ where for $i = 1,\ldots,n$ $\underline{a}_i : T \longrightarrow D$ is the steam supplied by In_i. Let $in_i \in Var_{\underline{D}}$ for $i = 1,\ldots,n$ and suppose $\rho \in States(\underline{A})$ is such that $\rho(in_i) = \underline{a}_i$ for $i = 1,\ldots,n$. If we additionally denote $\rho(v_{i-1})$, $\rho(v_i)$, and $\rho(v_{i+1})$ by l, v, and r respectively, then for $2 \le i \le n-1$ we have

$$\begin{aligned} M_{\underline{A}}(v_i := \sigma_i(t, \text{eval}(t, in_i), v_{i-1}, v_i, v_{i+1}))(\rho) &= \rho\{f_i(\rho(t), \underline{a}_i(\rho(t)), l, v, r) / v_i\} \\ &= \rho\{f_i(k, \underline{a}_i(k), l, v, r) / v_i\} \end{aligned}$$

when $\rho(t) = k$; here we see that v_i is assigned the value held by m_i at time $k+1$.

Intuitively then, the following program $S_{OE} \in \text{PIT}(\underline{\Sigma})$ should simulate OE:

$$S_{OE} = \left\{ \begin{array}{l} t := 0; \\ \textbf{do } l \textbf{ times} \\ \quad v_1, \ldots, v_n, t := e_1, \ldots, e_n, t+1 \\ \textbf{od} \end{array} \right.$$

where

$$e_1 = \sigma_1(t, \text{eval}(t, in_1), v_1, v_2)$$

and for $i = 2,\ldots,n-1$,

$$e_i = \sigma_i(t, \text{eval}(t, in_i), v_{i-1}, v_i, v_{i+1})$$

and

$$e_n = \sigma_n(t, \text{eval}(t, in_n), v_{n-1}, v_n)$$

Note that we cannot simulate OE for infinitely many time cycles in PIT, since PIT only has a *bounded* loop construct. However, in practice we would only ever want to look at the values held by OE's modules for a finite time, and thus the bounded loop is appropriate. For example, in S_{OE}, we have said to simulate OE for l cycles; if $\rho(l) = m(n+1)$ for some $m \geq 1$ then this is enough time to see OE sort the first m vectors of input which are read into the network. (Recall that the period of OE is $n+1$.)

Readers may convince themselves that S_{OE} simulates OE by executing S_{OE} on typical input. For example, take $D = \mathbb{N}$, $n = 6$, and calculate $M_{\underline{A}}(S_{OE})(\rho)$ where $\rho \in States(\underline{A})$ satisfies $\rho(l) = n+1$, $\rho(in_i)(0) = n+1-i$ for $i = 1,\ldots,n$, say. However, officially, the behaviour of OE is defined by the value function V_{OE}, and thus to claim that S_{OE} simulates OE is to claim that S_{OE} *implements* or *computes* V_{OE}, or that S_{OE} is a *correct* simulation of OE. Formally, we offer the following:

Lemma. *For any initial state* $\rho \in States(\underline{A})$, *define* ρ_k *for each* $k \geq 0$ *by*

$$\rho_0 = M_{\underline{A}}(t := 0)(\rho)$$

$$\rho_{k+1} = M_{\underline{A}}(S_0)(\rho_k)$$

where S_0 *is the body of the do-loop in* S_{OE}. *Also let* $\rho(in_i) = \underline{a}_i : T \longrightarrow D$ *and* $\rho(v_i) = x_i \in D$ *for* $i = 1,\ldots,n$. *Then for* $k = 0,1,2,\ldots$,

$$(\rho_k(v_1),\ldots,\rho_k(v_n)) = V_{OE}(k, \underline{a}, x)$$

where $\underline{a} = (\underline{a}_1,\ldots,\underline{a}_n) : T \longrightarrow D^n$ *and* $x = (x_1,\ldots,x_n) \in D^n$.

Proof. Exercise.

Discussion. The last example serves to illustrate the important fact that synchronous algorithms may be simulated directly in PIT; it is not difficult to see how a general synchronous network N could be simulated by a PIT program S_N by modifying the OE example. Moreover, since a PIT program is a formal object, it is clear that we could alternatively formalise synchronous networks by defining the behaviour of the network to be the state transformation $M_{\underline{A}}(S_N)$. Thus, in principle, we now have two ways of formalising synchronous algorithms: by using a PR scheme $\alpha_N \in \text{PR}(\underline{\Sigma})$ (cf. Notation 3.4.5) and by using a PIT program $S \in \text{PIT}(\underline{\Sigma})$. As we have said in the introduction to this chapter we will later prove that these methods are equivalent. Actually, it is the language FPIT that we show is equivalent to PR; FPIT can be thought of as an extension of PIT and thus synchronous algorithms can also be simulated directly in FPIT. See Martin and Tucker[1987] for an account of using CARESS, the implemented version of FPIT (cf. Section 1.3), to simulate synchronous algorithms.

6.2 PROGRAMMING WITH PROCEDURES.

In this section we will consider how PIT may be extended to include programs which involve *function procedures*. Function procedures are PIT programs which compute a function on the underlying data set (in a sense to be defined).

Below, we will first consider what it means for a PIT program to compute a function following the semantic treatment in Tucker and Zucker[1987] (work of 1979) This done, we will define function procedures using syntactic conditions adapted from Jervis[1988] (to appear) We also explore the combined use of augmentation (see Section 3.1.5) and function procedures in the *top-down design* of algorithms; ultimately we will implement each PR scheme in a top-down way using FPIT. (As we will see later, a program in FPIT is a more general kind of function procedure.)

6.2.1 Functionality.

We have seen two examples of how a PIT program S can compute a function f in an intuitive sense: the programs S_{fact} and S_{OE} of the previous examples computed a function f (the factorial function and V_{OE} respectively) in the sense that to compute $f(a_1,...,a_n)$ we chose an initial state under which S's input variables held the values $a_1,...,a_n$, and then under the final state S's output variables collectively held $f(a_1,...,a_n)$.

Let us introduce some definitions that allow us to formalise this idea:

6.2.2 Definitions.

(i) Let $w \in S^+$ with $|w| = n$. Then we define Var_w to be the set of all vectors $X = (X_1,...,X_n)$ such that $\{X_1,...,X_n\}$ are all distinct variables with $X_i \in Var_{w_i}$ for $i = 1,...,n$.

(ii) For each $w \in S^+$, and for each $X \in Var_w$, we define X_i to be the ith variable comprising X for $i = 1,...,n = |w|$. Thus, if $X = (x_1,...,x_n)$, then $X_i = x_i$ for $i = 1,...,n$.

(iii) We define π_A to be the S^+-indexed family

$$\pi_A = <\pi_A^w : w \in S^+>$$

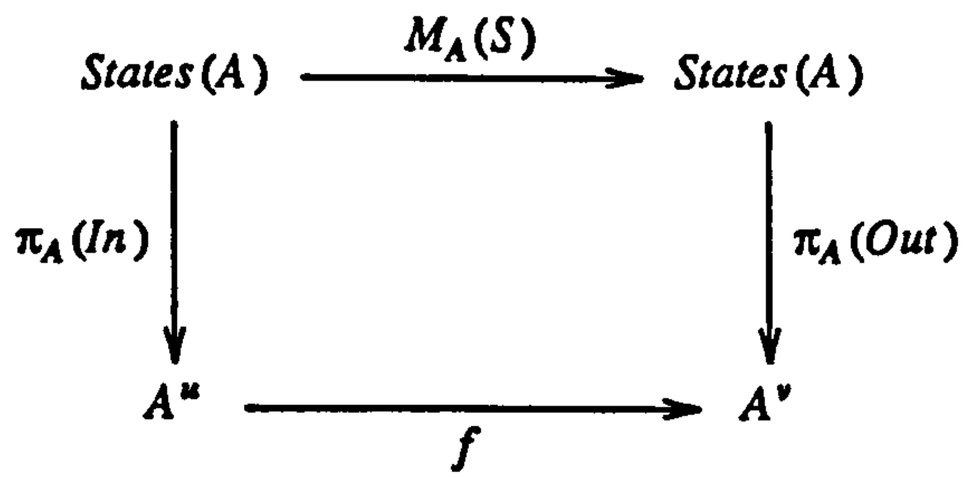
of *state projection functions* $\pi_A^w : Var_w \longrightarrow [States(A) \longrightarrow A^w]$. For each $w \in S^+$ with $|w| = n$, π_A^w (ambiguously denoted π_A) is defined by

$$(\forall X \in Var_w)(\forall \rho \in States(A))\ (\ \pi_A(X)(\rho) = (\rho(X_1),...,\rho(X_n))\ )$$

Definition. Let $S \in \mathrm{PIT}(\Sigma)$, and let $In \in Var_u$ and $Out \in Var_v$ for any $u,v \in S^+$. Then we refer to (S,In,Out) as an *input/output-triple of arity* (u,v). (More briefly, we say (S,In,Out) is an 'i/o-triple'.)

6.2.3 Definition.

Let (S,In,Out) be an i/o triple of arity (u,v) for some $u,v \in S^+$. Also let $f : A^u \longrightarrow A^v$ be any function. Then we say S *computes* f *relative to* In and Out if the following diagram commutes for every $\rho \in States(A)$:



That is, if

$$(\forall \rho \in States(A))\ \ (\pi_A(Out)(M_A(S)(\rho)) = f(\pi_A(In)(\rho)))$$

Additionally, we say an i/o-triple (S, In, Out) is *functional* if there is some function f which S computes relative to In and Out.

Example. Let $S = x,y := y,x$. Also let $In = Out = (x,y)$. Then we can show S computes f relative to In and Out where f is the function defined by $f(a,b) = (b,a)$.

Referring back to Examples 6.1.8, choose $\rho \in States(A)$ and calculate as follows:

$$\pi_A(Out)(M_A(S)(\rho)) = \pi_A(x,y)(M_A(S)(\rho))$$

(by definition of Out)

$$= (M_A(S)(\rho)(x),\ M_A(S)(\rho)(y))$$

(by definition of π_A)

$$= (\rho(y),\rho(x))$$

(see Examples 6.1.8)

$$= f(\rho(x),\rho(y))$$

(by definition of f)

$$= f(\pi_A(x,y)(\rho))$$

(by definition of π_A)

$$= f(\pi_A(In)(\rho))$$

(by definition of In).

Thus S computes f relative to In and Out as claimed.

Note. Let (S, In, Out) be an i/o-triple of arity (u,v), and let $f : A^u \longrightarrow A^v$. Suppose S computes f relative to In and Out. Is it the case that we can use S to compute $f(a)$ for *every* $a \in A^u$? If not, then surely our definition of a program computing a function is a bad one! Fortunately the answer is 'yes'. To see this, choose $a = (a_1,\ldots,a_n) \in A^u$ (so $|u| = n$). Now, since $States(A)$ includes *all* maps from variables into A, there is certainly a state ρ such that $\rho(In_i) = a_i$ for $i = 1,\ldots,n$ (assuming $In_1,\ldots,In_n$ are distinct variables). Of course, we now have $\pi_A(In)(\rho) = a$. In other words $\pi_A(In) : States(A) \longrightarrow A^u$ is a surjection, and so there is always a state which we can use to load the variables $In_1,\ldots,In_n$ with $a_1,\ldots,a_n$ respectively, and thus we *can* use S to compute $f(a)$ for any $a \in A^u$.

Exercise. Let S_{OE} be as in Examples 6.1.8. Prove that S_{OE} computes V_{OE} relative to $In=(l,in_1,\ldots,in_n,v_1,\ldots,v_n)$ and $Out=(v_1,\ldots,v_n)$. □

There are two questions concerning the functionality of i/o-triples which immediately suggest themselves, these being:

(i) If S computes f relative to In and Out, then is f unique?

(ii) Is it the case that for every (S,In,Out) that S computes *some* f relative to In and Out?

In answer to the first question we have:

6.2.4 Lemma. *Let (S,In,Out) be an i/o-triple of arity (u,v) for some $u,v \in S^+$. If S computes some $f:A^u \longrightarrow A^v$ relative to In and Out then f is unique.*

Proof. Suppose S computes both f and g relative to In and Out. Then $f,g:A^u \longrightarrow A^v$, and to prove the lemma we must show that $f(a)=g(a)$ for each $a \in A^u$.

Choose $a \in A^u$. Then since $\pi_A(In)$ is a surjection (see the previous note), there must be some $\rho \in States(A)$ such that $\pi_A(In)(\rho)=a$. Now, since S computes f relative to In and Out, we have

$$\begin{aligned} f(a) = f(\pi_A(In)(\rho)) &= \pi_A(Out)(M_A(S)(\rho)) \\ &= g(\pi_A(In)(\rho)) \end{aligned}$$

(since S also computes g)

$$= g(a)$$

Thus $f(a)=g(a)$ as claimed. □

We can answer the second question (in the negative) by means of the following counterexample:

Let Σ be the signature of Peano Arithmetic $\mathbb{N}$, and let $S \in PIT(\Sigma)$ be the program $S = y := x+z$. Now let $In=(x)$ and $Out=(y)$. We can now show that there is no function f of the form $f:\mathbb{N} \longrightarrow \mathbb{N}$ such that S computes f relative to In and Out.

First, it is easy to show that

$$(\forall \rho \in States(\mathbb{N}))\ (\pi_{\mathbb{N}}(Out)(M_{\mathbb{N}}(S)(\rho)) = \rho(x)+\rho(z)) \qquad (1)$$

Now, for a contradiction, assume that S *does* compute some f of the above form. Then for any $\rho \in States(\mathbb{N})$ we must have

$$f(\pi_{\mathbb{N}}(In)(\rho)) = f(\rho(x)) = \pi_{\mathbb{N}}(Out)(M_{\mathbb{N}}(S)(\rho)) = \rho(x)+\rho(z) \qquad (2)$$

from the definition of a functional i/o-triple and from (1). However, now let ρ and ρ' be any states with $\rho(x)=\rho'(x)$ but with $\rho(z) \neq \rho'(z)$. Then from (2) we have

$$\begin{aligned} f(\rho(x)) &= \rho(x)+\rho(z) \\ &\neq \rho'(x)+\rho'(z) \end{aligned}$$

(by hypothesis on ρ and ρ')

$$= \pi_{\mathbb{N}}(Out)(M_{\mathbb{N}}(S)(\rho'))$$

(by (1))

$$= f(\rho'(x))$$

since S computes f, and thus we have proved

$$f(\rho(x)) \neq f(\rho'(x)) \qquad (3)$$

Now, ρ and ρ' agree on x by hypothesis, so we can let $\rho(x)=\rho'(x)=n$ say, for some $n \in \mathbb{N}$. Thus from (3) we have $f(n) \neq f(n)$ which is plainly a contradiction, and so f does not exist: that is, S does not compute any function relative to In and Out. □

Why did the above i/o-triple (S, In, Out) fail to be functional? On closer inspection of Definition 6.2.3 we see that an i/o-triple is functional when the initial values of the input variables *uniquely determine* the final values of the output variables. This property is not satisfied by the by the last i/o-triple since the final value of y depends on the initial value of z, and z is not 'declared' to be an input variable. Of course, (S, In, Out) is functional if we take $In = (x, z)$.

We are interested in making PIT programs that compute functions so that we can compute functions denoted by PR schema and so simulate synchronous algorithms. Since not every i/o-triple computes a function we need conditions on i/o-triples that guarantee functionality. As explained in Tucker and Zucker[1987] in their work on while-programs, the functionality of i/o-triples is ultimately a semantic property and is algorithmically undecidable. However, in his work (also on while-programs) C. A. Jervis discovered *syntactic* conditions on i/o-triples that are sufficient (but not necessary) for functionality. We will now explain Jervis' idea in the context of PIT programs.

Given an i/o-triple (S, In, Out), each variable x occurring in S is of one of two kinds: either the behaviour of S (and hence the value of each output variable) is 'dependent' on the initial value of x or it is not. Intuitively, it those variables on whose initial value the behaviour of S depends that need to be input variables for (S, In, Out) to be functional. For example, consider S_1 and S_2 defined by

$$S_1 = y := z;\ x := 0;\ w := y + x$$

and

$$S_2 = y, x, z := z, 0, y + x$$

Here S_1 is independent of the initial value of x since this value is overwritten by the assignment $x := 0$ before the value of x is used. On the other hand S_2 does depend on x since the initial value of x is used in computing the value of w. More succinctly we can say that in S_1 x is *initialised* before it is used, whereas in S_2 it is not. Following Jervis[1988] we call variables that are not initialised *free*. In the obvious notation then, we have

$$init(S_1)=\{y, x, w\},\quad free(S_1)=\{z\},\quad init(S_2)=\{w\},\quad \text{and } free(S_2)=\{x, y, z\}$$

The key observation made by Jervis is that for a general while-program S, $init(S)$ can be defined by induction on the syntactic structure of S. After a preliminary set of definitions we will define $init(S)$ and $free(S)$ for PIT programs S.

6.2.5 Definitions.

(i) For each $w \in S^+$, and for each $X \in Var_w$, we define $var(X)$ to be the *set* (rather than vector) of all the variables occurring in X. Thus, if $X = (x_1, \ldots, x_n)$, then $var(X) = \{x_1, \ldots, x_n\}$.

(ii) For each $e \in EXP(\Sigma)$, we (ambiguously) define $var(e)$ to be the set of all variables occurring in the expression e; $var(e)$ can be defined by induction on the structural complexity of e in the

following way:

(a) $var(c) = \emptyset$.

(b) $var(x) = \{x\}$.

(c) $var(\sigma(e_1,...,e_n)) = var(e_1) \cup \cdots \cup var(e_n)$

(iii) Let $S \in PIT(\Sigma)$. We (ambiguously) define $var(S)$ to be all the variables occurring in S; $var(S)$ can be defined by induction on the structure of S as follows:

(a) $var(x_1,...,x_n := e_1,...,e_n) = \{x_1,...,x_n\} \cup var(e_1) \cup \cdots \cup var(e_n)$

(b) $var(S_1 ; S_2) = var(S_1) \cup var(S_2)$

(c) $var(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = var(b) \cup var(S_1) \cup var(S_2)$

(d) $var(\textbf{do } e \textbf{ times } S_0 \textbf{ od}) = var(e) \cup var(S_0)$

6.2.6 Definitions.

Let $S \in PIT(\Sigma)$. Then we define

(i) $init(S)$, the set of *initialised variables* of S, by induction on the structural complexity of S as follows:

(a) $init(x_1,...,x_n := e_1,...,e_n) = \{x_1,...,x_n\} - (var(e_1) \cup \cdots \cup var(e_n))$

(b) $init(S_1 ; S_2) = init(S_1) \cup (init(S_2) - var(S_1))$

(c) $init(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = (init(S_1) \cap init(S_2)) - var(b))$

(d) $init(\textbf{do } e \textbf{ times } S_0 \textbf{ od}) = \emptyset$

(ii) $free(S)$, the set of *free variables* of S, by $free(S) = var(S) - init(S)$.

6.2.7 Definition.

Let (S,In,Out) be an i/o-triple of arity (u,v) for some $u,v \in S^+$. We say (S,In,Out) is an input/output *program* of arity (u,v) if S, In, and Out, satisfy the following two *J-conditions*:

(i) $var(In) \supseteq free(S)$

(ii) $var(Out) \subseteq var(S) \cup var(In)$

If u and v are understood, then we refer to (S,In,Out) as an input/output program, or, more briefly, as an i/o-program. Additionally, for each $u,v \in S^+$ the collection of all i/o-programs of arity (u,v) is denoted $PITIO(\Sigma)_{u,v}$, and we define the $S^+ \times S^+$-indexed family $PITIO(\Sigma)$ by:

$$PITIO(\Sigma) = \langle PITIO(\Sigma)_{u,v} : u,v \in S^+ \rangle \qquad \square$$

The J-conditions guarantee functionality as is shown by the following theorem and corollary whose proofs we postpone temporarily.

6.2.8 Theorem. *Let* (S,In,Out) *be an i/o-program of arity* (u,v) *for some* $u,v \in S^+$. *Then for every* $\rho,\rho' \in States(A)$ *with* $\pi_A(In)(\rho)=\pi_A(In)(\rho')$,

$$\pi_A(Out)(M_A(S)(\rho)) = \pi_A(Out)(M_A(S)(\rho'))$$

□

6.2.9 Theorem. *Let* (S,In,Out) *be an i/o-program of arity* (u,v) *for some* $u,v \in S^+$. *Then there exists a function* $f : A^u \longrightarrow A^v$ *such that* S *computes* f *relative to* In *and* Out. □

Actually, not only can we prove that and i/o-program computes *some* function, we can also define it *explicitly*. We do this below where we additionally define the complexity of an i/o-program.

6.2.10 Definition.

Let A be a Σ-algebra, and let P be a performance measure for A which is based on clock C. Now let $u,v \in S^+$. Then for each $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$, we define

(i) the *functional meaning of* (S,In,Out) *in* A, $F_A(S,In,Out) : A^u \longrightarrow A^v$ by

$$F_A(S,In,Out)(a) = \pi_A(Out)(M_A(S)(\rho))$$

for each $a \in A^u$, where $\rho \in States(A)$ is any state such that $\pi_A(In)(\rho)=a$.

(ii) the *complexity of* (S,In,Out) *with respect to* P, $\lambda_P^{IO} : A^u \longrightarrow C^+$ by

$$\lambda_P^{IO}(S,In,Out)(a) = \lambda_P^{PIT}(S)(\rho)$$

for each $a \in A^u$, where $\rho \in States(A)$ is any state such that $\pi_A(In)(\rho)=a$. □

The functions $F_A(S,In,Out)$ and $\lambda_P^{IO}(S,In,Out)$ are certainly explicitly defined, but are they *well-defined*? In view of the previous counterexample (concerning the functionality of arbitrary i/o-triples), it is is essential we make sure that the values of $F_A(S,In,Out)(a)$ and $\lambda_P^{IO}(S,In,Out)(a)$ are independent of the choice of ρ such that $\pi_A(In)(\rho)=a$:

6.2.11 Lemma. *Let* (S,In,Out) *be an i/o-program of arity* (u,v) *for some* $u,v \in S^+$. *Then for every* $a \in A^u$, $F_A(S,In,Out)(a)$ *and* $\lambda_P^{IO}(S,In,Out)(a)$ *are well-defined.*

Proof. Let $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$. To see that $F_A(S,In,Out)$ is well-defined, first choose $a \in A^u$ and suppose there are two states $\rho,\rho' \in States(A)$ such that $\pi_A(In)(\rho)=a=\pi_A(In)(\rho')$. (Note that at least one such ρ exists by the surjectivity of $\pi_A(In)$.) Then by definition of $F_A(S,In,Out)$ we have $F_A(S,In,Out)(a)=\pi_A(Out)(M_A(S)(\rho))$ and $F_A(S,In,Out)(a)=\pi_A(Out)(M_A(S)(\rho'))$. Thus to show $F_A(S,In,Out)$ is well-defined, we must show $\pi_A(Out)(M_A(S)(\rho))=\pi_A(Out)(M_A(S)(\rho'))$; however, this is a simple consequence of Theorem 6.2.8.

To show that $\lambda_P^{IO}(S,In,Out)(a)$ is well-defined for each $a \in A^u$, we must show that $\lambda_P^{PIT}(S)(\rho)=\lambda_P^{PIT}(S)(\rho')$ when ρ and ρ' are any two states such that $\pi_A(In)(\rho)=a=\pi_A(In)(\rho')$; however this is immediate from the stronger version of Theorem 6.2.8 which we will discuss below. (See Theorem 6.2.13.) □

It remains to prove Theorems 6.2.8 and 6.2.9. The latter result is easy to prove from the first and so we consider it first:

Proof of Theorem 6.2.9. Let $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$. Then we must show S computes some $f : A^u \longrightarrow A^v$ relative to In and Out. This fact is a direct consequence of the following:

6.2.12 Lemma. *Let (S,In,Out) be an i/o-program of arity (u,v) for some $u,v \in S^+$. Then the following diagram commutes for every $\rho \in States(A)$:*

$$\begin{array}{ccc} States(A) & \xrightarrow{M_A(S)} & States(A) \\ \Big\downarrow{\scriptstyle \pi_A(In)} & & \Big\downarrow{\scriptstyle \pi_A(Out)} \\ A^u & \xrightarrow[F_A(S,In,Out)]{} & A^v \end{array}$$

That is, S computes $F_A(S,In,Out)$ relative to In and Out.

Proof. To prove the lemma, we must show that for every $\rho \in States(A)$ that

$$\pi_A(Out)(M_A(S)(\rho)) = F_A(S,In,Out)(\pi_A(In)(\rho)) \tag{4}$$

Choose $\rho \in States(A)$. Then by definition of $F_A(S,In,Out)$ we have:

$$F_A(S,In,Out)(\pi_A(In)(\rho)) = \pi_A(Out)(M_A(S)(\rho')) \tag{5}$$

where $\rho' \in States(A)$ is any state such that $\pi_A(In)(\rho') = \pi_A(In)(\rho)$. (Of course ρ is such a state, and so at least one such ρ' exists.) However, if $\pi_A(In)(\rho') = \pi_A(In)(\rho)$, then by Theorem 6.2.8 we have:

$$\pi_A(Out)(M_A(S)(\rho')) = \pi_A(Out)(M_A(S)(\rho)) \tag{6}$$

Thus, from (5) we have:

$$\begin{aligned} F_A(S,In,Out)(\pi_A(In)(\rho)) &= \pi_A(Out)(M_A(S)(\rho')) \\ &= \pi_A(Out)(M_A(S)(\rho)) \end{aligned}$$

(using (6)), and thus (4) holds for each $\rho \in States(A)$. □

It remains to prove Theorem 6.2.8. Actually, we will prove the theorem simultaneously with another statement that says that length of computation is only affected by the values of free variables: the theorem we will eventually prove is as follows:

6.2.13 Theorem. *Let (S,In,Out) be an i/o-program of arity (u,v) for some $u,v \in S^+$. Then for every $\rho,\rho' \in States(A)$ with $\pi_A(In)(\rho) = \pi_A(In)(\rho')$,*

$$\pi_A(Out)(M_A(S)(\rho)) = \pi_A(Out)(M_A(S)(\rho'))$$

Furthermore,

$$\lambda_P^{PIT}(S)(\rho) = \lambda_P^{PIT}(S)(\rho')$$ □

Before we can prove this theorem we need to establish a similar result concerning *functional expressions*:

6.2.14 Definition.

We define FEXP(Σ) to be the $S^+ \times S$-indexed family

$$\mathrm{FEXP}(\Sigma) = \langle \mathrm{FEXP}(\Sigma)_{w,s} : w \in S^+, s \in S \rangle$$

of collections $\mathrm{FEXP}(\Sigma)_{w,s}$ of *functional expressions of arity* (w,s). For each $w \in S^+$ and $s \in S$, each $\mathrm{FEXP}(\Sigma)_{w,s} \subseteq \mathrm{EXP}(\Sigma)_s \times Var_w$ is defined by

$$(\forall e \in \mathrm{EXP}(\Sigma)_s)(\forall X \in Var_w)\ \ ((e,X) \in \mathrm{FEXP}(\Sigma)_{w,s} \iff var(X) \supseteq var(e))$$

6.2.15 Lemma. *Let $w \in S^+$ and $s \in S$. Then for every $(e,X) \in \mathrm{FEXP}(\Sigma)_{w,s}$, if $\rho,\rho' \in States(A)$ are such that $\pi_A(X)(\rho) = \pi_A(X)(\rho')$ then*

$$E_A(e)(\rho) = E_A(e)(\rho') \tag{7}$$

Furthermore,

$$\lambda_p^{EXP}(e)(\rho) = \lambda_p^{EXP}(e)(\rho') \tag{8}$$

Proof. Let $(e,X) \in \mathrm{FEXP}(\Sigma)_{w,s}$ for some $w \in S^+$ and $s \in S$. Also let $\rho,\rho' \in States(A)$ be any states such that $\pi_A(X)(\rho) = \pi_A(X)(\rho')$. We will prove the lemma by showing that (7) and (8) hold by induction on the structural complexity of e as follows:

Basis Cases.

(i) *Constants.* Suppose $e = c$ for some $c \in \Sigma_{\lambda,s}$.

Then by definition of $E_A(e)$ we have:

$$\begin{aligned} E_A(e)(\rho) &= E_A(c)(\rho) \\ &= c^A \\ &= E_A(c)(\rho') \\ &= E_A(e)(\rho') \end{aligned}$$

Thus (7) holds for $e = c$ (independent of the hypotheses on ρ and ρ'.)

To see that (8) holds for $e = c$, we calculate as follows:

$$\begin{aligned} \lambda_p^{EXP}(e)(\rho) &= \lambda_p^{EXP}(c)(\rho) \\ &= c^P \\ &= \lambda_p^{EXP}(c)(\rho') \\ &= \lambda_p^{EXP}(e)(\rho') \end{aligned}$$

Thus (8) holds for $e = c$ (again independently of the hypotheses on ρ and ρ'.)

(ii) *Variables.* Suppose $e = x$ for some $x \in Var_s$.

First notice that since $(e,X) \in \mathrm{FEXP}(\Sigma)$, we have $var(X) \supseteq var(e) = \{x\}$. Thus $x \in var(X)$, and so $x = X_i$ for some $i \in [1,n]$ when $n = |w|$. Now, $\pi_A(X)(\rho) = \pi_A(X)(\rho')$ by hypothesis, and so $\rho(X_j) = \rho'(X_j)$ for $j = 1,\ldots,n$. In particular, we have

$$\rho(x) = \rho(X_i) = \rho'(X_i) = \rho'(x) \tag{9}$$

To see that (7) holds for $e = x$, we calculate as follows:

$$E_A(e)(\rho) = E_A(x)(\rho)$$
$$= \rho(x)$$
$$= \rho'(x)$$

(using (9))

$$= E_A(x)(\rho')$$
$$= E_A(e)(\rho')$$

as required.

To see that (8) holds for $e = x$, we calculate as follows:

$$\lambda_P^{EXP}(e)(\rho) = \lambda_P^{EXP}(x)(\rho)$$
$$= 1$$
$$= \lambda_P^{EXP}(x)(\rho')$$
$$= \lambda_P^{EXP}(e)(\rho')$$

Again, this is as required.

Induction. Suppose that $(e,X) \in \mathrm{FEXP}(\Sigma)_{w,s}$ is some fixed functional expression with the property that for each $s' \in S$ and for every $(e',X) \in \mathrm{FEXP}(\Sigma)_{w,s'}$ with e' of less structural complexity than e, that whenever $\rho,\rho' \in States(A)$ are any states that satisfy $\pi_A(X)(\rho) = \pi_A(X)(\rho')$, then

$$E_A(e')(\rho) = E_A(e')(\rho') \tag{10}$$

and

$$\lambda_P^{EXP}(e')(\rho) = \lambda_P^{EXP}(e')(\rho') \tag{11}$$

Let $\rho,\rho' \in States(A)$ satisfy $\pi_A(X)(\rho) = \pi_A(X)(\rho')$. We now show that (7) and (8) hold for (e,X).

(iii) *Operations.* Without loss of generality, we may suppose e is of the form $e = \sigma(e_1,\ldots,e_n)$, where for some $u \in S^+$, $\sigma \in \Sigma_{u,s}$, and $e_i \in \mathrm{EXP}(\Sigma)_{u_i}$ for $i = 1,\ldots,n = |u|$.

First notice that by the definitions of $\mathrm{FEXP}(\Sigma)$ and $var(e)$ we have

$$var(X) \supseteq var(e) = var(e_1) \cup \cdots \cup var(e_n)$$

and so $(e_i,X) \in \mathrm{FEXP}(\Sigma)_{w,u_i}$ for $i = 1,\ldots,n$.

Also for $i = 1,\ldots,n$, since each e_i is of less structural complexity than e, by the induction hypotheses (10) and (11) applied to (e_i,X) we have

$$E_A(e_i)(\rho) = E_A(e_i)(\rho') \tag{12}$$

and

$$\lambda_P^{EXP}(e_i)(\rho)) = \lambda_P^{EXP}(e_i)(\rho') \tag{13}$$

for $i = 1,\ldots,n$.

We can now show (7) holds for $e = \sigma(e_1,\ldots,e_n)$ as follows:

$$E_A(e)(\rho) = E_A(\sigma(e_1,\ldots,e_n))(\rho)$$
$$= \sigma^A(E_A(e_1)(\rho),\ldots,E_A(e_n)(\rho))$$

$$= \sigma^A(E_A(e_1)(\rho'),\ldots,E_A(e_n)(\rho'))$$

(using (12) with $i = 1,\ldots,n$)

$$= E_A(\sigma(e_1,\ldots,e_n))(\rho')$$
$$= E_A(e)(\rho)$$

Thus (7) holds for $e = \sigma(e_1,\ldots,e_n)$.

We now show (8) holds for $e = \sigma(e_1,\ldots,e_n)$ as follows:

$$\lambda_P^{EXP}(e)(\rho) = \lambda_P^{EXP}(\sigma(e_1,\ldots,e_n))(\rho)$$
$$= \sigma^P(E_A(e_1)(\rho),\ldots,E_A(e_n)(\rho))$$
$$+ \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_n)(\rho)\}$$

$$= \sigma^P(E_A(e_1)(\rho'),\ldots,E_A(e_n)(\rho'))$$
$$+ \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_n)(\rho)\}$$

(by (12) with $i = 1,\ldots,n$)

$$= \sigma^P(E_A(e_1)(\rho'),\ldots,E_A(e_n)(\rho'))$$
$$+ \max\{\lambda_P^{EXP}(e_1)(\rho'),\ldots,\lambda_P^{EXP}(e_n)(\rho')\}$$

(by (13) with $i = 1,\ldots,n$)

$$= \lambda_P^{EXP}(\sigma(e_1,\ldots,e_n))(\rho')$$

$$= \lambda_P^{EXP}(e)(\rho')$$

Thus (8) also holds for $e = \sigma(e_1,\ldots,e_n)$. □

Before we can prove Theorem 6.2.13 we need the following facts concerning *free* (). Each can be proved from the definition of *free* (S) using set-theoretic arguments; we leave the proofs as exercises.

6.2.16 Lemma. *Suppose* $S \in \text{PIT}(\Sigma)$ *is of the form* $S = S_1;S_2$ *for some* $S_1,S_2 \in \text{PIT}(\Sigma)$. *Then*

(a) $free(S_1;S_2) = free(S_1) \cup (free(S_2) - var(S_1))$

(b) $free(S_1;S_2) \supseteq free(S_1)$ □

6.2.17 Lemma. *Suppose* S *is of the form* $S =$ **if** b **then** S_1 **else** S_2 **fi** *for some* $b \in \text{EXP}(\Sigma)_B$, *and some* $S_1,S_2 \in \text{PIT}(\Sigma)$. *Then for* $i = 1,2$,

(a) $free(S) \supseteq free(S_i)$

(b) $free(S) \supseteq init(S_i) - var(S_{3-i})$ □

Proof of Theorem 6.2.13. We are now in a position to prove Theorem 6.2.13 which guarantees the functionality of i/o-programs.

For convenience we restate what we must prove:

Theorem. *Let* (S,In,Out) *be an i/o-program of arity* (u,v) *for some* $u,v \in S^+$. *Then for every* $\rho,\rho' \in States(A)$ *with* $\pi_A(In)(\rho)=\pi_A(In)(\rho')$,

$$\pi_A(Out)(M_A(S)(\rho)) = \pi_A(Out)(M_A(S)(\rho')) \tag{14}$$

Furthermore,

$$\lambda_P^{PIT}(S)(\rho) = \lambda_P^{PIT}(S)(\rho') \tag{15}$$

Proof. Let $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$, and let ρ and ρ' be such that $\pi_A(In)(\rho)=\pi_A(In)(\rho')$. Also let $|u|=n$ and $|v|=m$. We prove (14) and (15) simultaneously and uniformly in u and v, and by induction on the structural complexity of S.

Basis Case.

(i) *Multiple Assignment Statements.* Suppose S is of the form:

$$S = x_1,\ldots,x_r := e_1,\ldots,e_r$$

for some $r>0$.

First notice that since (S,In,Out) is an i/o-program with S a multiple assignment statement, we have

$$var(In) \supseteq free(S) = var(S)\text{–}init(S) = var(S)\text{–}(\{x_1,\ldots,x_r\}\text{–}(var(e_1)\cup\cdots\cup var(e_r)))$$

and so $var(In) \supseteq var(e_1)\cup\cdots\cup var(e_r)$ (since the variables of $e_1,\ldots,e_r$ are also variables of S). Hence, $var(In)\supseteq var(e_i)$ for $i=1,\ldots,r$, and so $(e_i,In)\in \mathrm{FEXP}(\Sigma)_{u,w_i}$ for $i=1,\ldots,r$. Thus by Lemma 6.2.15 we have

$$E_A(e_i)(\rho) = E_A(e_i)(\rho') \tag{16}$$

and

$$\lambda_P^{EXP}(e_i)(\rho) = \lambda_P^{EXP}(e_i)(\rho') \tag{17}$$

for $i=1,\ldots,r$.

Let $M_A(S)(\rho)=\rho_f$ ('f' for final state) and $M_A(S)(\rho')=\rho_f'$. Then to show (14) holds for S when S is a multiple assignment statement we must show

$$\rho_f(Out_i) = \rho_f'(Out_i) \tag{18}$$

for $i=1,\ldots,m$.

To show that (18) holds for $i=1,\ldots,m$, we first claim that for $i=1,\ldots,m$, if $Out_i \neq x_j$ for some $j\in[1,r]$, then $Out_i = In_k$ for some $k\in[1,n]$. To see why this is true, notice that

$$var(Out) \subseteq var(S)\cup var(In)$$

since (S,In,Out) is an i/o-program. Thus for $i=1,\ldots,m$,

$$Out_i \in var(Out) \subseteq var(S)\cup var(In) = \{x_1,\ldots,x_r\}\cup var(e_1)\cup\cdots\cup var(e_r)\cup var(In)$$

Now, if $Out_i \notin \{x_1,\ldots,x_r\}$ then

$$Out_i \in var(e_1) \cup \cdots \cup var(e_r) \cup var(In)$$

But $(S, In, Out) \in \text{PITIO}(\Sigma)$ and S is a multiple assignment statement, thus:

$$var(In) \supseteq free(S) = var(e_1) \cup \cdots \cup var(e_r)$$

and thus

$$Out_i \in var(e_1) \cup \cdots \cup var(e_r) \cup var(In) \subseteq var(In)$$

Thus $Out_i \in var(In)$, and so $Out_i = In_k$ for some $k \in [1,n]$ as claimed.

Now choose $i \in [1,m]$. We show (18) holds in each of the two following cases:
Case 1: $Out_i = x_j$ for some $j \in [1,r]$
Case 2: $Out_i \neq x_j$ for any $j \in [1,r]$

Case 1. First suppose $Out_i = x_j$ for some $j \in [1,r]$. Then

$$\rho_f(Out_i) = \rho_f(x_j)$$
$$= M_A(S)(\rho)(x_j)$$
$$= E_A(e_j)(\rho)$$

(since S is a multiple assignment statement)

$$= E_A(e_j)(\rho')$$

(by (16))

$$= M_A(S)(\rho')(x_j)$$

(since S is a multiple assignment statement)

$$= \rho'_f(x_j)$$
$$= \rho'_f(Out_i)$$

Thus (18) holds in this case.

Case 2. Now suppose $Out_i \neq x_j$ for any $j \in [1,r]$. Then as we have established above, it must be that $Out_i = In_k$ for some $k \in [1,n]$.

We can show (18) holds in this case as follows:

$$\rho_f(Out_i) = M_A(S)(\rho)(Out_i)$$
$$= \rho(Out_i)$$

(since S is a multiple assignment statement and Out_i does not occur on the left hand side of S)

$$= \rho(In_k)$$
$$= \rho'(In_k)$$

(by hypothesis on ρ and ρ')

$$= \rho'(Out_i)$$
$$= \rho'_f(Out_i)$$

(again since S is a multiple assignment statement and Out_i does not occur on the left hand side of S).

Thus (18) holds for $i = 1,\ldots,m$, and so (14) holds for S as claimed.

To show that (15) holds for multiple assignments S we calculate as follows:

$$\lambda_P^{PIT}(S)(\rho) = \max_{1\le i\le n}\{\lambda_P^{EXP}(e_i)(\rho)\}$$
$$= \max_{1\le i\le n}\{\lambda_P^{EXP}(e_i)(\rho')\}$$

(using (17) with $i = 1,\ldots,n$)

$$= \lambda_P^{PIT}(S)(\rho')$$

Induction. Let $S \in \mathrm{PIT}(\Sigma)$ be some fixed program. Suppose for any $u',v' \in S^+$ and for every $(S',In',Out') \in \mathrm{PITIO}(\Sigma)_{u',v'}$ where S' is of less structural complexity than S, that for any $\rho,\rho' \in States(A)$ which satisfy $\pi_A(In')(\rho) = \pi_A(In')(\rho')$ we have

$$\pi_A(Out')(M_A(S')(\rho)) = \pi_A(Out')(M_A(S')(\rho'))$$

and

$$\lambda_P^{PIT}(S')(\rho) = \lambda_P^{PIT}(S')(\rho')$$

We will now show (14) and (15) hold for S and for any $In \in Var_u$ and $Out \in Var_v$ according to the three following possible cases:

(ii) *Sequencing*. Suppose $S = S_1;S_2$ for some $S_1,S_2 \in \mathrm{PIT}(\Sigma)$.

Let $In_1 = In$, and let Out_1 be any vector of distinct variables such that $var(Out_1) = var(S_1) \cup var(In_1)$. Also let $In_2 = Out_1$ and $Out_2 = Out$.

We first show $(S_i,In_i,Out_i) \in \mathrm{PITIO}(\Sigma)$ for $i = 1,2$; that is, we must show (S_i,In_i,Out_i) satisfies the J-conditions for $i = 1,2$. (see Definition 6.2.7.)

First notice that

$$var(In_1) = var(In) \supseteq free(S) \supseteq free(S_1)$$

by Lemma 6.2.16(b).

Also,

$$var(Out_1) = var(S_1) \cup var(In_1) \subseteq var(S_1) \cup var(In_1)$$

Thus $(S_1,In_1,Out_1) \in \mathrm{PITIO}(\Sigma)_{u,w}$ where $w \in S^+$ is such that $Out_1 \in Var_w$.

We can establish $(S_2,In_2,Out_2) \in \mathrm{PITIO}(\Sigma)$ as follows:

$$var(In_2) = var(Out_1)$$
$$= var(S_1) \cup var(In_1)$$
$$= var(S_1) \cup var(In)$$
$$\supseteq var(S_1) \cup free(S)$$
$$= var(S_1) \cup (free(S_1) \cup (free(S_2) - var(S_1)))$$

(by Lemma 6.2.16(a))

$$\supseteq free(S_2)$$

(since $X \cup (Y \cup (Z - X)) \supseteq Z$ for any sets X, Y, and Z).

Also, we have:

$$var(Out_2) = var(Out)$$
$$\subseteq var(S) \cup var(In)$$
$$= var(S_1) \cup var(S_2) \cup var(In)$$
$$= var(S_2) \cup var(S_1) \cup var(In)$$
$$= var(S_2) \cup var(S_1) \cup var(In_1)$$
$$= var(S_2) \cup var(Out_1)$$

(by definition of Out_1)

$$= var(S_2) \cup var(In_2)$$

(by definition of In_2).

Thus $(S_2, In_2, Out_2) \in \text{PITIO}(\Sigma)_{w,v}$.

Since $(S_1, In_1, Out_1) \in \text{PITIO}(\Sigma)$ with S_1 of less structural complexity than S by the induction hypotheses applied to (S_1, In_1, Out_1), we have

$$\pi_A(Out_1)(M_A(S_1)(\rho_1)) = \pi_A(Out_1)(M_A(S_1)(\rho_2)) \tag{19}$$

and

$$\lambda_P^{PIT}(S_1)(\rho_1) = \lambda_P^{PIT}(S_1)(\rho_2) \tag{20}$$

where ρ_1 and ρ_2 are any states such that $\pi_A(In_1)(\rho_1) = \pi_A(In_1)(\rho_2)$.

Also, since $(S_2, In_2, Out_2) \in \text{PITIO}(\Sigma)$ with S_2 of less structural complexity than S by the induction hypotheses applied to (S_2, In_2, Out_2), we have

$$\pi_A(Out_2)(M_A(S_2)(\rho_3)) = \pi_A(Out_2)(M_A(S_2)(\rho_4)) \tag{21}$$

and

$$\lambda_P^{PIT}(S_2)(\rho_3) = \lambda_P^{PIT}(S_2)(\rho_4) \tag{22}$$

where ρ_3 and ρ_4 are any states such that $\pi_A(In_2)(\rho_3) = \pi_A(In_2)(\rho_4)$.

To show that (14) holds for $S = S_1; S_2$, first notice that

$$\pi_A(In_1)(\rho) = \pi_A(In)(\rho) = \pi_A(In)(\rho') = \pi_A(In_1)(\rho')$$

by hypothesis on ρ and ρ', and since $In_1 = In$.

Since $In_2 = Out_1$ we now have

$$\pi_A(In_2)(M_A(S_1)(\rho)) = \pi_A(Out_1)(M_A(S_1)(\rho))$$
$$= \pi_A(Out_1)(M_A(S_1)(\rho'))$$

(by (19) with $\rho_1 = \rho$ and $\rho_2 = \rho'$)

$$= \pi_A(In_2)(M_A(S_1)(\rho'))$$

(again since $In_2 = Out_1$).

We have now proved that

$$\pi_A(In_2)(M_A(S_1)(\rho)) = \pi_A(In_2)(M_A(S_1)(\rho')) \tag{23}$$

Now, (21) holds for any states ρ_3 and ρ_4 such that $\pi_A(In_2)(\rho_3) = \pi_A(In_2)(\rho_4)$. In particular, from (23) we know that $\rho_3 = M_A(S_1)(\rho)$ and $\rho_4 = M_A(S_1)(\rho')$ are two such states; thus by (21) and (23) we have

$$\pi_A(Out_2)(M_A(S_2)(M_A(S_1)(\rho))) = \pi_A(Out_2)(M_A(S_2)(M_A(S_1)(\rho')))$$

That is,

$$\pi_A(Out_2)(M_A(S)(\rho))) = \pi_A(Out_2)(M_A(S)(\rho'))) \tag{24}$$

by definition of $M_A(S)$. However $Out_2 = Out$, and so (14) holds for $S = S_1;S_2$ (by (24)).

To show that (15) holds for $S = S_1;S_2$ first notice that (22) holds for any states ρ_3 and ρ_4 such that $\pi_A(In_2)(\rho_3) = \pi_A(In_2)(\rho_4)$. In particular, from (23) we know that $\rho_3 = M_A(S_1)(\rho)$ and $\rho_4 = M_A(S_1)(\rho')$ are two such states; thus by (22) and (23) we have

$$\lambda_P^{PR}(S_2)(M_A(S_1)(\rho)) = \lambda_P^{PR}(S_2)(M_A(S_1)(\rho')) \tag{25}$$

To show that (15) holds for $S = S_1;S_2$ we calculate as follows:

$$\begin{aligned}\lambda_P^{PIT}(S)(\rho) &= \lambda_P^{PIT}(S_1)(\rho) + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho)) \\ &= \lambda_P^{PIT}(S_1)(\rho') + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho))\end{aligned}$$

(by (20) with $\rho_1 = \rho$ and $\rho_2 = \rho'$)

$$= \lambda_P^{PIT}(S_1)(\rho') + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho'))$$

(by (25))

$$= \lambda_P^{PIT}(S)(\rho')$$

Thus (15) holds for $S = S_1;S_2$ as claimed.

(iii) *Conditional.* Suppose S is of the form:

$$S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

for some $b \in \mathrm{EXP}(\Sigma)_B$, and some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$.

First notice that

$$var(In) \supseteq free(S) = var(S) - init(S) = var(S) - ((init(S_1) \cap init(S_2)) - var(b)) \supseteq var(b)$$

(since the variables of b are also variables of S). Thus $(b, In) \in \mathrm{FEXP}(\Sigma)_{u,B}$ and so by Lemma 6.2.15 we have

$$E_A(b)(\rho) = E_A(b)(\rho') \tag{26}$$

and

$$\lambda_P^{EXP}(b)(\rho) = \lambda_P^{EXP}(b)(\rho') \tag{27}$$

We first show $(S_i, In, Out) \in \mathrm{PITIO}(\Sigma)$ for $i = 1,2$; that is, we show that (S_i, In, Out) satisfies the J-conditions for $i = 1,2$.

We first show $var(In) \supseteq free(S_i)$ for $i = 1,2$ as follows:

$$\begin{aligned}var(In) &\supseteq free(S) \\ &= var(S) - init(S) \\ &= var(S) - ((init(S_1) \cap init(S_2)) - var(b)) \\ &\supseteq var(S) - (init(S_1) \cap init(S_2)) \\ &\supseteq var(S) - (init(S_1) \cup init(S_2)) \\ &\supseteq (var(S_1) \cup var(S_2)) - (init(S_1) \cup init(S_2)) \\ &\supseteq (var(S_1) - init(S_1)) \cup (var(S_2) \cup init(S_2))\end{aligned}$$

(since $init(S_i) \subseteq var(S_i)$ for $i = 1,2$)

$$= free(S_1) \cup free(S_2)$$
$$\supseteq free(S_i)$$

(for $i = 1$ and $i = 2$).

It remains to show $var(Out) \subseteq var(S_i) \cup var(In)$ for $i = 1,2$. For $i = 1$ we proceed as follows:

$$var(Out) \subseteq var(S) \cup var(In)$$
$$= var(S_1) \cup var(S_2) \cup var(In)$$
$$= var(S_1) \cup var(In) \cup var(S_2)$$
$$= var(S_1) \cup var(In) \cup free(S_2) \cup init(S_2)$$
$$\subseteq var(S_1) \cup var(In) \cup free(S) \cup init(S_2)$$

(by Lemma 6.2.17(a) with $i = 2$)

$$\subseteq var(S_1) \cup var(In) \cup init(S_2)$$

(since $free(S) \subseteq var(In)$)

$$\subseteq var(S_1) \cup var(In) \cup (init(S_2) \cap var(S_1)) \cup (init(S_2) - var(S_1))$$

(since $X = (X \cap Y) \cup (X - Y)$ for any sets X and Y)

$$\subseteq var(S_1) \cup var(In) \cup (init(S_2) - var(S_1))$$

(since $X \cap Y \subseteq Y$ for any sets X and Y)

$$\subseteq var(S_1) \cup var(In) \cup free(S)$$

(by Lemma 6.2.17(b) with $i = 2$)

$$= var(S_1) \cup var(In)$$

The truth of $var(Out) \subseteq var(S_2) \cup var(In)$ can be proved in much the same way; this we leave as an exercise.

Since $(S_i, In, Out) \in \text{PITIO}(\Sigma)$ for $i = 1,2$ with S_i of less structural complexity than S for $i = 1,2$, by the induction hypothesis applied to (S_i, In, Out), we have

$$\pi_A(Out_i)(M_A(S_i)(\rho)) = \pi_A(Out_i)(M_A(S_i)(\rho')) \tag{28}$$

and

$$\lambda_P^{PIT}(S_i)(\rho) = \lambda_P^{PIT}(S_i)(\rho') \tag{29}$$

for $i = 1,2$.

Now to see that (14) holds for conditional statements S, from the definition of $M_A(S)$ we have

$$\pi_A(Out)(M_A(S)(\rho)) = \begin{cases} \pi_A(Out)(M_A(S_1)(\rho)) & \text{if } E_A(b)(\rho) = tt \\ \pi_A(Out)(M_A(S_2)(\rho)) & \text{if } E_A(b)(\rho) = ff \end{cases}$$
$$= \begin{cases} \pi_A(Out)(M_A(S_1)(\rho')) & \text{if } E_A(b)(\rho') = tt \\ \pi_A(Out)(M_A(S_2)(\rho')) & \text{if } E_A(b)(\rho') = ff \end{cases}$$

(using (26) and (28) with $i = 1,2$)

$$= \pi_A(Out)(M_A(S)(\rho'))$$

To show that (15) holds for S, we use the definition of $\lambda_P^{PIT}(S)$ to calculate as follows:

$$\lambda_P^{PIT}(S)(\rho) = \lambda_P^{EXP}(b)(\rho) + \begin{cases} \lambda_P^{PIT}(S_1)(\rho) & \text{if } E_A(b)(\rho)=tt \\ \lambda_P^{PIT}(S_2)(\rho) & \text{if } E_A(b)(\rho)=ff \end{cases}$$

$$= \lambda_P^{EXP}(b)(\rho) + \begin{cases} \lambda_P^{PIT}(S_1)(\rho') & \text{if } E_A(b)(\rho)=tt \\ \lambda_P^{PIT}(S_2)(\rho') & \text{if } E_A(b)(\rho)=ff \end{cases}$$

(using (29) with $i=1,2$)

$$= \lambda_P^{EXP}(b)(\rho') + \begin{cases} \lambda_P^{PIT}(S_1)(\rho') & \text{if } E_A(b)(\rho')=tt \\ \lambda_P^{PIT}(S_2)(\rho') & \text{if } E_A(b)(\rho')=ff \end{cases}$$

(using (26) and (27))

$$= \lambda_P^{PIT}(S)(\rho')$$

Thus (8) holds for conditional statements S as claimed.

(iv) *Bounded Iteration.* Suppose S is of the form:

$$S = \textbf{do } e \textbf{ times } S_0 \textbf{ od}$$

for some $e \in \mathrm{EXP}(\Sigma)_\tau$ and some $S_0 \in \mathrm{PIT}(\Sigma)$.

First notice that since $init(S)=\emptyset$ here, we have

$$var(In) \supseteq free(S) = var(S) - init(S) = var(S) \supseteq var(e)$$

Thus $(e, In) \in \mathrm{FEXP}(\Sigma)_{\mu,\tau}$ and so by Lemma 6.2.15 we have

$$E_A(e)(\rho) = E_A(e)(\rho') \tag{30}$$

and

$$\lambda_P^{EXP}(e)(\rho) = \lambda_P^{EXP}(e)(\rho') \tag{31}$$

Also, it is easy to see that

$$var(In) \supseteq free(S) = var(S) \supseteq var(S_0) \supseteq free(S_0)$$

(since $init(S)=\emptyset$) and so $(S_0, In, In) \in \mathrm{PITIO}(\Sigma)_{\mu,\mu}$.

Since $(S_0, In, In) \in \mathrm{PITIO}(\Sigma)$ with S_0 of less structural complexity than S, by the induction hypothesis applied to (S_0, In, In), we have

$$\pi_A(In)(M_A(S_0)(\rho)) = \pi_A(In)(M_A(S_0)(\rho')) \tag{32}$$

and

$$\lambda_P^{PIT}(S_0)(\rho) = \lambda_P^{PIT}(S_0)(\rho') \tag{33}$$

Also notice that since (S, In, Out) is an i/o-program (and since $init(S)=\emptyset$) we have

$$var(Out) \subseteq var(S) \cup var(In) = free(S) \cup var(In) \subseteq var(In) \tag{34}$$

To show that (14) holds for loops S we first make the following claim which we will prove later.

Claim. For each $k \geq 0$ let ρ_k and ρ'_k be defined by:

$$\rho_0 = \rho$$
$$\rho_{k+1} = M_A(S_0)(\rho_k)$$

and

$$\rho'_0 = \rho'$$
$$\rho'_{k+1} = M_A(S_0)(\rho'_k)$$

where $\rho,\rho' \in States(A)$ are any states such that $\pi_A(In)(\rho) = \pi_A(In)(\rho')$. Then we claim

$$\pi_A(In)(\rho_k) = \pi_A(In)(\rho'_k) \tag{35}$$

for each $k \geq 0$.

Now, if $E_A(e)(\rho) = l$ say, then $E_A(e)(\rho') = l$ by (30). Moreover, from the definition of $M_A(S)$, it is obvious that $M_A(S)(\rho) = \rho_l$ and $M_A(S)(\rho') = \rho'_l$. Thus to show (14) holds for S we must show that

$$\rho_l(Out_i) = \rho'_l(Out_i) \tag{36}$$

for $i = 1,\ldots,m$.

Choose $i \in [1,m]$. Then from (34) $var(Out) \subseteq var(In)$, and so $Out_i = In_k$ for some $k \in [1,n]$. Thus,

$$\begin{aligned}\rho_l(Out_i) &= \rho_l(In_k)\\ &= \rho'_l(In_k)\end{aligned}$$

(by (35))

$$= \rho'_l(Out_i)$$

Thus (14) holds for loops S as claimed.

Proof of Claim. We prove (35) by sub-induction on k:

Sub-Basis. For $k = 0$ we calculate as follows:

$$\begin{aligned}\pi_A(In)(\rho_0) &= \pi_A(In)(\rho)\\ &= \pi_A(In)(\rho')\\ &= \pi_A(In)(\rho'_0)\end{aligned}$$

by hypothesis on ρ and ρ'.

Sub-Induction. Suppose that (35) holds for some fixed $k \geq 0$. To show that it holds for $k+1$ we first calculate as follows:

By the sub-induction hypothesis we have

$$\pi_A(In)(\rho_k) = \pi_A(In)(\rho'_k)$$

and so by (32) (which is true for *any* states ρ and ρ' such that $\pi_A(In)(\rho) = \pi_A(In)(\rho')$), we have

$$\pi_A(In)(M_A(S_0)(\rho_k)) = \pi_A(In)(M_A(S_0)(\rho'_k)) \tag{37}$$

Now, by definition of ρ_{k+1} we have

$$\begin{aligned}\pi_A(In)(\rho_{k+1}) &= \pi_A(In)(M_A(S_0)(\rho_k))\\ &= \pi_A(In)(M_A(S_0)(\rho'_k))\end{aligned}$$

(using (37))

$$= \pi_A(In)(\rho'_{k+1})$$

(by definition of ρ'_{k+1}).

By the principle of Mathematical Induction, the claim holds for every $k \geq 0$. □

We now show (15) holds for loops S.

First consider $\lambda_P^{PIT}(S)(\rho)$ for *any* $\rho \in States(A)$. Since S is a loop, we have from Definition 6.1.7 that $\lambda_P^{PIT}(S)(\rho)$ is defined by

$$\lambda_P^{PIT}(S)(\rho) = \lambda_P^{EXP}(e)(\rho) + \lambda_l \tag{38}$$

where $l = E_A(e)(\rho)$,

$$\lambda_0 = 0,$$

and for any $k \geq 0$,

$$\lambda_{k+1} = \lambda_k + \lambda_P^{PIT}(S_0)(\rho_k)$$

where for each $k \geq 0$, ρ_k is defined by:

$$\rho_0 = \rho,$$

and for any $k \geq 0$,

$$\rho_{k+1} = M_A(S_0)(\rho_k)$$

In order to prove (15) holds for loops S, let us rephrase $\lambda_P^{PIT}(S)$ is the following way. First define $\kappa: \mathbf{N} \times States(A) \longrightarrow C^+$ by:

$$\kappa(0,\rho) = \lambda_P^{EXP}(e)(\rho)$$

and

$$\kappa(k+1,\rho) = \kappa(k,\rho) + \lambda_P^{PIT}(S_0)(\rho_k)$$

for each $k \geq 0$ and any $\rho \in States(A)$.

A routine induction on k yields:

$$(\forall k \geq 0)(\forall \rho \in States(A))\ \ (\ \kappa(k,\rho) = \lambda_P^{EXP}(e)(\rho) + \lambda_k\ )$$

and thus from (38) we have:

$$(\forall \rho \in States(A))\ \ (\ \lambda_P^{PIT}(S)(\rho) = \kappa(E_A(e)(\rho),\rho)\ ) \tag{39}$$

We now make the following claim which we will prove later:

Claim. For every $k \geq 0$,

$$\kappa(k,\rho) = \kappa(k,\rho') \tag{40}$$

where $\rho,\rho' \in States(A)$ are any states such that $\pi_A(In)(\rho) = \pi_A(In)(\rho')$.

Assuming the claim holds, we can now show (15) holds for loops S as follows:

$$\lambda_P^{PIT}(S)(\rho) = \kappa(E_A(e)(\rho), \rho)$$

(by (39))

$$= \kappa(E_A(e)(\rho'),\rho')$$

(by the claim, and by (30))

$$= \lambda_P^{PIT}(S)(\rho')$$

(by (39)).

Thus (15) holds for loops S as claimed. □

Proof of Claim. We prove (35) by sub-induction on k:

Sub-Basis. For $k=0$ we calculate as follows:

$$\kappa(0,\rho) = \lambda_P^{EXP}(e)(\rho)$$
$$= \lambda_P^{EXP}(e)(\rho')$$

(by (31))

$$= \kappa(0,\rho')$$

Sub-Induction. Suppose that (40) holds for some fixed $k \geq 0$. To show that it holds for $k+1$ first notice that from the previous claim we have

$$\pi_A(In)(\rho_k) = \pi_A(In)(\rho_k')$$

and thus from (33) we have

$$\lambda_P^{PIT}(S_o)(\rho_k) = \lambda_P^{PIT}(S_o)(\rho_k') \qquad (41)$$

Now we can show that (40) holds for $k+1$:

$$\kappa(k+1,\rho) = \kappa(k,\rho) + \lambda_P^{PIT}(S_o)(\rho_k)$$
$$= \kappa(k,\rho) + \lambda_P^{PIT}(S_o)(\rho_k')$$

(by (41))

$$= \kappa(k,\rho') + \lambda_P^{PIT}(S_o)(\rho_k')$$

(by the sub-induction hypothesis)

$$= \kappa(k+1,\rho')$$

By the principle of Mathematical Induction, the claim holds for every $k \geq 0$. □

6.2.18 Augmentation and Top-Down Design.

Now that we know when a PIT program computes a function, let us explore how such programs can be used in conjunction with top-down design.

Let us consider the problem of solving some task T which is specified over a Σ-algebra A. Let us suppose that it is required we solve T by means of a PIT program S which only uses the operations of A: that is, we require $S \in \mathrm{PIT}(\Sigma)$ and in some general sense the solution to T is $M_A(S)$. Of course, it is the fact that '$M_A(S)$' is a formally defined concept that allows us to formally analyse S as a solution to T.

Now suppose that there is a natural algorithm for solving T, but one which is phrased in terms of some new function $f_A : A^w \longrightarrow A$, in addition to the operations of A. Then intuitively, to solve T by programming over A only, we are left with the subtask of implementing f_A in terms of the operations of A; this is top-down design of course. Furthermore, if the algorithm is coded as a PIT program S' say, then presumably S' will involve symbols $\sigma \in \Sigma$ as well as 'f' which is a new function identifier which names f_A.

Now, S' is not the required solution to T since S' involves the identifier $f \notin \Sigma$, and thus S involves computations which are not operations of A (viz f_A). However, suppose that we can implement f_A by means of an i/o-program $(S_f, In, Out) \in \mathrm{PITIO}(\Sigma)$ in the sense that S_f computes f_A relative to In and Out. Then informally, S' and (S_f, In, Out) constitute a solution to T over A when taken together: S'

solves T in terms of the operations of A and the function f_A; however, by hypothesis, we can compute $f_A(a)$ for any $a \in A^w$ by using operations of A only (since $S_f \in \mathrm{PIT}(\Sigma)$). Let us use the ideas of Section 3.1.5 concerning the augmentation of a data type to make this argument more rigorous.

To say that S' is a PIT program which uses the symbols Σ as well as the function identifier f is to say that $S' \in \mathrm{PIT}(\Omega)$ where $\Omega=(\Sigma,f)$; that is, Ω is the extension of Σ obtained by adding f to $\Sigma_{w,s}$ (recall $f_A : A^w \longrightarrow A_s$).

Now, the original solution to T was supposed to be $M_A(S)$ where $S \in \mathrm{PIT}(\Sigma)$. However, the program we have provided, S', involves an identifier $f \notin \Sigma$ for which there is not (necessarily) an interpretation in A, and so '$M_A(S')$' is not defined; in other words S' is not a satisfactory solution since we can not use our formal methods to analyse it. Of course, the intention was that f names f_A. If we define $B=(A,f_A)$, then B is an Ω-algebra wherein f *does* have an interpretation, namely f^B, where for each $a \in A^w$, $f^B(a)$ is defined by $f^B(a)=f_A(a)$. Thus B is an algebra in which f has the intended interpretation f_A. (Notice that above we should have said 'for each $a \in B^w$' rather than 'for each $a \in A^w$' since technically, f only has an interpretation in B. However, B has the same carriers as A (these algebras only differ in the number of basic operations), and so $B^w=A^w$ for each $w \in S^+$.)

Of course, the meaning of S' in B viz $M_B(S')$ *is* well-defined and so we can formally analyse S' as a solution to T, but we still have to answer the question as to the sense in which S' and (S_f,In,Out) together constitute computation over A.

Since (S_f,x,y) is an i/o-program over Σ, we know that S_f computes $F_A(S_f,x,y)$ relative to x and y by Lemma 6.2.12. However, we have said S_f computes f_A relative to x and y, and so $f_A=F_A(S_f,x,y)$ by Lemma 6.2.4 and thus $B=(A,f_A)=(A,F_A(S_f,x,y))$. Now, formal analysis of S' begins with $M_B(S')$, but by definition of B we now have:

$$M_B(S') = M_{(A,F_A(S_f,In,Out))}(S') : States(A) \longrightarrow States(A) \qquad (42)$$

The point to notice here is that the augmentation notation coupled with the functionality of (S_f,x,y) allows us to explicitly write down the semantics of S' in terms of the algebra A only. This is as it should be since in executing S' we ultimately only ever use the operations of A.

Since the above function f_A is not a primitive operation but a function which is defined by a program, we will call such functions *user-defined functions.*

6.2.19 Function Procedures.

Let us introduce a notation for i/o-programs which tells us by what name the function computed by the i/o-program is to be known as:

Definition. Let S be of the form

$$S = \textbf{function } y = f(x_1,\ldots,x_n) : S_0$$

where $y \in Var_s$ for some $s \in S$, $x=(x_1,\ldots,x_n) \in Var_w$ for some $w \in S^+$, $S_0 \in \mathrm{PIT}(\Sigma)$, and $f \notin \Sigma$. If $(S_0,x,y) \in \mathrm{PITIO}(\Sigma)_{w,s}$, then we say S is a *function procedure* of *arity* (w,s). We also define f to be the *identifier of* S, in symbols: $id(S)=f$.

□

Notice that a function procedure such as S above determines a new signature $\Omega=(\Sigma,id(S))$, and a new (Ω-) algebra $B=(A,F_A(S_o,x,y))$; thus Ω is a signature over which we can write programs which use $F_A(S_o,x,y)$ as a basic operation, and since B is an Ω-algebra, B is an algebra in which such programs have a well-defined formal meaning.

A function procedure is a convenient notation for expressing the implementation of a user-defined function: the intention behind the syntax of a function program of the above form is that $x_1,...,x_n$ are the inputs to the procedure, S_o is a program which calculates the value '$f(x_1,...,x_n)$' which is ultimately stored in the output variable y.

How does a function procedure implement a user-defined function? Given a user-defined function $f_A:A^w\longrightarrow A_s$, choose a function procedure of the above form; if $F_A(S_o,x,y)=f_A$, then there is a strong sense in which S implements f_A: as we have explained above, the interpretation of the symbol f in the algebra B is defined to be $F_A(S_o,x,y)=f_A$; this follows automatically from the definition of $B=(A,F_A(S_o,x,y))$.

Let us explore this sense of implementation more carefully. Suppose $S'\in PIT(\Omega)$ is a program which involves an expression of the form $f(e_1,...,e_n)$; for definiteness, assume S' involves the statement $z:=f(e_1,...,e_n)$. In order to ascertain the value assigned to z, we proceed as though f were any ordinary symbol $\sigma\in\Sigma$: given some initial state ρ, we first evaluate $e_1,...,e_n$ under ρ (in parallel), and then we apply 'f'. Suppose e_i evaluates to a_i under ρ for $i=1,...,n$; then the value assigned to z is intended to be $f_A(a_1,...,a_n)$.

Conceptually, we imagine the value $f_A(a_1,...,a_n)$ to be calculated as follows: once $a_1,...,a_n$ have been calculated, $x_1,...,x_n$ (of the function procedure) are assigned the values $a_1,...,a_n$ respectively, and then we execute S_o. Now, since (S_o,x,y) is an i/o-program we know that S_o computes $F_A(S_o,x,y)$ relative to x and y (see Lemma 6.2.12), and so

$$\pi_A(y)(M_A(S_o)(\rho'))=F_A(S_o,x,y)(a_1,...,a_n)=f_A(a_1,...,a_n)$$

where $\rho'\in States(A)$ is any state such that $\pi_A(x)(\rho')=(a_1,...,a_n)$ (see Definition 6.2.3). However, since y is a single variable here we have

$$M_A(S_o)(\rho')(y)=f_A(a_1,...,a_n) \tag{43}$$

Since we assign $a_1,...,a_n$ to $x_1,...,x_n$ prior to executing S_o, we do execute S_o in a state ρ' where $\pi_A(x)(\rho')=a$, and thus from (43), the final value of y is $f_A(a_1,...,a_n)$. Finally, as the function procedure notation is intended to imply, we think of y as the output variable of S, that is, the final value of y is the value returned to the calling program and so z is assigned the value $f_A(a_1,...,a_n)$ as intended.

The concept of an augmentation allows us to *prove* that z is assigned $f_A(a_1,...,a_n)$ in the following way. Formally, the value assigned to z is $M_B(z:=f(e_1,...,e_n))(\rho)(z)$ where ρ is any initial state. We now formally calculate this value as follows:

$$M_B(z:=f(e_1,...e_n))(\rho)(z)=\rho\{E_B(f(e_1,...,e_n))(\rho)/z\}(z)$$

$$= E_B(f(e_1,\ldots,e_n))(\rho)$$
$$= f^B(E_B(e_1)(\rho),\ldots,E_B(e_n)(\rho))$$
$$= f^B(a_1,\ldots,a_n)$$

(when $E_B(e_i)(\rho)=a_i$ for $i=1,\ldots,n$)

$$= F_A(S_o,x,y)(a_1,\ldots,a_n) \tag{44}$$

(by definition of B)

$$= f_A(a_1,\ldots,a_n)$$

Alternatively, since (S_o,x,y) computes $F_A(S_o,x,y)$, from (44) we have

$$M_B(z:=f(e_1,\ldots e_n))(\rho)(z) = F_A(S_o,x,y)(a_1,\ldots,a_n)$$
$$= \pi_A(y)(M_A(S_o)(\rho'))$$

(where ρ' is any state such that $\pi_A(x)(\rho')=a$)

$$= M_A(S_o)(\rho')(y)$$

This last expression for the value assigned to z clearly shows how that value is calculated.

Vector-Valued Functions. Later we will see instances of top-down design which use *vector*-valued user-defined functions $f_A : A^u \longrightarrow A^v$.

We extend the definition of a function procedure to the vector-valued case as follows:

Definition. Let S be of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

where $y=(y_1,\ldots,y_m)\in Var_v$ for some $v\in S^+$, $x=(x_1,\ldots,x_n)\in Var_u$ for some $u\in S^+$, $S_o\in \mathrm{PIT}(\Sigma)$, and $f_1,\ldots,f_n\notin\Sigma$. If $(S_o,x,y)\in \mathrm{PITIO}(\Sigma)_{u,v}$, then we say S is a *function procedure* of arity (u,v). Also we define $f_1,\ldots,f_m$ to be the *identifiers of* S, in symbols: $id(S)=(f_1,\ldots,f_m)$. □

Notice that a function program of this form defines a new signature $\Omega=(\Sigma,id(S))$ where f_i is adjoined to Σ_{u,v_i} for $i=1,\ldots,m$. Also, S determines an Ω-algebra $B=(A,F_A(S_o,x,y))$, but in this case A is augmented with the *coordinates* of $F_A(S_o,x,y)$ (see Section 3.1.5). Thus the interpretation of each f_i in B is defined by

$$f_i^B(a) = F_A(S_o,x,y)_i(a)$$

for each $a\in A^u$. (Here $F_A(S_o,x,y)_i(a)$ denotes the ith coordinate of $F_A(S_o,x,y)(a)$ for $i=1,\ldots,m$.)

The idea behind a function program in the vector-valued case is similar to the single-valued case:

Since (S_o,x,y) is an i/o-program, S_o computes $F_A(S_o,x,y)$ relative to x and y, and thus

$$\pi_A(y)(M_A(S_o)(\rho)) = F_A(S_o,x,y)(a) \tag{45}$$

when $\pi_A(x)(\rho)=a$. Taking the identity (45) coordinatewise yields:

$$M_A(S_o)(\rho)(y_i) = F_A(S_o,x,y)_i(a)$$

for $i=1,\ldots,m$.

The intention behind the syntax of S is that for each $f_i\in id(S)$, to evaluate an expression of the form $f_i(e_1,\ldots,e_n)$, we evaluate $e_1,\ldots,e_n$ and then assign the resulting values to $x_1,\ldots,x_n$ respectively; we then execute S_o and the final value of y_i is the value returned as the value of the expression.

Notice that as we have explained it, the process of evaluating $f_i(e_1,\ldots,e_n)$ is independent of i in the sense that we must always execute S_0 for any i. Thus in evaluating $f_i(e_1,\ldots,e_n)$ we also evaluate $f_j(e_1,\ldots,e_n)$ for every other $j \neq i$, although the result of these other evaluations is ignored (we only return the value of y_i). At first sight this seems rather inefficient, especially if we consider a statement such as

$$z_1,z_2 := f_i(e_1,\ldots,e_n),f_j(e_1,\ldots,e_n)$$

Here we must execute S_0 twice even though having executed S_0 once to evaluate $f_i(e_1,\ldots,e_n)$ (the final value of y_i), the value of $f_j(e_1,\ldots,e_n)$ is already available (the value of y_j). However, because of the concurrent interpretation of the multiple assignment statement, there is no loss of performance as we shall see below.

Exercise. In the context of the preceding discussion let S_1 be the statement

$$S_1 = z_1,\ldots,z_m := f_1(e_1,\ldots,e_n),\ldots,f_m(e_1,\ldots,e_n)$$

where $z=(z_1,\ldots,z_n) \in Var_\tau$ and $e_i \in \mathrm{EXP}(\Omega)_{\mu_i}$ for $i=1,\ldots,n$. Now prove the following:

(i) $S_1 \in \mathrm{PIT}(\Omega)$, and

(ii) If $\rho \in States(A)$ is such that $E_B(e_i)(\rho)=a_i$ for $i=1,\ldots,n$, then

$$\pi_A(z)(M_B(S_1)(\rho)) = f_A(a_1,\ldots,a_n)$$

Performance. We have seen how the concepts of signature extension and algebra augmentation allow us to formally analyse a program which is the product of top-down design. We will now explain how the performance of such a program may analysed via the extension of a performance measure (see Section 3.2).

In the notation of the preceding discussion, when a function procedure S is of arity (w,s) for some $w \in S^+$ and $s \in S$, S determines a signature $\Omega=(\Sigma,f)$ when $id(S)=f$, and an Ω-algebra $B=(A,F_A(S_0,x,y))$. Recall it was the construction of B that allowed us to write down the formal meaning of a program $S' \in \mathrm{PIT}(\Omega)$; furthermore, it is a consequence of the way that the augmentation of A is defined that the interpretation of f in B is $F_A(S_0,x,y)$; this formalises the intuition behind how '$f(\cdots)$' should be evaluated of course.

Now let P be a performance measure for A which is based on clock C. Then in order to calculate the complexity of $S' \in \mathrm{PIT}(\Omega)$, we need to extend P with a new performance estimation function for the new symbol f: if $F_A(S_0,x,y)$ has domain A^w then this new estimation function λ_f say, must be of the form $\lambda_f : A^w \longrightarrow C^+$. Furthermore, since we must execute S_0 to evaluate $f(\cdots)$, it is appropriate to choose λ_f to be the complexity of executing S_0.

Let $Q=(P,\lambda_P^{IO}(S_0,x,y))$. Then Q is a performance measure for B, and in particular, we have for any $a \in A^w$,

$$\begin{aligned} f^Q(a) &= \lambda_P^{IO}(S_0,x,y)(a) \\ &= \lambda_P^{PIT}(S_0)(\rho) \end{aligned}$$

where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho)=a$.

In the case that S is of arity (u,v) for some $u,v \in S^+$, then S is of the form:

$$S = \text{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_0$$

What is an appropriate performance measure for $B = (A, F_A(S_0,x,y))$ in this case?

First recall that here we extend Σ with a vector of function identifiers $id(S) = (f_1,\ldots,f_m)$, and A is augmented with the m coordinates of $F_A(S_0,x,y)$. Thus if Q is to be a performance measure for B, we must extend P with m new performance estimation functions $\lambda_1,\ldots,\lambda_m$ say, with $\lambda_i : A^u \longrightarrow C^+$ for $i = 1,\ldots,m$ since the domain of $F_A(S_0,x,y)$ is A^u. Now, to evaluate $f_i(\cdots)$ we must execute S_0 for *any* $i \in [1,m]$: the only difference between evaluating f_i and f_j (on the same arguments) is that the value returned is the final value of y_i for f_i, and the final value of y_j for f_j; we must execute S_0 in each case however.

Thus in the case of a vector-valued function, it is appropriate to define Q by defining each λ_i by

$$\lambda_i(a) = \lambda_P^{IO}(S_0,x,y)(a)$$

for every $a \in A^u$ and for $i = 1,\ldots,m$.

Notice that the complexity of f_i according to Q is independent of the choice of i, and so if S_1 is the program of the previous exercise, then:

$$\begin{aligned}
\lambda_Q^{PIT}(S_1)(\rho) &= \max_{1\le i\le n}\{\lambda_Q^{EXP}(f_i(e_1,\ldots,e_n))(\rho)\} \\
&= \max_{1\le i\le n}\{f_i^Q(E_B(e_1)(\rho),\ldots,E_B(e_n)(\rho))\} \\
&= \max_{1\le i\le n}\{\lambda_i(a_1,\ldots,a_n)\} \\
&\quad \text{(when } E_B(e_i)(\rho) = a_i \text{ for } i = 1,\ldots,n\text{)} \\
&= \max_{1\le i\le n}\{\lambda_P^{IO}(S_0,x,y)(a_1,\ldots,a_n)\} \\
&= \lambda_P^{IO}(S_0,x,y)(a_1,\ldots,a_n)
\end{aligned}$$

Thus the cost of executing S_1 is the cost of executing S_0 only once although conceptually S_0 is executed m times in executing S_1.

Further Remarks. In practice the top-down design of an algorithm leads to the use of more than one function procedure. Also, top-down design may be repeatedly applied to the task of implementing the function procedures themselves. The ultimate extension of the concept of a function program to cover these two cases results in two new languages: FPIT and *FG*. FPIT is a language of function procedures which involve lists of further function procedures as sub-programs, and FG is a language of lists of function procedures which we call *function groups*. Actually, we are only interested in FPIT since it is the target implementation language for PR. For our purposes, FG is only a notational contrivance to smooth the definition of FPIT, although from a theoretical perspective, FG is of interest in its own right (see Jervis[1988]).

6.3 THE LANGUAGE FPIT.

In this section we will define the syntax, semantics, and complexity theory of our simulation language FPIT. After defining the language and establishing some simple facts, we will explain our strategy for implementing PR schema in FPIT.

6.3.1 Syntax.

Let Σ be a standard S-sorted signature. Below, we define

(1) FPIT(Σ), the collection of *function programs over* Σ. FPIT(Σ) is an $S^+ \times S^+$ indexed family

$$\text{FPIT}(\Sigma) = \langle \text{FPIT}(\Sigma)_{u,v} : u,v \in S^+ \rangle$$

wherein each $S \in \text{FPIT}(\Sigma)_{u,v}$ we call a *function program of arity* (u,v).

(2) FG(Σ), the collection of Σ-*function-groups*.

Also, for each $S \in \text{FPIT}(\Sigma)$ and $G \in \text{FG}(\Sigma)$, we define $id(S)$, $id(G)$, $all(S)$, and $all(G)$, which are collections of function identifiers, and also $depth(S)$, $depth(G)$, which are measures of 'nestedness'. Further, for each $G \in \text{FG}(\Sigma)$ we also define $sig(G)$ which is a signature, and $|G|$ which is the *length* of G.

The definitions of FPIT(Σ) and FG(Σ) are mutually recursive and proceed as follows:

P1 *Simple Function Programs.* Suppose S is of the form:

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

where:

(a) $y = (y_1,\ldots,y_m) \in Var_v$ for some $v \in S^+$ (recall from Definition 6.2.2 that this means the variables $y_1,\ldots,y_m$ are distinct);

(b) $x = (x_1,\ldots,x_n) \in Var_u$ for some $u \in S^+$;

(c) $\{x_1,\ldots,x_n\} \cap \{y_1,\ldots,y_m\} = \emptyset$;

(d) $(S_o,x,y) \in \text{PITIO}(\Sigma)$;

(e) $\{f_1,\ldots,f_m\}$ is a collection of $m > 0$ *distinct* identifiers, and

(f) $\{f_1,\ldots,f_m\} \cap \Sigma = \emptyset$.

Then $S \in \text{FPIT}(\Sigma)$ with:

(i) arity (u,v); that is, $S \in \text{FPIT}(\Sigma)_{u,v}$;

(ii) $depth(S) = 0$;

(iii) $id(S) = (f_1,\ldots,f_m)$, and

(iv) $all(S) = id(S)$.

When S is of the above form, we refer to S_o as the *body* of S.

G1 *Single Function Groups.* Suppose G is of the form $G = S$ for some $S \in \mathrm{FPIT}(\Sigma)$ with arity (u,v) say. Then $G \in \mathrm{FG}(\Sigma)$ with:

(i) $depth(G) = depth(S)$;

(ii) $id(G) = id(S)$;

(iii) $all(G) = all(S)$;

(iv) $sig(G)$ is the (u,v)-extension of Σ: $sig(G) = (\Sigma, id(S))$, and

(v) $|G| = 1$.

P2 *Nested Function Programs.* Suppose S is of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_0$$

where:

(a) $y = (y_1,\ldots,y_m) \in Var_v$ for some $v \in S^+$;

(b) $x = (x_1,\ldots,x_n) \in Var_u$ for some $u \in S^+$;

(c) $\{x_1,\ldots,x_n\} \cap \{y_1,\ldots,y_m\} = \emptyset$;

(d) $G \in \mathrm{FG}(\Sigma)$;

(e) $(S_0, x, y) \in \mathrm{PITIO}(sig(G))$;

(f) $\{f_1,\ldots,f_m\}$ is a collection of $m > 0$ *distinct* identifiers, and

(g) $\{f_1,\ldots,f_m\} \cap sig(G) = \emptyset$.

Then $S \in \mathrm{FPIT}(\Sigma)$ with:

(i) arity (u,v);

(ii) $depth(S) = depth(G) + 1$;

(iii) $id(S) = (f_1,\ldots,f_m)$, and

(iv) $all(S) = id(S) \cup all(G)$.

When S is of the above form, we refer to S_0 as the body of S, and to G as the *group* of S.

G2 *Multiple Function Groups.* Suppose G is of the form $G = G_0;S$ where:

(a) $G_0 \in \mathrm{FG}(\Sigma)$, and

(b) $S \in \mathrm{FPIT}(sig(G_0))_{u,v}$ for some $u,v \in S^+$.

Then $G \in \mathrm{FG}(\Sigma)$ with:

(i) $depth(G) = \max\{depth(G_0), depth(S)\}$;

(ii) $id(G) = id(G_o) \cup id(S)$;

(iii) $all(G) = all(G_o) \cup all(S)$;

(iv) $sig(G)$ is the (u,v)-extension of $sig(G_o)$: $sig(G) = (sig(G_o), id(S))$, and

(v) $|G| = |G_o| + 1$. □

Notes.

(i) The syntax

$$\textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n):$$

which begins any function program S we call the *header* of S.

(ii) Throughout the remainder of this chapter we will use the notation $x = (x_1,\ldots,x_n)$ and $y = (y_1,\ldots,y_m)$ for the vectors of a function program's input and output variables respectively: invariably, when S is a function program of arity (u,v) we denote the lengths of u and v by n and m respectively.

(iii) Note that for any function group G the quantity '$sig(G)$' is dependent on the underlying signature. For example, if $G \in FG(\Sigma)$, then $sig(G)$ is really $sig_\Sigma(G)$ since $sig(G)$ is always an extension of Σ. This may seem a rather trivial observation, but in later sections this fact will be significant but obscured and so we make the point whilst it is still obvious.

Discussion. It should be apparent that the concept of a function program generalises the earlier idea of a function procedure in a natural way: a function program is a recursively defined function procedure in the sense that a function program is a function procedure which may involve further function procedures/programs as sub-programs. Before we give the semantics of FPIT and FG we comment on the auxiliary definitions made above.

First, the collection $id(S)$ is simply the identifiers declared in the header of S. Also, it should be clear that $all(S)$ and $all(G)$ comprise all the function identifiers declared anywhere in S and G respectively. The intuition behind $depth(S)$, $depth(G)$ and $|G|$ should be equally obvious. More interesting are $id(G)$ and $sig(G)$ since the definition of these collections of identifiers implicitly tell us FPIT's scoping rules (that is, the rules that tell us which function identifiers are available in which parts of a function program).

Note that any function group G can be written in the form $G = S_1; \cdots ; S_m$ for some $m \geq 1$ and some function programs $S_1,\ldots,S_m$. Now notice that $id(G)$ and $sig(G)$ are defined in such a way that $sig(G)$ contains the symbols of Σ and the identifiers declared by the headers of $S_1,\ldots,S_m$, but *not* the identifiers declared *within* these function programs. For example, if S_i has a local group G_i, then the identifiers introduced by G_i (namely $id(G_i)$) are not included in $id(G)$ and may not be used in the body of a function program whose group is G. Alternatively, condition (b) in clause G2 above says that when $G = S_1; \cdots ; S_m$ the body of S_i (for $i > 1$) *may* use the identifiers previously introduced by the headers of $S_1,\ldots,S_{i-1}$ (but not the identifiers declared within these function programs). Finally, we note that in clause P2 above, from (g) we infer that the identifiers of a nested function program S are not added to the operator symbols that the body of S may use, and thus from clause (e) of P2 we infer that function

programs may not call themselves; that is FPIT is not a recursive programming language.

6.3.2 Semantics.

Let A be a standard Σ-algebra. Below, we define

(1) For each $S \in \mathrm{FPIT}(\Sigma)$, $F_A(S)$, the *functional meaning of* S in A: F_A is an $S^+ \times S^+$-indexed family of mappings

$$F_A = \langle F_A^{u,v} : u,v \in S^+ \rangle$$

where for each $u,v \in S^+$, $F_A^{u,v}$ is a mapping

$$F_A^{u,v} : \mathrm{FPIT}(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow A^v]$$

(2) For each $G \in \mathrm{FG}(\Sigma)$, $Alg_A(G)$, which is a $sig(G)$-algebra.

Let $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$ and let $G \in \mathrm{FG}(\Sigma)$. The definitions of $F_A^{u,v}(S)$ (ambiguously denoted $F_A(S)$) and $Alg_A(G)$ are mutually recursive and follow the definitions of $\mathrm{FPIT}(\Sigma)$ and $\mathrm{FG}(\Sigma)$:

P1 *Simple Function Programs.* Suppose S is of the form:

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

Then by definition of $\mathrm{FPIT}(\Sigma)$, we have $(S_o,x,y) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$, and thus by Theorem 6.2.11, the functional meaning of (S_o,x,y) in A, viz $F_A(S_o,x,y)$, is a well-defined function $F_A(S_o,x,y) : A^u \longrightarrow A^v$.

Here we define $F_A(S)$ by:

$$F_A(S) = F_A(S_o,x,y)$$

That is,

$$F_A(S) : A^u \longrightarrow A^v$$

where for each $a \in A^u$,

$$F_A(S)(a) = F_A(S_o,x,y)(a)$$

G1 *Single Function Groups.* Suppose $G = S$ for some $S \in \mathrm{FPIT}(\Sigma)$.

Here $sig(G) = (\Sigma, id(S))$. Also, if S is of arity (u,v) say, and $id(S) = \{f_1,\ldots,f_m\}$, then $f_i \in sig(G)_{u,v_i}$ for $i = 1,\ldots,m$. Since A is a Σ-algebra, we can turn A into a $sig(G)$-algebra A_G say, by augmenting A with an interpretation of f_i for $i = 1,\ldots,m$; since $f_i \in sig(G)_{u,v_i}$, the interpretation of f_i in A_G must be a function from A^u into A_{v_i}. Let $F_A(S)_i$ denote the ith coordinate function of $F_A(S)$ for $i = 1,\ldots,m$. Then, since $F_A(S) : A^u \longrightarrow A^v$, we have $F_A(S)_i : A^u \longrightarrow A_{v_i}$. Since $F_A(S)_i$ is of the appropriate functionality, we can define $A_G = Alg_A(G)$ to be the augmentation of A obtained by adding $f_i^{A_o}$ for $i = 1,\ldots,m$, where $f_i^{A_o}$ is defined by

$$(\forall a \in A^u)\ (f_i^{A_o}(a) = F_A(S)_i(a))$$

Given the notation (A, f_A) when f_A is vector-valued (see Section 3.1.5), we can equivalently define $Alg_A(G)$ by

$$Alg_A(G) = (A, F_A(S))$$

P2 *Nested Function Programs.* Suppose S is of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

Then $G \in \mathrm{FG}(\Sigma)$ and $(S_o,x,y) \in \mathrm{PITIO}(sig(G))_{u,v}$ for some $u,v \in S^+$; and hence by Theorem 6.2.11, the functional meaning of (S_o,x,y) in $A_G = Alg_A(G)$, viz $F_{A_o}(S_o,x,y)$ is a well-defined function $F_{A_o} : A^u \longrightarrow A^v$.

Here we define $F_A(S)$ by:

$$F_A(S) = F_{A_o}(S_o,x,y)$$

That is,

$$F_A(S) : A^u \longrightarrow A^v$$

where for each $a \in A^u$,

$$F_A(S)(a) = F_{A_o}(S_o,x,y)(a)$$

G2 *Multiple Function Groups.* Suppose G is of the form $G = G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_o))$.

Here $sig(G) = (sig(G_o),id(S))$, and if S is of arity (u,v) then $id(S) = \{f_1,\ldots,f_m\}$, and $f_i \in sig(G)_{u,v_i}$ for $i = 1,\ldots,m$. Now, since $A_o = Alg_A(G_o)$ is a $sig(G_o)$-algebra, we can turn A_o into a $sig(G)$-algebra A_G say, by augmenting A_o with an interpretation of f_i for $i = 1,\ldots,m$; since $f_i \in sig(G)_{u,v_i}$, the interpretation of f_i in A_G must be a function from A^u into A_{v_i}.

Now, since $S \in \mathrm{FPIT}(sig(G_o))$, the functional meaning of S is a functional meaning in A_o, viz $F_{A_o}(S)$. Similar to the case of single function groups, let $F_{A_o}(S)_i$ denote the ith coordinate function of $F_{A_o}(S)$. Then, since $F_{A_o}(S) : A^u \longrightarrow A^v$, we have $F_{A_o}(S)_i : A^u \longrightarrow A_{v_i}$. Since $F_{A_o}(S)_i$ is of the appropriate functionality, we can define $A_G = Alg_A(G)$ to be the augmentation of A_o obtained by adding $f_i^{A_o}$ for $i = 1,\ldots,m$, where $f_i^{A_o}$ is defined by

$$(\forall a \in A^u)\ \ (f_i^{A_o}(a) = F_{A_o}(S)_i(a))$$

Equivalently, $Alg_A(G)$ is defined by

$$Alg_A(G) = (A_o, F_{A_o}(S))$$

Note. Notice that for any $G \in \mathrm{FG}(\Sigma)$, A and $A_G = Alg_A(G)$ have the same carriers, namely the carriers of A. (The only difference between A and A_G is that A_G has more operations.) A consequence of this is that $States(A) = States(A_G)$ which in turn means that the state projection function π_{A_o} is identically π_A. This usage of algebras as a subscript on functions can be contrasted with the case of the meaning or interpretation of a program, viz 'M_A': when S is a PIT program over the signature of a function group $G \in \mathrm{FG}(\Sigma)$, S may use symbols $\sigma \in sig(G)$ which do not have interpretations in A but in A_G; thus the meaning of S is properly $M_{A_o}(S)$ and not $M_A(S)$.

Discussion. Given a function program S with body S_o, input variables $x=(x_1,...,x_n)$, and output variables $y=(y_1,...,y_m)$, (S_o,x,y) is always an i/o-program and therefore always computes some function on the underlying algebra. Since the computation performed by a function program is the computation performed by its body, it is appropriate to define the semantics of a function program to be the function computed by its body (as we have done), and thus it is natural to say that each function program $S \in \mathrm{FPIT}(\Sigma)$ *computes* $F_A(S)$. Now, our current objective is to show that every synchronous algorithm can be simulated in FPIT by implementing PR in FPIT. We will do this by showing that any scheme $\alpha \in \mathrm{PR}(\Sigma)$ can be implemented in FPIT by some $S=S_\alpha \in \mathrm{FPIT}(\Sigma)$ which computes $[\![\alpha]\!]_A$; that is, by showing $F_A(S)(a)=[\![\alpha]\!]_A(a)$ for each a in the common domain of $F_A(S)$ and $[\![\alpha]\!]_A$. We will prove that this is the case for any Σ and A and hence true for a stream algebra $\underline{A}$ with signature $\underline{\Sigma}$; this is the ultimate proof that synchronous algorithms can be simulated in FPIT, in $\mathrm{FPIT}(\underline{\Sigma})$ to be precise.

We now conclude the definition of FPIT by giving a complexity theory for the language:

6.3.3 Performance.

Let A be a standard Σ-algebra, and let P be a performance measure for A which is based on clock C. Below, we define

(1) For each $S \in \mathrm{FPIT}(\Sigma)$, $\lambda_P^{FPIT}(S)$, the *length of computation of S with respect to P*: λ_P^{FPIT} is an S^+-indexed family of mappings

$$\lambda_P^{FPIT} = <\lambda_P^u : u \in S^+>$$

where for each $u \in S^+$, λ_P^u is a mapping

$$\lambda_P^u : \bigcup_{v \in S^+} \mathrm{FPIT}(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow C^+]$$

(2) For each $G \in \mathrm{FG}(\Sigma)$, P_G, which is a performance measure for the $sig(G)$-algebra $Alg_A(G)$ (and based on clock C).

Let $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$ and $G \in \mathrm{FG}(\Sigma)$. The definitions of $\lambda_P^u(S)$ (ambiguously denoted $\lambda_P^{FPIT}(S)$) and P_G are mutually recursive and follow the definitions of $\mathrm{FPIT}(\Sigma)$ and $\mathrm{FG}(\Sigma)$:

P1 *Simple Function Programs.* Suppose S is of the form:

$$S = \textbf{function } y_1,...,y_m = f_1(x_1,...,x_n),...,f_m(x_1,...,x_n) : S_o$$

Here we define $\lambda_P^{FPIT}(S)$ by

$$\lambda_P^{FPIT}(S) : A^u \longrightarrow C^+$$

where for each $a \in A^u$,

$$\lambda_P^{FPIT}(S)(a) = \lambda_P^{IO}(S_o,x,y)(a)$$

G1 *Single Function Groups.* Suppose $G=S$ for some $S \in \mathrm{FPIT}(\Sigma)$.

Then here, $sig(G)=\Sigma \cup id(S)$. Also, if S is of arity (u,v) say, and $id(S)=\{f_1,...,f_m\}$, then $f_i \in sig(G)_{u,v_i}$ for $i=1,...,m$. Now, since P is a performance measure for the Σ-algebra A, we can turn P into a performance measure P_G for the $sig(G)$-algebra $Alg_A(G)$ by extending P with a performance estimation for f_i for $i=1,...,m$; since $f_i \in sig(G)_{u,v_i}$, a performance estimation of f_i in P_G must be a

function from A^u into C^+. Now, since $\lambda_P^{FPIT}(S):A^u \longrightarrow C^+$, $\lambda_P^{FPIT}(S)$ is of exactly the right functionality, and so we can define P_G to be the extension of P obtained by adding $f_i^{P_o}$ for $i=1,\ldots,m$, where $f_i^{P_o}$ is defined by

$$(\forall a \in A^u)\ \ (f_i^{P_o}(a) = \lambda_P^{FPIT}(S)(a))$$

(Notice that with respect to P_G, the cost of evaluating an expression of the form $f_i(\cdots)$ is independent of i: we charge the cost of executing S for any coordinate i.) Equivalently, given the definition of Section 3.2.2, P_G is defined by

$$P_G = (P, \lambda_P^{FPIT}(S))$$

P2 *Nested Function Programs.* Suppose S is of the form

$$S = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

Then $G \in \mathrm{FG}(\Sigma)$ and $(S_o,x,y) \in \mathrm{PITIO}(sig(G))_{u,v}$ for some $u,v \in S^+$.

Here we define $\lambda_P^{FPIT}(S)$ by

$$\lambda_P^{FPIT}: A^u \longrightarrow C^+$$

where for each $a \in A^u$,

$$\lambda_P^{FPIT}(S)(a) = \lambda_{P_o}^{IO}(S_o)(\rho)$$

where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho)=a$. (Similar to the case of simple function programs, $\lambda_P^{FPIT}(S)$ is well-defined by Theorem 6.2.8; however, note that in the statement of that theorem we must take $\Sigma = sig(G)$.)

G2 *Multiple Function Groups.* Suppose G is of the form $G = G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_o))$.

Then here, $sig(G) = sig(G_o) \cup id(S)$, and since S is of arity (u,v), then $id(S) = \{f_1,\ldots,f_m\}$, and $f_i \in sig(G)_{u,v_i}$ for $i=1,\ldots,m$. Now, since $P_o = P_{G_\bullet}$ is a performance measure for the $sig(G_o)$-algebra $A_o = Alg_A(G_o)$, we can turn P_o into a performance measure P_G for the $sig(G)$-algebra $A_G = Alg_A(G)$ by extending $P_{G_\bullet}$ with a performance estimation for f_i for $i=1,\ldots,m$; since $f_i \in sig(G)_{u,v_i}$, a performance estimation for f_i in P_G must be a function from A^u into C^+.

Now, since $S \in \mathrm{FPIT}(sig(G_o))$, the length of computation function for S is a length of computation function with respect to P_o, viz $\lambda_{P_\bullet}^{FPIT}(S)$. Now, since $\lambda_{P_\bullet}^{FPIT}(S):A^u \longrightarrow C^+$, this function is of exactly the right functionality, and so we can define P_G to be the extension of P obtained by adding $f_i^{P_o}$ for $i=1,\ldots,m$, where $f_i^{P_o}$ is defined by

$$(\forall a \in A^u)\ \ (f_i^{P_o}(a) = \lambda_{P_o}^{FPIT}(S)(a))$$

(Again notice that the cost with respect to P_G of evaluating an expression of the form $f_i(\cdots)$ is independent of i.) Equivalently, P_G is defined by

$$P_G = (P_o, \lambda_{P_\bullet}^{FPIT}(S))$$ □

We have now concluded the definition of FPIT. Before we consider some examples, here are some simple facts concerning function programs and function groups:

6.3.4 Lemma. *Let Σ and Σ' be S-sorted standard signatures with $\Sigma' \supseteq \Sigma$. Then*

(i) *for each $u, v \in S^+$, and for every $S \in \mathrm{FPIT}(\Sigma)_{u,v}$, if all$(S) \cap \Sigma' = \emptyset$ then $S \in \mathrm{FPIT}(\Sigma')_{u,v}$, and*

(ii) *for every $G \in \mathrm{FG}(\Sigma)$, if all$(G) \cap \Sigma' = \emptyset$ then $G \in \mathrm{FG}(\Sigma')$.*

Proof. We will not prove this lemma formally since its formal proof is very long: one must prove (i) and (ii) by simultaneous induction on the depth of function programs and function groups and by subinduction on the length of function groups. Instead we will explain why the lemma is intuitively true.

First consider a program $S \in \mathrm{PIT}(\Sigma)$. To say that S is defined 'over' Σ is to say that S only involves operator symbols $\sigma \in \Sigma$. Thus if $\Sigma' \supseteq \Sigma$ then certainly $S \in \mathrm{PIT}(\Sigma')$: it is simply that S does not involve any of the (new) operator symbols in the difference $\Sigma' - \Sigma$.

For essentially the same reason, if $\Sigma' \supseteq \Sigma$ and the new symbols in Σ' do not occur in a function program $S \in \mathrm{FPIT}(\Sigma)$, then $S \in \mathrm{FPIT}(\Sigma')$. The only difference is that to have $S \in \mathrm{FPIT}(\Sigma')$ no function identifier introduced in the header of any (sub-) function program involved in S can be a member of Σ'. However, since *all*(S) is the collection of all function identifiers occurring in headers in S (including that of S itself), it is clear that *all*$(S) \cap \Sigma' = \emptyset$ is sufficient to guarantee $S \in \mathrm{FPIT}(\Sigma')$ proving part (i) of the lemma. (A similar argument can be made for part (ii) of the lemma.) □

6.3.5 Lemma. *Let $S_1, \ldots, S_n \in \mathrm{FPIT}(\Sigma)$. Also let (strings) $G_1, \ldots, G_n$ be defined by*

$$G_1 = S_1$$
$$G_{j+1} = G_j \; ; \; S_{j+1}$$

If all$(S_{j+1}) \cap sig(G_j) = \emptyset$ for $j = 1, \ldots, n-1$, then $G_i \in \mathrm{FG}(\Sigma)$ for $i = 1, \ldots, n$.

Proof. Omitted. (Hint: use induction on n together with the previous lemma.) □

6.3.6 Examples of FPIT.

In this section we will illustrate the programming of functions in FPIT; we will see worked examples of function groups on the way. We use some simple functions defined by PR schema. In this way we will see that FPIT is indeed tailored to the implementation of PR. In the next section we will define a compiler that incorporates and generalises the examples.

Let us first say what we mean by 'implementation':

Definition. Let $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ and $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u, v \in S^+$. Then we say S implements α if $F_A(S) = [\![\alpha]\!]_A$; that is, if

$$(\forall a \in A^u) \;\; (F_A(S)(a) = [\![\alpha]\!]_A(a))$$

Examples. As a first example, suppose $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and some $s \in S$. Then $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$. Now let S be defined by

$$S = \textbf{function } y = f(x_1, \ldots, x_n) : S_0$$

where:

$$y \in Var_s;$$

$x = (x_1,\ldots,x_n) \in Var_w$, and

$S_o = y := \sigma(x_1,\ldots,x_n)$.

To show S implements α choose $a \in A^u$. Then by definition of $F_A(S)$ we have:

$$F_A(S)(a) = F_A(S_o,x,y)(a)$$
$$= \pi_A(y)(M_A(y:=\sigma(x_1,\ldots,x_n))(\rho))$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho)=a$)

$$= \pi_A(y)(\rho\{E_A(\sigma(x_1,\ldots,x_n))(\rho)/y\})$$
$$= \pi_A(y)(\rho\{\sigma^A(\rho(x_1),\ldots,\rho(x_n))/y\})$$
$$= \pi_A(y)(\rho\{\sigma^A(\pi_A(x)(\rho))/y\})$$
$$= \sigma^A(\pi_A(x)(\rho))$$
$$= \sigma^A(a)$$
$$= [\![\sigma]\!]_A(a)$$
$$= [\![\alpha]\!]_A(a)$$

Thus S implements α as claimed.

It is not difficult to see that the other basis schema can be implemented by essentially the same function program S: for $\alpha=c^w$ and $\alpha=U_i^w$ we simply substitute $S_o=y:=c$ and $S_o=y:=x_i$ respectively. (We will prove this later.) □

The next example illustrates our strategy for implementing composite schema: if α is a composite scheme that is built up from subschema $\alpha_1,\ldots,\alpha_m$ say (for example $\alpha=\alpha_2\circ\alpha_1$), then we first compile each α_i into a function program S_i for $i=1,\ldots,m$; this defines a group $G=S_1;\cdots;S_m$, which in turn determines a $sig(G)$-algebra in which each (coordinate of) $[\![\alpha_i]\!]_A$ is a basic operation: it is then usually a simple task to write a PIT i/o-program over $sig(G)$ which implements α.

(2) Let α be defined by $\alpha = DC(\beta,\alpha_1,\alpha_2)$ where β, α_1 and α_2 are any schema with $\beta \in PR(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$ for some $u,v \in S^+$. ('B' names the sort 'Boolean' of course.) Then $\alpha \in PR(\Sigma)_{u,v}$.

The task we set ourselves is to implement α in FPIT(Σ) assuming that we can implement β, α_1 and α_2 in FPIT(Σ).

Let us first assume that there exists $S_b \in FPIT(\Sigma)_{u,B}$ such that S_b implements β. Then

$$(\forall a \in A^u)\ (F_A(S_b)(a) = [\![\beta]\!]_A(a)) \qquad (46)$$

Since $|B|=1$, it must be for some symbol 'b' that $id(S_b)=(b)$. Informally, it must be that the header of S_b is of the form:

function $y = b(x_1,\ldots,x_n)$: $\cdots$

where $x=(x_1,\ldots,x_n) \in Var_u$ (thus $|u|=n$), $y \in Var_B$, and '$\cdots$' is the body of S_b (possibly preceded by a local group).

Now define $G_1=S_b$. Then since $S_b \in FPIT(\Sigma)$ we have $G_1 \in FG(\Sigma)$ with $sig(G_1)=(\Sigma,b)$. This means that $sig(G_1)$ is the signature defined by

$$sig(G_1) = \langle sig(G_1)_{w,s} : w \in S^*, s \in S \rangle$$

where for each $w \in S^*$ and $s \in S$,

$$sig(G_1)_{w,s} = \begin{cases} \Sigma_{w,s} \cup \{ b \} & \text{if } w = u \text{ and } s = \mathrm{B} \\ \Sigma_{w,s} & \text{otherwise} \end{cases}$$

Also, $A_1 = Alg_A(G_1)$ is defined by

$$A_1 = (A, F_A(S_b))$$

and thus the interpretation of b in A_1 is

$$b^{A_1} = F_A(S_b) = [\![\beta]\!]_A \tag{47}$$

(using (46)).

Now suppose that there exists some function program $S_g \in \mathrm{FPIT}(\Sigma)_{u,v}$ which implements α_1. Then

$$(\forall a \in A^u) \ (F_A(S_g)(a) = [\![\alpha_1]\!]_A(a)) \tag{48}$$

Also, if $|v| = m$ it must be that $id(S_g) = (g_1, \ldots, g_m)$ for some function identifiers $g_1, \ldots, g_m$. That is, informally, S_g must be of the form

$$S_g = \textbf{function } y_1, \ldots, y_m = g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n) : \cdots$$

where $y = (y_1, \ldots, y_m) \in Var_v$, $x = (x_1, \ldots, x_n) \in Var_u$, and '$\cdots$' is the body of S_g (possibly preceded by a local group).

Now define $G_2 = G_1; S_g$. If we assume that $all(S_g) \cap sig(G_1) = \emptyset$ then we have $S_g \in \mathrm{FG}(sig(G_1))$ by Lemma 6.3.4, and so $G_2 \in \mathrm{FG}(\Sigma)$.

Now, $sig(G_2)$ is defined by $sig(G_2) = (sig(G_1), id(S_g))$. To be precise, $sig(G_2)$ is defined to be the extension of $sig(G_1)$ obtained by adding g_i to $sig(G_1)_{u,v_i}$ for $i = 1, \ldots, m$.

Additionally, $A_2 = Alg_A(G_2)$ is defined by

$$A_2 = (A_1, F_A(S_g))$$

Thus for $i = 1, \ldots, m$, the interpretation of g_i in A_2 is the ith coordinate of $F_A(S_g)$; that is, for $i = 1, \ldots, m$,

$$g_i^{A_2} = F_A(S_g)_i \tag{49}$$

Notice that the interpretation of $b \in sig(G_2)_{u,\mathrm{B}}$ in A_2 remains what it was in A_1 viz

$$b^{A_2} = b^{A_1} = [\![\beta]\!]_A \tag{50}$$

(using (47)).

Finally suppose that there exists some function program $S_h \in \mathrm{FPIT}(\Sigma)_{u,v}$ such that S_h implements α_2. Then

$$(\forall a \in A^u) \ (F_A(S_h)(a) = [\![\alpha_2]\!]_A(a)) \tag{51}$$

Also, since $|v| = m$ it must be that $id(S_h) = (h_1, \ldots, h_m)$ for some function identifiers $h_1, \ldots, h_m$.

Now define $G_3 = G_2; S_h$. If we assume that $all(S_h) \cap sig(G_2) = \emptyset$ then we have $S_h \in \mathrm{FG}(sig(G_2))$ by Lemma 6.3.4, and so $G_3 \in \mathrm{FG}(\Sigma)$.

Now, $sig(G_3)$ is defined by $sig(G_3) = (sig(G_2), id(S_h))$. To be precise, $sig(G_3)$ is defined to be the extension of $sig(G_2)$ obtained by adding h_i to $sig(G_2)_{u,v_i}$ for $i = 1, \ldots, m$.

Additionally, $A_3 = Alg_A(G_3)$ is defined by

$$A_3 = (A_2, F_A(S_h))$$

Thus for $i = 1,...,m$, the interpretation of h_i in A_3 is the ith coordinate of $F_A(S_h)$; that is, for $i = 1,...,m$,

$$h_i^{A_3} = F_A(S_h)_i \tag{52}$$

Notice that the interpretation of $b \in sig(G_3)_{u,\beta}$ in A_3 remains what it was in A_2 viz

$$b^{A_3} = b^{A_2} = [\![\beta]\!]_A \tag{53}$$

(using (50)). Similarly, for $i = 1,...,m$, the interpretation of each $g_i \in sig(G_3)_{u,v_i}$ in A_3 remains what it was in A_2 viz

$$g_i^{A_3} = g_i^{A_2} = F_A(S_g)_i \tag{54}$$

(using (49)).

We can now write down a function program $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ which implements α:

$$S = \textbf{function } y_1,...,y_m = f_1(x_1,...,x_n),...,f_m(x_1,...,x_n) : G : S_o$$

where:

$y = (y_1,...,y_m) \in Var_v$;

$x = (x_1,...,x_n) \in Var_u$;

$G = G_3 = S_b ; S_g ; S_h$, and

$S_o =$ **if** $b(x_1,...,x_n)$ **then** S_1 **else** S_2 **fi**, where:

$S_1 = y_1,...,y_m := g_1(x_1,...,x_n),...,g_m(x_1,...,x_n)$, and

$S_2 = y_1,...,y_m := h_1(x_1,...,x_n),...,h_m(x_1,...,x_n)$.

Let us first explain why $S \in \mathrm{FPIT}(\Sigma)_{u,v}$; afterwards we will prove that S implements α.

If S is to be admitted to FPIT(Σ) at all, then it must be as a nested function program. On inspecting the conditions for membership as a nested function program (see Definition 6.3.1, clause P2, subclauses (a)-(g)), it is clear that all we really need to establish is that $(S_o, x, y) \in \mathrm{PITIO}(sig(G))$; however, this should be obvious: certainly (S_o, x, y) is an i/o-program over some signature, and since S_o involves the operator symbols b, $g_1,...,g_m$, and $h_1,...,h_m$, all of which are members of $sig(G)$, it is obvious that (S_o, x, y) is an i/o-program over $sig(G)$.

Notice how straightforward the program S_o is: since $[\![\beta]\!]_A$ and (the coordinates of) $[\![\alpha_1]\!]_A$ and $[\![\alpha_2]\!]_A$ are available as named basic operations, implementing α is trivial.

Now let us prove that S implements α:

First notice that since $(S_o, x, y) \in \mathrm{PITIO}(sig(G))$, we have that $F_{A_o}(S_o, x, y)$ is a well-defined function (using Lemma 6.2.11 with $\Sigma = sig(G)$).

Thus for any $a \in A^u$ we have:

$$F_A(S)(a) = F_{A_o}(S_o, x, y)(a)$$

$$= F_{A_3}(S_0, x, y)(a)$$

(since $A_G = A_3$)

$$= \pi_A(y)(M_{A_3}(S_0)(\rho)) \tag{55}$$

where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$ (by definition of $F_A(S, In, Out)$ for arbitrary A and (S, In, Out)).

(Recall the discussion following Definition 6.3.2: in (55) we do not write '$\pi_{A_3}(...)$', nor do we say '$\rho \in States(A_3)$' as might be expected from the definition of the functional meaning of an i/o-program (see Definition 6.2.10). As we have said before, the reason for this is that the only difference between A and A_3 is that the latter has more basic operations; that is, A and A_3 have the same carriers, and thus $States(A) = States(A_3)$ and so $\pi_A = \pi_{A_3}$.)

Since S_0 is a conditional statement, from (55) we have for any $a \in A^u$, and any $\rho \in States(A)$ such that $\pi_A(x)(\rho) = a$:

$$F_A(S)(a) = \begin{cases} \pi_A(y)(\rho_1) & \text{if } E_{A_0}(b(x_1, ..., x_n))(\rho) = tt \\ \pi_A(y)(\rho_2) & \text{if } E_{A_0}(b(x_1, ..., x_n))(\rho) = ff \end{cases} \tag{56}$$

where $\rho_i = M_{A_0}(S_i)(\rho)$ for $i = 1,2$.

Now, using the definition of $E_A(e)$ for a general Σ-algebra A and expression e over Σ, we calculate as follows:

$$E_{A_0}(b(x_1, ..., x_n))(\rho) = b^{A_0}(\rho(x_1), ..., \rho(x_n))$$

$$= b^{A_3}(\rho(x_1), ..., \rho(x_n))$$

$$= [\![\beta]\!]_A(\rho(x_1), ..., \rho(x_n))$$

(from (53))

$$= [\![\beta]\!]_A(\pi_A(x)(\rho))$$

$$= [\![\beta]\!]_A(a)$$

Substituting for $\rho_1(y_i)$ for $i = 1, ..., m$ in (56) now yields

$$F_A(S)(a) = \begin{cases} \pi_A(y)(\rho_1) & \text{if } [\![\beta]\!]_A(a) = tt \\ \pi_A(y)(\rho_2) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases} \tag{57}$$

Now suppose $[\![\beta]\!]_A(a) = tt$. Then from (57) we have

$$F_A(S)(a) = \pi_A(y)(\rho_1)$$

$$= (\rho_1(y_1), ..., \rho_1(y_m)) \tag{58}$$

Now, since S_1 is a multiple assignment statement we have

$$\rho_1 = M_{A_0}(S_1)(\rho)$$

$$= \rho\{E_{A_0}(g_1(x_1, ..., x_n))(\rho)/y_1\}\{\cdots\}\{E_{A_0}(g_m(x_1, ..., x_n))(\rho)/y_m\}$$

Thus for $i = 1, ..., m$, we have

$$\rho_1(y_i) = E_{A_0}(g_i(x_1, ..., x_n))(\rho)$$

$$= E_{A_s}(g_i(x_1,\ldots,x_n))(\rho)$$

$$= g_i^{A_s}(\rho(x_1),\ldots,\rho(x_n))$$

$$= F_A(S_s)_i(\rho(x_1),\ldots,\rho(x_n))$$

(using (54))

$$= F_A(S_s)_i(\pi_A(x)(\rho))$$

$$= F_A(S_s)_i(a)$$

Substituting for $E_{A_\sigma}(g_i(x_1,\ldots,x_n))(\rho)$ in (58) yields:

$$F_A(S)(a) = (F_A(S_s)_1(a),\ldots,F_A(S_s)_m(a))$$

$$= F_A(S_s)(a)$$

(since $F_A(S_s)_1,\ldots,F_A(S_s)_m$ are the coordinate of $F_A(S_s)$)

$$= [\![\alpha_1]\!]_A(a)$$

(since S_s implements α; see (48)).

We have now proved that

$$[\![\beta]\!]_A(a) = tt \Rightarrow F_A(S)(a) = [\![\alpha_1]\!]_A(a) \tag{59}$$

In a similar way it is easy to show that

$$[\![\beta]\!]_A(a) = ff \Rightarrow F_A(S)(a) = [\![\alpha_2]\!]_A(a) \tag{60}$$

From (59) and (60) we now have for any $a \in A^u$:

$$F_A(S)(a) = \begin{cases} [\![\alpha_1]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ [\![\alpha]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

$$= [\![DC(\beta,\alpha_1,\alpha_2)]\!]_A(a)$$

$$= [\![\alpha]\!]_A(a)$$

Thus S implements α as claimed.

6.4 COMPILATION.

We now turn our attention to the general implementation problem of showing that we can implement every PR-scheme in FPIT; when we can do this we can simulate arbitrary synchronous algorithms of course.

To solve our implementation problem we will define a compiler $c^{PR}: PR(\Sigma) \longrightarrow FPIT(\Sigma)$ with the intention that for every $\alpha \in PR(\Sigma)$, $c^{PR}(\alpha) \in FPIT(\Sigma)$ implements α. Our strategy for compiling schema has been exemplified in Section 6.3.6: implementing basis schema is easy, and implementing composite schema is easy once the subschema have been compiled into function programs that determine an algebra in which the subschema are available as named basic operations.

In Chapter 7 we will prove that $c^{PR}(\alpha)$ implements α and we will additionally prove that the compiler preserves algorithm complexity.

6.4.1 Definition.

We define c^{PR} to be the $S^+\times S^+$-indexed family

$$c^{PR} = \langle c^{PR}_{u,v} : u,v \in S^+ \rangle$$

of mappings $c^{PR}_{u,v} : \mathrm{PR}(\Sigma)_{u,v} \longrightarrow \mathrm{FPIT}(\Sigma)_{u,v}$; we call c^{PR} the *PR-FPIT compiler.* For each $u,v \in S^+$, $c^{PR}_{u,v}$ (ambiguously denoted c^{PR}) is defined uniformly in u and v and by induction on the structural complexity of arguments α as described after the following

Notes.

(1) For each $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ we will first construct $c^{PR}(\alpha)$ and then prove it is a well-defined member of $\mathrm{FPIT}(\Sigma)_{u,v}$.

(2) In the following definition we will continually define $c^{PR}(\alpha)$ to be a function program whose header is of the form:

$$\textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : \cdots$$

We will always assume that such headers satisfy the trivial syntactic constraints on membership of FPIT(Σ); that is, we assume $\{x_1,\ldots,x_n\}$ and $\{y_1,\ldots,y_m\}$ are mutually exclusive sets of distinct variables.

(3) We further assume that $f_1,\ldots,f_m$ are always chosen to be *new* identifiers; that is, function identifiers not occurring in Σ nor in any *previous* part of the construction of $c^{PR}(\alpha)$. (In particular, if S above has a local group G, then we assume $\{f_1,\ldots,f_m\} \cap all(G) = \varnothing$.)

Basis Cases.

(i) *Constant Functions.* Suppose $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$. Then $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,\ldots,x_n) : S_0$$

where:

$y \in Var_s$;

$x = (x_1,\ldots,x_n) \in Var_w$, and

$S_0 = y := c$.

Well-Definedness. Given notes (2) and (3) above, and the definition of a simple function program, it should be clear that $(S_0,x,y) \in \mathrm{PITIO}(\Sigma)_{w,s}$ and thus that $c^{PR}(\alpha) \in \mathrm{FPIT}(\Sigma)_{w,s}$ as required.

(ii) *Algebraic Operations.* Suppose $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$. Then $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,\ldots,x_n) : S_o$$

where:

$y \in Var_s$;

$x = (x_1,\ldots,x_n) \in Var_w$, and

$S_o = y := \sigma(x_1,\ldots,x_n)$.

Well-Definedness. Again, it should be clear without further comment that $(S_o,x,y) \in \text{PITIO}(\Sigma)$ and thus that $c^{PR}(\alpha) \in \text{FPIT}(\Sigma)_{w,s}$.

(iii) *Projection Functions.* Suppose $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \le i \le n = |w|$. Then $\alpha \in \text{PR}(\Sigma)_{w,w_i}$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,\ldots,x_n) : S_o.$$

where:

$y \in Var_{w_i}$;

$x = (x_1,\ldots,x_n) \in Var_w$, and

$S_o = y := x_i$.

Well-Definedness. Again, it is easy to see that $(S_o,x,y) \in \text{PITIO}(\Sigma)$ and thus that $c^{PR}(\alpha) \in \text{FPIT}(\Sigma)_{w,w_i}$.

Induction. Let $\alpha \in \text{PR}(\Sigma)_{u,v}$ for some $u,v \in S$. Suppose that α has the property that for every $u',v' \in S^+$ and for every $\alpha' \in \text{PR}(\Sigma)_{u',v'}$ of less structural complexity than α, we have defined $c^{PR}(\alpha')$ and established that $c^{PR}(\alpha') \in \text{FPIT}(\Sigma)_{u',v'}$.

We now define $c^{PR}(\alpha)$ according to the four following possible cases:

(iv) *Definition-by-Cases.* Suppose α is of the form $\alpha = \text{DC}(\beta,\alpha_1,\alpha_2)$. Then for some $u,v \in S^+$, we have $\beta \in \text{PR}(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in \text{PR}(\Sigma)_{u,v}$.

Since β, α_1, and α_2 are all of less structural complexity than α, we can assume by the induction hypothesis that

$$S_b = c^{PR}(\beta) \in \text{FPIT}(\Sigma)_{u,B}$$
$$S_g = c^{PR}(\alpha_1) \in \text{FPIT}(\Sigma)_{u,v}$$

and

$$S_h = c^{PR}(\alpha_2) \in \text{FPIT}(\Sigma)_{u,v}$$

Now, since $|B| = 1$, it must be that for some symbol b, we have $id(S_b) = (b)$. Also, if $|v| = m$ say, then for some symbols $g_1,\ldots,g_m$ and $h_1,\ldots,h_m$, we have $id(S_g) = (g_1,\ldots,g_m)$ and $id(S_h) = (h_1,\ldots,h_m)$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_b ; S_g ; S_h$, and

$S_o = \textbf{if } b(x_1,\ldots,x_n) \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$, where:

$S_1 = y_1,\ldots,y_m := g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)$, and

$S_2 = y_1,\ldots,y_m := h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n)$.

Well-Definedness. It should clear that the second example in Section 6.3.6 was tailored to this stage in the compilation: $c^{PR}(\alpha)$ is exactly the function program S of the example. Note that the only assumptions we made in establishing that $S = c^{PR}(\alpha) \in \mathrm{PR}(\Sigma)$ were ones concerning the function identifiers in S_g and S_h. Specifically, we assumed that

$$all(S_g) \cap sig(G_1) = \emptyset$$

and

$$all(S_h) \cap sig(G_2) = \emptyset$$

where $G_1 = S_b$ and $G_2 = G_1;S_g$.

In this case of the compilation, the intention is that we *first* compile β, then α_1, and finally α_2: by note (3) above, we have

$$all(c^{PR}(\alpha_1)) \cap sig(G_1) = \emptyset$$

and

$$all(c^{PR}(\alpha_2)) \cap sig(G_2) = \emptyset$$

when $G_1 = c^{PR}(\beta)$ and $G_2 = G_1;c^{PR}(\alpha_1)$. These two conditions are sufficient to guarantee $c^{PR}(\alpha) \in \mathrm{FPIT}(\Sigma)_{u,v}$ (by Lemma 6.3.4 twice).

(v) *Vectorisation.* Suppose α is of the form $\alpha = <\alpha_1,\ldots,\alpha_m>$ for some $m \geq 1$.

Then for some $u \in S^+$ and $v \in S^m$, $\alpha_i \in \mathrm{PR}(\Sigma)_{u,v_i}$ for $i = 1,\ldots,m$.

Since α_i is of less structural complexity than α for $i = 1,\ldots,m$, we can assume by the induction hypothesis that

$$S_i = c^{PR}(\alpha_i) \in \mathrm{FPIT}(\Sigma)_{u,v_i}$$

for $i = 1,\ldots,m$.

Also, since $|v_i| = 1$ for $i = 1,\ldots,m$, it must be that for some symbols $g_1,\ldots,g_m$ we have $id(S_i) = (g_i)$ for $i = 1,\ldots,m$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_1 ; \cdots ; S_m$, and

$S_o = y_1,\ldots,y_m := g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)$.

Well-Definedness. Assume for the moment that $G \in \mathrm{FG}(\Sigma)$. On inspecting the conditions for membership as a nested function program (see Definition 6.3.1, clause P2, sub-clauses (a)-(g)), it is clear that given note (2) above, we only really need to establish that $(S_o,x,y) \in \mathrm{PITIO}(sig(G))$; however, this should be obvious: certainly (S_o,x,y) is an i/o-program over some signature, and since S_o involves the operator symbols $g_1,\ldots,g_m$ all of which are members of $sig(G)$, it is clear that (S_o,x,y) is an i/o-program over $sig(G)$.

To show $G \in \mathrm{FG}(\Sigma)$, by Lemma 6.3.5 it is sufficient to show

$$all(S_{i+1}) \cap sig(G_i) \tag{61}$$

for $i = 1,\ldots,m-1$, where $G_1 = S_1$ and $G_{k+1} = G_k;S_{k+1}$ for $k = 1,\ldots,m-1$.

In this case of the compilation the intention is that we compile $\alpha_1,\ldots,\alpha_m$ in order; α_1 first, and α_m last. Note (3) is sufficient to guarantee that (61) holds for $i = 1,\ldots,m$.

(vi) *Composition.* Suppose α is of the form $\alpha = \alpha_2 \circ \alpha_1$.

Then for some $u,v,w \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$.

Since α_1 and α_2 are of less structural complexity than α, we can assume by the induction hypothesis that

$$S_1 = c^{PR}(\alpha_1) \in \mathrm{FPIT}(\Sigma)_{u,w}$$

and

$$S_2 = c^{PR}(\alpha_2) \in \mathrm{FPIT}(\Sigma)_{w,v}$$

Also, since $|w| = k$ for some $k > 0$, it must be that for some symbols $g_1,\ldots,g_k$ we have $id(S_1) = (g_1,\ldots,g_k)$. Additionally, since $|v| = m > 0$, it must be that for some symbols $h_1,\ldots,h_m$ we have $id(S_2) = (h_1,\ldots,h_m)$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_1 ; S_2$, and

$S_o = y_1,...,y_m := e_1,...,e_m$, where:

$e_i = h_i(g_1(x_1,...,x_n),...,g_k(x_1,...,x_n))$ for $i = 1,...,m$.

Well-Definedness. Similar to the two previous induction cases above, it should be clear that $c^{PR}(\alpha) \in \text{FPIT}(\Sigma)_{u,v}$ if we first compile α_1 and then α_2.

(vii) *Primitive Recursion.* Suppose α is of the form $\alpha = *(\alpha_1,\alpha_2)$.

Then for some $u,v \in S^+$, $\alpha_1 \in \text{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \text{PR}(\Sigma)_{\tau u v,v}$.

Since α_1 and α_2 are of less structural complexity than α, we can assume by the induction hypothesis that

$$S_g = c^{PR}(\alpha_1) \in \text{FPIT}(\Sigma)_{u,v}$$

and

$$S_h = c^{PR}(\alpha_2) \in \text{FPIT}(\Sigma)_{\tau u v,v}$$

Also, since $|v| = m$, it must be that for some symbols $g_1,...,g_m$ and $h_1,...,h_m$ we have $id(S_g) = (g_1,...,g_m)$ and $id(S_h) = (h_1,...,h_m)$.

Compilation. Here we define $c^{PR}(\alpha)$ by

$$c^{PR}(\alpha) = \textbf{function } y_1,...,y_m = f_1(p,x_1,...,x_n),...,f_m(p,x_1,...,x_n) : G : S_o$$

where:

$y = (y_1,...,y_m) \in Var_v$;

$x = (p,x_1,...,x_n) \in Var_{\tau u}$;

$G = S_g$; S_h, and

$S_o = S_1$; do p times S_2 od, where:

$S_1 = z,y_1,...,y_m := \text{zero},g_1(x_1,...,x_n),...,g_m(x_1,...,x_n)$, and

$S_2 = z,y_1,...,y_m := \text{succ}(z),e_1,...,e_m$, where:

$z \in Var_\tau$, and

$e_i = h_i(z,x_1,...,x_n,y_1,...,y_m)$ for $i = 1,...,m$.

Well-Definedness. Similar to the three previous induction cases above, it should be clear that $c^{PR}(\alpha) \in \text{FPIT}(\Sigma)_{u,v}$ if we first compile α_1 and then α_2. □

6.4.2 Example.

As with any other compiler, c^{PR} produces a vast quantity of unreadable output when applied to all but the simplest input. We would have liked to include $c^{PR}(\alpha_{OE})$ and $c^{PR}(\gamma_{OE})$ (see Notation 3.4.5) to compare them with S_{OE} (see Examples 6.1.8) and the reader's solution to Exercise 2.4.5(3) respectively. Unfortunately however, we do not have the space to do so since $c^{PR}(\alpha_{OE})$ and $c^{PR}(\gamma_{OE})$ are each almost 400 lines long (for $n = 4$).

Let us content ourselves with a much simpler example. Consider the scheme $\alpha = +\circ\langle U_2^{NN}, U_1^{NN}\rangle$ over the natural numbers $\mathbb{N}$. Since α is defined by composition, c^{PR} will first compile $\alpha_1 = \langle U_2^{NN}, U_1^{NN}\rangle$, then $\alpha_2 = +$, and then put the resulting function programs together in a function group over which the composition α can be implemented. Considering the compilation of α_1, since this scheme is an instance of vectorisation, c^{PR} will first compile U_2^{NN} then U_1^{NN}, and then put these together in a function group over which β can be implemented. Thus the order in which the schema comprising α are compiled is first U_2^{NN}, then U_1^{NN}, then α_1, then α_2, and finally α itself.

A function program to implement α is obtained by routine application of c^{PR} according to Definition 6.4.1 as follows:

First U_2^{NN} is compiled, giving the function program

$$\textbf{function } y_1 = f_1(x_1, x_2) :$$
$$y_1 := x_2$$

Then U_1^{NN} is compiled giving

$$\textbf{function } y_2 = f_2(x_3, x_4) :$$
$$y_2 := x_3$$

These two function programs are now used to define a function program for $\alpha_1 = \langle U_2^{NN}, U_1^{NN}\rangle$:

$$\textbf{function } y_3, y_4 = f_3(x_5, x_6), f_4(x_5, x_6) :$$
$$\textbf{function } y_1 = f_1(x_1, x_2) :$$
$$y_1 := x_2 ;$$
$$\textbf{function } y_2 = f_2(x_3, x_4) :$$
$$y_2 := x_3 :$$
$$y_3, y_4 := f_1(x_5, x_6), f_2(x_5, x_6)$$

Now $\alpha_2 = +$ is compiled:

$$\textbf{function } y_5 = f_5(x_7, x_8) :$$
$$y_5 := +(x_7, x_8)$$

Finally $c^{PR}(\alpha_1)$ and $c^{PR}(\alpha_2)$ are used to provide and implementation of α:

$$\textbf{function } y_6 = f_6(x_9, x_{10}) :$$
$$\textbf{function } y_3, y_4 = f_3(x_5, x_6), f_4(x_5, x_6) :$$
$$\textbf{function } y_1 = f_1(x_1, x_2) :$$
$$y_1 := x_2 ;$$
$$\textbf{function } y_2 = f_2(x_3, x_4) :$$
$$y_2 := x_3 :$$
$$y_3, y_4 := f_1(x_5, x_6), f_2(x_5, x_6) ;$$
$$\textbf{function } y_5 = f_5(x_7, x_8) :$$
$$y_5 := +(x_7, x_8) :$$
$$y_6 := f_5(f_3(x_9, x_{10}), f_4(x_9, x_{10}))$$

We leave it to the reader to prove that $c^{PR}(\alpha)$ implements α and that $c^{PR}(\alpha)$ and α have the same time-complexity (to within a constant factor).

6.5 SOURCES.

The search for an appropriate implementation language for PR and the design of FPIT as such a language is the result of joint work between J. V. Tucker and myself. The detailed formal definition of the language, in particular the performance part of the definition, is my own work.

The multiple assignment statement has appeared on many occasions in the literature on programming language constructs. Since what seems to be its original appearance in Barron et al[1963], the multiple assignment statement has been used by E. W. Dijkstra and C. A. R. Hoare and studied by D. Gries; see Dijkstra[1976], Hoare[1985], and Gries[1978] respectively.

Independently of our work, the multiple assignment statment has been used by K. M. Chandy and J. Misra (see Chandy and Misra[1986a]) to define one or two systolic algorithms in the context of their general theory of algorithm development (see Chandy and Misra[1985] and Chandy and Misra[1986b]).

The study of i/o-triples and their functionality originates in Tucker and Zucker[1987] (work of 1979): Definition 6.2.3 is based the definition found in Section 4.3.2 of Tucker and Zucker[1987]. The functionality of i/o-triples has been further studied by C. A. Jervis in his work (Jervis[1988]) on the specification and implementation of abstract data types by means of while-programs computing over single-sorted abstract structures. We are indebted to Mr. Jervis for his original technical work on which Definitions 6.2.6 and 6.2.7 and Theorem 6.2.8 are based.

The language FPIT and a prototype compiler from PR into FPIT have been implemented on a VAX 11/780 by our colleague A. R. Martin as part of the DEDEKIND project (see Section 1.3).

CHAPTER 7
COMPUTATIONAL EQUIVALENCE

In this chapter we will prove that PR and FPIT are equivalent representation systems for synchronous algorithms as promised in Chapter 6. Our first task is to prove that the transformation c^{PR} of Section 6.4.1 preserves both the meaning and performance of PR schema; this will establish that every synchronous algorithm can be simulated in FPIT, and thus as a specification language for synchronous algorithms, FPIT is no weaker than PR. Our second task is to prove that FPIT is no stronger than PR as a specification language (recall that synchronous algorithms may be simulated directly in FPIT without using PR as an intermediate stage). To do this we will define a second transformation c^{FP}, which transforms arbitrary function programs into PR schema, and which we will prove both meaning and performance preserving.

We think of PR and FPIT as programming languages, and as transformations between these languages, we think of c^{PR} and c^{FP} as *compilers*. In Section 7.1 we develop a theoretical framework in which we may analyse compilers and *compiler correctness*; this framework is based on ideas first formulated in Morris[1973] and subsequently developed and extended in Thatcher, Wagner, and Wright[1980]. We will extend this framework to account for performance matters: we enrich our theory of compilation by considering when a compiler can be said to be performance preserving.

After proving c^{PR} is a correct and performance preserving compiler in Section 7.2, we will turn our attention to c^{FP}. In Section 7.3, after discussing some preliminary tools, we explain our strategy for compiling arbitrary function programs into PR and then define c^{FP}. In Section 7.4 we prove that c^{FP} is a correct and performance preserving compiler.

7.1 COMPILATION.

The theory of compilers and their correctness begins with the theory of programming languages. One may summarise a formal account of a programming language in the following general way: to formalise a given language, one gives a syntax L for the language, a semantic space S containing at least all possible behaviours of programs $\alpha \in L$, and a meaning function $M : L \longrightarrow S$ with the intention that for each $\alpha \in L$, $M(\alpha) \in S$ is the behaviour of α.

Given two languages with syntaxes L_1 and L_2 respectively, a compiler from L_1 into L_2 is intuitively just a mapping $c : L_1 \longrightarrow L_2$. When compiling a program $\alpha \in L_1$, we call α the *source program* and the image $c(\alpha) \in L_2$ the *object program*. Now let S_1 and S_2 be semantic spaces for L_1 and L_2 respectively and let $M_1 : L_1 \longrightarrow S_1$ and $M_2 : L_2 \longrightarrow S_2$ be meaning functions. Intuitively, a compiler $c : L_1 \longrightarrow L_2$ should be said to be correct if for every $\alpha \in L_1$, α and $c(\alpha)$ have the same behaviour: the behaviour of α is $M_1(\alpha)$, and the behaviour of $c(\alpha)$ is $M_2(c(\alpha))$, and thus to say that α and $c(\alpha)$ have the 'same' behaviour is to say that $M_2(c(\alpha))$ is equivalent to $M_1(\alpha)$ in some sense. For example, if $S_1 = S_2$, then c could be said to be correct if $M_2(c(\alpha)) = M_1(\alpha)$ for every source program α. More

generally, one has $S_1 \neq S_2$ and the behaviour of an object program $c(\alpha)$ is related to the behaviour of α via a function $\pi: S_1 \longrightarrow S_2$ which we think of as a representation or coding of L_1-behaviours over L_2-behaviours; we say $s_2 \in S_2$ is equivalent to $s_1 \in S_1$ if $\pi(s_1) = s_2$.

Morris[1973] and Thatcher, Wagner, and Wright[1980] offer the following definition: a compiler $c: L_1 \longrightarrow L_2$ is *correct* if the following diagram commutes for every $\alpha \in L_1$:

$$\begin{array}{ccc} L_1 & \xrightarrow{\quad M_1 \quad} & S_1 \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle \pi} \\ L_2 & \xrightarrow[\quad M_2 \quad]{} & S_2 \end{array}$$

That is, c is correct if

$$(\forall \alpha \in L_1)\ \ (M_2(c(\alpha)) = \pi(M_1(\alpha)))$$ □

Note that when $S_1 = S_2$ and π is the identity function (as will be the case in this chapter, but not in the next), the right-hand side of the commutative diagram collapses to a single point.

In this thesis we have insisted that a formal account of a language is incomplete without a complexity theory for the language: we have formalised the complexity of programs in all of our languages by providing a space C of complexity functions and a performance estimator $\lambda: L \longrightarrow C$ with the intention that for each $\alpha \in L$ $\lambda(\alpha)$ is the length of computation function for α. The addition of a complexity theory to formal accounts of languages allows us to consider compiler performance preservation characteristics of compilers in the following way: if L_1 and L_2 have complexity spaces C_1 and C_2 and performance estimators λ_1 and λ_2 respectively, then the complexity of any $\alpha \in L_1$ is $\lambda_1(\alpha)$ and the complexity of $c(\alpha)$ is $\lambda_2(c(\alpha))$; similar to compiler correctness, we can say that c is performance preserving if in some sense the complexity of $c(\alpha)$ is equivalent to that of α. Equality between $\lambda_2(c(\alpha))$ and $\lambda_1(\alpha)$ is too strong a condition in general for a performance preservation condition; instead we will say that c is performance preserving if $\lambda_2(c(\alpha)) \approx \lambda_1(\alpha)$ where '$\approx$' is the relation defined in Section 3.2.3.

Our current objective is to verify c^{PR} and c^{FP} (yet to be defined). We will adopt the theoretical framework of compiler correctness given by Morris[1973] and Thatcher, Wagner, and Wright[1980], but we will tailor it to our particular needs and extend it to cover compiler performance preservation. The verification of c^{PR} is straightforward and could be given now without further ado. However, the compiler c^{FP} is defined in terms of four auxiliary compilers which, like c^{PR} and c^{FP}, are compilers between functional languages. Below, after some preliminary definitions, we will customise definitions of 'compiler' and 'compiler correctness' to the case of such languages; these definitions serve to unify our theoretical treatment of the six compilers found in this chapter.

Definitions.

(i) Let X and Y be $S^+\times S^+$-indexed families of sets:

$$X = \langle X_{u,v} : u,v \in S^+ \rangle, \quad Y = \langle Y_{u,v} : u,v \in S^+ \rangle$$

Also let f be an $S^+\times S^+$-indexed family of mappings:

$$f = \langle f_{u,v} : u,v \in S^+ \rangle$$

Then we say f is a *word-indexed function* if $f_{u,v} : X_{u,v} \longrightarrow Y_{u,v}$ for each $u,v \in S^+$; in symbols we write $f : X \longrightarrow Y$. (The indexing pair (u,v) appears as a superscript if it is more convenient.)

(ii) Let A be an S-indexed family of sets:

$$A = \langle A_s : s \in S \rangle$$

Then we define $FN(A)$ to be the collection of all functions on A as an $S^+\times S^+$-indexed family:

$$FN(A) = \langle FN(A)_{u,v} : u,v \in S^+ \rangle$$

where $FN(A)_{u,v} = [A^u \longrightarrow A^v]$ for each $u,v \in S^+$.

Definitions. Let Σ be an S-sorted signature, let A be a Σ-algebra and let P be a performance measure for A which is based on clock C.

(i) Let $L(\Sigma)$ be a language over Σ as a $S^+\times S^+$-indexed family of sets:

$$L(\Sigma) = \langle L(\Sigma)_{u,v} : u,v \in S^+ \rangle$$

Then we say $L(\Sigma)$ is a *functional language over* Σ.

(ii) Let $L(\Sigma)$ be a functional language over S-sorted signature Σ, and let $M_A : L(\Sigma) \longrightarrow FN(A)$. Then we say $L(\Sigma)$ has *functional semantics* M_A. (Note that by clause (i) of the preceding definitions this means that M_A must be a word-indexed function with $M_A^{u,v} : L(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow A^v]$ for each $u,v \in S^+$.)

(iii) Let $L(\Sigma)$ be a functional language over S-sorted signature Σ, and let λ_P be an S^+-indexed family:

$$\lambda_P = \langle \lambda_P^u : u \in S^+ \rangle$$

where for each $u \in S^+$,

$$\lambda_P^u : \bigcup_{v \in S^+} L(\Sigma)_{u,v} \longrightarrow [A^u \longrightarrow C^+]$$

Then we say $L(\Sigma)$ has *functional complexity theory* λ_P.

Exercises. Let Σ be a standard signature, let A be a standard Σ-algebra, and let P be a performance measure for A. Show

(i) PR(Σ) is a functional language with functional semantics $[\![.]\!]_A$ (as defined in Section 3.3.2) and functional complexity theory λ_P (as defined in Section 3.3.3).

(ii) PITIO(Σ) is a functional language with functional semantics F_A (as defined in Definition 6.2.10(i)) and functional complexity theory λ_P^{IO} (as defined in Definition 6.2.10(ii)).

(iii) FPIT(Σ) is a functional language with functional semantics F_A (as defined in Section 6.3.2) and functional complexity theory λ_P^{FP} (as defined in Section 6.3.3).

7.1.1 Definitions.

For $i = 1,2$, let Σ_i be an S-sorted signature, and let $L_i(\Sigma_i)$ be a functional language over Σ_i. Also let A_1 and A_2 be Σ_1- and Σ_2-algebras respectively, where A_1 and A_2 have the same carriers; that is, for each $s \in S$ let $(A_1)_s = (A_2)_s = A_s$ for some S-indexed family of sets $A = <A_s : s \in S>$ (so $FN(A_1)$ and $FN(A_2)$ are both equal to $FN(A)$). Additionally let P_1 and P_2 be performance measures for A_1 and A_2 respectively, where both P_1 and P_2 are based on the same clock C. Then we make the following definitions:

(i) A *compiler from* $L_1(\Sigma_1)$ *into* $L_2(\Sigma_2)$ is any mapping $c : L_1(\Sigma_1) \longrightarrow L_2(\Sigma_2)$. (Thus a compiler form $L_1(\Sigma_1)$ into $L_2(\Sigma_2)$ is any word-indexed family c of mappings with $c_{u,v} : L_1(\Sigma_1)_{u,v} \longrightarrow L_2(\Sigma_2)_{u,v}$ for each $u,v \in S^+$.)

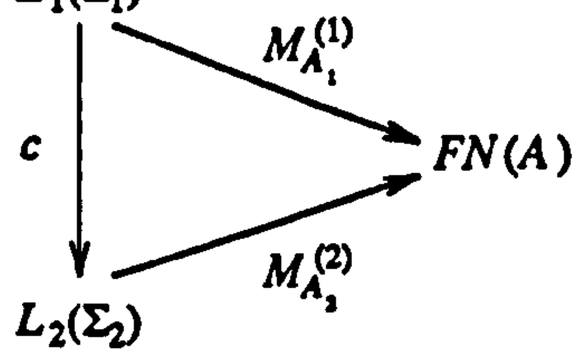
(ii) Suppose that $L_1(\Sigma_1)$ and $L_2(\Sigma_2)$ have functional semantics

$$M_{A_1}^{(1)} : L_1(\Sigma_1) \longrightarrow FN(A)$$

and

$$M_{A_2}^{(2)} : L_2(\Sigma_2) \longrightarrow FN(A)$$

respectively. Then we say a compiler c from $L_1(\Sigma_1)$ into $L_2(\Sigma_2)$ is *correct with respect to* $M_{A_1}^{(1)}$ and $M_{A_2}^{(2)}$ if the following diagram commutes for every $\alpha \in L_1(\Sigma_1)$:



That is, if

$$(\forall \alpha \in L_1(\Sigma_1)_{u,v}) \ \ ((M_{A_2}^{(2)})_{u,v}(c_{u,v}(\alpha)) = (M_{A_1}^{(1)})_{u,v}(\alpha))$$

for every $u,v \in S^+$.

(iii) Suppose $L_1(\Sigma_1)$ and $L_2(\Sigma_2)$ have a functional complexity theories $\lambda_{P_1}^{(1)}$ and $\lambda_{P_2}^{(2)}$ respectively. Then we say a compiler c from $L_1(\Sigma_1)$ into $L_2(\Sigma_2)$ is *performance preserving with respect to* $\lambda_{P_1}^{(1)}$ and $\lambda_{P_2}^{(2)}$ if

$$(\forall \alpha \in L_1(\Sigma_1)) \ \ (\lambda_{P_2}^{(2)}(c(\alpha)) \approx \lambda_{P_1}^{(1)}(\alpha)) \qquad \square$$

Clause (iii) of this last definition requires further comment. Suppose $\alpha \in L_1(\Sigma_1)_{u,v}$ for some $u,v \in S^+$. Then from Section 3.2.3 we have that

$$\lambda_{P_2}^{(2)}(c(\alpha)) \approx \lambda_{P_1}^{(1)}(\alpha) \qquad (1)$$

iff for some (integer) constants $\mu \geq 1$ and $\nu \geq 1$ we have

$$(\forall a \in A^u) \ \ (\lambda_{P_2}^{(2)}(c(\alpha))(a) \leq \mu . \lambda_{P_1}^{(1)}(\alpha)(a)) \qquad (2)$$

and

$$(\forall a \in A^u)\ (\lambda_{p_1}^{(1)}(\alpha)(a) \leq \nu . \lambda_{p_2}^{(2)}(c(\alpha))(a)) \tag{3}$$

The first part of (1), namely (2), is of more interest than the second part (3): in compiling a program α to $c(\alpha)$, it is usually the case that $c(\alpha)$ has more instructions or commands than α and so intuitively we expect the complexity of α to be less than of $c(\alpha)$; that is, we anticipate any compiler to have $\nu = 1$. On the other hand, it is of interest to know to a bound on the complexity of $c(\alpha)$, especially when $c(\alpha)$ is a program which implements or simulates α; having devised an algorithm α and established that it has satisfactory performance characteristics, (2) tells us a bound on the execution time of the implementation without us having to calculate the complexity of the implementation explicitly.

Below, we begin the proof of computational equivalence of PR and FPIT. First, here are some preliminary notes:

Notes.

(i) In the forthcoming technical work, sort sets, signatures, Σ-algebras, and performance measures will always be *standard*. In particular, when we quantify over signatures Σ with phrases such as '*Let* Σ *be any signature* ...', we always mean any standard signature Σ. (Similar remarks apply to sort sets, algebras, and performance measures as well.)

(ii) Recall the complexity function λ_p for PR (as defined in Definition 3.3.3). To make the notation uniform we will subsequently use the notation 'λ_p^{PR}' instead of 'λ_p'.

7.2 VERIFICATION OF THE PR-FPIT COMPILER.

In this section we will prove that c^{PR} is correct and performance preserving.

7.2.1 Theorem. *Let* Σ *be any* S*-sorted signature, let* A *be any* Σ*-algebra, and let* P *be a performance measure for* A *which is based on clock* C*. Then* c^{PR} *is a compiler from* PR(Σ) *into* FPIT(Σ) *which is correct with respect to* $[\![.]\!]_A$ *and* F_A*, and performance preserving with respect to* λ_p^{PR} *and* λ_p^{FP}.

Proof. First notice that c^{PR} : PR(Σ) $\longrightarrow$ FPIT(Σ) by Definition 6.4.1: c^{PR} was defined to be an $S^+ \times S^+$-indexed family of mappings, and within the definition we showed that when $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ for some $u, v \in S^+$ we had $c^{PR}(\alpha) = c^{PR}_{u,v}(\alpha) \in \mathrm{FPIT}(\Sigma)_{u,v}$; thus c^{PR} is word-indexed and so a compiler from PR(Σ) into FPIT(Σ).

To prove correctness of c^{PR} we must show that for every $u, v \in S^+$ and for any $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, $c^{PR}(\alpha)$ satisfies:

$$(\forall a \in A^u)\ (F_A(c^{PR}(\alpha))(a) = [\![\alpha]\!]_A(a)) \tag{4}$$

To prove that c^{PR} is performance preserving we must show that for every $u, v \in S^+$ and for any $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, there exist $\mu, \nu \geq 1$ such that $c^{PR}(\alpha)$ satisfies:

$$(\forall a \in A^u)\ (\lambda_p^{FP}(c^{PR}(\alpha))(a) \leq \mu . \lambda_p^{PR}(\alpha)(a)) \tag{5}$$

and

$$(\forall a \in A^u)\ (\lambda_p^{PR}(\alpha)(a) \leq \nu . \lambda_p^{FP}(c^{PR}(\alpha))(a)) \tag{6}$$

Choose $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ for some $u, v \in S^+$. We prove c^{PR} is correct and performance preserving by proving that (4) holds, and by proving that (5) holds for some $\mu = \mu_\alpha$ and that (6) holds for $\nu = 1$. The proofs

are simultaneous, uniform in u and v, and by induction on the structural complexity of α, and proceed as follows:

Basis Cases.

(i) *Constant Functions.* Suppose $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$, and for some $w \in S^+$. Then $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$.

In this case $c^{PR}(\alpha)$ is defined by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,\ldots,x_n) : S_o$$

where:

$y \in Var_s$;

$x = (x_1,\ldots,x_n) \in Var_w$, and

$S_o = y := c$.

Correctness. To show (4) holds for $\alpha = c^w$, choose $a \in A^w$ and calculate as follows:

$$\begin{aligned} F_A(c^{PR}(\alpha))(a) &= F_A(S_o,x,y)(a) \\ &= \pi_A(y)(M_A(S_o)(\rho)) \end{aligned}$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$\begin{aligned} &= \pi_A(y)(M_A(y := c)(\rho)) \\ &= \pi_A(y)(\rho\{c^A/y\}) \\ &= c^A \\ &= [\![c^w]\!]_A(a) \\ &= [\![\alpha]\!]_A(a) \end{aligned}$$

Thus (4) holds for $\alpha = c^w$ as claimed; that is, c^{PR} correctly compiles schema of the form c^w.

Performance Preservation. To show that c^{PR} preserves the performance of schema of the form c^w, we must show that (5) holds for some $\mu \in \mathbb{N}$, and additionally, that (6) also holds.

First notice that for any $a \in A^w$ we have

$$\begin{aligned} \lambda_P^{FP}(c^{PR}(\alpha))(a) &= \lambda_P^{IO}(S_o,x,y)(a) \\ &= \lambda_P^{PIT}(S_o)(\rho) \end{aligned}$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$\begin{aligned} &= \lambda_P^{PIT}(y:=c)(\rho) \\ &= \lambda_P^{EXP}(c)(\rho) \\ &= c^P \\ &= \lambda_P^{PR}(c^w)(a) \\ &= \lambda_P^{PR}(\alpha)(a) \end{aligned}$$

Taking $\mu = 1$, it is now easy to see that $c^{PR}(\alpha)$ satisfies both performance preservation conditions (5) and (6); that is, c^{PR} preserves the performance of schema of the form c^w as claimed.

(ii) *Algebraic Operations.* Suppose $\alpha=\sigma$ for some $\sigma\in\Sigma_{w,s}$ for some $w\in S^+$and some $s\in S$. Then $\alpha\in \mathrm{PR}(\Sigma)_{w,s}$.

In this case $c^{PR}(\alpha)$ is defined by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,\ldots,x_n) : S_0$$

where:

$y\in Var_s$;

$x=(x_1,\ldots,x_n)\in Var_w$, and

$S_0 = y := \sigma(x_1,\ldots,x_n)$.

Correctness. See the first example in Section 6.3.6.

Performance Preservation. To show that c^{PR} preserves the performance of schema of the form c^w, we must show that (5) holds for some $\mu\in\mathbb{N}$, and additionally, that (6) also holds.

First notice that for any $a\in A^w$ we have:

$$\begin{aligned}\lambda_p^{FP}(c^{PR}(\alpha))(a) &= \lambda_p^{IO}(S_0,x,y)(a)\\ &= \lambda_p^{PIT}(S_0)(\rho)\end{aligned}$$

(where $\rho\in States(A)$ is any state such that $\pi_A(x)(\rho)=a$)

$$\begin{aligned}&= \lambda_p^{PIT}(y:=\sigma(x_1,\ldots,x_n))(\rho)\\ &= \lambda_p^{EXP}(\sigma(x_1,\ldots,x_n))(\rho)\\ &= \sigma^P(\rho(x_1),\ldots,\rho(x_n)) + \max_{1\le i\le n}\{\lambda_p^{EXP}(x_i)(\rho)\}\\ &= \sigma^P(\pi_A(x)(\rho) + \max_{1\le i\le n}\{1,\ldots,1\}\\ &= \sigma^P(\pi_A(x)(\rho)) + 1\\ &= \sigma^P(a) + 1\\ &= \lambda_p^{PR}(\sigma)(a) + 1\\ &= \lambda_p^{PR}(\alpha)(a) + 1 \qquad (7)\end{aligned}$$

To see that (5) holds for α and some choice of μ, take $\mu=2$, $a\in A^w$, and calculate as follows:

$$\lambda_p^{FP}(c^{PR}(\alpha))(a) = \lambda_p^{PR}(\alpha)(a) + 1$$

(from (7))

$$\le \lambda_p^{PR}(\alpha)(a) + \lambda_p^{PR}(\alpha)(a)$$

(since $\lambda_p^{PR}(\alpha)(a)\ge 1$ for any arguments α and a)

$$\begin{aligned}&= 2.\lambda_p^{PR}(\alpha)(a)\\ &= \mu.\lambda_p^{PR}(\alpha)(a)\end{aligned}$$

Thus (5) holds for α and this choice of μ as claimed.

To see that (6) holds for α, choose $a\in A^w$ and calculate as follows:

$$\lambda_P^{PR}(\alpha)(a) = \lambda_P^{FP}(c^{PR}(\alpha))(a) - 1$$

(from (7))

$$< \lambda_P^{FP}(c^{PR}(\alpha))(a)$$

Thus (6) holds for α as claimed.

(iii) *Projection Functions.* Suppose $\alpha = U_i^w$ for some $w \in S^+$ and some i with $1 \leq i \leq n = |w|$. Then $\alpha \in PR(\Sigma)_{w,w_i}$.

Then in this case $c^{PR}(\alpha)$ is defined by

$$c^{PR}(\alpha) = \textbf{function } y = f(x_1,...,x_n) : S_o$$

where:

$y \in Var_{w_i}$;

$x = (x_1,...,x_n) \in Var_w$, and

$S_o = y := x_i$.

Correctness. To show (4) holds for $\alpha = U_i^w$, choose $a \in A^w$ and calculate as follows:

$$F_A(c^{PR}(\alpha))(a) = F_A(S_o,x,y)(a)$$
$$= \pi_A(y)(M_A(y := x_i)(\rho))$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$= \pi_A(y)(\rho\{E_A(x_i)(\rho)/y\})$$
$$= \pi_A(y)(\rho\{\rho(x_i)/y\})$$
$$= \rho(x_i)$$
$$= a_i$$

(when $a = (a_1,...,a_n)$)

$$= [\![U_i^w]\!]_A(a)$$
$$= [\![\alpha]\!]_A(a)$$

Thus (4) holds for $\alpha = U_i^w$ as claimed; that is, c^{PR} correctly compiles schema of the form U_i^w.

Performance Preservation. To show that c^{PR} preserves the performance of schema of the form U_i^w, we must show that (5) holds for some $\mu \in \mathbb{N}$, and additionally, that (6) also holds.

First notice that for any $a \in A^w$ we have

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = \lambda_P^{IO}(S_o,x,y)(a)$$
$$= \lambda_P^{PIT}(S_o)(\rho)$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$= \lambda_P^{PIT}(y := x_i)(\rho)$$
$$= \lambda_P^{EXP}(x_i)(\rho)$$
$$= 1$$
$$= \lambda_P^{PR}(U_i^w)(a)$$

$$= \lambda_p^{PR}(\alpha)(a)$$

Taking $\mu = 1$, it is now easy to see that $c^{PR}(\alpha)$ satisfies both performance preservation conditions (5) and (6); that is, c^{PR} preserves the performance of schema of the form U_i^w as claimed.

Induction. Let $\alpha \in PR(\Sigma)_{u,v}$ for some $u,v \in S^*$. Suppose that α has the property that for every $u',v' \in S^+$ and for every $\alpha' \in PR(\Sigma)_{u',v'}$, with α' of less structural complexity than α, c^{PR} is correct and performance preserving on α'; that is, assume:

(a) $(\forall a \in A^{u'})\ (\ F_A(c^{PR}(\alpha'))(a) = [\![\alpha']\!]_A(a)\ )$

(b) there exists $\mu' \in \mathbb{N}^+$ such that $(\forall a \in A^{u'})\ (\ \lambda_p^{FP}(c^{PR}(\alpha'))(a) \le \mu' . \lambda_p^{PR}(\alpha')(a)\ )$

(c) $(\forall a \in A^{u'})\ (\ \lambda_p^{PR}(\alpha')(a) \le \lambda_p^{FP}(c^{PR}(\alpha'))(a)\ )$

We now show c^{PR} is correct and performance preserving on α according to the four following possible cases:

(iv) *Definition-by-Cases.* Suppose α is of the form $\alpha = DC(\beta,\alpha_1,\alpha_2)$. Then $\beta \in PR(\Sigma)_{u,B}$, and $\alpha_1,\alpha_2 \in PR(\Sigma)_{u,v}$ (since $\alpha \in PR(\Sigma)_{u,v}$ by hypothesis).

In this case $c^{PR}(\alpha)$ is defined by

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_b\ ;\ S_g\ ;\ S_h$ where:

$S_b = c^{PR}(\beta) \in FPIT(\Sigma)_{u,B}$ with $id(S_b) = (b)$ for some function identifier b;

$S_g = c^{PR}(\alpha_1) \in FPIT(\Sigma)_{u,v}$ with $id(S_g) = (g_1,\ldots,g_m)$ for some function identifiers $g_1,\ldots,g_m$, and

$S_h = c^{PR}(\alpha_2) \in FPIT(\Sigma)_{u,v}$ with $id(S_h) = (h_1,\ldots,h_m)$ for some function identifiers $h_1,\ldots,h_m$,

and

$S_o =$ **if** $b(x_1,\ldots,x_n)$ **then** S_1 **else** S_2 **fi**, where:

$S_1 = y_1,\ldots,y_m := g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)$, and

$S_2 = y_1,\ldots,y_m := h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n)$.

Correctness. (It should clear that the second example in Section 6.3.6 was tailored to this stage in the compilation: $c^{PR}(\alpha)$ is exactly the function program S of the example.)

First notice that since β, α_1, and α_2 are all of less structural complexity than α, by the induction hypothesis (a) applied to β, α_1, and α_2 respectively, we have:

$$(\forall a \in A^u)\ (F_A(S_b)(a) = [\![\beta]\!]_A(a)) \tag{8}$$

(since $S_b = c^{PR}(\beta)$)

$$(\forall a \in A^u)\ (F_A(S_g)(a) = [\![\alpha_1]\!]_A(a)) \tag{9}$$

(since $S_g = c^{PR}(\alpha_2)$)

$$(\forall a \in A^u)\ (F_A(S_h)(a) = [\![\alpha_2]\!]_A(a)) \tag{10}$$

(since $S_h = c^{PR}(\alpha_2)$).

Let $A_G = Alg_A(G)$. Then to show (4) holds for α, choose $a \in A^u$ and calculate as follows:

$$F_A(c^{PR}(\alpha))(a) = F_{A_o}(S_o, x, y)(a)$$
$$= \pi_A(y)(M_{A_o}(S_o)(\rho))$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$= \begin{cases} \pi_A(y)(\rho_1) & \text{if } E_{A_o}(b(x_1,\ldots,x_n))(\rho) = tt \\ \pi_A(y)(\rho_2) & \text{if } E_{A_o}(b(x_1,\ldots,x_n))(\rho) = ff \end{cases} \tag{11}$$

where $\rho_i = M_{A_o}(S_i)(\rho)$ for $i = 1,2$ (since S_o is a conditional statement).

Similar to Section 6.3.6, it is easy to show that

$$E_{A_o}(b(x_1,\ldots,x_n))(\rho) = F_A(S_b)(a) = [\![\beta]\!]_A(a) \tag{12}$$

(using (8))

$$\pi_A(y)(\rho_1) = F_A(S_g)(a) = [\![\alpha_1]\!]_A(a) \tag{13}$$

(using (9)) and

$$\pi_A(y)(\rho_2) = F_A(S_h)(a) = [\![\alpha_2]\!]_A(a) \tag{14}$$

(using (10)).

Now using (12), (13), and (14) in (11), we have for any $a \in A^u$ that

$$F_A(c^{PR}(\alpha))(a) = \begin{cases} [\![\alpha_1]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ [\![\alpha_2]\!]_A(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

$$= [\![DC(\beta,\alpha_1,\alpha_2)]\!]_A(a)$$

$$= [\![\alpha]\!]_A(a)$$

Thus (4) holds for schema $\alpha = DC(\beta,\alpha_1,\alpha_2)$; that is, c^{PR} correctly compiles schema of the form $DC(\beta,\alpha_1,\alpha_2)$.

Performance Preservation. To see that c^{PR} preserves the performance of α we must show that (5) holds for α and for some choice of μ, and we must additionally show that (6) also holds.

Let us first relate the performance of $c^{PR}(\alpha)$ and α by calculating as follows:

Choose $a \in A^u$. Then by definition of λ_P^{FP} we have:

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = \lambda_{P_o}^{IO}(S_o, x, y)(a)$$
$$= \lambda_{P_o}^{PIT}(S_o)(\rho)$$

(where $\rho \in States(A)$ is any state with $\pi_A(x)(\rho) = a$)

$$= \lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho) + \begin{cases} \lambda_{P_o}^{PIT}(S_1)(\rho) & \text{if } E_{A_o}(b(x_1,\ldots,x_n))(\rho) = tt \\ \lambda_{P_o}^{PIT}(S_2)(\rho) & \text{if } E_{A_o}(b(x_1,\ldots,x_n))(\rho) = ff \end{cases}$$

(since S_o is a conditional statement)

$$= \lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho) + \begin{cases} \lambda_{P_o}^{PIT}(S_1)(\rho) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_{P_o}^{PIT}(S_2)(\rho) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases} \quad (15)$$

(from (12)).

Now consider $\lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho)$: by definition of λ_P^{EXP} for arbitrary P we have:

$$\lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho) = b^{P_o}(\rho(x_1),\ldots,\rho(x_n)) + \max_{1 \le i \le n}\{\lambda_{P_o}^{EXP}(x_i)(\rho)\}$$
$$= b^{P_o}(\rho(x_1),\ldots,\rho(x_n)) + \max_{1 \le i \le n}\{1,\ldots,1\}$$
$$= b^{P_o}(\pi_A(x)(\rho)) + 1$$
$$= b^{P_o}(a) + 1 \quad (16)$$

However, by the definitions of G and P_G, the performance estimation for the symbol 'b' in P_G is the complexity of executing S_b. Thus from (16) we have:

$$\lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho) = b^{P_o}(a) + 1$$
$$= \lambda_P^{FP}(S_b)(a) + 1 \quad (17)$$

Now consider $\lambda_{P_o}^{PIT}(S_1)(\rho)$:

$$\lambda_{P_o}^{PIT}(S_1)(\rho) = \lambda_{P_o}^{PIT}(y_1,\ldots,y_m := g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n))(\rho)$$
$$= \max_{1 \le i \le m}\{\lambda_{P_o}^{EXP}(g_i(x_1,\ldots,x_n))(\rho)\}$$
$$= \max_{1 \le i \le m}\{g_i^{P_o}(\pi_A(x)(\rho)) + 1\}$$
$$= \max_{1 \le i \le m}\{g_i^{P_o}(a) + 1\}$$
$$= \max_{1 \le i \le m}\{g_i^{P_o}(a)\} + 1 \quad (18)$$

However, with respect to P_G, the cost of evaluating g_i on an argument a is the cost of executing S_g independent of the choice of i (see Definition 6.3.3). Thus from (18) we have:

$$\lambda_{P_o}^{PIT}(S_1)(\rho) = \max_{1 \le i \le m}\{g_i^{P_o}(a)\} + 1$$

$$= \max_{1 \le i \le m} \{ \lambda_P^{FP}(S_g)(a) \} + 1$$
$$= \lambda_P^{FP}(S_g)(a) + 1 \qquad (19)$$

In a similar way it is easy to show that for any $\rho \in States(A)$ with $\pi_A(x)(\rho) = a$,

$$\lambda_{P_o}^{PIT}(S_2)(\rho) = \max_{1 \le i \le m} \{ h_i^{P_o}(a) \} + 1 \cdot$$
$$= \max_{1 \le i \le m} \{ \lambda_P^{FP}(S_h)(a) \} + 1$$
$$= \lambda_P^{FP}(S_h)(a) + 1 \qquad (20)$$

Substituting for $\lambda_{P_o}^{EXP}(b(x_1,\ldots,x_n))(\rho)$, $\lambda_{P_o}^{PIT}(S_1)$, and $\lambda_{P_o}^{PIT}(S_2)(\rho)$ in (15) from (17), (19), and (20) respectively, we have:

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = \lambda_P^{FP}(S_b)(a) + 1 + \begin{cases} \lambda_P^{FP}(S_g)(a) + 1 & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{FP}(S_h)(a) + 1 & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$
$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = 2 + \lambda_P^{FP}(S_b)(a) + \begin{cases} \lambda_P^{FP}(S_g)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{FP}(S_h)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases} \qquad (21)$$

To see that (5) holds for α for some choice of μ, first notice that since β, α_1, and α_2 are all of less structural complexity than α, by the induction hypothesis (b) applied to β, α_1, and α_2 respectively, we have that there exist constants $\mu_b, \mu_g, \mu_h \in \mathbb{N}^+$ such that:

$$(\forall a \in A^u) \ (\lambda_P^{FP}(S_b)(a) \le \mu_b \,.\, \lambda_P^{PR}(\beta)(a)) \qquad (22)$$

(since $S_b = c^{PR}(\beta)$)

$$(\forall a \in A^u) \ (\lambda_P^{FP}(S_g)(a) \le \mu_g \,.\, \lambda_P^{PR}(\alpha_1)(a)) \qquad (23)$$

(since $S_g = c^{PR}(\alpha_2)$)

$$(\forall a \in A^u) \ (\lambda_P^{FP}(S_h)(a) \le \mu_h \,.\, \lambda_P^{PR}(\alpha_2)(a)) \qquad (24)$$

(since $S_h = c^{PR}(\alpha_2)$).

Also, by definition of λ_P^{PR} we have that for any $a \in A^u$,

$$\lambda_P^{PR}(\alpha)(a) = \lambda_P^{PR}(\beta)(a) + \begin{cases} \lambda_P^{PR}(\alpha_1)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{PR}(\alpha_2)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases} \qquad (25)$$

We can now show (5) holds for α for the choice $\mu = 2 + \max\{\mu_b, \mu_g, \mu_h\}$ in both of the following two possible cases:

Case 1: $[\![\beta]\!]_A(a) = tt$
Case 2: $[\![\beta]\!]_A(a) = ff$

Case 1. Suppose $[\![\beta]\!]_A(a) = tt$. Then from (25) we have:

$$\lambda_P^{PR}(\alpha)(a) = \lambda_P^{PR}(\beta)(a) + \lambda_P^{PR}(\alpha_1)(a) \qquad (26)$$

and from (21) we have:

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = 2 + \lambda_P^{FP}(S_b)(a) + \lambda_P^{FP}(S_g)(a)$$
$$\le 2 + \mu_b \,.\, \lambda_P^{PR}(\beta)(a) + \lambda_P^{FP}(S_g)(a)$$

(using the induction hypothesis (22))

$$\leq 2 + \mu_b \,.\, \lambda_P^{PR}(\beta)(a) + \mu_g \,.\, \lambda_P^{PR}(\alpha_1)(a)$$

(using the induction hypothesis (23))

$$\leq 2 + \max\{\mu_b, \mu_g\} \,.\, (\lambda_P^{PR}(\beta)(a) + \lambda_P^{PR}(\alpha_1)(a))$$
$$= 2 + \max\{\mu_b, \mu_g\} \,.\, \lambda_P^{PR}(\alpha)(a)$$

(from (26))

$$\leq 2 \,.\, \lambda_P^{PR}(\alpha)(a) + \max\{\mu_b, \mu_g\} \,.\, \lambda_P^{PR}(\alpha)(a)$$

(since $\lambda_P^{PR}(\alpha)(a) \geq 1$ for any α and any a)

$$= (2 + \max\{\mu_b, \mu_g\}) \,.\, \lambda_P^{PR}(\alpha)(a)$$
$$\leq (2 + \max\{\mu_b, \mu_g, \mu_h\}) \,.\, \lambda_P^{PR}(\alpha)(a)$$
$$= \mu \,.\, \lambda_P^{PR}(\alpha)(a)$$

Thus (5) holds for α and μ in this case.

Case 2. Suppose $[\![\beta]\!]_A(a) = ff$. Then similar to Case 1 above, it is easy to show from (21) and the induction hypothesis (24) that

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) \leq 2 + \mu_b \,.\, \lambda_P^{PR}(\beta)(a) + \mu_h \,.\, \lambda_P^{PR}(\alpha_2)(a)$$
$$\leq 2 + \max\{\mu_b, \mu_h\} \,.\, (\lambda_P^{PR}(\beta)(a) + \lambda_P^{PR}(\alpha_2)(a))$$
$$= 2 + \max\{\mu_b, \mu_h\} \,.\, \lambda_P^{PR}(\alpha)(a)$$

(using (25) and the case hypothesis: $[\![\beta]\!]_A(a) = ff$).

$$\leq 2 \,.\, \lambda_P^{PR}(\alpha)(a) + \max\{\mu_b, \mu_h\} \,.\, \lambda_P^{PR}(\alpha)(a)$$
$$\leq (2 + \max\{\mu_b, \mu_h\}) \,.\, \lambda_P^{PR}(\alpha)(a)$$
$$\leq (2 + \max\{\mu_b, \mu_g, \mu_h\}) \,.\, \lambda_P^{PR}(\alpha)(a)$$
$$= \mu \,.\, \lambda_P^{PR}(\alpha)(a)$$

Thus (5) also holds for α and μ in the case $[\![\beta]\!]_A(a) = ff$, and so (5) holds for α for our choice of μ for every $a \in A^{\omega}$.

We now show that the second performance preservation condition also holds for α; that is we now show (6) holds:

First, since β, α_1, and α_2 are all of less structural complexity than α, by the induction hypothesis (c) applied to β, α_1, and α_2 respectively, we have:

$$(\forall a \in A^{\omega}) \;\; (\lambda_P^{PR}(\beta)(a) \leq \lambda_P^{FP}(S_b)(a)) \tag{27}$$

$$(\forall a \in A^{\omega}) \;\; (\lambda_P^{PR}(\alpha_1)(a) \leq \lambda_P^{FP}(S_g)(a)) \tag{28}$$

and

$$(\forall a \in A^{\omega}) \;\; (\lambda_P^{PR}(\alpha_2)(a) \leq \lambda_P^{FP}(S_h)(a)) \tag{29}$$

Also, from (25) we have for any $a \in A^{\omega}$,

$$\lambda_P^{PR}(\alpha)(a) = \lambda_P^{PR}(\beta)(a) + \begin{cases} \lambda_P^{PR}(\alpha_1)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{PR}(\alpha_2)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

$$\leq \lambda_P^{FP}(S_b)(a) + \begin{cases} \lambda_P^{PR}(\alpha_1)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{PR}(\alpha_2)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

(using the induction hypothesis (27))

$$\leq \lambda_P^{FP}(S_b)(a) + \begin{cases} \lambda_P^{FP}(S_g)(a) & \text{if } [\![\beta]\!]_A(a) = tt \\ \lambda_P^{FP}(S_g)(a) & \text{if } [\![\beta]\!]_A(a) = ff \end{cases}$$

(using the induction hypotheses (28) and (29))

$$= \lambda_P^{FP}(c^{PR}(\alpha))(a) - 2$$

(from (21)).

$$< \lambda_P^{FP}(c^{PR}(\alpha))(a)$$

Thus (6) holds for α as claimed.

(v) *Vectorisation.* Suppose α is of the form $\alpha = <\alpha_1,\ldots,\alpha_m>$ for some $m \geq 1$. Then $\alpha_i \in \mathrm{PR}(\Sigma)_{u,v_i}$ for $i = 1,\ldots,m$ (since $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ by hypothesis).

In this case $c^{PR}(\alpha)$ is defined by:

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_1 ; \cdots ; S_m$, where:

$S_i = c^{PR}(\alpha_i) \in \mathrm{FPIT}(\Sigma)_{u,v_i}$ with $id(S_i) = (g_i)$ for some function identifier g_i for $i = 1,\ldots,m$,

and

$S_o = y_1,\ldots,y_m := g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)$.

Correctness. First notice that since $\alpha_1,\ldots,\alpha_m$ are all of less structural complexity than α, by the induction hypothesis (a) applied to α_i for $i = 1,\ldots,m$, we have:

$$(\forall a \in A^u)\ (\ F_A(S_i)(a) = [\![\alpha_i]\!]_A(a)\ ) \tag{30}$$

for $i = 1,\ldots,m$.

Let $A_G = Alg_A(G)$. Then similar to the previous induction case above, it follows from the definitions of G and A_G, that the interpretation of $g_i \in sig(G)_{u,v_i}$ in A_G is $F_A(S_i)$; that is,

$$(\forall a \in A^u)\ (\ g_i^{A_G}(a) = F_A(S_i)(a)\ ) \tag{31}$$

for $i = 1,\ldots,m$.

To show (4) holds for α, choose $a \in A^u$ and calculate as follows:

$$F_A(c^{PR}(\alpha))(a) = F_{A_o}(S_o, x, y)(a)$$
$$= \pi_A(y)(M_{A_o}(S_o)(\rho))$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$= (\rho'(y_1), \ldots, \rho'(y_m)) \tag{32}$$

when $\rho' = M_{A_o}(S_o)(\rho)$.

Now, since S_o is a multiple assignment statement, we have:

$$\rho' = M_{A_o}(S_o)(\rho)$$
$$= \rho\{b_1/y_1\}\{\cdots\}\{b_m/y_m\} \tag{33}$$

where for $i = 1, \ldots, m$, $b_i \in A_{v_i}$ is defined by

$$b_i = E_{A_o}(g_i(x_1, \ldots, x_n))(\rho)$$
$$= g_i^{A_o}(\rho(x_1), \ldots, \rho(x_n))$$
$$= g_i^{A_o}(\pi_A(x)(\rho))$$
$$= g_i^{A_o}(a)$$
$$= F_A(S_i)(a) \tag{34}$$

(from (31)).

Substituting for $b_1, \ldots, b_m$ (as given by (34) with $i = 1, \ldots, m$ respectively) in (33) now yields:

$$\rho' = \rho\{F_A(S_1)(a)/y_1\}\{\cdots\}\{F_A(S_m)(a)/y_m\}$$

and so substituting this formula for ρ' in (32) gives:

$$F_A(c^{PR}(\alpha))(a) = (F_A(S_1)(a), \ldots, F_A(S_m)(a))$$
$$= ([\![\alpha_1]\!]_A(a), \ldots, [\![\alpha_m]\!]_A(a))$$

(by the induction hypothesis (30) with $i = 1, \ldots, m$)

$$= [\![<\alpha_1, \ldots, \alpha_m>]\!]_A(a)$$
$$= [\![\alpha]\!]_A(a)$$

Thus (4) holds for α as claimed; that is, c^{PR} correctly compiles schema of the form $<\alpha_1, \ldots, \alpha_m>$.

Performance Preservation. To see that c^{PR} preserves the performance of α we must show that (5) holds for α and for some choice of μ, and we must additionally show that (6) also holds.

Let us first relate the performance of $c^{PR}(\alpha)$ and α by calculating as follows:

Choose $a \in A^u$. Then by definition of λ_p^{FP} we have:

$$\lambda_p^{FP}(c^{PR}(\alpha))(a) = \lambda_{P_o}^{IO}(S_o, x, y)(a)$$
$$= \lambda_{P_o}^{PIT}(S_o)(\rho)$$

(where $\rho \in States(A)$ is any state with $\pi_A(x)(\rho) = a$)

$$= \max_{1 \le i \le m}\{k_i\} \tag{35}$$

where for $i = 1, \ldots, m$, $k_i \in \mathbb{N}$ is defined by

$$
\begin{aligned}
k_i &= \lambda_{P_o}^{EXP}(g_i(x_1,\ldots,x_n))(\rho) \\
&= g_i^{P_o}(\pi_A(x)(\rho)) + \max_{1\le i\le n}\{\lambda_{P_o}^{EXP}(x_i)(\rho)\} \\
&= g_i^{P_o}(\pi_A(x)(\rho)) + 1 \\
&= g_i^{P_o}(a) + 1
\end{aligned} \tag{36}
$$

However, from the definitions of G and P_G, it follows that the cost of evaluating g_i with respect to P_G is the cost of executing S_i for $i = 1,\ldots,m$. Thus from (36) we have:

$$
\begin{aligned}
k_i &= g_i^{P_o}(a) + 1 \\
&= \lambda_P^{FP}(S_i)(a) + 1
\end{aligned} \tag{37}
$$

for $i = 1,\ldots,m$.

Substituting for k_i from (37) in (35) yields:

$$
\begin{aligned}
\lambda_P^{FP}(c^{PR}(\alpha))(a) &= \max_{1\le i\le m}\{k_i\} \\
&= \max_{1\le i\le m}\{\lambda_P^{FP}(S_i)(a) + 1\} \\
&= \max_{1\le i\le m}\{\lambda_P^{FP}(S_i)(a)\} + 1
\end{aligned} \tag{38}
$$

To see that (5) holds for α and for some choice of μ, first notice that since $\alpha_1,\ldots,\alpha_m$ are all of less structural complexity than α, by the induction hypothesis (b) applied to $\alpha_1,\ldots,\alpha_m$ we have that there exist constants $\mu_1,\ldots\mu_m \in \mathbb{N}^+$ respectively such that:

$$(\forall a \in A^*)\ (\lambda_P^{FP}(S_i)(a) \le \mu_i \,.\, \lambda_P^{PR}(\alpha_i)(a)) \tag{39}$$

for $i = 1,\ldots,m$.

To show (5) holds for α for the choice $\mu = 1 + \mu_o$ where $\mu_o = \max\{\mu_1,\ldots,\mu_m\}$, choose $a \in A^*$ and calculate as follows:

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = \max_{1\le i\le m}\{\lambda_P^{FP}(S_i)(a)\} + 1$$

(from (38))

$$\le \max_{1\le i\le m}\{\mu_i \,.\, \lambda_P^{PR}(\alpha_i)(a)\} + 1$$

(by (39) with $i = 1,\ldots,n$)

$$\le \max_{1\le i\le m}\{\mu_i\} \,.\, \max_{1\le i\le m}\{\lambda_P^{PR}(\alpha_i)(a)\} + 1$$

(since for any $a,b,c,d \ge 1$, $\max\{a \,.\, b, c \,.\, d\} \le \max\{a,c\} \,.\, \max\{b,d\}$)

$$
\begin{aligned}
&= \mu_o \,.\, \max_{1\le i\le m}\{\lambda_P^{PR}(\alpha_i)(a)\} + 1 \\
&= \mu_o \,.\, \lambda_P^{PR}(\langle\alpha_1,\ldots,\alpha_m\rangle)(a) + 1 \\
&= \mu_o \,.\, \lambda_P^{PR}(\alpha)(a) + 1 \\
&\le \mu_o \,.\, \lambda_P^{PR}(\alpha)(a) + \lambda_P^{PR}(\alpha)(a)
\end{aligned}
$$

(since $\lambda_P^{PR}(\alpha)(a) \ge 1$ for any arguments α and a)

$$= \mu \,.\, \lambda_P^{PR}(\alpha)(a)$$

Thus (5) holds for α for our choice of μ as claimed.

We now show that the second performance preservation condition also holds for α; that is we now show (6) holds:

First, since $\alpha_1,\ldots,\alpha_m$ are all of less structural complexity than α, by the induction hypothesis (c) applied to $\alpha_1,\ldots,\alpha_m$ we have:

$$(\forall a \in A^u)\ (\lambda_p^{PR}(\alpha_i)(a) \leq \lambda_p^{FP}(S_i)(a)) \tag{40}$$

for $i = 1,\ldots,m$.

Now, by definition of λ_p^{PR} we have for any $a \in A^u$ that

$$\lambda_p^{PR}(\alpha)(a) = \max_{1 \leq i \leq m} \{\lambda_p^{PR}(\alpha_i)(a)\}$$

$$\leq \max_{1 \leq i \leq m} \{\lambda_p^{FP}(S_i)(a)\}$$

(by the induction hypothesis (40) with $i = 1,\ldots,m$)

$$= \lambda_p^{FP}(c^{PR}(\alpha))(a) - 1$$

(from (38))

$$< \lambda_p^{FP}(c^{PR}(\alpha))(a)$$

Thus (6) holds for α as claimed.

(vi) *Composition.* Suppose α is of the form $\alpha = \alpha_2 \circ \alpha_1$. Then for some $w \in S^+$, $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$ (since $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ by hypothesis).

In this case $c^{PR}(\alpha)$ is defined by:

$$c^{PR}(\alpha) = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G = S_g\ ;\ S_h$, where:

$S_g = c^{PR}(\alpha_1) \in \mathrm{FPIT}(\Sigma)_{u,w}$ with $id(S_g) = (g_1,\ldots,g_k)$ for some function identifiers $g_1,\ldots,g_k$, and

$S_h = c^{PR}(\alpha_2) \in \mathrm{FPIT}(\Sigma)_{w,v}$ with $id(S_h) = (h_1,\ldots,h_m)$ for some function identifiers $h_1,\ldots,h_m$,

and

$S_o = y_1,\ldots,y_m := e_1,\ldots,e_m$, where:

$e_i = h_i(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n))$ for $i = 1,\ldots,m$, where $k = |w|$.

Correctness. First notice that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (a) applied to α_1 and α_2 respectively, we have:

$$(\forall a \in A^u)\ \ (F_A(S_g)(a) = [\![\alpha_1]\!]_A(a)) \tag{41}$$

and

$$(\forall a \in A^w)\ \ (F_A(S_h)(a) = [\![\alpha_2]\!]_A(a)) \tag{42}$$

Let $A_G = Alg_A(G)$. Then to show (4) holds for α, choose $a \in A^u$ and calculate as follows:

$$F_A(c^{PR}(\alpha))(a) = F_{A_o}(S_o, x, y)(a)$$
$$= \pi_A(y)(M_{A_o}(S_o)(\rho))$$

(where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = a$)

$$= (\rho'(y_1), \ldots, \rho'(y_m)) \tag{43}$$

when $\rho' = M_{A_o}(S_o)(\rho)$.

Now, for $i = 1, \ldots, k$, let $c_i \in A_{w_i}$ be defined by

$$c_i = E_{A_o}(g_i(x_1, \ldots, x_n))(\rho) \tag{44}$$
$$= g_i^{A_o}(\pi_A(x)(\rho))$$
$$= g_i^{A_o}(a) \tag{45}$$

Then similar to the previous induction cases above, it follows from the definitions of G and A_G, that the interpretation of $g_i \in sig(G)_{u,w_i}$ in A_G is $F_A(S_g)_i$. Thus from (45) we have:

$$c_i = g_i^{A_o}(a)$$
$$= F_A(S_g)_i(a) \tag{46}$$

Now, since S_o is a multiple assignment statement, we have:

$$\rho' = M_{A_o}(S_o)(\rho)$$
$$= \rho\{b_1/y_1\}\{\cdots\}\{b_m/y_m\} \tag{47}$$

where for $i = 1, \ldots, m$, $b_i \in A_{v_i}$ is defined by

$$b_i = E_{A_o}(h_i(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)))(\rho)$$
$$= h_i^{A_o}(E_{A_o}(g_1(x_1, \ldots, x_n))(\rho), \ldots, E_{A_o}(g_k(x_1, \ldots, x_n))(\rho))$$
$$= h_i^{A_o}(c_1, \ldots, c_k) \tag{48}$$

(by definition of $c_1, \ldots c_k$; see (44) above).

It is not difficult to see that similar to previous induction cases, the interpretation of $h_i \in sig(G)_{w,v_i}$ is the ith coordinate of $F_A(S_h)$, viz $F_A(S_h)_i$. Thus from (48) we have:

$$b_i = h_i^{A_o}(c_1, \ldots, c_k)$$
$$= F_A(S_h)_i(c_1, \ldots, c_k)$$
$$= F_A(S_h)_i(F_A(S_g)_1(a), \ldots, F_A(S_g)_k(A))$$

(using (46) with $i = 1, \ldots, k$)

$$= F_A(S_h)_i(F_A(S_g)(a))$$

(since $F_A(S_g)_1,\ldots,F_A(S_g)_k$ are the coordinates of $F_A(S_g)$).

$$= F_A(S_h)_i([\![\alpha_1]\!]_A(a)) \tag{49}$$

(by the induction hypothesis (41)).

Substituting for b_i from (49) in (47) now yields:

$$\rho' = M_{A_o}(S_o)(\rho)$$

$$= \rho\{F_A(S_h)_1([\![\alpha_1]\!]_A(a))/y_1\}\{\cdots\}\{F_A(S_h)_m([\![\alpha_1]\!]_A(a))/y_m\} \tag{50}$$

Thus from (43) we now have for any $a \in A^{\mu}$ that

$$F_A(c^{PR}(\alpha))(a) = (\rho'(y_1),\ldots,\rho'(y_m))$$

$$= (F_A(S_h)_1([\![\alpha_1]\!]_A(a)),\ldots,F_A(S_h)_m([\![\alpha_1]\!]_A(a)))$$

(using (50))

$$= F_A(S_h)([\![\alpha_1]\!]_A(a))$$

(since $F_A(S_h)_1,\ldots,F_A(S_h)_m$ are the coordinates of $F_A(S_h)$).

$$= [\![\alpha_2]\!]_A([\![\alpha_1]\!]_A(a))$$

(by the induction hypothesis (42)).

$$= [\![\alpha_2 \circ \alpha_1]\!]_A(a)$$

$$= [\![\alpha]\!]_A(a)$$

Thus (4) holds for α as claimed; that is, c^{PR} correctly compiles schema of the form $\alpha_2 \circ \alpha_1$.

Performance Preservation. To see that c^{PR} preserves the performance of α we must show that (5) holds for α and for some choice of μ, and we must additionally show that (6) also holds.

Let us first relate the performance of $c^{PR}(\alpha)$ and α by calculating as follows:

Choose $a \in A^{\mu}$. Then by definition of λ_p^{FP} we have:

$$\lambda_p^{FP}(c^{PR}(\alpha))(a) = \lambda_{P_o}^{IO}(S_o,x,y)(a)$$

$$= \lambda_{P_o}^{PIT}(S_o)(\rho)$$

$$= \lambda_{P_o}^{PIT}(y_1,\ldots,y_m := e_1,\ldots,e_m)(\rho)$$

$$= \max_{1\le i\le m}\{\lambda_{P_o}^{EXP}(e_i)(\rho)\}$$

$$= \max_{1\le i\le m}\{\lambda_{P_o}^{EXP}(h_i(e'_1,\ldots,e'_k))(\rho)\} \tag{51}$$

where $e'_i = g_i(x_1,\ldots,x_n)$ for $i = 1,\ldots,k$ (by definition of the expressions $e_1,\ldots,e_m$).

Now consider the complexity of evaluating g_i for $i = 1,\ldots,k$ with respect to P_G: from the definitions of G and P_G, this is the cost of executing S_g. Thus, for any $\rho \in States(A)$ with $\pi_A(x)(\rho) = a$, we have:

$$\lambda_{P_o}^{EXP}(e'_i)(\rho) = \lambda_{P_o}^{EXP}(g_i(x_1,\ldots,x_n))(\rho)$$

$$= g_i^{P_o}((\rho(x_1),\ldots\rho(x_n))) + \max_{1\le i\le n}\{\lambda_{P_o}^{EXP}(x_i)(\rho)\}$$

$$= g_i^{P_o}(\pi_A(x)(\rho)) + 1$$

$$= g_i^{P_o}(a) + 1$$

$$= \lambda_P^{FP}(S_g)(a) + 1$$

Thus $\lambda_{P_o}^{EXP}(e'_i)(\rho)$ is independent of the index i and so

$$\max_{1 \le i \le k} \{ \lambda_{P_o}^{EXP}(e'_i)(\rho) \} = \lambda_P^{FP}(S_g)(a) + 1 \tag{52}$$

Similarly, the complexity of evaluating h_i for $i = 1,\ldots,m$ with respect to P_G is the cost of executing S_h. Thus, for any $\rho \in States(A)$ with $\pi_A(x)(\rho) = a$, we have:

$$\lambda_{P_o}^{EXP}(h_i(e'_1,\ldots,e'_k))(\rho) = h_i^{P_o}(E_{A_o}(e'_1)(\rho),\ldots,E_{A_o}(e'_k)(\rho)) + \max_{1 \le i \le k} \{ \lambda_{P_o}^{EXP}(e'_i)(\rho) \}$$

$$= \lambda_P^{FP}(S_h)(E_{A_o}(e'_1)(\rho),\ldots,E_{A_o}(e'_k)(\rho)) + \max_{1 \le i \le k} \{ \lambda_{P_o}^{EXP}(e'_i)(\rho) \}$$

$$= \lambda_P^{FP}(S_h)(E_{A_o}(e'_1)(\rho),\ldots,E_{A_o}(e'_k)(\rho)) + \lambda_P^{FP}(S_g)(a) + 1 \tag{53}$$

(using (52)).

Now, from (44) and (46) we have that for any $\rho \in States(A)$ with $\pi_A(x)(\rho) = a$,

$$E_{A_o}(e'_i)(\rho) = F_A(S_g)_i(a) \tag{54}$$

for $i = 1,\ldots,k$.

Thus, from (53) we have

$$\lambda_{P_o}^{EXP}(h_i(e'_1,\ldots,e'_k))(\rho) = \lambda_P^{FP}(S_h)(E_{A_o}(e'_1)(\rho),\ldots,E_{A_o}(e'_k)(\rho)) + \lambda_P^{FP}(S_g)(a) + 1$$

$$= \lambda_P^{FP}(S_h)(F_A(S_g)_1(a),\ldots,F_A(S_g)_k(a)) + \lambda_P^{FP}(S_g)(a) + 1$$

(using (54) with $i = 1,\ldots,k$)

$$= \lambda_P^{FP}(S_h)(F_A(S_g)(a)) + \lambda_P^{FP}(S_g)(a) + 1 \tag{55}$$

(since $F_A(S_g)_1,\ldots,F_A(S_g)_k$ are the coordinates of $F_A(S_g)$).

To see that (5) holds for α for some choice of μ, first notice that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (b) applied to α_1 and α_2 respectively, we have that there exist constants $\mu_1,\mu_2 \in \mathbb{N}^+$ such that:

$$(\forall a \in A^w)\ (\lambda_P^{FP}(S_g)(a) \le \mu_1 . \lambda_P^{PR}(\alpha_1)(a)) \tag{56}$$

(since $S_g = c^{PR}(\alpha_2)$)

$$(\forall a \in A^w)\ (\lambda_P^{FP}(S_h)(a) \le \mu_2 . \lambda_P^{PR}(\alpha_2)(a) \tag{57}$$

(since $S_h = c^{PR}(\alpha_2)$).

Now choose $\mu = 1+\max\{\mu_1,\mu_2\}$. Then to show (5) holds for α and this choice of μ, choose $a \in A^w$ and calculate as follows:

$$\lambda_P^{FP}(c^{PR}(\alpha))(a) = \max_{1 \le i \le m} \{ \lambda_{P_o}^{EXP}(h_i(e'_1,\ldots,e'_k))(\rho) \}$$

(from (51))

$$= \lambda_P^{FP}(S_g)(a) + \lambda_P^{FP}(S_h)(F_A(S_g)(a)) + 1 \tag{58}$$

(using (55))

$$\leq \mu_1 . \lambda_P^{PR}(\alpha_1)(a) + \lambda_P^{FP}(S_h)(F_A(S_g)(a)) + 1$$

(by the induction hypothesis (56))

$$\leq \mu_1 . \lambda_P^{PR}(\alpha_1)(a) + \mu_2 . \lambda_P^{PR}(\alpha_2)(F_A(S_g)(a)) + 1$$

(by the induction hypothesis (57))

$$\leq \max\{\mu_1,\mu_2\} . (\lambda_P^{PR}(\alpha_1)(a) + \lambda_P^{PR}(\alpha_2)(F_A(S_g)(a))) + 1$$

$$= \max\{\mu_1,\mu_2\} . (\lambda_P^{PR}(\alpha_1)(a) + \lambda_P^{PR}(\alpha_2)([\![\alpha_1]\!]_A(a))) + 1$$

(since c^{PR} correctly compiles α_1; see (41) above)

$$= \max\{\mu_1,\mu_2\} . \lambda_P^{PR}(\alpha_2 \circ \alpha_1)(a) + 1$$

$$= \max\{\mu_1,\mu_2\} . \lambda_P^{PR}(\alpha)(a) + 1$$

$$\leq \max\{\mu_1,\mu_2\} . \lambda_P^{PR}(\alpha)(a) + \lambda_P^{PR}(\alpha)(a)$$

(since $\lambda_P^{PR}(\alpha)(a) \geq 1$ for any arguments α and a)

$$= \mu . \lambda_P^{PR}(\alpha)(a)$$

Thus (5) holds for α for our choice of μ as claimed.

We now show that the second performance preservation condition also holds for α; that is we now show (6) holds:

First, since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (c) applied to α_1 and α_2 respectively, we have:

$$(\forall a \in A^u) \ (\lambda_P^{PR}(\alpha_1)(a) \leq \lambda_P^{FP}(S_g)(a)) \tag{59}$$

and

$$(\forall a \in A^w) \ (\lambda_P^{PR}(\alpha_2)(a) \leq \lambda_P^{FP}(S_h)(a)) \tag{60}$$

Now, by definition of λ_P^{PR} we have for any $a \in A^u$ that

$$\lambda_P^{PR}(\alpha)(a) = \lambda_P^{PR}(\alpha_2 \circ \alpha_1)(a)$$

$$= \lambda_P^{PR}(\alpha_1)(a) + \lambda_P^{PR}(\alpha_2)([\![\alpha_1]\!]_A(a))$$

$$\leq \lambda_P^{FP}(S_g)(a) + \lambda_P^{PR}(\alpha_2)([\![\alpha_1]\!]_A(a))$$

(by the induction hypothesis (59))

$$\leq \lambda_P^{FP}(S_g)(a) + \lambda_P^{FP}(S_h)(F_A(S_g)(a))$$

(since c^{PR} correctly compiles α_1)

$$= \lambda_P^{FP}(c^{PR}(\alpha))(a) - 1$$

(by (58))

$$< \lambda_P^{FP}(c^{PR}(\alpha))(a)$$

Thus (6) holds for α as claimed.

(vii) *Primitive Recursion.* Suppose α is of the form $\alpha = *(\alpha_1,\alpha_2)$. Then the arity of α in this case must be $(\tau uv, v)$ for some $u,v \in S^+$, and $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\tau uv,v}$.

In this case $c^{PR}(\alpha)$ is defined by

$$c^{PR}(\alpha) = \textbf{function } y_1,\ldots,y_m = f_1(p,x_1,\ldots,x_n),\ldots,f_m(p,x_1,\ldots,x_n) : G : S_0$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (p,x_1,\ldots,x_n) \in Var_{\tau u}$;

$G = S_g ; S_h$, where

$S_g = c^{PR}(\alpha_1) \in \mathrm{FPIT}(\Sigma)_{u,v}$ with $id(S_g) = (g_1,\ldots,g_m)$ for some function identifiers $g_1,\ldots,g_m$, and

$S_h = c^{PR}(\alpha_2) \in \mathrm{FPIT}(\Sigma)_{\tau uv,v}$ with $id(S_h) = (h_1,\ldots,h_m)$ for some function identifiers $h_1,\ldots,h_m$

and

$S_0 = S_1$; **do** p **times** S_2 **od**, where:

$S_1 = z,y_1,\ldots,y_m := \mathrm{zero}, g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)$, and

$S_2 = z,y_1,\ldots,y_m := \mathrm{succ}(z),e_1,\ldots,e_m$, where:

$z \in Var_\tau$, and

$e_i = h_i(z,x_1,\ldots,x_n,y_1,\ldots,y_m)$ for $i = 1,\ldots,m$.

Correctness. First notice that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (a) applied to α_1 and α_2 respectively, we have:

$$(\forall a \in A^u)\ (F_A(S_g)(a) = [\![\alpha_1]\!]_A(a)) \tag{61}$$

and

$$(\forall t \in T)(\forall a \in A^u)(\forall b \in A^v)\ (F_A(S_h)(t,a,b) = [\![\alpha_2]\!]_A(t,a,b)) \tag{62}$$

Now let $S_3 =$ **do** p **times** S_2 **od**; then $S_0 = S_1;S_3$, and so for any $\rho \in States(A)$,

$$M_{A_o}(S_0)(\rho) = M_{A_o}(S_3)(M_{A_o}(S_1)(\rho))$$
$$= M_{A_o}(S_3)(\rho')$$

when $\rho' = M_{A_o}(S_1)(\rho)$ and where $A_G = Alg_A(G)$. But since S_3 is a loop we have:

$$M_{A_o}(S_0)(\rho) = \rho_l$$

where $l = E_{A_o}(p)(\rho) = \rho(p)$, and where for each $k \geq 0$, $\rho_k \in States(A)$ is defined by

$$\rho_0 = \rho'$$

and

$$\rho_{k+1} = M_{A_o}(S_2)(\rho_k)$$

We now make the following claim which we will prove later:

Claim. Let $x' = (x_1,\ldots,x_n) \in Var_u$ and let $a \in A^u$. Then for every $k \geq 0$, and for every $\rho \in States(A)$ with $\pi_A(x')(\rho) = a$,

$$\rho_k(z) = k, \tag{63}$$

$$\pi_A(x')(\rho_k) = a, \tag{64}$$

and,

$$\pi_A(y)(\rho_k) = [\![\alpha]\!]_A(k,a) \tag{65}$$

To see that c^{PR} correctly compiles α, we must show that for every $k \geq 0$ and every $a \in A^u$,

$$F_A(c^{PR}(\alpha))(k,a) = [\![\alpha]\!]_A(k,a) \tag{66}$$

Now, by definition of F_A we have for any $k \geq 0$ and $a \in A^u$ that

$$\begin{aligned} F_A(c^{PR}(\alpha))(k,a) &= F_{A_o}(S_o,x,y)(k,a) \\ &= \pi_A(y)(M_{A_o}(S_o)(\rho)) \end{aligned} \tag{67}$$

where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = (k,a)$. (Notice here that unlike the previous induction cases, the vector x is not $(x_1,\ldots,x_n)$, but $(p,x_1,\ldots,x_n)$, that is, x is a member of Var_{Tu} and not Var_u; thus it makes sense that $\pi_A(x)(\rho)$ should be of the form (k,a) for some number k and some $a \in A^u$.)

Now, if $\pi_A(x)(\rho) = (k,a)$, then it follows that $\pi_A(x')(\rho) = a$ and $\rho(p) = k$; thus by the definition of $M_{A_o}(S_o)$ we have:

$$\begin{aligned} \pi_A(y)(M_{A_o}(S_o)(\rho)) &= \pi_A(y)(\rho_k) \\ &= [\![\alpha]\!]_A(k,a) \end{aligned} \tag{68}$$

(by (65)).

It is now easy to see that (66) holds for each $k \geq 0$ and each $a \in A^u$ from (67) and (68); that is, c^{PR} correctly compiles α as claimed.

Proof of Claim. Let $\rho \in States(A)$ be such that $\pi_A(x')(\rho) = a$. We now prove (63), (64), and (65) hold by simultaneous sub-induction on k:

Sub-Basis. For $k = 0$, we have $\rho_k = \rho_0 = \rho' = M_{A_o}(S_1)(\rho)$. But since S_1 is a multiple assignment statement, we have:

$$\rho_0 = \rho\{0/z\}\{g_1^{A_o}(a)/y_1\}\{\cdots\}\{g_m^{A_o}(a)/y_m\} \tag{69}$$

Similar to the previous induction cases, it is not difficult to see that the interpretation of $g_i \in sig(G)_{u,v_i}$ in A_G is the ith coordinate of $F_A(S_g)$. Thus from (69) we have:

$$\rho_0 = \rho\{0/z\}\{F_A(S_g)_1(a)/y_1\}\{\cdots\}\{F_A(S_g)_m(a)/y_m\} \tag{70}$$

Hence,

$$\rho_0(z) = 0$$

and so (63) certainly holds for $k = 0$.

Also, it easily follows from (70) that for $i = 1,\ldots,n$,

$$\rho_o(x_i) = \rho(x_i) = a_i$$

and so (64) holds for $k = 0$.

Also from (70) we have for $i = 1,\ldots,m$,

$$\rho_o(y_i) = F_A(S_g)_i(a)$$

and thus

$$\begin{aligned}\pi_A(y)(\rho_o) &= (\rho_o(y_1),\ldots,\rho_o(y_m)) \\ &= (F_A(S_g)_1(a),\ldots,F_A(S_g)_m(a)) \\ &= F_A(S_g)(a)\end{aligned}$$

(since $F_A(S_g)_1,\ldots,F_A(S_g)_m$ are the coordinates of $F_A(S_g)$)

$$= [\![\alpha_1]\!]_A(a)$$

(by the induction hypothesis (61))

$$\begin{aligned}&= [\![*(\alpha_1,\alpha_2)]\!]_A(0,a) \\ &= [\![\alpha]\!]_A(0,a)\end{aligned}$$

Thus (65) also holds for $k = 0$.

Sub-Induction. Suppose for some fixed $l \in \mathbb{N}$ that for any $\rho \in States(A)$ with $\pi_A(x')(\rho) = a$ that

$$\rho_l(z) = l, \tag{71}$$

$$\pi_A(x')(\rho_l) = a, \tag{72}$$

and,

$$\pi_A(y)(\rho_l) = [\![\alpha]\!]_A(l,a) \tag{73}$$

We now show that (63), (64), and (65) hold for $k = l+1$:

By definition of ρ_{l+1} we have

$$\begin{aligned}\rho_{l+1} &= M_{A_o}(S_1)(\rho_l) \\ &= \rho_l\{\rho_l(z)+1/z\}\{E_{A_o}(e_1)(\rho_l)/y_1\}\{\cdots\}\{E_{A_o}(e_m)(\rho_l)/y_m\}\end{aligned} \tag{74}$$

Thus,

$$\begin{aligned}\rho_{l+1}(z) &= \rho_l(z)+1 \\ &= l+1\end{aligned}$$

(by the sub-induction hypothesis (71)), and so (63) holds for $k = l+1$.

Also, it is not difficult to see (from (74)) that the values of $x_1,\ldots,x_n$ are unchanged in executing S_2, and so

$$\rho_{l+1}(x_i) = \rho_l(x_i) = a_i$$

for $i = 1,\ldots,n$. Thus (64) also holds for $k = l+1$.

To show that (65) holds for $k = l+1$, first notice that for $i = 1,\ldots,m$,

$$\begin{aligned}E_{A_o}(e_i)(\rho_l) &= E_{A_o}(h_i(z,x_1,\ldots,x_n,y_1,\ldots,y_m))(\rho_l) \\ &= \hat{h_i}^{A_o}(\rho_l(z),\rho_l(x_1),\ldots,\rho_l(x_n),\rho_l(y_1),\ldots,\rho_l(y_m)) \\ &= \hat{h_i}^{A_o}(\rho_l(z),\pi_A(x')(\rho_l),\pi_A(y)(\rho_l))\end{aligned}$$

$$= h_i^{A_o}(l,\pi_A(x')(\rho_l),\pi_A(y)(\rho_l))$$

(by the sub-induction hypothesis (71))

$$= h_i^{A_o}(l,a,\pi_A(y)(\rho_l))$$

(by the sub-induction hypothesis (72))

$$= h_i^{A_o}(l,a,[\![\alpha]\!]_A(l,a)) \tag{75}$$

(by the sub-induction hypothesis (73)).

However, is not difficult to see that the interpretation of $h_i \in sig(G)_{\tau\iota\omega,\nu_i}$ in A_G is the ith coordinate of $F_A(S_k)$. Thus from (75) we have:

$$E_{A_o}(e_i)(\rho_l) = F_A(S_k)_i(l,a,[\![\alpha]\!]_A(l,a)) \tag{76}$$

for $i = 1,\ldots,m$.

Now,

$$\pi_A(y)(\rho_{l+1}) = (\rho_{l+1}(y_1),\ldots,\rho_{l+1}(y_m))$$
$$= (E_{A_o}(e_1)(\rho_l),\ldots,E_{A_o}(e_m)(\rho_l))$$

(using (74))

$$= (F_A(S_k)_1(l,a,[\![\alpha]\!]_A(l,a)),\ldots,F_A(S_k)_m(l,a,[\![\alpha]\!]_A(l,a)))$$

(using (76) with $i = 1,\ldots,m$)

$$= F_A(S_k)(l,a,[\![\alpha]\!]_A(l,a))$$

(since $F_A(S_k)_1,\ldots,F_A(S_k)_m$ are the coordinates of $F_A(S_k)$)

$$= [\![\alpha_2]\!]_A(l,a,[\![\alpha]\!]_A(l,a))$$

(by the induction hypothesis (62))

$$= [\![*(\alpha_1,\alpha_2)]\!]_A(l+1,a)$$

$$= [\![\alpha]\!]_A(l+1,a)$$

Thus thus (65) holds for $k = l+1$ as claimed, completing the sub-induction. □

Performance Preservation. To see that c^{PR} preserves the performance of α we must show that (5) holds for α and for some choice of μ, and we must additionally show that (6) also holds.

Let us first relate the performance of $c^{PR}(\alpha)$ and α by calculating as follows:

Choose $k \geq 0$ and $a \in A^u$. Then by definition of λ_p^{FP} we have:

$$\lambda_p^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_o}^{IO}(S_0,x,y)(k,a)$$
$$= \lambda_{P_o}^{PIT}(S_0)(\rho)$$

(where $\rho \in States(A)$ is any state with $\pi_A(x)(\rho) = (k,a)$)

$$= \lambda_{P_o}^{PIT}(S_1)(\rho) + \lambda_{P_o}^{PIT}(S_3)(\rho') \tag{77}$$

(recall $S = S_1;S_3$ and $\rho' = M_A(S_1)(\rho)$).

Now, since S_3 is a loop, we have:

$$\lambda_{P_e}^{PIT}(S_3)(\rho') = \lambda_{P_e}^{EXP}(p)(\rho') + \lambda_l \tag{78}$$

where $l = E_{A_e}(p)(\rho') = \rho'(p)$, and where for each $k \geq 0$, λ_k is defined by

$$\lambda_0 = 0$$
$$\lambda_{k+1} = \lambda_k + \lambda_{P_e}^{PIT}(S_2)(\rho_k)$$

where ρ_k is as defined above.

Now, if $\pi_A(x)(\rho) = (k,a)$ then $\rho(p) = k$ (recall that the first element of x is $p \in Var_\tau$). Also, since $\rho' = M_{A_e}(S_1)(\rho)$, and since p does not occur on the left hand side of the multiple assignment S_1, it follows that $\rho'(p) = k = l$ and so we can substitute for $\lambda_{P_e}^{PIT}(S_3)$ from (78) in (77), yielding:

$$\lambda_P^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_e}^{PIT}(S_1)(\rho) + \lambda_{P_e}^{EXP}(p)(\rho') + \lambda_k$$

for any state ρ such that $\pi_A(x)(\rho) = (k,a)$. However, since the expression 'p' is just a variable, it immediately follows that:

$$\lambda_P^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_e}^{PIT}(S_1)(\rho) + 1 + \lambda_k \tag{79}$$

In order to prove that c^{PR} preserves the performance of α, let us define

$$\kappa : \mathbb{N} \times States(A) \longrightarrow \mathbb{N}$$

by

$$\kappa(0,\rho) = 1$$
$$\kappa(k+1,\rho) = \kappa(k,\rho) + \lambda_{P_e}^{PIT}(S_2)(\rho_k) \tag{80}$$

for each $k \geq 0$ and $\rho \in States(A)$. (Here ρ_k is as defined immediately prior to the previous claim).

Then a routine induction on k yields:

$$(\forall k \geq 0)(\forall \rho \in States(A)) \quad (\kappa(k,\rho) = 1 + \lambda_k) \tag{81}$$

Thus, for any $\rho \in States(A)$ with $\pi_A(x)(\rho) = (k,a)$, we have

$$\lambda_P^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_e}^{PIT}(S_1)(\rho) + 1 + \lambda_k$$

(from (79))

$$= \lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \tag{82}$$

(using (81))

We now make the following claim which we prove below:

Claim. Let $k \geq 0$ and $a \in A^u$. Then there exists $\mu_0 \in \mathbb{N}^+$ such that for every $\rho \in States(A)$ with $\pi_A(x)(\rho) = (k,a)$,

$$\lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \leq \mu_0 . \lambda_P^{PR}(\alpha)(k,a) \tag{83}$$

To see that (5) holds for α and the choice $\mu = \mu_0$, we calculate as follows:

$$\lambda_P^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho)$$

(from (82))

$$\leq \mu_0 . \lambda_P^{PR}(\alpha)(k,a)$$

(by (83))

$$= \mu . \lambda_P^{PR}(\alpha)(k,a)$$

Proof of Claim.

To prove the claim, first notice that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (b) applied to α_1 and α_2 respectively, there exist $\mu_g, \mu_h \in \mathbb{N}^+$ such that

$$(\forall a \in A^u) \ (\lambda_P^{FP}(S_g)(a) \leq \mu_g . \lambda_P^{PR}(\alpha_1)(a)) \tag{84}$$

and

$$(\forall k \geq 0)(\forall a \in A^u)(\forall b \in A^v) \ (\lambda_P^{FP}(S_h)(k,a,b) \leq \mu_h . \lambda_P^{PR}(\alpha_2)(k,a,b)) \tag{85}$$

Also notice that by the definitions of G and P_G, the cost of evaluating each g_i is the cost of executing S_g for $i = 1,...,m$; that is,

$$(\forall a \in A^u) \ (g_i^{P_o}(a) = \lambda_P^{FP}(S_g)(a)) \tag{86}$$

for $i = 1,...,m$.

Similarly, we also have for $i = 1,...,m$

$$(\forall k \geq 0)(\forall a \in A^u)(\forall b \in A^v) \ (h_i^{P_o}(k,a,b) = \lambda_P^{FP}(S_h)(k,a,b)) \tag{87}$$

Take $\mu_0 = \max\{ 1+2 . \text{zero}^P . \mu_g, 1+\mu_h \}$ where $\text{zero}^P \geq 1$ is the complexity of evaluating $\text{zero} \in \Sigma_{\lambda,\tau}$ with respect to P. Also let $k \geq 0$, $a \in A^u$, and let $\rho \in States(A)$ satisfy $\pi_A(x)(\rho) = (k,a)$. We prove (83) holds for our choice of μ_0 by sub-induction on k as follows:

Sub-Basis. First we calculate the complexity of executing S_1 as follows:

$$\begin{aligned} \lambda_{P_o}^{PIT}(S_1)(\rho) &= \lambda_{P_o}^{PIT}(z, y_1,..., y_m := \text{zero}, g_1(x_1,..., x_n),..., g_m(x_1,..., x_n))(\rho) \\ &= \max\{ \lambda_{P_o}^{EXP}(\text{zero})(\rho), \lambda_{P_o}^{EXP}(g_1(x_1,..., x_n))(\rho),..., \lambda_{P_o}^{EXP}(g_m(x_1,..., x_n))(\rho) \} \end{aligned} \tag{88}$$

By definition of $\lambda_{P_o}^{EXP}$ we have

$$\begin{aligned} \lambda_{P_o}^{EXP}(g_i(x_1,..., x_n))(\rho) &= g_i^{P_o}(\rho(x_1),..., \rho(x_n)) + \max_{1 \leq i \leq n}\{ \lambda_{P_o}^{EXP}(x_i)(\rho) \} \\ &= g_i^{P_o}(\pi_A(x')(\rho)) + 1 \\ &= g_i^{P_o}(a) + 1 \\ &= \lambda_P^{FP}(S_g)(a) + 1 \end{aligned} \tag{89}$$

(by (86)).

Now, the cost of evaluating 'zero' is zero^P of course. Thus, from (88) we have

$$\begin{aligned} \lambda_{P_o}^{PIT}(S_1)(\rho) &= \max\{ \lambda_{P_o}^{EXP}(\text{zero})(\rho), \lambda_{P_o}^{EXP}(g_1(x_1,..., x_n))(\rho),..., \lambda_{P_o}^{EXP}(g_m(x_1,..., x_n))(\rho) \} \\ &= \max\{ \text{zero}^P, \lambda_P^{FP}(S_g)(a) + 1,..., \lambda_P^{FP}(S_g)(a) + 1 \} \end{aligned}$$

(using (89) with $i = 1,..., m$)

$$= \max\{ \text{zero}^P, \lambda_P^{FP}(S_g)(a) + 1 \} \tag{90}$$

Now, it is not difficult to see (using basic arithmetic) that:

$$\max\{ \text{zero}^P, \lambda_P^{FP}(S_g)(a) + 1 \} \leq 2 . \text{zero}^P . \lambda_P^{FP}(S_g)(a) \tag{91}$$

and

$$\max\{\,\mathrm{zero}^P, \lambda_P^{FP}(S_g)(a)+1\,\} \geq \lambda_P^{FP}(S_g)(a)+1 \tag{92}$$

We can now show the claim holds for $k=0$ as follows:

$$\lambda_{P_o}^{PIT}(S_1)(\rho) + \kappa(0,\rho) = \lambda_{P_o}^{PIT}(S_1)(\rho)+1$$

(since $\kappa(0,\rho)=1$ for any ρ)

$$= \max\{\,\mathrm{zero}^P, \lambda_P^{FP}(S_g)(a)+1\,\}+1$$

(by (90))

$$\leq 2.\mathrm{zero}^P.\lambda_P^{FP}(S_g)(a)+1$$

(by (91))

$$\leq 2.\mathrm{zero}^P.\mu_g.\lambda_P^{PR}(\alpha_1)(a)+1$$

(by the induction hypothesis (84))

$$\leq 2.\mathrm{zero}^P.\mu_g.\lambda_P^{PR}(\alpha_1)(a)+\lambda_P^{PR}(\alpha_1)(a)$$

(since $\lambda_P^{PR}(\alpha)(a)\geq 1$ for any arguments α and a).

$$= (1+2.\mathrm{zero}^P.\mu_g).\lambda_P^{PR}(\alpha_1)(a)$$

$$\leq \max\{\,1+2.\mathrm{zero}^P.\mu_g, 1+\mu_h\,\}.\lambda_P^{PR}(\alpha_1)(a)$$

$$= \mu_o.\lambda_P^{PR}(\alpha_1)(a)$$

(by definition of μ_o)

$$= \mu_o.\lambda_P^{PR}(*(\alpha_1,\alpha_2))(0,a)$$

$$= \mu_o.\lambda_P^{PR}(\alpha)(0,a)$$

Sub-Induction. Assume we have proved that for some fixed $k\geq 0$ and any $a\in A^u$, whenever $\rho\in States(A)$ is such that $\pi_A(x)(\rho)=(k,a)$, then

$$\lambda_{P_o}^{PIT}(S_1)(\rho)+\kappa(k,\rho)\leq \mu_o.\lambda_P^{PR}(\alpha)(k,a) \tag{93}$$

Now let $\rho\in States(A)$ be such that $\pi_A(x)(\rho)=(k+1,a)$. Then we must show that (83) holds for this state.

First notice that the complexity of S_1 under a state ρ is independent of the value of p (recall $x=(p,x')\in Var_{Tu}$ here) under ρ (see (88)), and thus if we define $\rho''\in States(A)$ by $\rho''=\rho\{\,k/p\,\}$ then

$$\lambda_{P_o}^{PIT}(S_1)(\rho'') = \lambda_{P_o}^{PIT}(S_1)(\rho) \tag{94}$$

The value of $\kappa(k,\rho)$ is also independent of the value of p under ρ (see (80)); thus

$$\kappa(k,\rho'') = \kappa(k,\rho) \tag{95}$$

Now, since $\pi_A(x)(\rho)=(k+1,a)$ we have $\pi_A(x)(\rho'')=(k,a)$, and so by the sub-induction hypothesis (93) we have:

$$\lambda_{P_o}^{PIT}(S_1)(\rho'')+\kappa(k,\rho'')\leq \mu_o.\lambda_P^{PR}(\alpha)(k,a)$$

and thus

$$\lambda_{P_o}^{PIT}(S_1)(\rho)+\kappa(k,\rho)\leq \mu_o.\lambda_P^{PR}(\alpha)(k,a) \tag{96}$$

by (94) and (95). We will use this fact (96) later.

Now consider of $\lambda_{P_o}^{PIT}(S_2)(\rho)$: by definition of S_2, we have for any $\rho \in States(A)$,

$$\lambda_{P_o}^{PIT}(S_2)(\rho) = \lambda_{P_o}^{PIT}(z, y_1, \ldots, y_m := \mathrm{succ}(z), e_1, \ldots, e_m)(\rho)$$
$$= \max\{\lambda_{P_o}^{EXP}(\mathrm{succ}(z))(\rho), \lambda_{P_o}^{EXP}(e_1)(\rho), \ldots, \lambda_{P_o}^{EXP}(e_m)(\rho)\} \quad (97)$$

Since 'z' is just a variable, and since the complexity of evaluating 'succ' is unity for any argument (by definition of a standard performance measure), it should be clear that

$$\lambda_{P_o}^{EXP}(\mathrm{succ}(z))(\rho) = 2 \quad (98)$$

Also, for $i = 1, \ldots, m$ we have

$$\lambda_{P_o}^{EXP}(e_i)(\rho) = \lambda_{P_o}^{EXP}(h_i(z, x_1, \ldots, x_n, y_1, \ldots, y_m))(\rho) + 1$$
$$= h_i^{P_o}(\rho(z), \rho(x_1), \ldots, \rho(x_n), \rho(y_1), \ldots, \rho(y_m)) + 1$$
$$= h_i^{P_o}(\rho(z), \pi_A(x')(\rho), \pi_A(y)(\rho)) + 1$$
$$= \lambda_P^{FP}(S_h)(\rho(z), \pi_A(x')(\rho), \pi_A(y)(\rho)) + 1 \quad (99)$$

Notice this formula (99) is independent of i.

From (97), the complexity of S_2 under an arbitrary state ρ is now:

$$\lambda_{P_o}^{PIT}(S_2)(\rho) = \max\{\lambda_{P_o}^{EXP}(\mathrm{succ}(z))(\rho), \lambda_{P_o}^{EXP}(e_1)(\rho), \ldots, \lambda_{P_o}^{EXP}(e_m)(\rho)\}$$
$$= \max\{2, \lambda_{P_o}^{EXP}(e_1)(\rho), \ldots, \lambda_{P_o}^{EXP}(e_m)(\rho)\}$$

(from (98))

$$= \max\{2, \lambda_P^{FP}(S_h)(\rho(z), \pi_A(x')(\rho), \pi_A(y)(\rho)) + 1\}$$

(using (99) with $i = 1, \ldots, m$)

$$= 1 + \max\{1, \lambda_P^{FP}(S_h)(\rho(z), \pi_A(x')(\rho), \pi_A(y)(\rho))\}$$
$$= 1 + \lambda_P^{FP}(S_h)(\rho(z), \pi_A(x')(\rho), \pi_A(y)(\rho)) \quad (100)$$

Note that this formula (100) holds for arbitrary states ρ.

Now recall the previous claim concerning the state ρ_k (see equations (63) - (65)): if ρ is an initial state with $\pi_A(x')(\rho) = a$, then under ρ_k the value of z is k, and moreover, $\pi_A(x')(\rho_k) = a$, and $\pi_A(y)(\rho_k) = [\![\alpha]\!]_A(k, a)$. Thus, from (100) we have

$$\lambda_{P_o}^{PIT}(S_2)(\rho_k) = 1 + \lambda_P^{FP}(S_h)(k, a, [\![\alpha]\!]_A(k, a)) \quad (101)$$

To complete the sub-induction step we calculate as follows:

$$\lambda_{P_o}^{PIT}(S_1)(\rho) + \kappa(k+1, \rho) = \lambda_{P_o}^{PIT}(S_1)(\rho) + \kappa(k, \rho) + \lambda_{P_o}^{PIT}(S_2)(\rho_k)$$

(by definition of κ)

$$\leq \mu_o \cdot \lambda_P^{PR}(\alpha)(k, a) + \lambda_{P_o}^{PIT}(S_2)(\rho_k)$$

(by (96))

$$= \mu_o \cdot \lambda_P^{PR}(\alpha)(k, a) + 1 + \lambda_P^{FP}(S_h)(k, a, [\![\alpha]\!]_A(k, a))$$

(from (101))

$$= \mu_o \cdot \lambda_P^{PR}(\alpha)(k, a) + 1 + \mu_h \cdot \lambda_P^{PR}(\alpha_2)(k, a, [\![\alpha]\!]_A(k, a))$$

(by the induction hypothesis (85))

$$\leq \mu_0 . \lambda_P^{PR}(\alpha)(k,a) + (1 + \mu_h) . \lambda_P^{PR}(\alpha_2)(k,a,[\![\alpha]\!]_A(k,a))$$

(since $\lambda_P^{PR}(\alpha)(a) \geq 1$ for any arguments α and a)

$$\leq \mu_0 . (\lambda_P^{PR}(\alpha)(k,a) + \lambda_P^{PR}(\alpha_2)(k,a,[\![\alpha]\!]_A(k,a)))$$

(by definition of μ_0)

$$= \mu_0 . \lambda_P^{PR}(*(\alpha_1,\alpha_2))(k+1,a)$$

$$= \mu_0 . \lambda_P^{PR}(\alpha)(k+1,a)$$

□

We must now show that $c^{PR}(\alpha)$ satisfies the second performance preservation condition, namely (6) above.

First, from (82) we have

$$\lambda_P^{FP}(c^{PR}(\alpha))(k,a) = \lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \tag{102}$$

where $\rho \in States(A)$ is any state such that $\pi_A(x)(\rho) = (k,a)$.

Claim. Let $k \geq 0$ and $a \in A^u$. Then for every $\rho \in States(A)$ with $\pi_A(x)(\rho) = (k,a)$,

$$\lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \geq \lambda_P^{PR}(\alpha)(k,a) \tag{103}$$

Of course, (6) easily follows from (102) and (103), thus it remains to prove the claim:

Proof of Claim. First notice that since α_1 and α_2 are of less structural complexity than α, by the induction hypothesis (c) applied to α_1 and α_2 respectively, we have

$$(\forall a \in A^u)\ (\lambda_P^{PR}(\alpha_1)(a) \leq \lambda_P^{FP}(S_g)(a)) \tag{104}$$

and

$$(\forall k \geq 0)(\forall a \in A^u)(\forall b \in A^v)\ (\lambda_P^{PR}(\alpha_2)(k,a,b) \leq \lambda_P^{FP}(S_h)(k,a,b)) \tag{105}$$

Now let $k \geq 0$ and $a \in A^u$, and let $\rho \in States(A)$ satisfy $\pi_A(x)(\rho) = (k,a)$. Then we will prove (103) holds by sub-induction on k as follows:

Sub-Basis. For $k = 0$ we calculate as follows:

$$\lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(0,\rho) = \lambda_{P_e}^{PIT}(S_1)(\rho) + 1$$

$$> \lambda_{P_e}^{PIT}(S_1)(\rho)$$

$$= \max\{\text{zero}^P, \lambda_P^{FP}(S_g)(a) + 1\}$$

(by (90))

$$\geq \lambda_P^{FP}(S_g)(a) + 1$$

(by (92))

$$\geq \lambda_P^{PR}(\alpha_1)(a) + 1$$

(by the induction hypothesis (104))

$$> \lambda_P^{PR}(\alpha_1)(a)$$

$$= \lambda_P^{PR}(*(\alpha_1,\alpha_2))(0,a)$$

$$= \lambda_P^{PR}(\alpha)(0,a)$$

Sub-Induction. Let us assume that for some fixed $k \geq 0$ and any $a \in A^u$, that whenever $\rho \in States(A)$ is such that $\pi_A(x)(\rho) = (k,a)$, then

$$\lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \geq \lambda_P^{PR}(\alpha)(k,a) \tag{106}$$

Now let $\rho \in States(A)$ be such that $\pi_A(x)(\rho) = (k+1,a)$. Then we must show that (103) holds for this state.

Similar to the proof of the previous claim, notice that the complexity of S_1 under a state ρ is independent of the value of p under ρ, and so is the value of $\kappa(k,\rho)$ is equally independent.

Thus if we define $\rho'' \in States(A)$ by $\rho'' = \rho\{k/p\}$ then since $\pi_A(x)(\rho'') = (k,a)$, by the sub-induction hypothesis (106) we have:

$$\lambda_{P_e}^{PIT}(S_1)(\rho'') + \kappa(k,\rho'') \geq \lambda_P^{PR}(\alpha)(k,a)$$

and thus

$$\lambda_{P_e}^{PIT}(S_1)(\rho) + \kappa(k,\rho) \geq \mu_0 . \lambda_P^{PR}(\alpha)(k,a) \tag{107}$$

We can now complete the sub-induction as follows:

$$\begin{aligned}
\lambda_{P_e}^{PIT}(S_1) + \kappa(k+1,\rho) &= \lambda_{P_e}^{PIT}(S_1) + \kappa(k,\rho) + \lambda_{P_e}^{PIT}(S_2)(\rho_k) \\
&\geq \lambda_P^{PR}(\alpha)(k,a) + \lambda_{P_e}^{PIT}(S_2)(\rho_k)
\end{aligned}$$

(by (107))

$$= \lambda_P^{PR}(\alpha)(k,a) + 1 + \lambda_P^{FP}(S_h)(k,a,[\![\alpha]\!]_A(k,a))$$

(by (101))

$$\geq \lambda_P^{PR}(\alpha)(k,a) + 1 + \lambda_P^{PR}(\alpha_2)(k,a,[\![\alpha]\!]_A(k,a))$$

(by the induction hypothesis (105))

$$\begin{aligned}
&= 1 + \lambda_P^{PR}(*(\alpha_1,\alpha_2))(k+1,a) \\
&= 1 + \lambda_P^{PR}(\alpha)(k+1,a) \\
&> \lambda_P^{PR}(\alpha)(k+1,a)
\end{aligned}$$

□

Discussion. The preceding theorem quantifies over all (standard) signatures Σ, all (standard) Σ-algebras A, and all (standard) performance measures P. If we denote c^{PR} by $c^{PR}(\Sigma)$ to explicitly name the underlying signature, then $c^{PR}(\underline{\Sigma})$ is a correct and performance preserving compiler from PR($\underline{\Sigma}$) into FPIT($\underline{\Sigma}$); correct with respect to $[\![\,.\,]\!]_{\underline{A}}$ and $F_{\underline{A}}$, and performance preserving with respect to λ_Q^{PR} and λ_Q^{FP} when Q is a performance measure for $\underline{A}$. Since PR($\underline{\Sigma}$) is a language in which we can formalise any synchronous algorithm whose modules determine the Σ-algebra A, the preceding theorem guarantees that any such algorithm can be (correctly) simulated in FPIT($\underline{\Sigma}$) by a function program with performance equivalent to the performance of the original algorithm. This completes one half of our equivalence conjecture of course.

7.3 COMPILING FUNCTION PROGRAMS.

It remains to establish that every function program can be transformed into a PR scheme whose performance is equivalent to the original program. Before we can explain how this transformation is to be accomplished, we introduce one new compiler and revisit an old one. In Sections 7.3.1-7.3.6 we will prove the existence of a correct and performance preserving compiler c^{IO} : PITIO(Σ)$\longrightarrow$PR(Σ); as we will see later, c^{IO} will be used for compiling the body of a given function program into PR. In Section 7.3.7 we revisit the transformation c of Definition 3.5.6; as we will see later, our strategy for defining c^{FP} relies on the existence of compilers from PR($sig\,(G)$) into PR(Σ) for each $G \in$ FG(Σ); we will use c to construct precisely such compilers.

After defining and verifying c^{IO} we establish some useful technical results concerning c, and then in Section 7.3.11 we explain how c^{FP} is to be constructed using c^{IO} and c.

7.3.1 Compiling Functional Expressions.

Shortly, we will define c^{IO} which maps PIT i/o-programs into PR; first however, we need to be able to transform expressions (from which programs are built) into PR:

Definition. We define c^{EXP} to be the $S^+\times S$-indexed family:

$$c^{EXP} = <c^{EXP}_{w,s} : w \in S^+, s \in S>$$

of mappings $c^{EXP}_{w,s}$: FEXP(Σ)$_{w,s}\longrightarrow$PR(Σ)$_{w,s}$. Each $c^{EXP}_{w,s}$ (ambiguously denoted c^{EXP}) is defined uniformly in w and s and by induction on the structural complexity of the expression part of arguments (e,X) as follows:

Let $(e,X) \in$ FEXP(Σ)$_{w,s}$ for some $w \in S^+$ and $s \in S$. We will first construct $c^{EXP}(e,X)$ and then establish that it is well-defined member of PR(Σ)$_{w,s}$.

Basis Cases.

(i) *Constants.* Suppose $e = c$ for some $c \in \Sigma_{\lambda,s}$.

Construction. In this case we define $c^{EXP}(e,X)$ by:

$$c^{EXP}(e,X) = c^w$$

Well-Definedness. In this case it is easy to see that $c^{EXP}(e,X) \in$ PR(Σ)$_{w,s}$ (See clause (i) of Definition 3.3.1.)

(ii) *Variables.* Suppose $e = x$ for some $x \in Var_s$.

Construction. In this case we define $c^{EXP}(e,X)$ by:

$$c^{EXP}(e,X) = U^w_i$$

where i is is the least k such that $X_k = x$.

Well-Definedness. First notice that since $(e,X) \in$ FEXP(Σ), we have $var(X) \supseteq var(e) = \{x\}$. Thus $x \in var(X)$, and so the number i is well-defined; that is $x = X_i$ for some $i \in [1,n]$ when $n = |w|$. Now, $x = X_i$ and so the sorts of these two variables must be the same. Since $x \in Var_s$ by hypothesis, the sort of x is s. Also, $X \in Var_w$ by hypothesis, and so the sort of X_i is w_i. Thus $w_i = s$ and so

$$c^{EXP}(e,X) = U_i^w \in PR(\Sigma)_{w,w_i} = PR(\Sigma)_{w,s}$$

as required. (Refer to the projection clause of Definition 3.3.1).

Induction. Let $e \in EXP(\Sigma)_s$ be some fixed expression. Suppose for every $s' \in S$ and for every $(e',X) \in FEXP(\Sigma)_{w,s'}$, where e' is of less structural complexity than e, that we have constructed $c^{EXP}(e',X) \in PR(\Sigma)_{w,s'}$.

We will now construct $c^{EXP}(e,X) \in PR(\Sigma)_{w,s}$.

(iii) *Operations.* Without loss of generality we may assume that e is of the form: $e = \sigma(e_1,...,e_n)$ where for some $u \in S^+, \sigma \in \Sigma_{u,s}$, and $e_i \in FEXP(\Sigma)_{u_i}$ for $i = 1,...,n = |u|$.

Construction. Here we define $c^{EXP}(e,X)$ by:

$$c^{EXP}(e,X) = \sigma \circ < c^{EXP}(e_1,X),...,c^{EXP}(e_n,X) >$$

Well-Definedness. First notice that since $(e,X) \in FEXP(\Sigma)$ it is easy to show (as in the proof of Lemma 6.2.15) that $(e_i,X) \in FEXP(\Sigma)_{w,\mu_i}$ for $i = 1,...,n$. Now, $e_1,...,e_n$ are all less structurally complex than e, and so by the induction hypothesis applied to (e_i,X) for $i = 1,...,n$, we have $c^{EXP}(e_i,X) \in PR(\Sigma)_{w,\mu_i}$ for $i = 1,...,n$. Thus

$$< c^{EXP}(e_1,X),...,c^{EXP}(e_n,X) > \in PR(\Sigma)_{w,\mu}$$

and hence, since $\sigma \in PR(\Sigma)_{u,s}$, we have $c^{EXP}(e,X) \in PR(\Sigma)_{w,s}$ as required. (Refer to the composition clause of Definition 3.3.1). □

7.3.2 Lemma. *For every $w \in S^+$ and $s \in S$ and for every $(e,X) \in FEXP(\Sigma)_{w,s}$,*

$$(\forall \rho \in States(A))\ (\ [\![c^{EXP}_{w,s}(e,X)]\!]_A(\pi_A(X)(\rho)) = E_A(e)(\rho)\ ) \qquad (108)$$

Furthermore,

$$(\forall \rho \in States(A))\ (\ \lambda^{PR}_\rho(c^{EXP}_{w,s}(e,X))(\pi_A(X)(\rho)) = \lambda^{EXP}_\rho(e)(\rho)\ ) \qquad (109)$$

Note. Technically, c^{EXP} is not a compiler since $FEXP(\Sigma)$ does not have a functional semantics (since the meaning of an expression is defined in terms of states). However, $FEXP(\Sigma)$ is an $S^+ \times S^+$-indexed family of sets (with $FEXP(\Sigma)_{u,v} = \emptyset$ for $|v| \neq 1$), and c^{EXP} is word-indexed (since $c^{EXP}_{w,s}: FEXP(\Sigma)_{w,s} \longrightarrow PR(\Sigma)_{w,s}$ for each $w \in S^+$ and $s \in S$), so we will refer to c^{EXP} as a 'compiler', and we will refer to (108) and (109) as 'correctness' and 'performance preservation' conditions, Definitions 7.1.1 notwithstanding.

Proof of Lemma 7.3.2. Choose $e \in EXP(\Sigma)_s$ for some $s \in S$ and $X \in Var_w$ for some $w \in S^+$. We prove (108) and (109) uniformly in s by induction on the structural complexity of e. Throughout the proof we abbreviate $c^{EXP}_{w,s}$ by c^{EXP}.

Basis Cases.

(i) *Constants.* If $e = c$ for some $c \in \Sigma_{\lambda,s}$, then $c^{EXP}(e,X) = c^w$.

Correctness. Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}[\![c^{EXP}(e,X)]\!]_A(\pi_A(X)(\rho)) &= c^A \\ &= E_A(c)(\rho) \\ &= E_A(e)(\rho)\end{aligned}$$

Thus (108) holds as claimed.

Performance Preservation. Again choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}\lambda_P^{PR}(c^{EXP}(e,X))(\pi_A(X)(\rho)) &= c^P \\ &= \lambda_P^{EXP}(c)(\rho) \\ &= \lambda_P^{EXP}(e)(\rho)\end{aligned}$$

Thus (109) holds as claimed.

(ii) *Variables.* If $e = x$ for some $x \in Var_s$, then $c^{EXP}(e,X) = U_i^w$ where i is the least k such that $x = X_k$.

Correctness. Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}[\![c^{EXP}(e,X)]\!]_A(\pi_A(X)(\rho)) &= [\![U_i^w]\!]_A(\rho(X_1),\ldots,\rho(X_n)) \\ &= \rho(X_i) \\ &= \rho(x) \\ &= E_A(x)(\rho) \\ &= E_A(e)(\rho)\end{aligned}$$

Thus (108) holds as claimed.

Performance Preservation. Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}\lambda_P^{PR}(c^{EXP}(e,X))(\pi_A(X)(\rho)) &= 1 \\ &= \lambda_P^{EXP}(x)(\rho) \\ &= \lambda_P^{EXP}(e)(\rho)\end{aligned}$$

Thus (109) holds as claimed.

Induction. Suppose that $(e,X) \in \mathrm{FEXP}(\Sigma)_{w,s}$ is some fixed functional expression with the property that for each $s' \in S$ and every $(e',X) \in \mathrm{FEXP}(\Sigma)_{w,s'}$ with e' is of less structural complexity than e, that we have proved:

$$(\forall \rho \in States(A))\ (\ [\![\alpha']\!]_A(\pi_A(X)(\rho)) = E_A(e')(\rho)\ ) \tag{110}$$

and

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PR}(\alpha')(\pi_A(X)(\rho)) = \lambda_P^{EXP}(e')(\rho)\ ) \tag{111}$$

where $\alpha' = c^{EXP}(e',X) \in \mathrm{PR}(\Sigma)_{w,s'}$.

We now show that (108) and (109) hold for (e,X).

(iii) *Operations.* Without loss of generality, we may suppose e is of the form $e = \sigma(e_1,\ldots,e_n)$, where for some $u \in S^+$, $\sigma \in \Sigma_{u,s}$, and $e_i \in \mathrm{EXP}(\Sigma)_{u_i}$ for $i = 1,\ldots,n = |u|$.

Now, for $i = 1,\ldots,n$, since each e_i is of less structural complexity than e, by the induction hypotheses (110) and (111) applied to (e_i, X) we have

$$(\forall \rho \in States(A))\ (\ [\![\alpha_i]\!]_A(\pi_A(X)(\rho)) = E_A(e_i)(\rho)\ ) \tag{112}$$

and

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PR}(\alpha_i)(\pi_A(X)(\rho)) = \lambda_P^{EXP}(e_i)(\rho)\ ) \tag{113}$$

where $\alpha_i = c^{EXP}(e_i, X) \in \mathrm{PR}(\Sigma)_{w,\mu_i}$ for $i = 1,\ldots,n$.

Correctness. To show that c^{EXP} is correct, choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}
[\![c^{EXP}(e, X)]\!]_A(\pi_A(X)(\rho)) &= [\![\sigma \circ <\alpha_1,\ldots,\alpha_n>]\!]_A(\pi_A(X)(\rho)) \\
&= \sigma^A([\![<\alpha_1,\ldots,\alpha_n>]\!]_A(\pi_A(X)(\rho))) \\
&= \sigma^A([\![\alpha_1]\!]_A(\pi_A(X)(\rho)),\ldots,[\![\alpha_n]\!]_A(\pi_A(X)(\rho))) \\
&= \sigma^A(E_A(e_1)(\rho),\ldots,E_A(e_n)(\rho))
\end{aligned}$$

(using (112) with $i = 1,\ldots,n$)

$$\begin{aligned}
&= E_A(\sigma(e_1,\ldots,e_n)(\rho)) \\
&= E_A(e)(\rho)
\end{aligned}$$

Thus (108) holds as claimed.

Performance Preservation. Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}
\lambda_P^{PR}(c^{EXP}(e, X))(\pi_A(X)(\rho)) &= \lambda_P^{PR}(\sigma \circ <\alpha_1,\ldots,\alpha_n>)(\pi_A(X)(\rho)) \\
&= \sigma^P([\![<\alpha_1,\ldots,\alpha_n>]\!]_A(\pi_A(X)(\rho))) \\
&\quad + \max\{\lambda_P^{PR}(\alpha_1)(\pi_A(X)(\rho)),\ldots,\lambda_P^{PR}(\alpha_n)(\pi_A(X)(\rho))\}
\end{aligned}$$

$$\begin{aligned}
&= \sigma^P([\![\alpha_1]\!]_A(\pi_A(X)(\rho)),\ldots,[\![\alpha_n]\!]_A(\pi_A(X)(\rho))) \\
&\quad + \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_n)(\rho)\}
\end{aligned}$$

(using (113) with $i = 1,\ldots,n$)

$$\begin{aligned}
&= \sigma^P(E_A(e_1)(\rho),\ldots,E_A(e_n)(\rho)) \\
&\quad + \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_n)(\rho)\}
\end{aligned}$$

(using (112) with $i = 1,\ldots,n$)

$$= \lambda_P^{EXP}(\sigma(e_1,\ldots,e_n))(\rho)$$

$$= \lambda_P^{EXP}(e)(\rho)$$

Thus (109) holds as claimed. □

7.3.3 Compiling I/O-programs.

We can now define the compiler c^{IO} : PITIO(Σ) $\longrightarrow$ PR(Σ) mentioned above.

Definition. We define c^{IO} to be the $S^+ \times S^+$-indexed family

$$c^{IO} = < c^{IO}_{u,v} : u,v \in S^+ >$$

of mappings $c^{IO}_{u,v}$: PITIO(Σ)$_{u,v}$ $\longrightarrow$ PR(Σ)$_{u,v}$. For each $u,v \in S^+$, $c^{IO}_{u,v}$ (ambiguously denoted c^{IO}) is defined uniformly in u, v, *In*, and *Out*, and by induction on the structural complexity of the program part of arguments (S,*In*,*Out*) as follows:

Choose (S,*In*,*Out*) $\in$ PITIO(Σ)$_{u,v}$ for some $u,v \in S^+$. We will first construct c^{IO}(S,*In*,*Out*) and then prove it is a well-defined member of PR(Σ)$_{u,v}$ (thus establishing that c^{IO} is word-indexed and hence a compiler from PITIO(Σ) into PR(Σ)).

Basis Case.

(i) *Multiple Assignment Statements.* Suppose S is of the form:

$$S = x_1,...,x_r := e_1,...,e_r$$

for some $r > 0$.

Construction. Let β be the scheme defined by

$$\beta = < \alpha_1,...,\alpha_r, U^u_1,...,U^u_n >$$

where $\alpha_i = c^{EXP}(e_i, In)$ for $i = 1,...,r$ (and $n = |u|$).

Also, let γ be the scheme defined by

$$\gamma = < \gamma_1,...,\gamma_m >$$

where for $i = 1,...,m$,

$$\gamma_i = \begin{cases} U^{wu}_j & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ U^{wu}_{r+k} & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases} \qquad (114)$$

wherein $w \in S^+$ is such that $(x_1,...,x_r) \in Var_w$ (so $|w| = r$).

In this case we define c^{IO}(S,*In*,*Out*) by

$$c^{IO}(S,In,Out) = \gamma \circ \beta$$

Well-definedness. We must show $\gamma \circ \beta \in$ PR(Σ)$_{u,v}$. To do this, we will show $\beta \in$ PR(Σ)$_{u,wu}$ and $\gamma \in$ PR(Σ)$_{wu,v}$:

To see that $\beta \in$ PR(Σ)$_{u,wu}$, first notice that exactly as in the basis case of the proof of Theorem 6.2.13, it is easy to show that $(e_i, In) \in$ FEXP(Σ)$_{u,w_i}$, and so $c^{EXP}(e_i, In) \in$ PR(Σ)$_{u,w_i}$ for $i = 1,...,r$. Now, $U^u_i \in$ PR(Σ)$_{u,u_i}$ for $i = 1,...,n$; thus all the coordinate schemes of β have the same domain type u, and they are all single-valued schemes, so certainly β is a legal vectorisation. Moreover, it is not difficult to see that the codomain type of β is wu, that is, $\beta \in$ PR(Σ)$_{u,wu}$ as claimed.

We will now show $\gamma \in$ PR(Σ)$_{wu,v}$. First recall from the proof of Theorem 6.2.13, that for $i = 1,...,m$ if $Out_i \neq x_j$ for any $j \in [1,r]$ then there is some $k \in [1,n]$ such that $Out_i = In_k$, and hence each scheme is unambiguously defined by (114) above.

Now, notice that every coordinate scheme of γ has common domain type wu, and each is a projection scheme and therefore single-valued. Thus γ is certainly a legal vectorisation with domain type wu. To show that $\gamma \in PR(\Sigma)_{wu,v}$ we must show $\gamma_i \in PR(\Sigma)_{wu,v_i}$ for $i = 1,\ldots,m$.

Choose $i \in \{1,m\}$. We show $\gamma_i \in PR(\Sigma)_{u,v_i}$ in each of the following two cases:

Case 1: $Out_i = x_j$ for some $j \in [1,r]$
Case 2: $Out_i \neq x_j$ for any $j \in [1,r]$

Case 1. First suppose $Out_i = x_j$ for some $j \in [1,r]$. Then $\gamma_i = U_j^{wu}$. Now, since $1 \leq j \leq r$ and $|w| = r$, we have $\gamma_i \in PR(\Sigma)_{wu,w_j}$ by definition of w. However, $x_j = Out_i$, and so the sort of x_j is the same as that of Out_i: the sort of x_j is w_j and the sort of Out_i is v_i; thus $w_j = v_i$ and $\gamma_i \in PR(\Sigma)_{wu,v_i}$ as required.

Case 2. Now suppose $Out_i \neq x_j$ for any $j \in [1,r]$. Then $Out_i = In_k$ for some $k \in [1,n]$, and so γ_i is defined to be the projection U_{r+k}^{wu}; thus $\gamma_i \in PR(\Sigma)_{wu,u_k}$ (since $|w| = r$). However, if $Out_i = In_k$ then these variables must be of the same sort; the sort of Out_i is v_i and the sort of In_k is u_k, and thus $v_i = u_k$, and so $\gamma_i \in PR(\Sigma)_{wu,v_i}$ as required.

Induction. Let $S \in PIT(\Sigma)$ be some fixed program. Suppose for every $u',v' \in S^+$ and every $(S',In',Out') \in PITIO(\Sigma)_{u',v'}$ where S' is of less structural complexity than S, that we have constructed $c^{IO}(S',In',Out')$ and established that it is a well-defined member of $PR(\Sigma)_{u',v'}$.

We will now define $c^{IO}(S,In,Out) \in PR(\Sigma)_{u,v}$ for any $u,v \in S^+$ and for any $In \in Var_u$ and $Out \in Var_v$ according to the three following possible cases:

(ii) *Sequencing.* Suppose $S = S_1;S_2$ for some $S_1,S_2 \in PIT(\Sigma)$.

Construction. Let $In_1 = In$, and let Out_1 be any vector of distinct variables such that $var(Out_1) = var(S_1) \cup var(In_1)$. Also let $In_2 = Out_1$ and $Out_2 = Out$.

In this case we define $c^{IO}(S,In,Out)$ by

$$c^{IO}(S,In,Out) = c^{IO}(S_2,In_2,Out_2) \circ c^{IO}(S_1,In_1,Out_1)$$

We claim $c^{IO}(S,In,Out) \in PR(\Sigma)_{u,v}$:

Well-Definedness. To show $c^{IO}(S,In,Out) \in PR(\Sigma)_{u,v}$, we must show that $c^{IO}(S_1,In_1,Out_1) \in PR(\Sigma)_{u,w}$ and $c^{IO}(S_2,In_2,Out_2) \in PR(\Sigma)_{w,v}$ for some $w \in S^+$.

First notice that we can show $(S_i,In_i,Out_i) \in PITIO(\Sigma)$ for $i = 1,2$ exactly as in case (ii) of the proof of Theorem 6.2.13. Since $(S_i,In_i,Out_i) \in PITIO(\Sigma)$ and S_i is of less structural complexity than S for $i = 1,2$, by the induction hypothesis applied to (S_1,In_1,Out_1), we have that $c^{IO}(S_1,In_1,Out_1) \in PR(\Sigma)_{u,w}$ (where $w \in S^+$ is such that $Out_1 \in Var_w$), and by the induction hypothesis applied to (S_2,In_2,Out_2), we have $c^{IO}(S_2,In_2,Out_2) \in PR(\Sigma)_{w,v}$. Thus $c^{IO}(S,In,Out) \in PR(\Sigma)_{u,v}$ as claimed. (Refer to the composition clause of Definition 3.3.1.)

(iii) *Conditional.* Suppose S is of the form:

$$S = \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

for some $b \in \mathrm{EXP}(\Sigma)_{\mathbf{B}}$, and some $S_1, S_2 \in \mathrm{PIT}(\Sigma)$.

Construction. In this case we define $c^{IO}(S, In, Out)$ by

$$c^{IO}(S, In, Out) = \mathrm{DC}(c^{EXP}(b, In), c^{IO}(S_1, In, Out), c^{IO}(S_2, In, Out))$$

We claim $c^{IO}(S, In, Out) \in \mathrm{PR}(\Sigma)_{u,v}$:

Well-Definedness. First notice that exactly as in case (iii) of the proof of Theorem 6.2.13, we can show $(S_i, In, Out) \in \mathrm{PITIO}(\Sigma)$ for $i = 1,2$. Since $(S_i, In, Out) \in \mathrm{PITIO}(\Sigma)$ for $i = 1,2$ with S_i of less structural complexity than S for $i = 1,2$, by the induction hypothesis applied to (S_i, In, Out), we have $c^{IO}(S_i, In, Out) \in \mathrm{PR}(\Sigma)_{u,v}$ for $i = 1,2$. Furthermore, since $c^{EXP}(b, In) \in \mathrm{PR}(\Sigma)_{u,\mathbf{B}}$, we have $c^{IO}(S, In, Out) \in \mathrm{PR}(\Sigma)_{u,v}$ as claimed. (Refer to the definition-by-cases clause of Definition 3.3.1.)

(iv) *Bounded Iteration.* Suppose S is of the form:

$$S = \textbf{do } e \textbf{ times } S_0 \textbf{ od}$$

for some $e \in \mathrm{EXP}(\Sigma)_{\mathbf{T}}$ and some $S_0 \in \mathrm{PIT}(\Sigma)$.

Construction. In this final case we define $c^{IO}(S, In, Out)$ by:

$$c^{IO}(S, In, Out) = \beta \circ \gamma \circ \delta$$

where:

$$\beta = \langle U^u_{\theta(1)}, \ldots, U^u_{\theta(m)} \rangle$$

where $\theta(j)$ is the least k such that $Out_j = In_k$;

$$\gamma = it(c^{IO}(S_0, In, In))$$

and

$$\delta = \langle c^{EXP}(e, In), U^u_1, \ldots, U^u_n \rangle$$

(Recall the definition of the derived composition tool $it(\alpha)$ from Section 3.3.4.)

Well-Definedness. First notice that exactly as in case (iv) of the proof of Theorem 6.2.13, we can show $(e, In) \in \mathrm{FEXP}(\Sigma)_{u,\mathbf{T}}$ and so $c^{EXP}(e, In)$ is a well-defined member of $\mathrm{PR}(\Sigma)_{u,\mathbf{T}}$. (Refer to Definition 7.3.1.)

Also as in the proof of Theorem 6.2.13, we can show $(S_0, In, In) \in \mathrm{PITIO}(\Sigma)$, and since S_0 is of less structural complexity than S, by the induction hypothesis applied to (S_0, In, In), we have $\alpha_0 = c^{IO}(S_0, In, In) \in \mathrm{PR}(\Sigma)_{u,\mu}$.

Now notice that since $\alpha_0 \in \mathrm{PR}(\Sigma)_{u,\mu}$, we have $\gamma \in \mathrm{PR}(\Sigma)_{\tau u,\mu}$. Also $c^{EXP}(e, In) \in \mathrm{PR}(\Sigma)_{u,\mathbf{T}}$ and so it is easy to see that $\delta \in \mathrm{PR}(\Sigma)_{u,\tau u}$. Thus $\gamma \circ \delta \in \mathrm{PR}(\Sigma)_{u,\mu}$. Now, the domain of $[\![\beta]\!]_A$ is certainly A^u, and so to show $c^{IO}(S, In, Out) \in \mathrm{PR}(\Sigma)_{u,v}$ we must show that $\beta \in \mathrm{PR}(\Sigma)_{u,v}$; that is, that the codomain of $[\![\beta]\!]_A$ is A^v. Since the jth coordinate of β is $U^u_{\theta(j)}$, this means we must show $u_{\theta(j)} = v_j$ for $j = 1,\ldots,m$:

First notice that since S is a loop, we have $free(S) = var(S)$ and so $var(Out) \subseteq var(In)$ (since $var(Out) \subseteq var(S) \cup var(In)$), and hence $Out_j \in var(In)$ for $j = 1,\ldots,m$; that is, the number $\theta(j)$ is well-defined for $j = 1,\ldots,m$. Now, for $j = 1,\ldots,m$, $Out_j = In_{\theta(j)}$ by definition of θ, and so the sorts of these

variables are the same. Thus $v_j = u_{\theta(j)}$ for $j = 1,\ldots,m$, and hence $\beta \in \mathrm{PR}(\Sigma)_{u,v}$ as required. □

We now state and prove a result from which it is easy to derive that c^{IO} is a correct and performance preserving compiler:

7.3.4 Theorem. *For any $u,v \in S^+$ and for every $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$,*

$$(\forall \rho \in States(A))\ (\ [\![c^{IO}_{u,v}(S,In,Out)]\!]_A(\pi_A(In)(\rho)) = \pi_A(Out)(M_A(S)(\rho))\ ) \qquad (115)$$

Furthermore, there exists $\mu \in \mathbb{N}^+$ such that

$$(\forall \rho \in States(A))\ (\ \lambda^{PR}_p(c^{IO}_{u,v}(S,In,Out))(\pi_A(In)(\rho)) \le \mu \cdot \lambda^{PIT}_p(S)(\rho)\ ) \qquad (116)$$

Additionally,

$$(\forall \rho \in States(A))\ (\ \lambda^{PIT}_p(S)(\rho) \le \lambda^{PR}_p(c^{IO}_{u,v}(S,In,Out))(\pi_A(In)(\rho))\ ) \qquad (117)$$

Proof. Let $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$. We prove (115), (116), and (117) simultaneously and uniformly in u and v, and by induction on the structural complexity of S. Throughout, we abbreviate $c^{IO}_{u,v}$ by c^{IO}.

Basis Case.

(i) *Multiple Assignment Statements.* Suppose S is of the form:

$$S = x_1,\ldots,x_r := e_1,\ldots,e_r$$

for some $r > 0$. Then $\alpha = c^{IO}(S,In,Out)$ is defined by:

$$\alpha = \gamma \circ \beta$$

where

$$\beta = \langle \alpha_1,\ldots,\alpha_r,U^u_1,\ldots,U^u_n \rangle$$

where for $i = 1,\ldots,r$, $\alpha_i = c^{EXP}(e_i,In)$, and where γ is the scheme defined by

$$\gamma = \langle \gamma_1,\ldots,\gamma_m \rangle$$

where for $i = 1,\ldots,m$,

$$\gamma_i = \begin{cases} U^{wu}_j & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ U^{wu}_{r+k} & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases}$$

and where $w \in S^+$ is such that $(x_1,\ldots,x_r) \in Var_w$.

Correctness. To show that c^{IO} correctly compiles multiple assignment statements we must show

$$[\![\alpha]\!]_A(\pi_A(In)(\rho)) = \pi_A(Out)(M_A(S)(\rho)) \qquad (118)$$

for every $\rho \in States(A)$.

First let us consider β. For any $\rho \in States(A)$ we calculate as follows:

$$\begin{aligned} [\![\beta]\!]_A(\pi_A(In)(\rho)) &= [\![\langle \alpha_1,\ldots,\alpha_r,U^u_1,\ldots,U^u_n \rangle]\!]_A(\pi_A(In)(\rho)) \\ &= ([\![\alpha_1]\!]_A(\pi_A(In)(\rho)),\ldots,[\![\alpha_r]\!]_A(\pi_A(In)(\rho)),\rho(In_1),\ldots,\rho(In_n)) \end{aligned}$$

(since $[\![U^u_i]\!]_A(\pi_A(In)(\rho)) = \rho(In_i)$ for $i = 1,\ldots,n$)

$$= (E_A(e_1)(\rho),\ldots,E_A(e_r)(\rho),\rho(In_1),\ldots,\rho(In_n))$$

This last step follows since for $i = 1,\ldots,r$, $\alpha_i = c^{EXP}(e_i,In)$ and c^{EXP} is a correct compiler. (See Lemma 7.3.2.) Thus we have shown:

$$(\forall \rho \in States(A))\ (\ [\![\beta]\!]_A(\pi_A(In)(\rho)) = ((E_A(e_1)(\rho),....,E_A(e_r),\rho(In_1),....,\rho(In_n))\ ) \qquad (119)$$

Now let us consider schemes γ_i for $i = 1,...,m$. Let $a \in A^w$ and $b \in A^u$. Then we calculate

$$[\![\gamma_i]\!]_A(a,b) = \begin{cases} a_j & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ b_k & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases}$$

Now take $a' = (E_A(e_1)(\pi_A(\rho)),....,E_A(e_r)(\pi_A(\rho)))$ and $b' = (\rho(In_1),....,\rho(In_n))$. Then $a' \in A^w$ and $b' \in A^u$, and so

$$[\![\gamma_i]\!]_A(a',b') = \begin{cases} E_A(e_j)(\rho) & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ \rho(In_k) & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases} \qquad (120)$$

We can now show (118) holds coordinatewise. For any $a \in A^u$ we have:

$$[\![\alpha]\!]_A(a) = [\![\gamma \circ \beta]\!]_A(a)$$
$$= [\![<\gamma_1,....,\gamma_m>]\!]_A([\![\beta]\!]_A(a))$$

Thus for any $\rho \in States(A)$ we have:

$$[\![\alpha]\!]_A(\pi_A(In)(\rho)) = [\![<\gamma_1,....,\gamma_m>]\!]_A([\![\beta]\!]_A(\pi_A(In)(\rho)))$$
$$= [\![<\gamma_1,....,\gamma_m>]\!]_A(E_A(e_1)(\rho),....,E_A(e_r)(\rho),\rho(In_1),....,\rho(In_n))$$

using (119).

Thus the ith coordinate of $[\![\alpha]\!]_A(\pi_A(In)(\rho))$ is

$$[\![\gamma_i]\!]_A(E_A(e_1)(\rho),....,E_A(e_r),\rho(In_1),....,\rho(In_n))$$
$$= \begin{cases} E_A(e_j)(\rho) & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ \rho(In_k) & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases} \qquad (121)$$

using (120).

Now consider the ith coordinate of $\pi_A(Out)(M_A(S)(\rho))$: this is simply $M_A(S)(\rho)(Out_i)$. However,

$$M_A(S)(\rho) = \rho\{E_A(e_1)(\rho)/x_1\}\{\ \cdots\ \}\{E_A(e_r)(\rho)/x_r\}$$

Thus by definition of a variant of a state, the ith coordinate of $\pi_A(Out)(M_A(S)(\rho)$ is:

$$M_A(S)(\rho)(Out_i) = \rho\{E_A(e_1)(\rho)/x_1\}\{E_A(e_r)(\rho)/x_r\}(Out_i)$$
$$= \begin{cases} E_A(e_j)(\rho) & \text{if } Out_i = x_j \text{ for some } j \in [1,r] \\ \rho(In_k) & \text{if } Out_i = In_k \text{ for some } k \in [1,n] \end{cases}$$

$$= [\![\gamma_i]\!]_A(E_A(e_1)(\rho),....,E_A(e_r)(\rho),\rho(In_1),....,\rho(In_n))$$

Thus the ith coordinates of $\pi_A(Out)(M_A(S)(\rho))$ and $[\![\alpha]\!]_A(\pi_A(In)(\rho))$ agree for $i = 1,...,m$ and for all $\rho \in States(A)$. Thus (118) holds and so c^{IO} correctly compiles multiple assignment statements.

Performance Preservation. To show that (116) and (117) hold for (S,In,Out), we will first relate $\lambda_P^{PR}(\alpha)$ to $\lambda_P^{PIT}(S)$ as follows:

First consider β; for any $\rho \in States(A)$ we have:

$$\lambda_P^{PR}(\beta)(\pi_A(In)(\rho)) = \lambda_P^{PR}(<\alpha_1,....,\alpha_r,U_1^u,....,U_n^u>)(\pi_A(In)(\rho))$$
$$= \max\{\ \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)),....,\lambda_P^{PR}(\alpha_r)(\pi_A(In)(\rho)),1,....,1\ \}$$
$$= \max\{\ \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)),....,\lambda_P^{PR}(\alpha_r)(\pi_A(In)(\rho))\ \}$$

(since $\lambda_P^{PR}(\alpha)(a) \geq 1$ for any α and any a)

$$= \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_r)(\rho)\}$$

This last step follows since for $i=1,\ldots,r$, $\alpha_i = c^{EXP}(e_i,In)$ and c^{EXP} is a performance preserving compiler. (See Lemma 7.3.2.) Thus we have shown:

$$(\forall \rho \in States(A))\ (\lambda_P^{PR}(\beta)(\pi_A(In)(\rho)) = \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_r)(\rho)\}) \quad (122)$$

Now let us consider γ. Since every coordinate of γ is a projection symbol, we have

$$(\forall a \in A^w)(\forall b \in A^v)\ (\lambda_P^{PR}(\gamma)(a,b) = 1) \quad (123)$$

Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{PR}(\gamma \circ \beta)(\pi_A(In)(\rho))\\ &= \lambda_P^{PR}(\beta)(\pi_A(In)(\rho)) + \lambda_P^{PR}(\gamma)(\|\beta\|_A(\pi_A(In)(\rho)))\\ &= \max\{\lambda_P^{EXP}(e_1)(\rho),\ldots,\lambda_P^{EXP}(e_r)(\rho)\} + 1\end{aligned}$$

(using (122) and (123))

$$= \lambda_P^{PIT}(S)(\rho) + 1 \quad (124)$$

To see that (116) holds for (S,In,Out) and some choice of μ, take $\mu=2$. Then from (124) we have for any $\rho \in States(A)$ that

$$\begin{aligned}\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{PIT}(S)(\rho) + 1\\ &\leq \lambda_P^{PIT}(S)(\rho) + \lambda_P^{PIT}(S)(\rho)\end{aligned}$$

(since $\lambda_P^{PIT}(S)(\rho) \geq 1$ for any S and any ρ)

$$= \mu \cdot \lambda_P^{PIT}(S)(\rho)$$

Thus (116) holds for this choice of μ as required.

To see that (117) holds, we have from (124) that for any $\rho \in States(A)$,

$$\begin{aligned}\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{PIT}(S)(\rho) + 1\\ &\geq \lambda_P^{PIT}(S)(\rho)\end{aligned}$$

Thus (117) holds as claimed, and thus c^{IO} preserves the performance of multiple assignment statements.

Induction. Let $S \in \mathrm{PIT}(\Sigma)$ be some program with the property that for every $S' \in \mathrm{PIT}(\Sigma)$ of less structural complexity than S, whenever $(S',In',Out') \in \mathrm{PITIO}(\Sigma)_{u',v'}$ for some $In' \in Var_{u'}$ and $Out' \in Var_{v'}$ for any $u',v' \in S^+$, we have that c^{IO} correctly compiles (S',In',Out'), that is:

$$(\forall \rho \in States(A))\ (\|c^{IO}(S',In',Out')\|_A(\pi_A(In')(\rho)) = \pi_A(Out')(M_A(S')(\rho)))$$

Furthermore, assume that there exists $\mu' \in \mathbb{N}^+$ such that:

$$(\forall \rho \in States(A))\ (\lambda_P^{PR}(c^{IO}(S',In',Out'))(\pi_A(In')(\rho)) \leq \mu' \cdot \lambda_P^{PIT}(S')(\rho))$$

and, additionally,

$$(\forall \rho \in States(A))\ (\lambda_P^{PIT}(S')(\rho) \leq \lambda_P^{PR}(c^{IO}(S',In',Out'))(\pi_A(In')(\rho)))$$

Now let In and Out be such that $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$ for some $u,v \in S^+$. We will now show c^{IO} correctly compiles (S,In,Out) according to the three following possible cases:

(ii) *Sequencing.* Suppose $S = S_1;S_2$ for some $S_1,S_2 \in \mathrm{PIT}(\Sigma)$.

In this case we take $In_1 = In \in Var_u$, $Out_1 \in Var_w$ such that $var(Out) = var(S_1) \cup var(In_1)$, $In_2 = Out_1 \in Var_w$, and $Out_2 = Out \in Var_v$, and then $\alpha = c^{IO}(S,In,Out)$ is defined by:

$$\alpha = c^{IO}(S,In,Out) = \alpha_2 \circ \alpha_1$$

where $\alpha_1 = c^{IO}(S_1,In_1,Out_1) \in \mathrm{PR}(\Sigma)_{u,w}$, and $\alpha_2 = c^{IO}(S_2,In_2,Out_2) \in \mathrm{PR}(\Sigma)_{w,v}$.

Since S_1 and S_2 are of less structural complexity than S, we have by the induction hypothesis applied to (S_1,In_1,Out_1) and (S_2,In_2,Out_2), that:

$$(\forall \rho \in States(A))\ (\ [\![\alpha_1]\!]_A(\pi_A(In_1)(\rho)) = \pi_A(Out_1)(M_A(S_1)(\rho))\ ) \tag{125}$$

and

$$(\forall \rho \in States(A))\ (\ [\![\alpha_2]\!]_A(\pi_A(In_2)(\rho)) = \pi_A(Out_2)(M_A(S_2)(\rho))\ ) \tag{126}$$

respectively.

To show c^{IO} correctly compiles (S,In,Out), choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}
[\![c^{IO}(S,In,Out)]\!]_A(\pi_A(In)(\rho)) &= [\![\alpha]\!]_A(\pi_A(In)(\rho)) \\
&= [\![\alpha_2 \circ \alpha_1]\!]_A(\pi_A(In)(\rho)) \\
&= [\![\alpha_2]\!]_A([\![\alpha_1]\!]_A(\pi_A(In)(\rho))) \\
&= [\![\alpha_2]\!]_A(\pi_A(Out_1)(M_A(S_1)(\rho)))
\end{aligned}$$

(by (125) and the fact that $In_1 = In$)

$$= \pi_A(Out_2)(M_A(S_2)(M_A(S_1)(\rho)))$$

(by (126) and the fact that $Out_1 = In_2$)

$$= \pi_A(Out)(M_A(S)(\rho))$$

since $Out_2 = Out$ and $S = S_1;S_2$.

Thus c^{IO} correctly compiles the sequencing operation.

Performance Preservation. Since S_1 and S_2 are of less structural complexity than S, we have by the induction hypothesis applied to (S_1,In_1,Out_1) and (S_2,In_2,Out_2), that there exist $\mu_i \in \mathbb{N}^+$ for $i = 1,2$ such that:

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PR}(\alpha_i)(\pi_A(In_i)(\rho)) \le \mu_i \,.\, \lambda_P^{PIT}(S_i)(\rho)\ ) \tag{127}$$

for $i = 1,2$, and furthermore,

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PIT}(S_i)(\rho) \le \lambda_P^{PR}(\alpha_i)(\pi_A(In_i)(\rho))\ ) \tag{128}$$

again for $i = 1,2$.

We can now show c^{IO} preserves the performance of (S,In,Out) as follows:

Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}
\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{PR}(\alpha_2 \circ \alpha_1)(\pi_A(In)(\rho)) \\
&= \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) + \lambda_P^{PR}(\alpha_2)([\![\alpha_1]\!]_A(\pi_A(In)(\rho))) \\
&= \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) + \lambda_P^{PR}(\alpha_2)(\pi_A(Out_1)(M_A(S_1)(\rho)))
\end{aligned} \tag{129}$$

since $In = In_1$ and c^{IO} correctly compiles (S_1,In_1,Out_1).

To see (116) holds for (S,In,Out) for some choice of μ, take $\mu = \max\{\mu_1,\mu_2\}$, choose $\rho \in States(A)$ and calculate as follows:

$$\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) \le \mu_1 \,.\, \lambda_P^{PIT}(S_1)(\rho) + \lambda_P^{PR}(\alpha_2)(\pi_A(Out_1)(M_A(S_1)(\rho)))$$

(from (129) using (127) with $i = 1$)

$$\leq \mu_1 . \lambda_P^{PIT}(S_1)(\rho) + \mu_2 . \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho))$$

(using (127) with $i=2$ and the fact that $Out_1 = In_2$)

$$\leq \max\{\mu_1,\mu_2\} . (\lambda_P^{PIT}(S_1)(\rho) + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho)))$$

(using the fact that if $a,b,m,n \in \mathbb{N}$ with $n,m \geq 1$, then $an+bm \leq \max\{a,b\}(n+m)$)

$$\leq \max\{\mu_1,\mu_2\} . \lambda_P^{PIT}(S_1;S_2)(\rho)$$

$$= \mu . \lambda_P^{PIT}(S)(\rho)$$

Thus (116) holds for this choice of μ as claimed.

To see that (117) holds for (S,In,Out), we have from (129) that for any $\rho \in States(A)$,

$$\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) = \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) + \lambda_P^{PR}(\alpha_2)(\pi_A(Out_1)(M_A(S_1)(\rho)))$$

$$\geq \lambda_P^{PIT}(S_1)(\rho) + \lambda_P^{PIT}(S_2)(M_A(S_1)(\rho))$$

(using the fact that $In_2 = Out_1$, together with (128) for $i=1,2$)

$$= \lambda_P^{PIT}(S_1;S_2)(\rho)$$

$$= \lambda_P^{PIT}(S)(\rho)$$

as required.

Thus c^{IO} preserves the performance of (S,In,Out).

(iii) *Conditional.* Suppose $S =$ **if** b **then** S_1 **else** S_2 **fi** for some $b \in \mathrm{EXP}(\Sigma)_B$, and some $S_1,S_2 \in \mathrm{PIT}(\Sigma)$.

Then $\alpha = c^{IO}(S,In,Out)$ is defined by:

$$\alpha = c^{IO}(S,In,Out) = \mathrm{DC}(c^{EXP}(b,In),\alpha_1,\alpha_2)$$

where $\alpha_i = c^{IO}(S_i,In_i,Out_i)$ for $i=1,2$.

Correctness. First notice that by the correctness of the expression compiler, we have:

$$(\forall \rho \in States(A))\ (\ [\![\alpha_b]\!]_A(\pi_A(In)(\rho)) = E_A(b)(\rho)\ ) \qquad (130)$$

where $\alpha_b = c^{EXP}(b,In)$.

Secondly, since S_1 and S_2 are of less structural complexity than S, we have by the induction hypothesis applied to (S_i,In_i,Out_i) for $i=1,2$:

$$(\forall \rho \in States(A))\ (\ [\![\alpha_i]\!]_A(\pi_A(In)(\rho)) = \pi_A(Out)(M_A(S_i)(\rho))\ ) \qquad (131)$$

for $i=1,2$.

To show c^{IO} correctly compiles (S,In,Out) choose $\rho \in States(A)$ and calculate as follows:

$$[\![c^{IO}(S,In,Out)]\!]_A(\pi_A(In)(\rho)) = [\![\alpha]\!]_A(\pi_A(In)(\rho))$$

$$= [\![\mathrm{DC}(\alpha_b,\alpha_1,\alpha_2)]\!]_A(\pi_A(In)(\rho))$$

$$= \begin{cases} [\![\alpha_1]\!]_A(\pi_A(In)(\rho)) & \text{if } [\![\alpha_b]\!]_A(\pi_A(In)(\rho)) = tt \\ [\![\alpha_2]\!]_A(\pi_A(In)(\rho)) & \text{if } [\![\alpha_b]\!]_A(\pi_A(In)(\rho)) = ff \end{cases}$$

$$= \begin{cases} [\![\alpha_1]\!]_A(\pi_A(In)(\rho)) & \text{if } E_A(b)(\rho) = tt \\ [\![\alpha_2]\!]_A(\pi_A(In)(\rho)) & \text{if } E_A(b)(\rho) = ff \end{cases}$$

(by (130))

$$= \begin{cases} \pi_A(Out)(M_A(S_1)(\rho)) & \text{if } E_A(b)(\rho)=tt \\ \pi_A(Out)(M_A(S_2)(\rho)) & \text{if } E_A(b)(\rho)=ff \end{cases}$$

(by (131) with $i=1$ and $i=2$)

$$= \begin{cases} \pi_A(Out)(M_A(S)(\rho)) & \text{if } E_A(b)(\rho)=tt \\ \pi_A(Out)(M_A(S)(\rho)) & \text{if } E_A(b)(\rho)=ff \end{cases}$$

$$= \pi_A(Out)(M_A(S)(\rho))$$

Thus c^{IO} correctly compiles conditional statements.

Performance Preservation. First notice that by Lemma 7.3.2 we have:

$$(\forall \rho \in States(A)) \ (\lambda_P^{PR}(\alpha_b)(\pi_A(In)(\rho)) = \lambda_P^{EXP}(b)(\rho)) \tag{132}$$

where $\alpha_b = c^{EXP}(b, In)$.

Secondly, since S_1 and S_2 are of less structural complexity than S, we have by the induction hypothesis applied to (S_i, In_i, Out_i) for $i=1,2$ that there exists $\mu_i \in \mathbb{N}^+$ such that:

$$(\forall \rho \in States(A)) \ (\lambda_P^{PR}(\alpha_i)(\pi_A(In)(\rho)) \leq \mu_i \,.\, \lambda_P^{PIT}(S_i)(\rho)) \tag{133}$$

for $i=1,2$, and, furthermore,

$$(\forall \rho \in States(A)) \ (\lambda_P^{PIT}(S_i)(\rho) \leq \lambda_P^{PR}(\alpha_i)(\pi_A(In)(\rho))) \tag{134}$$

again for $i=1,2$.

To show c^{IO} preserves the performance of (S, In, Out) choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned} \lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{PR}(\mathrm{DC}(\alpha_b, \alpha_1, \alpha_2))(\pi_A(In)(\rho)) \\ &= \lambda_P^{PR}(\alpha_b)(\pi_A(In)(\rho)) \\ &\quad + \begin{cases} \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) & \text{if } [\![\alpha_b]\!]_A(\pi_A(In)(\rho))=tt \\ \lambda_P^{PR}(\alpha_2)(\pi_A(In)(\rho)) & \text{if } [\![\alpha_b]\!]_A(\pi_A(In)(\rho))=ff \end{cases} \\ &= \lambda_P^{EXP}(b)(\rho) + \begin{cases} \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) & \text{if } E_A(b)(\rho)=tt \\ \lambda_P^{PR}(\alpha_2)(\pi_A(In)(\rho)) & \text{if } E_A(b)(\rho)=ff \end{cases} \end{aligned} \tag{135}$$

since c^{EXP} is a correct and performance preserving compiler.

To see that (116) holds for some choice of μ, take $\mu = \max\{\mu_1, \mu_2\}$.

First suppose ρ is such that $E_A(b)(\rho)=tt$. Then from (135) above we have that for any $\rho \in States(A)$,

$$\begin{aligned} \lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{EXP}(b)(\rho) + \lambda_P^{PR}(\alpha_1)(\pi_A(In)(\rho)) \\ &\leq \lambda_P^{EXP}(b)(\rho) + \mu_1 \,.\, \lambda_P^{PIT}(S_1)(\rho) \end{aligned}$$

(using (127) with $i=1$)

$$\begin{aligned} &\leq \lambda_P^{EXP}(b)(\rho) + \mu \,.\, \lambda_P^{PIT}(S_1)(\rho) \\ &\leq \mu \,.\, (\lambda_P^{EXP}(b)(\rho) + \lambda_P^{PIT}(S_1)(\rho)) \end{aligned}$$

(since $\mu \geq 1$)

$$= \mu \,.\, \lambda_P^{PIT}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\rho)$$

(since $E_A(b)(\rho)=tt$)

$$= \mu . \lambda_P^{PIT}(S)(\rho)$$

Thus (116) holds in the case $E_A(b)(\rho) = tt$.

In almost identical fashion, it is possible to prove that

$$\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) \leq \lambda_P^{PIT}(S)(\rho) \tag{136}$$

when ρ is such that $E_A(b)(\rho) = ff$ (this we leave as an exercise), and thus we conclude (136) holds for every $\rho \in States(A)$; that is, (116) holds for α as claimed.

To see that (117) holds for (S, In, Out), first suppose ρ is such that $E_A(b)(\rho) = tt$. Then from (135) and the induction hypothesis (134) with $i = 1$, we have

$$\begin{aligned} \lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &\geq \lambda_P^{EXP}(b)(\rho) + \lambda_P^{PIT}(S_1)(\rho) \\ &= \lambda_P^{PIT}(S)(\rho) \end{aligned} \tag{137}$$

Also, if $E_A(b)(\rho) = ff$, then again from (135) and the induction hypothesis (134) with $i = 2$, we have

$$\begin{aligned} \lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &\geq \lambda_P^{EXP}(b)(\rho) + \lambda_P^{PIT}(S_2)(\rho) \\ &= \lambda_P^{PIT}(S)(\rho) \end{aligned} \tag{138}$$

Hence from (137) and (138) we have

$$(\forall \rho \in States(A))\ \ (\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho) \geq \lambda_P^{PIT}(S)(\rho))$$

Thus (117) holds for (S, In, Out) as claimed.

(iv) *Bounded Iteration.* Suppose S = **do** e **times** S_0 **od** for some $e \in \mathrm{EXP}(\Sigma)_\tau$ and some $S_0 \in \mathrm{PIT}(\Sigma)$.

Then $\alpha = c^{IO}(S, In, Out)$ is defined by:

$$\alpha = c^{IO}(S, In, Out) = \beta \circ \gamma \circ \delta$$

where:

$$\beta = \langle U^{\mu}_{\theta(1)}, \ldots, U^{\mu}_{\theta(m)} \rangle$$

where $\theta(j)$ is the least k such that $Out_j = In_k$;

$$\gamma = it(\alpha_0)$$

where

$$\alpha_0 = c^{IO}(S_0, In, In)$$

and

$$\delta = \langle \alpha_e, U^{\mu}_1, \ldots, U^{\mu}_n \rangle$$

where $\alpha_e = c^{EXP}(e, In)$.

To show that c^{IO} correctly compiles S, let us first consider δ. Choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned} [\![\delta]\!]_A(\pi_A(In)(\rho)) &= [\![\langle \alpha_e, U^{\mu}_1, \ldots, U^{\mu}_n \rangle]\!]_A(\pi_A(In)(\rho)) \\ &= ([\![\alpha_e]\!]_A(\pi_A(In)(\rho)), [\![U^{\mu}_1]\!]_A(\pi_A(In)(\rho)), \ldots, [\![U^{\mu}_n]\!]_A(\pi_A(In)(\rho))) \\ &= (E_A(e)(\rho), \rho(In_1), \ldots, \rho(In_n)) \end{aligned}$$

(since the expression compiler is correct)

$$= (E_A(e)(\rho), \pi_A(In)(\rho)) \tag{139}$$

Now consider β. For any $\rho \in States(A)$ we have:

$$\begin{aligned}[[\beta]]_A(\pi_A(In)(\rho)) &= [[<U^u_{\theta(1)},\ldots,U^u_{\theta(m)}>]]_A(\pi_A(In)(\rho)) \\ &= ([[U^u_{\theta(1)}]]_A(\pi_A(In)(\rho)),\ldots,[[U^u_{\theta(m)}]]_A(\pi_A(In)(\rho))) \\ &= (\rho(In_{\theta(1)}),\ldots,\rho(In_{\theta(m)})) \\ &= (\rho(Out_1),\ldots,\rho(Out_m))\end{aligned}$$

(by definition of θ)

$$= \pi_A(Out)(\rho) \qquad (140)$$

To conclude the proof of correctness, let us temporarily assume:

$$(\forall k \in \mathbb{N})(\forall \rho \in States(A))\ \ ([[\gamma]]_A(k,\pi_A(In)(\rho)) = \pi_A(In)(\rho_k)) \qquad (141)$$

where $\rho_0 = \rho$ and for each $l \in \mathbb{N}$, $\rho_{l+1} = M_A(S_0)(\rho_l)$.

To show c^{IO} correctly compiles S choose $\rho \in States(A)$ and calculate as follows:

$$\begin{aligned}[[c^{IO}(S,In,Out)]]_A(\pi_A(In)(\rho)) &= [[\alpha]]_A(\pi_A(In)(\rho)) \\ &= [[\beta \circ \gamma \circ \delta]]_A(\pi_A(In)(\rho)) \\ &= [[\beta]]_A([[\gamma]]_A([[\delta]]_A(\pi_A(In)(\rho)))) \\ &= [[\beta]]_A([[\gamma]]_A(k,\pi_A(In)(\rho)))\end{aligned}$$

(where $k = E_A(e)(\rho)$ by (139))

$$= [[\beta]]_A(\pi_A(In)(\rho_k))$$

(by (141))

$$= \pi_A(Out)(\rho_k)$$

(by (140))

$$= \pi_A(Out)(M_A(S)(\rho))$$

by definition of k, ρ_k, and M_A. Thus c^{IO} correctly compiles S as claimed.

It remains to prove (141). First notice that since S_0 is of less structural complexity than S, by the induction hypothesis applied to (S_0,In,In) we have:

$$(\forall \rho \in States(A))\ \ ([[\alpha_0]]_A(\pi_A(In)(\rho)) = \pi_A(In)(M_A(S_0)(\rho)) \qquad (142)$$

To prove (141), we must show that

$$(\forall \rho \in States(A))\ \ ([[\gamma]]_A(k,\pi_A(In)(\rho) = \pi_A(In)(\rho_k)) \qquad (143)$$

holds for all $k \in \mathbb{N}$; this we do by sub-induction on k as follows:

Sub-Basis. Using Lemma 3.3.5, for any $\rho \in States(A)$ we have

$$\begin{aligned}[[\gamma]]_A(0,\pi_A(In)(\rho)) &= [[it(\alpha_0)]]_A(0,\pi_A(In)(\rho)) \\ &= \pi_A(In)(\rho) \\ &= \pi_A(In)(\rho_0)\end{aligned}$$

since $\rho_0 = \rho$ by definition.

Sub-Induction. Suppose for some fixed $l \in \mathbb{N}$ we have shown that for $k = 0,\ldots, l$,

$$(\forall \rho \in States(A))\ (\ [\![\gamma]\!]_A(k, \pi_A(In)(\rho) = \pi_A(In)(\rho_k)\ ) \tag{144}$$

To show (143) holds for $k = l+1$ choose $\rho \in States(A)$ and calculate as follows:

$$[\![\gamma]\!]_A(l+1, \pi_A(In)(\rho) = [\![it(\alpha_o)]\!]_A(l+1, \pi_A(In)(\rho))$$
$$= [\![\alpha_o]\!]_A([\![it(\alpha_o)]\!]_A(l, \pi_A(In)(\rho)))$$

(using Lemma 3.3.5)

$$= [\![\alpha_o]\!]_A(\pi_A(In)(\rho_l))$$

(using the sub-induction hypothesis)

$$= \pi_A(In)(M_A(S_o)(\rho_l))$$

(by (142))

$$= \pi_A(In)(\rho_{l+1})$$

by definition of ρ_{l+1}. Thus (143) holds for $k = l+1$, and so by the principle of mathematical induction (143) holds for all $k \in \mathbb{N}$.

Performance Preservation. To show that c^{IO} preserves the performance of S, let us first consider δ. Choose $\rho \in States(A)$ and calculate as follows:

$$\lambda_P^{PR}(\delta)(\pi_A(In)(\rho)) = \lambda_P^{PR}(<\alpha_e, U_1^u, \ldots, U_n^u>)(\pi_A(In)(\rho))$$
$$= \max\{\lambda_P^{PR}(\alpha_e)(\pi_A(In)(\rho), 1, \ldots, 1\}$$
$$= \lambda_P^{PR}(\alpha_e)(\pi_A(In)(\rho))$$

(since $\lambda_P^{PR}(\alpha)(a) \geq 1$ for any α and any a)

$$= \lambda_P^{EXP}(e)(\rho) \tag{145}$$

since c^{EXP} is a performance preserving compiler.

Also, it is not difficult to see that

$$(\forall a \in A^u)\ (\ \lambda_P^{PR}(\beta)(a) = 1\ ) \tag{146}$$

Also, since S_o is of less structural complexity than S, by the induction hypothesis applied to $(S_o, In, In) \in \mathrm{PITIO}(\Sigma)_{u,u}$ there exists $\mu_o \in \mathbb{N}^+$ such that

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PR}(\alpha_o)(\pi_A(In)(\rho)) \leq \mu_o . \lambda_P^{PIT}(S_o)(\rho)\ ) \tag{147}$$

and, furthermore,

$$(\forall \rho \in States(A))\ (\ \lambda_P^{PIT}(S_o)(\rho) \leq \lambda_P^{PR}(\alpha_o)(\pi_A(In)(\rho))\ ) \tag{148}$$

Now let us consider $\lambda_P^{PR}(\gamma)(k, a)$ for any $k \in \mathbb{N}$ and $a \in A^u$. Using the definition of γ we have:

$$\lambda_P^{PR}(\gamma)(k, a) = \lambda_P^{PR}(it(\alpha_o))(k, a)$$
$$= \lambda_P^{PR}(*(Id^u, \alpha_o \circ <U_{2+n}^w, \ldots, U_{1+2n}^w>)(k, a)$$

where $w = \tau uu \in S^{1+2n}$.

Thus

$$\lambda_P^{PR}(\gamma)(0, a) = \lambda_P^{PR}(Id^u)(a)$$
$$= \lambda_P^{PR}(<U_1^u, \ldots, U_n^u>)(a)$$

$$= 1 \tag{149}$$

Also, for any $k \in \mathbb{N}$,

$$\lambda_P^{PR}(\gamma)(k+1,a) = \lambda_P^{PR}(\gamma)(k,a) + \lambda_P^{PR}(\alpha_0 \circ < U_{2+n}^w, \ldots, U_{1+2n}^w >)(k,a,[\![\gamma]\!]_A(k,a))$$

$$= \lambda_P^{PR}(\gamma)(k,a) + \lambda_P^{PR}(< U_{2+n}^w, \ldots, U_{1+2n}^w >)(k,a,[\![\gamma]\!]_A(k,a))$$
$$+ \lambda_P^{PR}(\alpha_0)([\![< U_{2+n}^w, \ldots, U_{1+2n}^w >]\!]_A(k,a,[\![\gamma]\!]_A(k,a)))$$

$$= \lambda_P^{PR}(\gamma)(k,a) + 1 + \lambda_P^{PR}(\alpha_0)([\![\gamma]\!]_A(k,a)) \tag{150}$$

Now consider $\lambda_P^{PIT}(S)(\rho)$: since S is a loop, we have from Definition 6.1.7(iv) that

$$\lambda_P^{PIT}(S)(\rho) = \lambda_P^{EXP}(e)(\rho) + \lambda_l \tag{151}$$

where $l = E_A(e)(\rho)$,

$$\lambda_0 = 0,$$

and for any $k \in \mathbf{N}$,

$$\lambda_{k+1} = \lambda_k + \lambda_P^{PIT}(S_0)(\rho_k)$$

where for each $k \in \mathbf{N}$, ρ_k is defined by:

$$\rho_0 = \rho,$$

and for any $k \in \mathbf{N}$,

$$\rho_{k+1} = M_A(S_0)(\rho_k)$$

In order to prove (116) holds for α, let us rephrase $\lambda_P^{PIT}(S)$ is the following way. First define $\kappa: \mathbf{N} \times States(A) \longrightarrow C^+$ by:

$$\kappa(0,\rho) = \lambda_P^{EXP}(e)(\rho)$$

and

$$\kappa(k+1,\rho) = \kappa(k,\rho) + \lambda_P^{PIT}(S_0)(\rho_k)$$

for each $k \in \mathbb{N}$ and $\rho \in States(A)$.

A routine induction on k yields:

$$(\forall k \in \mathbb{N})(\forall \rho \in States(A)) \ \ (\kappa(k,\rho) = \lambda_P^{EXP}(e)(\rho) + \lambda_k)$$

and thus by (151) we have:

$$(\forall \rho \in States(A)) \ \ (\lambda_P^{PIT}(S)(\rho) = \kappa(E_A(e)(\rho),\rho)) \tag{152}$$

We now make the following claim which we will prove later:

Claim.

$$(\forall k \in \mathbb{N})(\forall \rho \in States(A)) \ \ (\lambda_P^{PR}(\gamma)(k,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq \nu \cdot \kappa(k,\rho))$$

where $\nu = \max\{3, 1+\mu_0\}$.

Assuming the Claim holds, we can now show (116) holds for (S,In,Out) for the choice $\mu = 1+\nu$ as follows: choose $\rho \in States(A)$; then

$$\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) = \lambda_P^{PR}(\beta \circ \gamma \circ \delta)(\pi_A(In)(\rho))$$
$$= \lambda_P^{PR}(\delta)(\pi_A(In)(\rho)) + \lambda_P^{PR}(\gamma)([\![\delta]\!]_A(\pi_A(In)(\rho))) \tag{153}$$

$$+ \lambda_P^{PR}(\beta)(\llbracket \gamma \circ \delta \rrbracket_A(\pi_A(In)(\rho)))$$
$$= \lambda_P^{EXP}(e)(\rho) + \lambda_P^{PR}(\gamma)(E_A(e)(\rho), \pi_A(In)(\rho)) + 1$$

(using (145), (139), and (146) respectively)

$$\leq \nu \cdot \kappa(E_A(e)(\rho), \rho) + 1$$

(using the Claim)

$$= \nu \cdot \lambda_P^{PIT}(S)(\rho) + 1$$

(by (152))

$$\leq \nu \cdot \lambda_P^{PIT}(S)(\rho) + \lambda_P^{PIT}(S)(\rho)$$

(since $\lambda_P^{PIT}(S)(\rho) \geq 1$ for any S and any ρ)

$$= (1+\nu) \cdot \lambda_P^{PIT}(S)(\rho)$$
$$= \mu \cdot \lambda_P^{PIT}(S)(\rho)$$

Thus (116) holds for (S, In, Out) and this choice of μ as claimed.

Proof of Claim. We must prove for every $k \in \mathbf{N}$ and every $\rho \in States(A)$, that

$$\lambda_P^{PR}(\gamma)(k, \pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq \nu \cdot \kappa(k, \rho) \tag{154}$$

We prove (154) holds for each $k \in \mathbf{N}$ and $\rho \in States(A)$ by sub-induction on k:

Sub-Basis. For $k = 0$, we must prove for every $\rho \in States(A)$ that

$$\lambda_P^{PR}(\gamma)(0, \pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq \nu \cdot \kappa(0, \rho)$$

that is, that

$$1 + \lambda_P^{EXP}(e)(\rho) \leq \max\{3, 1+\mu_0\} \cdot \lambda_P^{EXP}(e)(\rho) \tag{155}$$

(using (149) and the definitions of ν and κ).

We can establish (155) as follows:

$$1 + \lambda_P^{EXP}(e)(\rho) \leq 2 \cdot \lambda_P^{EXP}(e)(\rho) \tag{156}$$

(since $\lambda_P^{EXP}(e)(\rho) \geq 1$ for any e and any ρ).

However, it is certainly true that

$$2 \cdot \lambda_P^{EXP}(e)(\rho) \leq \max\{3, 1+\mu_0\} \cdot \lambda_P^{EXP}(e)(\rho) \tag{157}$$

since in the worst case, $\max\{3, 1+\mu_0\} = 2$ (because $\mu_0 \geq 1$). Thus from (156) and (157) we have

$$1 + \lambda_P^{EXP}(e)(\rho) \leq 2 \cdot \lambda_P^{EXP}(e)(\rho) \leq \max\{3, 1+\mu_0\} \cdot \lambda_P^{EXP}(e)(\rho)$$

and hence (155) holds as required.

Sub-Induction. Suppose for some fixed $l \in \mathbf{N}$, that we have proved

$$(\forall \rho \in States(A)) \ (\lambda_P^{PR}(\gamma)(l, \pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq \nu \cdot \kappa(l, \rho)) \tag{158}$$

We now show (154) holds for every $\rho \in States(A)$ for $k = l+1$.

Choose $\rho \in States(A)$. Then from (150) we have for any $a \in A^u$,

$$\lambda_P^{PR}(\gamma)(l+1, a) = \lambda_P^{PR}(\gamma)(l, a) + 1 + \lambda_P^{PR}(\alpha_0)(\llbracket \gamma \rrbracket_A(l, a)) \tag{159}$$

Now, when $a = \pi_A(In)(\rho)$, we have from (141)

$$\llbracket \gamma \rrbracket_A(k, a) = \pi_A(In)(\rho_k)$$

for any $k \in \mathbf{N}$ and any $\rho \in States(A)$. Substituting for $\llbracket \gamma \rrbracket_A$ in (159) yields (for any $\rho \in States(A)$)

$$\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) = \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + 1 + \lambda_P^{PR}(\alpha_o)(\pi_A(In)(\rho_l))$$
$$\leq \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l)$$

(using (147)). Thus for any $\rho \in States(A)$ we have

$$\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + \lambda_P^{EXP}(e)(\rho)$$
$$\leq 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + \nu . \kappa(l,\rho) \qquad (160)$$

(using (158)).

Recall $\nu = \max\{3, 1+\mu_o\}$. There are now two cases to consider:

Case 1: $\max\{3, 1+\mu_o\} = 3$

Case 2: $\max\{3, 1+\mu_o\} = 1+\mu_o$

Case 1. Suppose $\max\{3, 1+\mu_o\} = 3$. Then from (160) we have that for any $\rho \in States(A)$,

$$\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \leq 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + \nu . \kappa(l,\rho)$$
$$= 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + 3 . \kappa(l,\rho)$$
$$\leq \lambda_P^{PIT}(S_o)(\rho_l) + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + 3 . \kappa(l,\rho)$$

(since $\lambda_P^{PIT}(S)(\rho) \geq 1$ for any S and any ρ)

$$= (1+\mu_o) . \lambda_P^{PIT}(S_o)(\rho_l) + 3 . \kappa(l,\rho)$$
$$\leq 3 . \lambda_P^{PIT}(S_o)(\rho_l) + 3 . \kappa(l,\rho)$$

(since $\max\{3, +\mu_o\} = 3$ here)

$$= \max\{3, 1+\mu_o\} . (\kappa(l,\rho) + \lambda_P^{PIT}(S_o)(\rho_l))$$

(since $\max\{3, +\mu_o\} = 3$ here)

$$= \nu . \kappa(l+1,\rho)$$

(by definition of κ). Thus (154) holds in this case.

Case 2. Suppose $\max\{3, 1+\mu_o\} = 1+\mu_o$. Then from (160) we have that for any $\rho \in States(A)$,

$$\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) \leq 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + \nu . \kappa(l,\rho)$$
$$= 1 + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + (1+\mu_o) . \kappa(l,\rho)$$

(since $\max\{3, 1+\mu_o\} = 1+\mu_o$ here).

$$\leq \lambda_P^{PIT}(S_o)(\rho_l) + \mu_o . \lambda_P^{PIT}(S_o)(\rho_l) + (1+\mu_o) . \kappa(l,\rho)$$

(since $\lambda_P^{PIT}(S)(\rho) \geq 1$ for any S and any ρ)

$$= (1+\mu_o) . (\kappa(l,\rho) + \lambda_P^{PIT}(S_o)(\rho_l))$$
$$= \max\{3, 1+\mu_o\} . (\kappa(l,\rho) + \lambda_P^{PIT}(S_o)(\rho_l))$$

(since $\max\{3, 1+\mu_o\} = 1+\mu_o$ here)

$$= \nu . (\kappa(l+1,\rho))$$

(by definition of ν and κ). Thus (154) holds in this case also.

By the principle of mathematical induction, (154) holds for all $k \in \mathbb{N}$, and so the Claim holds. □

We will now prove (117) holds for (S, In, Out).

Claim.

$$(\forall k \in \mathbb{N})(\forall \rho \in States(A))\ \ (\lambda_P^{PR}(\gamma)(k,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \geq \kappa(k,\rho)) \tag{161}$$

Let us prove (117) holds for (S,In,Out) before we verify this Claim.

From (153) we have

$$\begin{aligned}\lambda_P^{PR}(\alpha)(\pi_A(In)(\rho)) &= \lambda_P^{EXP}(e)(\rho) + \lambda_P^{PR}(\gamma)(E_A(e)(\rho),\pi_A(In)(\rho)) + 1\\ &> \lambda_P^{EXP}(e)(\rho) + \lambda_P^{PR}(\gamma)(E_A(e)(\rho),\pi_A(In)(\rho))\\ &\geq \kappa(E_A(e)(\rho),\rho)\end{aligned}$$

(using (161))

$$= \lambda_P^{PIT}(S)(\rho)$$

(by (152)). Thus (117) holds for (S,In,Out).

Assuming the above Claim holds, we have now shown that c^{IO} preserves the performance of (S,In,Out) when S is a loop, and thus c^{IO} preserves the performance of every (S,In,Out) (by structural induction). □

Proof of Claim. We must show for every $k \in \mathbb{N}$ and every $\rho \in States(A)$ that

$$\lambda_P^{PR}(\gamma)(k,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \geq \kappa(k,\rho) \tag{162}$$

We show (162) holds by sub-induction on k.

Sub-Basis. For $k=0$, we must prove for every $\rho \in States(A)$ that

$$1 + \lambda_P^{EXP}(e)(\rho) \geq \lambda_P^{EXP}(e)(\rho)$$

(using (149) and the definition of κ); this is obviously true.

Sub-Induction. Suppose for some fixed $l \in \mathbb{N}$, that we have proved

$$(\forall \rho \in States(A))\ \ (\lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) \geq \kappa(l,\rho)) \tag{163}$$

We now show (162) holds for every $\rho \in States(A)$ for $k=l+1$.

Choose $\rho \in States(A)$. Then from (150) we have

$$\begin{aligned}\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) &= \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + 1 + \lambda_P^{PR}(\alpha_o)([\![\gamma]\!]_A(k,\pi_A(In)(\rho)))\\ &\geq \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + \lambda_P^{PR}(\alpha_o)([\![\gamma]\!]_A(k,\pi_A(In)(\rho)))\\ &= \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + \lambda_P^{PR}(\alpha_o)(\pi_A(In)(\rho_l))\end{aligned}$$

(using (141))

$$\geq \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + \lambda_P^{PIT}(S_o)(\rho_l) \tag{164}$$

(using (148)). Adding $\lambda_P^{EXP}(e)(\rho)$ to both sides of (164) yields:

$$\begin{aligned}\lambda_P^{PR}(\gamma)(l+1,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) &\geq \lambda_P^{PR}(\gamma)(l,\pi_A(In)(\rho)) + \lambda_P^{EXP}(e)(\rho) + \lambda_P^{PIT}(S_o)(\rho_l)\\ &\geq \kappa(l,\rho) + \lambda_P^{PIT}(S_o)(\rho_l)\end{aligned}$$

(by the sub-induction hypothesis (163))

$$= \kappa(l+1,\rho)$$

(by definition of κ).

By the principle of mathematical induction, (162) holds for all $k \in \mathbb{N}$, and so the Claim holds. □

7.3.5 Corollary.

Let $u,v \in S^+$, *and let* $(S,In,Out) \in \mathrm{PITIO}(\Sigma)_{u,v}$. *Then,*

$$(\forall a \in A^u)\ (\ [\![c^{IO}(S,In,Out)]\!]_A(a) = F_A(S,In,Out)(a)\ )$$

Furthermore, there exists $\mu \geq 1$ *such that*

$$(\forall a \in A^u)\ (\ \lambda_p^{PR}(c^{IO}(S,In,Out))(a) \leq \mu.\lambda_p^{IO}(S,In,Out)(a)\ )$$

Additionally,

$$(\forall a \in A^u)\ (\ \lambda_p^{IO}(S,In,Out)(a) \leq \lambda_p^{PR}(c^{IO}(S,In,Out))(a)\ )$$

Proof. Immediate from the previous theorem and the definitions of $F_A(S,In,Out)$ and $\lambda_p^{IO}(S,In,Out)$ (see Definitions 6.2.10).

7.3.6 Lemma. *Let* Σ *be any signature and let* $c^{IO}(\Sigma)$ *be the word-indexed transformation of Section 7.3.3 (previously denoted* c^{IO}*). Then*

(i) $c^{IO}(\Sigma)$ *is a compiler from* $\mathrm{PITIO}(\Sigma)$ *into* $\mathrm{PR}(\Sigma)$*; that is,*

$$c^{IO}(\Sigma):\mathrm{PITIO}(\Sigma) \longrightarrow \mathrm{PR}(\Sigma)$$

(ii) Let A *be a* Σ*-algebra. Then* $c^{IO}(\Sigma)$ *is correct with respect to* $[\![.]\!]_A$ *and* F_A*; that is, for every* $(S,In,Out) \in \mathrm{PITIO}(\Sigma)$,

$$[\![c^{IO}(\Sigma)(S,In,Out)]\!]_A = F_A(S,In,Out)$$

(iii) Let P *be a performance measure for* A. *Then* $c^{IO}(\Sigma)$ *is a performance preserving compiler with respect to* λ_p^{PR} *and* λ_p^{IO}*; that is, for every* $(S,In,Out) \in \mathrm{PITIO}(\Sigma)$,

$$\lambda_p^{PR}(c^{IO}(\Sigma)(S,In,Out)) \approx \lambda_p^{IO}(S,In,Out)$$

Proof. Immediate from Corollary 7.3.5 and the fact that $c^{IO}(\Sigma)$ is word-indexed. □

7.3.7 Augmentation and Compilation.

Recall the map c of Section 3.5 (see Definition 3.5.6); given a (u,v)-extension (Σ,Φ) of a signature Σ, c is a map, dependent on Φ and a scheme $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, in symbols: $c(\Phi,\alpha)$, which transformed each scheme $\alpha' \in \mathrm{PR}(\Sigma,\Phi)$ into a scheme $c(\alpha') \in \mathrm{PR}(\Sigma)$ by replacing each occurrence in α' of each $\phi \in \Phi$ by a coordinate of α.

Now notice that c is actually properly dependent on Σ as well as Φ and α, since in applying c to a scheme $\alpha' \in \mathrm{PR}(\Sigma,\Phi)$ we must know if a given symbol $\sigma \in (\Sigma,\Phi)$ is a symbol of Σ (if it is, then we leave it alone, if not, then we replace it by a coordinate of α). In the forthcoming definition of $c^{FP}:\mathrm{FPIT}(\Sigma) \longrightarrow \mathrm{PR}(\Sigma)$ we will use c as a basic mechanism for replacing function identifiers by schema, but we will use it in the context of a variety of different signatures, and thus we must extend our notation to make explicit reference to the underlying signature involved: that is, when α is a scheme over Σ we must denote the transformation c by $c(\Sigma,\Phi,\alpha)$ or $c(\Sigma)(\Phi,\alpha)$ for example. We will use the notation '$c^{AUG}(\Sigma)(\Phi,\alpha)$', and we will write '$c^{AUG}(\Sigma)(\Phi,\alpha)$' and '$c^{AUG}(\Sigma)(\Phi,\alpha)(\alpha')$' for what was previously denoted by '$c(\Phi,\alpha)$' and '$c(\Phi,\alpha)(\alpha')$' respectively.

Similar remarks apply to the compiler c^{IO} of Section 7.3.3: we will use c^{IO} to transform an i/o-program to a PR scheme in the context of different signatures, and so we will subsequently write $c^{IO}(\Sigma)$ and $c^{IO}(\Sigma)(S,In,Out)$ for what was previously denoted by c^{IO} and $c^{IO}(S,In,Out)$ respectively.

Here are three facts concerning c^{AUG}. The first is simply a reformulation of c^{AUG} as a compiler in the sense of Section 7.1; the last two are technical results that will be useful later.

7.3.8 Lemma. *Let Σ be any signature and let (Σ,Φ) be any (u,v)-extension of Σ. Also, for each $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, let $c^{AUG}(\Sigma)(\Phi,\alpha)$ be the transformation of Definition 3.5.6 (previously denoted $c(\Phi,\alpha)$). Then*

(i) $c^{AUG}(\Sigma)(\Phi,\alpha)$ *is a compiler from* $\mathrm{PR}(\Sigma,\Phi)$ *into* $\mathrm{PR}(\Sigma)$*; that is,*

$$c^{AUG}(\Sigma)(\Phi,\alpha):\mathrm{PR}(\Sigma,\Phi)\longrightarrow\mathrm{PR}(\Sigma)$$

(ii) Let A be a Σ-algebra and let $B=(A,[\![\alpha]\!]_A)$. Then $c^{AUG}(\Sigma)(\Phi,\alpha)$ is correct with respect to $[\![\,.\,]\!]_A$ and $[\![\,.\,]\!]_B$; that is, for every $\alpha' \in \mathrm{PR}(\Sigma,\Phi)$,

$$[\![c^{AUG}(\Sigma)(\Phi,\alpha)(\alpha')]\!]_A = [\![\alpha']\!]_B$$

(iii) Let P be a performance measure for A which is based on clock C, and let $Q=(P,\lambda)$ where $\lambda:A^{\mu}\longrightarrow C^{+}$ is any map with $\lambda\approx\lambda_P^{PR}(\alpha)$. Then $c^{AUG}(\Sigma)(\Phi,\alpha)$ is a performance preserving compiler with respect to λ_P^{PR} and λ_Q^{PR}; that is, for every $\alpha' \in \mathrm{PR}(\Sigma,\Phi)$,

$$\lambda_P^{PR}(c^{AUG}(\Sigma)(\Phi,\alpha)(\alpha')) \approx \lambda_Q^{PR}(\alpha')$$

Proof. Immediate from Lemma 3.5.8 and the fact that $c^{AUG}(\Sigma)(\Phi,\alpha)$ is word-indexed (see Lemma 3.5.7). □

7.3.9 Lemma. *Let Σ be any signature, and let $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for any $u,v \in S^{+}$. Now let $G \in \mathrm{FG}(\Sigma)$ be the single function group $G=S$, and additionally let $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$. Then*

(i) $c^{AUG}(\Sigma)(id(S),\alpha)$ *is a compiler from* $\mathrm{PR}(sig(G))$ *into* $\mathrm{PR}(\Sigma)$*; that is,*

$$c^{AUG}(\Sigma)(id(S),\alpha):\mathrm{PR}(sig(G))\longrightarrow\mathrm{PR}(\Sigma)$$

(ii) If A is a Σ-algebra with $[\![\alpha]\!]_A = F_A(S)$, then $c^{AUG}(\Sigma)(id(S),\alpha)$ is correct with respect to $[\![\,.\,]\!]_A$ and $[\![\,.\,]\!]_{A_G}$ where $A_G = Alg_A(G)$; that is, for every $\alpha' \in \mathrm{PR}(sig(G))$,

$$[\![c^{AUG}(\Sigma)(\alpha')]\!]_A = [\![\alpha']\!]_{A_G}$$

(iii) If P is a performance measure for A with $\lambda_P^{PR}(\alpha)\approx\lambda_P^{FP}(S)$, then $c^{AUG}(\Sigma)(id(S),\alpha)$ is a performance preserving compiler with respect to λ_P^{PR} and $\lambda_{P_G}^{PR}$; that is, for every $\alpha' \in \mathrm{PR}(sig(G))$,

$$\lambda_P^{PR}(c^{AUG}(\Sigma)(\alpha')) \approx \lambda_{P_G}^{PR}(\alpha')$$

Proof. Part (i) of the lemma is immediate by Lemma 7.3.8 since $sig(G)$ is defined to be the (u,v)-extension $sig(G)=(\Sigma,id(S))$.

Part (ii) of the lemma is also immediate by Lemma 7.3.8 since $A_G=(A,F_A(S))$, but $F_A(S)=[\![\alpha]\!]_A$ by hypothesis, and so $A_G=(A,[\![\alpha]\!]_A)$.

Similarly, part (iii) of the lemma follows from Lemma 7.3.8 since P_G is defined by $P_G=(P,\lambda_P^{FP}(S))$, and $\lambda_P^{FP}(S)\approx\lambda_P^{PR}(\alpha)$ by hypothesis. □

7.3.10 Lemma. *Let Σ be any signature, and let $G\in \mathrm{FG}(\Sigma)$ be the multiple function group $G=G_o;S$ for some $G_o\in \mathrm{FG}(\Sigma)$ and some $S\in \mathrm{FPIT}(sig(G_o))$. If S is of arity (u,v) for some $u,v\in S^+$, then for every $\alpha\in \mathrm{PR}(sig(G_o))_{u,v}$,*

(i) $c^{AUG}(sig(G_o))(id(S),\alpha)$ *is a compiler from* $\mathrm{PR}(sig(G))$ *into* $\mathrm{PR}(sig(G_o))$*; that is,*

$$c^{AUG}(sig(G_o))(id(S),\alpha):\mathrm{PR}(sig(G))\longrightarrow\mathrm{PR}(sig(G_o))$$

(ii) If A is a Σ-algebra with $[\![\alpha]\!]_{A_o}=F_{A_o}(S)$ where $A_o=Alg_A(G_o)$, then $c^{AUG}(sig(G_o))(id(S),\alpha)$ is correct with respect to $[\![\,.\,]\!]_{A_o}$ and $[\![\,.\,]\!]_{A_o}$ where $A_G=Alg_A(G)$; that is, for every $\alpha'\in \mathrm{PR}(sig(G))$,

$$[\![c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha')]\!]_{A_o}=[\![\alpha']\!]_{A_o}$$

(iii) If P is a performance measure for A with $\lambda_{P_o}^{PR}(\alpha)\approx\lambda_{P_o}^{FP}(S)$ where $P_o=P_{G_o}$, then $c^{AUG}(sig(G_o))(id(S),\alpha)$ is a performance preserving compiler with respect to $\lambda_{P_o}^{PR}$ and $\lambda_{P_o}^{PR}$; that is, for every $\alpha'\in \mathrm{PR}(sig(G))$,

$$\lambda_{P_o}^{PR}(c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha'))\approx\lambda_{P_o}^{PR}(\alpha')$$

Proof. The proof of parts (i)-(iii) of this lemma is a straightforward rewrite of Lemma 7.3.9: simply substitute '$sig(G_o)$' for 'Σ', 'A_o' for 'A', and 'P_o' for 'P'. □

7.3.11 Compiling Function Programs.

We are now in a position to explain our strategy for compiling arbitrary function programs into PR.

Our objective is to define a map $c^{FP}:\mathrm{FPIT}(\Sigma)\longrightarrow\mathrm{PR}(\Sigma)$, such that if $S\in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v\in S^+$, then $\alpha=c^{FP}(S)\in \mathrm{PR}(\Sigma)_{u,v}$ with

$$[\![c^{FP}(S)]\!]_A=F_A(S) \tag{165}$$

and

$$\lambda_P^{PR}(c^{FP}(S))\approx\lambda_P^{FP}(S) \tag{166}$$

Thus c^{FP} will be correct with respect to $[\![\,.\,]\!]_A$ and F_A, and performance preserving with respect to λ_P^{PR} and λ_P^{FP}.

As we shall see below, in defining c^{FP} we will use an auxiliary compiler called c^{FG}; for each $G\in \mathrm{FG}(\Sigma)$, $c^{FG}(G)$ is a mapping

$$c^{FG}(G):\mathrm{PR}(sig(G))\longrightarrow\mathrm{PR}(\Sigma)$$

which, as we will also explain below, is actually a correct and performance preserving compiler.

To begin with, let us consider the problem of compiling function programs $S\in \mathrm{FPIT}(\Sigma)_{u,v}$ for given $u,v\in S^+$. First suppose S is simple, that is suppose S is of the form:

$$S=\textbf{function } y_1,\ldots,y_m=f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n):S_o$$

where:

$y = (y_1,....,y_m) \in Var_v$;

$x = (x_1,....,x_n) \in Var_u$, and

$(S_o,x,y) \in \text{PITIO}(\Sigma)_{u,v}$.

Now, $F_A(S)$ is defined by $F_A(S)(a) = F_A(S_o,x,y)(a)$ for each $a \in A^u$, and so we need to find some α such that $[\![\alpha]\!]_A(a) = F_A(S_o,x,y)(a)$ for each $a \in A^u$. However, $\alpha = c^{IO}(\Sigma)(S_o,x,y)$ is precisely such a scheme by Lemma 7.3.6. Thus if we define $c^{FP}(S)$ by

$$c^{FP}(S) = c^{IO}(\Sigma)(S_o,x,y) \tag{167}$$

then

$$\begin{aligned}[\![c^{FP}(S)]\!]_A &= [\![c^{IO}(\Sigma)(S_o,x,y)]\!]_A \\ &= F_A(S_o,x,y) \\ &= F_A(S)\end{aligned}$$

That is,

$$[\![c^{FP}(S)]\!]_A = F_A(S) \tag{168}$$

Furthermore, $\lambda_P^{FP}(S)$ is defined by $\lambda_P^{FP}(S)(a) = \lambda_P^{IO}(S_o,x,y)(a)$ for each $a \in A^u$, and we know that $c^{IO}(\Sigma)$ is performance preserving, thus

$$\begin{aligned}\lambda_P^{PR}(c^{FP}(S)) &= \lambda_P^{PR}(c^{IO}(S_o,x,y)) \\ &\approx \lambda_P^{IO}(S_o,x,y) \\ &= \lambda_P^{FP}(S)\end{aligned}$$

That is,

$$\lambda_P^{PR}(c^{FP}(S)) \approx \lambda_P^{FP}(S) \tag{169}$$

Hence c^{FP} correctly compiles S to a scheme of equivalent complexity as required.

Now suppose S is nested, that is, suppose S is of the form:

$$S = \textbf{function } y_1,....,y_m = f_1(x_1,....,x_n),....,f_m(x_1,....,x_n) : G : S_o$$

where:

$y = (y_1,....,y_m) \in Var_v$;

$x = (x_1,....,x_n) \in Var_u$;

$G \in \text{FG}(\Sigma)$, and

$(S_o,x,y) \in \text{PITIO}(sig(G))_{u,v}$

Here $F_A(S)$ is defined by $F_A(S)(a) = F_{A_o}(S_o,x,y)(a)$ for each $a \in A^u$, and so we need to find some $\alpha \in \text{PR}(\Sigma)_{u,v}$ such that $[\![\alpha]\!]_A(a) = F_{A_o}(S_o,x,y)(a)$ for each $a \in A^u$. Now consider the scheme $\alpha' = c^{IO}(sig(G))(S_o,x,y)$: since $sig(G)$ is a (standard) signature, we know from Lemma 7.3.6 that $c^{IO}(sig(G))$ is a compiler from PITIO($sig(G)$) into PR($sig(G)$), and since $(S_o,x,y) \in \text{PITIO}(sig(G))_{u,v}$ by hypothesis, we have $\alpha' \in \text{PR}(sig(G))$. Furthermore, since $c^{IO}(sig(G))$ is correct and performance preserving we have

$$[\![\alpha']\!]_{A_o} = F_{A_o}(S_o,x,y) = F_A(S)$$

where $A_G = Alg_A(G)$, and

$$\lambda_{P_o}^{PR}(\alpha') \approx \lambda_{P_o}^{IO}(S_o,x,y) = \lambda_P^{FP}(S)$$

Thus α' would be the required implementation $c^{FP}(S)$ if only α' were a scheme over Σ instead of $sig(G)$.

In fact, as we mentioned above and will explain below, the group G determines a correct and performance preserving compiler $c^{FG}(G)$ from PR($sig(G)$) into PR(Σ). We can apply $c^{FG}(G)$ to α' to obtain the required implementation of S: Define $c^{FP}(S)$ by

$$c^{FP}(S) = c^{FG}(G)(\alpha') = c^{FG}(G)(c^{IO}(sig(G))(S_o,x,y))$$

Then,

$$\begin{aligned}[\![c^{FP}(S)]\!]_A &= [\![c^{FG}(G)(c^{IO}(sig(G))(S_o,x,y))]\!]_A \\ &= [\![c^{IO}(sig(G))(S_o,x,y)]\!]_{A_o} \\ &= F_{A_o}(S_o,x,y) \\ &= F_A(S)\end{aligned}$$

That is,

$$[\![c^{FP}(S)]\!]_A = F_A(S) \tag{170}$$

Furthermore,

$$\begin{aligned}\lambda_P^{PR}(c^{FP}(S)) &= \lambda_P^{PR}(c^{FG}(G)(c^{IO}(sig(G))(S_o,x,y))) \\ &\approx \lambda_{P_o}^{PR}(c^{IO}(sig(G))(S_o,x,y))) \\ &\approx \lambda_{P_o}^{IO}(S_o,x,y) \\ &= \lambda_P^{FP}(S)\end{aligned}$$

That is,

$$\lambda_P^{PR}(c^{FP}(S)) \approx \lambda_P^{FP}(S) \tag{171}$$

Hence c^{FP} correctly compiles S to a scheme of equivalent complexity as required.

Function Groups. We claimed above that each $G \in FG(\Sigma)$ determines a correct and performance preserving compiler $c^{FG}(G)$ form PR($sig(G)$) into PR(Σ). Here we will informally explain our strategy for constructing such transformations in some simple cases. (We will formally define both c^{FP} and c^{FG} in all cases in the next section.)

First suppose G is a single function group; that is, suppose G is of the form $G = S$ for some $S \in FPIT(\Sigma)$. Now consider a scheme $\alpha' \in PR(sig(G))$. Then α' involves (as basic operations) the identifiers of S as well as the symbols of Σ, and so to transform α' into a scheme over Σ, we must replace each occurrence of each $f \in id(S)$ in α' by some $\alpha_f \in PR(\Sigma)$.

Now suppose α is some scheme with $\alpha \in PR(\Sigma)_{u,v}$ where (u,v) is the arity of S. If

$$[\![\alpha]\!]_A = F_A(S) \tag{172}$$

and

$$\lambda_P^{PR}(\alpha) \approx \lambda_P^{FP}(S) \tag{173}$$

then by Lemma 7.3.9, $c^{AUG}(\Sigma)(id(S),\alpha)$ is the required transform of α'. To see this explicitly, define $c^{FG}(G)$ by $c^{FG}(G)(\alpha')=c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')$ for each $\alpha' \in \mathrm{PR}(sig(G))$. Then by Lemma 7.3.9(i), $c^{FG}(G)$ is a word-indexed mapping

$$c^{FG}(G):\mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

Furthermore, for any $\alpha' \in \mathrm{PR}(sig(G))$ we have

$$\begin{aligned}[\![c^{FG}(G)(\alpha')]\!]_A &= [\![c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')]\!]_A \\ &= [\![\alpha']\!]_{A_o}\end{aligned}$$

by Lemma 7.3.9(ii).

Furthermore,

$$\begin{aligned}\lambda_P^{PR}(c^{FG}(G)(\alpha')) &= \lambda_P^{PR}(c^{AUG}(\Sigma)(id(S),\alpha)(\alpha') \\ &\approx \lambda_{P_o}^{PR}(\alpha')\end{aligned}$$

by Lemma 7.3.9(iii).

We see that c^{FG} correctly compiles each scheme over $sig(G)$ to a scheme over Σ with equivalent performance. However, the construction of $c^{FG}(G)$ relies on our finding some $\alpha \in \mathrm{PR}(\Sigma)$ satisfying (172) and (173) above. However, as we have previously explained, $\alpha=c^{FP}(S)$ is intended to be precisely such a scheme: see (168) and (169) in the case that S is simple (we leave the case of S nested to the formal definitions below).

Now suppose G is a multiple function group; that is, suppose G is of the form $G=G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(\Sigma)$.

Since G_o is a shorter function group than G, that is, $|G_o|<|G|$, let us imagine that by an inductive argument on the length of G that we have defined $c^{FG}(G_o)$ and established that it is a correct and performance preserving compiler from $\mathrm{PR}(sig(G_o))$ into $\mathrm{PR}(\Sigma)$. Also suppose there is some scheme α such that $\alpha \in \mathrm{PR}(sig(G_o))_{u,v}$ when (u,v) is the arity of S, and

$$[\![\alpha]\!]_{A_o} = F_{A_o}(S) \tag{174}$$

where $A_o=Alg_A(G_o)$, and

$$\lambda_{P_o}^{PR}(\alpha) \approx \lambda_{P_o}^{FP}(S) \tag{175}$$

where $P_o=P_{G_o}$. (Recall that since S is a function program over $sig(G_o)$, the meaning of S is a meaning in A_o, and the performance of S is a performance with respect to P_o.)

Under these hypotheses on α, we know that $c^{AUG}(sig(G))(id(S),\alpha)(\alpha')$ is a correct and performance preserving compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(sig(G_o))$ by Lemma 7.3.10. Thus, if we now define $c^{FG}(G)$ by

$$c^{FG}(G)(\alpha') = c^{FG}(G_o)(c^{AUG}(sig(G))(id(S),\alpha)(\alpha'))$$

then it is not difficult to see that $c^{FG}(G)$ will be correct and performance preserving compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$. For example, it is easy to show that c^{FG} *is* a compiler: if $\alpha' \in \mathrm{PR}(sig(G))$, then α'' defined by

$$\alpha'' = c^{AUG}(sig(G))(id(S),\alpha)(\alpha')$$

is a member of PR($sig(G_o)$) (since $c^{AUG}(sig(G))(id(S),\alpha)$ is a compiler from PR($sig(G)$) into PR($sig(G_o)$)). Now, $c^{FG}(G)(\alpha)$ is defined by

$$c^{FG}(G)(\alpha') = c^{FG}(G_o)(\alpha')$$

and so $c^{FG}(G)(\alpha') \in \mathrm{PR}(\Sigma)$ (since $c^{FG}(G_o)$ is a compiler from PR($sig(G_o)$) into PR(Σ)).

Furthermore, it is simple to see that $c^{FG}(G)$ is correct: choose $\alpha' \in \mathrm{PR}(sig(G))$, then

$$\begin{aligned}[\![c^{FG}(G)(\alpha')]\!]_A &= [\![c^{FG}(G_o)(c^{AUG}(sig(G))(id(S),\alpha)(\alpha'))]\!]_A \\ &= [\![c^{AUG}(sig(G))(id(S),\alpha)(\alpha')]\!]_{A_o} \\ &= [\![\alpha']\!]_{A_o}\end{aligned}$$

Performance preservation of c^{FG} is equally easy to prove using the performance preservation of $c^{FG}(G_o)$ and $c^{AUG}(sig(G))(id(S),\alpha)$.

Let us now return to be the problem of finding $\alpha \in \mathrm{PR}(sig(G_o))$ (the scheme which we assumed to exist satisfying (174) and (175) above).

Since S is a function program over $sig(G_o)$, and since we have argued above that c^{FP} correctly compiles function programs over a signature Σ into a PR scheme over Σ, it is reasonable to expect that $c^{FP}(S)$ is a PR scheme over $sig(G_o)$ (and which implements S). Whilst this is essentially correct, we will get into difficulties when we try to use formal analysis unless we make c^{FP} explicitly dependent on the underlying signature Σ: we will subsequently write $c^{FP}(\Sigma)$ for c^{FP} and $c^{FP}(\Sigma)(S)$ for $c^{FP}(S)$. As we will see in the forthcoming formal definition of $c^{FP} = c^{FP}(\Sigma)$ and c^{FG}, this change in notation means that $c^{FG}(G)$ must be written $c^{FG}(\Sigma)(G)$ when G is a function group over Σ. □

We are now in a position to formally define the two compilers which we use to compile function programs into PR-schema.

7.3.12 Definition.

Let Σ be any signature. Below we define

(1) The *FPIT-PR compiler* $c^{FP}(\Sigma) : \mathrm{FPIT}(\Sigma) \longrightarrow \mathrm{PR}(\Sigma)$, and

(2) For each $G \in \mathrm{FG}(\Sigma)$, a map $c^{FG}(\Sigma)(G)$ which we call an *FG-compiler*: for each $G \in \mathrm{FG}(\Sigma)$, $c^{FG}(\Sigma)(G) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$.

To establish that $c^{FP}(\Sigma)$ *is* a compiler, for each $S \in \mathrm{FPIT}(\Sigma)$ we will first define $c^{FP}(\Sigma)(S)$ and then show that $c^{FP}(\Sigma)(S) \in \mathrm{PR}(\Sigma)_{u,v}$ when S is of arity (u,v). Similarly, for each $G \in \mathrm{FG}(\Sigma)$ we first define $c^{FG}(\Sigma)(G)$ and then show it is a word-indexed mapping from PR($sig(G)$) into PR(Σ).

The definitions of $c^{FP}(\Sigma)(S)$ and $c^{FG}(\Sigma)(G)$ are uniform in Σ and by simultaneous induction on the depth of S and G. Fix $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$ and $G \in \mathrm{FG}(\Sigma)$: then the definitions of $c^{FP}(\Sigma)(S)$ and $c^{FG}(\Sigma)(G)$ proceed as follows:

Basis Case (1). Suppose $S \in \mathrm{FPIT}(\Sigma)$ with $depth(S)=0$: then S is a simple function program of the form:

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$(S_o,x,y) \in \mathrm{PITIO}(\Sigma)_{u,v}$

Construction. In this case we define $c^{FP}(\Sigma)(S)$ by

$$c^{FP}(\Sigma)(S) = c^{IO}(\Sigma)(S_o,x,y)$$

Well-definedness. It is easy to see that $c^{FP}(\Sigma)(S)$ is well-defined in this case: by Lemma 7.3.6, $c^{IO}(\Sigma)$ is a compiler from $\mathrm{PITIO}(\Sigma)$ into $\mathrm{PR}(\Sigma)$, and $(S_o,x,y) \in \mathrm{PITIO}(\Sigma)_{u,v}$. Hence,

$$c^{FP}(\Sigma)(S) = c^{IO}(\Sigma)(S_o,x,y) \in \mathrm{PR}(\Sigma)_{u,v}$$

as required.

Basis Case (2). Suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=0$. We will define $c^{FG}(\Sigma)(G)$ by sub-induction on the length of G as follows:

Sub-Basis. Suppose $|G|=1$: then G is of the form $G=S$ for some $S \in \mathrm{FPIT}(\Sigma)$.

Construction. In this sub-case we define $c^{FG}(\Sigma)(G)$ by

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha)$$

where $\alpha = c^{FP}(\Sigma)(S)$.

Well-Definedness. Suppose $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$. Then since $depth(G)=0$ it follows that $depth(S)=0$; thus, as in Basis Case (1) above, since $\alpha = c^{FP}(\Sigma)(S)$, we know that α is a well-defined member of $\mathrm{PR}(\Sigma)_{u,v}$. By Lemma 7.3.9 then, it follows that $c^{AUG}(\Sigma)(id(S),\alpha)$ is a compiler from $\mathrm{PR}(sig(G))$ to $\mathrm{PR}(\Sigma)$; that is,

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

as required.

Sub-Induction. Suppose that for some fixed $l \geq 1$ that for every function group $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=0$ and $|G|=l$, that we have defined $c^{FG}(\Sigma)(G)$ and established that

$$c^{FG}(\Sigma)(G) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

Now suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=0$ and $|G|=l+1$: then G must be of the form $G = G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_o))$.

Construction. In this case we define $c^{FG}(\Sigma)(G)$ by the composition:

$$c^{FG}(\Sigma)(G) = c^{FG}(\Sigma)(G_o) \circ c^{AUG}(sig(G_o))(id(S),\alpha)$$

where $\alpha = c^{FP}(sig(G_o))(S)$.

Well-Definedness. To show that $c^{FG}(\Sigma)$ is a compiler from PR($sig(G)$) into PR(Σ), we must show

$$c^{FG}(\Sigma): \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma) \tag{176}$$

However, since the composition of two word-indexed maps is again a word-indexed map, it follows from the definition of $c^{FG}(\Sigma)$, that (176) holds if we can show

$$c^{FG}(\Sigma)(G_o): \mathrm{PR}(sig(G_o)) \longrightarrow \mathrm{PR}(\Sigma) \tag{177}$$

and

$$c^{AUG}(sig(G_o))(id(S),\alpha): \mathrm{PITIO}(sig(G)) \longrightarrow \mathrm{PR}(sig(G_o)) \tag{178}$$

Since $|G| = l+1$ with $depth(G)=0$, it follows that $|G_o| = l$ and $depth(G_o)=0$. Thus by the sub-induction hypothesis applied to G_o we may immediately assume that (177) is satisfied.

To see that (178) also holds, again using the fact that $depth(G)=0$, it follows that $depth(S)=0$, and so if $S \in \mathrm{FPIT}(sig(G_o))_{u,v}$, then S must be a simple function program of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$(S_o,x,y) \in \mathrm{PITIO}(sig(G_o))_{u,v}$.

Also since $depth(S)=0$, $\alpha = c^{FP}(sig(G_o))(S)$ must be defined as in Basis Case (1) above; that is,

$$\alpha = c^{FP}(sig(G_o))(S) = c^{IO}(sig(G_o))(S_o,x,y)$$

Now, by Lemma 7.3.6 we know that $c^{IO}(sig(G_o))$ is a compiler from PITIO($sig(G_o)$) into PR($sig(G_o)$); that is,

$$c^{IO}(sig(G_o)): \mathrm{PITIO}(sig(G_o)) \longrightarrow \mathrm{PR}(sig(G_o))$$

But (S_o,x,y) is of arity (u,v) and so $\alpha = c^{IO}(sig(G_o)) \in \mathrm{PR}(sig(G_o))_{u,v}$. Thus, by Lemma 7.3.10, $c^{AUG}(sig(G_o))(id(S),\alpha)$ is a compiler from PR($sig(G)$) into PR($sig(G_o)$); that is,

$$c^{AUG}(sig(G_o))(id(S),\alpha): \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(sig(G_o))$$

Thus (178) also holds, completing the sub-induction, and also the basis case for function groups G.

Induction Cases. For some fixed $d \geq 0$ assume the following:

(1) For any signature Σ_o and for any $S \in \mathrm{FPIT}(\Sigma_o)$ with $depth(S) \leq d$, we have defined $c^{FPIT}(\Sigma_o)(S)$ and shown that it is a well-defined member of $\mathrm{PR}(\Sigma_o)_{u,v}$ when S is of arity (u,v).

(2) For any signature Σ_o and for any $G \in \mathrm{FG}(\Sigma_o)$ with $depth(G) \leq d$, we have defined $c^{FG}(\Sigma_o)(G)$ and shown that it is a compiler from PR($sig(G)$) into PR(Σ_o); that is,

$$c^{FG}(\Sigma_o)(G): \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma_o)$$

Induction Case (1). Suppose $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ with $depth(S)=d+1$: then S must be of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_0$$

for some $G \in \mathrm{FG}(\Sigma)$ and where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$(S_0,x,y) \in \mathrm{PITIO}(sig(G))_{u,v}$.

Construction. In this case we define $c^{FP}(\Sigma)(S)$ by the composition

$$c^{FP}(\Sigma)(S) = c^{FG}(\Sigma)(G) \circ c^{IO}(sig(G))(S_0,x,y)$$

Well-Definedness. To show that $c^{FP}(\Sigma)(S)$ is well-defined here we must show

$$c^{FP}(\Sigma)(S) \in \mathrm{PR}(\Sigma)_{u,v} \tag{179}$$

(since $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ by hypothesis). Now, since the composition of two word-indexed maps is a word-indexed map, it follows from the definition of $c^{FP}(\Sigma)(S)$, that (179) holds if we can show

$$c^{FG}(\Sigma)(G) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma) \tag{180}$$

and

$$c^{IO}(sig(G)) : \mathrm{PITIO}(sig(G)) \longrightarrow \mathrm{PR}(sig(G)) \tag{181}$$

Now, (181) certainly holds by Lemma 7.3.6, and so it remains to see that (180) holds. However, since $depth(S)=d+1$ it must be that $depth(G)=d$, and so by induction hypothesis (2) applied to G it follows that $c^{FG}(\Sigma)(G)$ is a compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$; that is,

$$c^{FG}(\Sigma)(G) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

This is (180) above of course.

Induction Case (2). Suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=d+1$. We will define $c^{FG}(\Sigma)(G)$ by sub-induction on the length of G as follows:

Sub-Basis. Suppose $|G| = 1$: then G is of the form $G = S$ for some $S \in \mathrm{FPIT}(\Sigma)$.

Construction. In this sub-case we define $c^{FG}(\Sigma)(G)$ by

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha)$$

where $\alpha = c^{FP}(\Sigma)(S)$.

Well-Definedness. Suppose $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$. Then since $depth(G)=d+1$ it follows that $depth(S)=d+1$; thus, as in Induction Case (1) above, since $\alpha = c^{FP}(\Sigma)(S)$, we know that α is a well-defined member of $\mathrm{PR}(\Sigma)_{u,v}$. By Lemma 7.3.9 then, it follows that $c^{AUG}(\Sigma)(id(S),\alpha)$ is a compiler from from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$, and thus

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

as required.

Sub-Induction. Suppose that for some fixed $l \geq 1$ that for every function group $G \in \mathrm{FG}(\Sigma)$ with $depth(G) = d+1$ and $|G| = l$, that we have defined $c^{FG}(\Sigma)(G)$ and established that

$$c^{FG}(\Sigma)(G) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(\Sigma)$$

Now suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G) = d+1$ and $|G| = l+1$: then G must be of the form $G = G_0;S$ for some $G_0 \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_0))$.

Construction. In this case we define $c^{FG}(\Sigma)(G)$ by the composition

$$c^{FG}(\Sigma)(G) = c^{FG}(\Sigma)(G_0) \circ c^{AUG}(sig(G_0))(id(S),\alpha)$$

where $\alpha = c^{FP}(sig(G_0))(S)$.

Well-Definedness. To show that $c^{FG}(\Sigma)(G)$ is well-defined we must show

$$c^{FG}(\Sigma)(G_0) : \mathrm{PR}(sig(G_0)) \longrightarrow \mathrm{PR}(\Sigma) \tag{182}$$

and

$$c^{AUG}(sig(G_0))(id(S),\alpha) : \mathrm{PR}(sig(G)) \longrightarrow \mathrm{PR}(sig(G_0)) \tag{183}$$

However, it follows from Lemma 7.3.10 that condition (182) will be satisfied if we can show that α is a well-defined member of $\mathrm{PR}(sig(G_0))_{u,v}$ when S is of arity (u,v); that is, if

$$\alpha = c^{FP}(\Sigma)(sig(G_0))(S) \in \mathrm{PR}(sig(G_0))_{u,v} \tag{184}$$

Thus it remains to show that (182) and (184) hold:

By definition of $depth(G)$, $depth(G) = \max\{depth(G_0), depth(S)\}$. But $depth(G) = d+1$ here, and so there are three possibilities for the depths of G_0 and S:

Case (a): $depth(G_0) \leq d$ and $depth(S) = d+1$
Case (b): $depth(G_0) = d+1$ and $depth(S) = d+1$
Case (c): $depth(G_0) = d+1$ and $depth(S) \leq d$

We will now establish that (182) and (184) are satisfied in each of these cases.

Case (a). Suppose $depth(G_0) \leq d$ and $depth(S) = d+1$. Then by the main induction hypothesis (2) applied to G_0 we may assume that $c^{FG}(\Sigma)(G_0)$ is a well-defined map of the form:

$$c^{FG}(\Sigma)(G_0) : \mathrm{PR}(sig(G_0)) \longrightarrow \mathrm{PR}(\Sigma)$$

This is precisely condition (182) above of course.

Also, since $S \in \mathrm{FPIT}(sig(G_0))$ with $depth(S) = d+1$ it must be that S is a nested function program of the form

$$S = \textbf{function } y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G' : S_0$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$;

$G' \in \mathrm{FG}(sig(G_0))$, and

$(S_0,x,y) \in \mathrm{PITIO}(sig(G'))$

Now, since $depth(S) = d+1$, it must be that $c^{FP}(sig(G_0))(S)$ is as defined in Induction Case (1) above,

namely,

$$c^{FP}(sig(G_o))(S) = c^{FG}(sig(G_o))(G') \circ c^{IO}(sig(G'))(S_o, x, y)$$

Thus to show that (184) is satisfied, we must show

$$c^{FG}(sig(G_o))(G') : \mathrm{PR}(sig(G')) \longrightarrow \mathrm{PR}(sig(G_o)) \tag{185}$$

and

$$c^{IO}(sig(G')) : \mathrm{PITIO}(sig(G')) \longrightarrow \mathrm{PR}(sig(G')) \tag{186}$$

Now, (186) is certainly true by Lemma 7.3.6, and so it remains to show that (185) is satisfied. However, since $depth(S) = d+1$, it must be that $depth(G') = d$, and so by the main induction hypothesis (2) applied to G' we have

$$c^{FG}(sig(G_o))(G') : \mathrm{PR}(sig(G')) \longrightarrow \mathrm{PR}(sig(G_o))$$

This is precisely (185) of course, and so (184) is satisfied as claimed.

Case (b). Suppose $depth(G_o) = d+1$ and $depth(S) = d+1$. Then since $|G| = l+1$ it follows that $|G_o| = l$. Thus by the sub-induction hypothesis (2) applied to G_o we may assume that $c^{FG}(\Sigma)(G_o)$ is a well-defined map of the form:

$$c^{FG}(\Sigma)(G_o) : \mathrm{PR}(sig(G_o)) \longrightarrow \mathrm{PR}(\Sigma)$$

This is condition (182) of course.

Since $depth(S) = d+1$, condition (184) holds in this case for exactly the same reasons that it did in Case (a) above.

Case (c). Suppose $depth(G_o) = d+1$ and $depth(S) \leq d$. Then since $depth(G_o) = d+1$ it follows that condition (182) holds for exactly the same reasons that it held in Case (a) above.

Finally, it is easy to see that condition (184) holds in this case by the (main) induction hypothesis (1).

This completes the sub-induction and also the main induction for function groups. □

7.4 VERIFICATION OF THE FPIT-PR COMPILER.

We now prove $c^{FP}(\Sigma)$ is correct and performance preserving.

7.4.1 Theorem. *Let Σ be any signature, let A be a Σ-algebra, and let P be a performance measure for A. Then*

(1) $c^{FP}(\Sigma)$ *is correct with respect to* $[\![.]\!]_A$ *and* F_A*; that is, for every* $S \in \mathrm{FPIT}(\Sigma)$,

$$[\![c^{FP}(\Sigma)(S)]\!]_A = F_A(S) \tag{187}$$

Furthermore, $c^{FP}(\Sigma)$ *is a performance preserving compiler with respect to* λ_P^{PR} *and* λ_P^{FP}*; that is, for every* $S \in \mathrm{FPIT}(\Sigma)$,

$$\lambda_P^{PR}(c^{FP}(\Sigma)(S)) \approx \lambda_P^{FP}(S) \tag{188}$$

(2) for every $G \in \mathrm{FG}(\Sigma)$, $c^{FG}(\Sigma)(G)$ *is correct with respect to* $[\![.]\!]_A$ *and* $[\![.]\!]_{A_G}$ *where* $A_G = Alg_A(G)$*; that is,*

$$(\forall \alpha \in \mathrm{PR}(sig(G)))\ (\ [\![c^{FG}(\Sigma)(G)(\alpha)]\!]_A = [\![\alpha]\!]_{A_G}\ ) \tag{189}$$

Furthermore, $c^{FG}(\Sigma)(G)$ *is a performance preserving compiler with respect to* λ_P^{PR} *and* $\lambda_{P_G}^{PR}$*; that is,*

$$(\forall \alpha \in \mathrm{PR}(sig(G)))\ (\ \lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha)) \approx \lambda_{P_G}^{PR}(\alpha)\ ) \tag{190}$$

Proof. Let $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$, and let $G \in \mathrm{FG}(\Sigma)$. We will prove that (187) and (188) hold for S, and that (189) and (190) hold for G simultaneously and by induction on the depth of S and G respectively; additionally the proof is uniform in Σ.

Basis Case (1). Suppose $depth(S)=0$: then S is a simple function program of the form:

$$S = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_0$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$(S_0,x,y) \in \mathrm{PITIO}(\Sigma)$

In this case $c^{FP}(\Sigma)(S)$ was defined by

$$c^{FP}(\Sigma)(S) = c^{IO}(\Sigma)(S_0,x,y)$$

Correctness. It is easy to see that $c^{FP}(\Sigma)(S)$ is correct in this case:

$$[\![c^{FP}(\Sigma)(S)]\!]_A = [\![c^{IO}(\Sigma)(S_0,x,y)]\!]_A$$
$$= F_A(S_0,x,y)$$

(since $c^{IO}(\Sigma)$ is correct; see Lemma 7.3.6(ii))

$$= F_A(S)$$

(by definition of F_A).

Thus (187) holds for function programs of depth 0; that is, $c^{FP}(\Sigma)$ correctly compiles function programs of depth 0.

Performance Preservation. It is equally easy to see that $c^{FP}(\Sigma)(S)$ is performance preserving:

$$\lambda_P^{PR}(c^{FP}(\Sigma)(S)) = \lambda_P^{PR}(c^{IO}(\Sigma)(S_o,x,y))$$
$$\simeq \lambda_P^{IO}(S_o,x,y)$$

(since $c^{IO}(\Sigma)$ is performance preserving; see Lemma 7.3.6(iii))

$$= \lambda_P^{FP}(S)$$

(by definition of λ_P^{FP}).

Thus (188) holds for function programs of depth 0; that is, $c^{FP}(\Sigma)$ compiles any function program of depth 0 into a scheme of equivalent complexity.

Basis Case (2). Suppose $G \in FG(\Sigma)$ with $depth(G)=0$. We will prove that (189) and (190) hold by sub-induction on the length of G as follows:

Sub-Basis. Suppose $|G| = 1$: then G is of the form $G = S$ for some $S \in FPIT(\Sigma)$.

In this sub-case $c^{FG}(\Sigma)(G)$ was defined by

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha)$$

where $\alpha = c^{FP}(\Sigma)(S)$.

Correctness. First notice that $\alpha \in PR(\Sigma)_{u,v}$ (since $c^{FP}(\Sigma)$ is a compiler from $FPIT(\Sigma)$ into $PR(\Sigma)$ and S is of arity (u,v)). Also, since $depth(G)=0$ it follows that $depth(S)=0$; thus, as in Basis Case (1) above, it is easy to prove that $[\![\alpha]\!]_A = F_A(S)$. Hence, by Lemma 7.3.9(ii), for every $\alpha' \in PR(sig(G))$,

$$[\![c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o} \qquad (191)$$

where $A_G = Alg_A(G)$.

Thus, for every $\alpha' \in PR(sig(G))$,

$$[\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')]\!]_A$$
$$= [\![\alpha']\!]_{A_o}$$

(from (191)).

Hence (187) holds for function groups of depth 0 and length 1; that is, for such function groups G $c^{FG}(\Sigma)(G)$ is correct with respect to A and A_G as claimed.

Performance Preservation. Again since $depth(S)=0$ and $\alpha \in PR(\Sigma)_{u,v}$, it is easy to prove that as in Basis Case (1) above, $\lambda_P^{PR}(\alpha) \simeq \lambda_P^{FP}(S)$. Hence, by Lemma 7.3.9(iii), for every $\alpha' \in PR(sig(G))$,

$$\lambda_P^{PR}(c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')) \simeq \lambda_{P_o}^{PR}(\alpha') \qquad (192)$$

Thus, for every $\alpha' \in PR(sig(G))$,

$$\lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha')) = \lambda_P^{PR}(c^{AUG}(\Sigma)(id(S),\alpha)(\alpha'))$$
$$\simeq \lambda_{P_o}^{PR}(\alpha')$$

(using (192)).

Hence (190) holds for function groups of depth 0 and length 1; that is, for such function groups G $c^{FG}(\Sigma)(G)$ is performance preserving with respect to P and P_G as claimed.

Sub-Induction. Suppose that for some fixed $l \geq 1$ that for every function group $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=0$ and $|G|=l$, that we have proved $c^{FG}(\Sigma)(G)$ correct and performance preserving; that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ [\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \tag{193}$$

and

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ \lambda_p^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{p_o}^{PR}(\alpha')\ ) \tag{194}$$

Now suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=0$ and $|G|=l+1$: then G must be of the form $G=G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_o))$.

In this case $c^{FG}(\Sigma)(G)$ was defined by the composition:

$$c^{FG}(\Sigma)(G) = c^{FG}(\Sigma)(G_o) \circ c^{AUG}(sig(G_o))(id(S),\alpha)$$

where $\alpha = c^{FP}(sig(G_o))(S)$.

Correctness. To show that $c^{FG}(\Sigma)$ is a correct compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$, we must show

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ [\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \tag{195}$$

To see that (195) holds, let us first assume that

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ (\ [\![c^{FG}(\Sigma)(G_o)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \tag{196}$$

and

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ [\![c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')]\!]_{A_o} = [\![\alpha']\!]_{A_o}\ ) \tag{197}$$

where $A_G = Alg_A(G)$ and $A_o = Alg_A(G_o)$.

Now choose $\alpha' \in \mathrm{PR}(sig(G))$, then by definition of $c^{FG}(\Sigma)(G)$ we have

$$\begin{aligned} [\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A &= [\![c^{FG}(\Sigma)(G_o) \circ c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha')]\!]_A \\ &= [\![c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha')]\!]_{A_o} \end{aligned}$$

(by (196))

$$= [\![\alpha']\!]_{A_o}$$

(by (197)).

Hence (189) holds for function groups of depth 0 and length l+1: by the principle of mathematical induction, (189) therefore holds for function groups of depth 0 and of any length l; that is, for any G with $depth(G)=0$, $c^{FG}(\Sigma)(G)$ is correct with respect to A and A_G as claimed.

It remains to show that (196) and (197) hold.

Since $|G|=l+1$ with $depth(G)=0$, it follows that $|G_o|=l$ and $depth(G_o)=0$. Thus by the sub-induction hypothesis (193) applied to G_o we may immediately assume that (196) is satisfied.

To see that (197) also holds, again using the fact that $depth(G)=0$, it follows that $depth(S)=0$, and so if S is of arity (u,v), then S must be a simple function program of the form

$$S = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : S_o$$

where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$$(S_o, x, y) \in \mathrm{PITIO}(sig(G_o))_{u,v}$$

Also since $depth(S)=0$, $\alpha = c^{FP}(sig(G_o))(S)$ is defined by

$$\alpha = c^{FP}(sig(G_o))(S) = c^{IO}(sig(G_o))(S_o, x, y)$$

Now, by Lemma 7.3.6(ii) we know that $c^{IO}(sig(G_o))$ is a correct compiler from $\mathrm{PITIO}(sig(G_o))$ into $\mathrm{PR}(sig(G_o))$; that is,

$$(\forall (S, In, Out) \in \mathrm{PITIO}(sig(G_o)))\ (\ [\![c^{IO}(sig(G_o))(S, In, Out)]\!]_{A_\bullet} = F_{A_\bullet}(S, In, Out)\ ) \qquad (198)$$

Now, since $\alpha = c^{IO}(sig(G_o))(S_o, x, y)$, it follows from (198) that

$$[\![\alpha]\!]_{A_\bullet} = [\![c^{IO}(sig(G_o))(S_o, x, y)]\!]_{A_\bullet} = F_{A_\bullet}(S_o, x, y)$$

Thus, by Lemma 7.3.10(ii), $c^{AUG}(sig(G_o))(id(S), \alpha)$ is a correct compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(sig(G_o))$; that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ [\![c^{AUG}(sig(G_o))(id(S), \alpha)(\alpha')]\!]_{A_\bullet} = [\![\alpha']\!]_{A_o}\ )$$

Thus (197) also holds, completing the correctness sub-induction, and also the basis case for the correctness of $c^{FG}(\Sigma)(G)$.

Performance Preservation. To show that $c^{FG}(\Sigma)$ is a performance preserving compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$, we must show

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ \lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{P_\bullet}^{PR}(\alpha)\ ) \qquad (199)$$

To see that (198) holds, let us first assume

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ (\ \lambda_P^{PR}(c^{FG}(\Sigma)(G_o)(\alpha')) \approx \lambda_{P_\bullet}^{PR}(\alpha')\ ) \qquad (200)$$

and

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ \lambda_{P_\bullet}^{PR}(c^{AUG}(\Sigma)(id(S), \alpha)(\alpha')) \approx \lambda_{P_o}^{PR}(\alpha')\ ) \qquad (201)$$

where $P_o = P_{G_\bullet}$.

Now choose $\alpha' \in \mathrm{PR}(sig(G))$. Then from the definition of $c^{FG}(\Sigma)(G)$ we have:

$$\begin{aligned}\lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha')) &= \lambda_P^{PR}(c^{FG}(\Sigma)(G_o) \circ c^{AUG}(sig(G_o))(id(S), \alpha)(\alpha')) \\ &\approx \lambda_{P_\bullet}^{PR}(c^{AUG}(sig(G_o))(id(S), \alpha)(\alpha'))\end{aligned}$$

(by (200))

$$\approx \lambda_{P_o}^{PR}(\alpha')$$

(by (201)).

Hence (190) holds for function groups of depth 0 and length 1+1: by the principle of mathematical induction, (190) therefore holds for function groups of depth 0 and of any length l; that is, for any G with $depth(G)=0$, $c^{FG}(\Sigma)(G)$ is performance preserving with respect to P and P_G as claimed.

It remains to show that (200) and (201) hold.

Since $|G_o| = l$ and $depth(G_o)=0$, we can apply the sub-induction hypothesis (194) to G_o immediately yielding (200).

To see that (201) also holds, first notice that by Lemma 7.3.6(iii) we know that $c^{IO}(sig(G_o))$ is a performance preserving compiler from PITIO($sig(G_o)$) into PR($sig(G_o)$); that is,

$$(\forall (S,In,Out) \in \text{PITIO}(sig(G_o)))\ (\ \lambda_{P_\bullet}^{PR}(c^{IO}(sig(G_o))(S,In,Out)) \approx \lambda_{P_\bullet}^{IO}(S,In,Out)\ ) \qquad (202)$$

Now, since $\alpha = c^{IO}(sig(G_o))(S_o,x,y)$, it follows from (202) that

$$\lambda_{P_\bullet}^{PR}(c^{IO}(sig(G_o))(S_o,x,y)) \approx \lambda_{P_\bullet}^{IO}(S_o,x,y)$$

Thus, by Lemma 7.3.10(iii), $c^{AUG}(sig(G_o))(id(S),\alpha)$ is a performance preserving compiler from PR($sig(G)$) into PR($sig(G_o)$); that is,

$$(\forall \alpha' \in \text{PR}(sig(G)))\ (\ \lambda_{P_\bullet}^{PR}(c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha')) = \lambda_{P_\bullet}^{PR}(\alpha')\ )$$

Thus (201) also holds as required.

Induction Cases. Let Σ_o be any signature, and let A_o be a Σ_o-algebra, and let P_o be a performance measure for A_o. Then for some fixed $d \geq 0$ assume the following:

(1) for every $S \in \text{FPIT}(\Sigma_o)$ with $depth(S) \leq d$, $c^{FP}(\Sigma)$ correctly compiles S into a scheme of equivalent performance; that is, we assume

$$[\![c^{FP}(\Sigma)(S)]\!]_A = F_A(S) \qquad (203)$$

and

$$\lambda_P^{PR}(c^{FP}(\Sigma)(S)) \approx \lambda_P^{FP}(S) \qquad (204)$$

(2) for every $G \in \text{FG}(\Sigma_o)$ with $depth(G) \leq d$, $c^{FG}(\Sigma)(G)$ is a correct and performance preserving compiler from PR($sig(G)$) into PR(Σ); that is, we assume

$$(\forall \alpha' \in \text{PR}(sig(G)))\ (\ [\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \qquad (205)$$

and

$$(\forall \alpha' \in \text{PR}(sig(G)))\ (\ \lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{P_o}^{PR}(\alpha')\ ) \qquad (206)$$

Induction Case (1). Suppose $S \in \text{FPIT}(\Sigma)_{u,v}$ with $depth(S) = d+1$: then S must be of the form

$$S = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G : S_o$$

for some $G \in \text{FG}(\Sigma)$ and where:

$y = (y_1,\ldots,y_m) \in Var_v$;

$x = (x_1,\ldots,x_n) \in Var_u$, and

$(S_o,x,y) \in \text{PITIO}(sig(G))_{u,v}$.

In this case $c^{FP}(\Sigma)(S)$ was defined by the composition

$$c^{FP}(\Sigma)(S) = c^{FG}(\Sigma)(G) \bullet c^{IO}(sig(G))(S_o,x,y)$$

Correctness. To show that $c^{FP}(\Sigma)(S)$ is correct we must show

$$[\![c^{FP}(\Sigma)(S)]\!]_A = F_A(S) \qquad (207)$$

To show that (207) holds first assume:

$$(\forall \alpha' \in \text{PR}(sig(G)))\ (\ [\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \qquad (208)$$

and

$$(\forall (S,In,Out) \in \mathrm{PITIO}(sig(G)))\ \ ([\![c^{IO}(sig(G))(S,In,Out)]\!]_{A_o} = F_{A_o}(S,In,Out)) \qquad (209)$$

where $A_G = Alg_A(G)$. Then by definition of $c^{FP}(\Sigma)(S)$ we have

$$[\![c^{FP}(\Sigma)(S)]\!]_A = [\![c^{FG}(\Sigma)(G) \circ c^{IO}(sig(G))(S_o,x,y)]\!]_A$$
$$= [\![c^{IO}(sig(G))(S_o,x,y)]\!]_{A_o}$$

(by (208))

$$= F_{A_o}(S_o,x,y)$$

(by (209))

$$= F_A(S)$$

(by definition of $F_A(S)$).

Thus (187) holds for function programs of depth d+1; that is, $c^{FP}(\Sigma)$ correctly compiles function programs of depth d+1.

It remains to show that (208) and (209) hold.

First, (209) certainly holds by Lemma 7.3.6(ii), and so it remains to see that (208) holds. However, since $depth(S)=d+1$ it must be that $depth(G)=d$, and so by the induction hypothesis (205) applied to G it follows that $c^{FG}(\Sigma)(G)$ is a correct compiler from $\mathrm{PR}(sig(G))$ into $\mathrm{PR}(\Sigma)$; that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ \ ([\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o})$$

This is (208) above of course.

Performance Preservation. To show that $c^{FP}(\Sigma)(S)$ is performance preserving we must show

$$\lambda_P^{PR}(c^{FP}(\Sigma)(S)) \approx \lambda_P^{FP}(S) \qquad (210)$$

To show that (210) holds first assume:

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ \ (\lambda_P^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{P_o}^{PR}(\alpha')) \qquad (211)$$

and

$$(\forall (S,In,Out) \in \mathrm{PITIO}(sig(G)))\ \ (\lambda_{P_o}^{PR}(c^{IO}(sig(G))(\alpha')) \approx \lambda_{P_o}^{IO}(S,In,Out)) \qquad (212)$$

Then by definition of $c^{FP}(\Sigma)(S)$ we have:

$$\lambda_P^{PR}(c^{FP}(\Sigma)(S)) = \lambda_P^{PR}(c^{FG}(\Sigma)(G) \circ c^{IO}(sig(G))(S_o,x,y))$$
$$\approx \lambda_{P_o}^{PR}(c^{IO}(sig(G))(S_o,x,y))$$

(by (211))

$$\approx \lambda_{P_o}^{IO}(S_o,x,y)$$

(by (212))

$$= \lambda_P^{FP}(S)$$

(by definition of $\lambda_P^{FP}(S)$).

Thus (188) holds for function programs of depth d+1; that is, $c^{FP}(\Sigma)$ compiles any function program of depth d+1 into a scheme of equivalent complexity.

It remains to show that (211) and (212) hold.

First, (212) certainly holds by Lemma 7.3.6(iii), and so it remains to see that (211) holds. However, since $depth(S)=d+1$ it must be that $depth(G)=d$, and so by induction hypothesis (206) applied to G it follows that $c^{FG}(\Sigma)(G)$ is a performance preserving compiler from PR($sig(G)$) into PR(Σ); that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ \ (\lambda_p^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{p_o}^{PR}(\alpha'))$$

This is (211) above of course.

Induction Case (2). Suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=d+1$. We will prove that $c^{FG}(\Sigma)(G)$ is correct and performance preserving by sub-induction on the length of G as follows:

Sub-Basis. Suppose $|G|=1$: then G is of the form $G=S$ for some $S \in \mathrm{FPIT}(\Sigma)$.

In this sub-case $c^{FG}(\Sigma)(G)$ was defined by

$$c^{FG}(\Sigma)(G) = c^{AUG}(\Sigma)(id(S),\alpha)$$

where $\alpha=c^{FP}(\Sigma)(S)$.

Correctness. Suppose $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ for some $u,v \in S^+$. Then since $depth(G)=d+1$ it follows that $depth(S)=d+1$; thus, as in Induction Case (1) above, since $\alpha=c^{FP}(\Sigma)(S)$, we know that α is a well-defined member of $\mathrm{PR}(\Sigma)_{u,v}$. By Lemma 7.3.9(ii) then, it follows that $c^{AUG}(\Sigma)(id(S),\alpha)$ is a correct compiler from from PR($sig(G)$) into PR(Σ), and thus for every $\alpha' \in \mathrm{PR}(sig(G))$,

$$[\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}$$

That is, $c^{FG}(\Sigma)(G)$ is correct as claimed.

Performance Preservation. Since $S \in \mathrm{FPIT}(\Sigma)_{u,v}$ and $depth(S)=d+1$, it follows as in Induction Case (1) above, that α is a well-defined member of $\mathrm{PR}(\Sigma)_{u,v}$. By Lemma 7.3.9(iii) then, it follows that $c^{AUG}(\Sigma)(id(S),\alpha)$ is a performance preserving compiler from from PR($sig(G)$) into PR(Σ), and thus for every $\alpha' \in \mathrm{PR}(sig(G))$,

$$\lambda_p^{PR}(c^{FG}(\Sigma)(G)(\alpha')) = \lambda_p^{PR}(c^{AUG}(\Sigma)(id(S),\alpha)(\alpha')) = \lambda_{p_o}^{PR}(\alpha')$$

That is, $c^{FG}(\Sigma)(G)$ is performance preserving as claimed.

Sub-Induction. Suppose that for some fixed $l \geq 1$ that for every function group $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=d+1$ and $|G|=l$, we have proved that $c^{FG}(\Sigma)(G)$ is a correct and performance preserving compiler; that is assume that we have shown:

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ \ ([\![c^{FG}(\Sigma)(G)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}) \tag{213}$$

and

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ \ (\lambda_p^{PR}(c^{FG}(\Sigma)(G)(\alpha')) \approx \lambda_{p_o}^{PR}(\alpha')) \tag{214}$$

Now suppose $G \in \mathrm{FG}(\Sigma)$ with $depth(G)=d+1$ and $|G|=l+1$: then G must be of the form $G=G_o;S$ for some $G_o \in \mathrm{FG}(\Sigma)$ and some $S \in \mathrm{FPIT}(sig(G_o))$.

In this case $c^{FG}(\Sigma)(G)$ was defined by the composition

$$c^{FG}(\Sigma)(G) = c^{FG}(\Sigma)(G_o) \circ c^{AUG}(sig(G_o))(id(S),\alpha)$$

where $\alpha=c^{FP}(sig(G_o))(S)$.

Correctness. As in the case of multiple function groups of depth 0, to show that $c^{FG}(\Sigma)(G)$ is correct it is sufficient for us show

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ (\ [\![c^{FG}(\Sigma)(G_o)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ ) \tag{215}$$

and

$$(\forall \alpha' \in \mathrm{PR}(sig(G)))\ (\ [\![c^{AUG}(sig(G_o))(id(S),\alpha)(\alpha')]\!]_{A_o} = [\![\alpha']\!]_{A_o}\ ) \tag{216}$$

where $A_G = Alg_A(G)$ and $A_o = Alg_A(G_o)$.

However, it follows from Lemma 7.3.10(iii) that condition (216) will be satisfied if we can show

$$[\![\alpha]\!]_{A_o} = [\![c^{FP}(sig(G_o))(S)]\!]_{A_o} = F_{A_o}(S) \tag{217}$$

Thus it remains to show that (215) and (217) hold:

By definition of $depth(G)$, $depth(G) = \max\{depth(G_o), depth(S)\}$. But $depth(G) = d+1$ here, and so there are three possibilities for the depths of G_o and S:

Case (a): $depth(G_o) \le d$ and $depth(S) = d+1$
Case (b): $depth(G_o) = d+1$ and $depth(S) = d+1$
Case (c): $depth(G_o) = d+1$ and $depth(S) \le d$

We will now establish that (215) and (217) are satisfied in each of these cases.

Case (a). Suppose $depth(G_o) \le d$ and $depth(S) = d+1$. Then by the main induction hypothesis (2) applied to G_o we may assume that $c^{FG}(\Sigma)(G_o)$ is a correct compiler from $\mathrm{PR}(sig(G_o))$ into $\mathrm{PR}(\Sigma)$; that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ (\ [\![c^{FG}(\Sigma)(G_o)(\alpha')]\!]_A = [\![\alpha']\!]_{A_o}\ )$$

This is precisely condition (215) above of course.

Also, since $S \in \mathrm{FPIT}(sig(G_o))$ with $depth(S) = d+1$ it must be that S is a nested function program of the form

$$S = \textbf{function}\ y_1,\ldots,y_m = f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n) : G' \mathbin{:} S_o$$

for some $G' \in \mathrm{FG}(sig(G_o))$, and where:

$y = (y_1,\ldots,y_m) \in Var_y$;

$x = (x_1,\ldots,x_n) \in Var_x$, and

$(S_o, x, y) \in \mathrm{PITIO}(sig(G'))$.

To show that (217) holds first notice that since $depth(S) = d+1$, $\alpha = c^{FP}(sig(G_o))(S)$ is as defined in Induction Case (1) above; that is,

$$\alpha = c^{FP}(sig(G_o))(S) = c^{FG}(sig(G_o))(G') \circ c^{IO}(sig(G'))(S_o, x, y)$$

Now assume

$$(\forall \alpha' \in \mathrm{PR}(sig(G')))\ (\ [\![c^{FG}(sig(G_o))(G')(\alpha')]\!]_{A_o} = [\![\alpha']\!]_{A'}\ ) \tag{218}$$

and

$$(\forall (S, In, Out) \in \mathrm{PITIO}(sig(G')))\ (\ [\![c^{IO}(sig(G'))(\alpha')]\!]_{A'} = F_{A'}(S, In, Out)\ ) \tag{219}$$

where $A_o = Alg_A(G_o)$ and $A' = Alg_{A_o}(G')$. Then by definition of α we have:

$$[\![\alpha]\!]_{A_\bullet}\ [\![c^{FG}(sig(G_0))(G') \circ c^{IO}(sig(G'))(S_0,x,y)]\!]_{A_\bullet}$$
$$= [\![c^{IO}(sig(G'))(S_0,x,y)]\!]_{A'}$$

(by (218))

$$= F_{A'}(S_0,x,y)$$

(by (219))

$$= F_{A_\bullet}(S)$$

(by definition of $F_{A_\bullet}(S)$).

Hence (217) holds as claimed, assuming we can show (218) and (219) hold of course:

Now, (219) is certainly true by Lemma 7.3.6(ii), and so it remains to show that (218) is satisfied. However, since $depth(S)=d+1$, it must be that $depth(G')=d$, and so by the main induction hypothesis (205) applied to G' we have

$$(\forall \alpha' \in \mathrm{PR}(sig(G'))) \ ([\![c^{FG}(sig(G_0))(G')(\alpha')]\!]_{A_\bullet} = [\![\alpha']\!]_{A'})$$

This is precisely (218) of course, and so (217) is satisfied as claimed.

Case (b). Suppose $depth(G_0)=d+1$ and $depth(S)=d+1$. Then since $|G|=l+1$ it follows that $|G_0|=l$. Thus by the sub-induction hypothesis (213) applied to G_0 we may assume that $c^{FG}(\Sigma)(G_0)$ is a correct compiler from $\mathrm{PR}(sig(G_0))$ into $\mathrm{PR}(\Sigma)$; that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G_0))) \ ([\![c^{FG}(\Sigma)(G_0)(\alpha')]\!]_{A} = [\![\alpha']\!]_{A_\bullet})$$

This is condition (215) of course.

Since $depth(S)=d+1$, condition (217) holds in this case for exactly the same reasons that it did in Case (a) above.

Case (c). Suppose $depth(G_0)=d+1$ and $depth(S)\leq d$. Then since $depth(G_0)=d+1$ it follows that condition (215) holds for exactly the same reasons that it held in Case (a) above.

Finally, it is easy to see that condition (217) holds in this case by the (main) induction hypothesis (203) applied to S.

This completes the sub-induction and also the main induction for the correctness of $c^{FP}(\Sigma)$.

Performance Preservation. To show that $c^{FG}(\Sigma)(G)$ is performance preserving, as in the case of multiple function groups with depth 0, it is sufficient for us to show

$$(\forall \alpha' \in \mathrm{PR}(sig(G_0))) \ (\lambda_P^{PR}(c^{FG}(\Sigma)(G_0)(\alpha')) \approx \lambda_{P_\bullet}^{PR}(\alpha')) \qquad (220)$$

and

$$(\forall (S,In,Out) \in \mathrm{PR}(sig(G))) \ (\lambda_{P_\bullet}^{PR}(c^{AUG}(sig(G_0))(id(S),\alpha)(\alpha')) \approx \lambda_{P_\bullet}^{PR}(\alpha')) \qquad (221)$$

where $P_0=P_{G_\bullet}$.

However, it follows from Lemma 7.3.10(iii) that condition (221) will be satisfied if we can show that

$$\lambda_{P_\bullet}^{PR}(\alpha) = \lambda_{P_\bullet}^{PR}(c^{FP}(\Sigma)(sig(G_0))(S)) \approx \lambda_{P_\bullet}^{FP}(S) \qquad (222)$$

Thus it remains to show that (220) and (222) hold:

Again there are three possibilities for the depths of G_o and S:

Case (a): $depth(G_o) \leq d$ and $depth(S) = d+1$
Case (b): $depth(G_o) = d+1$ and $depth(S) = d+1$
Case (c): $depth(G_o) = d+1$ and $depth(S) \leq d$

We will now establish that (220) and (222) are satisfied in each of these cases.

Case (a). Suppose $depth(G_o) \leq d$ and $depth(S) = d+1$. Then by the main induction hypothesis (206) applied to G_o we may assume that $c^{FG}(\Sigma)(G_o)$ is a performance preserving compiler from PR($sig(G_o)$) into PR(Σ); that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ \ (\lambda_P^{PR}(c^{FG}(\Sigma)(G_o)(\alpha')) \approx \lambda_{P_\bullet}^{PR}(\alpha'))$$

This is precisely condition (220) above of course.

Also, since $S \in \mathrm{FPIT}(sig(G_o))$ with $depth(S) = d+1$ it must be that S is a nested function program of the above form. Now, since $depth(S) = d+1$, $c^{FP}(sig(G_o))(S)$ is as defined in Induction Case (1) above; that is,

$$c^{FP}(sig(G_o))(S) = c^{FG}(sig(G_o))(G') \circ c^{IO}(sig(G'))(S_o, x, y)$$

Now assume

$$(\forall \alpha' \in :\mathrm{PR}(sig(G')))\ \ (\lambda_{P_\bullet}^{PR}(c^{FG}(sig(G_o))(G')(\alpha')) \approx \lambda_{P'}^{PR}(\alpha')) \tag{223}$$

and

$$(\forall (S, In, Out) \in \mathrm{PITIO}(sig(G')))\ \ (\lambda_{P'}^{PR}(c^{IO}(sig(G'))(\alpha')) \approx \lambda_{P'}^{IO}(S, In, Out)) \tag{224}$$

where $P' = (P_o)_{G'}$ (which is a performance measure for $A' = Alg_{A\bullet}(G')$). Then by definition of α, we now have:

$$\lambda_{P_\bullet}^{PR}(\alpha) = \lambda_{P_\bullet}^{PR}(c^{FG}(sig(G_o))(G') \circ c^{IO}(sig(G'))(S_o, x, y))$$
$$\approx \lambda_{P'}^{PR}(c^{IO}(sig(G'))(S_o, x, y))$$

(by (223))

$$\approx \lambda_{P'}^{IO}(S_o, x, y)$$

(by (224))

$$= \lambda_{P_\bullet}^{FP}(S)$$

(by definition of $\lambda_{P_\bullet}^{FP}(S)$).

Thus (222) holds as required, assuming (223) and (224) hold of course:

Now, (224) is certainly true by Lemma 7.3.6(iii), and so it remains to show that (223) is satisfied. However, since $depth(S) = d+1$, it must be that $depth(G') = d$, and so by the main induction hypothesis (206) applied to G' we have

$$(\forall \alpha' \in \mathrm{PR}(sig(G')))\ \ (\lambda_{P_\bullet}^{PR}(c^{FG}(sig(G_o))(G')(\alpha')) \approx \lambda_{P'}^{PR}(\alpha'))$$

This is precisely (223) of course, and so (222) is satisfied as claimed.

Case (b). Suppose $depth(G_o)=d+1$ and $depth(S)=d+1$. Then since $|G|=l+1$ it follows that $|G_o|=l$. Thus by the sub-induction hypothesis (214) applied to G_o we may assume that $c^{FG}(\Sigma)(G_o)$ is a performance preserving compiler from PR($sig(G_o)$) into PR(Σ); that is,

$$(\forall \alpha' \in \mathrm{PR}(sig(G_o)))\ \ (\lambda_P^{PR}(c^{FG}(\Sigma)(G_o)(\alpha')) \approx \lambda_P^{PR}(\alpha'))$$

This is condition (220) of course.

Since $depth(S)=d+1$, condition (222) holds in this case for exactly the same reasons that it did in Case (a) above.

Case (c). Suppose $depth(G_o)=d+1$ and $depth(S)\leq d$. Then since $depth(G_o)=d+1$ it follows that condition (220) holds for exactly the same reasons that it held in Case (a) above.

Finally, it is easy to see that condition (222) holds in this case by the (main) induction hypothesis (204) applied to S.

This completes the sub-induction and also the main induction for the performance preservation of $c^{FG}(\Sigma)(G)$. □

Discussion. The preceding theorem quantifies over all (standard) signatures Σ, all (standard) Σ-algebras A, and all (standard) performance measures P. An immediate consequence of the theorem is that $c^{FP}(\underline{\Sigma})$ is a correct and performance preserving compiler from FPIT($\underline{\Sigma}$) into PR($\underline{\Sigma}$); correct with respect to $F_{\underline{A}}$ and $[\![\,.\,]\!]_{\underline{A}}$, and performance preserving with respect to λ_Q^{FP} and λ_Q^{PR} where Q is a performance measure for $\underline{A}$. Thus whatever we can compute with FPIT($\underline{\Sigma}$) we can define by a scheme from PR($\underline{\Sigma}$) with equal time complexity.

In the discussion at the end of Section 6.1 we hinted that PIT could be used as an alternative means of formalising synchronous algorithms; we suggested that a synchronous network N determined a Σ-algebra A and the algorithm had a PIT($\underline{\Sigma}$)-simulation S_N. Of course, decorating S_N with an appropriate function header would give a simulation $S_N' \in$ FPIT($\underline{\Sigma}$), and thus FPIT can be viewed as an alternative means of formalising synchronous algorithms. Applying $c^{FP}(\underline{\Sigma})$ to S_N' yields a PR($\underline{\Sigma}$) scheme that is a definition of the function computed by S_N', and thus if we had promoted FPIT as *the* specification language for synchronous algorithms, then the existence of $c^{FP}(\underline{\Sigma})$ would imply that PR is an equivalent specification language. Given the discussion following Theorem 7.2.1, this completes our equivalence conjecture: *PR and FPIT are equivalent specification languages.*

An important consequence of the equivalence of PR and FPIT is that any theory of PR-definability is automatically a theory of FPIT-computation and vice versa, given the following

7.4.2 Corollary.

For any standard signature Σ *and standard* Σ*-algebra* A, *define* FPIT(A) *by*

$$\mathrm{FPIT}(A) = \{ F_A(S) : S \in \mathrm{FPIT}(\Sigma) \}$$

Then FPIT(A)=PR(A), *that is, for every function* f_A,

$$f_A \in \mathrm{FPIT}(A) \iff f_A \in \mathrm{PR}(A)$$

Proof. Immediate from the existence and correctness of c^{PR} and $c^{FP}(\Sigma)$. □

With this definition of FPIT(A) (which is intuitively the collection of all FPIT(Σ)-computable functions on A), theorems about PR become theorems about FPIT. For example, recall the following fact about PR (first considered in Section 3.5):

$$\mathrm{PR}(A) = \mathrm{PR}(A, f_A) \iff f_A \in \mathrm{PR}(A)$$

Intuitively, this says that primitive recursive programming over an algebra A is the same as primitive recursive programming over A augmented with f_A exactly when f_A is PR-computable over A. An immediate consequence of Corollary 7.4.2 is:

$$\mathrm{FPIT}(A) = \mathrm{FPIT}(A, f_A) \iff f_A \in \mathrm{FPIT}(A)$$

Another fact from Section 3.5 concerning PR that carries over to FPIT is the isomorphism invariance of PR-computability: it is trivial to prove that FPIT-computability is an isomorphism invariant. from Lemma 3.5.9 and the equivalence of PR and FPIT.

Here we see that theoretical observations about PR automatically lead to *dual* observations about FPIT; by symmetry, any theoretical statement about FPIT has a dual concerning PR. For example, in the next chapter we will discuss the hierarchical implementation of synchronous algorithms; we use PR as the principle mathematical tool (since PR is the official specification language for synchronous algorithms), however, since PR is equivalent to FPIT, the chapter could equally well use FPIT.

7.5 SOURCES.

The computational equivalence of independent specification languages for synchronous algorithms has been the joint project of J. V. Tucker and myself; all the proofs in this chapter and the strategies behind them are my own work however.

The equivalence of PR and FPIT generalises the fact that the primitive recursive functions on the natural numbers are precisely the functions computable by (bounded-) loop programs over the natural numbers; this was first proved in Meyer and Ritchie[1967]. This result belongs to the long tradition of equivalence between different models of computation starting with the equivalence of the λ-calculus and Turing Machines proved by Turing.

The normal proofs of computational equivalence of formalisms pay insufficient attention to (i) the matter of when a program can be said to 'compute a function', (ii) the precise nature of the transformation between formalisms, and (iii) performance aspects. Furthermore, in the context of (many-sorted) abstract structures the situation is complicated by the need for *simultaneous* primitive (or course-of-values) recursion since this requires that a parallel feature be added to loop programs.

These points are made in Chapter 4 of Tucker and Zucker[1987], and from the point of view of the synthesis of programming language semantics and the theory of computability as described therein, the equivalence of PR and FPIT as proven here is the 'correct' and definitive proof that is required.

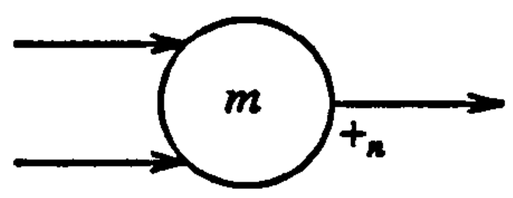
Moreover, to provide rigorous scientific foundations for the practical use of PR and FPIT, Theorems 7.2.1 and 7.4.1 are just what is required, containing as they do, compilers and proofs of their correctness and performance preservation.

Finally, the definition of a correct compiler given here is based on the definitions given in Morris[1973] and Thatcher, Wagner, and Wright[1980]; for more information see the volume Jones[1980]. The idea of a performance preserving compiler appears to be new.

CHAPTER 8
TOP-DOWN DESIGN

We have seen how PR is a useful tool for specifying, verifying, and simulating synchronous algorithms. In this chapter we will use PR to study *top-down design.*

We begin in Section 8.1 by considering what it means to 'substitute' one synchronous network for a module in another synchronous network; network-for-module substitution is the principal mechanism employed in implementing a given synchronous network at a desired level of data abstraction. For example, suppose N is a synchronous network over an algebra A involving the algebra $\mathbf{Z}_n = \{0,1,\ldots,2^n-1\}$ of *integers modulo* 2^n. Suppose that one of N's modules m is specified by addition $+_n : \mathbf{Z}_n \times \mathbf{Z}_n \longrightarrow \mathbf{Z}_n$ on $\mathbf{Z}_n$ defined by $z_1 +_n z_2 = (z_1 + z_2) \; mod \; 2^n$ for each $z_1, z_2 \in \mathbf{Z}_n$. Since addition is a binary single-valued operation, and since an element of $\mathbf{Z}_n$ is considered as atomic data in A, in the picture of N the module m will have two input channels and one output channel:



Now, there is a familiar sense in which $\mathbf{Z}_n$ can be represented or *coded* over $\mathbf{Z}_1 = \{0,1\}$: with each $z \in \mathbf{Z}_n$ we associate the vector $b = (b_1,\ldots,b_n) \in \mathbf{Z}_1^n$ where $b_1 \cdots b_n$ is the binary representation of z. Conventionally such a vector b is called an *n-bit word.* We can now consider implementing N over $\mathbf{Z}_1$ in the following way. With respect to $\mathbf{Z}_1$ an element $z \in \mathbf{Z}_n$ is a n-tuple of elements of $\mathbf{Z}_1$. Since channels can only carry a single datum in our model this means that wherever N has a channel to carry an element $z \in \mathbf{Z}_n$ we need n channels to carry the n bits of z's binary representation, and wherever N has a source that supplies elements of $\mathbf{Z}_n$, we need n sources each suppling elements of $\mathbf{Z}_1$. Since addition on $\mathbf{Z}_n$ is no longer available as a basic operation, it must be implemented by a network N_+ over $\mathbf{Z}_1$ with $2n$ input channels and n output channels as illustrated in Figure 8.1; the network inputs and outputs will be elements of $\mathbf{Z}_1$ of course.

A key idea is that we can substitute this network for the module depicted above wherever it occurs in (the picture of) N; if we can implement all the modules of N by networks over $\mathbf{Z}_1$, then by substituting these networks for the relevant modules the network N will be transformed into a new network N' over $\mathbf{Z}_1$.

More generally, given a synchronous network $N = N_1$ over (high level) algebra $A = A_1$, if we can implement each module of N_1 by a synchronous network over (lower level) algebra A_2, then we can substitute these networks for the modules of N_1 to obtain an implementation N_2 of N_1 over A_2. If the modules of N_2 can themselves be implemented by further networks over (even lower level) algebra A_3 then the substitution of these networks for the modules of N_2 gives an implementation N_3 of N_2 (and

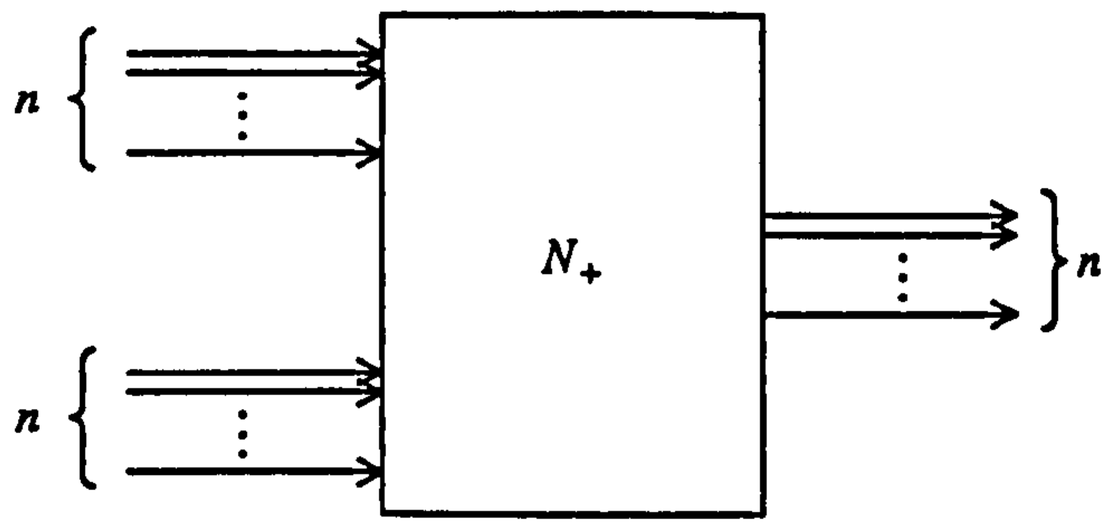


Figure 8.1 - The network N_+.

hence N_1) over A_3. Our strategy for implementing $N = N_1$ over a given algebra B is therefore to repeatedly apply top-down design generating a sequence $N_1, N_2, N_3, \ldots$ with N_1 over A_1, N_2 over A_2, etc. until we reach some nth stage where $A_n = B$, for then N_n is the required implementation of N over B.

Of course, it is central to the concept of 'top-down' design that in the step from N_i to N_{i+1} ($i = 1, \ldots, n-1$), the correctness of N_{i+1} should follow immediately from the correctness of N_i and the correctness of the implementations of N_i's modules; this obviates the need for formal verification at low levels of data abstraction where the design may be too complex to verify directly.

We will formalise the top-down design of synchronous networks using PR. If Σ_i is the signature of A_i ($i = 1, \ldots, n$) then N_i is specifiable in $\mathrm{PR}(\underline{\Sigma}_i)$ and N_{i+1} in $\mathrm{PR}(\underline{\Sigma}_{i+1})$, and so the transformation from N_i to N_{i+1} can be formalised as an instance of compilation via a mapping $c_i : \mathrm{PR}(\underline{\Sigma}_i) \longrightarrow \mathrm{PR}(\underline{\Sigma}_{i+1})$. In this theoretical framework we begin with the PR specification of $N = N_1$, $\alpha_1 = \alpha_N \in \mathrm{PR}(\underline{\Sigma}_1)$, from which we obtain the PR specification of N_2 namely $\alpha_2 = c_1(\alpha_1) \in \mathrm{PR}(\underline{\Sigma}_2)$, and then the PR specification of N_3 namely $\alpha_3 = c_2(\alpha_2) \in \mathrm{PR}(\underline{\Sigma}_3)$, and so on, until we reach $\alpha_n = c_{n-1}(\alpha_{n-1}) \in \mathrm{PR}(\underline{\Sigma}_n)$, the PR specification of the required implementation over A_n. Note that it is the correctness of the compilers $c_1, \ldots, c_{n-1}$ that formalises the idea that the step from one design to the next is 'correctness preserving'. Moreover, in this thesis we have insisted that the formal definition of a data type is incomplete without an account of the complexity of data and operations. Thus each algebra A_i in the *hierarchy* $A_1, \ldots, A_n$ comes equipped with its own performance measure P_i by default; in this way we can account for the complexity of N when implemented at lower levels. However, the tediousness and difficulty of calculating algorithm performance increases with algorithm complexity and it will be expedient therefore if $c_1, \ldots, c_{n-1}$ have regard for algorithm performance: we want to be able to *predict* the complexity of α_n without having to calculate it directly.

Actually, in this chapter we will work with a generalisation of synchronous networks that we call *network systems*. The reason for this is that in the transformation from one network to another by network-for-module substitution, it is possible that different modules of the first network are implemented over quite different algebras with *different clocks*, and so network-for-module substitution does not give another synchronous network but a 'network-of-synchronous-networks'; a network system that

is to say.

In Section 8.1 we introduce network systems. We will show that whilst network systems are not strictly synchronous algorithms in the sense of previous chapters, they are still specifiable in PR and so our previous work on synchronous algorithms carries over to network systems; for example, a network system over a Σ-algebra A is specifiable in PR($\underline{\Sigma}$) and so can be simulated in FPIT($\underline{\Sigma}$).

In Sections 8.2 - 8.4 we develop the mathematical tools for analysing top-down design. In general, when transforming or compiling an algorithm over one algebra A_1 to an algorithm over another algebra A_2, A_1 and A_2 need not have the same carrier sets and so it is necessary to formally define what it means for the data sets of one algebra to be 'represented' over those of another; the central technical concept here is a *coding* (a generalisation of a Gödel numbering). Also in general, A_1 and A_2 will have different performance measures P_1 and P_2 based on different clocks C_1 and C_2 respectively. Thus to relate the performance of an algorithm over A_1 to that of another algorithm over A_2, it is necessary to consider relationships between the underlying clocks, *retimings* that is to say. Codings and retimings are the subject matter of Section 8.2. In Section 8.3 we consider what it means for a PR scheme over one algebra to 'implement' a function defined over another. In particular we consider how to implement the constants and operations of one algebra by PR schema over another, and we define the complexity of these implementations. The ideas of coding, retiming, and implementation of operations, sum up to a definition of what it means for A_1 to be 'A_2-computable' (including performance matters); these ideas are all key components in the general framework for the analysis of top-down design discussed above. In Section 8.4 we complete the general framework by proving a very general result (Theorem 8.4.1) which states that under a certain but natural hypothesis on the algebras involved, we only need to provide implementations of the operations of the high level algebra by schema over the lower level algebra to *automatically generate* a correct compiler from the high level to the lower level. (The construction of these compilers is 'semantics directed' in the sense of the volume Jones[1980].) Furthermore, these compilers are performance preserving in the sense that they preserve the complexity of the implementation (of the operations of the high level algebra): 'good' or efficient implementations lead to efficient compilers, but 'bad' or inefficient implementations lead to inefficient compilers.

The material of these three technical sections is developed independently of the concept of a synchronous algorithm since it is of interest in its own right for the following two reasons. First, whilst our work explicitly concerns PR, our ideas can be viewed as a theory of top-down design for any functional language (in the sense of Section 7.1): in particular, given the equivalence of FPIT to PR, this chapter could be exclusively concerned with an FPIT- (rather than PR-) representation of synchronous algorithms. Second, our mathematics can be seen as the starting-point for a generalisation of the Mal'cev-Rabin theory of *computable algebra* (see Mal'cev[1961] and Rabin[1960]).

Finally, in Section 8.5, we show that our theory of top-down design is directly applicable to synchronous algorithms by considering the hierarchical decomposition of the FIR network of Section 5.1 in detail.

PR Systems. From the end of the next section onwards, we will be exclusively concerned with implementing the constants and operations of one algebra by PR schema over another. In order to keep track of the sort sets, signatures, algebras, performance measures and clocks involved, we make the following

Definition. Let S be a (standard) sort set, let Σ be a (standard) signature, and let A be a (standard) Σ-algebra. Also let P be a (standard) performance measure for A which is based on clock C. Then we say L is a *PR system* where L is the tuple

$$L = (S, \Sigma, A, C, P, \mathrm{PR}(\Sigma), [\![.]\!]_A, \lambda_P^{PR})$$

(Here '$[\![.]\!]_A$' and 'λ_P^{PR}' are the usual semantic evaluation and performance estimation functions for PR(Σ) of course.)

We will use a PR system in the following way: we will say 'let L be a PR system' without explicitly writing L as a tuple; the idea is that any PR system with name 'L' is a tuple with sort set S, S-sorted signature Σ, Σ-algebra A, etc. In the case of a subscripted PR system as for example when we say 'let L_i be a PR system for $i = 1,\ldots,n$', we understand L_i to be the tuple

$$L_i = (S_i, \Sigma_i, A_i, C_i, P_i, \mathrm{PR}(\Sigma_i), [\![.]\!]_{A_i}, \lambda_{P_i}^{PR})$$

for some sort set S_i, S_i-sorted signature Σ_i, Σ_i-algebra A_i etc.

We use the letter 'L' for PR systems since a PR system is a *level of computational abstraction* in the sense of Thompson and Tucker[1985]. Subsequently we will use 'L_i' as an adjective which we read as 'level i'. For example, if L_1 and L_2 are PR systems, then we refer to a collection of schemes over Σ_2 that implement the constants and operations of A_1 as 'an L_2-implementation of A_1', read: 'a level 2 implementation of A_1'. Similarly, we will encounter an '(L_1,L_n)-compiler' which is a compiler which maps 'level one' schema to 'level n' schema; that is, a mapping $c : \mathrm{PR}(\Sigma_1) \longrightarrow \mathrm{PR}(\Sigma_n)$.

Also, since we will be exclusively concerned with PR in this chapter, we no longer need the superscript 'PR' on our complexity function λ_P^{PR} which will be subsequently denoted simply λ_P.

8.1 NETWORK SYSTEMS.

In this section we will develop a new model of synchronous computation that we call *network systems*: a network system is essentially a synchronous network whose modules have been replaced by other synchronous networks.

In Section 8.1.1 we explain how network systems are made from synchronous networks, and we informally explain the intended semantics of network systems. In Section 8.1.2 we formalise the semantics of a network system by writing down (generalised) value functions for a network system. Throughout Sections 8.1.1 - 8.1.2 we assume that all the algebras involved (are many-sorted but) have the same underlying family of carriers: the case where the data sets are not the same is explained in Section 8.1.3.

8.1.1 Network Substitution.

Let us begin with some pictures. Consider Figures 8.2 - 8.6. Figure 8.2 illustrates a 3-module synchronous network N. In Figures 8.3, 8.4, and 8.5 we see networks N_1, N_2, and N_3 respectively, and the result of substituting N_1, N_2, and N_3 for m_1, m_2 and m_3 of N respectively is the network system M of Figure 8.6.

Notice that in Figure 8.6 the 'bounding boxes' of the subnetworks (shown with a dotted rather than solid line for legibility) have been left in the picture of M; as we will explain below, the boundaries are intuitively significant in defining the semantics of a network system and cannot simply be 'rubbed out'. Two other new features in the figures are modules shown with a double circle, and sources and channels drawn with a dashed line. The double circles are just a helpful identification of those modules that supply sinks; the significance of such modules will be explained later. The features shown with a dashed line are to be ignored for the time being; these features concern nonautonomous modules and will be described later also.

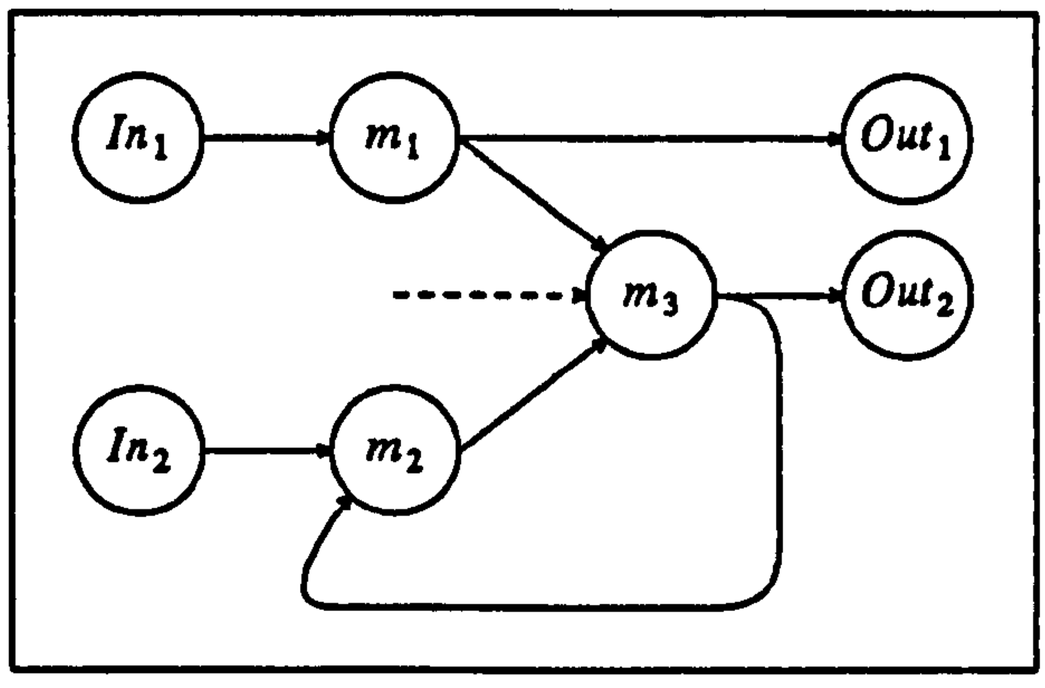


Figure 8.2 - N.

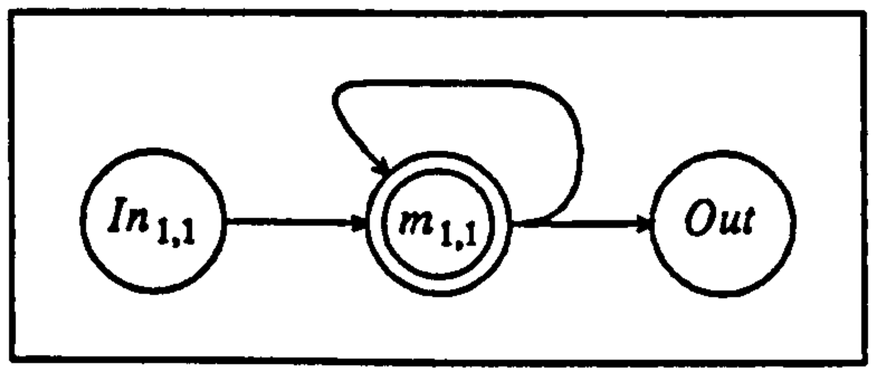


Figure 8.3 - N_1.

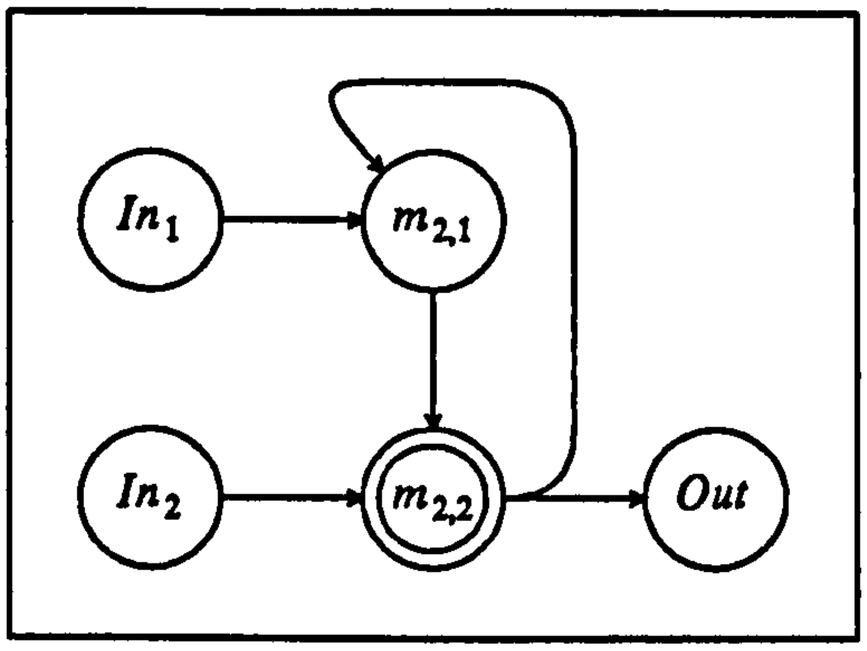


Figure 8.4 - N_2.

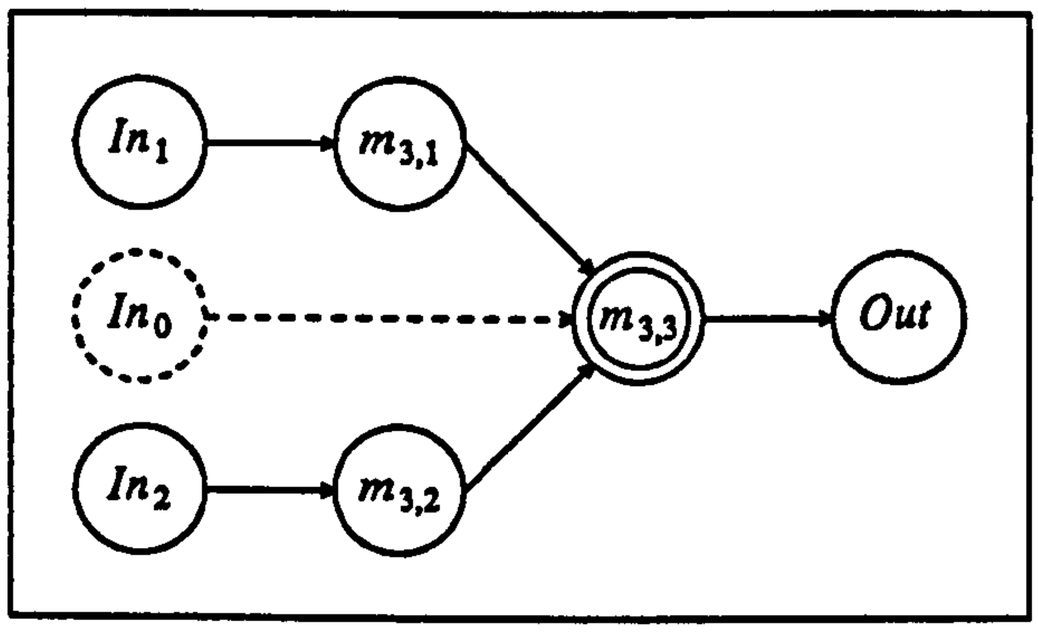


Figure 8.5 - N_3.

More generally, we begin with a synchronous network N defined over an algebra A with clock T; N has modules $m_1,...,m_k$ say, with m_i specified by σ_i^A for $i = 1,...,k$. For the time being we will assume *all* of N's modules are autonomous with respect to T; we will extend the model to include nonautonomous modules after we have introduced the basic ideas. Now let N_i be the synchronous network which replaces module m_i in a network system M for $i = 1,...,k$. Most generally, each N_i is defined over a *different* algebra, B_i say; each B_i will involve a *local* clock T_i as a carrier. The fact that each B_i has its own clock is intended to reflect the idea that $N_1,...,N_k$ will in general compute at different speeds; this is why we do not (cannot) 'rub out' the boundaries of subnetworks in the picture of a network system. For the time being we assume that the carriers of B_i are the same as those of A except for A's carrier T which is replaced by T_i in B_i. (Actually, when m_i is nonautonomous T must appear in B_i, but as another data set, *not* as B_i's clock which is T_i.)

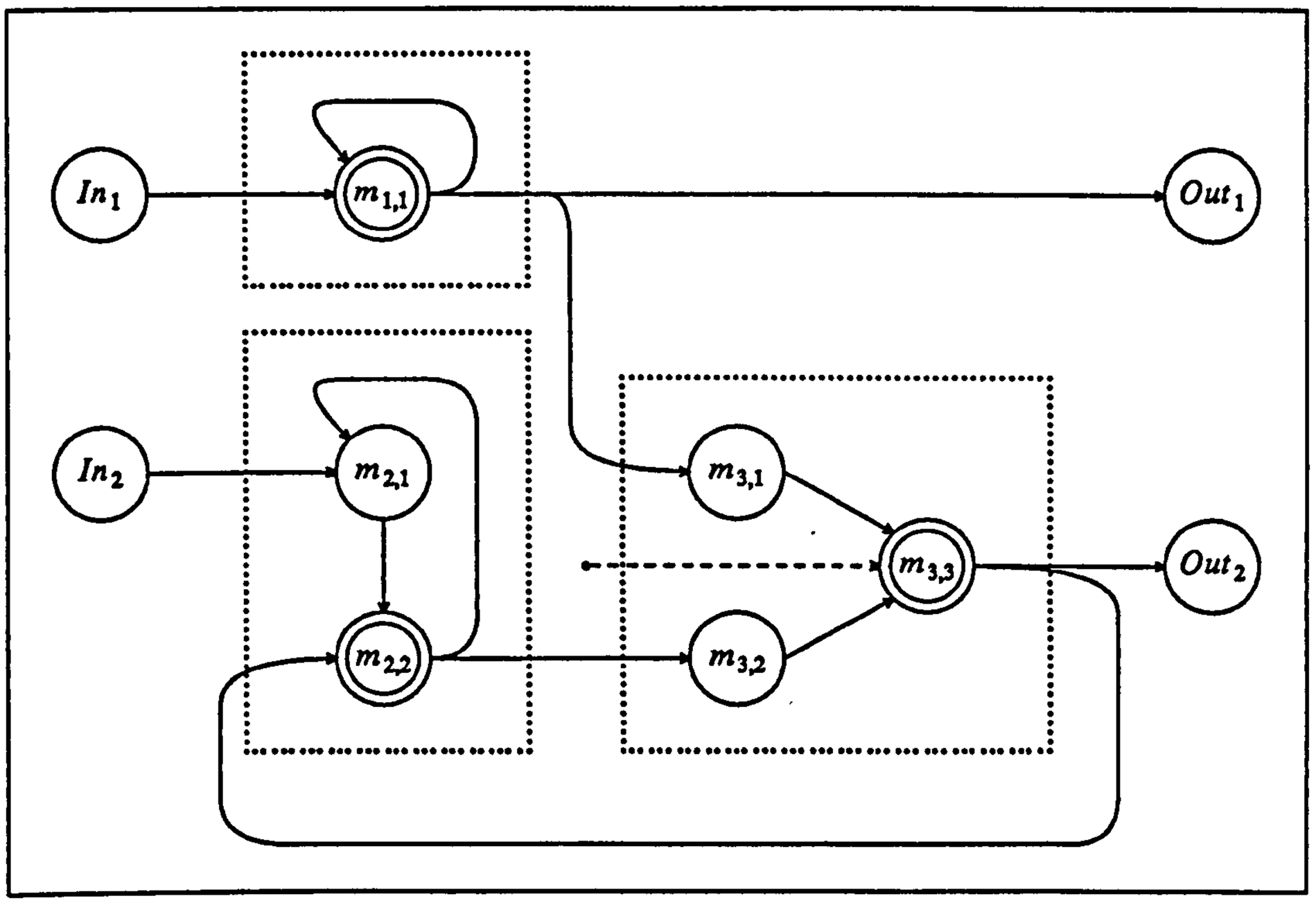


Figure 8.6 - M.

Eventually we will want to discuss the correctness of the transformation induced by network substitution, so it is necessary for us to first say what constitutes a 'legal' substitution.

Consider an (autonomous) module m_i (for $i \in [1,k]$). Since m_i will have some $n_i \geq 1$ inputs (and one output) it makes sense to require N_i has n_i sources (and a single sink). Also, the substitution of N_i for m_i should be 'sort-preserving': if $\sigma_i^A : A^{v_i} \longrightarrow A_{s_i}$ for some word $v_i \in S^+$ and some sort $s_i \in S$, then ($n_i = |v_i|$ and) the sources of N_i should be such that they supply vectors from A^{v_i}, and N_i's sink should receive data from A_{s_i}. Here we introduce some terminology: let us call the (unique) module of N_i that supplies N_i's sink with output data N_i's *exit module*; then we require that N_i's exit module to hold data from A_{s_i}. (Alternatively, the operation of B_i that specifies N_i's exit module must have codomain A_{s_i}.) Modules that are not exit modules we call *interior modules*. We will also refer to 'the exit modules of a network system' meaning the exit modules of the network system's sub-networks. Exit modules are shown with a double circle in our figures.

As further notation, suppose N_i has some $l_i \geq 1$ modules denoted $m_{i,j}$ ($j = 1,\ldots,l_i$, $i = 1,\ldots,k$); we denote the sort of data held by $m_{i,j}$ by $s_{i,j} \in S$. We also define w_i by $w_i = s_{i,1} \cdots s_{i,l_i}$ for $i = 1,\ldots,k$. For future use, note that under these hypotheses on $N_1,\ldots,N_k$, each N_i will have a *static* output specification f_{N_i} of functionality:

$$f_{N_i} : T_i \times A^{v_i} \times A^{w_i} \longrightarrow A_{s_i}$$

for $i = 1,\ldots,k$. (Static output specifications and their formal status are discussed in Section 2.4.4; the reason why we are interested in a static rather than dynamic specification of N_i will become clear later.)

Finally, for $i = 1,\ldots,k$ let ζ_i denote some given vector of constants from B_i, $\zeta_i = (c_{i,1},\ldots,c_{i,l_i}) \in A^{w_i}$.

A network system M is obtained from a k-module network N and sub-networks $N_1,\ldots,N_k$ by substituting N_i for m_i in such a way that the communication topology of N is preserved: if m_i has its jth input from source In_p (in N) then so does N_i (in M), and if the jth input to m_i comes from a module m_q, then the jth input to N_i is from the exit module of N_q. (Cf. Figures 8.2 - 8.6.)

Operational Behaviour. The essence of our interpretation of the behaviour of M is that like a synchronous network, M proceeds in 'steps'. These steps (and hence the behaviour of M) are defined in terms of some given functions $\lambda_1,\ldots,\lambda_k$ and the given vectors $\zeta_1,\ldots,\zeta_k$. The purpose of $\lambda_i : A^{v_i} \longrightarrow T_i$ $(i = 1,\ldots,k)$ will be apparent shortly. Each ζ_i is used as the initial values on which N_i is *always* to be executed.

We imagine M to begin in an initial state wherein the modules of M are holding certain initial values and the sources are supplying some input data; M is now ready to perform its first step. Generally, the nth ($n \geq 1$) step proceeds as follows. Each module is about to read data from a neighbouring module (possibly the exit module of a neighbouring network) and/or from a source as indicated by the channels in the picture of M; thus the input to each sub-network N_i at the start of the nth step is a vector $a = a_{i,n} \in A^{v_i}$. Each N_i is now executed in parallel on the input $a_{i,n}$ and initial values ζ_i for $\lambda_i(a_{i,n})$ cycles of T_i. We imagine each N_i to be executed in isolation from all the other sub-networks: during sub-network's execution it reads no new input data and we imagine the input that was available to the sub-network at the beginning of the nth step ($a_{i,n}$) to be held *fixed* for the duration of the sub-network's execution. After time $\tau = \max_{1 \leq i \leq k} \{ \lambda_i(a_{i,n}) \}$ has elapsed, the nth step is deemed to have ended and the next step begins; it is at this time that we imagine the next inputs to M to be available at the sources. (Sub-networks that finish before time τ simply wait for the others to catch up.)

(Notice that this description of the synchronous behaviour of a network system is very similar to the original description of what was 'synchronous' about a synchronous network given in Section 2.1: network systems are more general than synchronous networks in the sense that in Section 2.1 τ was assumed to be the maximum execution time over all modules and all inputs to the modules, whereas in this chapter τ does not need to be an upper bound.)

There are three key components in the above informal description of M's behaviour: the functions $\lambda_1,\ldots,\lambda_k$; the assumption that sub-networks are always executed on the same initial values $(\zeta_1,\ldots,\zeta_k)$, and the assumption that input data to a sub-network is held fixed for the duration of the sub-network's execution. We will comment on each idea in turn.

The role played by $\lambda_1,\ldots,\lambda_k$ should be clear: although we understand $N_1,\ldots,N_k$ to operate *asynchronously* with respect to each other during each step of M, the functions $\lambda_1,\ldots,\lambda_k$ allow us to give M a 'synchronous semantics': the steps performed by M are defined or determined by $\lambda_1,\ldots,\lambda_k$; they define a

new clock T' wherein each $t' \in T'$ represents the beginning of the nth step when $t' = n$. The roles played by the assumptions that sub-networks are executed on the same initial values and that input data is held fixed are more subtle: these assumptions make the semantics of a network system *modular*.

In the notation used above, let us say that N_i *implements* σ_i^A (or m_i) (*with respect to* λ_i *and* ζ_i) if

$$f_{N_i}(\lambda_i(a), a, \zeta_i) = \sigma_i^A(a)$$

for every $a \in A^{*_i}$. Then, essentially, N_i implements m_i if it computes $\sigma_i^A(a)$ in finite time when the input a is held fixed. The crucial aspect of this definition is that if N_i implements σ_i^A outside M, that is, when executed on its own, then it will implement σ_i^A when executed as a part of M: in general, that the behaviour of a component of a system is the same both inside and outside the system is central to top-down design of course. By fixing the input to a network we turn the network from a function on streams of data into a function on data (this is appropriate since the modules of N are specified by functions on data, not streams of data). Moreover, and this is essentially the reason for not 'rubbing out' subnetwork boundaries in network systems, without the constant initial values, each execution of a sub-network in M would begin on the initial values that were left from the previous step. However, the concept of a 'step of M' is intuitively undefined when we consider a sub-network in isolation from M; this makes it difficult to say how the behaviour of the sub-networks should be 'the same' inside and outside M.

Our definition of 'N_i implements m_i' is the 'right' definition in the sense that with this definition, network-for-module substitution is behaviour- and hence correctness-preserving as we will now explain. (Throughout the following general discussion of a synchronous network N and a network system M, the reader should refer to Figures 8.2 and 8.6.)

Consider executing N and M together in a stepwise manner. Suppose m_i and the exit module of N_i hold the same initial data for $i = 1,...,k$. Suppose also that the sources of N and M are supplying the same input data. Assuming that N_i implements m_i (with respect to λ_i and ζ_i) for $i = 1,...,k$, it should be clear that when N and M have completed their respective first steps, each module m_i will hold the same value as that held by the exit module of N_i: if the input to m_i is some $a \in A^{*_i}$ then m_i will hold $\sigma_i^A(a)$ at the end of N's first step; but by hypothesis the input to N_i is also a, and so after $\lambda_i(a)$ steps of T_i, and so at the end of M's first step, the exit module of N_i will hold $f_{N_i}(\lambda_i(a), a, \zeta_i) = \sigma_i^A(a)$. Intuitively, if the sources continue to supply the same data to N and M, the values held by the exit modules of M at the end of the n th step will be the same as those held by the modules of N at the end of its n th step.

An interesting point that arises here is the relationship between the clock of N, namely T, and that of M, namely T'. Intuitively, T and T' are different clocks since a cycle of T is conceptually indivisible, whereas a cycle of T' is decomposed into sub-cycles of lower-level clocks. However, in the above discussion of the simultaneous 'stepwise' execution of N and M, it is clear that if N computes an output value after 5 cycles (say), that is, if an output is available when the time according to T is $t = 5$, then M will also produce (the same) output value after 5 cycles, that is, when the time according to T' is $t' = 5$. More generally, when the time according to T is $t = n$, the time according to T' will be $t' = n$, and so to compare the behaviour of N and M (in particular, for the purpose of studying correctness) we must use

the fact that T and T' are *isomorphic.*

The relationship of isomorphism is an identity relationship, and so we must think of T and T' as being the same. Whilst this may seem a little surprising, it is in fact quite appropriate if we think of N as being *abstracted* from M, instead of M as being made from N. Recall how the clock T of a synchronous network N was defined originally back in Chapter 2. We said that T was defined by normalising to unity the maximum time over all modules taken to compute an output on any given input. Now given a network system M that implements N, the time taken for a module to compute on a given input is intuitively the time taken for its (subnetwork-) implementation to compute on that input. In other words, when we regard N as being an abstraction of M, T is actually *defined* to be T'.

These remarks notwithstanding, we will often write 'T'' for a network system's clock, and 't'' for elements of T'. It is helpful to do this since it is a convenient way of remembering in algebraic calculations whether we are considering a synchronous network or a network system. For example, a stream input to a synchronous network will continue to be written '$\underline{a} : T \longrightarrow A$', whereas for a stream input to a network system we will write '$\underline{a}' : T' \longrightarrow A$'. (A minor problem with this is that occasionally we will encounter mixed expressions such as '$\underline{a}'(t)$' or '$\underline{a}(t')$', but this is of no significance given the fact that $T \cong_\phi T'$ for some isomorphism ϕ.)

8.1.2 Formalisation.

We will now formalise the informal description of a general network system's behaviour given above. Doing so will allow us to prove the correctness of network-for-module substitution. (In the text that follows we continue with the notation used above.)

Suppose the original network N has a dynamic specification V_N of functionality

$$V_N : T \times [T \longrightarrow A^u] \times A^w \longrightarrow A^w$$

for some $u, w \in S^+$. In fact, since the codomain of σ_i^A is A_{s_i} for $i = 1,\ldots,k$, it must be that $w = s_1 \cdots s_k$. Here we call $[T \longrightarrow A^u]$ and A^w N's *input space* and N's *state space* respectively.

The steps performed by M determine the clock T', and new data is read in with each cycle of T'. Since N's input space is $[T \longrightarrow A^u]$, and since M has the same sources as N, it makes sense for us to regard $[T' \longrightarrow A^u]$ as M's input space. What should M's state space be?

The set A^w is N's state space in the sense that a vector $a = (a_1,\ldots,a_k) \in A^w$ is sufficient to describe the values held by the modules of N at each time $t \in T$ (with the intention that module m_i holds a_i of course). This might suggest that since each subnetwork N_i of M has l_i modules, a vector of length $l = l_1 + \cdots + l_k$ is what we want to describe the 'state' of M at each time $t' \in T'$ (since M has l modules). However, we will argue against this choice; we will argue that it should be A^w (the same as N's):

We certainly want to specify the values held by M's exit modules since these form the input to neighbouring networks for the next step and so partly determine what happens on the next step. Also, since the sinks of M are supplied with output data by exit modules, these values ultimately tell us M's output at each time t'. What about the values held by interior modules? We will not specify these

because the values held by the interior modules of each N_i at a given time t' are of no interest: they are overwritten by ζ_i the next time N_i is executed and so they do not affect the behaviour of M. Thus we only want to specify the values held by M's exit modules at each time t'. However, at each time t' the exit modules of $N_1,...,N_k$ collectively hold a vector $a=(a_1,...,a_k)\in A^w$ (with the intention that a_i is the value held by N_i's exit module for $i=1,...,k$), and so we take A^w as M's state space.

Returning to the formal specification of our network system M, similar to the way that N was dynamically specified by the function

$$V_N=(V_1,...,V_k):T\times[T\longrightarrow A^u]\times A^w\longrightarrow A^w$$

we will now give a formal dynamic specification U_M for M; this U_M is of functionality

$$U_M=(U_1,...,U_k):T'\times[T'\longrightarrow A^u]\times A^w\longrightarrow A^w$$

with the intention that for each $t'\in T'$, $\underline{a}':T'\longrightarrow A^u$, and $x\in A^w$, the expression '$U_i(t',\underline{a}',x)$' is read as 'the value held by the ith exit module at time t' when M is executed on input stream $\underline{a}'$ and initial values x': when $x=(x_1,...,x_k)\in A^w$, x_i is intended to be the value held by the exit module of N_i at time $t'=0$. (Note that we only need to specify the initial values of exit modules since the initial values of interior modules are overwritten with $\zeta_1,...,\zeta_k$.)

We assume for the time being that all of N's modules are autonomous since the case of nonautonomous modules complicates the basic ideas.

Like the coordinates $V_1,...,V_k$ of V_N, it is natural to define the coordinates $U_1,...,U_k$ of U_M by means of equations. For $i=1,...,k$, define U_i at $t'=0$ by

$$U_i(0,\underline{a}',x)=x_i \tag{1}$$

for each $\underline{a}':T'\longrightarrow A^u$ and $x=(x_1,...,x_k)\in A^w$. This equation is appropriate given our interpretation of the vector x.

To define U_i $(i=1,...,k)$ at successive times we use the following equation:

$$U_i(t'+1,\underline{a}',x)=f_{N_i}(\lambda_i(b),b,\zeta_i) \tag{2}$$

where $b=(b_1,...,b_{n_i})$ is defined by

$$b_j=\begin{cases}\underline{a}'_p(t') & \text{if the } j\text{th input to } m_i \text{ is from source } In_p\\ U'_q(t',\underline{a}',x) & \text{if the } j\text{th input to } m_i \text{ is from module } m_q\end{cases}$$

for $j=1,...,n_i$. (Here we could have said 'if the jth input to N_i is from the exit module of N_q' instead of 'if the jth input to m_i is from module m_q', but the latter is shorter.)

Discussion. Four points arise here. First, the equations (1) and (2) (when written out for $i=1,...,k$) formalise the idea that M operates in steps which can have a data-dependent duration. Note that the equations do not tell us *how* the sub-networks are *to be* synchronised, only that they *are* synchronised.

Second, note that the defining equations for $U_1,...,U_k$ are almost identical to those for the coordinates $V_1,...,V_k$ of V_N. In essence, all that has changed is that in (2) (for U_i) the expression '$f_{N_i}(\lambda_i(b),b,\zeta_i)$' replaces what would be (for V_i) '$\sigma_i^A(b)$' where σ_i^A is the function that specifies m_i. Now, when N_i implements m_i we have $f_{N_i}(\lambda_i(a),a,\zeta_i)=\sigma_i^A(a)$ for any $a\in A^{v_i}$, and so it is easy to see

that for each $t \in T$, $\underline{a} : T \longrightarrow A^{u}$ and $x \in A^{w}$, $V_N(t,\underline{a},x) = U_M(t,\underline{a},x)$ can be proved by a routine induction on t; this we leave as an exercise. Network-for-module substitution is therefore correctness preserving when the networks implement the modules they replace.

Third, note that whilst we have introduced the idea of a network implementing a module to help explain the semantics of a network system, the definition of U_M is independent of this idea: U_M is defined for the substitution of *any* sub-networks $N_1,\ldots,N_k$ (subject to sort-theoretic constraints of course).

Finally, we note that we can substitute $N_1,\ldots,N_k$ into any network over A. That is, given another network N' which involves (perhaps a subset of) $m_1,\ldots,m_k$, but in a different configuration, the substitution of $N_1,\ldots,N_k$ for the modules of N' is still well-defined, as is the definition of $U_{M'}$, the formal specification of the result of the substitution M'.

Nonautonomous Modules. We now extend our concept of a network system to include the case that the original network involves nonautonomous modules.

Continuing with the previous notation, suppose a module m_i ($i \in [1,k]$) is nonautonomous with respect to T; then $\sigma_i^A : T \times A^{v_i} \longrightarrow A_{s_i}$ for some $v_i \in S^+$ and $s_i \in S$. Module m_3 of N of Figure 8.2 is intended to be nonautonomous; the dashed channel is intended to supply the current time $t \in T$ to the module. Whilst we have not included an explicit channel to supply the time to nonautonomous modules in figures previous to this chapter, doing so now will make the (pictorial) correspondence between N and M much clearer.

We extend the previous definition of what it means for N_i to implement module m_i (with respect to a mapping λ_i and a vector ζ_i of constants) as follows: first suppose $\lambda_i : T \times A^{v_i} \longrightarrow T_i$ and that f_{N_i} has functionality

$$f_{N_i} : T_i \times (T \times A^{v_i}) \times A^{w_i} \longrightarrow A_{s_i}$$

for some word w_i. (Here the parentheses in the domain of f_{N_i} are purely for emphasis.) We say that N_i implements $\sigma_i^A : T \times A^{v_i} \longrightarrow A_{s_i}$ with respect to λ_i and $\zeta_i \in A^{w_i}$ if

$$f_{N_i}(\lambda_i(t,a),t,a,\zeta_i) = \sigma_i(t,a) \tag{3}$$

for every $t \in T$ and $a \in A^{v_i}$. □

Two points arise here. First note that T appears in the domain of f_{N_i} as a *data set* and not the clock with respect to which we define the behaviour of N_i which is T_i. The second (related) point is that for T to appear where it does in the domain of f_{N_i}, that is, for f_{N_i} to have the given functionality, in addition to $n_i = |v_i|$ sources to supply 'a' to N_i in (3), N_i needs an additional source to supply data from the set T. The 'dashed' source in Figure 8.5 corresponds to m_3 in N of Figure 8.2 being nonautonomous.

The appropriate equation to define U_i at time $t'+1$ in the case of m_i being nonautonomous is:

$$U_i(t'+1,\underline{a}',x) = f_{N_i}(\lambda_i(t',b),t',b,\zeta_i) \tag{4}$$

where $b = (b_1,\ldots,b_{n_i})$ is as in the case of an autonomous module.

Discussion. The function U_M formalises our intuitive idea of a network system in the same way that V_N formalises the concept of a synchronous network. Also, recall that V_N was itself formalised by the PR scheme α_N (see Theorem 3.4.3); this α_N is the starting point for formal analysis of N, analysis of correctness and simulation for example. A question that arises here is: what is the formal status of 'U_M'? In particular: is there a PR scheme which formalises it? The answer to this last question is 'yes'; there is a scheme α_M which, when interpreted in the appropriate algebra, is U_M. We will now explain why.

Consider the function $I_i : A^{v_i} \longrightarrow A_{s_i}$ ('I' for implementation) defined by

$$(\forall a \in A^{v_i})\ (\ I_i(a) = f_{N_i}(\lambda_i(a), a, \zeta_i)\ )$$

(Notice this function is defined over the algebra determined by N_i, namely B_i.) To obtain the defining equations for U_M from those for V_N we simply replace '$\sigma_i^A(a)$' in the equation for V_i at time t+1 by '$I_i(a)$' to get the equation for U_i at time t'+1. Intuitively then, since the scheme α_N formalises the specification of N 'by value function', all we have to do to obtain a corresponding PR-formalisation of the definition of U_M is find a PR scheme ι_i for each I_i, and then replace each occurrence of the operator symbol σ_i in α_N with this scheme. Since f_{N_i} is primitive recursive over B_i (via the scheme δ_{N_i}; see Theorem 3.4.4), it follows that if λ_i is also primitive recursive over B_i, then so is I_i; that is, there exists some $\iota_i \in \mathrm{PR}(\Omega_i)$ such that $[\![\iota_i]\!]_{B_i} = I_i$ when Ω_i is the signature of B_i. Now let B be the algebra comprising all the clocks and operations of $B_1, \ldots, B_k$. Then the scheme-for-operator-symbol substitution c defined by

$$c(\alpha) = \alpha\{\iota_1/\sigma_1\}\{\ \cdots\ \}\{\iota_k/\sigma_k\}$$

(as in the Augmentation part of Section 3.5, the notation $\{\beta/\sigma\}$ is read as 'except that β is substituted for σ') maps each PR scheme over the signature of $\underline{A}$ to a PR scheme over the signature of $\underline{B}$. Moreover, it is not difficult to check that if we define $\alpha_M = c(\alpha_N)$ then α_M satisfies $[\![\alpha_M]\!]_{\underline{B}} = U_M$. Importantly then, whilst a network system is not strictly a synchronous network in the sense of Chapter 2, it is still specifiable in PR: indeed, it is specifiable over the algebra that we want it to be specified.

We will not stop to prove any of the assertions made above here. The point to notice is that with our definition of what it means for a synchronous network to 'implement' a module, network-for-module substitution is modelled by scheme-for-operator substitution, and so like the way that the theory of synchronous networks is the theory of PR, the theory of hierarchical network decomposition is the theory of implementing the operations of one algebra by PR schema over another. In Sections 8.2 - 8.4 we will develop a general theory of computing one algebra by PR schema over another, and we will apply this theory to a concrete example in Section 8.5.

Before we do so however, there is one final case to be considered:

8.1.3 Change in Data Sets.

In this section we will explore how to formalise the result of network substitution when the subnetwork implementations of a synchronous network's modules are defined over different algebras with different data sets.

The kind of change we have in mind is where each datum a of a high-level algebra A is represented by a vector of data over a lower-level algebra B. We will consider a more general setting in the general theory to follow, but for the time being let us assume that A and B are single-sorted, and that for each $a \in A$ there is some $b = (b_1,...,b_r) \in B^r$ to represent a over B. The obvious example here is that introduced at the beginning of the chapter, namely: $A = \mathbf{Z}_n$, $B = \mathbf{Z}_1$, and $r = n$; here a datum $z \in \mathbf{Z}_n$ is represented by $b = (b_1,...,b_n) \in \mathbf{Z}_1^n$ iff $b_1 \cdots b_n$ is the binary representation of z.

Let us begin with an n-source, k-module, m-sink synchronous algorithm N over A with clock T. We will leave the case that N has nonautonomous modules to the general theory in the following sections: here we will assume that all of N's modules are autonomous. If we wish to transform N to a network system M over B, it is intuitively clear that each source and sink of N will become r sources and r sinks respectively, and each channel in N will become r channels in M; correspondingly, each l-input module in N will become an lr-input, r-output synchronous network over B (r outputs since a module is always single-valued.)

We think of such a network system as having a 'synchronous semantics' in essentially the same way as before: M operates in steps wherein each sub-network is executed in parallel for a (possibly data-dependent) number of clock cycles on a fixed input, and the steps determine a clock T'. The only difference here is that since a sub-network now has r outputs, it has r exit modules, the value of each of which needs to be specified at each time $t' \in T'$. Let us explain how this is accomplished:

Suppose that V_N has functionality

$$V_N : T \times [T \longrightarrow A^n] \times A^k \longrightarrow A^k$$

(Recall A is single-sorted here.) In the transformation from N to M induced by sub-networks $N_1,...,N_k$, M will have rn sources and rk exit modules, and so U_M needs to have functionality

$$U_M : T' \times [T' \longrightarrow B^{rn}] \times B^{rk} \longrightarrow B^{rk}$$

Let us use the notation '$U_{i,j}$' with i ranging over $[1,k]$ and j over $[1,r]$ to denote the (rk) coordinates of U_M: here the idea is that for an input stream $\underline{a}' : T' \longrightarrow B^{rn}$ and a vector $y \in B^{rk}$ to specify the initial values of the (rk) exit modules, $U_{i,j}(t',\underline{a}',y)$ is to be the value held by the jth exit module of N_i at time t'. We carry the double-indexing to $\underline{a}'$ and y in the obvious way: each source In_i of N becomes r sources $In_{i,1},...,In_{i,r}$ (say) and so we use '$\underline{a}'_{i,j}$' to denote the stream of data supplied by $In_{i,j}$. Of course we use '$y_{i,j}$' to denote the initial value held by the jth exit module of N_i at $t' = 0$.

What are the defining equations for each $U_{i,j}$? At $t' = 0$ this is straightforward: given our interpretation of the vector y, the equation

$$U_{i,j}(0,\underline{a}',y) = y_{i,j}$$

is appropriate for $i = 1,...,k$ and $j = 1,...,r$.

To define $U_{i,j}$ at time $t'+1$ we must first consider how the sub-networks $N_1,...,N_k$ are specified. Since each N_i is regarded as being executed on fixed input data it is again appropriate to consider a static specification of N_i, and since we are only interested the values held by N_i's exit modules we want the static output specification f_{N_i}. Suppose this function has functionality

$$f_{N_i}: T_i \times B^{m_i} \times B^{k_i} \longrightarrow B^r$$

Here n_i is the number of inputs to m_i and k_i is the number of modules in N_i. Then for a given function $\lambda_i : B^{m_i} \longrightarrow T_i$ and vector of constants $\zeta_i \in B^{k_i}$, $f_{N_i}(\lambda_i(a), a, \zeta_i)$ is the *vector* of values collectively held by N_i's exit modules at the end of a step (of M) when a was the input to N_i at the beginning of the step.

Now, the defining equation for the ith coordinate V_i of V_N is

$$V_i(t+1, \underline{a}, x) = \sigma_i^A(b) \tag{5}$$

where $b = (b_1, \ldots, b_{n_i})$ is defined by

$$b_j = \begin{cases} \underline{a}_p(t) & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ V_q(t, \underline{a}, x) & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

In the case of M, the equation (5) expands to r equations for $U_{i,j}(t'+1, \underline{a}', y)$ (for $j = 1, \ldots, r$). From what we have said above, it should be clear that if f_{N_i} denotes the jth coordinate function of f_{N_i} then the appropriate equation is

$$U_{i,j}(t'+1, \underline{a}', x) = f_{N_i}(\lambda_i(b_i), b_i, \zeta_i)$$

where $b_i = (b_{i,1}, \ldots, b_{i,n_i})$ is defined by

$$b_{ij} = \begin{cases} (\underline{a}'_{p,1}(t'), \ldots, \underline{a}'_{p,r}(t')) & \text{if the } j\text{th input to } m_i \text{ is from source } In_p \\ (U_{q,1}(t', \underline{a}', y), \ldots, U_{q,r}(t', \underline{a}', y)) & \text{if the } j\text{th input to } m_i \text{ is from module } m_q \end{cases}$$

for $j = 1, \ldots, n_i$. □

As in the previous case (where there is no change in data sets), we should show that there is a PR scheme that formalises the above definition of U_M. This we leave to the general theory and the worked example in the following sections.

8.2 CODING AND RETIMING.

In this section we will first formally define what it means for data from one algebra to be 'represented' over another algebra; such a definition is essential to compare algorithm behaviour at different levels of abstraction. Secondly (beginning in Section 8.2.16), we introduce a class of retimings that we use to relate algorithm performance at different levels of abstraction.

Throughout this section let A_i be an S_i-sorted algebra for some sort set S_i for $i = 1,2,3$. (Actually, the material of this section concerns only the carriers of these algebras, and not the constants and operations; thus it suffices to take A_i to be an S_i-indexed family of sets for the purposes of this section. Ultimately, A_i and A_{i+1} will be the algebras of PR systems L_i and L_{i+1}, where we think of A_i as being 'higher-level' than A_{i+1}.)

Also, for a general S-sorted algebra (S-indexed family of sets) A, we define

$$A^+ = \bigcup_{w \in S^+} A^w$$

Finally, if $f : X \longrightarrow Y$ is a surjection, then we write '$f : X \xrightarrow{\text{onto}} Y$'.

8.2.1 Coding Theory.

In order for us to implement a high level algorithm over a low level data type, we must first discuss how high level data can be represented at the lower level. We have already seen one example of this: $\mathbb{Z}_n$ was represented over $\mathbb{Z}_1$ in the sense that $b=(b_1,...,b_n)\in\mathbb{Z}_1^n$ codes or represents $z\in\mathbb{Z}_n$ iff $b_1\cdots b_n$ is z expressed in binary notation. Of course, every $z\in\mathbb{Z}_n$ has such a representation.

Most generally, when we wish to represent elements of A_1 over A_2, that for *every* $a\in A_1$ there is *some* $a'\in A_2$ representing it, is clearly a necessary property, and one that is easily incorporated in the following

8.2.2 Definition.

Let $\gamma: S_1\longrightarrow[A_2^+\longrightarrow A_1]$. We say γ is an A_2-*coding* of A_1 (or A_1 is A_2-*coded by* γ) if there exists a mapping $w=w_\gamma: S_1\longrightarrow S_2^+$ such that for each $s\in S_1$,

$$\gamma(s): A_2^{w(s)}\xrightarrow{\text{onto}}A_1^s$$

When we need to refer explicitly to the mapping w we say γ is an A_2-coding of A_1 *via* w.

Notes.

(i) When A_1 and A_2 are both standard, we will always assume the standard domains (viz T and $\mathbb{B}$) are coded by themselves; that is, for $s\in\{\mathrm{T},\mathrm{B}\}$, $w(s)=s$, and $\gamma(s)$ is the identity mapping; in this case we say both w and γ are *standard*. (To be more precise, we should say that for each $s\in\{\mathrm{T},\mathrm{B}\}$ $\gamma(s)$ is the unique isomorphism between A_1^s and A_2^s; recall such an isomorphism does exist from Section 3.1.8.)

(ii) Notice that an A_2-coding γ of A_1 can be rephrased as an S_1-indexed family of functions $\gamma_s: A_2^{w(s)}\xrightarrow{\text{onto}}A_1^s$. We will not do this for purely notational reasons: whilst it is consistent with our previous definitions (of semantic functions etc.) to use an S-indexed notation for γ, the subscripts rapidly become unwieldy and so we write '$\gamma(s)$' rather than 'γ_s'. □

The idea behind the above definition is that each carrier of the algebra A_1 is coded by a Cartesian product of carriers of the algebra A_2 in the sense that every element of $a\in A_1^s$ has a representation or *code* $b\in A_2^{w(s)}$ such that $\gamma(s)(b)=a$.

8.2.3 Example.

Let A_1 comprise the two carriers $\mathbb{Z}$ and $\mathbb{Z}_n$. Then A_1 is S_1-sorted when $S_1=\{\mathrm{Z},\mathrm{Z}_n\}$ (say) with Z naming $\mathbb{Z}$ and Z_n naming $\mathbb{Z}_n$. Let A_2 comprise $\mathbb{B}$ and $\mathbb{N}$ as carriers. Then A_2 is S_2-sorted when $S_2=\{\mathrm{B},\mathrm{N}\}$ with B naming $\mathbb{B}$ and N naming $\mathbb{N}$.

Coding $\mathbb{Z}_n$ over A_2 is straightforward: define $w: S_1\longrightarrow S_2^+$ at $s=\mathrm{Z}_n$ by $w(\mathrm{Z}_n)=\mathrm{B}\cdots\mathrm{B}$ (n times B), and define $\gamma: S_1\longrightarrow[A_2^+\longrightarrow A_1]$ at $s=\mathrm{Z}_n$ by

$$\gamma(\mathrm{Z}_n)(b_1,...,b_n)=\sum_{k=0}^{k=n-1}2^k\cdot c_{n-k}$$

where $b_1,...,b_n\in\mathbb{B}$, and $c_i=1$ if $b_i=tt$, and $c_i=0$ if $b_i=ff$ for $i=1,...,n$. Clearly,

$$\gamma(Z_n) : \mathbb{B}^n \xrightarrow{\text{onto}} \mathbf{Z}_n$$

That is,

$$\gamma(Z_n) : A_2^{w(z_n)} \xrightarrow{\text{onto}} A_1^{z_n}$$

Thus, if we take $n = 4$ (say), then $3 \in \mathbf{Z}_n$ has code (ff,ff,tt,tt).

It remains to code $\mathbf{Z}$ over A_2, that is, we must define $w(Z)$ and $\gamma(Z)$.

Informally, there are two parts to an integer $z \in \mathbf{Z}$: its sign (positive or negative), and its modulus (or absolute value) $|z|$. The modulus of an integer is just a number $n \in \mathbb{N}$ of course, and the sign of an integer can be represented by a Boolean $b \in \mathbb{B}$: $b = tt$ meaning 'positive', and $b = ff$ meaning 'negative'. We use this idea to define the remaining part of our coding as follows:

Let $w(Z) = BN \in S_2^2$. Also, define $\gamma(Z) = \theta$ where $\theta : \mathbb{B} \times \mathbb{N} \longrightarrow \mathbf{Z}$ is defined by

$$\theta(b,n) = \begin{cases} n & \text{if } b = tt \\ -n & \text{if } b = ff \end{cases} \tag{6}$$

Then it is easy to check that

$$\gamma(Z) : \mathbb{B} \times \mathbb{N} \xrightarrow{\text{onto}} \mathbf{Z}$$

That is,

$$\gamma(Z) : A_2^{w(z)} \xrightarrow{\text{onto}} A_1^{z}$$

and thus A_1 is A_2-coded by γ via w.

Thus for example, $-3 \in \mathbf{Z}$ has code $(ff,3)$, whereas $+3 \in \mathbf{Z}$ has code $(tt,3)$. Notice that $0 \in \mathbf{Z}$ has *two* codes namely $(ff,0)$ and $(tt,0)$; injectivity is not a constraint that we need to impose on our codings. □

We now wish to extend a coding of a carrier set to a coding of a Cartesian product of carriers. Intuitively, if certain vectors b_i are codes of certain elements a_i then the vector $(b_1,\ldots,b_n)$ is a likely candidate for a code of the vector $(a_1,\ldots,a_n)$. Before we can formalise this idea, we need the idea of a *direct product* of functions:

8.2.4 Direct Products.

Let X_i, Y_i be any sets for $i = 1,2.$ and let $f : X_1 \longrightarrow Y_1$ and let $g : X_2 \longrightarrow Y_2$. The *direct product* of f and g is the mapping $(f \times g) : X_1 \times X_2 \longrightarrow Y_1 \times Y_2$ defined by

$$(\forall x_1 \in X_1)(\forall x_2 \in X_2) \ \ ((f \times g)(x_1,x_2) = (f(x_1),g(x_2)))$$

Here are three simple facts about direct products:

8.2.5 Lemma. *The direct product operator is associative.*

Proof. Let $f_i : X_i \longrightarrow Y_i$ for any sets X_i and Y_i for $i = 1,2,3$. Then to prove the lemma we must show

$$(\forall x_1 \in X_1)(\forall x_2 \in X_2)(\forall x_3 \in X_3) \ \ (((f_1 \times f_2) \times f_3)(x_1,x_2,x_3) = (f_1 \times (f_2 \times f_3))(x_1,x_2,x_3))$$

This we leave as an exercise. □

In view of the preceding Lemma we can write $(f_1 \times \cdots \times f_n)$ for the direct product of $f_1,\ldots,f_n$ (that is, without further parentheses).

8.2.6 Lemma. *Function composition distributes over direct products.*

Proof. Let $f_i : Y_i \longrightarrow Z_i$ and $g_i : X_i \longrightarrow Y_i$ for any sets X_i, Y_i, Z_i for $i = 1,2$. Then to prove the Lemma we must show

$$(\forall x_1 \in X_1)(\forall x_2 \in X_2)\ \ (((f_1 \times f_2) \circ (g_1 \times g_2))(x_1,x_2) = ((f_1 \circ g_1) \times (f_2 \circ g_2))(x_1,x_2))$$

This we leave as an exercise. □

8.2.7 Lemma. *Surjectivity is preserved under direct products.*

Proof. Let $f_i : X_i \xrightarrow{\text{onto}} Y_i$ for any sets X_i, Y_i for $i = 1,\ldots,n$. Then to prove the Lemma we must show

$$(f_1 \times \cdots \times f_n) : X_1 \times \cdots \times X_n \xrightarrow{\text{onto}} Y_1 \times \cdots \times Y_n$$

Again, we leave this as an exercise. □

Using the notion of a direct product we can code Cartesian products in the following way:

8.2.8 Definition.

Let A_1 be A_2-coded by γ via w. We extend w and γ as follows:

(i) Extend w from $w : S_1 \longrightarrow S_2^+$ to $w : S_1^+ \longrightarrow S_2^+$ by defining

$$w(v) = w(v_1) \cdots w(v_n)$$

for each $v \in S_1^+$ with $|v| = n$. (Here '$\cdots$' is concatenation over S_2^+ which we understand as the point-wise extension of concatenation over S_2; for example, if $u \in S_2^m$ and $v \in S_2^n$, then by $u \cdot v$ we mean the word $w \in S_2^{n+m}$ defined by

$$w_i = \begin{cases} u_i & \text{if } 1 \le i \le m \\ v_{i-m} & \text{otherwise} \end{cases}$$

for $i = 1,\ldots,n+m$.)

(ii) Extend γ from $\gamma : S_1 \longrightarrow [A_2^+ \longrightarrow A_1]$ to $\gamma : S_1^+ \longrightarrow [A_2^+ \longrightarrow A_1^+]$ by defining

$$\gamma(v) = (\gamma(v_1) \times \cdots \times \gamma(v_n))$$

for each $v \in S_1^+$ with $|v| = n$.

Notice that since

$$\gamma(v_i) : A_2^{w(v_i)} \xrightarrow{\text{onto}} A_1^{v_i}$$

for $i = 1,\ldots,n$, we have

$$\gamma(v) : A_2^{w(v_1)} \times \cdots \times A_2^{w(v_n)} \xrightarrow{\text{onto}} A_1^{v_1} \times \cdots \times A_1^{v_n}$$

by Lemma 8.2.7, that is,

$$\gamma(v) : A_2^{w(v)} \xrightarrow{\text{onto}} A_1^{v}$$

Example. Let γ and w be as in Example 8.2.3. Then applying the preceding definition to the product $\mathbb{Z} \times \mathbb{Z}_4 \times \mathbb{Z}$ for example, we have

$$w(\mathrm{ZZ_4Z}) = w(\mathrm{Z})w(\mathrm{Z_4})w(\mathrm{Z}) = (\mathrm{BN})(\mathrm{BBBB})(\mathrm{BN})$$

(where '$\mathrm{Z_4}$' names $\mathbb{Z}_4$) and

$$\gamma(\mathrm{ZZ_4Z}) = \gamma(\mathrm{Z}) \times \gamma(\mathrm{Z_4}) \times \gamma(\mathrm{Z}) : (\mathbb{B} \times \mathbb{N}) \times (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}) \times (\mathbb{B} \times \mathbb{N}) \longrightarrow \mathbb{Z} \times \mathbb{Z}_4 \times \mathbb{Z}$$

(Here we have included extra parentheses purely for emphasis.) Also, if we take $z_1 = +2$, $z = 13$, and

$z_2=-2$ for example, then $(z_1,z,z_2)\in \mathbf{Z}\times\mathbf{Z}_4\times\mathbf{Z}$ has code $(tt,2,tt,tt,ff,tt,ff,2)$. To formally verify this, we calculate as follows:

$$\begin{aligned}\gamma(\mathrm{ZZ_4Z})(tt,2,tt,tt,ff,tt,ff,2) &= (\gamma(\mathrm{Z})\times\gamma(\mathrm{Z_4})\times\gamma(\mathrm{Z}))(tt,2,tt,tt,ff,tt,ff,2)\\ &= (\gamma(\mathrm{Z})(tt,2),\gamma(\mathrm{Z_4})(tt,tt,ff,tt),\gamma(\mathrm{Z})(ff,2))\\ &= (2,13,-2)\end{aligned}$$

□

We can now support the intuition above: the following Lemma can be read as saying 'a vector of codes of elements is a code of the vector of elements'.

8.2.9 Lemma. *If A_1 is A_2-coded by γ via w, then for each $v\in S_1^+$,*

$$\gamma(v):A_2^{w(v)}\xrightarrow{\text{onto}}A_1^v$$

Proof. Immediate from Lemma 8.2.7. □

We are now in a position to define compiler correctness in the context of two algebras which have different carrier sets:

8.2.10 Definition.

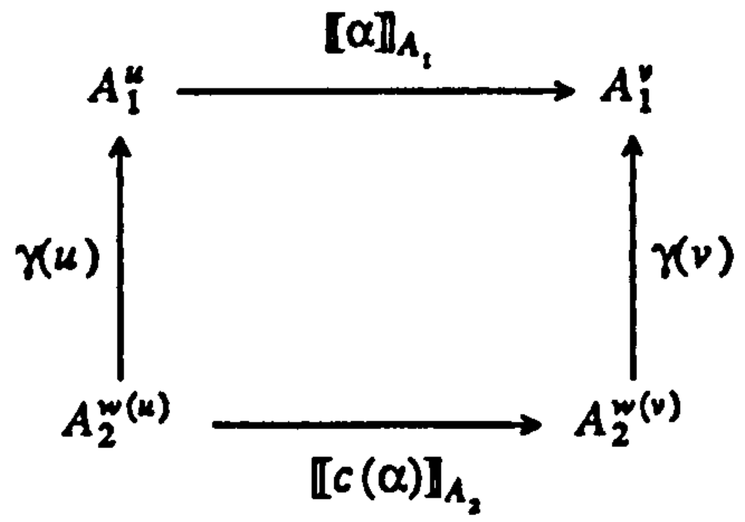
Let L_1 and L_2 be PR systems, and suppose A_1 is A_2-coded by γ via w. Then

(i) An (L_1,L_2)-*compiler* c is a $S_1^+\times S_1^+$-indexed family of mappings

$$c=<c^{u,v}:u,v\in S_1^+>$$

such that $c^{u,v}:\mathrm{PR}(\Sigma_1)_{u,v}\longrightarrow\mathrm{PR}(\Sigma_2)_{w(u),w(v)}$ for each $u,v\in S_1^+$.

(ii) An (L_1,L_2)-compiler c is said to be *correct with respect to* γ if for every $u,v\in S_1^+$, and for every $\alpha\in\mathrm{PR}(\Sigma_1)_{u,v}$, the following diagram commutes for every $a\in A_2^{w(u)}$:



That is, if

$$(\forall a\in A_2^{w(u)})\ (\gamma(v)([\![c^{u,v}(\alpha)]\!]_{A_2}(a))=[\![\alpha]\!]_{A_1}(\gamma(u)(a)))$$

Discussion. An (L_1,L_2)-compiler is correct when for each source program and for each (legal) input, given a *code* of the input, the object program computes the *code* of the output of the source program. Notice that in the case that A_1 and A_2 have the same (or isomorphic) carriers, we can take γ to be the identity function (or the unique isomorphism) and the above definition of compiler correctness coincides with Definition 7.1.1(iii). Additionally, we note that the idea of a coding is independent of PR and so whilst the preceding definition mentions PR explicitly, it is easy to see how it can be generalised to other functional languages (in the sense of Section 7.1).

In Section 8.2.16 we will consider compiler performance preservation in the current more general setting (where L_1 and L_2 have different underlying algebras, performance measures, and clocks); however, before we leave the concept of a coding, we will gather together some useful facts about codings in general. In particular, in order to discuss compiling down through many levels of data abstraction $A_1,\ldots,A_n$ we need to be able to compose codings. After the following technical lemma we will prove that the composition of two codings is again a coding. Thereafter we establish that if A_i is A_{i+1}-coded for $i=1,\ldots,n-1$, then there exists an A_n-coding of A_1.

8.2.11 Lemma. *Let γ be an A_2-coding of A_1, let S_0 be any (sort) set, and let $w: S_0 \longrightarrow S_1^+$ be any mapping. Then for every $v \in S_0^+$,*

$$\gamma(w(v)) = \gamma(w(v_1)) \times \cdots \times \gamma(w(v_n))$$

where $n = |v|$.

Proof. Let $v \in S_0^+$ with $|v| = n$, and for $i=1,\ldots,n$ let $w(v_i) = u_i \in S_1^+$ and let $r_i = |u_i|$ for $i=1,\ldots,n$. Additionally, suppose $u_i = s_1^i \cdots s_{r_i}^i$ for some sorts $s_j^i \in S_1$. Now, from Definition 8.2.8 we have

$$\begin{aligned}\gamma(w(v)) &= \gamma(w(v_1) \cdots w(v_i) \cdots w(v_n)) \\ &= \gamma(u_1 \cdots u_i \cdots u_n)\end{aligned}$$

(by definition of $u_1,\ldots,u_n$)

$$= \gamma(s_1^1,\ldots,s_{r_1}^1,\ldots,s_1^i,\ldots,s_{r_i}^i,\ldots,s_1^n,\ldots,s_{r_n}^n)$$

(by definition of s_j^i for $j=1,\ldots,r_i$, $i=1,\ldots,n$)

$$= \gamma(s_1^1) \times \cdots \times \gamma(s_{r_1}^1) \times \cdots \times \gamma(s_1^i) \times \cdots \times \gamma(s_{r_i}^i) \times \cdots \times \gamma(s_1^n) \times \cdots \times \gamma(s_{r_n}^n)$$

(see Definition 8.2.8)

$$= (\gamma(s_1^1) \times \cdots \times \gamma(s_{r_1}^1)) \times \cdots \times (\gamma(s_1^i) \times \cdots \times \gamma(s_{r_i}^i)) \times \cdots \times (\gamma(s_1^n) \times \cdots \times \gamma(s_{r_n}^n))$$

(by Lemma 8.2.5)

$$= \gamma(s_1^1 \cdots s_{r_1}^1) \times \cdots \times \gamma(s_1^i \cdots s_{r_i}^i) \times \cdots \times \gamma(s_1^n \cdots s_{r_n}^n)$$

(see Definition 8.2.8)

$$= \gamma(u_1) \times \cdots \times \gamma(u_i) \times \cdots \times \gamma(u_n)$$

(by definition of $u_1,\ldots,u_n$)

$$= \gamma(w(v_1)) \times \cdots \times \gamma(w(v_n))$$

□

The following lemma tells us that we can compose codings in a natural way:

8.2.12 Composition Lemma. *Let γ_i be an A_{i+1}-coding of A_i via $w_i: S_i \longrightarrow S_{i+1}^+$ for $i=1,2$, and define $\gamma: S_1 \longrightarrow [A_3^+ \longrightarrow A_1]$ by*

$$\gamma(s) = \gamma_1(s) \circ \gamma_2(w_1(s)) \tag{7}$$

for each $s \in S_1$. Also define $w = w_\gamma: S_1 \longrightarrow S_3^+$ by $w = w_2 \circ w_1$. Then,

(i) γ is an A_3-coding of A_1 via w, and

(ii) For each $v \in S_1^+$, $\gamma(v) = \gamma_1(v) \circ \gamma_2(w_1(v))$

Proof. To show (i), first notice since γ_1 is an A_2-coding of A_1 via w_1 we have

$$w_1 : S_1 \longrightarrow S_2^+ \tag{8}$$

and for each $s \in S_1$,

$$\gamma_1(s) : A_2^{w_1(s)} \xrightarrow{\text{onto}} A_1^s \tag{9}$$

Secondly, since γ_2 is an A_3-coding of A_2 via w_2 we have

$$w_2 : S_2^+ \longrightarrow S_3^+ \tag{10}$$

(when the domain of w_2 has been extended to S_2^+; see Definition 8.2.8), and for each $v \in S_1^+$,

$$\gamma_2(v) : A_3^{w_2(v)} \xrightarrow{\text{onto}} A_2^v \tag{11}$$

by Lemma 8.2.9.

Thus by (8) and (10), $w : S_1 \longrightarrow S_3^+$, and by the definition of γ (7) and by (9) and (11) (with $v = w_1(s) \in S_2^+$), we have

$$\gamma(s) : A_3^{w_2(w_1(s))} \xrightarrow{\text{onto}} A_1^s$$

for each $s \in S_1$ since surjectivity is preserved under functional composition (proof: exercise), and thus γ is an A_3-coding of A_1 via w as claimed.

To show (ii), choose $v \in S_1^+$ and let $|v| = n$. Then from the definition of $\gamma(v)$ (see Definition 8.2.8) we have

$$\begin{aligned}
\gamma(v) &= \gamma(v_1) \times \cdots \times \gamma(v_n) \\
&= \gamma_1(v_1) \circ \gamma_2(w_1(v_1)) \times \cdots \times \gamma_1(v_n) \circ \gamma_2(w_1(v_n)) \\
&\quad \text{(by (7))} \\
&= (\gamma_1(v_1) \times \cdots \times \gamma_1(v_n)) \circ (\gamma_2(w_1(v_1)) \times \cdots \times \gamma_2(w_1(v_n))) \\
&\quad \text{(by Lemma 8.2.6)} \\
&= \gamma_1(v) \circ (\gamma_2(w_1(v_1)) \times \cdots \times \gamma_2(w_1(v_n))) \\
&\quad \text{(see Definition 8.2.8)} \\
&= \gamma_1(v) \circ \gamma_2(w_1(v)) \\
&\quad \text{(by Lemma 8.2.11).}
\end{aligned}$$

□

8.2.13 Notation.

In the situation of Lemma 8.2.12, we write $\gamma = \gamma_1 \circ \gamma_2$ for the mapping γ defined by (7) of that lemma. □

An easy corollary to Composition Lemma 8.2.12 is the following

8.2.14 Hierarchy Lemma. *Let A_i be A_{i+1}-coded by γ_i via w_i for $i = 1, \ldots, n$, and define $\gamma^{(n)} : S_1 \longrightarrow [A_{n+1}^+ \longrightarrow A_1]$ by*

$$\gamma^{(n)}(s) = \gamma_1(w_0(s)) \circ \cdots \circ \gamma_n(w_{n-1}(s)) \tag{12}$$

for each $s \in S_1$, and where $w_0(s) = s$ for each $s \in S_1$. Also define $w^{(n)} = w_{\gamma^{(n)}}$ by

$$w^{(n)} = w_n \circ \cdots \circ w_1 \tag{13}$$

Then A_1 is A_{n+1}-coded by $\gamma^{(n)}$ via $w^{(n)}$.

Proof. By induction on n.

Basis. Suppose $n=1$. Here, A_1 is A_{n+1}-coded by $\gamma^{(n)}$ via $w^{(n)}$ by hypothesis (since A_{n+1} is A_2, and it is easy to see $\gamma^{(1)}=\gamma_1$ from (12) and $w^{(1)}=w_1$ from (13)).

Induction. Suppose for some fixed $k \geq 1$ that for $n=1,\ldots,k$, if A_i is A_{i+1}-coded by γ_i via w_i for $i=1,\ldots,n$, and if $\gamma^{(n)}: S_1 \longrightarrow [A_{n+1}^+ \longrightarrow A_1]$ is defined by

$$\gamma^{(n)}(s) = \gamma_1(w_0(s)) \circ \cdots \circ \gamma_n(w_{n-1}(s)) \tag{14}$$

for each $s \in S_1$, and where $w_0(s)=s$ for each $s \in S_1$, then, A_1 is A_{n+1}-coded by $\gamma^{(n)}$ via $w^{(n)}=w_{\gamma^{(n)}}$ where $w^{(n)}$ is defined by

$$w^{(n)} = w_n \circ \cdots \circ w_1 \tag{15}$$

Now suppose A_i is A_{i+1}-coded by γ_i via w_i for $i=1,\ldots,k+1$. We must show A_1 is A_{k+2}-coded by $\gamma^{(k+1)}$ via $w^{(k+1)}=w_{\gamma^{(k+1)}}$ where $\gamma^{(k+1)}: S_1 \longrightarrow [A_{k+2}^+ \longrightarrow A_1]$ and $w^{(k+1)}$ are defined by

$$\gamma^{(k+1)}(s) = \gamma_1(w_0(s)) \circ \cdots \circ \gamma_{k+1}(w_k(s)) \tag{16}$$

for each $s \in S_1$, and

$$w^{(k+1)} = w_{k+1} \circ \cdots \circ w_1 \tag{17}$$

where for $i=1,\ldots,k+1$, $w_i = w_{\gamma_i}$.

By the induction hypothesis, we can assume A_1 is A_{k+1}-coded by $\gamma^{(k)}$ via $w^{(k)}$, and by hypothesis A_{k+1} is A_{k+2}-coded by γ_{k+1} via w_{k+1}. Thus, by Lemma 8.2.12(i) A_1 is A_{k+2}-coded by γ via w when for each $s \in S_1$, $\gamma(s)$ is defined by

$$\gamma(s) = \gamma^{(k)}(s) \circ \gamma_{k+1}(w^{(k)}(s))$$

and where w is defined by

$$w = w_{k+1} \circ w^{(k)}$$

However, $\gamma^{(k)}(s) \circ \gamma_{k+1}(w^{(k)}(s)) = \gamma^{(k+1)}$ by (14) and (16), and $w_{k+1} \circ w^{(k)} = w^{(k+1)}$ by (15) and (17).

Thus A_1 is A_{k+2}-coded by $\gamma^{(k+1)}$ via $w^{(k+1)}$ as claimed, completing the induction step. □

8.2.15 Notation.

In the situation of the preceding lemma, we write

$$\gamma = \gamma_1 \circ \cdots \circ \gamma_n$$

for the mapping γ defined by (12) of the lemma.

8.2.16 Retiming.

We now wish to consider performance preservation properties of (L_1,L_2)-compilers c. Notice that for given $\alpha \in \mathrm{PR}(\Sigma_1)$ we cannot directly compare the complexity of $c(\alpha)$ to that of α since these complexities are measured with respect to different clocks.

Suppose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ and let A_1 be A_2-coded by γ via w; then $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$. Given an input $a \in A_2^{w(u)}$, it is appropriate to want to relate the execution time of $c(\alpha)$ on a to that of α on $\gamma(u)(a)$, formalising the idea that we want to compare α and $c(\alpha)$ on the 'same' input. The time taken to execute $c(\alpha)$ on a is $\lambda_{P_2}(c(\alpha))(a) \in C_2$ whereas the time taken to execute α on $\gamma(u)(a)$ is

$\lambda_{P_1}(\alpha)(\gamma(u)(a)) \in C_1$. However, as we remarked in Chapter 3, the choice of performance measures P_1 and P_2 may reflect different implementation concerns and thus the underlying system clocks are properly *not* identified.

Definition. Let C_1 and C_2 be clocks. Then a (C_1,C_2)-*retiming* is any mapping $\Psi: C_1 \longrightarrow C_2$. □

Intuitively, a retiming $\Psi: C_1 \longrightarrow C_2$ tells us the time $\Psi(c)$ on the low level clock C_2 when the high level clock C_1 says that the time is c. Using the idea of a retiming we frame the following:

Definition. Let c be an (L_1,L_2)-compiler, and let A_1 be A_2-coded by γ via w. Now let $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u,v \in S_1^+$ and let Ψ be a (C_1,C_2)-retiming. Then we say c preserves the performance of α *with respect to* Ψ if

$$(\forall a \in A_2^{w(u)})\ (\lambda_{P_2}(c(\alpha))(a) \leq \Psi(\lambda_{P_1}(\alpha)(\gamma(u)(a))))$$

Discussion. In the case where A_1 and A_2 are both S-sorted with the same (or isomorphic) carries and γ is the identity function (or unique isomorphism), our definition of relative compiler correctness (Definition 8.2.10) degenerated into our previous definition of a correct compiler given in Chapter 7. However, it is clear that a corresponding remark cannot be made about compiler performance preservation: the definition of a performance preserving compiler given in Definition 7.1.1(iii) requires that the complexity of $c(\alpha)$ for each scheme α bounds, and is bounded by the complexity of α, whereas the above definition of relative performance preservation does not require that the complexity of α be bounded by the complexity of $c(\alpha)$. The reason for this is that in this chapter we not interested in the computational equivalence of languages *per se*, rather, we are interested in top-down design and so our primary objective is to guarantee the end product $c(\alpha)$ in the senses that (a) it correctly implements α, and (b) it is an efficient implementation by which mean that its complexity is bounded by (some function of) the complexity of α. Condition (b), the 'interesting' half of Definition 7.1.1(iii), can be recovered in the following way:

Suppose C_1 and C_2 are the same clock C (that is, in addition to supposing that A_1 and A_2 have the same carriers and w and γ being identity functions). Now suppose that c is an (L_1,L_2)-compiler which preserves the performance of any scheme α with respect to $\Psi = \Psi_\alpha: C_1 \longrightarrow C_2$ defined by $\Psi(c) = k_\alpha . c$ for some constant $k_\alpha \geq 1$; this means that

$$(\forall a \in A_2^u)\ (\lambda_{P_2}(c(\alpha))(a) \leq k_\alpha . \lambda_{P_1}(\alpha)(a))$$

when the domain of $[\![\alpha]\!]_{A_2}$ is A_2^u. Such a compiler satisfies the first compiler preservation condition of Definition 7.1.1(iii).

8.2.17 Polynomial Retimings.

The case of retimings Ψ which are of the form $\Psi(c) = k . c$ is a natural case to consider since a compiler which preserves the performance of a scheme α with respect to such a retiming preserves the complexity of α to within a constant factor; said differently, there is a *linear* relationship between the complexities of α and $c(\alpha)$ which we view as an identity relationship. A straightforward generalisation of this idea is a *quadratic* relationship between the complexities of α and $c(\alpha)$; that is when the

complexity of $c(\alpha)$ is bounded by (a constant multiple of) the *square* of the complexity of α. More generally, a compiler can be performance preserving in the sense that there is a *polynomial* relationship between the complexities of α and $c(\alpha)$:

Definition. Let $d \geq 0$ and let $n_0,\ldots,n_d \in \mathbb{N}$ where $n_d \neq 0$. Then a *polynomial* is a mapping $p : \mathbb{N} \longrightarrow \mathbb{N}$ of the form

$$p(x) = n_0 + n_1 . x + \cdots + n_d . x^d$$

which we also write as

$$p(x) = \sum_{k=0}^{k=d} n_k . x^k$$

We call d the *degree* of p, in symbols: $deg\ p = d$. The collection of all polynomials (of any degree) is denoted $\mathbb{N}[x]$. (Note that the function $p : \mathbb{N} \longrightarrow \mathbb{N}$ defined by $p(x) = 0$ for each $x \in \mathbb{N}$ is not a polynomial according to this definition).

8.2.18 Definition.

Let C_1 and C_2 be clocks. A (C_1,C_2)-retiming Ψ is said to be *d-adic* if $\Psi \in \mathbb{N}[x]$ with $deg\ \Psi = d$. (For '0-adic' read 'constant', for '1-adic' read 'linear', and for '2-adic' read 'quadratic' etc.) □

8.2.19 Definition.

Let L_1 and L_2 be PR systems, and suppose A_1 is A_2-coded by γ via w. Then a (L_1,L_2)-compiler c is said to be *d-adic* if for every $u,v \in S_1^+$ and for every $\alpha \in PR(\Sigma_1)_{u,v}$, there exists a d-adic retiming $\Psi_\alpha : C_1 \longrightarrow C_2$ such that c preserves the performance of α with respect to Ψ_α; that is, if

$$(\forall a \in A_2^{w(u)})\ \ (\lambda_{P_2}(c^{u,v}(\alpha))(a) \leq \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(u)(a))))$$

Discussion. The idea behind a 'd-adic' compiler is that the number d is a measure of the efficiency of the compiler. For example, if $d = 0$ then the execution time of every compiled scheme is bounded by a constant independent of the complexity of the original scheme, which is very efficient indeed! Alternatively, a 1-adic or linear compiler has the property that the execution time of every compiled scheme is bounded by some constant multiple of the execution time of the original scheme; this is not as efficient as a 0-adic compiler, but intuitively quite acceptable. Of course, as the number d increases, d-adic compilers are progressively less efficient objects; for example, a 3-adic or cubic compiler produces object schema whose complexity is bounded by (a constant multiple of) the *cube* of the complexity of the source scheme.

8.2.20 Retiming Theory.

Below we gather together some isolated definitions and elementary facts concerning $\mathbb{N}[x]$ (in its capacity as a class of retiming functions Ψ); the proof of each of the lemmas below is left as an exercise.

8.2.21 Definitions.

(a) Let $\Psi \in \mathbb{N}[x]$ be defined by

$$\Psi(x) = a_0 + a_1 . x + \cdots + a_d . x^d$$

Then we say Ψ is *homogeneous* iff $a_0 = 0$. (Equivalently, Ψ is homogeneous iff $\Psi(0) = 0$).

(b) Let $\Psi \in \mathbf{N}[x]$. We say Ψ is *monotone* if for every $x,y \in \mathbb{N}$,

$$x \leq y \implies \Psi(x) \leq \Psi(y)$$

(c) Let $\Psi_1,\Psi_2 \in \mathbf{N}[x]$ be defined by

$$\Psi_1(x) = \sum_{i=0}^{i=d} a_i \,.\, x^i \quad \text{and} \quad \Psi_2(x) = \sum_{j=0}^{j=d'} b_j \,.\, x^j$$

Then we define $(\Psi_1+\Psi_2) \in \mathbb{N}[x]$ by

$$(\Psi_1+\Psi_2)(x) = \sum_{k=0}^{k=d''} c_k \,.\, x^k$$

where $d'' = \max\{d,d'\}$ and for $k = 0,\ldots,d''$ c_k is defined by

$$c_k = \begin{cases} (a_k+b_k) & \text{if } 0 \leq k \leq \min\{d,d'\} \\ a_k & d > d' \text{ and } k > d' \\ b_k & \text{otherwise} \end{cases}$$

Notice that $deg\,(\Psi_1+\Psi_2) = \max\{deg\,\Psi_1, deg\,\Psi_2\}$.

(d) Let $\Psi_1,\Psi_2 \in \mathbb{N}[x]$ be as defined above. Then we define $\max(\Psi_1,\Psi_2) \in \mathbb{N}[x]$ by

$$\max(\Psi_1,\Psi_2)(x) = \sum_{k=0}^{k=d''} c_k \,.\, x^k$$

where $d'' = \max\{d,d'\}$ and for $k = 0,\ldots,d''$ c_k is defined by

$$c_k = \begin{cases} \max\{a_k,b_k\} & \text{if } 0 \leq k \leq \min\{d,d'\} \\ a_k & d > d' \text{ and } k > d' \\ b_k & \text{otherwise} \end{cases}$$

Notice that $deg \max(\Psi_1,\Psi_2) = \max\{deg\,\Psi_1, deg\,\Psi_2\}$.

(e) Let $\Psi_1,\Psi_2 \in \mathbb{N}[x]$. Then we define $(\Psi_1 \circ \Psi_2) \in \mathbf{N}[x]$ by

$$(\Psi_1 \circ \Psi_2)(x) = \Psi_1(\Psi_2(x))$$

Notice that $deg\,(\Psi_1 \circ \Psi_2) = deg\,\Psi_1 \,.\, deg\,\Psi_2$.

8.2.22 Lemma. *Let $\Psi_1,\Psi_2 \in \mathbf{N}[x]$. If Ψ_1 and Ψ_2 are both homogeneous, then for every $x,y \in \mathbb{N}$*

$$\Psi_1(x) + \Psi_2(y) \leq (\Psi_1+\Psi_2)(x+y)$$

□

8.2.23 Lemma. *Let $\Psi \in \mathbb{N}[x]$. If Ψ is homogeneous then Ψ is monotone.* □

8.2.24 Lemma. *Let $\Psi_1,\Psi_2,\Psi_3 \in \mathbb{N}[x]$, and let x and y be any natural numbers. Then,*

(a) $\max(\Psi_1,\Psi_2)(x) = \max(\Psi_2,\Psi_1)(x)$

(b) $\max(\Psi_1,\max(\Psi_2,\Psi_3))(x) = \max(\max(\Psi_1,\Psi_2),\Psi_3)(x)$

(c) $\max\{\Psi_1(x),\Psi_2(y)\} \leq \max(\Psi_1,\Psi_2)(\max\{x,y\})$

(d) $\Psi_1(x) \leq \max(\Psi_1,\Psi_2)(\max\{x,y\})$ □

(In view of part (b) of the above lemma we will subsequently write $\max(\Psi_1,\ldots,\Psi_n)$ without further parentheses).

For future use we explicitly phrase the two following facts about polynomials in terms of retimings:

8.2.25 Composition Lemma. *Let C_i be a clock for $i=1,2,3$, and let $\Psi_i : C_i \longrightarrow C_{i+1}$ be a (C_i,C_{i+1})-d_i-adic retiming for $i=1,2$. Then $\Psi \in \mathbb{N}[x]$ defined by $\Psi=\Psi_2 \circ \Psi_1$ is a (C_1,C_3)-d-adic retiming, where $d=d_1 . d_2$.* □

8.2.26 Hierarchy Lemma. *Let C_i be a clock for $i=1,\ldots,n+1$, for any $n \geq 1$, and let $\Psi_i : C_i \longrightarrow C_{i+1}$ be a (C_i,C_{i+1})-d_i-adic retiming for $i=1,\ldots,n$. Then $\Psi \in \mathbb{N}[x]$ defined by $\Psi=\Psi_n \circ \cdots \circ \Psi_1$ is a (C_1,C_{n+1})-d-adic retiming, where $d=d_1 \times \cdots \times d_n$.* □

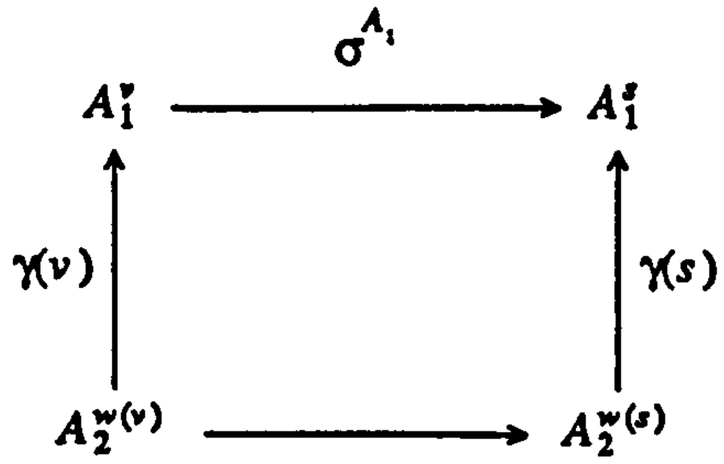
8.3 IMPLEMENTATION THEORY.

We wish be able to state (and prove) a theorem concerning the automatic generation of correct compilers. As we will see in the next section, the existence of a correct (L_1,L_2)-compiler will be guaranteed when we have implementations of the constants and operations of A_1 over A_2. In this section we will define the concept of an *L_2-implementation* of an algebra A_1; such an implementation is a formal package comprising an A_2-coding of A_1 together with two collections of $\mathrm{PR}(\Sigma_2)$-schema that tell us how to compute the constants and operations of A_1 (with respect to the coding) respectively. Also, to keep performance matters firmly in view, we will define the *complexity* of an L_2-implementation; as we will see later, a compiler that is generated by a given implementation is performance preserving in the sense that the compiler never has complexity worse than that of the implementation which generated it.

The way in which we define an implementation of an operation and its complexity is straightforward and will be considered first:

8.3.1 Definition.

Let A_1 be a Σ_1-structure for S_1-sorted signature Σ_1, and let L_2 be a PR system. If A_1 is A_2-coded by γ via $w=w_\gamma$ then we say A_1 is *L_2-tracked* (or sometimes: 'A_2'-tracked) if for every $v \in S_1^+$ and $s \in S_1$, and for every $\sigma \in (\Sigma_1)_{v,s}$, there exists some *$L_2$-tracking* $\alpha_\sigma \in \mathrm{PR}(\Sigma_2)_{w(v),w(s)}$ *of* σ such that the following diagram commutes for every $a \in A_2^{w(v)}$:



That is, if

$$(\forall a \in A_2^{w(v)})\ (\ \gamma(s)([\![\alpha_\sigma]\!]_{A_2}(a)) = \sigma^{A_1}(\gamma(v)(a))\ ) \tag{18}$$

We collect such implementations into a family $Tr = <\alpha_\sigma : \sigma \in \Sigma_1>$ and say A_1 is (L_2)-tracked *by* Tr. Additionally, if P_1 is a performance measure for A_1 which is based on clock C_1, then we say Tr is *d-adic* if there exists an (C_1,C_2)-d-adic retiming Ψ such that for *every* $v \in S_1^+$, $s \in S_1$ and *every* $\sigma \in (\Sigma_1)_{v,s}$,

$$(\forall a \in A_2^{w(v)})\ (\ \lambda_{P_2}(\alpha_\sigma)(a) \le \Psi(\sigma^{P_1}(\gamma(v)(a)))\ )$$

8.3.2 Example.

Continuing Example 8.2.3, let $neg : \mathbb{Z} \longrightarrow \mathbb{Z}$ be negation on $\mathbb{Z}$ (so $neg(3) = -3$, $neg(-2) = 2$, and $neg(0) = 0$ for example). Given the code $(b,n) \in \mathbb{B} \times \mathbb{N}$ of an integer $z \in \mathbb{Z}$, the code of $neg(z)$ is simply $(\sim b, n)$. ('$\sim$' is logical negation.)

Formally, first observe that by definition of θ (see (6)) we have

$$(\forall b \in \mathbb{B})(\forall n \in \mathbb{N})\ (\ \theta(\sim b, n) = neg(\theta(b,n))\ ) \tag{19}$$

Now define α_{neg} by $\alpha_{neg} = <\sim \circ U_1^{BN}, U_2^{BN}>$. Then $\alpha_{neg} \in PR(\Sigma_2)_{BN,BN} = PR(\Sigma_2)_{w(Z),w(Z)}$, and for each $(b,n) \in \mathbb{B} \times \mathbb{N} = A_2^{w(Z)}$ we have

$$\begin{aligned} \gamma(Z)([\![\alpha_{neg}]\!]_{A_2}(b,n)) &= \theta([\![\alpha_{neg}]\!]_{A_2}(b,n)) \\ &= \theta(\sim b, n) \end{aligned}$$

(by definition of α_{neg})

$$= neg(\theta(b,n))$$

(by (19))

$$= neg(\gamma(Z)(b,n))$$

Thus α_{neg} is an L_2-tracking of neg.

8.3.3 Implementing Constants.

We will now consider how to implement or compute the constants of an algebra A_1 by a scheme over A_2. Since constants c^{A_1} only ever appear in $PR(\Sigma_2)$-schema as constant function schema c^w for some $w \in S_1^+$, our objective is to find L_2-implementations of all constant-valued functions over A_1.

In order for us to be able to compute the constant-valued functions of A_1 over A_2, we clearly need some means of accessing or 'reaching' the constants of A_1 from A_2. Recall Example 8.2.3; there, $0 \in \mathbb{Z}$ was accessible from $\mathbb{B} \cup \mathbb{N}$ in the sense that

$$\gamma(Z)(tt,0) = 0 \tag{20}$$

Here we can 'reach' $0 \in \mathbf{Z}$ by virtue of the fact that $\mathbf{Z}$ is $(\mathbb{B} \cup \mathbb{N})^+$-coded. As a slightly more general example, $+2 \in \mathbf{Z}$ can be 'reached' from $\mathbb{B} \cup \mathbb{N}$ in the sense that

$$\gamma(Z)(tt,succ\,(succ\,(0))) = +2 \tag{21}$$

Here we rely on knowing what the coding γ actually is. Now, to impose properties on codings is to invite them to have properties which may not be isomorphism invariant contradicting the intuitive idea that isomorphic algebras are identical. Instead, we will impose a condition on the algebras themselves; this condition generalises a common aspect of (20) and (21) above, namely that in both cases the argument to $\gamma(Z)$ is obtained by *finitely many applications of operations to constants*. We formalise this idea in the following way:

8.3.4 Definition.

Let Σ be an S-sorted signature. We define the collection $T(\Sigma)$ of (*finitely generated*) Σ-*terms* by

$$T(\Sigma) = \bigcup_{w \in S^*} T(\Sigma)_w$$

where for each $w \in S^+$, $T(\Sigma)_w$ is defined uniformly in w by structural induction as follows:

Basis.

(i) *Constants.* For each $s \in S$, if $c \in \Sigma_{\lambda,s}$ then $c \in T(\Sigma)_s$.

Induction.

(ii) *Vectorisation.* If $\rho_i \in T(\Sigma)_{s_i}$ for some $s_i \in S$, $i=1,\ldots,n \geq 1$, then $<\rho_1,\ldots,\rho_n> \in T(\Sigma)_w$ when $w = s_1 \cdots s_n$.

(iii) *Composition.* If $\rho \in T(\Sigma)_w$ for some $w \in S^+$ and $\sigma \in \Sigma_{w,s}$ for some $s \in S$, then $\sigma \circ \rho \in T(\Sigma)_s$.

In addition, if A is a Σ-structure with performance measure P, then we extend the semantic evaluation mapping $(.)^A : \Sigma \longrightarrow A$ to $(.)^A : \Sigma \cup T(\Sigma) \longrightarrow A$ and the performance estimation mapping $(.)^P : \Sigma \longrightarrow P$ to $(.)^P : \Sigma \cup T(\Sigma) \longrightarrow P$ by defining

$$<\rho_1,\ldots,\rho_n>^A = (\rho_1^A,\ldots,\rho_n^A)$$
$$<\rho_1,\ldots,\rho_n>^P = \max_{1 \leq i \leq n}\{\ \rho_i^P\ \}$$

and,

$$(\sigma \circ \rho)^A = \sigma^A(\rho^A)$$
$$(\sigma \circ \rho)^P = \rho^P + \sigma^P(\rho^A)$$

□

The concept of a constant of A_1 that is 'reachable from' A_2 can now be formalised as follows:

8.3.5 Definition.

Let A_i be a Σ_i-structure for S_i-sorted signature Σ_i for $i = 1,2$. Also suppose A_1 is A_2-coded by γ. For each $s \in S_1$, and for each $c \in (\Sigma_1)_{\lambda,s}$ we say $\rho_c \in T(\Sigma_2)_{w_\gamma(s)}$ *generates* c^{A_1} if $\gamma(s)(\rho_c^{A_2}) = c^{A_1}$. (We will also say ρ_c is an A_2-*generation of* c^{A_1}.) Additionally, we say A_1 is A_2-*generated* if every constant of A_1 is generated by some $\rho_c \in T(\Sigma_2)$.

As further notation, when A_1 is A_2-generated we collect all the A_2-generations into a family denoted $Gn = <\rho_c : c \in \Sigma_1>$, and say A_1 is A_2-generated *by* Gn. □

We now make the central definition of this section, namely that of an L_2-implementation of an algebra A_1:

8.3.6 Definition.

Let A_1 be a Σ_1-structure for S_1-sorted signature Σ_1, and let L_2 be a PR system. Now suppose

(i) A_1 is A_2-coded by some coding γ,

(ii) A_1 is A_2-generated by some family Gn of L_2-generators,

(iii) A_1 is L_2-tracked by some family Tr of L_2-tracking functions.

Then we refer to $I = (\gamma, Gn, Tr)$ as an L_2-*implementation* (or sometimes an 'A_2'-implementation) of A_1. As additional notation, when we need to refer to the coding γ of I without explicit reference to the rest of the triple, we write γ_I. Also, I is said to be d-*adic* if Tr is d-adic.

Discussion. As we have said, our goal is to prove the existence of a correct (L_1, L_2)-compiler given an L_2-implementation of A_1. The proof of existence is constructive, and so we can think of an implementation as *generating* the compiler. We will additionally prove that a compiler so generated is performance preserving in the sense that it preserves the complexity of source schema up to the complexity of the implementation that generated it: if I is d-adic then the compiler generated by I is also d-adic. (Actually the case $d = 0$ is an exception: 0-adic implementations generate *linear* compilers.) However, before we can prove this fact we must return to the matter of computing constant functions:

Generations $\rho_c \in T(\Sigma_2)$ of constants c^{A_1} are the key to implementing the constant-valued functions of A_1 by schema over A_2; after a preliminary lemma we will prove this fact.

8.3.7 Lemma. *Let L be a PR system, and let $\rho \in T(\Sigma)_v$ for any $v \in S^+$. Then, for any $u \in S^+$, there exists $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ such that*

$$(\forall a \in A^u)\ (\ [\![\alpha]\!]_A(a) = \rho^A\ ) \tag{22}$$

and

$$(\forall a \in A^u)\ (\ \lambda_p(\alpha)(a) = \rho^p\ ) \tag{23}$$

Proof. We construct α as a function of ρ and v, in symbols, $\alpha = \Delta(\rho, v)$. The construction is by induction on the structure of ρ, and is uniform in v.

Basis Case.

(i) *Constants.* Suppose $\rho \in \Sigma_{\lambda,s}$ for some $s \in S$. Then $\rho = c$ for some constant symbol c (and here, $v = s$).

Take $\alpha = \Delta(\rho, v) = c^u$, then $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, and for each $a \in A^u$ we have

$$[\![\alpha]\!]_A(a) = [\![c^u]\!]_A(a)$$

$$= c^A$$
$$= \rho^A$$

(by definition of ρ) as required.

To show (23), choose $a \in A^u$. Then we calculate as follows:

$$\lambda_P(\alpha)(a) = \lambda_P(c^u)(a)$$
$$= c^P$$
$$= \rho^P$$

(by definition of ρ) as required.

Induction. Let ρ be some fixed (non-basis) element of $T(\Sigma)$.We assume for every $v_0 \in S^+$ and $\rho_0 \in T(\Sigma)_{v_0}$ of less structural complexity than ρ, that for any $u_0 \in S^+$, there exists $\alpha_0 = \Delta(\rho_0, u_0) \in PR(\Sigma)_{u_0,v_0}$ such that

$$(\forall a \in A^{u_0})\ (\ [\![\alpha_0]\!]_A(a) = \rho_0^A\ ) \tag{24}$$

and,

$$(\forall a \in A^{u_0})\ (\ \lambda_P(\alpha_0)(a) = \rho_0^P\ ) \tag{25}$$

We now show the Lemma holds for ρ according to the following two possible cases:

(ii) *Vectorisation.* Suppose $\rho \in T(\Sigma)_v$ but $|v| \neq 1$. Then $\rho = <\rho_1,...,\rho_n>$ for some $\rho_i \in T(\Sigma)_{v_i}$ for $i = 1,...,n = |v|$.

For any $u \in S^+$ define $\alpha = \Delta(\rho,u)$ by $\alpha = <\Delta(\rho_1,u),...,\Delta(\rho_n,u)>$. Now, since each ρ_i is of less structural complexity than ρ, we have $\alpha_i = \Delta(\rho_i,u) \in PR(\Sigma)_{u,v_i}$, and

$$(\forall a \in A^u)\ (\ [\![\alpha_i]\!]_A(a) = \rho_i^A\ ) \tag{26}$$

by the induction hypothesis (24), for $i = 1,...,n$. Thus $\alpha \in PR(\Sigma)_{u,v}$ and for each $a \in A^u$ we have

$$[\![\alpha]\!]_A(a) = [\![<\alpha_1,...,\alpha_n>]\!]_A(a)$$
$$= ([\![\alpha_1]\!]_A(a),...,[\![\alpha_n]\!]_A(a))$$
$$= (\rho_1^A,...,\rho_n^A)$$

(by (26))

$$= <\rho_1,...,\rho_n>^A$$
$$= \rho^A$$

(by definition of ρ) as required.

To show (23), again since each ρ_i is of less structural complexity than ρ, we have

$$(\forall a \in A^u)\ (\ \lambda_P(\alpha_i)(a) = \rho_i^P\ ) \tag{27}$$

by the induction hypothesis (25), for $i = 1,...,n$. Thus, for each $a \in A^u$,

$$\lambda_P(\alpha)(a) = \lambda_P(<\alpha_1,...,\alpha_n>)(a)\ )$$
$$= \max_{1 \le i \le n}\{\ \lambda_P(\alpha_i)(a)\ \}$$
$$= \max_{1 \le i \le n}\{\ \rho_i^P\ \}$$

(by (27))

$$= \langle \rho_1, \ldots, \rho_n \rangle^P$$
$$= \rho^P$$

(by definition of ρ) as required.

(iii) *Composition.* Suppose $\rho \in T(\Sigma)_v$ and $|v| = 1$ but $\rho \notin \Sigma$. Then $v = s$ for some $s \in S$, and $\rho = \sigma \circ \rho_0$ where for some $w \in S^+$, $\sigma \in \Sigma_{w,s}$ and $\rho_0 \in T(\Sigma)_w$.

For any $u \in S^+$ define $\alpha = \Delta(\rho, u)$ by $\alpha = \sigma \circ \Delta(\rho_0, u)$. Now, since ρ_0 is of less structural complexity than ρ we have $\alpha_0 = \Delta(\rho_0, u) \in PR(\Sigma)_{u,w}$, and

$$(\forall a \in A^u) \ ([\![\alpha_0]\!]_A(a) = \rho_0^A) \tag{28}$$

by the induction hypothesis (24). Thus $\alpha \in PR(\Sigma)_{u,v}$, and for each $a \in A^u$ we have

$$[\![\alpha]\!]_A(a) = [\![\sigma \circ \alpha_0]\!]_A(a)$$
$$= [\![\sigma]\!]_A([\![\alpha_0]\!]_A(a))$$
$$= [\![\sigma]\!]_A(\rho_0^A)$$

(by (28))

$$= \sigma^A(\rho_0^A)$$
$$= (\sigma \circ \rho_0)^A$$
$$= \rho^A$$

(by definition of ρ) as required.

To show (23), again since ρ_0 is of less structural complexity than ρ, we have

$$(\forall a \in A^u) \ (\lambda_P(\alpha_0)(a) = \rho_0^P) \tag{29}$$

by the induction hypothesis (25). Thus, for each $a \in A^u$,

$$\lambda_P(\alpha)(a) = \lambda_P(\sigma \circ \alpha_0)(a)$$
$$= \lambda_P(\alpha_0)(a) + \lambda_P(\sigma)([\![\alpha_0]\!]_A(a))$$
$$= \rho_0^P + \lambda_P(\sigma)([\![\alpha_0]\!]_A(a))$$

(by (29))

$$= \rho_0^P + \lambda_P(\sigma)(\rho_0^A)$$

(by (28))

$$= \rho_0^P + \sigma^P(\rho_0^A)$$
$$= (\sigma \circ \rho_0)^P$$
$$= \rho^P$$

(by definition of ρ) again as required. □

8.3.8 Lemma. *Let A_1 be a Σ_1-structure for S_1-sorted signature Σ_1, and let L_2 be a PR system. Suppose A_1 is A_2-coded by γ via $w = w_\gamma$ and A_2-generated by $Gn = <\rho_c : c \in \Sigma_1>$. Then, for each $s \in S_1$ and for each $c \in (\Sigma_1)_{\lambda,s}$, there exists $\alpha_{c,v} \in \mathrm{PR}(\Sigma_2)_{w(v),w(s)}$ for any $v \in S_1^+$, such that*

$$(\forall a \in A_2^{w(v)})\ (\gamma(s)([\![\alpha_{c,v}]\!]_{A_2}(a)) = c^{A_1}) \tag{30}$$

and,

$$(\forall a \in A_2^{w(v)})\ (\lambda_{P_2}(\alpha_{c,v})(a) = \rho_c^{P_2}) \tag{31}$$

Proof. Take $\alpha_{c,v} = \Delta(\rho_c, w(v))$ where ρ_c generates c^{A_1} (where Δ is as in the proof of Lemma 8.3.7). Then $\alpha \in \mathrm{PR}(\Sigma)_{w(v),w(s)}$ since $\rho_c \in T(\Sigma_2)_{w(s)}$. Now, from Lemma 8.3.7 we know that for any $a \in A_2^{w(v)}$,

$$\begin{aligned}\gamma(s)([\![\alpha_{c,v}]\!]_{A_2}(a)) &= \gamma(s)(\rho_c^{A_2}) \\ &= c^{A_1}\end{aligned}$$

(since ρ_c generates c^{A_1}), and thus (30) holds as claimed.

To show that (31) holds, we have immediately from Lemma 8.3.7 that

$$(\forall a \in A_2^{w(v)})\ (\lambda_{P_2}(\alpha_{c,v})(a) = \rho_c^{P_2})$$ □

8.4 COMPILER THEOREMS.

In this section we will first state and prove our central theorem concerning the generation of correct and performance preserving compilers, and secondly we show our theory is compositional in the sense that the composition of two correct and performance preserving compilers is again a correct and performance preserving compiler; this result allows us to easily prove a theorem concerning the generation of a compiler which compiles high level schema down through a hierarchy of PR systems $L_1, \ldots, L_n$.

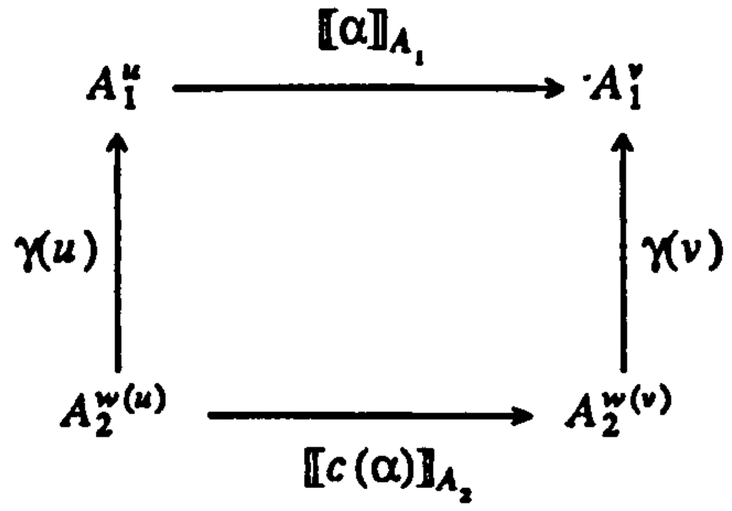
8.4.1 Implementation Theorem. *Let L_1 and L_2 be PR systems. Let I be an L_2-implementation of A_1. Then there exists an (L_1, L_2)-compiler c, correct with respect to γ_I. Furthermore, if I is d-adic, then c is d'-adic where d' satisfies*

$$\begin{aligned} d' &= 1 \quad \text{if } d = 0 \\ &\leq d \quad \text{if } d \geq 1 \end{aligned} \tag{32}$$

Proof. Let A_1 be A_2-coded by $\gamma = \gamma_I$ via w. Then to prove the theorem we must show that for each $\alpha \in \mathrm{PR}(\Sigma_1)$, there exists $c(\alpha) \in \mathrm{PR}(\Sigma_2)$ such that

(a) if $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u, v \in S_1^+$ then $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$,

(b) if $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u, v \in S_1^+$ then the following diagram commutes for every $a \in A_2^{w(u)}$:



that is,

$$(\forall a \in A_2^{w(u)})\ (\ \gamma(v)([\![c(\alpha)]\!]_{A_2}(a)) = [\![\alpha]\!]_{A_1}(\gamma(u)(a))\ ) \tag{33}$$

and,

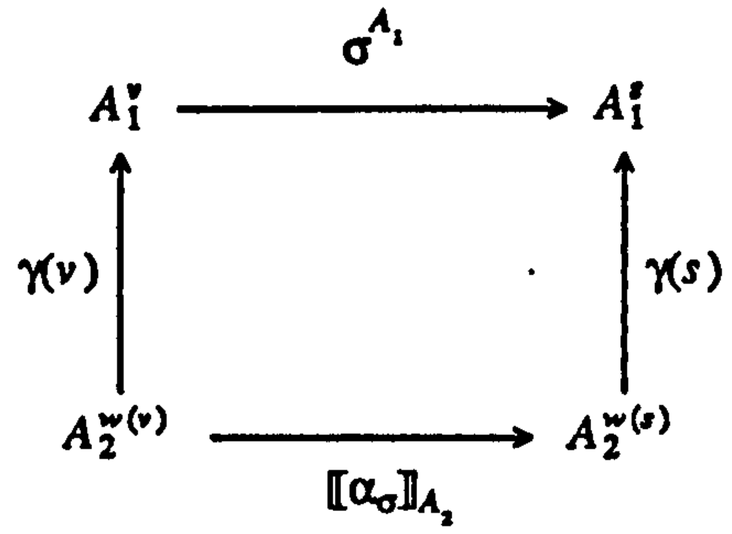
(c) there exists a d'-adic retiming Ψ_α, such that (32) holds of d', and if $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$, then

$$(\forall a \in A_2^{w(u)})\ (\ \lambda_{P_2}(c(\alpha))(a) \le \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(u)(a)))\ ) \tag{34}$$

In contrast to previous compiler theorems, the construction of c and the proof of its well-definedness (as a $S_1^\dagger \times S_1^\dagger$-indexed mapping) is not difficult and so we will define $c(\alpha)$ and establish its well-definedness simultaneously with the proofs of correctness and performance preservation (that is, d'-adicness).

Let us begin the proof by unraveling the rather compact hypothesis of the theorem:

Let $I = (\gamma, Gn, Tr)$ be an L_2-implementation of A_1. Then, from Definition 8.3.1, for each $v \in S_1^\dagger$ and $s \in S_1$, and for each $\sigma \in (\Sigma_1)_{v,s}$ there exists $\alpha_\sigma \in Tr$ with $\alpha_\sigma \in \mathrm{PR}(\Sigma_2)_{w(v),w(s)}$ (here $w = w_\gamma$) such that the following diagram commutes for every $a \in A_2^{w(v)}$:



That is,

$$(\forall a \in A_2^{w(v)})\ (\ \gamma(s)([\![\alpha_\sigma]\!]_{A_2}(a)) = \sigma^{A_1}(\gamma(v)(a))\ ) \tag{35}$$

Furthermore, if I is d-adic, there exists $\Psi : C_1 \longrightarrow C_2$ with $\Psi \in \mathbb{N}[x]$ and $deg\,\Psi = d$ such that

$$(\forall a \in A_2^{w(v)})\ (\ \lambda_{P_2}(\alpha_\sigma)(a) \le \Psi(\sigma^{P_1}(\gamma(v)(a)))\ ) \tag{36}$$

Now choose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u, v \in S_1^\dagger$. Using an inductive argument on the structural complexity of α which is uniform in u and v, we will first define $c(\alpha)$, establish that is well-defined, and then we prove correctness; then we construct a retiming Ψ_α which we show has a degree d' which satisfies (32) above, and then finally we show that c preserves the performance of α with respect to Ψ_α.

Note. Actually, the proof of part (c) the theorem relies on the fact that each Ψ_α is homogeneous; this is a further property that we establish in the basis steps of the proof and show is preserved in the induction steps.

Basis.

(i) *Constant Functions.* Suppose $\alpha = c^\nu$ for some $c \in (\Sigma_1)_{\lambda,s}$ for some $s \in S_1$ and some $\nu \in S_1^+$. Then $\alpha \in \mathrm{PR}(\Sigma_1)_{\nu,s}$.

Compilation. Define $c(\alpha)$ by $c(\alpha) = \alpha_{c,\nu}$ where $\alpha_{c,\nu}$ is as in the statement of Lemma 8.3.8.

Well-definedness. Here it is easy to see that $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{w(\nu),w(s)}$; this is given by Lemma 8.3.8.

Correctness. To see that c correctly compiles α, that is, that (33) holds for $c(\alpha)$, choose $a \in A_2^{w(\nu)}$ and calculate as follows:

$$\begin{aligned}\gamma(s)([\![c(\alpha)]\!]_{A_2}(a)) &= \gamma(s)([\![\alpha_{c,\nu}]\!]_{A_2}(a)) \\ &= c^{A_1}\end{aligned}$$

(by Lemma 8.3.8)

$$\begin{aligned}&= [\![c^\nu]\!]_{A_1}(\gamma(\nu)(a)) \\ &= [\![\alpha]\!]_{A_1}(\gamma(\nu)(a))\end{aligned}$$

Thus c correctly compiles constant functions.

Retiming. Define $\Psi_\alpha \in \mathbb{N}[x]$ by

$$\Psi_\alpha(x) = \left\lceil \frac{\rho_c^{P_2}}{c^{P_1}} \right\rceil . x$$

where $\lceil y \rceil$ denotes the smallest natural number greater than or equal to y. Then Ψ_α is homogeneous and $d' = deg\,\Psi_\alpha = 1$ satisfying (32). (If $d \neq 0$ then $d \geq 1$, and so $d' = 1 \leq d$.)

Performance Preservation. To see that c is d'-adic on α, that is, that (34) holds for $c(\alpha)$ and Ψ_α, choose $a \in A_2^{w(\nu)}$ and calculate as follows:

$$\begin{aligned}\lambda_{P_2}(c(\alpha))(a) &= \lambda_{P_2}(\alpha_{c,\nu})(a) \\ &= \rho_c^{P_2}\end{aligned}$$

(by Lemma 8.3.8)

$$\begin{aligned}&\leq \left\lceil \frac{\rho_c^{P_2}}{c^{P_1}} \right\rceil . c^{P_1} \\ &= \Psi_\alpha(c^{P_1})\end{aligned}$$

$$= \Psi_\alpha(\lambda_{P_1}(c^\nu)(\gamma(\nu)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(v)(a)))$$

Hence c is d'-adic on constant functions.

(ii) *Algebraic Operations.* Suppose $\alpha = \sigma$ for some $\sigma \in (\Sigma_1)_{v,s}$ for some $v \in S_1^+$ and $s \in S_1$. Then $\alpha \in \mathrm{PR}(\Sigma_1)_{v,s}$.

Compilation. Define $c(\alpha)$ by $c(\alpha) = \alpha_\sigma$ where $\alpha_\sigma \in I$ tracks σ.

Well-definedness. Here $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{w(v),w(s)}$ since I is an implementation (see Definition 8.3.6).

Correctness. To see that c correctly compiles α, that is, that (33) holds for $c(\alpha)$, choose $a \in A_2^{w(v)}$ and calculate as follows:

$$\gamma(s)([\![c(\alpha)]\!]_{A_2}(a)) = \gamma(s)([\![\alpha_\sigma]\!]_{A_2}(a))$$
$$= \sigma^{A_1}(\gamma(v)(a))$$

(by (35))

$$= [\![\sigma]\!]_{A_1}(\gamma(v)(a))$$
$$= [\![\alpha]\!]_{A_1}(\gamma(v)(a))$$

Thus c correctly compiles algebraic operations.

Retiming. Define Ψ_α by

$$\Psi_\alpha(x) = \Psi(x) + \Psi(0).(x-1)$$

Notice

$$\Psi_\alpha(0) = \Psi(0) - \Psi(0) = 0$$

thus Ψ_α is homogeneous (see Definition 8.2.21(a)), and it is easy to prove that

$$(\forall x \in \mathbb{N}) \ (\ \Psi_\alpha(x) \geq \Psi(x)\ ) \tag{37}$$

We must now prove that $d' = deg\,\Psi_\alpha$ satisfies (32) and that c is performance preserving. There are two cases to consider: (a) $deg\,\Psi_\alpha = 0$, and (b) $deg\,\Psi_\alpha > 0$. First, we write

$$\Psi(x) = a_0 + \ldots + a_d.x^d$$

Case (a). Suppose $deg\,\Psi = 0$. Then $\Psi(x) = a_0$, and so

$$\Psi_\alpha(x) = a_0 + a_0.(x-1) = a_0.x$$

Thus $d' = deg\,\Psi_\alpha = 1$ satisfying (32).

Performance Preservation. To see that c is d'-adic on α, that is, that (34) holds for $c(\alpha)$ and Ψ_α, choose $a \in A_2^{w(v)}$ and calculate as follows:

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(\alpha_\sigma)(a)$$
$$\leq \Psi(\sigma^{P_1}(\gamma(v)(a)))$$

(by (36))

$$= a_0$$

(def. Ψ)

$$= a_0 . 1$$

$$\leq a_0 . \sigma^{P_1}(\gamma(v)(a))$$

(since $\sigma^P(a) \geq 1$ for any σ, P, and a)

$$= \Psi_\alpha(\sigma^{P_1}(\gamma(v)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\sigma)(\gamma(v)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(v)(a)))$$

Hence c is d'-adic on algebraic operations in this case.

Case (b). Suppose $deg\,\Psi > 0$. Then

$$\Psi_\alpha(x) = \Psi(x) + \Psi(0) . (x-1)$$

$$= \Psi(x) + a_0 . (x-1)$$

$$= a_0 + \ldots + a_d . x^d + a_0 . x - a_0$$

$$= (a_0 + a_1) . x + \ldots + a_d . x^d$$

Thus $d' = deg\,\Psi_\alpha = deg\,\Psi$ satisfying (32).

Performance Preservation. To see that c is d'-adic on α, that is, that (34) holds for $c(\alpha)$ and Ψ_α, choose $a \in A_2^{w(v)}$ and calculate as follows:

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(\alpha_\sigma)(a)$$

$$\leq \Psi(\sigma^{P_1}(\gamma(v)(a)))$$

(by (36))

$$\leq \Psi_\alpha(\sigma^{P_1}(\gamma(v)(a)))$$

(by (37))

$$= \Psi_\alpha(\lambda_{P_1}(\sigma)(\gamma(v)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(v)(a)))$$

Hence c is d'-adic on algebraic operations in both case (a) and case (b).

(iii) *Projections.* Suppose $\alpha = U_i^v$ for some $v \in S_1^+$ and some i with $1 \leq i \leq n = |v|$. Then $\alpha \in PR(\Sigma_1)_{v,v_i}$.

Compilation. In order for some $c(\alpha) \in PR(\Sigma_2)$ to 'track' α, we must have $c(\alpha) \in PR(\Sigma_2)_{w(v),w(v_i)}$, that is, $c(\alpha) \in PR(\Sigma_2)_{w(v_1)\cdots w(v_n),w(v_i)}$. Further, given input $a = (a^1,\ldots,a^n) \in A_2^{w(v_1)} \times \cdots \times A_2^{w(v_n)}$, $c(\alpha)$ must 'project out' $a^i \in A_2^{w(v_i)}$. However, as we may only project out single values in PR (and not vectors of values), $c(\alpha)$ will be a vectorisation of projections:

Let $l_i = |w(v_i)|$ for $i = 1,...,n$, and let $l = \sum_{i=1}^{n} l_i$; then $l = |w(v)|$. Now, for $i = 1,...,n$, if we define m_i by

$$m_i = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=1}^{k=i} l_k & \text{if } i > 1 \end{cases}$$

then m_i is one less than the index of the first element of vector $a^i \in A_2^{w(v_i)}$ in $a = (a^1,...,a^i,...,a^n) \in A_2^{w(v)}$, That is, when $a \in A_2^{w(v)}$ is written as a list $a = (a_1,...,a_l)$ of elements as opposed to a list $a = (a^1,...,a^n)$ of vectors, we have

$$a^i = (a_{m_i+1},...,a_{m_i+l_i}) \tag{38}$$

We now define $c(\alpha)$ by $c(\alpha) = <\alpha_1^i,...,\alpha_{l_i}^i>$ where for $k = 1,...,l_i$ $\alpha_k^i = U_{m_i+k}^{w(v)}$.

Well-definedness. It should be clear from what we have just said that $\alpha_k^i \in PR(\Sigma_2)_{w(v),s_{i,k}}$ where $s_{i,k} = w(v)_{m_i+k}$ for $i = 1,...,n$ and for $k = 1,...,l_i$; $\alpha_k^i \in PR(\Sigma_2)_{w(v),w(v_i)_k}$ for $k = 1,...,l_i = |w(v_i)|$, and hence $c(\alpha) \in PR(\Sigma_2)_{w(v),w(v_i)}$ as required.

Correctness. To see that c correctly compiles α, that is, that (33) holds for $c(\alpha)$, choose $a \in A_2^{w(v)}$ and calculate as follows:

$$\begin{aligned} \gamma(v_i)([\![c(\alpha)]\!]_{A_2}(a)) &= \gamma(v_i)([\![<\alpha_1^i,...,\alpha_{l_i}^i>]\!]_{A_2}(a)) \\ &= \gamma(v_i)([\![\alpha_1^i]\!]_{A_2}(a),...,[\![\alpha_{l_i}^i]\!]_{A_2}(a)) \\ &= \gamma(v_i)(a_{m_i+1},...,a_{m_i+l_i}) \end{aligned}$$

(by definition of α_k^i)

$$= \gamma(v_i)(a^i)$$

(by (38))

$$= (\gamma(v_1)(a^1),...,\gamma(v_n)(a^n))_i$$

$$= (\gamma(v)(a))_i$$

(see Definition 8.2.8)

$$= [\![U_i^v]\!]_{A_1}(\gamma(v)(a))$$

$$= [\![\alpha]\!]_{A_1}(\gamma(v)(a))$$

Thus c correctly compiles projection functions.

Retiming. Define Ψ_α by $\Psi_\alpha(x) = x$. Then Ψ_α is homogeneous and $d' = deg\,\Psi_\alpha = 1$ satisfying (32).

Performance Preservation. To see that c is d'-adic on α, that is, that (34) holds for $c(\alpha)$ and Ψ_α, choose $a \in A_2^{w(v)}$ and calculate as follows:

$$\begin{aligned} \lambda_{P_2}(c(\alpha))(a) &= \lambda_{P_2}(<\alpha_1^i,...,\alpha_{l_i}^i>)(a) \\ &= \max_{1 \le k \le l_i} \{ \lambda_{P_2}(\alpha_k^i)(a) \} \end{aligned}$$

$$= \max_{1 \le k \le l_i} \{ \lambda_{P_2}(U_{m_i+k}^{w(v)})(a) \}$$

(by definition of α_k^i)

$$= \max_{1 \le k \le l_i} \{ 1 \}$$

$$= 1$$

$$= \Psi_\alpha(1)$$

$$= \Psi_\alpha(\lambda_{P_1}(U_i^n)(\gamma(v)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(v)(a)))$$

Thus c is d'-adic on projection functions.

Induction. Let $\alpha \in \mathrm{PR}(\Sigma_1)$ be some fixed (non-basis) scheme with the property that for every $\alpha_0 \in \mathrm{PR}(\Sigma_1)$ of less structural complexity than α, that

(I) if $\alpha_0 \in \mathrm{PR}(\Sigma_1)_{u_0,v_0}$ for some $u_0,v_0 \in S_1^+$ then $c(\alpha_0) \in \mathrm{PR}(\Sigma_2)_{w(u_0),w(v_0)}$ and

(II) if $\alpha_0 \in \mathrm{PR}(\Sigma_2)_{u_0,v_0}$ for some $u_0,v_0 \in S_1^+$, then

$$(\forall a \in A_2^{w(u_0)}) \ (\gamma(v_0)([\![c(\alpha_0)]\!]_{A_2}(a)) = [\![\alpha_0]\!]_{A_1}(\gamma(u_0)(a)))$$

and

(III) there exists some homogeneous d_0-adic retiming Ψ_{α_0} such that

$$d_0 = 1 \quad \text{if } d = 0$$
$$\le d \quad \text{if } d \ge 1$$

and if $\alpha_0 \in \mathrm{PR}(\Sigma_2)_{u_0,v_0}$ for some $u_0,v_0 \in S_1^+$, then

$$(\forall a \in A_2^{w(u_0)}) \ (\lambda_{P_2}(c(\alpha_0))(a) \le \Psi_{\alpha_0}(\lambda_{P_1}(\alpha_0)(\gamma(u_0)(a))))$$

Now suppose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u,v \in S_1^+$. According to the four possible cases (definition-by-cases, vectorisation, composition, and primitive recursion), we will now construct $c(\alpha)$, establish that it is well-defined member of $\mathrm{PR}(\Sigma_2)_{w(u),w(v)}$, and that it is correct; then we construct a homogeneous retiming Ψ_α such that $d' = deg\,\Psi_\alpha$ satisfies (32) and that c is d'-adic on α.

(iv) *Definition-by-cases.* Suppose $\alpha = \mathrm{DC}(\beta,\alpha_1,\alpha_2)$ for some $v \in S_1^+$ and some $\beta,\alpha_1,\alpha_2 \in \mathrm{PR}(\Sigma_1)$. Then $\beta \in \mathrm{PR}(\Sigma_1)_{u,B}$ and $\alpha_i \in \mathrm{PR}(\Sigma_1)_{u,v}$ for $i = 1,2$ since $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ by hypothesis.

Since β,α_1,α_2 are all of less structural complexity than α, by induction hypothesis (I) we have $c(\beta) \in \mathrm{PR}(\Sigma_2)_{w(u),w(B)}$ and $c(\alpha_i) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$ for $i = 1,2$. (Note that $w(B) = B$ since the coding γ is standard.) Furthermore, by induction hypothesis (II) we have

$$(\forall a \in A_2^{w(u)}) \ (\gamma(B)([\![c(\beta)]\!]_{A_2}(a)) = [\![\beta]\!]_{A_1}(\gamma(u)(a))) \tag{39}$$

and for $i = 1,2$,

$$(\forall a \in A_2^{w(u)}) \ (\gamma(v)([\![c(\alpha_i)]\!]_{A_2}(a)) = [\![\alpha_i]\!]_{A_1}(\gamma(u)(a))) \tag{40}$$

Also, by induction hypothesis (III), there exist homogeneous d_i-adic retimings $\Psi_o = \Psi_\beta$, and $\Psi_i = \Psi_{\alpha_i}$ for $i = 1,2$, such that for $i = 0,1,2$, $d_i = deg\,\Psi_i$ satisfies

$$\begin{array}{ll} d_i = 1 & \text{if } d = 0 \\ \leq d & \text{if } d \geq 1 \end{array} \tag{41}$$

and

$$(\forall a \in A_2^{w(u)})\ (\lambda_{P_2}(c(\beta))(a) \leq \Psi_i(\lambda_{P_1}(\beta)(\gamma(u)(a)))) \tag{42}$$

and for $i = 1,2$,

$$(\forall a \in A_2^{w(u)})\ (\lambda_{P_2}(c(\alpha_i))(a) \leq \Psi_i(\lambda_{P_1}(\alpha_i)(\gamma(u)(a)))) \tag{43}$$

Compilation. Define $c(\alpha)$ by $c(\alpha) = DC(c(\beta), c(\alpha_1), c(\alpha_2))$.

Well-definedness. We have already seen that $c(\beta) \in PR(\Sigma_2)_{w(u),B}$ and $c(\alpha_i) \in PR(\Sigma_2)_{w(u),w(v)}$ for $i = 1,2$, and thus $\alpha \in PR(\Sigma_2)_{w(u),w(v)}$ as required.

Correctness. First notice that for any $a \in A_2^{w(u)}$, if $[\![c(\beta)]\!]_{A_2}(a) = tt^{A_2}$ then

$$[\![\beta]\!]_{A_1}(\gamma(u)(a)) = \gamma(B)([\![c(\beta)]\!]_{A_2}(a))$$

(from (39))

$$= \gamma(B)(tt^{A_2})$$

(by hypothesis)

$$= tt^{A_1}$$

(since $\gamma(B)$ is a homomorphism, see Definition 8.2.2(i)).

Thus,

$$[\![c(\beta)]\!]_{A_2}(a) = tt^{A_2} \Rightarrow [\![\beta]\!]_{A_1}(\gamma(u)(a)) = tt^{A_1} \tag{44}$$

In a similar way it is easy to show that

$$[\![c(\beta)]\!]_{A_2}(a) = ff^{A_2} \Rightarrow [\![\beta]\!]_{A_1}(\gamma(u)(a)) = ff^{A_1} \tag{45}$$

We can now show c correctly compiles α as follows. Choose $a \in A_2^{w(u)}$. Then by definition of $c(\alpha)$ we have

$$[\![c(\alpha)]\!]_{A_2}(a) = [\![DC(c(\beta), c(\alpha_1), c(\alpha_2)]\!]_{A_2}(a)$$

$$= \begin{cases} [\![c(\alpha_1)]\!]_{A_2}(a) & \text{if } [\![c(\beta)]\!]_{A_2}(a) = tt^{A_2} \\ [\![c(\alpha_2)]\!]_{A_2}(a) & \text{if } [\![c(\beta)]\!]_{A_2}(a) = ff^{A_2} \end{cases}$$

$$= \begin{cases} [\![c(\alpha_1)]\!]_{A_2}(a) & \text{if } [\![\beta]\!]_{A_1}(\gamma(u)(a)) = tt^{A_1} \\ [\![c(\alpha_2)]\!]_{A_2}(a) & \text{if } [\![\beta]\!]_{A_1}(\gamma(u)(a)) = ff^{A_1} \end{cases}$$

(by (44) and (45)).

Thus,

$$\gamma(v)([\![c(\alpha)]\!]_{A_2}(a)) = \begin{cases} \gamma(v)([\![c(\alpha_1)]\!]_{A_2}(a)) & \text{if } [\![\beta]\!]_{A_1}(\gamma(u)(a)) = tt^{A_1} \\ \gamma(v)([\![c(\alpha_2)]\!]_{A_2}(a)) & \text{if } [\![\beta]\!]_{A_1}(\gamma(u)(a)) = ff^{A_1} \end{cases}$$

$$= \begin{cases} [\![\alpha_1]\!]_{A_1}(a) & \text{if } [\![\beta]\!]_{A_1}(a) = tt^{A_1} \\ [\![\alpha_2]\!]_{A_1}(a) & \text{if } [\![\beta]\!]_{A_1}(a) = ff^{A_1} \end{cases}$$

(by (40) with $i=1$ and $i=2$)

$$= [\![DC(\beta,\alpha_1,\alpha_2)]\!]_{A_2}(\gamma(u)(a))$$

$$= [\![\alpha]\!]_{A_1}(\gamma(u)(a))$$

Thus c correctly compiles α as claimed.

Retiming. Define Ψ_α by

$$\Psi_\alpha = \Psi_0 + \max(\Psi_1, \Psi_2)$$

To show that $d' = deg\,\Psi_\alpha$ satisfies (32), first recall from Definitions 8.2.21 that

$$d' = deg\,(\Psi_0 + \max(\Psi_1,\Psi_2)) = \max\{ deg\,\Psi_0, \max\{ deg\,\Psi_1, deg\,\Psi_2 \} \}$$
$$= \max\{ d_0, \max\{ d_1, d_2 \} \} = \max\{ d_0, d_1, d_2 \}$$

If $d = deg\,\Psi = 0$ then $d_0 = d_1 = d_2 = 1$ by (41), and thus $d' = 1$ as required. Alternatively, if $d \neq 0$, then $d_i \leq d$ for $i = 0,1,2$ (again by (41)), and so $\max\{ d_0, d_1, d_2 \} \leq d$, again as required.

Performance Preservation. Choose $a \in A_2^{w(u)}$. Then by definition of $c(\alpha)$ we have

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(DC(\beta,\alpha_1,\alpha_2))(a)$$

$$= \lambda_{P_2}(c(\beta))(a) + \begin{cases} \lambda_{P_2}(c(\alpha_1))(a) & \text{if } [\![c(\beta)]\!]_{A_2}(a) = tt^{A_2} \\ \lambda_{P_2}(c(\alpha_2))(a) & \text{if } [\![c(\beta)]\!]_{A_2}(a) = ff^{A_2} \end{cases} \quad (46)$$

To show that c preserves the performance of α with respect to Ψ_α, first suppose $[\![c(\beta)]\!]_{A_2}(a) = tt^{A_2}$. Then from (46) we have

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(c(\beta))(a) + \lambda_{P_2}(c(\alpha_1))(a)$$
$$\leq \Psi_0(\lambda_{P_1}(\beta)(\gamma(u)(a))) + \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))$$

(by (42) and (43) with $i=1$)

$$\leq \Psi_0(\lambda_{P_1}(\beta)(\gamma(u)(a))) + \max(\Psi_1,\Psi_2)(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))$$

(by Lemma 8.2.24(d))

$$\leq (\Psi_0 + \max(\Psi_1,\Psi_2))(\lambda_{P_1}(\beta)(\gamma(u)(a)) + \lambda_{P_1}(\alpha_1)(\gamma(u)(a)))$$

(by Lemma 8.2.22)

$$= \Psi_\alpha(\lambda_{P_1}(\beta)(\gamma(u)(a)) + \lambda_{P_1}(\alpha_1)(\gamma(u)(a))) \quad (47)$$

However, since $[\![c(\beta)]\!]_{A_2}(a) = tt^{A_2}$, we have $[\![\beta]\!]_{A_1}(\gamma(u)(a)) = tt^{A_1}$ by (44), and so

$$\lambda_{P_1}(\alpha)(\gamma(u)(a)) = \lambda_{P_1}(\beta)(\gamma(u)(a)) + \lambda_{P_1}(\alpha_1)(\gamma(u)(a)) \tag{48}$$

Thus from (47) we have

$$\begin{aligned}\lambda_{P_2}(c(\alpha))(a) &\leq \Psi_\alpha(\lambda_{P_1}(\beta)(\gamma(u)(a)) + \lambda_{P_1}(\alpha_1)(\gamma(u)(a))) \\ &= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(u)(a)))\end{aligned}$$

(from (48)).

Thus c preserves the performance of α when $[\![c(\beta)]\!]_{A_2}(a) = tt^{A_2}$. It is equally easy to prove that c preserves the performance of α when $[\![c(\beta)]\!]_{A_2}(a) = ff^{A_2}$; this we leave as an exercise.

(v) *Vectorisation.* Suppose $\alpha = <\alpha_1,\ldots,\alpha_n>$ for some $\alpha_i \in \mathrm{PR}(\Sigma_1)$ for $i = 1,\ldots,n \geq 1$. Then $\alpha_i \in \mathrm{PR}(\Sigma_1)_{u,v_i}$ for $i = 1,\ldots,n = |v|$, since by hypothesis $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$.

Since each α_i is of less structural complexity than α, by induction hypothesis (I) we have $c(\alpha_i) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v_i)}$ for $i = 1,\ldots,n$. Furthermore, by induction hypothesis (II) we have

$$(\forall a \in A_2^{w(u)})\ (\ \gamma(v_i)([\![c(\alpha_i)]\!]_{A_2}(a)) = [\![\alpha_i]\!]_{A_1}(\gamma(u)(a))\ ) \tag{49}$$

for $i = 1,\ldots,n$.

Also, by induction hypothesis (III), there exists a homogeneous d_i-adic retiming Ψ_i such that

$$\begin{aligned} d_i &= 1 \quad \text{if } d = 0 \\ &\leq d \quad \text{if } d \geq 1 \end{aligned} \tag{50}$$

and

$$(\forall a \in A_2^{w(u)})\ (\ \lambda_{P_2}(c(\alpha_i))(a) \leq \Psi_i(\lambda_{P_1}(\alpha_i)(\gamma(u)(a)))\ ) \tag{51}$$

for $i = 1,\ldots,n$.

Compilation. Intuitively, the required scheme $c(\alpha)$ is essentially the vectorisation of the schema $c(\alpha_i)$; however, since we may only vectorise single-valued schema in PR, we must first project out the coordinates of each $c(\alpha_i)$ before vectorising.

Let $l_i = |w(v_i)|$ for $i = 1,\ldots,n$ then we define $c(\alpha)$ by

$$c(\alpha) = <\alpha_1^1,\ldots,\alpha_{l_1}^1,\ldots,\alpha_1^n,\ldots,\alpha_{l_n}^n>$$

where for $i = 1,\ldots,n$ and $j = 1,\ldots l_i$,

$$\alpha_j^i = U_j^{w(v_i)} \circ c(\alpha_i)$$

Well-definedness. Clearly, since $c(\alpha_i) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v_i)}$ for $i = 1,\ldots,n$, we have $\alpha_j^i \in \mathrm{PR}(\Sigma_2)_{w(u),s_{ij}}$ where $s_{ij} = w(v_i)_j$ for $i = 1,\ldots,n$ and $j = 1,\ldots,l_i$; thus $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$ as required.

Correctness. First notice that for each $a \in A_2^{w(u)}$,

$$\begin{aligned}[\![\alpha_j^i]\!]_{A_2}(a) &= [\![U_j^{w(v_i)} \circ c(\alpha_i)]\!]_{A_2}(a) \\ &= [\![U_j^{w(v_i)}]\!]_{A_2}([\![c(\alpha_i)]\!]_{A_2}(a)) \\ &= ([\![c(\alpha_i)]\!]_{A_2}(a))_j \end{aligned} \tag{52}$$

Thus, for each $a \in A_2^{w(u)}$ we have

$$\gamma(v)(\llbracket c(\alpha)\rrbracket_{A_2}(a)) = \gamma(v)(\llbracket <\alpha_1^1,\ldots,\alpha_j^i,\ldots,\alpha_{l_n}^n> \rrbracket_{A_2}(a))$$
$$= \gamma(v)(\llbracket \alpha_1^1 \rrbracket_{A_2}(a),\ldots,\llbracket \alpha_j^i \rrbracket_{A_2}(a),\ldots,\llbracket \alpha_{l_n}^n \rrbracket_{A_2}(a))$$
$$= \gamma(v)((\llbracket c(\alpha_1)\rrbracket_{A_2}(a))_1,\ldots,(\llbracket c(\alpha_i)\rrbracket_{A_2}(a))_j,\ldots,(\llbracket c(\alpha_n)\rrbracket_{A_2}(a))_{l_n})$$

(by (52))

$$= \gamma(v)(\llbracket c(\alpha_1)\rrbracket_{A_2}(a),\ldots,\llbracket c(\alpha_i)\rrbracket_{A_2}(a),\ldots,\llbracket c(\alpha_n)\rrbracket_{A_2}(a))$$
$$= (\gamma(v_1)(\llbracket c(\alpha_1)\rrbracket_{A_2}(a)),\ldots,\gamma(v_i)(\llbracket c(\alpha_i)\rrbracket_{A_2}(a)),\ldots,\gamma(v_n)(\llbracket c(\alpha_n)\rrbracket_{A_2}(a)))$$

(see Definition 8.2.8)

$$= (\llbracket \alpha_1 \rrbracket_{A_1}(\gamma(u)(a)),\ldots,\llbracket \alpha_i \rrbracket_{A_1}(\gamma(u)(a)),\ldots,\llbracket \alpha_n \rrbracket_{A_1}(\gamma(u)(a)))$$

(by (49))

$$= \llbracket <\alpha_1,\ldots,\alpha_n> \rrbracket_{A_1}(\gamma(u)(a))$$
$$= \llbracket \alpha \rrbracket_{A_1}(\gamma(u)(a))$$

Thus c correctly compiles vectorisations.

Retiming. Define Ψ_α by

$$\Psi_\alpha(x) = x + \max_{1\le i\le n}(\Psi_i)(x)$$

Then Ψ_α is homogeneous. Furthermore,

$$d' = deg\,\Psi_\alpha = \max\{\,1,\max\{\,d_1,\ldots,d_n\,\}\,\} = \max\{\,d_1,\ldots,d_n\,\}$$

(The last step follows since $\Psi_1,\ldots,\Psi_n$ are homogeneous and so $d_i = deg\,\Psi_i > 0$ for $i = 1,\ldots,n$.) Now, if $d = deg\,\Psi = 0$ then $d_i = 1$ for $i = 1,\ldots,n$ by (50), and so $d' = \max\{\,1,\ldots,1\,\} = 1$ satisfying (32). Alternatively, if $d \neq 0$ then $d_i \le d$ for $i = 1,\ldots,n$ (again by (50)) and so $d' = \max\{\,d_1,\ldots,d_n\,\} \le d$, again, as required.

Performance Preservation. For each $a \in A_2^{w(u)}$ we have

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(<\alpha_1^1,\ldots,\alpha_j^i,\ldots,\alpha_{l_n}^n>)(a)$$
$$= \max_{1\le j\le l_i,\,1\le i\le n}\{\,\lambda_{P_2}(\alpha_j^i)(a)\,\}$$
$$= \max_{1\le j\le l_i,\,1\le i\le n}\{\,\lambda_{P_2}(U_j^{w(v_i)}\circ c(\alpha_i))(a)\,\}$$

(by definition of α_j^i for $i = 1,\ldots,n$ and for $j = 1,\ldots,l_i$)

$$= \max_{1\le j\le l_i,\,1\le i\le n}\{\,1+\lambda_{P_2}(c(\alpha_i))(a)\,\}$$
$$= \max_{1\le i\le n}\{\,1+\lambda_{P_2}(c(\alpha_i))(a)\,\}$$
$$\le 1+\max_{1\le i\le n}\{\,\Psi_i(\lambda_{P_1}(\alpha_i)(\gamma(u)(a)))\,\}$$

(by (51) with $i = 1,\ldots,n$)

$$\le 1+\max_{1\le i\le n}(\Psi_i)(\max_{1\le k\le n}\{\,\lambda_{P_1}(\alpha_k)(\gamma(u)(a))\,\})$$

(by Lemma 8.2.24(c) and induction on n)

$$\leq \max_{1\leq k\leq n}\{\ \lambda_{P_1}(\alpha_k)(\gamma(u)(a))\ \} + \max_{1\leq i\leq n}(\Psi_i)(\max_{1\leq k\leq n}\{\ \lambda_{P_1}(\alpha_k)(\gamma(u)(a))\ \})$$

(since $\lambda_P(\alpha)(a)\geq 1$ for any P, α, and a)

$$= \Psi_\alpha(\max_{1\leq k\leq n}\{\ \lambda_{P_1}(\alpha_k)(\gamma(u)(a))\ \})$$

$$= \Psi_\alpha(\lambda_{P_1}(<\alpha_1,\ldots,\alpha_n>)(\gamma(u)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(u)(a)))$$

Thus c is d'-adic on vectorisations.

(vi) *Composition.* Suppose $\alpha=\alpha_2\circ\alpha_1$ for some α_1 and $\alpha_2\in \mathrm{PR}(\Sigma_1)$. Then $\alpha_1\in \mathrm{PR}(\Sigma_1)_{u,v_0}$ and $\alpha_2\in \mathrm{PR}(\Sigma_1)_{v_0,v}$ for some $v_0\in S_1^+$, since $\alpha\in \mathrm{PR}(\Sigma_1)_{u,v}$ by hypothesis.

Since α_1 and α_2 are of less structural complexity than α, by induction hypothesis (I) $c(\alpha_1)\in \mathrm{PR}(\Sigma_2)_{w(u),w(v_0)}$ and $c(\alpha_2)\in \mathrm{PR}(\Sigma_2)_{w(v_0),w(v)}$. Furthermore, by induction hypothesis (II) we have

$$(\forall a\in A_2^{w(u)})\ (\ \gamma(v_0)([\![c(\alpha_1)]\!]_{A_2}(a)) = [\![\alpha_1]\!]_{A_1}(\gamma(u)(a))\ ) \tag{53}$$

and

$$(\forall a_0\in A_2^{w(v_0)})\ (\ \gamma(v)([\![c(\alpha_2)]\!]_{A_2}(a_0)) = [\![\alpha_2]\!]_{A_1}(\gamma(v_0)(a_0))\ ) \tag{54}$$

Also, by induction hypothesis (III), for $i=1,2$ there exist homogeneous d_i-adic retimings $\Psi_i=\Psi_{\alpha_i}$ such that

$$\begin{aligned} d_i &= 1 \quad \text{if } d=0\\ &\leq d \quad \text{if } d\geq 1 \end{aligned} \tag{55}$$

and

$$(\forall a\in A_2^{w(u)})\ (\ \lambda_{P_2}(c(\alpha_1)(a)) \leq \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))\ ) \tag{56}$$

and

$$(\forall a_0\in A_2^{w(v_0)})\ (\ \lambda_{P_2}(c(\alpha_2)(a_0)) \leq \Psi_2(\lambda_{P_1}(\alpha_2)(\gamma(v_0)(a_0)))\ ) \tag{57}$$

Compilation. Define $c(\alpha)$ by $c(\alpha)=c(\alpha_2)\circ c(\alpha_1)$.

Well-definedness. Clearly, since $c(\alpha_1)\in \mathrm{PR}(\Sigma_2)_{w(u),w(v_0)}$ and $c(\alpha_2)\in \mathrm{PR}(\Sigma_2)_{w(v_0),w(v)}$, we have $c(\alpha)\in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$ as required.

Correctness. To see that c correctly compiles α, that is, that (33) holds for $c(\alpha)$, choose $a\in A_2^{w(u)}$ and calculate as follows:

$$\begin{aligned} \gamma(v)([\![c(\alpha)]\!]_{A_2}(a)) &= \gamma(v)([\![c(\alpha_2)\circ c(\alpha_1)]\!]_{A_2}(a))\\ &= \gamma(v)([\![c(\alpha_2)]\!]_{A_2}([\![c(\alpha_1)]\!]_{A_2}(a)))\\ &= [\![\alpha_2]\!]_{A_1}(\gamma(v_0)([\![c(\alpha_1)]\!]_{A_2}(a))) \end{aligned}$$

(by (54))

$$= [\![\alpha_2]\!]_{A_1}([\![\alpha_1]\!]_{A_1}(\gamma(u)(a)))$$

(by (53))

$$= [\![\alpha_2 \circ \alpha_1]\!]_{A_1}(\gamma(u)(a))$$
$$= [\![\alpha]\!]_{A_1}(\gamma(u)(a))$$

Thus c correctly compiles compositions.

Retiming. Define Ψ_α by

$$\Psi_\alpha(x) = (\Psi_1 + \Psi_2)(x)$$

Thus Ψ_α is homogeneous. Furthermore,

$$d' = deg\,\Psi_\alpha = deg\,(\Psi_1 + \Psi_2) = \max\{deg\,\Psi_1, deg\,\Psi_2\} = \max\{d_1, d_2\}$$

Thus, if $d = deg\,\Psi = 0$, then $d_1 = d_2 = 1$ by (55), and so $d' = \max\{1,1\} = 1$ as required. Alternatively, if $d \neq 0$ then $d_i \leq d$ by (55) and so $d' = \max\{d_1, d_2\} \leq d$, again as required.

Performance Preservation. To see that c is d'-adic on α, that is, that (34) holds for $c(\alpha)$ and Ψ_α, choose $a \in A_2^{w(u)}$ and calculate as follows:

$$\lambda_{P_2}(c(\alpha))(a) = \lambda_{P_2}(c(\alpha_2) \circ c(\alpha_1))(a)$$
$$= \lambda_{P_2}(c(\alpha_1))(a) + \lambda_{P_2}(c(\alpha_2))([\![c(\alpha_1)]\!]_{A_2}(a))$$
$$\leq \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a))) + \Psi_2(\lambda_{P_1}(\alpha_2)(\gamma(v_0)([\![c(\alpha_1)]\!]_{A_2}(a))))$$

(by (56),(57))

$$= \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)) + \Psi_2(\lambda_{P_1}(\alpha_2)([\![\alpha_1]\!]_{A_1}(\gamma(u)(a)))$$

(by (53))

$$\leq \Psi_\alpha(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)) + \lambda_{P_1}(\alpha_2)([\![\alpha_1]\!]_{A_1}(\gamma(u)(a))))$$

(by Lemma 8.2.22)

$$= \Psi_\alpha(\lambda_{P_1}(\alpha_2 \circ \alpha_1)(\gamma(u)(a)))$$
$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(u)(a)))$$

Thus c is d'-adic on compositions.

(vii) *Primitive Recursion.* Suppose $\alpha = *(\alpha_1, \alpha_2)$ for some $\alpha_1, \alpha_2 \in PR(\Sigma_1)$. In this final case $\alpha \in PR(\Sigma_1)_{Tu,v}$ for some $u, v \in S_1^+$ so we must have $\alpha_1 \in PR(\Sigma_1)_{u,v}$ and $\alpha_2 \in PR(\Sigma_1)_{Tuv,v}$.

Since α_1 and α_2 are of less structural complexity than α, by induction hypothesis (I), $c(\alpha_1) \in PR(\Sigma_2)_{w(u),w(v)}$ and $c(\alpha_2) \in PR(\Sigma_2)_{Tw(u)w(v),w(v)}$ (notice $w(T) = T$ since the coding is standard). Furthermore, by induction hypothesis (II) we have

$$(\forall a \in A_2^{w(u)})\ (\gamma(v)([\![c(\alpha_1)]\!]_{A_2}(a)) = [\![\alpha_1]\!]_{A_1}(\gamma(u)(a)))\quad (58)$$

and,

$$(\forall t \in T)(\forall a \in A_2^{w(u)})(\forall a' \in A_2^{w(v)})\ (\gamma(v)([\![c(\alpha_2)]\!]_{A_2}(t,a,a')) = [\![\alpha_2]\!]_{A_1}(\gamma(Tuv)(t,a,a')))\quad (59)$$

Also, by induction hypothesis (III), for $i = 1,2$ there exist homogeneous d_i-adic retimings $\Psi_i = \Psi_{\alpha_i}$ such that

$$\begin{aligned} d_i &= 1 \quad \text{if } d = 0 \\ &\leq d \quad \text{if } d \geq 1 \end{aligned} \tag{60}$$

and,

$$(\forall a \in A_2^{w(u)})\ (\lambda_{P_2}(c(\alpha_1)) \leq \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))) \tag{61}$$

and

$$(\forall t \in T)(\forall a \in A_2^{w(u)})(\forall a' \in A_2^{w(v)})\ (\lambda_{P_2}(c(\alpha_2))(a) \leq \Psi_2(\lambda_{P_1}(\alpha_2)(\gamma(Tuv)(t,a,a')))) \tag{62}$$

Compilation. Define $c(\alpha)$ by $c(\alpha) = *(c(\alpha_1), c(\alpha_2))$

Well-definedness. Clearly, by Definition 3.3.1(vii), since $c(\alpha_1) \in \mathrm{PR}(\Sigma_2)_{w(u),w(v)}$ and $c(\alpha_2) \in \mathrm{PR}(\Sigma_2)_{Tw(u)w(v),w(v)}$, we have $c(\alpha) \in \mathrm{PR}(\Sigma_2)_{Tw(u),w(v)} = \mathrm{PR}(\Sigma_2)_{w(Tu),w(v)}$ as required.

Correctness. We show for each $t \in T$,

$$(\forall a \in A_2^{w(u)})\ (\gamma(v)([\![c(\alpha)]\!]_{A_2}(t,a)) = [\![\alpha]\!]_{A_1}(\gamma(Tu)(t,a))) \tag{63}$$

by sub-induction on t.

Sub-Basis. Choose $a \in A_2^{w(u)}$. Then

$$\begin{aligned} \gamma(v)([\![c(\alpha)]\!]_{A_2}(0,a)) &= \gamma(v)([\![*(c(\alpha_1), c(\alpha_2))]\!]_{A_2})(0,a)) \\ &= \gamma(v)([\![c(\alpha_1)]\!]_{A_2}(a)) \\ &= [\![\alpha_1]\!]_{A_1}(\gamma(u)(a)) \end{aligned}$$

(by (58))

$$\begin{aligned} &= [\![*(\alpha_1, \alpha_2)]\!]_{A_1}(0,\gamma(u)(a)) \\ &= [\![\alpha]\!]_{A_1}(0,\gamma(u)(a)) \\ &= [\![\alpha]\!]_{A_1}(\gamma(T)(0),\gamma(u)(a)) \end{aligned}$$

(since γ is standard)

$$= [\![\alpha]\!]_{A_1}(\gamma(Tu)(0,a))$$

(see Definition 8.2.8).

Sub-Induction. Assume for some fixed $k \in \mathbf{N}$, that for $t = 0,\ldots,k$,

$$(\forall a \in A_2^{w(u)})\ (\gamma(v)([\![\alpha]\!]_{A_2}(t,a)) = [\![\alpha]\!]_{A_1}(\gamma(Tu)(t,a))) \tag{64}$$

We now show (63) holds for $t = k+1$.

For each $a \in A_2^{w(u)}$ we have

$$\begin{aligned} \gamma(v)([\![c(\alpha)]\!]_{A_2}(k+1,a)) &= \gamma(v)([\![*(c(\alpha_1), c(\alpha_2))]\!]_{A_2}(k+1,a)) \\ &= \gamma(v)([\![c(\alpha_2)]\!]_{A_2}(k,a,[\![\alpha]\!]_{A_2}(k,a))) \\ &= [\![\alpha_2]\!]_{A_1}(\gamma(Tuv)(k,a,[\![\alpha]\!]_{A_2}(k,a))) \end{aligned}$$

(by (59))

$$= [\![\alpha_2]\!]_{A_1}(\gamma(T)(k),\gamma(u)(a),\gamma(v)([\![\alpha]\!]_{A_2}(k,a)))$$

(see Definition 8.2.8)

$$= [\![\alpha_2]\!]_{A_1}(\gamma(T)(k),\gamma(u)(a),[\![\alpha]\!]_{A_1}(\gamma(Tu)(k,a)))$$

(by (64))

$$= [\![\alpha_2]\!]_{A_1}(\gamma(T)(k),\gamma(u)(a),[\![\alpha]\!]_{A_1}(\gamma(T)(k),\gamma(u)(a)))$$

(see Definition 8.2.8)

$$= [\![\alpha_2]\!]_{A_1}(k,\gamma(u)(a),[\![\alpha]\!]_{A_1}(k,\gamma(u)(a)))$$

(since γ is standard)

$$= [\![*(\alpha_1,\alpha_2)]\!]_{A_1}(k+1,\gamma(u)(a))$$

$$= [\![\alpha]\!]_{A_1}(k+1,\gamma(u)(a))$$

$$= [\![\alpha]\!]_{A_1}(\gamma(T)(k+1),\gamma(u)(a))$$

(since γ is standard)

$$= [\![\alpha]\!]_{A_1}(\gamma(Tu)(k+1,a))$$

(see Definition 8.2.8).

Thus (63) holds for each $t \in T$ and so c correctly compiles primitive recursions.

Retiming. Define Ψ_α by

$$\Psi_\alpha(x) = \max(\Psi_1,\Psi_2)(x)$$

Thus Ψ_α is homogeneous, and exactly as in case (vi) we can show (from (60)) that $d' = deg\,\Psi_\alpha$ satisfies (32).

Performance Preservation. We now show for each $t \in T$,

$$(\forall a \in A_2^{w(u)})\ (\ \lambda_{P_2}(c(\alpha))(t,a) \le \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(t,a)))\ ) \qquad (65)$$

by sub-induction on t.

Sub-Basis. For each $a \in A_2^{w(u)}$ we have

$$\lambda_{P_2}(c(\alpha))(0,a) = \lambda_{P_2}(*(c(\alpha_1),c(\alpha_2)))(0,a)$$

$$= \lambda_{P_2}(c(\alpha_1))(a)$$

$$\le \Psi_1(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))$$

(by (61))

$$\le \Psi_\alpha(\lambda_{P_1}(\alpha_1)(\gamma(u)(a)))$$

(by Lemma 8.2.24(d))

$$= \Psi_\alpha(\lambda_{P_1}(*(\alpha_1,\alpha_2))(0,\gamma(u)(a)))$$

$$= \Psi_\alpha(\lambda_{P_1}(*(\alpha_1,\alpha_2))(\gamma(T)(0),\gamma(u)(a)))$$

(since γ is standard)

$$= \Psi_\alpha(\lambda_{P_1}(*(\alpha_1,\alpha_2))(\gamma(Tu)(0,a)))$$

(see Definition 8.2.8)

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(0,a)))$$

Thus (65) holds for $t = 0$.

Sub-Induction. Assume for some fixed $k \in \mathbf{N}$ that for $t = 0,\ldots,k$,

$$(\forall a \in A_2^{w(u)})\ (\ \lambda_{P_2}(c(\alpha))(t,a) \leq \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(t,a)))\ ) \tag{66}$$

We now show (65) holds for $t = k+1$.

For each $a \in A_2^{w(u)}$ we have

$$\begin{aligned}
\lambda_{P_2}(c(\alpha))(k+1,a) &= \lambda_{P_2}(*(c(\alpha_1), c(\alpha_2)))(k+1,a) \\
&= \lambda_{P_2}(c(\alpha_2))(k,a,[\![c(\alpha)]\!]_{A_2}(k,a)) + \lambda_{P_2}(c(\alpha))(k,a) \\
&\leq \Psi_2(\lambda_{P_1}(\alpha_2)(\gamma(Tuv)(k,a,[\![c(\alpha)]\!]_{A_2}(k,a)))) \\
&\quad + \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(k,a)))
\end{aligned}$$

(by (62) and (66))

$$\begin{aligned}
&= \Psi_2(\lambda_{P_1}(\alpha_2)(k,\gamma(u)(a),[\![\alpha]\!]_{A_1}(k,\gamma(u)(a)))) \\
&\quad + \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(k,a)))
\end{aligned}$$

(by (63) and Definition 8.2.8)

$$\begin{aligned}
&\leq \Psi_\alpha(\lambda_{P_1}(\alpha_2)(k,\gamma(u)(a),[\![\alpha]\!]_{A_1}(k,\gamma(u)(a)))) \\
&\quad + \Psi_\alpha(\lambda_{P_1}(\alpha)(\gamma(Tu)(k,a)))
\end{aligned}$$

(by Lemma 8.2.24 parts (a) and (d))

$$\begin{aligned}
&\leq \Psi_\alpha(\ \lambda_{P_1}(\alpha_2)(\gamma(Tuv)(k,\gamma(u)(a),[\![\alpha]\!]_{A_1}(k,\gamma(u)(a))) \\
&\quad + \lambda_{P_1}(\alpha)(\gamma(Tu)(k,a))\ )
\end{aligned}$$

(by Lemma 8.2.22)

$$= \Psi_\alpha(\lambda_{P_1}(\alpha)([\![\alpha]\!]_{A_1}(\gamma(Tu)(k+1,a))))$$

since $\alpha = *(\alpha_1, \alpha_2)$ and $\gamma(T)(k+1) = \gamma(T)(k)+1$, completing the sub-induction step.

Thus (66) holds for $t = k+1$ and so (65) holds for all $t \in T$, that is, c is d'-adic on primitive recursions. □

We now prove that our formal methods are hierarchical in the sense that the composition two correct and performance preserving compilers is again a correct and performance preserving compiler; ultimately, this allows us to automatically compile down through a hierarchy of PR systems $L_1,\ldots,L_n$.

8.4.2 Composition Lemma. *Let L_i be a PR system for $i = 1,2,3$. Suppose A_{i+1} is A_i-coded by γ_i via w_i for $i = 1,2$, and further suppose c_i is an (L_i, L_{i+1})-d_i-adic compiler, correct with respect to γ_i for some $d_i \in \mathbf{N}$ for $i = 1,2$. Define $c : \mathrm{PR}(\Sigma_1) \longrightarrow \mathrm{PR}(\Sigma_3)$ by $c = c_2 \circ c_1$. Then c is an (L_1, L_3)-d-adic compiler, correct with respect to $\gamma = \gamma_1 \circ \gamma_2$, where $d = d_1 \cdot d_2$.*

Proof. We will first show c is well-defined, and then we show it is correct with respect to γ, and finally we show it is d-adic.

Well-definedness. Choose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u,v \in S_1^+$ and consider $c(\alpha)$. Now, $c(\alpha)=c_2(c_1(\alpha))$, and $c_1(\alpha) \in \mathrm{PR}(\Sigma_2)_{w_1(u),w_1(v)}$ since c_1 is well-defined and so $c_2(c_1(\alpha)) \in \mathrm{PR}(\Sigma_3)_{w_2(w_1(u)),w_2(w_1(v))}$ since c_2 is also well-defined. Thus, by definition of c and $w(=w_2 \circ w_1)$, $c(\alpha) \in \mathrm{PR}(\Sigma_3)_{w(u),w(v)}$, that is, c is an (L_1,L_3)-compiler.

Correctness. For $i=1,2$, since c_i is a correct compiler we have

$$(\forall a \in A_{i+1}^{w_i(u)}) \ (\gamma_i(v)([\![c_i(\alpha)]\!]_{A_{i+1}}(a)) = [\![\alpha]\!]_{A_i}(\gamma_i(u)(a))) \tag{67}$$

for every $\alpha \in \mathrm{PR}(\Sigma_i)_{w_i(u),w_i(v)}$ for each $u,v \in S_i^+$ for $i=1$ and $i=2$.

To see that c is correct, again choose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u,v \in S_1^+$ and consider $c(\alpha)$. For each $a \in A_3^{w(u)}$ we have

$$\begin{aligned}
\gamma(v)([\![c(\alpha)]\!]_{A_3}(a)) &= \gamma(v)([\![c_2(c_1(\alpha))]\!]_{A_3}(a)) \\
&= \gamma_1(v)(\gamma_2(w_1(v))([\![c_2(c_1(\alpha))]\!]_{A_3}(a))) \\
&\text{(see Notation 8.2.13)} \\
&= \gamma_1(v)([\![c_1(\alpha)]\!]_{A_2}(\gamma_2(w_1(u))(a))) \\
&\text{(by (67) with } i=2) \\
&= [\![\alpha]\!]_{A_1}(\gamma_1(u)(\gamma_2(w_1(u))(a))) \\
&\text{(by (67) with } i=1) \\
&= [\![\alpha]\!]_{A_1}(\gamma(v)(a)) \\
&\text{(see Notation 8.2.13)}
\end{aligned}$$

Thus c is correct with respect to γ as claimed.

d-adicness. Since c_i is d_i-adic for $i=1,2$, we know that for every $u,v \in S_i^+$ and for each $\alpha \in \mathrm{PR}(\Sigma_i)_{w_i(u),w_i(v)}$, there exists $\Psi_\alpha^{(i)} \in \mathbb{N}[x]$ with degree d_i, and

$$(\forall a \in A_{i+1}^{w_i(u)}) \ (\lambda_{P_{i+1}}(c_i(\alpha)(a)) \le \Psi_\alpha^{(i)}(\lambda_{P_i}(\alpha)(\gamma_i(u)(a)))) \tag{68}$$

To see that c is d-adic where $d=d_1 . d_2$, choose $\alpha \in \mathrm{PR}(\Sigma_1)_{u,v}$ for some $u,v \in S_1^+$. Now define $\Psi_\alpha \in \mathbb{N}[x]$ by

$$\Psi_\alpha = \Psi_{c_1(\alpha)}^{(2)} \circ \Psi_\alpha^{(1)} \tag{69}$$

Then $\Psi_\alpha \in \mathbb{N}[x]$ with $deg\, \Psi_\alpha = d_1 . d_2$ by Lemma 8.2.25. Furthermore, for each $a \in A_3^{w(u)}$,

$$\begin{aligned}
\lambda_{P_3}(c(\alpha))(a)) &= \lambda_{P_3}(c_2(c_1(\alpha)))(a) \\
&\le \Psi_{c_1(\alpha)}^{(2)}(\alpha)(\lambda_{P_2}(c_1(\alpha))(\gamma_2(w_1(u))(a))) \qquad (70) \\
&\text{(by (68) with } i=2).
\end{aligned}$$

Now, from (68) with $i=1$ we have

$$\lambda_{P_2}(c_1(\alpha))(\gamma_2(w_1(u))(a)) \le \Psi_\alpha^{(1)}(\lambda_{P_1}(\alpha)(\gamma_1(u)(\gamma_2(w_1(u))(a))))$$

Thus, since $\Psi_{c_1(\alpha)}^{(2)}$ is homogeneous, it is monotone by Lemma 8.2.23, and so

$$\Psi_{c_1(\alpha)}^{(2)}(\lambda_{P_2}(c_1(\alpha))(\gamma_2(w_1(u))(a))) \le \Psi_{c_1(\alpha)}^{(2)}(\Psi_\alpha^{(1)}(\lambda_{P_1}(\alpha)(\gamma_1(u)(\gamma_2(w_1(u))(a))))) \tag{71}$$

From (70) we have

$$\lambda_{P_3}(c(\alpha))(a)) \leq \Psi^{(2)}_{c_1(\alpha)}(\alpha)(\lambda_{P_2}(c_1(\alpha))(\gamma_2(w_1(u))(a)))$$
$$\leq \Psi^{(2)}_{c_1(\alpha)}(\Psi^{(1)}_{\alpha}(\lambda_{P_1}(\alpha)(\gamma_1(u)(\gamma_2(w_1(u))(a)))))$$

(by (71))

$$= (\Psi^{(2)}_{c_1(\alpha)} \circ \Psi^{(1)}_{\alpha})(\lambda_{P_1}(\alpha)(\gamma_1(u)(\gamma_2(w_1(u))(a))))$$
$$= \Psi_{\alpha}(\lambda_{P_1}(\alpha)(\gamma_1(u)(\gamma_2(w_1(u))(a))))$$

(by (69))

$$= \Psi_{\alpha}(\lambda_{P_1}(\alpha)(\gamma(u)(a)))$$

(see Lemma 8.2.12).

Thus c is d-adic as claimed. □

8.4.3 Hierarchy Theorem. *Let $L_1,\ldots,L_{n+1}$ be PR systems for some $n \geq 1$. Suppose that for $i = 1,\ldots,n$ there exists an L_{i+1}-d_i-adic implementation I_i of L_i for some $d_i \in \mathbb{N}$. Then there exists an $(L_1$-$L_{n+1})$-d-adic compiler c_n, correct with respect to $\gamma^{(n)}$ where $d = d^{(n)}$ satisfies:*

$$d \leq \times\{ d_i \in (d_1,\ldots,d_n) : d_i \neq 0 \}$$

and where $\gamma^{(n)} : S_1 \longrightarrow [A^+_{n+1} \longrightarrow A_1]$ is the A_{n+1}-coding of A_1 defined by:

$$\gamma^{(n)} = \gamma_1 \circ \ldots \circ \gamma_n$$

where for $i = 1,\ldots,n$, $\gamma_i = \gamma_{I_i}$.

Notes.

(i) For arbitrary finite $S \subseteq \mathbb{N}$, $\underset{s \in S}{\times} S$ is the number obtained by multiplying all the elements of S together; by convention $\underset{s \in S}{\times} S = 1$ when S is empty.

(ii) Throughout the following proof we will abbreviate w_{γ_i} by w_i.

(iii) Recall the definition of $\gamma_1 \circ \cdots \circ \gamma_n = \gamma^{(n)}$ from Notation 8.2.15; notice $w_{\gamma^{(n)}} = w^{(n)}$ is defined by $w^{(n)} = w_n \circ \cdots \circ w_1$.

(iv) Recall from Lemma 8.2.14 $\gamma^{(n)}$ *is* an A_{n+1}-coding of A_1; thus that c_n should be correct with respect to it makes sense.

Proof. By induction on n.

Basis. Suppose $n = 1$. We are given an L_2-d_1-adic implementation I_1 of L_1 for some $d_1 \in \mathbb{N}$. By the Implementation Theorem then, there exists an $(L_1$-$L_2)$-d_0-adic compiler c, correct with respect to γ_1 such that

$$\begin{aligned} d_0 &= 1 \quad \text{if } d_1 = 0 \\ &\leq d_1 \quad \text{if } d_1 \geq 1 \end{aligned} \tag{72}$$

To prove the theorem in this case, take $c_n = c$, then we only need to show

$$d \leq \times\{ d_i \in (d_1,\ldots,d_n) : d_i \neq 0 \}$$

where $d = d_0$; but this is obvious from (72) above and note (i), since $n = 1$ here.

Induction. Suppose for some fixed $k \in \mathbb{N}^+$ that for $n = 1,...,k$, if we are given PR systems $L_1,...,L_{n+1}$ together with, for $i = 1,...,n$, an L_{i+1}-d_i-adic implementation I_i of L_i for some $d_i \in \mathbb{N}$, then there exists an $(L_1$-$L_{n+1})$-d-adic compiler c_n, correct with respect to $\gamma^{(n)}$ where $d = d^{(n)}$ satisfies:

$$d \leq \times\{ d_i \in (d_1,...,d_n) : d_i \neq 0 \}$$

Now suppose we are given PR systems $L_1,...,L_{k+2}$ together with, for $i = 1,...,k+1$, an L_{i+1}-d_i-adic implementation I_i of L_i for some $d_i \in \mathbb{N}$.

Now, since I_{k+1} is an L_{k+2}-d_{k+1}-adic implementation of L_{k+1}, by the Implementation Theorem there exists an $(L_{k+1}$-$L_{k+2})$-d_o-adic compiler c, correct with respect to γ_{k+1} such that

$$\begin{aligned} d_o &= 1 && \text{if } d_{k+1} = 0 \\ &\leq d_{k+1} && \text{if } d_{k+1} \geq 1 \end{aligned} \tag{73}$$

Also, by the induction hypothesis there exists an $(L_1$-$L_{k+1})$-$d^{(k)}$-adic compiler c_k, correct with respect to $\gamma^{(k)}$ where

$$d^{(k)} \leq \times\{ d_i \in (d_1,...,d_k) : d_i \neq 0 \} \tag{74}$$

We will now construct c_{k+1} and show it to be correct with respect to $\gamma^{(k+1)}$, and $d^{(k+1)}$-adic:

Compilation. For each $\alpha \in \mathrm{PR}(\Sigma_1)$ define $c_{k+1}(\alpha)$ by

$$c_{k+1}(\alpha) = (c \circ c_k)(\alpha)$$

Then $c_{k+1} : \mathrm{PR}(\Sigma_1) \longrightarrow \mathrm{PR}(\Sigma_{k+2})$ is a correct (L_1, L_{k+2})-d-adic compiler by Lemma 8.4.2 where $d = d_o . d^{(k)}$.

To see c_{k+1} is $d^{(k+1)}$–adic, we already know c_{k+1} is d-adic where $d = d_o . d^{(k)}$, so we must show

$$d = d_o . d^{(k)} \leq d^{(k+1)} = \times\{ d_i \in (d_1,...,d_{k+1}) : d_i \neq 0 \} \tag{75}$$

Case (a). Suppose $d_{k+1} = 0$. Then

$$\begin{aligned} d &= d_o . d^{(k)} \\ &= 1 . d^{(k)} \end{aligned}$$

(by (73))

$$= \times\{ d_i \in (d_1,...,d_k) : d_i \neq 0 \}$$

(by (74))

$$= \times\{ d_i \in (d_1,...,d_{k+1}) : d_i \neq 0 \}$$

by the case hypothesis.

Case (b).

Suppose $d_{k+1} \geq 1$, then

$$\begin{aligned} d &= d_o . d^{(k)} \\ &\leq d_{k+1} . d^{(k)} \end{aligned}$$

(by (73))

$$= d_{k+1} . \times\{ d_i \in (d_1,...,d_k) : d_i \neq 0 \}$$

(by (74))

$$= \times\{ d_i \in (d_1,...,d_{k+1}) : d_i \neq 0 \}$$

as required. Thus (75) holds in all cases, and c_{k+1} is $d^{(k+1)}$-adic as claimed.

□

8.5 AN EXAMPLE.

In this section we will consider two stages in the top-down design of the FIR network of Section 5.1. In the first stage we will implement the modules of FIR by synchronous (sub-) networks that use modules which are more primitive than those of FIR, but which process the same kind of data. In the second stage we will implement the modules of the sub-networks by further synchronous networks over different, 'lower-level' data. Having explained and formally specified the two levels of implementation, we will show that the network implementations lead to an implementation in the formal sense of Definition 8.3.6, and that we can apply Hierarchy Theorem 8.4.3 to establish correctness of the final design.

Preliminaries. In the following sections we freely use certain nomenclature and notation that have not been seen for some time. In particular, the reader should recall:

(i) Definitions and notations relating to standard algebras A and stream algebras $\underline{A}$ (see Section 3.1.8).

(ii) The definition of $B = A|_{\Omega}$ from Definitions 3.1.4(v). In particular note that $A = \underline{A}|_{\Sigma}$ when Σ is the signature of A, and so $\sigma^{\underline{A}} = \sigma^{A}$ for each $\sigma \in \Sigma$. An easy corollary to this fact is that if B has signature Ω and $\Omega \supseteq \Sigma$, then $\alpha \in PR(\Sigma)_{u,v}$ implies $\alpha \in PR(\Omega)$ and $[\![\alpha]\!]_B = [\![\alpha]\!]_A$.

(iii) For a synchronous network N, the definition of, and the differences between: N's dynamic value function V_N, its static value function v_N, and its static output specification f_N (see Sections 2.4 and 3.4). Also recall the primitive recursiveness of V_N and f_N via the schema α_N and δ_N respectively (see Notation 3.4.5).

(iv) The definition of the 'implementation function' I and its PR formalisation ι from the discussion at the end of Section 8.1.2.

Finally, the reader should also recall the discussion at the end of Section 8.1.1 concerning the (isomorphic) clocks T and T' of a synchronous network N and a network system M (obtained from N) respectively. In the text below we will usually write '$t \in T$' and '$t' \in T'$' for emphasis, although this sometimes leads to 'mixed' notation as mentioned at the end of Section 8.1.1.

8.5.1 The Draft Algorithm.

Let us begin with the FIR network of Section 5.1. FIR was described as an n-module network over a ring R; for simplicity we will take $n = 3$ (see Figure 8.7), and for definiteness we take R to be the ring of integers $\mathbf{Z} = \{ \ldots,-2,-1,0,1,2,\ldots \}$. In this setting FIR is (semantically) formalised as follows:

Let A_1 be the algebra comprising the three carriers: FIR's clock T; the Booleans $\mathbb{B}$, and the integers $\mathbf{Z}$. Then A_1 is S_1-sorted where $S_1 = \{ \mathrm{T},\mathrm{B},\mathrm{Z} \}$ (say). In addition to standard constants and operations, A_1 has the three operations $\sigma_1^{A_1}$, $\sigma_2^{A_1}$, and $\sigma_3^{A_1}$ defined as follows (cf. Section 5.1.2):

$$\sigma_1^{A_1}: \mathbf{Z} \longrightarrow \mathbf{Z}$$

where

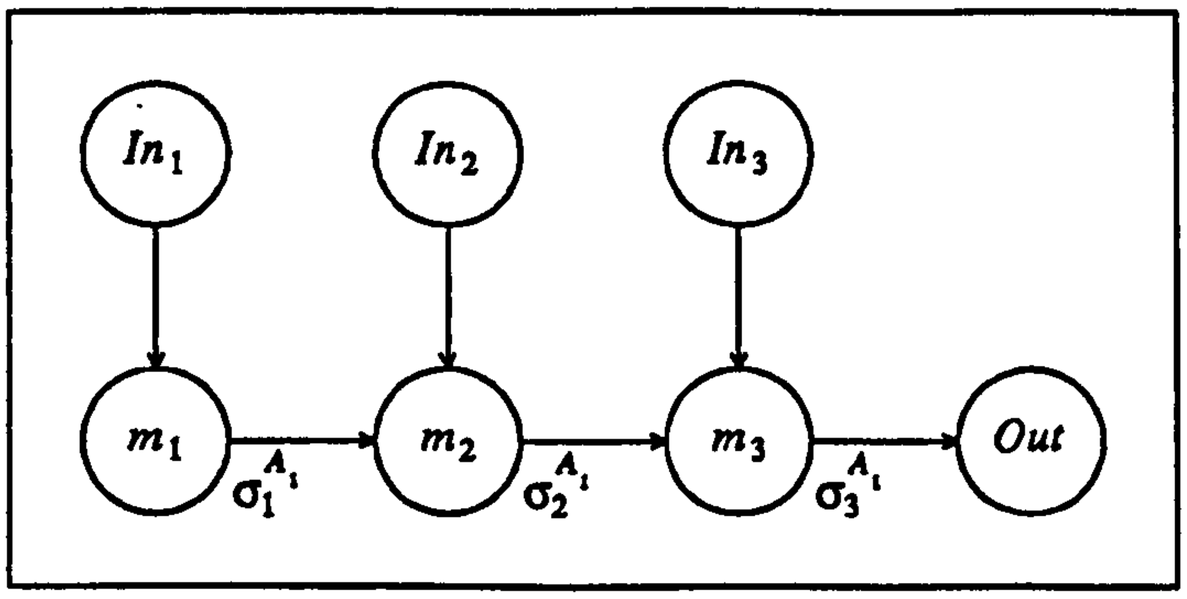


Figure 8.7 - FIR illustrated for $n = 3$.

$$(\forall z \in \mathbf{Z})\ (\ \sigma_1^{A_1}(z) = w_1 . z\ )$$

and for $i = 2,3$,

$$\sigma_i^{A_1} : \mathbf{Z} \times \mathbf{Z} \longrightarrow \mathbf{Z}$$

where

$$(\forall z_1, z_2 \in \mathbf{Z})\ (\ \sigma_i^{A_1}(z_1, z_2) = z_2 + w_i . z_1\ ) \qquad (76)$$

Of course, $\sigma_i^{A_1}$ is the functional specification of module m_i of FIR (as depicted in Figure 8.7) for $i = 1,2,3$.

If we extend A_1 with streams over $\mathbf{Z}$, namely $[T \longrightarrow \mathbf{Z}]$, and stream evaluation $eval_{A_1}^{\mathbf{Z}} : T \times [T \longrightarrow \mathbf{Z}] \longrightarrow \mathbf{Z}$, then this new algebra is $\underline{A}_1$ which is $\underline{S}_1$-sorted when $\underline{S}_1$ is S_1 extended with the symbol '$\underline{\mathbf{Z}}$' (to name streams over $\mathbf{Z}$, a carrier of $\underline{A}_1$).

FIR is now formalised by its value function V_{FIR} where

$$V_{FIR} = (V_1, V_2, V_3) : T \times [T \longrightarrow \mathbf{Z}^3] \times \mathbf{Z}^3 \longrightarrow \mathbf{Z}^3$$

is defined coordinatewise by

$$V_i(0, \underline{a}, x) = x_i$$

for $i = 1,2,3$, and

$$V_1(t+1, \underline{a}, x) = \sigma_1^{A_1}(\underline{a}_1(t))$$

and for $i = 2,3$,

$$V_i(t+1, \underline{a}, x) = \sigma_i^{A_1}(\underline{a}_i(t), V_{i-1}(t, \underline{a}, x))$$

for each time $t \in T$, input stream $\underline{a} = (\underline{a}_1, \underline{a}_2, \underline{a}_3) : T \longrightarrow \mathbf{Z}^3$, and initial values $x = (x_1, x_2, x_3) \in \mathbf{Z}^3$.

Alternatively, if Σ_1 is the signature of A_1 then FIR is formalised as a scheme $\alpha_{FIR} \in PR(\underline{\Sigma}_1)$ of arity (T$\underline{\mathrm{D}}$D,D) where D = ZZZ $\in S_1^3$, and $[\![\alpha_{FIR}]\!]_{\underline{A}_1} = V_{FIR}$ (see Theorem 3.4.3).

8.5.2 The First Level of Implementation.

Let us implement the modules of FIR (or rather, their functional specifications) by synchronous networks over a lower-level algebra A_2.

Ignoring m_1 for the time being, on inspecting the modules of FIR we see that module m_i $(i=2,3)$ performs a 'multiplication by a constant' and an addition in one step (see (76)); let us now assume that 'multiplication by a constant' and addition are the only available primitive operations.

Consider Figure 8.8 which illustrates a network N_i to implement $\sigma_i^{A_1}$ (for $i=2,3$). N_i is a synchronous network over an algebra B_i which comprises a clock T_i, the Booleans $\mathbb{B}$, the integers $\mathbf{Z}$, and in addition to standard constants and operations, B_i has an operation $\sigma_{i,j}^{B_i}$ to specify module $m_{i,j}$ of N_i for $j=1,2$. These operations are:

$$\sigma_{i,1}^{B_i}: \mathbf{Z} \longrightarrow \mathbf{Z}$$

where

$$(\forall z \in \mathbf{Z}) \ (\sigma_{i,1}^{B_i}(z) = w_i \, . \, z \ )$$

and

$$\sigma_{i,2}^{B_i}: \mathbf{Z} \times \mathbf{Z} \longrightarrow \mathbf{Z}$$

where

$$(\forall z_1, z_2 \in \mathbf{Z}) \ (\sigma_{i,2}^{B_i}(z_1, z_2) = z_1 + z_2 \ )$$

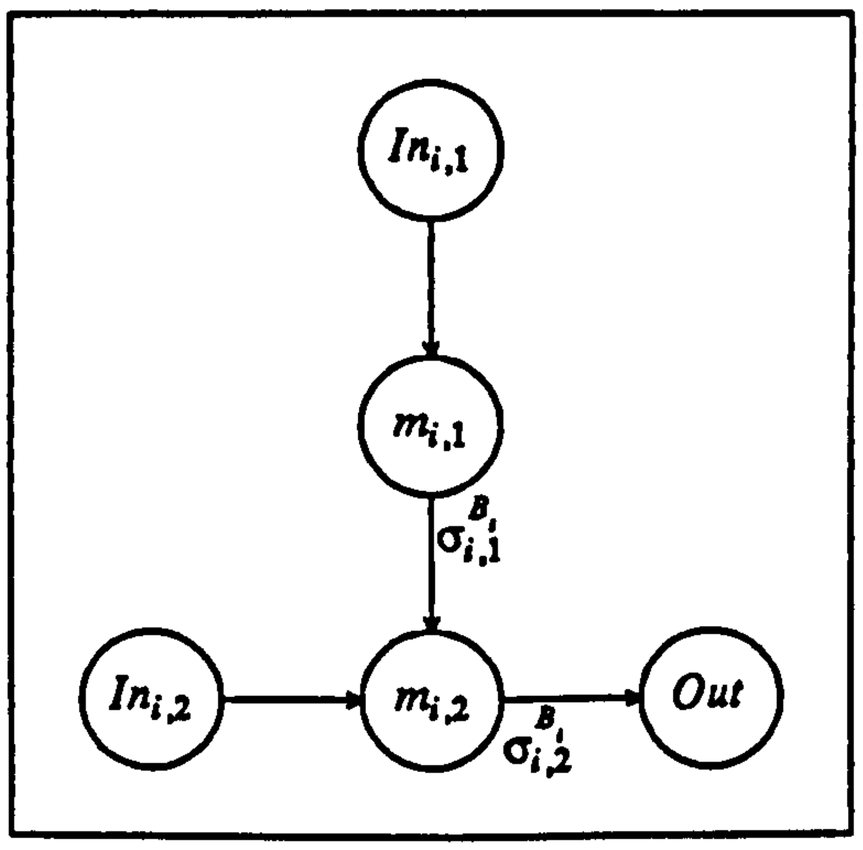


Figure 8.8 - Network N_i for $i=2,3$.

Since N_i has two sources and two modules, N_i is formalised via the static value function v_{N_i} where

$$v_{N_i} = (v_{i,1}, v_{i,2}) : T_i \times \mathbf{Z}^2 \times \mathbf{Z}^2 \longrightarrow \mathbf{Z}^2$$

is defined coordinatewise by

$$v_{i,j}(0,a,x) = x_j$$

for $j = 1,2$, and

$$v_{i,1}(t+1,a,x) = \sigma_{i,1}^{B_i}(a_1)$$

and

$$v_{i,2}(t+1,a,x) = \sigma_{i,2}^{B_i}(a_2, v_{i,1}(t,a,x))$$

for each time $t \in T_i$, fixed input $a = (a_1,a_2) \in \mathbb{Z}^2$, and initial values $x = (x_1,x_2) \in \mathbb{Z}^2$.

Since N_i has a single sink supplied by module $m_{i,2}$ of N_i, the static output specification f_{N_i} is

$$f_{N_i} : T_i \times \mathbb{Z}^2 \times \mathbb{Z}^2 \longrightarrow \mathbb{Z}$$

defined by

$$f_{N_i}(t,a,x) = v_{i,2}(t,a,x)$$

for each $t \in T_i$, $a \in \mathbb{Z}^2$, and $x \in \mathbb{Z}^2$.

Now let $\lambda_i : \mathbb{Z}^2 \longrightarrow T_i$ be defined by $\lambda_i(a) = 2$ for each $a \in \mathbb{Z}^2$. Then it is easy to show that for each $a \in \mathbb{Z}^2$,

$$f_{N_i}(\lambda_i(a),a,\zeta_i) = \sigma_i^{A_i}(a) \tag{77}$$

for any vector $\zeta_i \in \mathbb{Z}^2$ of initial values, and thus N_i implements $\sigma_i^{A_i}$ with respect to λ_i and $\zeta_i = (0,0)$ say.

For $\sigma_1^{A_1}$, since this operation is just 'multiplication by a constant', the network N_1 of Figure 8.9 suffices to implement it. N_1 is defined over algebra B_1 with clock T_1, where B_1 comprises T_1, $\mathbb{B}$, and $\mathbb{Z}$ as carriers, and (in addition to standard operations) the single operation $\sigma_{1,1}^{B_1} : \mathbb{Z} \longrightarrow \mathbb{Z}$ defined by $\sigma_{1,1}^{B_1}(z) = \sigma_1^{A_1}(z) = w_1 . z$ for each $z \in \mathbb{Z}$.

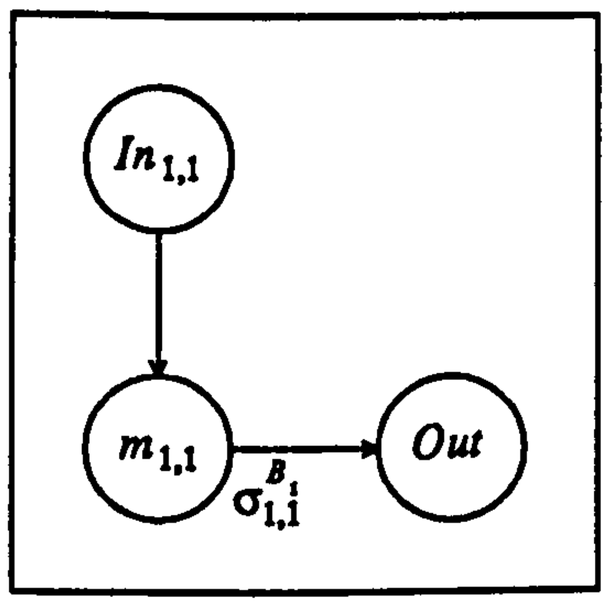


Figure 8.9 - N_1.

Clearly, N_1 has static output specification f_{N_1} of the form

$$f_{N_1} : T_1 \times \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$$

Moreover, it is obvious that for any initial value $\zeta_1 \in \mathbb{Z}$ we have

$$f_{N_1}(\lambda_1(a),a,\zeta_1) = \sigma_1^{A_1}(a) \tag{78}$$

for each $a \in \mathbf{Z}$ when $\lambda_1 : \mathbf{Z} \longrightarrow T_1$ is defined by $\lambda_1(a) = 1$ for each $a \in \mathbf{Z}$. Thus N_1 implements $\sigma_1^{A_1}$ with respect to λ_1 and $\zeta_1 = 0$ say.

On substituting the networks N_1, N_2, and N_3 for modules m_1, m_2, and m_3 of FIR respectively, we obtain the network system M of Figure 8.10.

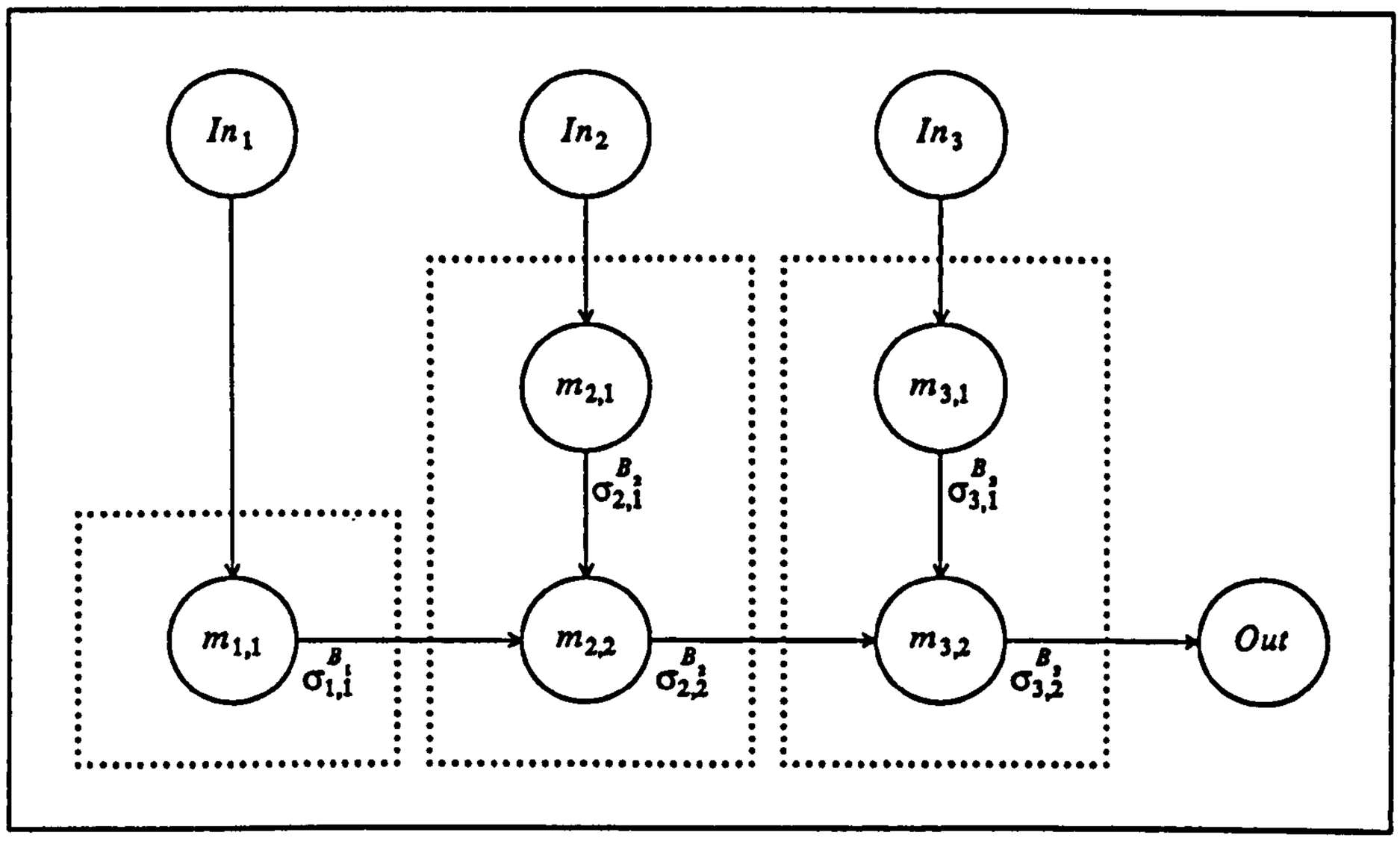


Figure 8.10 - M.

The steps performed by M determine a (new) clock T' (isomorphic to T). Let A_2 be the algebra comprising T', $\mathbb{B}$, and all the carriers, constants, and operations of B_1, B_2, and B_3 (including their standard constants, operations, and domains). Then as explained in Section 8.1, $\underline{A}_2$ involves all the data sets and operations needed to define the behaviour of M. We will now show (by way of preparation for the Hierarchy Theorem), that the network implementations of the modules of FIR determine an $\underline{A}_2$-implementation of $\underline{A}_1$.

First note that A_2 has carriers T', $\mathbb{B}$, T_1, T_2, T_3, and $\mathbf{Z}$, and so A_2 is S_2-sorted when $S_2 = \{\mathrm{T},\mathrm{B},\mathrm{T}_1,\mathrm{T}_2,\mathrm{T}_3,\mathrm{Z}\}$: here the symbol '$\mathrm{T}_i$' names T_i for $i = 1,2,3$ of course, but note that in A_2 'T' names T' whereas in A_1 it names T. Thus, technically, we have $\mathrm{zero}^{A_1} \in T$ and $\mathrm{succ}^{A_1} = succ_T : T \longrightarrow T$, but $\mathrm{zero}^{A_2} \in T'$ and $\mathrm{succ}^{A_2} = succ_{T'} : T' \longrightarrow T'$.

The stream algebra $\underline{A}_2$ is formed by adding $[T' \longrightarrow \mathbf{Z}]$ to A_2 as a new carrier, $eval^{\mathbf{Z}}_{A_2} : T' \times [T' \longrightarrow \mathbf{Z}] \longrightarrow \mathbf{Z}$ as a new operation, and $\underline{S}_2$ is $S_2 \cup \{\underline{\mathrm{Z}}\}$: we will now construct an $\underline{A}_2$-

implementation $I_1 = <\gamma_1, Gn_1, Tr_1>$ of $\underline{A}_1$.

First we must show that $\underline{A}_1$ is $\underline{A}_2$-coded; but this is trivial since all the carriers of $\underline{A}_1$ are carriers of $\underline{A}_2$. We leave the reader to formally show that if $w_1 : \underline{S}_1 \longrightarrow \underline{S}_2^+$ is the identity function, and if each component $\gamma(s)$ of $\gamma_1 : \underline{S}_1 \longrightarrow [\underline{A}_2^+ \longrightarrow \underline{A}_1]$ are also identity functions on the appropriate sets, then $\underline{A}_1$ is $\underline{A}_2$-coded by γ_1 via w_1.

It is also easy to see that $\underline{A}_1$ is $\underline{A}_2$-generated: for any algebra A, the constants of $\underline{A}$ are the constants of A, and in our case the constants of A_1 are constants in A_2, thus the constants of $\underline{A}_1$ are constants in $\underline{A}_2$. Thus $\underline{A}_1$ is $\underline{A}_2$-generated by $Gn_1 = <\rho_c : c \in \underline{\Sigma}_1>$ wherein $\rho_c = c \in T(\underline{\Sigma}_2)$ for each constant symbol c.

Now we must show that $\underline{A}_1$ is $\underline{A}_2$-tracked. Of course, the operations of $\underline{A}_1$ which are common to $\underline{A}_1$ and $\underline{A}_2$ implement themselves. For example, we have $\text{succ} \in (\underline{\Sigma}_1)_{T,T}$ and $\text{succ} \in (\underline{\Sigma}_2)_{T,T}$ when $\underline{\Sigma}_i$ is the signature of $\underline{A}_i$ for $i = 1,2$. In $\underline{A}_1$ the symbol succ is interpreted as the successor function on T:

$$\text{succ}^{\underline{A}_1} = \text{succ}^{A_1} = succ_T : T \longrightarrow T$$

whereas in $\underline{A}_2$,

$$\text{succ}^{\underline{A}_2} = \text{succ}^{A_2} = succ_{T'} : T' \longrightarrow T'$$

using the fact that $\underline{A}_i|_{\Sigma_i} = A_i$ for $i = 1,2$. However, T and T' are the same (isomorphic) and so it is obvious that $\alpha_{succ} = \text{succ}$ tracks $\text{succ}^{\underline{A}_2}$; we leave the details to the reader. (Hint: use the isomorphism invariance of PR-computability; see Lemma 3.5.9.)

We now consider the remaining operations of $\underline{A}_1$, namely $\sigma_i^{\underline{A}_1} = \sigma_i^{A_1}$ for $i = 1,2,3$.

Let Ω_i be the signature of B_i for $i = 1,2,3$; then there exists $\delta_{N_i} \in \text{PR}(\Omega_i)$ such that $[\![\delta_{N_i}]\!]_{B_i} = f_{N_i}$ by Lemma 3.4.4. Now let α_{λ_i} be defined by

$$\alpha_{\lambda_i} = \begin{cases} \text{succ}_i \circ \text{zero}_i^{\text{Z}} & \text{if } i = 1 \\ \text{succ}_i \circ \text{succ}_i \circ \text{zero}_i^{\text{ZZ}} & \text{if } i = 2,3 \end{cases}$$

(Here 'zero_i' and 'succ_i' are the names of T_i's zero and successor function respectively, and 'zero_i^{Z}' and '$\text{zero}_i^{\text{ZZ}}$' are instances of the generic constant function schema 'c^{w}'.) Then $\alpha_{\lambda_i} \in \text{PR}(\Omega_i)$ with arity (Z, T_i) for $i = 1$ and (ZZ, T_i) for $i = 2,3$. Furthermore, it should be clear that $[\![\alpha_{\lambda_i}]\!]_{B_i} = \lambda_i$ for $i = 1,2,3$.

Now let ι_1 be defined by

$$\iota_1 = \delta_{N_1} \circ <\alpha_{\lambda_1}, \text{U}_1^{\text{Z}}, \text{zero}^{\text{Z}}>$$

Then $\iota_1 \in \text{PR}(\Omega_1)_{\text{Z},\text{Z}}$ and it is easy to check that for each $a \in \mathbf{Z}$ we have

$$[\![\iota_1]\!]_{B_1}(a) = f_{N_1}(\lambda_1(a), a, \zeta_1) = \sigma_1^{A_1}(a) \tag{79}$$

(using (78)). However, since $B_1 = \underline{A}_2|_{\Omega_1}$, we have $\iota_1 \in \text{PR}(\underline{\Sigma}_2)_{\text{Z},\text{Z}} = \text{PR}(\underline{\Sigma}_2)_{w_1(\text{Z}),w_1(\text{Z})}$ with $[\![\iota_1]\!]_{\underline{A}_2} = [\![\iota_1]\!]_{B_1}$. Using the facts that $\gamma_1(\text{Z})$ is the identity function on $\mathbf{Z}$, and $\sigma_1^{A_1} = \sigma_1^{\underline{A}_1}$, we have from (79) that

$$\gamma_1(\text{Z})([\![\iota_1]\!]_{\underline{A}_2}(a)) = \sigma_1^{\underline{A}_1}(\gamma_1(\text{Z})(a))$$

for each $a \in \mathbf{Z}$; that is, ι_1 is an $\underline{A}_2$-tracking of $\sigma_1^{\underline{A}_1}$.

□

We can find $\underline{A}_2$-trackings of $\sigma_i^{\underline{A}_1}$ for $i=2,3$ in much the same way:

Let ι_i $(i=2,3)$ be defined by

$$\iota_i = \delta_{N_i} \circ <\alpha_{\lambda_i}, U_1^{zz}, U_2^{zz}, zero^{zz}, zero^{zz}>$$

Then $\iota_i \in PR(\Omega_i)_{zz,z}$ and for each $a \in \mathbb{Z}^2$ we have

$$[\![\iota_i]\!]_{B_i}(a) = f_{N_i}(\lambda_i(a), a, \zeta_i) = \sigma_i^{\underline{A}_1}(a) \tag{80}$$

(using (77)). However, as in the case $i=1$, since $B_i = \underline{A}_2|_{\Omega_i}$ for $i=2,3$ we have $\iota_i \in PR(\underline{\Sigma}_2)_{zz,z} = PR(\underline{\Sigma}_2)_{w_1(zz),w_1(z)}$ and $[\![\iota_i]\!]_{\underline{A}_2} = [\![\iota_i]\!]_{B_i}$. Thus from (80) we have (again since $\gamma_1(z)$ is the identity function and $\sigma_i^{\underline{A}_1} = \sigma_i^{\underline{A}_1}$)

$$\gamma_1(z)([\![\iota_i]\!]_{\underline{A}_2}(a)) = \sigma_i^{\underline{A}_1}(\gamma_1(zz)(a))$$

for each $a \in \mathbb{Z}^2$; that is, ι_i is an $\underline{A}_2$-tracking of $\sigma_i^{\underline{A}_1}$.

Now define $Tr_1 \subseteq PR(\underline{\Sigma}_2)$ by $Tr_1 = <\alpha_\sigma : \sigma \in \underline{\Sigma}_1>$ where

$$\alpha_\sigma = \begin{cases} \iota_i & \text{if } \sigma = \sigma_i \text{ for } i \in \{1,2,3\} \\ \sigma & \text{otherwise} \end{cases}$$

Then $\underline{A}_1$ is $\underline{A}_2$-tracked by Tr_1, and we conclude that $I_1 = <\gamma_1, Gn_1, Tr_1>$ is an $\underline{A}_2$-implementation of $\underline{A}_1$.

Complexity. What is the complexity of I_1? Of course, the answer to this question is dependent on the performance measures we choose for $\underline{A}_1$ and $\underline{A}_2$. Suppose P_1 and P_2 are uniform performance measures for $\underline{A}_1$ and $\underline{A}_2$ respectively.

From Definition 8.3.1 we must show that for some fixed polynomial $\Psi \in \mathbb{N}[x]$, for every $\sigma \in \underline{\Sigma}_1$ we have

$$\lambda_{P_2}(\alpha_\sigma)(a) \leq \Psi(\sigma^{P_1}(\gamma(u)(a)))$$

for every $a \in \underline{A}_2^{w_1(u)}$ when $\underline{A}_1^u$ is the domain of $\sigma^{\underline{A}_1} = \sigma^{\underline{A}_1}$. However, since P_1 is uniform, this means that we must show

$$\lambda_{P_2}(\alpha_\sigma)(a) \leq \Psi(1) \tag{81}$$

Now, for $\sigma \neq \sigma_i$ for any $i \in \{1,2,3\}$ we have $\alpha_\sigma = \sigma$ and so for each a in the common domain of $\sigma^{\underline{A}_1}$ and $[\![\alpha_\sigma]\!]_{\underline{A}_2}$ we have

$$\lambda_{P_2}(\alpha_\sigma)(a) = \lambda_{P_2}(\sigma)(a) = \sigma^{P_2}(a) = 1$$

(since P_2 is uniform), and so we must show $1 \leq \Psi(1)$; but this is true for any Ψ (since our polynomials have natural number coefficients, not all of which can be zero).

It remains to consider $\sigma = \sigma_i$ for $i \in \{1,2,3\}$. First consider σ_1: for any a in the domain of $\sigma_1^{\underline{A}_1}$ we have

$$\begin{aligned} \lambda_{P_2}(\alpha_{\sigma_1})(a) &= \lambda_{P_2}(\iota_1)(a) \\ &= \lambda_{P_2}(\delta_{N_1})(1,a,0) + \max\{\lambda_{P_2}(\alpha_{\lambda_1})(a), 1, 1\} \end{aligned}$$

(by definition of ι_1)

$$= \lambda_{P_2}(\delta_{N_1})(1,a,0) + 2$$
$$= 2+2.1 + 2 = 6$$

by Lemma 3.5.5

Similarly, for i=2,3, for any a in the domain of $\sigma_i^{A_1}$ we have

$$\lambda_{P_2}(\alpha_{\sigma_i})(a) = \lambda_{P_2}(\iota_i)(a)$$
$$= \lambda_{P_2}(\delta_{N_i})(2,a,0,0) + \max\{\lambda_{P_2}(\alpha_{\lambda_i})(a),1,1,1,1\}$$

(by definition of ι_i)

$$= \lambda_{P_2}(\delta_{N_i})(2,a,0,0) + 3$$
$$= 2+2.3 + 3 = 11$$

again by Lemma 3.5.5.

Thus, if we define $\Psi \in \mathbb{N}[x]$ by $\Psi(x)=11x$ then we see that I_1 is *linear* implementation of $\underline{A}_1$.

8.5.3 The Second Level of Implementation.

In this section we will further decompose FIR by implementing the operations of A_2, that is, the modules of M, by further synchronous networks. We will represent the integers $\mathbb{Z}$ over $\mathbb{B}\times\mathbb{N}$ via the function $\theta : \mathbb{B}\times\mathbb{N} \xrightarrow{\text{onto}} \mathbb{Z}$ defined in Example 8.2.3. Without further ado, it is clear that this change in data sets will lead to an implementation of M, let us call it M', which will be as sketched in Figure 8.11: comparing this figure with Figure 8.10, we see that each source In_i of M (which supplies elements of $\mathbb{Z}$) has become two sources $I_{i,\mathbb{B}}$ and $I_{i,\mathbb{N}}$ in M' (supplying elements of $\mathbb{B}$ and $\mathbb{N}$ respectively). Correspondingly, each channel in M has become a pair of channels with one of the channels carrying Booleans, and the other carrying elements of $\mathbb{N}$. In the same way, M's sink Out has become two sinks $O_{\mathbb{B}}$ and $O_{\mathbb{N}}$. (Here we have adopted the convention that in a vertical pair of channels, the leftmost is the Boolean channel, and in a horizontal pair, the lower channel is the Boolean channel). Also, each k-input module in M has become a $2k$-input, 2-output network over $\mathbb{B}$ and $\mathbb{N}$.

We will now describe synchronous network implementations for the modules of M. Since a typical module of M is denoted by '$m_{i,j}$', we will denote a network to implement $m_{i,j}$ by '$N_{i,j}$'; this will be a synchronous network over an algebra '$B_{i,j}$' whose clock is '$T_{i,j}$' (here $j=1$ when $i=1$, but j ranges over $\{1,2\}$ when $i=2$ or $i=3$).

Implementing Multiplication. Let us first implement module $m_{i,1}$ of m for $i=1,2,3$. Module $m_{i,1}$ is specified by the operation $\sigma_{i,1}^{A_2}$ which multiplies its argument by the constant w_i for $i=1,2,3$. For simplicity let us assume that the weights w_1, w_2, and w_3, are all positive and so can be regarded as elements of $\mathbb{N}$.

Given an integer $z_1 \in \mathbb{Z}$, let $z=\sigma_i^{A_2}(z_1)=w_i \,.\, z_1$. Since w_i is positive, z has the same sign as z_1, and modulus of z is $f_i(|z_1|)$ where $f_i : \mathbb{N} \longrightarrow \mathbb{N}$ is defined by $f_i(n)=w_i \,.\, n$ for each $n \in \mathbb{N}$ for $i=1,2,3$. (Note that $|\,.\,| : \mathbb{Z} \longrightarrow \mathbb{N}$.)

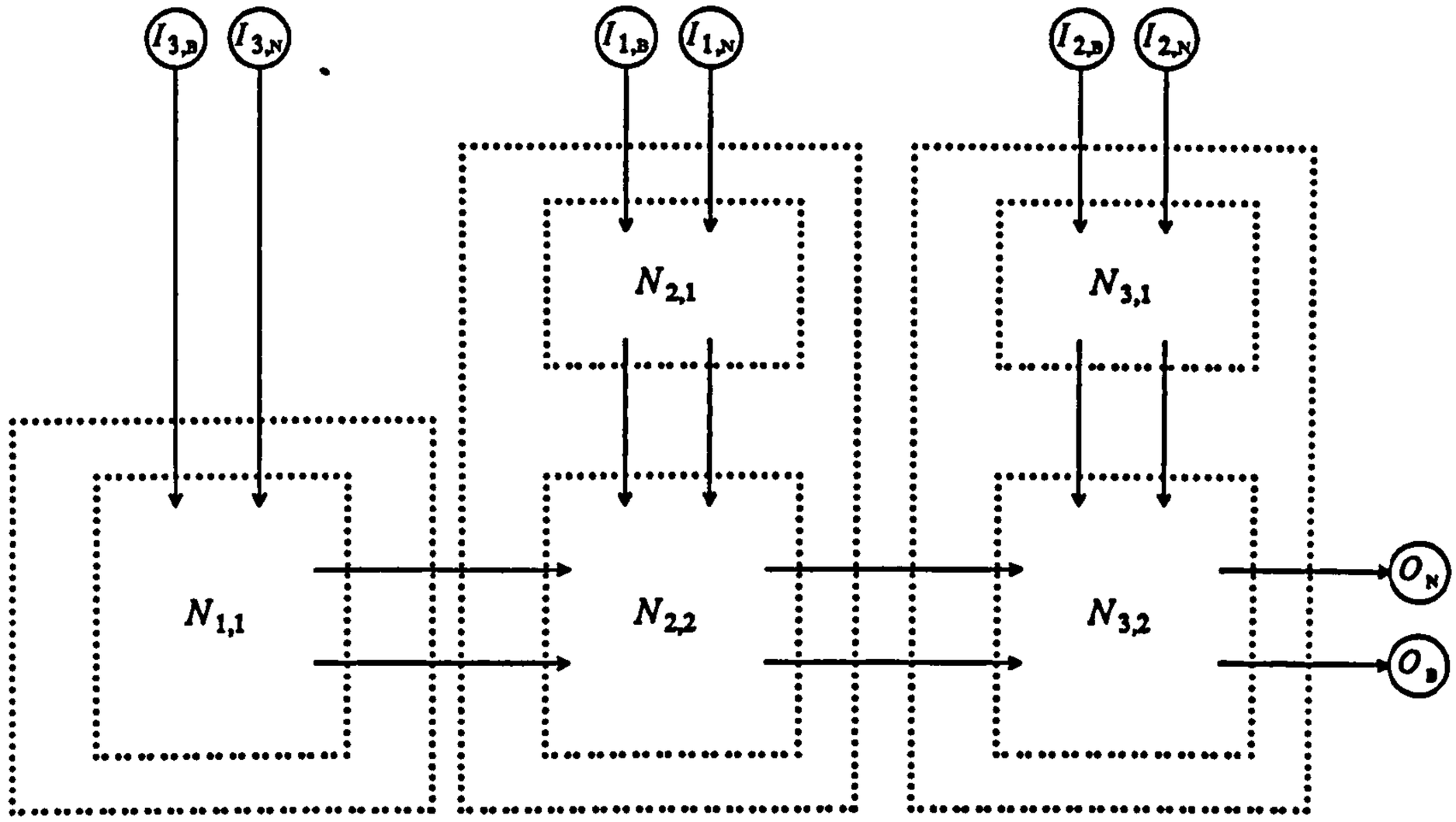


Figure 8.11.

Now, given a code $(b,n) \in \mathbb{B} \times \mathbb{N}$ of an integer $z \in \mathbb{Z}$, z is positive iff $b = tt$ or $b = ff$ and $n = 0$, and $|z| = n$. Thus if $(b_1,n_1) \in \mathbb{B} \times \mathbb{N}$ is the code of z_1 above, then the code of $z = w_i \cdot z_1$ is (b,n) where $b = b_1$ and $n = f_i(n_1)$. Thus for $i = 1,2,3$ we have

$$(\forall b \in \mathbb{B})(\forall n \in \mathbb{N}) \ \ (\theta(id(b),f_i(n)) = \sigma^{\wedge}_{i,1}(\theta(b,n))) \qquad (82)$$

where $id : \mathbb{B} \longrightarrow \mathbb{B}$ is the identity function.

The observation or property (82) leads immediately to the network $N_{i,1}$ of Figures 8.12 ($i = 1$) and 8.13 ($i = 2,3$). (Note that these figures show the same network: the reason for the two figures is that for $i = 1$ the outputs need to emerge from the right, whereas for $i = 2,3$ they need to emerge from the bottom; see Figure 8.11.) Of course, $N_{i,1}$ is a synchronous network over $B_{i,1}$ when $B_{i,1}$ has a clock $T_{i,1}$ and $\mathbb{B}$ and $\mathbb{N}$ as carriers, and id and f_i as operations (in addition to standard constants and operations). Furthermore $N_{i,1}$ will have a static output specification of the form

$$f_{N_{i,1}} : T_{i,1} \times \mathbb{B} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \longrightarrow \mathbb{B} \times \mathbb{N}$$

Trivially, for any $\zeta_{i,1} \in \mathbb{B} \times \mathbb{N}$ we have

$$(\forall b \in \mathbb{B})(\forall n \in \mathbb{N}) \ \ (f_{N_{i,1}}(1,b,n,\zeta_{i,1}) = (id(b),f_i(n))) \qquad (83)$$

Thus, if $\lambda_{i,1} : \mathbb{B} \times \mathbb{N} \longrightarrow T_{i,1}$ is defined by $\lambda_{i,1}(b,n) = 1$ for each $b \in \mathbb{B}$ and $n \in \mathbb{N}$, then from (82) and (83) we have

$$(\forall b \in \mathbb{B})(\forall n \in \mathbb{N}) \ \ (\theta(f_{N_{i,1}}(\lambda_{i,1}(b,n),b,n,\zeta_{i,1})) = \sigma^{\wedge}_{i,1}(\theta(b,n))) \qquad (84)$$

for any initial values $\zeta_{i,1} \in \mathbb{B} \times \mathbb{N}$.

Thus $N_{i,1}$ implements $\sigma_{i,1}^{A_2}$ with respect to $\lambda_{i,1}$ and $\zeta_{i,1}=(tt,0)$ say.

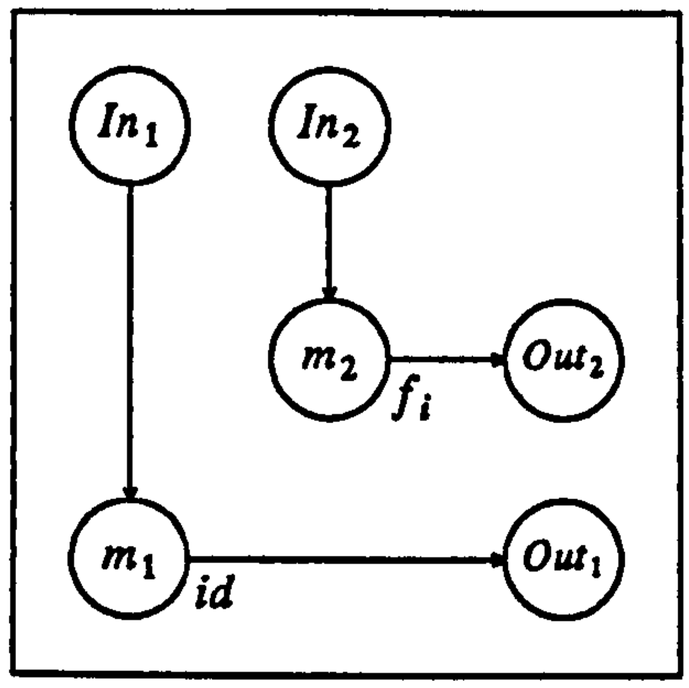


Figure 8.12 - $N_{1,1}$.

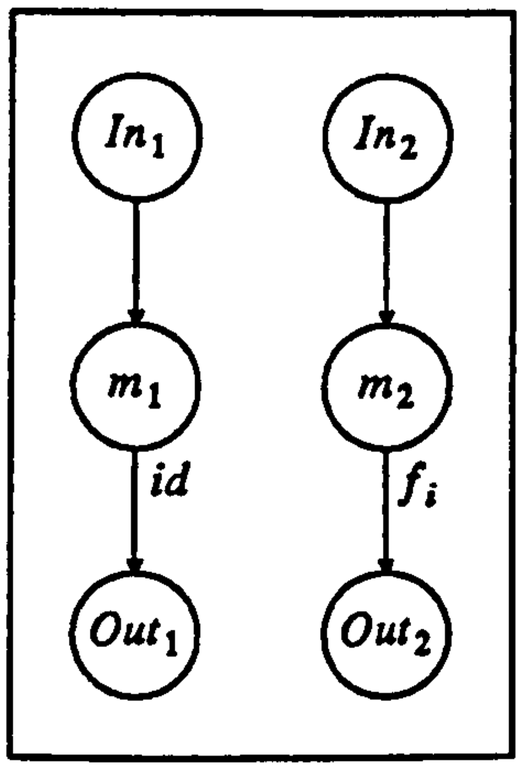


Figure 8.13 - $N_{i,1}$ for $i=2,3$.

Implementing Addition. It remains to implement the addition modules of M, that is, module $m_{i,2}$ for $i=2,3$. We will now describe a network $N_{i,2}$ to implement $\sigma_{i,2}^{A_2}$ (addition on $\mathbb{Z}$) for $i=2,3$. Note that since $m_{2,2}$ and $m_{3,2}$ are specified by the same function, addition on $\mathbb{Z}$, the networks $N_{2,2}$ and $N_{3,2}$ (and the algebras over which they are defined) will be the same.

Let $+_N$ denote addition on $\mathbb{N}$. Also let $-_N$ denote subtraction on $\mathbb{N}$: conventionally, $n_1 -_N n_2$ is defined to be zero if $n_2 > n_1$.

Given $z_1, z_2 \in \mathbb{Z}$, let $z = z_1 + z_2$. Then we can describe z by describing its sign and modulus in terms of the signs and moduli of z_1 and z_2:

First, if z_1 and z_2 are both positive or both negative, then the sign of z is the common sign of z_1 and z_2, and the modulus of z is $|z_1|+_N|z_2|$. Secondly, if z_1 is positive and z_2 is negative, then z is positive if $|z_1| \geq |z_2|$, and negative if $|z_1| < |z_2|$; also, $|z| = |z_1|-_N|z_2|$ if $|z_1| \geq |z_2|$, and $|z| = |z_2|-_N|z_1|$ if $|z_1| < |z_2|$. Finally, if z_1 is negative and z_2 is positive, then z is positive if $|z_1| < |z_2|$, and negative if $|z_1| \geq |z_2|$; also, $|z| = |z_1|-_N|z_2|$ if $|z_1| \geq |z_2|$, and $|z| = |z_2|-_N|z_1|$ if $|z_1| < |z_2|$.

We can use this description to devise a synchronous network N_+ to implement addition on $\mathbb{Z}$ over $\mathbb{B}$ and $\mathbb{N}$ in the following way:

Let $(b_i,n_i) \in \mathbb{B}\times\mathbb{N}$ be the code of z_i for $i = 1,2$. Then the code of z is (b,n) where

$$b = \begin{cases} tt & \text{if } b_1 = tt \text{ and } b_2 = tt \\ ff & \text{if } b_1 = ff \text{ and } b_2 = ff \\ tt & \text{if } b_1 = tt \text{ and } b_2 = ff \text{ and } n_1 \geq n_2 \\ ff & \text{if } b_1 = tt \text{ and } b_2 = ff \text{ and } n_1 < n_2 \\ ff & \text{if } b_1 = ff \text{ and } b_2 = tt \text{ and } n_1 \geq n_2 \\ tt & \text{if } b_1 = ff \text{ and } b_2 = tt \text{ and } n_2 < n_2 \end{cases}$$

and

$$n = \begin{cases} n_1+_N n_2 & \text{if } b_1 = tt \text{ and } b_2 = tt \\ n_1+_N n_2 & \text{if } b_1 = ff \text{ and } b_2 = ff \\ n_1-_N n_2 & \text{if } b_1 = tt \text{ and } b_2 = ff \text{ and } n_1 \geq n_2 \\ n_2-_N n_1 & \text{if } b_1 = tt \text{ and } b_2 = ff \text{ and } n_1 < n_2 \\ n_1-_N n_2 & \text{if } b_1 = ff \text{ and } b_2 = tt \text{ and } n_1 \geq n_2 \\ n_2-_N n_1 & \text{if } b_1 = ff \text{ and } b_2 = tt \text{ and } n_2 < n_2 \end{cases}$$

respectively.

More concisely, b and n are equivalently defined by

$$b = (b_1 \wedge (n_1 \geq n_2)) \vee (b_2 \wedge (n_1 < n_2)) \tag{85}$$

and

$$n = dc\,(b_1 \oplus b_2, (n_1-_N n_2)+_N (n_2-_N n_1), n_1+_N n_2) \tag{86}$$

wherein $\oplus: \mathbb{B}^2 \longrightarrow \mathbb{B}$ is 'exclusive or' defined by

$$(\forall b_1,b_2 \in \mathbb{B})\ (\ b_1 \oplus b_2 = (b_1 \wedge \sim b_2) \vee (\sim b_1 \wedge b_2)\ )$$

and $dc : \mathbb{B}\times\mathbb{N}^2 \longrightarrow \mathbb{N}$ is defined by

$$(\forall b \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N})\ \ dc_N(b,n_1,n_2) = \begin{cases} n_1 & \text{if } b = tt \\ n_2 & \text{if } b = ff \end{cases}$$

This description of addition on $\mathbb{Z}$ leads immediately to the synchronous network $N_+ = N_{i,2}$ $(i = 2,3)$ of Figure 8.14. Note that the '*dc*' module has three inputs channels: the middle channel carries elements of $\mathbb{B}$ and the top and bottom channels elements of $\mathbb{N}$; we imagine that the modules chooses the top value if the Boolean is true, the bottom value otherwise.

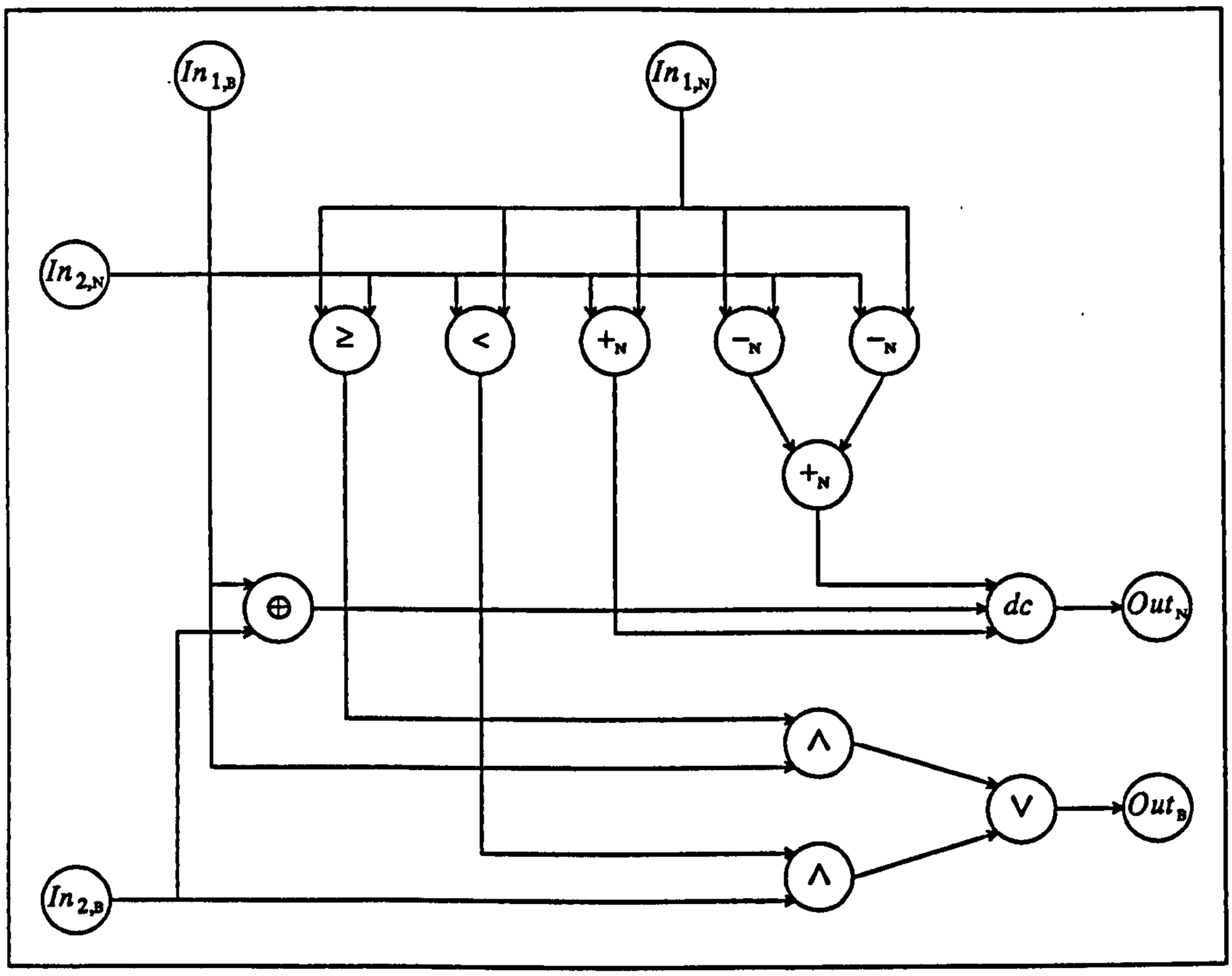


Figure 8.14 - $N_{i,2}$ for $i = 2,3$.

Of course, $N_{i,2}$ is a synchronous network over $B_{i,2}$ when $B_{i,2}$ has a clock $T_{i,2}$ (say) and $\mathbb{B}$ and $\mathbb{N}$ as carriers, and $\oplus$, $\wedge$, $\vee$, $\geq$, $<$, $+_N$, $-_N$, and dc as operations (in addition to standard constants and operations). Furthermore, $N_{i,2}$ has six modules specified by Boolean-valued functions, five modules specified by natural number-valued functions, two pairs of sources supplying pairs from $\mathbb{B}\times\mathbb{N}$, and two sinks receiving pairs from $\mathbb{B}\times\mathbb{N}$. Thus $N_{i,2}$ has a static output specification $f_{N_{i,2}}$ of functionality

$$f_{N_{i,2}} : T_{i,2}\times(\mathbb{B}\times\mathbb{N})^2\times\mathbb{B}^6\times\mathbb{N}^5 \longrightarrow \mathbb{B}\times\mathbb{N}$$

Moreover, it is easy to show that for any $\zeta_{i,2} \in \mathbb{B}^6\times\mathbb{N}^5$ we have

$$(\forall b_1,b_2 \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N}) \qquad (87)$$
$$f_{N_{i,2}}(3,b_1,n_1,b_2,n_2,\zeta_{i,2}) = (f_{\mathbb{B}}(b_1,n_1,b_2,n_2), f_{\mathbb{N}}(b_1,n_1,b_2,n_2))$$

where $f_{\mathbb{B}} : (\mathbb{B}\times\mathbb{N})^2 \longrightarrow \mathbb{B}$ is defined by

$$(\forall b_1,b_2 \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N}) \quad (f_{\mathbb{B}}(b_1,n_1,b_2,n_2) = (b_1 \wedge (n_1 \geq n_2)) \vee (b_2 \wedge (n_1 < n_2)))$$

and $f_{\mathbb{N}} : (\mathbb{B}\times\mathbb{N})^2 \longrightarrow \mathbb{N}$ is defined by

$$(\forall b_1,b_2 \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N}) \quad (f_{\mathbb{N}}(b_1,n_1,b_2,n_2) = dc(b_1 \oplus b_2, (n_1 -_{\mathbb{N}} n_2) + (n_2 -_{\mathbb{N}} n_1), n_1 +_{\mathbb{N}} n_2))$$

(Compare these definitions with (85) and (86) respectively.) Now, as explained above, given

$(b_i,n_i) \in (\mathbb{B}\times\mathbb{N})^2$ for $i=1,2$, the pair $(b,n) \in \mathbb{B}\times\mathbb{N}$ is the code of $z=z_1+z_2$ when (b_i,n_i) is the code of z_i for $i=1,2$ and b and n are defined by

$$b = f_{\mathbb{B}}(b_1,n_1,b_2,n_2)$$

and

$$n = f_{\mathbb{N}}(b_1,n_1,b_2,n_2)$$

respectively. That is,

$$(\forall b_1,b_2 \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N}) \qquad (88)$$
$$\theta(f_{\mathbb{B}}(b_1,n_1,b_2,n_2), f_{\mathbb{N}}(b_1,n_1,b_2,n_2)) = \theta(b_1,n_1) + \theta(b_2,n_2) = \sigma^{\Lambda}_{i,2}(\theta(b_1,n_1), \theta(b_2,n_2))$$

Thus, if $\lambda_{i,2}:(\mathbb{B}\times\mathbb{N})^2 \longrightarrow T_{i,2}$ is defined by $\lambda_{i,2}(a)=3$ for each $a \in (\mathbb{B}\times\mathbb{N})^2$, then from (87) and (88) we have

$$(\forall b_1,b_2 \in \mathbb{B})(\forall n_1,n_2 \in \mathbb{N}) \qquad (89)$$
$$\theta(f_{N_{i,2}}(\lambda_{i,2}(b_1,n_1,b_2,n_2), b_1, n_1, b_2, n_2, \zeta_{i,2})) = \sigma^{\Lambda}_{i,2}(\theta(b_1,n_1), \theta(b_2,n_2))$$

for any initial values $\zeta_{i,2} \in \mathbb{B}^6\times\mathbb{N}^5$.

Thus $N_{i,2}$ implements $\sigma^{\Lambda}_{i,2}$ with respect to $\lambda_{i,2}$ and $\zeta_{i,2}=(tt,tt,tt,tt,tt,tt,0,0,0,0,0)$ say. □

On substituting the networks $N_{i,j}$ for the modules $m_{i,j}$ of M, we obtain the network system M' of Figure 8.15. Whilst we have not yet considered network systems that are obtained from a synchronous network via *two* levels of substitution, an informal account of the operation of M' can be extrapolated from the discussion of Section 8.1 in the obvious way: as before with M, M' operates in steps wherein one step is to execute N_1, N_2, and N_3 simultaneously for the requisite number of steps; the only difference here is that in M', N_1, N_2, and N_3 are themselves network systems, and so 'one step' of N_i $(i=1,2,3)$ in M' is to execute its sub-networks simultaneously for the appropriate number of cycles. (Specifically, $N_{i,j}$ is executed for $\lambda_{i,j}(a)$ cycles of $T_{i,j}$ whenever the fixed input to $N_{i,j}$ is a.)

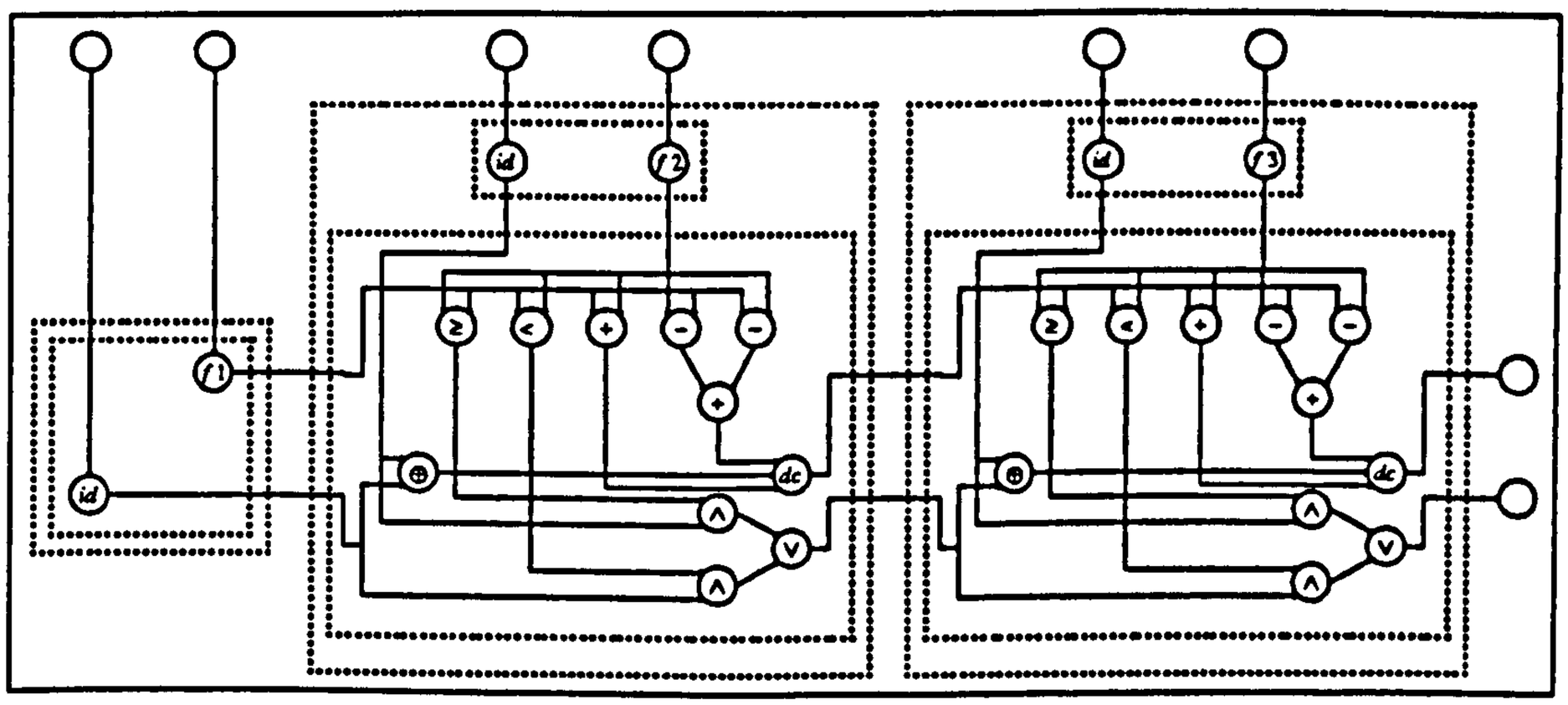


Figure 8.15 - The final design M''.

Let us denote the clock determined by M''s steps by T''. Then M' is specified in terms of T'', streams of the form $\underline{b} : T'' \longrightarrow \mathbb{B}$ and $\underline{n} : T'' \longrightarrow \mathbb{N}$, and the operations of the five algebras $B_{1,1}$, $B_{2,1}$, $B_{2,2}$, $B_{3,1}$, and $B_{3,2}$. More precisely, let A_3 be the algebra comprising the carriers

$$T'', \mathbb{B}, T_1, T_2, T_3, T_{1,1}, T_{2,1}, T_{2,2}, T_{3,1}, T_{3,2}, \mathbb{B}, \text{and } \mathbb{N}$$

In addition to standard constants and operations, A_3 has the operations

$$\oplus, \wedge, \vee, \geq, <, +_N, -_N, dc, id, f_1, f_2, \text{and } f_3$$

The first occurrence of '$\mathbb{B}$' in the list of A_3's carriers is, as always, there to make A_3 into a standard algebra. The second occurrence is as the carrier of an algebra B (say) whose single domain is $\mathbb{B}$ and operations are $\oplus$, id, $\wedge$, and $\vee$.

Now let S_3 be defined by $S_3 = \{\mathrm{T}, \mathrm{B}, \mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_3, \mathrm{T}_{1,1}, \mathrm{T}_{2,1}, \mathrm{T}_{2,2}, \mathrm{T}_{3,1}, \mathrm{T}_{3,2}, \mathrm{B}', \mathrm{N}\}$ Then A_3 is S_3-sorted with 'T' naming T'', 'N' naming $\mathbb{N}$, and 'B'' naming B.

Of course, $\underline{A}_3$ is constructed from A_3 in the usual way: we add $[T'' \longrightarrow \mathbb{B}]$ and $[T'' \longrightarrow \mathbb{N}]$ as new carriers, and $eval^{\mathbb{B}}_{A_3} : T'' \times [T'' \longrightarrow \mathbb{B}] \longrightarrow \mathbb{B}$ and $eval^{\mathbb{N}}_{A_3} : T'' \times [T'' \longrightarrow \mathbb{N}] \longrightarrow \mathbb{N}$ as new operations. $\underline{A}_3$ is $\underline{S}_3$-sorted when $\underline{S}_3 = S_3 \cup \{\underline{\mathrm{B}}', \underline{\mathrm{N}}\}$.

We are now in a position to construct an $\underline{A}_3$-implementation $I_2 = <\gamma_2, Tr_2, Gn_2>$ of $\underline{A}_2$:

First we must show that $\underline{A}_2$ is $\underline{A}_3$-coded. Let $w_2 : \underline{S}_2 \longrightarrow \underline{S}_3^+$ be defined by

$$w_2(s) = \begin{cases} \mathrm{B}'\mathrm{N} & \text{if } s = \mathrm{Z} \\ \underline{\mathrm{B}}'\underline{\mathrm{N}} & \text{if } s = \underline{\mathrm{Z}} \\ s & \text{otherwise} \end{cases}$$

for each $s \in \underline{S}_2$. Now let

$$\Theta : [T'' \longrightarrow \mathbb{B} \times \mathbb{N}] \longrightarrow [T' \longrightarrow \mathbb{Z}]$$

be defined by

$$\Theta(\underline{b}, \underline{n})(t) = \theta(\underline{b}(t), \underline{n}(t))$$

for each $\underline{b} : T'' \longrightarrow \mathbb{B}$, $\underline{n} : T'' \longrightarrow \mathbb{N}$, and $t \in T'$. Now define $\gamma_2 : \underline{S}_2 \longrightarrow [\underline{A}_3^+ \longrightarrow \underline{A}_2]$ $\gamma_2(\mathrm{Z}) = \theta$, $\gamma_2(\underline{\mathrm{Z}}) = \Theta$, and for other $s \in \underline{S}_2$ let $\gamma_2(s)$ be the identity function on $(\underline{A}_3)_s = (\underline{A}_2)_s$. It is not difficult to prove that $\gamma_2(s) : \underline{A}_3^{w_2(s)} \xrightarrow{\text{onto}} \underline{A}_1^s$ for each $s \in \underline{S}_2$, and thus $\underline{A}_2$ is $\underline{A}_3$-coded by γ_2 via w_2; we leave the details as an exercise.

To show that $\underline{A}_2$ is $\underline{A}_3$-generated is straightforward: the only constant of $\underline{A}_2$ which does not generate itself in $\underline{A}_3$ is $0 \in \mathbb{Z}$. However, it is easy to see that $\rho_{\mathrm{zero}} = <\mathrm{true}, \mathrm{zero}_N> \in T(\underline{\Sigma}_3)_{\mathrm{BN}}$ generates $0 \in \mathbb{Z}$. We conclude that $\underline{A}_2$ is $\underline{A}_3$-generated by $Gn_2 \subseteq T(\underline{\Sigma}_3)$ defined by $Gn_2 = <\rho_c : c \in \underline{\Sigma}_2>$ where

$$\rho_c = \begin{cases} \rho_{\mathrm{zero}} & \text{if } c = \mathrm{zero} \in (\underline{\Sigma}_2)_{\lambda,\mathrm{T}} \\ c & \text{otherwise} \end{cases}$$

Finally, we must show that $\underline{A}_2$ is $\underline{A}_3$-tracked. It is not difficult to check that all the standard operations of $\underline{A}_2$ are tracked by themselves; the remaining operations to be tracked are $\mathrm{eval}_{\mathrm{Z}}^{\underline{A}_3} = eval^{\mathbb{Z}}_{A_3}$ and the operations that specify the modules of N_1, N_2, and N_3.

Let us consider stream evaluation first. Let Σ_3 be the signature of A_3, and let α_{eval} be defined by $\alpha_{eval} = <\alpha_1,\alpha_2>$ where

$$\alpha_1 = eval_B \circ <U_1^{TB'N},U_2^{TB'N}>$$

and

$$\alpha_2 = eval_B \circ <U_1^{TB'N},U_3^{TB'N}>$$

Then $\alpha_{eval} \in PR(\underline{\Sigma}_3)$ with arity $(T\underline{B}'\underline{N},B'N)$ which is as required since $eval_Z \in (\underline{\Sigma}_2)_{TZ,Z}$ and $w_2(T\underline{Z}) = T\underline{B}'\underline{N}$ and $w_2(Z) = B'N$.

To show that α_{eval} tracks $eval_Z^{\underline{A}_2}$, choose $t \in T''$, $\underline{b} : T'' \longrightarrow \mathbb{B}$, and $\underline{n} : T'' \longrightarrow \mathbb{N}$, and calculate as follows:

$$\begin{aligned}\gamma_2(Z)([\![\alpha_{eval}]\!]_{\underline{A}_3}(t,\underline{b},\underline{n})) &= \gamma_2(Z)(\underline{b}(t),\underline{n}(t)) \\ &= \theta(\underline{b}(t),\underline{n}(t)) \end{aligned} \tag{90}$$

However,

$$\begin{aligned} eval_Z^{\underline{A}_2}(\gamma_2(T\underline{Z})(t,\underline{b},\underline{n})) &= eval_Z^{\underline{A}_2}(\gamma_2(T)(t),\gamma_2(\underline{Z})(\underline{b},\underline{n})) \\ &= eval_Z^{\underline{A}_2}(t,\Theta(\underline{b},\underline{n})) \\ &= \Theta(\underline{b},\underline{n})(t) \\ &= \theta(\underline{b}(t),\underline{n}(t)) \\ &= \gamma_2(Z)([\![\alpha_{eval}]\!]_{\underline{A}_3}(t,\underline{b},\underline{n})) \end{aligned}$$

(from (90)). Thus α_{eval} tracks $eval_Z^{\underline{A}_2}$ as required.

Now let us construct an $\underline{A}_3$-tracking of $\sigma_{i,1}^{\underline{A}_2}$ (for $i = 1,2,3$). Afterwards we will conclude the $\underline{A}_3$-implementation of $\underline{A}_2$ by constructing $\underline{A}_3$-trackings for $\sigma_{i,2}^{\underline{A}_2}$ (for $i = 2,3$).

Let $\Omega_{i,1}$ be the signature of $B_{i,1}$ (for $i = 1,2,3$); then there exists $\delta_{N_{i,1}} \in PR(\Omega_{i,1})$ such that $[\![\delta_{N_{i,1}}]\!]_{B_{i,1}} = f_{N_{i,1}}$ by Lemma 3.4.4. Now let $\alpha_{\lambda_{i,1}}$ be defined by

$$\alpha_{\lambda_{i,1}} = succ_{i,1} \circ zero_{i,1}^{B'N}$$

(Here '$zero_{i,1}$' and '$succ_{i,1}$' are the names of $T_{i,1}$'s zero and successor function respectively.) Then $\alpha_{\lambda_{i,1}} \in PR(\Omega_{i,1})$ with arity $(B'N,T_{i,1})$. Furthermore, it should be clear that $[\![\alpha_{\lambda_{i,1}}]\!]_{B_{i,1}} = \lambda_{i,1}$ $(i = 1,2,3)$.

Let $\iota_{i,1}$ be defined by

$$\iota_{i,1} = \delta_{N_{i,1}} \circ <\alpha_{\lambda_{i,1}},U_1^{B'N},U_2^{B'N},true^{B'N},zero^{B'N}>$$

Then $\iota_i \in PR(\Omega_i)_{B'N,B'N}$ it is easy to see that

$$(\forall b \in \mathbb{B})(\forall n \in \mathbb{N})\ (\ \theta([\![\iota_{i,1}]\!]_{B_{i,1}}(b,n)) = \theta(f_{N_{i,1}}(\lambda_i(b,n),b,n,\zeta_{i,1})) = \sigma_{i,1}^{\underline{A}_3}(\theta(b,n))\ ) \tag{91}$$

(using (84)). However, since $B_{i,1} = \underline{A}_3|_{\Omega_{i,1}}$, we have $\iota_{i,1} \in PR(\underline{\Sigma}_3)_{B'N,B'N} = PR(\underline{\Sigma}_3)_{w_3(Z),w_3(Z)}$ and $[\![\iota_{i,1}]\!]_{\underline{A}_3} = [\![\iota_{i,1}]\!]_{B_{i,1}}$. Thus from (91) we have (using the facts that $\sigma_{i,1}^{\underline{A}_3} = \sigma_{i,1}^{\underline{A}_3}$ and $\gamma_2(Z) = \theta$)

$$\gamma_2(Z)([\![\iota_{i,1}]\!]_{\underline{A}_3}(b,n)) = \sigma_{i,1}^{\underline{A}_3}(\gamma_2(Z)(b,n))$$

for each $(b,n) \in \mathbb{B} \times \mathbb{N}$; that is, $\iota_{i,1}$ is an $\underline{A}_3$-tracking of $\sigma_{i,1}^{\underline{A}_2}$.

□

Now let us construct an $\underline{A}_3$-tracking of $\sigma_{i,2}^{\underline{A}_2}$ (for $i=2,3$).

Let $\Omega_{i,2}$ be the signature of $B_{i,2}$ (for $i=2,3$); then there exists $\delta_{N_{i,2}} \in \mathrm{PR}(\Omega_{i,2})$ such that $[\![\delta_{N_{i,2}}]\!]_{B_{i,2}} = f_{N_{i,2}}$ by Lemma 3.4.4. Now let $\alpha_{\lambda_{i,2}}$ be defined by

$$\alpha_{\lambda_{i,2}} = \mathrm{succ}_{i,2} \circ \mathrm{succ}_{i,2} \circ \mathrm{succ}_{i,2} \circ \mathrm{zero}_{i,2}^{\mathrm{B'NB'N}}$$

(Here '$\mathrm{zero}_{i,2}$' and '$\mathrm{succ}_{i,2}$' are the names of $T_{i,2}$'s zero and successor function respectively.) Then $\alpha_{\lambda_{i,2}} \in \mathrm{PR}(\Omega_{i,2})$ with arity $(\mathrm{B'NB'N}, \mathrm{T}_{i,2})$. Furthermore, it should be clear that $[\![\alpha_{\lambda_{i,2}}]\!]_{B_{i,2}} = \lambda_{i,2}$ $(i=2,3)$.

Let $\iota_{i,2}$ be defined by

$$\iota_{i,2} = \delta_{N_{i,2}} \circ < \alpha_{\lambda_{i,2}}, \mathrm{U}_1^u, \mathrm{U}_2^u, \mathrm{U}_3^u, \mathrm{U}_4^u, \mathrm{true}^u, \mathrm{true}^u, \mathrm{true}^u, \mathrm{true}^u, \mathrm{true}^u, \mathrm{true}^u, \mathrm{zero}^u, \mathrm{zero}^u, \mathrm{zero}^u >$$

where $u = \mathrm{B'NB'N} \in \underline{S}_3^4$. Then $\iota_{i,2} \in \mathrm{PR}(\Omega_{i,2})_{u,\mathrm{B'N}}$ and it is easy to see that

$$(\forall b_1, b_2 \in \mathbb{B})(\forall n_1, n_2 \in \mathbb{N}) \qquad (92)$$
$$\theta([\![\iota_{i,2}]\!]_{B_{i,2}}(b_1,n_1,b_2,n_2)) = \theta(f_{N_{i,2}}(\lambda_{i,2}(b_1,n_1,b_2,n_2), b_1, n_1, b_2, n_2, \zeta_{i,2})) = \sigma_{i,2}^{\underline{A}_2}(\theta(b_1,n_1), \theta(b_2,n_2))$$

(using (89)). However, since $B_{i,2} = \underline{A}_3|_{\Omega_{i,2}}$, we have $\iota_{i,2} \in \mathrm{PR}(\underline{\Sigma}_3)_{u,\mathrm{B'N}} = \mathrm{PR}(\underline{\Sigma}_3)_{w_2(\mathrm{ZZ}),w_2(\mathrm{Z})}$ and $[\![\iota_{i,2}]\!]_{\underline{A}_3} = [\![\iota_{i,2}]\!]_{B_{i,2}}$. Thus from (92) we have (again using $\sigma_{i,2}^{\underline{A}_2} = \sigma_{i,2}^{\underline{A}_2}$ and $\gamma_2(\mathrm{Z}) = \theta$)

$$(\forall b_1, b_2 \in \mathbb{B})(\forall n_1, n_2 \in \mathbb{N})$$
$$\gamma_2(\mathrm{Z})([\![\iota_{i,2}]\!]_{\underline{A}_3}(b_1,n_1,b_2,n_2)) = \sigma_{i,2}^{\underline{A}_2}(\gamma_2(\mathrm{ZZ})(b_1,n_1,b_2,n_2))$$

Thus $\iota_{i,2}$ is an $\underline{A}_3$-tracking of $\sigma_{i,2}^{\underline{A}_2}$.

Now define $Tr_2 \subseteq \mathrm{PR}(\underline{\Sigma}_3)$ by $Tr_2 = <\alpha_\sigma : \sigma \in \underline{\Sigma}_2>$ where

$$\alpha_\sigma = \begin{cases} \iota_{i,j} & \text{if } \sigma = \sigma_{i,j} \text{ for } i=1 \text{ and } j=1, \text{ or } i \in \{2,3\} \text{ and } j \in \{1,2\} \\ \alpha_{\mathrm{eval}} & \text{if } \sigma = \mathrm{eval}_2 \\ \sigma & \text{otherwise} \end{cases}$$

Then $\underline{A}_2$ is $\underline{A}_3$-tracked by Tr_2, and we conclude that $I_2 = <\gamma_2, Gn_2, Tr_2>$ is an $\underline{A}_3$-implementation of $\underline{A}_2$.

Complexity. Let P_2 and P_3 be uniform performance measures for $\underline{A}_2$ and $\underline{A}_3$ respectively. Then as in the calculation of the complexity of I_1, to calculate the complexity of I_2, we only need to show that complexity of the tracking with the worst execution time is bounded by a constant to conclude that I_2 is a linear implementation. The scheme $\iota_{i,2}$ is the most expensive: it is not difficult to check (using Lemma 3.5.5) that for each $a \in (\mathbb{B} \times \mathbb{N})^2$

$$\lambda_{P_3}(\iota_{i,2})(a) = 2 + 2.4 + 4 = 12$$

Thus I_2 is a linear implementation of $\underline{A}_2$.

8.5.4 Applying the Hierarchy Theorem.

In this section we will use Hierarchy Theorem 8.4.3 to verify that M' is a correct implementation of FIR.

First let us make sure we know what it is that should be 'correct' about M'.

Observe that the compiler $c = c_2 \circ c_1$ constructed in the proof of Theorem 8.4.3 is a 'two-pass' compiler: given $\alpha \in PR(\underline{\Sigma}_1)$, on the first pass c (or c_1) will first substitute each scheme 'ι_i' for every occurrence of 'σ_i' in α yielding $c_1(\alpha)$; on the second pass c (or c_2) will substitute each scheme 'ι_{ij}' for every occurrence of 'σ_{ij}' in $c_1(\alpha)$ yielding $c_2(c_1(\alpha)) = c(\alpha)$. Since the schema ι_i and ι_{ij} formalise the static execution of the substituted networks, then the expression '$[\![c(\alpha_{FIR})]\!]_{\underline{A}_3}$' is the formal expression for executing M' (in the same way that expression '$[\![\alpha_{FIR}]\!]_{\underline{A}_1}$' is a formal expression for executing FIR).

Now suppose that the input to FIR is $\underline{z} = (\underline{z}_1, \underline{z}_2, \underline{z}_3) : T \longrightarrow \mathbb{Z}^3$, and the initial values are $x = (x_1, x_2, x_3) \in \mathbb{Z}^3$. Then the state of FIR at time $t \in T$ is $[\![\alpha_{FIR}]\!]_{\underline{A}_1}(t, \underline{z}, x) \in \mathbb{Z}^3$

To analyse the correctness of M', we must execute it on streams and initial values that correspond to those given to FIR. Let

$$\Phi : [T'' \longrightarrow \mathbb{B} \times \mathbb{N}] \longrightarrow [T \longrightarrow \mathbb{Z}]$$

be defined by

$$\Phi(\underline{b}, \underline{n})(t) = \theta(\underline{b}(t), \underline{n}(t))$$

for each $\underline{b} : T'' \longrightarrow \mathbb{B}$, $\underline{n} : T'' \longrightarrow \mathbb{N}$, and each $t \in T$. Then to analyse the correctness of M' it is appropriate to compare $[\![\alpha_{FIR}]\!]_{\underline{A}_1}(t, \underline{z}, x)$ with $[\![c(\alpha_{FIR})]\!]_{\underline{A}_3}(t, \underline{a}, y) \in (\mathbb{B} \times \mathbb{N})^3$ where $\underline{a} = (\underline{a}_1, \underline{a}_2, \underline{a}_3) : T'' \longrightarrow (\mathbb{B} \times \mathbb{N})^3$ is such that $\underline{z}_i = \Phi(\underline{a}_i)$ for $i = 1,2,3$, and $y = (y_1, y_2, y_3) \in (\mathbb{B} \times \mathbb{N})^3$ is such that $x_i = \theta(y_i)$ for $i = 1,2,3$. Now, since the state of FIR at each time $t \in T$ is an element of $\mathbb{Z}^3$ and the state of M' at each time $t \in T''$ is an element of $(\mathbb{B} \times \mathbb{N})^3$, the appropriate correctness condition is

$$(\theta \times \theta \times \theta)([\![c(\alpha_{FIR})]\!]_{\underline{A}_3}(t, \underline{a}, y)) = [\![\alpha_{FIR}]\!]_{\underline{A}_1}(t, \underline{z}, x)$$

(where '$\times$' is the direct product operator; see Section 8.2.4). However, given the relationships between $\underline{a}$ and $\underline{z}$ and between y and x, an equivalent correctness condition is

$$(\forall t \in T'')(\forall \underline{a} : T'' \longrightarrow (\mathbb{B} \times \mathbb{N})^3)(\forall y \in (\mathbb{B} \times \mathbb{N})^3) \qquad (93)$$
$$(\theta \times \theta \times \theta)([\![c(\alpha_{FIR})]\!]_{\underline{A}_3}(t, \underline{a}, y)) = [\![\alpha_{FIR}]\!]_{\underline{A}_1}(t, (\Phi \times \Phi \times \Phi)(\underline{a}), (\theta \times \theta \times \theta)(y))$$

It is trivial to prove (93) by Hierarchy Theorem 8.4.3: by the theorem, since $\underline{A}_i$ is $\underline{A}_{i+1}$-coded by γ_i via w_i for $i = 1,2$, we know that for each $u, v \in \underline{S}_1^+$, and for every $\alpha \in PR(\underline{\Sigma}_1)_{u,v}$, we have $c(\alpha) \in PR(\underline{\Sigma}_3)_{w(u),w(v)}$ where $w = w_2 \circ w_1$, and for every $a \in \underline{A}_3^{w(u)}$ we have

$$\gamma(v)([\![c(\alpha)]\!]_{\underline{A}_3}(a)) = [\![\alpha]\!]_{\underline{A}_1}(\gamma(u)(a)) \qquad (94)$$

where $\gamma = \gamma_1 \circ \gamma_2$ (cf. Notation 8.2.13). Now, $\alpha_{FIR} \in PR(\underline{\Sigma}_1)_{TDD,D}$ (where D=ZZZ), and so $c(\alpha_{FIR}) \in PR(\underline{\Sigma}_3)_{w(TDD),w(D)}$. If we now let $E = B'NB'NB'N \in \underline{S}_3^6$, then $\underline{A}_3^E = (\mathbb{B} \times \mathbb{N})^3$ and $\underline{A}_3^E = [T'' \longrightarrow (\mathbb{B} \times \mathbb{N})^3]$. Furthermore,

$$w(D) = w(ZZZ) = w_2(w_1(ZZZ)) = w_2(ZZZ) = w_2(Z)w_2(Z)w_2(Z) = B'NB'NB'N = E$$

and so $w(TDD) = TEE$; that is, $c(\alpha_{FIR}) \in PR(\underline{\Sigma}_3)_{TEE,E}$ and so $[\![c(\alpha_{FIR})]\!]_{\underline{A}_3}$ is a mapping of functionality

$$[\![c(\alpha_{FIR})]\!]_{\underline{A}_3} : T'' \times [T'' \longrightarrow (\mathbb{B} \times \mathbb{N})^3] \times (\mathbb{B} \times \mathbb{N})^3 \longrightarrow (\mathbb{B} \times \mathbb{N})^3$$

Hence by the correctness of c (see (94)) we have

$$\gamma(D)([\![c(\alpha_{FIR})]\!]_{\underline{A}_3}(a)) = [\![\alpha_{FIR}]\!]_{\underline{A}_1}(\gamma(TDD)(a)) \qquad (95)$$

for each $a \in \underline{A}_3^{TEE}$, that is for each $(t, \underline{a}, y) \in T'' \times [T'' \longrightarrow (\mathbb{B} \times \mathbb{N})^3] \times (\mathbb{B} \times \mathbb{N})^3$.

Now, it is not difficult to check that $(\theta\times\theta\times\theta)=\gamma(D)$, $\gamma(T)$ is the identity function, and $(\Phi\times\Phi\times\Phi)=\gamma(\underline{D})$, and so

$$(\theta\times\theta\times\theta)(\llbracket c(\alpha_{FIR})\rrbracket_{\underline{A}_3}(t,\underline{a},y)) = \gamma(D)(\llbracket c(\alpha_{FIR})\rrbracket_{\underline{A}_3}(t,\underline{a},y))$$
$$= \llbracket\alpha_{FIR}\rrbracket_{\underline{A}_1}(\gamma(T\underline{D}D)(t,\underline{a},y))$$

(by (95))

$$= \llbracket\alpha_{FIR}\rrbracket_{\underline{A}_1}((\gamma(T)\times\gamma(\underline{D})\times\gamma(D))(t,\underline{a},y))$$
$$= \llbracket\alpha_{FIR}\rrbracket_{\underline{A}_1}(\gamma(T)(t),\gamma(\underline{D})(\underline{a}),\gamma(D)(y))$$
$$= \llbracket\alpha_{FIR}\rrbracket_{\underline{A}_1}(t,(\Phi\times\Phi\times\Phi)(\underline{a}),(\theta\times\theta\times\theta)(y))$$

Thus (93) holds as claimed.

Performance. From the performance part of Hierarchy Theorem 8.4.3, we know that for any performance measures P_1 and P_3 for $\underline{A}_1$ and $\underline{A}_3$ respectively, we have

$(\forall t\in T'')(\forall\underline{a}:T''\longrightarrow(\mathbb{B}\times\mathbb{N})^3)(\forall y\in(\mathbb{B}\times\mathbb{N})^3)$

$$\lambda_{P_3}(c(\alpha_{FIR}))(t,\underline{a},y)\leq\Psi(\lambda_{P_1}(\alpha)(\gamma(T\underline{D}D)(t,\underline{a},y)))$$

where the degree of Ψ is the product of the (non-zero) degrees of I_1 and I_2. We have shown that if if P_1 and P_3 are uniform, and if P_2 is a uniform performance measure for $\underline{A}_2$, then both I_1 and I_2 have degree 1, and thus $deg\,\Psi=1$. In other words, the complexity of M' (officially $\lambda_{P_3}(c(\alpha_{FIR}))$) is bounded by a constant multiple of the complexity of FIR (officially $\lambda_{P_1}(\alpha_{FIR})$). Notice that this informs us, given the computational equivalence of PR and FPIT, that the complexity of *simulating* M' is bounded by a constant multiple of the complexity of *simulating* FIR.

8.6 SOURCES.

This chapter contains (i) informal and formal analysis of top-down design for synchronous concurrent algorithms, and (ii) a theory of data type implementation, based on a generalised computability theory, and the development of compilers. Both are new and my own work.

In the case of (i), my colleague Dr. K. McEvoy has independently worked on a theory of hierarchical network decomposition based on graph substitutions: see McEvoy[1986]; apart from this we are not aware of any other theoretical work on the subject. In the case of (ii), data type implementation based on generalised computability theory is a subject I will discuss below; the compilers presented here are instances of 'semantics directed compiler generation' in the sense of the volume Jones[1980].

The generalisation of computability theory to algebraic structures begins with Fröhlich and Shepherdson[1956] and was subsequently developed by A. I. Mal'cev and M. O. Rabin; see Mal'cev[1961] and Rabin[1960] respectively. Mal'cev defined a (single-sorted) algebra A to be computable if (a) A can be coded via a Gödel numbering $\alpha:\mathbb{N}\longrightarrow A$, (b) each operation of A can be tracked by a recursive function on $\mathbb{N}$, and (c) the relation $\equiv_\alpha$ defined by $n\equiv_\alpha m$ iff $\alpha(n)=\alpha(m)$ is recursive. He also defined the notion of a semicomputable and co-semicomputable algebra by taking $\equiv_\alpha$ to be r.e. and co-r.e. respectively; and the primitive recursive algebras by insisting that all the recursive machinery be primitive recursive. Independently, Rabin had an equivalent definition of computability applicable to

fields.

The definitions of Mal'cev were imported into the theory of programming languages by the work of J. A. Bergstra and J. V. Tucker on the scope and limits of algebraic specifications for data types (under both initial and final algebra semantics). See the comprehensive survey Meseguer and Goguen[1985] for an exposition. I have found Bergstra and Tucker[1986] to be a particularly useful guide to the subject.

We have chosen to work with algebras that are primitive recursively computable since this is appropriate for synchronous algorithms. However, we may also consider more general notions: on taking the classes of inductively definable or inductively cov definable functions discussed in Section 3.6.1, and adapting Definition 8.3.6, the definitions of inductively definable and cov definable algebras A over an algebra B can be made *mutato mutandis*.

CHAPTER 9
CONCLUDING REMARKS

In this final chapter I will summarise what we have seen, point out some of the more interesting matters arising, and indicate some directions for future work.

In Chapter 2 synchronous algorithms and the principal specification technique, 'by value functions' were introduced. In Chapter 3 the system PR was introduced, and it was shown that the specification of synchronous algorithms is formalisable in PR. As a language PR can be extended with respect to (i) its use as a notation for synchronous algorithms, and (ii) computability theory. I will discuss (i) now and (ii) below with the discussion of Chapters 6 and 7.

With respect to PR as a notation for synchronous algorithms, the syntax of PR owes its current rather austere form to the fact that for the work presented here to be of practical use, the notation should be machine-readable. It is also possible that PR can be used as a programming notation for synchronous algorithms in which case the notation should certainly be sweetened with constructs that are tailored to expressing synchronous algorithms in general, and systolic algorithms in particular. Actually, some work on making PR more usable has already been done. In the version of PR that has been implemented by my colleague A. R. Martin (for the purpose of implementing the compiler from PR to FPIT), one may declare names for Cartesian products of domains and structure PR schema in a nested way. For example, the scheme $*(U_1^N, succ \circ U_3^{NNN})$ which denotes addition on the natural numbers, can be entered thus:

```
domain
     nat_pair    = nat * nat;
     nat_triple = nat * nat * nat

function add : nat_pair -> nat

    function value_at_zero : nat -> nat
    is U[ nat, 1 ]

    function add_one_to_third_argument : nat_triple -> nat
    is SUCC . U[ nat_triple, 3 ]

is *( value_at_zero, add_one_to_third_argument )
```

The ability to structure schema in this way is obviously advantageous when experimenting with large or complicated schema.

One other point that arises in Chapter 3 is the non-primitive-recursiveness of the specification G_N. As previously noted, the situation can be rectified by adding function abstraction to PR as a new function building tool. This will lead to a new theory of functionals that may well be of mathematical interest, and be of use to the theory of synchronous algorithms.

Also in Chapter 3, performance measures were introduced to provide a general account of the complexity of algebraic data and operations upon which I built an account of the performance of PR

schema. However, to a hardware designer 'performance' is is a broader notion encompassing such concepts as 'area' and 'power consumption' for example. The refinement of theoretical estimates of performance and their reconciliation with electrical models of circuits is an important subject.

In Chapters 4 and 5, the PR formalism was tested out by verifying the sorters of Chapter 2 and applying the specification methodology to more examples. As far as examples are concerned, future work will involve finding new algorithms to test out the expressive power of PR. Work in this direction is already in progress: apart from other algorithms that I have specified (but not had the space to include here), my colleagues K. M. Hobley and S. M. Eker are using PR to specify synchronous algorithms that arise in digital hardware for signal processing and graphics algorithms respectively.

With respect to the verification of PR-specified synchronous algorithms, the correctness proofs given here used conventional mathematical rigour. Naturally, the reason for this is that the approach to the formalisation of synchronous algorithms given here is 'model based' rather than 'axiomatic'. Questions arise as to the kinds of proof systems that are appropriate for verification based on axioms, and ultimately for mechanical verification. K. M. Hobley is working on the application of suitable first-order many-sorted logical languages, based on the work of Tucker and Zucker[1987].

One noteworthy idea that arises from the verification of the sorters is the fact that to prove a PCA sorts *every* input vector one only needs to show that it sorts *one particular* vector (a 'reverse' vector). This fact suggests the idea of a *critical point* for an algorithm; that is, a single point in the input space of the algorithm at which the correctness of the algorithm implies the correctness of the algorithm at all points. Perhaps a search for critical points for algorithms in general would be fruitful.

In Chapter 6 the language FPIT was introduced, and in Chapter 7 the computational equivalence of PR and FPIT was proved. The use of compilers in the proof underwrites the idea of a design environment in which one can flip between alternative, but equivalent, representations of a given synchronous algorithm. The use of FPIT as an independent tool for programming synchronous algorithms has been investigated by A. R. Martin in connection with this environment: see Martin and Tucker[1987]. The design and implementation of the environment has been undertaken by A. R. Martin also. I hope to contribute to this project, especially with respect to the programming of streams and the invention of compilers that produce readable code.

I will now return to the matter of extensions to PR and FPIT. Of course, on adding a new feature to PR (or FPIT), to maintain computational equivalence one must either define the new feature in terms of existing ones, or add an corresponding feature to FPIT (or PR). Extensions that make PR more usable are of the first kind. As far as the second kind of extensions are concerned, the natural generalisations to consider are adding the least number operator and course-of-values recursion to PR, and correspondingly, the while-statement and arrays to FPIT. Both of these are considered in Tucker and Zucker[1987] where the proof of computational equivalence (not including equivalence of performance) is sketched(!). However, in the context of the work presented here, these generalisations must be carefully considered: the while-statement seems to be intimately connected with the programming of streams, and arrays can be used to model modules with finite memory; a convincing analysis of these ideas is far from clear

however.

Finally, in Chapter 8, a theory of top-down design for synchronous algorithms was presented. This theory was based on a new definition of what it means for one algebra to be 'computable' over another. There are a number of semantic issues that arise from the definition of network systems that require investigation. For example, the semantics of a network system are built 'bottom-up' from the lowest level synchronous networks, whereas the compiler that produces the PR representation of the system works top-down. Another point, strongly related to the last, is whether the clock T' of a network system M should be the same as, or different from, the clock T of the synchronous network N from which M was obtained by network-for-module substitution.

A final possible area of future study is an investigation of the implications of replacing PR in Definition 8.3.6 by any of more general classes of computable functions mentioned above.

REFERENCES

Aarts and Korst[1987].

E. H. L. Aarts and J. H. M. Korst, "Boltzmann Machines and their Applications", pp. 34-50 in *Proc. Parallel Architectures and Languages Europe Vol. 1*, ed. J. W. de Bakker, A. J. Nijman, and P. C. Treleaven,Springer-Verlag (LNCS 258), Berlin (1987).

Akl[1985].

S. G. Akl, *Parallel Sorting Algorithms*, Academic Press (1985).

Asveld and Tucker[1982].

P. R. J. Asveld and J. V. Tucker, "Complexity Theory and the Operational Structure of Algebraic Programming Systems", *Acta Informatica* **17** pp. 451-476 (1982).

Backhouse[1983].

R. Backhouse, "Specification and Proof of a Regular Language Recogniser in Synchronous CCS", Tech. Rpt. CSM-53, University of Essex (1983).

Backus[1978].

J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Comm. ACM* **21**(8) pp. 613-641 (1978).

Bakker[1980].

J. W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall International, London (1980).

Barron et al[1963].

D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey, "The Main Features of CPL", *The Computer Journal* **6** pp. 134-143 (1963).

Barros and Johnson[1983].

J. C. Barros and B. W. Johnson, "Equivalence of The Arbiter, The Synchronizer, The Latch, and The Inertial Delay", *IEEE Trans. Comp.* C-32(7) pp. 603-614 (1983).

Batcher[1968].

K. E. Batcher, "Sorting Networks and Their Applications", pp. 307-314 in *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J. (1968).

Baudet[1983].

G. Baudet, "Design and Complexity of VLSI Algorithms", in *Foundations of Computer Science IV*, ed. J. W. de Bakker, (1983).

Bergstra and Klop[1986].

J. A. Bergstra and J. W. Klop, "Algebra of Communicating Processes", pp. 89-138 in *Proc. CWI Symp. Mathematics and Computer Science*, ed. J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra,North-Holland, Amsterdam (1986).

Bergstra and Tucker[1986].

J. A. Bergstra and J. V. Tucker, "Algebraic Specifications of Computable and Semicomputable Data Types", Tech. Rept. 2.86, Centre for Theoretical Computer Science, University of Leeds (1986).

Beyers[1981].

J. Beyers, "A 32-bit VLSI Chip", in *Digest of Technical Papers, Proc IEEE Int. Solid-State Circ. Conf.*, (1981).

Bitton et al[1984].

D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting", *Computing Surveys* 16(3) pp. 287-318 (1984).

Bochmann[1980].

G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example", Tech. Rept., Stanford University (1980).

Bochmann[1982].

G. V. Bochmann, "Hardware Specification with Temporal Logic: An Example", *IEEE Trans. Comp.* C-31(3) pp. 223-331 (1982).

Boyer and Moore[1979].

R. S. Boyer and J. Strother Moore, *A Computational Logic*, Academic Press (1979).

Brookes[1983].

S. D. Brookes, "Reasoning About Synchronous Systems", Tech. Rept. CMU-CS-84-145, Carnegie-Mellon University (1983).

Bryant[1984].

R. E. Bryant, "A Switch-Level Simulator for MOS Digital Systems", *IEEE Trans. Comp.* C-33(2) pp. 160-177 (1984).

Bryant[1986].

R. E. Bryant, "Can a Simulator Verify a Circuit?", pp. 160-177 in *Formal Aspects of VLSI*, ed. G. J. Milne and P. A. Subrahmanyam,North-Holland, Amsterdam (1986).

Burstall, MacQueen, and Sanella[1980].

R. M. Burstall, D. B. MacQueen, and D. T. Sanella, "HOPE: An Experimental Applicative Language", Tech. Rept. CSR-62-80, University of Edinburgh (1980).

Camilleri, Gordon, and Melham[1986].

A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification using Higher-Order Logic", Tech. Rept. 91, Computer Laboratory., University of Cambridge (1986).

Cappello and Steiglitz[1984].

P. R. Cappello and K. Steiglitz, "Unifying VLSI Array Design with Linear Transformations of Space-Time", in *Advances in Computing Research, 2: VLSI Theory*, ed. F. Preperata, (1984).

Chandy and Misra[1985].

K. M. Chandy and J. Misra, "Parallelism and Programming: The Proper Perspective", Tech. Rpt. Department of Computer Studies, University of Texas at Austin (1985).

Chandy and Misra[1986a].

K. M. Chandy and J. Misra, "Systolic Algorithms as Programs", *Distributed Computing* 1(3)(1986).

Chandy and Misra[1986b].

M. Chandy and J. Misra, "An example of stepwise refinement of distributed programs: quiescence detection", *ACM Trans. on Programming Languages and Systems* 8 pp. 326-343 (1986).

Church[1940].

A. Church, "A Formulation of the Simple Theory of Types", *J. Symbolic Logic* 5(1940).

Codd[1968].

E. F. Codd, *Cellular Automata*, Academic Press (1968).

Cohn[1987].

A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the First Levels", in *Proc. Hardware Verification Workshop, Calgary 1987 (to appear)*, (1987).

Conway, Berlekamp, and Guy[1982].

J. Conway, E. R. Berlekamp, and R. K. Guy, *Winning Ways*, Academic Press (1982).

Cullyer[1987].

W. J. Cullyer, "Implementing Safety-Critical Systems: the VIPER Microprocessor", in *Proc. Hardware Verification Workshop, Calgary 1987 (to appear)*, (1987).

Dedekind[1888].

R. Dedekind, *Was Sind und Was Sollen die Zahlen?*, Braunschweig (1888).

Delosme and Ipsen[1987a].

J. M. Delosme and I. Ipsen, "Overview of SAGA and CONDENSE", Tech. Rept., Yale University (1987).

Delosme and Ipsen[1987b].

J. M. Delosme and I. Ipsen, "Efficient Systolic Arrays for the Solution of Toeplitz Systems: An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI", in *Systolic Arrays*, ed. W. Moore, A. McCabe, and R. Urquhart,Adam Hilger (1987).

Dew and Tucker[1983].

P. M. Dew and J. V. Tucker, "An Experimental Study of a Timing Assumption in VLSI Complexity Theory", Tech. Rpt. 168, Department of Computer Studies, University of Leeds (1983).

Dijkstra[1976].

Edsgar W. Dijkstra, *A Discipline of Programming*, Prentice-Hall International, London (1976).

Ehrig and Mahr[1985].

H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specification I - Examples and Initial Semantics", in *EATCS Monograph Series Vol. 6*, Springer-Verlag, Berlin (1985).

Eker[1986].

S. M. Eker, "An Investigation into the Role of PR() in Laying Out Synchronous Systems", Final-Year Project Report, Department of Computer Studies, University of Leeds (1986).

Eker and Tucker[1987].

S. M. Eker and J. V. Tucker, "Specification, Derivation, and Verification of Concurrent Line Drawing Algorithms and Architectures", in *Theoretical Foundations for Computer Graphics and Computer Aided Design*, ed. R. A. Earnshaw,Springer-Verlag (in preparation), Berlin (1987).

Engler[1968].

E. Engler, "Algebraic Properties of Structures", *Mathematical Systems Theory* 1 pp. 183-195 (1968).

Eveking[1985].

H. Eveking, "The Application of CHDLs to the Abstract Specification of Hardware", pp. 167-178 in *Proc. 7th Int. Conf. Computer Hardware Description Languages and Their Applications*, ed. C. J. Koomen and T. Moto-oka,North-Holland, Amsterdam (1985).

Fischer and Kung[1985].

A. L. Fischer and H. T. Kung, "Special-Purpose VLSI Architectures: General Discussion and a Case Study", pp. 153-169 in *VLSI and Modern Signal Processing*, ed. S. Y. Kung, H. J. Whitehouse, and T. Kailath,Prentice-Hall International, London (1985).

Fortes and Moldovan[1986].

J. Fortes and D. I. Moldovan, "Partitioning and Mapping Algorithms onto Fixed Size VLSI Algorithms", *IEEE Trans. Comp.* C-35(1) pp. 1-12 (1986).

Fröhlich and Shepherdson[1956].

A. Frohlich and J. C. Shepherdson, "Effective Procedures in Field Theory", *Philosophical Transactions of the Royal Society, London, Series A* 248 pp. 407-432 (1956).

Gachet, Joinnault, and Quinton[1987].

P. Gachet, B. Joinnault, and P. Quinton, "Synthesising Systolic Arrays using DIASTOL", pp. 25-36 in *Systolic Arrays*, ed. W. Moore, A. McCabe, and R. Urquhart,Adam Hilger (1987).

Goguen, Thatcher, and Wagner[1978].

J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", pp. 80-144 in *Current Trends in Programming Methodology, IV, Data Structuring*, ed. R. T. Yeh,Prentice-Hall International, London (1978).

Gordon[1986].

M. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying

Hardware", pp. 153-177 in *Formal Aspects of VLSI*, ed. G. J. Milne and P. A. Subrahmanyam,North-Holland, Amsterdam (1986).

Gordon[1987].

M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", in *Proc. Hardware Verification Workshop, Calgary 1987 (to appear)*, (1987).

Gordon, Milner, and Wadsworth[1979].

M. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Springer-Verlag (LNCS 78), Berlin (1979).

Greibach[1975].

S. A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*, Springer-Verlag (LNCS 36), Berlin (1975).

Gries[1978].

D. Gries, "The Multiple Assignment Statement", *IEEE Transactions on Software Engineering* SE-4 pp. 89-93 (1978).

Guerra and Melhem[1986].

C. Guerra and R. G. Melhem, "Synthesising Non-uniform Systolic Designs", pp. 765-771 in *Proc. IEEE Conf. Parallel Processing*, (1986).

Hanna and Daeche[1984].

F. K. Hanna and N. Daeche, "The VERITAS Theorem-Prover and its Formal Specification", Tech. Rept., University of Kent (1984).

Hanna and Daeche[1986].

F. K. Hanna and N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study", pp. 179-213 in *Formal Aspects of VLSI*, ed. G. J. Milne and P. A. Subrahmanyam,North-Holland, Amsterdam (1986).

Harman and Tucker[1987].

N. A. Harman and J. V. Tucker, "The Formal Specification of a Digital Correlator I: Abstract User Process", Tech. Rpt. 9.87, Centre for Theoretical Computer Science, University of Leeds (1987).

Hennessy[1986].

M. Hennessy, "Proving Systolic Algorithms Correct", *ACM Transactions on Programming Languages and Systems* 8(3) pp. 344-387 (1986).

Hennie[1961].

F. C. Hennie, *Iterative Arrays of Logical Circuits*, MIT Press (1961).

Hoare[1985].

C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, London (1985).

Huang and Lengauer[1987].

C. H. Huang and C. Lengauer, "An Implemented Method for Incremental Systolic Design", pp. 160-177 in *Proc. Parallel Architectures and Languages Europe Vol. 1*, ed. J. W. de Bakker, A. J. Nijman, and P. C. Treleaven,Springer-Verlag (LNCS 258), Berlin (1987).

Hunt[1986].

W. A. Hunt, "The FM5801: A Verified Microprocessor", Tech. Rept. 47, University of Texas at Austin (1986).

Inmos[1984].

Inmos Limited, *Occam Programming Manual*, Prentice-Hall International, London (1984).

Jervis[1988].

C. A. Jervis, "On the Specification, Implementation, and Verification of Data Types", Ph.D. Thesis, Department of Computer Studies, University of Leeds (in preparation) (1988).

Johnson[1984].

S. D. Johnson, *Synthesis of Digital Circuits from Recursion Equations*, MIT Press (1984).

Johnsson and Cohen[1981].

L. Johnsson and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks", pp. 213-224 in *VLSI Systems and Computations*, ed. H. T. Kung, R. Sproull, and G. Steel,Springer-Verlag, Berlin (1981).

Jones[1980].

N. D. Jones, *Semantics Directed Compiler Generation*, Springer-Verlag (LNCS 259), Berlin (1980).

Joyce[1987].

J. Joyce, "Formal Verification and Implementation of a Microprocessor", in *Proc. Hardware Verification Workshop, Calgary 1987 (to appear)*, (1987).

Karp, Miller, and Winograd[1967].

R. M. Karp, R. Miller, and S. Winograd, "The Organisation of Computations for Uniform Recurrence Equations", *J. ACM* 14(3) pp. 563-590 (1967).

Knuth[1973].

D. E. Knuth, *The Art of Computer Programming (Vol. 3)*, Addison-Wesley (1973).

Kung[1982].

H. T. Kung, "Why Systolic Architectures?", *Computer* 15(1) pp. 37-46 (1982).

Kung and Leiserson[1979].

H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)", pp. 256-282 in *Sparse Matrix Proceedings 1978*, ed. I. S. Duff and G. W. Stewart, (1979).

Kung and Lin[1983].

H. T. Kung and W. T. Lin, "An Algebra for VLSI Algorithm Design", Tech. Rept. CMU-CS-84-

100, Carnegie-Mellon University (1983).

Kutzler and Lichtenberger[1983].

B. Kutzler and F. Lichtenberger, *Bibliography on abstract data types*, Springer-Verlag, Berlin (1983).

Lam and Mostow[1985].

M. Lam and J. Mostow, "A Transformational Model of VLSI Systolic Design", *Computer* 18(2) pp. 42-52 (1985).

Leiserson and Saxe[1983].

C. E. Leiserson and J. B. Saxe, "Optimising Synchronous Systems", *VLSI and Computer Systems* 1 pp. 41-68 (1983).

Li and Wah[1985].

G. J. Li and B. W. Wah, "The Design of Optimal Systolic Arrays", *IEEE Trans. Comp.* C-34(1) pp. 66-77 (1985).

Mal'cev[1961].

A. I. Mal'cev, "Constructive Algebras", *Russian Mathematical Surveys* 16 pp. 77-129 (1961).

Malachi and Owicki[1981].

Y. Malachi and S. Owicki, "Temporal Specification of Self-Timed Systems", pp. 203-212 in *VLSI Systems and Computations*, ed. H. T. Kung, R. Sproull, and G. Steel,Springer-Verlag, Berlin (1981).

Martin and Tucker[1987].

A. R. Martin and J. V. Tucker, "The Concurrent Assignment Representation of Synchronous Systems", pp. 365-386 in *Proc. Parallel Architectures and Languages Europe Vol. 2*, ed. J. W. de Bakker, A. J. Nijman, and P. C. Treleaven,Springer-Verlag (LNCS 259), Berlin (1987).

McCulloch and Pitts[1943].

W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics, Series* 2 5 pp. 115-133 (1943).

McEvoy[1986].

K. McEvoy, "A Formal Model for the Hierarchical Design of Synchronous and Systolic Algorithms", Tech. Rept. 7.86, Centre for Theoretical Computer Science, University of Leeds (1986).

Mead[1983].

C. A. Mead, "Structural and Behavioural Composition of VLSI", pp. 3-8 in *VLSI '83*, ed. F. Anceau and E. J. Aas,North-Holland, Amsterdam (1983).

Mead and Conway[1980].

C. A. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).

Meinke[1987].

K. Meinke, "A Graph-Theoretic Model of Synchronous Computation", Ph.D. Thesis, University

of Leeds (to appear) (1987).

Melhem and Rheinboldt[1984].

R. G. Melhem and W. C. Rheinboldt, "A Mathematical Model for the Verification of Systolic Networks", *SIAM J. Computing* 13(3) pp. 541-565 (1984).

Meseguer and Goguen[1985].

J. Meseguer and J. A. Goguen, "Initiality, Induction, and Computability", pp. 459-541 in *Algebraic Methods in Semantics*, ed. M. Nivat and J. Reynolds,Cambridge University Press (1985).

Meyer and Ritchie[1967].

A. Meyer and D. Ritchie, *The Complexity of Loop Programs*, Proc. 22nd. ACM National Conf. (1967).

Milne[1982].

G. J. Milne, "CIRCAL: A Calculus for Circuit Description", Tech. Rept. CSR-122-82, University of Edinburgh (1982).

Milne[1983].

G. J. Milne, "A Formal Basis for the Analysis of Circuit Timing", Tech. Rept. CSR-138-83, University of Edinburgh (1983).

Milne[1986].

G. J. Milne, "Design Transformation and Chip Planning", pp. 1-22 in *Formal Aspects of VLSI*, ed. G. J. Milne and P. A. Subrahmanyam,North-Holland, Amsterdam (1986).

Milner[1980].

R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag (LNCS 92), Berlin (1980).

Minsky and Pappert[1969].

M. Minsky and S. Pappert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press (1969).

Miranker and Winkler[1984].

W. L. Miranker and A. Winkler, "Spacetime Representations of Systolic Computational Structures", *Computing* 32 pp. 93-114 (1984).

Mishra and Clarke[1983].

B. Mishra and E. M. Clarke, "Automatic and Hierarchical Verification of Circuits using Temporal Logic", Tech. Rept. CMU-CS-83-155, Carnegie-Mellon University (1983).

Moldovan[1984].

D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays", pp. 158-164 in *Proc. IEE ICCD: VLSI in Computers*, (1984).

Mongenet and Perrin[1987].

C. Mongenet and G. R. Perrin, "Synthesis of Systolic Arrays for Inductive Problems", pp. 260-277 in *Proc. Parallel Architectures and Languages Europe Vol. 1*, ed. J. W. de Bakker, A. J.

Nijman, and P. C. Treleaven,Springer-Verlag (LNCS 258), Berlin (1987).

Morris[1973].

L. Morris, "Advice on Structuring Compilers and Proving Them Correct", pp. 144-152 in *Proc. ACM Symp. Principles of Programming Languages*, (1973).

Moskowski[1983].

B. C. Moskowski, "A Temporal Logic for Multi-Level Reasoning About Hardware", pp. 79-90 in *VLSI '83*, ed. F. Anceau and E. J. Aas,North-Holland, Amsterdam (1983).

Moskowski[1986].

B. C. Moskowski, *Executing Temporal Logic Programs*, Cambridge University Press (1986).

Neumann[1987].

J. von Neumann, *Papers of John von Neumann on Computing and Computing Theory*, MIT Press (1987).

Nielson[1984].

H. Riis Nielson, "Hoare Logics for the Run-Time Analysis of Programs", Tech. Rept. CST-20-84 (Ph.D. Thesis), Edinburgh University (1984).

Péter[1967].

R. Peter, *Recursive Functions*, Academic Press (1967).

Quinton[1987].

P. Quinton, "The Systematic Design of Systolic Arrays", pp. 261-302 in *Automata Networks in Computer Science*, ed. F. Fogelman Soulie, Y. Robert, and M. Tchuente,Manchester University Press (to appear) (1987).

Rabin[1960].

M. O. Rabin, "Computable Algebra, General Theory and the Theory of Computable Fields", *Transactions American Mathematical Society* 98 pp. 341-360 (1960).

Rajopadhye and Fujimoto[1987].

S. V. Rajopadhye and R. M. Fujimoto, "Systolic Array Synthesis by Static Analysis of Program Dependencies", pp. 295-310 in *Proc. Parallel Architectures and Languages Europe Vol. 1*, ed. J. W. de Bakker, A. J. Nijman, and P. C. Treleaven,Springer-Verlag (LNCS 258), Berlin (1987).

Rajopadhye, Purushothaman, and Fujimoto[1986].

S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On Synthesising Systolic Arrays form Recurrence Equations with Linear Dependencies", pp. 488-503 in *Proc. Sixth Ann. Conf. Foundations of Software Technology and Theoretical Computer Science*, ed. K. V. Nori,Springer-Verlag (LNCS 241), Berlin (1986).

Ramakrishnan, Fussel, and Silberschatz[1985].

I. V. Ramakrishnan, D. S. Fussel, and A. Silberschatz, "On Mapping Homogeneous Graphs on a Linear Array-processor Model", *IEEE Trans. Comp.* C-35 pp. 189-209 (1985).

Rem[1981].

M. Rem, "The VLSI Challenge: Complexity Bridling", pp. 65-73 in *VLSI '81*, ed. J. P. Gray,Academic Press (1981).

Rem[1983].

M. Rem, "Partially Ordered Computations, with Applications to VLSI Design", pp. 1-44 in *Foundations of Computer Science IV: Distributed Systems Part 2, Semantic and Logic*, ed. J. W. de Bakker, J. van Leeuwen,Mathematics Centre, Amsterdam (1983).

Rosenberg[1985].

A. L. Rosenberg, "References to the Literature on VLSI Algorithmics and Related Mathematical and Practical Issues", *Bulletin of the European Association for Theoretical Computer Science* 25(1985).

Seitz[1980].

C. L. Seitz, *System Timing*, (Chapter 7 of Mead and Conway [1980].) (1980).

Sheeran[1983].

M. Sheeran, "μFP: An Algebraic Design Language", Ph.D. Thesis, Oxford University (1983).

Shepherdson[1986].

J. C. Shepherdson, "Algorithmic Procedures, Generalised Turing Algorithms, and Elementary Recursion Theory", pp. 285-308 in *Harvey Friedman's Research on the Foundations of Mathematics*, ed. L. A. Harrington, (1986).

Snepscheut[1985].

J. L. A. van der Snepscheut, "A Derivation of a Distributed Implementation of Warshall's Algorithm", Tech. Rept. CS 8505, Groningen University (1985).

Sutherland and Mead[1977].

I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science", *Scientific American* 237(3) pp. 210-228 (1977).

Thatcher, Wagner, and Wright[1980].

J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More Advice On Structuring Compilers and Proving Them Correct", pp. 165-188 in *Semantics Directed Compiler Generation*, ed. N. D. Jones,Springer-Verlag (LNCS 259), Berlin (1980).

Thompson and Tucker[1985].

B. C. Thompson and J. V. Tucker, "Theoretical Considerations in Algorithm Design", pp. 856-878 in *Proc. NATO ASI Fundamental Algorithms for Computer Graphics*, ed. R. A. Earnshaw,Springer-Verlag, Berlin (1985).

Thompson[1980].

C. D. Thompson, "A Complexity Theory for VLSI", Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University (1980).

Tucker and Zucker[1987].

J. V. Tucker and J. I. Zucker, *Program Correctness over Abstract Data Types with Error-State Semantics*, North-Holland (in press), Amsterdam (1987).

Turing[1936].

A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proc. London Mathematical Society, Series* 2 42 pp. 230-265 (1936).

Turner[1982].

D. A. Turner, "MIRANDA: A Non-Strict Functional Language with Polymorphic Types", pp. 1-16 in *Proc. Conf. on Functional Programming Languages and Computer Architecture*, ed. J. P. Jouannaud,Springer-Verlag (LNCS 201), Berlin (1982).

Wagner[1981].

E. G. Wagner, "Lecture Notes on the Algebraic Specification of Data Types", Tech. Rpt. Mathematical Sciences Center, IBM Thomas J Watson Research Center (1981).

Weijland[1987].

W. P. Weijland, "A Systolic Algorithm for Matrix-Vector Multiplication", Tech. Rept. FVI 87-08, Amsterdam University (1987).

Weiser and Davis[1981].

U. Weiser and A. L. Davis, "A Wavefront Notation Tool for VLSI Array Design", pp. 226-234 in *VLSI Systems and Computation*, ed. H. T. Kung, R. Sproull, and G. Steel,Springer-Verlag, Berlin (1981).

Winskel[1987].

G. Winskel, "Models and Logics for MOS Circuits", in *Proc. Hardware Verification Workshop, Calgary 1987 (to appear)*, (1987).