Algebraic Stream Processing

Robert Stephens

Abstract

This thesis is concerned with the development of algebraic techniques for the study of systems that compute over infinite sequences of data called *streams*. We call the mathematical representation of systems that take streams as input and produce streams as output *stream transformers* (STs) and we call the implementations of STs *stream processing systems* (SPSs). Collectively, we call the study of STs and SPSs *stream processing*.

Stream processing encompasses many classes of systems that are studied in Computer Science. For example, *dataflow systems*, *signal processing systems*, *reactive systems* and *synchronous concurrent algorithms* can all be formalized mathematically as STs. These classes of systems are broad and include all traditional forms of sequential and parallel hardware and many specialized models of computation including *artificial neural networks* and *systolic architectures*.

We identify and analyse the typically higher-order approaches to stream processing in the literature. From this analysis we motivate an alternative approach to the specification of SPSs as STs based on an essentially first-order equational representation. This technique is called *Cartesian form specification*. More specifically, while STs are properly second-order objects we show that using Cartesian forms, the second-order *models* needed to formalize STs are so weak that we may use and develop well-understood first-order methods from computability theory and mathematical logic to reason about their properties. Indeed, we show that by specifying STs equationally in Cartesian form as *primitive recursive functions* we have the basis of a new, general purpose and mathematically sound *theory* of stream processing that emphasizes the formal specification and formal verification of STs. The main topics that we address in the development of this theory are as follows.

We present a theoretically well-founded general purpose stream processing language *ASTRAL* (*A*lgebraic *S*tream *TRA*nsformer *L*anguage) that supports the use of modular specification techniques for full second-order STs.

We show how ASTRAL specifications can be given a Cartesian form semantics using the language *PREQ* that is an equational characterization of the primitive recursive functions. In more detail, we show that by compiling ASTRAL specifications into an equivalent Cartesian form in PREQ we can use first-order equational logic with induction as a logical calculus to reason about STs. In particular, using this calculus we identify a syntactic class of correctness statements for which the verification of ASTRAL programmes is decidable relative to this calculus.

We define an effective algorithm based on term re-writing techniques to implement this calculus and hence to automatically verify a very broad class of STs including conventional hardware devices.

Finally, we analyse the properties of this abstract algorithm as a proof assistant and discuss various techniques that have been adopted to develop software tools based on this algorithm.

Acknowledgements

There are a number of people that have given me their help during the writing of this thesis.

First, I am very grateful to my supervisor Ben Thompson for many hours of guidance over the past three years. Ben is a conscientious teacher and I hope that I have inherited some of his careful attention to detail. The formulation of the first Cartesian composition compiler in Chapter 4 is joint work between myself and Ben.

Secondly, I would like to thank John Tucker for his advice on many aspects of my research during my time at Swansea. I am also grateful to Mark Summerfield for his collaboration on the formulation of the implementation of ASTRAL in Chapter 6 and to Barry Hearn for allowing me to use a version of his ATLAS system, especially for helping me adapt ATLAS to my specific needs in Chapter 8.

Thirdly, I must also thank Karl Meinke for several useful conversations on aspects of my work, and Brian McConnell, Matt Poole and L Jason Steggles for their suggestion and comments on reference material that occurs in Chapter 3. I also thank Brian for proof reading my thesis. In addition, the other members of the Theory Group have my thanks for making Swansea a very pleasant working environment over the past three years.

Finally, I would like to thank my parents, Don and Pearl, and the Science and Engineering Research Council for their financial support.

Statement 1

Except where stated otherwise the work contained in this thesis is the result of the author's investigations.

Statement 2

I consent for this thesis to be made available for photocopying and for inter-library loan. I also consent for the title and abstract to be made available to outside organizations.

Declaration

This thesis has not already been accepted in substance for any other degree. This thesis is also not being concurrently submitted in candidature for any other degree.

Submitted in accordance with the requirements for the degree of Ph.D. at The University of Wales, Swansea.

Dedicated to my Wife Rachel for her Constant Love and Support.

Contents

List of Figures

Time like an ever rolling stream,
Bears all its sons away;
They fly forgotten, as a dream;
Dies at the opening day.

Psalm 90

Chapter 1

Introduction

Most people are other people.
Their thoughts are someone else's opinions,
their lives a mimicry,
their passions a quotation.

Oscar Wilde

1.1 Stream Transformers

For a large variety of modern applications computing devices are often required to have a continuous mode of operation. However, from the perspective of 'classical' theoretical computer science the formal analysis of the computation performed by such systems is not straightforward, as most conventional mathematical models rely on a device's *termination* to provide a meaningful semantics. Moreover, modern electronic devices can be of great complexity at both the hardware and software level, but most existing models of computation are low-level formalisms, and hence have limited practical applications for the specification of complex systems.

For example, consider *correctness*; that is, the ability of a system to perform consistently without errors. This is the most important property of so-called *safety-critical systems* such as flight control systems (*fly-by-wire* systems) and medical monitoring equipment. In general, correctness cannot be established by *testing* such systems, and hence correctness can only be proved formally using appropriate mathematical analysis. Despite this fact, for a number of reasons (see Chapter 8), at the moment theoretical computer science has had little 'real-world' success in establishing correctness, and has also had some notable failures (Cullyer [1985] – see Stavridou [1993] for a discussion).

The development of more appropriate approaches to model 'non-terminating systems' is not a new idea, and there is already a large diversity of techniques with many aims and objectives (see Chapter 3). However, even though the origins of the study of non-terminating systems can be traced back to at least the 1960s, with a few notable exceptions since that period this research has been fragmented and pre-occupied with practical rather than theoretical issues. As such, while there has been much interesting research and several specialized theories have been developed, we argue that a *general theory* of such systems has not emerged in the literature; that is, a theory that encompasses topics such as the scope and limits of stream-based computation, a study of the computability of stream processing primitives and constructs, and the decidability of the verification of a SPS relative to some abstract specification.

It is our aim in this thesis to develop the basis of a theory of stream processing that addresses these and other issues. In particular, our aim is the development of an *algebraic* theory of non-terminating computation with an emphasis on formally establishing correctness. In order to achieve this aim we will combine two 'new' techniques for representing non-terminating systems: (1) we will specify non-terminating systems at a more abstract level than in existing research; and (2) we will use an *essentially first-order* specification technique. More specifically:

(1) our analysis of non-terminating systems is based on the idea that at the conceptual level they can be viewed as devices that receive infinite sequences as input and produce infinite sequences as output. An infinite sequence, that we refer to as a *stream*, is essentially a list of elements $a_0, a_1, a_2, \ldots$ taken from some data set of interest A, and can be formalized mathematically as a function $a : T \to A$, wherein $T = \mathrm{N} = \{0, 1, 2, \ldots\}$ represents discrete time. We call a system that takes n streams as input and produce m streams as output for some $n, m \geq 1$ a *stream transformer* (ST) and characterize such a system as a *functional*

$$\Phi : [T \to A]^n \to [T \to A]^m.$$

As a general term we refer to the study of the theoretical and practical aspects of STs as *stream*

processing.

(2) our study of STs is based on an alternative specification technique that we call *Cartesian form specification.* In particular, this is the reason that we refer to our work as *algebraic* as a *Cartesian form stream transformer* (CFST) can be represented using *first-order* specification techniques. Indeed, we believe that it is one of the fundamental strengths and differences of the analysis of STs that we present in this thesis that it is based on first-order methods, and hence our techniques are both *effective* (readily implementable by software) and well-understood from a theoretical perspective.

Our study of *algebraic stream processing* culminates in Theorem 18 in Chapter 7 that states that a large and useful class of STs, that encompass many conventional types of hardware and software, can be verified formally and automatically using straightforward techniques. However, as we will see, the proof of Theorem 18 requires several important intermediate results, one of the most notable of which is Theorem 7. This latter result establishes the practical applicability of CFSTs as a general purpose specification technique. We now introduce these ideas in more detail.

1.2 Thesis Overview

The research that we present in this thesis develops both practical and theoretical techniques that are appropriate for the study of STs, and in particular for the formal specification and formal verification of STs. In the remainder of this introduction we motivate the main issues that will be the focus of our work, and briefly discuss the particular topics addressed in each chapter. A more detailed introduction can be found at the end of Chapter 3 once we have had the benefit of our mathematical preliminaries and a survey of the literature.

As our introduction suggests we see one of the main applications of our work as being the development of techniques that are appropriate for specifying and verifying safety-critical systems, with a particular interest in safety-critical hardware. Furthermore, all of the techniques that we develop are also appropriate for the many other types of *stream processing systems* (SPSs) that can already be found in the literature: including *dataflow systems*, *reactive systems*, certain classes of special purpose functional and logic programming languages and *synchronous concurrent algorithms* (SCAs). Indeed, it is the class of SCAs, that themselves encompasses many types of computational models including: *artificial neural networks*, *cellular automata* and *coupled-map lattice dynamical systems*, that will provide the starting point of our work. Most specifically, because SCAs are also particularly appropriate for the formal study of hardware as STs. A detailed discussion of SCA theory can be found in Thompson and Tucker [1991].

Before we can motivate the reasons that we believe SCAs provide an appropriate starting point to develop a theory of stream processing in the sense defined in the previous section, there is a certain level of familiarity with specific mathematical concepts that the reader must have. In particular, like the theory of SCAs, our theory of stream processing is based on ideas from the theory of *universal algebra.* As such, we will present a brief, but sufficient introduction to *signatures*, *algebras* and related concepts that can be used to formalize rigorously the abstract

notion of *data types*.

At this point the reader familiar with these concepts may be somewhat surprised, as universal algebra has been designed to formalize *first-order* specification. In contrast, the informal description of STs that we have presented in our introduction clearly shows that STs receive functions as input, and hence are properly *second-order* objects. Despite this fact and even though higher-order algebraic techniques exists (see for example Meinke [1992b]) we will show that by adopting the method of Cartesian form specification, based on the work of B C Thompson and J V Tucker (see Thompson [1987]), it is wholly appropriate to apply these first-order techniques to the study of second-order systems. This is the reason that we refer to our methods as 'essentially first-order'. We will return to this point throughout this thesis.

1.2.1 Algebraic Preliminaries

In Chapter 2 we present our algebraic preliminaries. We also present the basic ideas behind the theory of *equational specification* and *term re-writing* that will play a central role in our work for the formal specification and formal verification of STs respectively. All other necessary preliminary definitions and further specific concepts based on the general ideas that we present in Chapter 2 are developed in the first chapter in which they are used.

1.2.2 A Survey of Stream Processing

In Chapter 3 we carefully motivate the advantages that we believe our techniques have over existing approaches to stream processing with a detailed literature survey. In particular, later in Chapter 6 this literature survey will enable us to identify a clear correspondence between our theoretical tools and their practical applications by highlighting some of the relevant features of our formal specification and programming language *ASTRAL*.

We begin our literature survey with a brief historical perspective of the development of stream processing since its origins in the 1960s. This is followed by a more detailed analysis of each specific area of research that can be identified. Wherever possible, as we are developing a theory of stream processing, we are careful to separate theoretical and practical issues. More specifically, we are careful to distinguish between language design and implementation issues, and semantics and formal specification.

Most importantly we conclude Chapter 3 with a detailed discussion of SCAs and their theoretical and practical advantages. This enables us to set a specific agenda of research for the rest of the thesis that is sufficient as the basis of a theory of stream processing in the sense defined in our introduction.

1.2.3 Primitive Recursion

One of the issues that is of particular importance in developing theoretical tools in this thesis is *computability*. Indeed, we base the semantics of the formal specification language ASTRAL that we develop in Chapter 6 on an equational formulation of the class of *primitive recursive functions*. In the same way that universal algebra precisely formalizes the concept of data, the explicit use of primitive recursion provides a solid mathematical foundation for our methods

from the perspective of the class of functions that we may specify.

In Chapter 4 we present the language *PR* developed by B C Thompson in Thompson [1987] that is one method of generalizing to an abstract algebraic setting the work of S Kleene (see for example Cutland [1980]) that was concerned with primitive recursive computation over the natural numbers. We also present several computability theoretic extensions of PR including the language μPR (see Tucker and Zucker [1988]) that over an appropriate algebra provides a general model of computation equivalent to Turing machine computation on abstract structures.

PR and its extensions are functional languages that provide convenient mathematical tools to establish certain facts, including the realization of Cartesian form specification as a practical technique. In more detail, by expressing STs in Cartesian form we will show that we may formally reason about their properties using first-order techniques and hence exploit several theoretical advantages over higher-order specification methods. However, when expressed in Cartesian form it is not immediately obvious that STs are *compositional* and thus appears to limit their potential as a practical specification tool. In particular, given two *Cartesian form stream transformers* (CFSTs) h and g it is not obvious that we may uniformly construct a CFST f such that $f = h \circ g$, and so it is not clear that we may apply modular specification techniques.

Chapter 4 is devoted to a detailed discussion of the problem of *Cartesian composition* including its rigorous formalization as a mathematical problem (Theorem 7) and to the development of an effective solution. Indeed, the proof of Theorem 7 is based on the correctness of a formal compiler $\mathbb{C}$ that given two μPR schemes can construct a single μPR scheme with the required semantics. For example, in the case of the μPR schemes α_h and α_g representing the functions h and g above, the result of the compilation $\mathbb{C}(\alpha_h, \alpha_g)$ is a μPR scheme α_f such that the semantics of α_f is the function f.

1.2.4 Primitive Recursive Equational Specification

Like a Turing machine the language PR (and its extensions) provide a useful tool for establishing certain theoretical facts. However, PR also shares a further similarity with a Turing machine in the sense that it is a very low-level specification method. To address this problem, in Chapter 5 we develop the equational language *PREQ*, also based on the technique of primitive recursive specification, that provides a more appropriate syntax for the representation of large systems. In particular, it is PREQ and not PR that we use as the semantics of our specification language ASTRAL.

In developing PREQ, if we wish to use it as a tool for the formal verification of STs then we must establish that PREQ does precisely capture the class of primitive recursive functions. As part of our research, in Chapter 5 we prove this fact formally and in so doing also provide a mechanism to constructively exploit the properties of the compiler $\mathbb{C}$. In more detail, in order to establish formally that PREQ captures the class of primitive recursive functions we use two further formal compilers: the compiler $\mathbb{C}^{\text{PREQ}}$ that maps PR schemes into PREQ; and the compiler $\mathbb{C}^{\text{PR}}$ that maps PREQ specification into PR.

The equivalence of PREQ with PR (that is, PREQ's *soundness* and *adequacy* with respect to the class of primitive recursive functions) is stated formally in Theorem 10 wherein $\mathbb{C}^{\text{PR}}$ is

used to establish soundness, and $\mathbb{C}^{\mathrm{PREQ}}$ is used to establish adequacy. However, in addition, as we wish to use the compilers $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$ as the basis of software tools, a large part of Chapter 5 is also devoted to establishing the efficiency of our compilation techniques from the perspective of the number of equations that are produced by $\mathbb{C}^{\mathrm{PREQ}}$. An important concept that we use in this discussion is the formulation of a *normal form* representation for PR schemes.

The final section of Chapter 5 is devoted to establishing one further theoretical property of PREQ specifications relative to their use for formal verification, but this time from the particular perspective of *automated reasoning*; that is, we establish that when PREQ specifications are orientated as *left-to-right* re-write rules they produce *term re-writing systems* (TRSs) that are *complete*. Indeed, the construction of a complete TRS from a PREQ specification (Theorem 11) forms a crucial part of the development of our automated verification techniques for STs in Chapter 7.

1.2.5 ASTRAL

Having completed the first part of our research agenda, in Chapter 6 we are in a position to begin the development of the specification language ASTRAL that is specifically tailored for the representation of STs.

The first part of our development of ASTRAL is concerned with the formulation of its abstract syntax and semantics. Specifically, rather than formulate an independent semantics for ASTRAL we prefer to derive the meaning of an ASTRAL specification by compiling it into a Cartesian form specification in PREQ. This enables ASTRAL to incorporate high-level language features, but also to exploit the theoretical properties that are possessed by PREQ specifications. As such, it is necessary for us to design several further formal compilers. In particular, to allow modular specification techniques to be applied we define a generalized version of the Cartesian composition compiler $\mathbb{C}$ that we presented in Chapter 4. This new compiler is tailored to accommodate practical as well as theoretical concerns and we discuss its actual implementation in Chapter 8.

The second part of the development of ASTRAL is concerned with practical issues relating to the implementation of its abstract syntax. In more detail, we develop an implementation of ASTRAL in a form that is suitable as a high-level, declarative programming and specification language. However, rather than do this by formally presenting a BNF, we motivate our implementation by the use of several small case studies. In particular, we use the numerous stream processing primitives and operations that we present as part of our literature survey to demonstrate the effectiveness of ASTRAL as a specification tool. For example, while ASTRAL formally derives its semantics from a primitive recursive function, and hence does not provide a general model of computation in the sense of a Turing machine, we show that *in practice* this is *not* a limiting feature from the perspective of the class of systems that we can specify.

1.2.6 Automated Verification

As we indicated in the first part of this introduction, one of our main research aims is the formal verification of a stream transformer's correctness to provide appropriate theoretical tools for the

verification of certain types of safety-critical systems.

In order to achieve this aim in Chapter 7 we begin with a discussion of the relationship between using equational logic as a formal calculus and the intended semantics of an equational specification; that is, we discuss the difference between *loose semantics*, and *initial semantics* that is typically the intended meaning of a formal specification (see Goguen [1988]). This discussion motivates the definition of a formal calculus *EQWIL*, based on the rules of equational logic, but extended with induction, that is suitable for reasoning about the properties of PREQ specification relative to their initial semantics.

Using EQWIL as a formal deduction system we show that by defining two functions (*VER* and *EVER*), both based on term re-writing techniques, we can draw on our previous results (in particular Theorem 11) to show that it is possible to fully automate deductions about the correctness of STs using TRSs created from PREQ specifications.

More specifically, we show that it is possible to reduce deductions about the initial truth of Cartesian form (weak second-order) equational correctness statements in weak second-order systems of equations to deductions about the initial truth of first-order equations in strictly first-order systems of equations. Furthermore, by identifying certain syntactic classes of correctness statement we show that by specifying STs as ASTRAL programmes a very broad class of hardware devices can be proved formally correct using completely automated software tools. Indeed, continuing our use of constructive techniques we do this by using EVER to define a further function *AV* that given two ASTRAL programmes can automatically decide if they can be proved equivalent using the calculus EQWIL. The formal statement of AV's properties is given by Theorem 19 that is accompanied by a explanation of our theoretical results particular practical implications. Moreover, Theorem 19 has important applications to the verification of SCAs, including one useful result that shows that deciding the equivalence of two cellular automata using the calculus EQWIL is decidable.

1.2.7 Implementing a Proof Tool for STs: a Case Study

To complete the demonstration of the the practical applications of our work, we conclude our research with a discussion of the implementation of software tools based on some of the abstract functions that we have defined. We do this with the use of a small case study: the RS-Flip-Flop that is a commonly occurring hardware device.

The software that we have developed has concentrated on the implementation of two specific functions: the generalized version of the compiler $\mathbb{C}$ and the function EVER, that are sufficient to demonstrate the effectiveness of a full implementation of the function AV.

We present the high-level algorithms on which our implementations are based and demonstrate both the practical and theoretical benefits that these algorithms provide. Specifically, from the perspective of efficiency we show that it is possible to identify the first point at which an automated verification can be aborted. Also, from a theoretical perspective we show that without modifying the underlying formal calculus EQWIL on which our implementation is based, it is possible to deduce existential as well as universal quantification on certain variables. This has useful applications for the verification of hardware devices as it is often appropriate to prove a device is correct relative to some specific initial configuration of the devices memory.

Finally, we discuss the steps necessary to implement a full implementation of AV and present some techniques that would be sufficient to increase the performance of our prototype tools to make them suitable to verify complex hardware devices such as modern microprocessors.

Chapter 2

Algebraic Preliminaries

How many things I have no need of!

Socrates

2.1 Overview

The mathematical framework that we will work within in this thesis is based on the theory of *abstract data types* and more generally the theory of *universal algebra*. As such this chapter is structured as follows:

In Section 2.2 we have some general preliminaries concerning our basic notation for defining sets and functions.

In Section 2.3 we present an introduction to the fundamental algebraic concepts that will be used throughout this thesis.

This is followed in Section 2.4 by some further algebraic preliminaries concerned specifically with stream processing. In particular, we present a concise notation to classify the various types of stream processing systems that we will encounter in Chapter 3.

We conclude this chapter in Section 2.5 with a discussion of the work of Kahn [1974] that has been used widely to provide a semantics for stream processing systems. An understanding of *Kahn's method* is useful as it clarifies some of the points that we raise in our literature survey that follows in the next chapter. However, this section is not essential and can be omitted on a first reading by the reader who is not familiar with the necessary concepts from *domain theory* (see for example Stoltenberg-Hansen *et al.* [1994]).

2.2 General Preliminaries

We now define some general mathematical preliminaries that we will use throughout this thesis.

2.2.1 Set Notation

As is standard we use $\wp(x)$ to denote the *power set of* x, and the operators $\bigcup$, $-$, and $\bigcap$ to denote set-theoretic *union*, *difference* and *intersection* respectively. We denote the size (cardinality) of a set x by $|x|$.

2.2.2 Functions

We denote the domain and range of a function f by $\mathrm{dom}(f)$ and $\mathrm{ran}(f)$ respectively.

We represent the fact that a function f is partial by writing $f : A \rightsquigarrow B$ for some data sets A and B. When f is a partial function we denote the fact that f *is defined* on some element $x \in \mathrm{dom}(f)$ by $f(x)\downarrow$ and the fact that f *is undefined* on some element $x \in \mathrm{dom}(f)$ by $f(x)\uparrow$. Furthermore, when the value $x \in \mathrm{dom}(f)$ is either understood or unimportant we will simply write $f\downarrow$ and $f\uparrow$ respectively. We also use the symbols '$\downarrow$' and '$\uparrow$' on their own to mean 'defined' and 'undefined' respectively. Continuing the use of this notation we define

$$\mathrm{dom}(f)\downarrow = \bigcup_{x \in \mathrm{dom}(f)} f(x)\downarrow .$$

Of course in general the properties $f(x)\downarrow$ and $f(x)\uparrow$ for some f and for some $x \in \mathrm{dom}(f)$ are undecidable. Therefore, if we do make use of a partial function as part of a definition we will

always ensure that it is possible to decide whether for each $x \in \mathrm{dom}(f)$ if $f(x)\downarrow$. To this end we often find it useful to use the *set maplet* notation to define a (partial) function. For example, if

$$f : \mathrm{N} \rightsquigarrow \mathrm{N}$$

is defined by

$$(\forall n \in \mathrm{N}) \quad f(n) = \begin{cases} 5 & \text{if } n = 2, \\ 6 & \text{if } n = 3, \\ 7 & \text{if } n = 4, \text{ and} \\ \uparrow & \text{otherwise} \end{cases}$$

then we write

$$f = \{2 \mapsto 5, 3 \mapsto 6, 4 \mapsto 7\}.$$

Continuing the visualization of functions as sets of 'maplet pairs', for two functions $f, g : A \rightsquigarrow B$ for some data sets A and B we write:

$$f \bigcap g = \varnothing$$

to mean that

$$(\forall a \in A) \quad (f(a)\downarrow \Longrightarrow g(a)\uparrow) \wedge (g(a)\downarrow \Longrightarrow f(a)\uparrow);$$

and

$$f \supseteq g$$

to mean that

$$(\forall a \in A) \quad (g(a)\downarrow \Longrightarrow f(a)\downarrow \wedge (f(a) = g(a))).$$

Furthermore, if $f \bigcap g = \varnothing$ then we write $h = f \bigcup g$ to denote the function $h : A \rightsquigarrow B$ defined by

$$(\forall a \in A) \quad h(a) = \begin{cases} f(a) & \text{if } f(a)\downarrow, \\ g(a) & \text{if } g(a)\downarrow, \text{ and} \\ \uparrow \text{ otherwise.} & \end{cases}$$

Finally, we write $h = f[\{n_1, \ldots, n_k\}]$ for some $n_i \in A$ for $i = 1, \ldots, k$ to mean the function $h : A \rightsquigarrow B$ defined by

$$(\forall a \in A) \quad h(a) = \begin{cases} f(a) & \text{if } a = n_i \text{ for some } i \in \{1, \ldots, k\} \text{ and } f(a)\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

and hence h is the *restriction of* f *to the domain* $\{n_1, \ldots, n_k\}$.

2.3 Algebraic Preliminaries

For the purposes of the first chapters of this thesis of a few basic concepts from the theory of universal algebra are sufficient: *sort sets*, *signatures*, *algebras*, *reducts*, *standard algebras*, *terms* and *term evaluation*. As such, the reader familiar with these ideas can move directly to Section 2.4 on Page 19.

The reader requiring a more detailed introduction to the theory of universal algebra can consult Goguen *et al.* [1978], Ehrig and Mahr [1985], Meseguer and Goguen [1985], Goguen [1990], and most specifically for a presentation using the notation of this thesis Meinke and Tucker [1992].

2.3.1 Sorts

A *sort set* is a finite set $S = \{s_1, s_2, \ldots, s_n\}$ wherein each $s_i \in S$ for $i = 1, \ldots, n \in \mathbb{N}$ is called either a *sort name* or a *sort symbol*, with the intention that it will be used to name a *carrier* over which we may perform some computation (see Section 2.3.3). In particular, we say S is *standard* if it contains the sort symbols **n** and **b** with the intention that these symbols name the natural numbers and the Booleans respectively (see Sections 2.3.4 and 2.3.5).

We write $w \in S^*$ to indicate that w is a member of the set of all *strings* (also called *words*) over S including the empty sequence λ. We also define $S^+ = S^* - \{\lambda\}$ to be the set of non-empty words and S^n for each $n \in \mathbb{N}$ to be the set of all words of length n.

We denote the length of a word w by $|w|$. Thus, if $w = \lambda$ then $|w| = 0$ and if $w \in S^n$ then $|w| = n$.

Finally, we denote the ith sort of w by w_i; that is, $w = w_1 \cdots w_{|w|}$ for each $w \in S^*$.

2.3.2 Signatures

An *S-sorted signature* Σ is an $S^* \times S$-indexed family

$$\Sigma = < \Sigma_{w,s} \mid w \in S^*, s \in S >$$

of disjoint sets. The signature is *finite* when $\Sigma_{w,s}$ is finite and non-empty for finitely many $(w, s) \in S^* \times S$.

Each set $\Sigma_{\lambda,s}$, with typical member 'c', is called the set of *constant symbols of sort s*, and each set $\Sigma_{w,s}$ for any $w \in S^+$, with typical member 'σ', is called the set of *operator symbols of type $w \to s$*. In particular, we sometimes write $\sigma : w \to s$ to indicate σ is a member of $\Sigma_{w,s}$.

An S-sorted signature Σ is *standard* if S is standard and the usual constants and operator symbols associated with the carriers **n** and **b** (see Sections 2.3.4 and 2.3.5) occur in Σ. Finally, a signature Σ is *non-void* if for each $s \in S$ we have $|\Sigma_{\lambda,s}| \geq 1$. Indeed, for technical reasons (see for example MacQueen and Sanella [1985]) in this thesis we always assume that every signature without stream sorts is non-void. However, we *do* allow stream signatures (weak second-order signatures) to be void, but only in their stream sorts (see Definition 2.4.2).

2.3.3 Algebras

Let Σ be an S-sorted signature. An *S-sorted Σ-algebra* A (also called a *Σ-structure*) consists of an S-indexed family

$$A =< A_s \mid s \in S >$$

of sets, where each set A_s is called the *carrier of sort s* (also called the *domain of sort s*), and a $S^* \times S$-indexed family

$$\Sigma^A = < \Sigma^A_{w,s} \mid w \in S^*, s \in S >$$

defined such that
(1) for each $s \in S$ and each $c \in \Sigma_{\lambda,s}$ there exists a constant $c^A \in A_s$, and
(2) for each $w \in S^+$, for each $s \in S$ and for each $\sigma \in \Sigma_{w,s}$ there exists a mapping $\sigma^A : A^w \rightarrow A_s \in \Sigma^A_{w,s}$, wherein A^w denotes the Cartesian product $A_{w_1} \times A_{w_2} \times \cdots \times A_{w_{|w|}}$.

We define A^n for each $n \in \mathbb{N}$ to be the set of all Cartesian products of length n made from the carriers of A and use A^+ to denote $\bigcup_{n \geq 1} A^n$.

For notational convenience we sometimes informally and ambiguously denote an algebra with n constants and m operations by $A = (A; c_1, \ldots, c_n; \sigma_1, \ldots, \sigma_m)$. Finally, we write $Alg(\Sigma)$ for the class of all Σ-algebras, and if a property holds for each algebra $A \in Alg(\Sigma)$ then we say it is *uniform* in A.

2.3.4 Natural Numbers

Although we have already used the symbol $\mathbb{N}$ informally, we now give a more formal algebraic definition.

We use the distinguished sort symbol **n** to name the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. We also ambiguously use $\mathbb{N}$ to denote the single-sorted algebra of natural numbers defined by $\mathbb{N} = (\mathbb{N}; 0; succ)$ wherein '0' (zero) and '*succ*' (successor) have their usual interpretations. For historical reasons we will also use T to denote the natural numbers when they are being used to measure time (see Section 2.4). Finally, we use $\mathbb{N}^+$ to denote $\mathbb{N} - \{0\}$.

2.3.5 Booleans

We use the distinguished sort symbol **b** to name the set $\mathbb{B} = \{tt, ff\}$ of Boolean truth values. We also ambiguously use $\mathbb{B}$ to denote the single-sorted algebra of Booleans defined by $\mathbb{B} = (\mathbb{B}; tt, ff; not, and, or)$, wherein '*tt*' (true), '*ff*' (false), '*not*', '*and*' and '*or*' have their usual interpretations.

2.3.6 Reducts

Given a Σ-algebra A, we can form a new algebra B from A by 'forgetting' some of A's operations.

Formally, let A be an S-sorted Σ-algebra and let Ω be an S'-sorted signature with $S' \subseteq S$ and $\Omega \subseteq \Sigma$; that is, let $\Omega_{w,s} \subseteq \Sigma_{w,s}$ for each $w \in (S')^+$ and for each $s \in S'$. We define the *Ω-reduct* of A to be the S'-sorted Ω-algebra B whose carriers are a sub-set of the carriers of

A and each of whose operations σ^B is defined by $\sigma^B = \sigma^A$. In symbols, we write $B = A|_\Omega$ to indicate that B is a reduct of A.

2.3.7 Standard Algebras

For any S-sorted Σ algebra A we define A to be *standard* if both of the algebras $\mathbb{N}$ and $\mathbb{B}$ are reducts of A.

Standard algebras play a central role in this thesis in the sense that for our purposes, in general, a standard algebra containing *only* the natural numbers and the Booleans is the smallest algebra over which we wish to 'compute'.

2.3.8 Variables

Let S be any sort set. We define an S-indexed family of *variable symbols* by

$$X = < X_s \mid s \in S >$$

wherein X_s for each $s \in S$ is either a finite (possible empty) or countable infinite collection of variable symbols of sort s.

2.3.9 Terms

Let Σ be any S-sorted signature and let X be an S-indexed family of variable symbols such that Σ and X are *pairwise disjoint*; that is, such that for each $w \in S^*$ and for each $s, s' \in S$ we have $\Sigma_{w,s} \cap X_{s'} = \varnothing$ so that there is no confusion between symbols.

For each $s \in S$ we define $T(\Sigma, X)_s$, the set of *terms of sort* s uniformly in $s \in S$ by induction on the structural complexity of a term t as follows:

Basis Cases

(1) If $t = c$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$ then $t \in T(\Sigma, X)_s$.

(2) If $t = x$ for some $x \in X_s$ for some $s \in S$ then $x \in T(\Sigma, X)_s$.

Induction

(3) If $t = \sigma(t_1, \ldots, t_n)$ for some $\sigma \in \Sigma_{w,s}$, for some $w = s_1 \cdots s_n \in S^+$, for some $s \in S$ and for some $t_i \in T(\Sigma, X)_{s_i}$ for $i = 1, \ldots, n$ then $t \in T(\Sigma, X)_s$.

We denote the S-indexed family of terms $T(\Sigma, X)$ defined by

$$T(\Sigma, X) = < T(\Sigma, X)_s \mid s \in S > .$$

If a term $t \in T(\Sigma, X)_s$ for some $s \in S$ is defined without the use of variable symbols; that is, if t is defined without the use of Case (2) of the inductive definition above then we say that t is either a *closed term* or a *ground term*. In particular, we write $t \in T(\Sigma)_s$ to indicate that t is a closed term and gather together all closed terms into an S-indexed family in the same way as above:

$$T(\Sigma) =< T(\Sigma)_s \mid s \in S > .$$

If t is not a closed term then we say that it is an *open term* indicating that it contains at least one variable symbol.

For two terms t and t' we say that t *is a sub-term of* t' denoted $t \subseteq t'$ if either $t = t'$ or t occurs as part of t'. For example, if $t = add(x, y)$ and $t' = mult(add(x, y), z)$ then $x \subseteq t \subseteq t'$ and $z \subseteq t'$, but $z \not\subseteq t$.

In the sequel we will find it necessary to replace sub-terms of a term τ with other terms. For example, given the terms t and t' as defined above we might wish to replace the variable z with the term t to derive the term $mult(add(x, y), add(x, y))$. In order to denote such a replacement we will use the following notation: given terms τ, η and ρ we will write

$$\tau[\eta/\rho]$$

to indicate the term τ with all sub-terms η replaced by ρ. Thus, $t'[z/t]$ with t and t' as defined above is $mult(add(x, y), add(x, y))$. If η is not a sub-term of τ then we define $\tau[\eta/\rho]$ to be simply τ itself.

Finally, in Chapter 7 we will also find it useful to extended this notation by replacing a set of sub-terms of a term relative to some indexing in the following way: for some term τ and terms η_i and ρ_i for $i = 1, \ldots, n \in \mathbb{N}^+$ we will write

$$\tau[\eta_i/\rho_i]_{i=1}^{i=n}$$

to mean

$$((((\tau[\eta_1/\rho_1])[\eta_2/\rho_2]) \cdots)[\eta_n/\rho_n])$$

wherein the bracketing is simply for clarity to indicate the order in which we assume the replacements take place as this may be significant in the case that either $\eta_j \subset \eta_k$ for some $j, k \in \{1, \ldots n\}$ such that $j < k$, or $\eta_j \subset \rho_k$ for some $j, k \in \{1, \ldots n\}$ such that $j > k$.

2.3.10 Term Evaluation

Given an algebra A and a function $\eta : X \to A$ a *term evaluation map* is an S-indexed family of functions

$$V^\eta = < V_s^\eta : T(\Sigma, X)_s \to A_s \mid s \in S >$$

that interprets every term as an element in the carriers of A. In particular, a term evaluation map $V_s^\eta : T(\Sigma, X)_s \to A_s$ (ambiguously denoted V^η) is defined uniformly in $s \in S$ by induction on the structural complexity of a term t as follows:

Basis Cases

(1) If $t = c$, for some $c \in \Sigma_{\lambda,s}$ and for some $s \in S$ then

$$V^\eta(t) = c^A.$$

(2) If $t = x$, for some $x \in X_s$ and for some $s \in S$ then

$$V^\eta(t) = \eta(t).$$

Induction

(3) If $t = \sigma(t_1, \ldots, t_n)$, for some $\sigma \in \Sigma_{w,s}$, for some $w = s_1 \cdots s_n \in S^+$, for some $s \in S$ and for some $t_i \in T(\Sigma, X)_s$, for $i = 1, \ldots, n$ then

$$V^\eta(t) = \sigma^A(V^\eta(t_1), \ldots, V^\eta(t_n)).$$

Term evaluation maps will play a significant role in the formulation of a denotational semantics for the specification language PREQ in Chapter 5.

2.3.11 Equations

Using terms we are now able to formally define the concept of an equation and systems of equations.

Let Σ be any S-sorted signature and let X be an S-indexed family of variable symbols such that Σ and X are *pairwise disjoint.* We define $\mathrm{EQ}(\Sigma, X)_s$ the set of *equations of sort* s such that $e \in \mathrm{EQ}(\Sigma, X)_s$ if, and only if e is an expression of the form

$$t = t'$$

for some $t, t' \in T(\Sigma, X)_s$. Similarly to terms, we also gather together equations into an S-indexed family $\mathrm{EQ}(\Sigma, X)$ defined by

$$\mathrm{EQ}(\Sigma, X) = < \mathrm{EQ}(\Sigma, X)_s \mid s \in S >$$

and draw a distinction between *closed* and *open equations*; that is, if $e = (t = t')$ for some $t, t' \in T(\Sigma)_s$ then e is a *closed equation* denoted $e \in \mathrm{EQ}(\Sigma)_s$. Otherwise, if $e = (t = t')$ for some $t, t' \in T(\Sigma, X)_s$ such that either $t \notin T(\Sigma)_s$ or $t' \notin T(\Sigma)_s$ then e is an *open equation.* Again, we gather together closed equations into an S-indexed family

$$\mathrm{EQ}(\Sigma) = < \mathrm{EQ}(\Sigma)_s \mid s \in S > .$$

Extending the use of our term replacement notation defined above, in the sequel for some equation $e = (t = t')$ and for some terms τ and τ' we will write $e[\tau/\tau']$ to mean the equation $t[\tau/\tau'] = t'[\tau/\tau']$.

For the purposes of this thesis we define a *system of equations* $E = \{e_1, e_2, \ldots, e_n\}$ to be a finite set such that $e_i \in \mathrm{EQ}(\Sigma, X)$ for $i = 1, \ldots, n \in \mathbb{N}^+$ and write $E \subset \mathrm{EQ}(\Sigma, X)$. For example, if Σ is some standard signature and $X \supseteq \{x\}$ wherein x is of sort **b** then the system of equations $E^{\mathbb{B}} \subset \mathrm{EQ}(\Sigma, X)$ defining the standard operations of the Boolean algebra is defined by: $E^{\mathbb{B}} = \{not(tt) = ff, not(ff) = tt, and(tt, x) = x, and(x, tt) = x, and(x, ff) = ff, and(ff, x) = ff, or(tt, x) = tt, or(x, tt) = tt, or(ff, x) = x, or(x, ff) = x\}$.

2.3.12 Term Re-Writing Systems

The concepts of *term re-writing* and *term re-writing systems* (TRSs) will play a fundamental role in Chapter 7 and 8 wherein we automate the process of making formal deductions about the correctness of SPSs relative to a specification. However, while term re-writing is based on straightforward ideas it is a complex area of research with many subtleties. Therefore in this section we present an informal overview of the key ideas underlying the subject that will be sufficient to understand the theorems presented in Chapter 7. The reader that does not find this brief and informal account sufficient is directed to Dershowitz and Jouannaud [1990] and Klint [1993] for more detailed introductions.

Essentially, a TRS is a system of equations $E \subset \mathrm{EQ}(\Sigma, X)$ wherein each $e = (t = t') \in E$ is orientated into what is referred to as a *re-write rule*, denoted $t \mapsto t'$, indicating that for the purpose of deduction we will only use an equality in one direction. However, we may include either of the re-write rules $t \mapsto t'$ and $t' \mapsto t$ or both in a TRS if we wish. In particular, we write $\mathrm{TRS}(\Sigma, X)_s$ to denote the set of all sets R constructed from each set $E \subset \mathrm{EQ}(\Sigma, X)_s$ in this way; that is, $R \subset \mathrm{TRS}(\Sigma, X)_s$ if, and only if there exists a corresponding $E \subset \mathrm{EQ}(\Sigma, X)_s$ such that for each $e = (t = t') \in E$ we have either $t \mapsto t' \in R$, or $t' \mapsto t \in R$, or $t \mapsto t' \in R$ and $t' \mapsto t \in R$. We also gather each $\mathrm{TRS}(\Sigma, X)_s$ for each $s \in S$ into an S-indexed family $\mathrm{TRS}(\Sigma, X)$.

The motivation for working with a TRS rather than a system of equations is that it provides a mechanism by which we may automate equational deduction. In more detail, Birkhoff's Soundness and Completeness Theorem (see for example Meinke and Tucker [1992]) shows (1) that any equation that can be proved using *equational logic* is *valid – soundness*; and (2) that any equation that is valid can be proved by equational logic – *completeness* (see Chapter 7). This result is very useful from the perspective of formal verification as it shows, in a precise mathematical sense, that for reasoning about equations the formal deduction system provided by equational logic is ideal. However, from the particular perspective of automated verification it is not straightforward to implement equational logic directly to provide the basis of verification tools, and so this has stimulated research into term re-writing.

More specifically, it is the combination of Birkhoff's Theorem with a further fundamental result – the Correspondence Theorem (see for example Klop [1992]) – that has stimulated term re-writing research. In particular, the Correspondence Theorem shows that if we form a TRS from a system of equations E by orienting each equation as both a left-to-right and right-to-left re-write rule then the set of equations that were deducible from E using equational logic is precisely the set of equations that is deducible from R using term re-writing. Therefore, term re-writing is equivalent in its proof theoretic strength to equational logic, and in addition from the perspective of implementation is more straightforward.

Given a term t and a TRS R containing a rule $r = (\eta \mapsto \rho)$, term re-writing is based on the principle of *re-writing* t to some equivalent τ using r (written either $t \rightarrow_r \tau$ or just $t \rightarrow_R \tau$ if the rule used from R is either understood or unimportant) relative to some appropriate *substitution*. For example, if t is the term $and(tt, ff)$, R is defined by orienting each equation in $E^{\mathbb{B}}$ (as defined above) into a left-to-right re-write rule, and the rule r taken from R is $and(tt, x) \mapsto x$ then $t \rightarrow_r ff$ with the substitution $x = ff$. Thus informally, a substitution is nothing more than an

instantiation of the variables in η that makes η syntactically identical to a (part of) term t, and the result of re-writing with this substitution is t (with that part) replaced with ρ under the same substitution of variables.

It is clear how given a TRS $R \subset \mathrm{TRS}(\Sigma, X)$ the process of automatically re-writing a term can be readily implemented. In particular, we can decide if an equation $e = (t = t') \in \mathrm{EQ}(\Sigma, X)$ is valid by repeatedly re-writing t and t' with rules from R (written $t \twoheadrightarrow_R \tau$ and $\tau' \twoheadrightarrow_R \tau'$) and testing to see if $\tau = \tau'$. However, this technique now raises several points. First, recall that in general to make an equivalent TRS R from a system of equations E the Correspondence Theorem required that we had to add a re-write rule to R made by orienting each $e \in E$ as both a left-to-right and right-to-left re-write rule. Specifically, in general a TRS R may either have two rules of the form $r_1 = (\eta \mapsto \rho)$ and $r_2 = (\rho \mapsto \eta)$ or two rules of the form $r_1' = (\eta' \mapsto \rho')$ and $r_2' = (\eta' \mapsto \rho'')$ and so it not clear that: (1) re-writing a term is a terminating procedure; and (2) re-writing a term has a unique result. We will show that it is straightforward to deduce that a TRS created from a PREQ specification has precisely these two properties and hence is ideal for automated reasoning.

The basic term re-writing concepts that we have discussed are made precise in the following definitions, that also include some further terminology we will use in the sequel.

Let $R \subset \mathrm{TRS}(\Sigma, X)$ be any TRS and let $t \in T(\Sigma, X)$ be any term.

Definition 1. The term t is a *normal form* under R if there is no rule $r \in R$ such that $t \rightarrow_r \tau$; that is, if it is not a reducible expression (*redex*) under R.

Definition 2. R is *weakly terminating* (also called *weakly normalizing*) if for each term t there exists a normal form ν such that $t \twoheadrightarrow_R \nu$.

Definition 3. R is *strongly terminating* (also called *strongly normalizing*) if for each term t the rules of R cannot be used to re-write t an infinite number of times.

Definition 4. R is *confluent* (also called *Church-Rosser*) if for each term t if $t \twoheadrightarrow_R \tau$ and $t \twoheadrightarrow_R \tau'$ for some terms τ, τ' then $\tau \twoheadrightarrow_R \mu$ and $\tau' \twoheadrightarrow_R \mu$ for some term μ.

Definition 5. R is *complete* if R is both *strongly terminating* and *confluent.*

Definition 6. R is *left-linear* if for each rule $r = (\eta \mapsto \rho) \in R$ each variable $x \in X$ occurs at most once in η.

Definition 7. R is *overlapping* if there exist two rules $r = (\eta \mapsto \rho), r' = (\eta' \mapsto \rho') \in R$ (including the case where $r = r'$) such that there exist ground-term substitutions ς and ς' that make $\varsigma(\eta)$ a proper sub-term of $\varsigma'(\eta')$, and the outermost function symbol of $\varsigma(\eta)$ occurs as part of η'.

Definition 8. R is *orthogonal* if it is left-linear and non-overlapping.

2.4 Streams, Stream Transformers and Stream Processing

We now formalize algebraically the three most fundamental concepts in the context of our research: *streams*, *stream algebras* and *stream transformers*. This also enables us to give a rigorous definition to the ideas of *stream processing systems* and *stream processing*. In particular, we classify the types of stream processing systems that can be found in the literature to allow us to use a concise notation in our literature survey in Chapter 3.

2.4.1 Clocks

A *clock* is any algebra *isomorphic* to the natural numbers; that is, any algebra that is identical up to a re-naming of constants and operations in the underlying signature. We will denote a clock by $T = (T; 0; succ)$ and use the distinguished sort symbol $\mathbf{t}$ to name the set $T = \{0, 1, 2, \ldots\}$. As such, because $\mathbb{N}$ and T are essentially the same algebra in the sequel we will sometimes use $\mathbb{N}$ and T interchangeably and similarly $\mathbf{n}$ and $\mathbf{t}$.

2.4.2 Streams and Stream Algebras

Let A be any standard S-sorted Σ-algebra. For each $s \in S$ we denote a *stream of sort* s by

$$[T \to A]_s.$$

A typical member $a \in [T \to A]_s$ is a function $a : T \to A_s$ such that for each $t \in T$ the tth element of a is written $a(t)$. We will denote the set of all streams over algebra A by $[T \to A]$; that is,

$$[T \to A] = \{[T \to A]_s \mid s \in S\}.$$

In addition, we denote an arbitrary Cartesian product of stream carriers of length $n \in \mathbb{N}$ from A by $[T \to A]^n$. When we wish to be specific about the type of each member of a Cartesian product of stream carriers we will write $[T \to A]^u$ for some $u = s_1 \cdots s_n \in S^+$ to denote

$$[T \to A]_{s_1} \times \cdots \times [T \to A]_{s_n}.$$

We define $\underline{S} = S \cup \{\underline{s} | s \in S\}$ with the intention that $\underline{s}$ names the set $[T \to A]_s$.
We define the $\underline{S}$-sorted signature

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

wherein for each $w \in \underline{S}^*$ and for each $s \in S$

$$\underline{\Sigma}_{w,s} = \begin{cases} \Sigma_{w,s} \cup \{eval_s\} & \text{if } w = \mathbf{t}\ \underline{s}; \\ \Sigma_{w,s} & \text{otherwise.} \end{cases}$$

We define $\underline{A}$ to be the $\underline{S}$-sorted $\underline{\Sigma}$-algebra wherein for each $s \in S$, $\underline{A}_s = A_s$, $\underline{A}_{\underline{s}} = [T \to A]_s$ and $eval_s^{\underline{A}} : T \times [T \to A]_s \to A_s$ is defined by

$$(\forall t \in T)\,(\forall a \in [T \to A]_s) \quad eval_s^{\underline{A}}(t, a) = a(t).$$

2.4.3 Cartesian Products in Stream Algebras

Often when working with stream algebras we will assume that the elements of a Cartesian product $a = (a_1, \ldots, a_n)$ for some $n \in \mathbb{N}$ can be taken from both stream and non-stream carriers. To avoid any confusion in later sections we now clarify the use of our notation when working with vectors in stream algebras.

For any standard algebra A the algebra $\underline{A}$ comprises sets of 'ordinary' data $\underline{A}_s = A_s$ and sets of stream data $\underline{A}_{\underline{s}} = [T \to A]_s$ for each $s \in S$. Furthermore, we will see Cartesian products of data sets over $\underline{A}$ occurring in three ways. First, for $w \in S^*$ the Cartesian product $\underline{A}^w$ is just A^w by the definition of $\underline{A}$; that is, $\underline{A}^w$ is a product of ordinary data sets only. Secondly, again for $w \in S^*$, we define

$$[T \to A]^w = [T \to A]_{w_1} \times \cdots \times [T \to A]_{w_{|w|}}$$

representing a product of stream data sets only. We also use $\underline{A}^{\underline{w}}$ to denote $[T \to A]^w$. Finally, we will see mixtures of ordinary and stream data sets in a Cartesian product such as $\underline{A}^w$ for $w \in \underline{S}^+$. For example, if $w = s_1\, \underline{s}_2\, s_3 \in \underline{S}^*$ then

$$\underline{A}^w = A_{s_1} \times [T \to A]_{s_2} \times A_{s_3}.$$

2.4.4 Stream Transformers

Let A be any S-sorted algebra. If

$$F : [T \to A]^u \times A^x \to [T \to A]^v \times A^y$$

is some function for some $u, v, x, y \in S^*$ such that $|u| + |v| \geq 1$ then we say that F is a *stream transformer*. In particular, initially we will be interested in stream transformers of the form

$$F : [T \to A]^u \to [T \to A]^v;$$

for some $u, v \in S^+$ that is, stream transformers with no non-stream input and no non-stream output. However, in Chapter 4 we will make our theoretical tools sufficiently general to model the extended forms of STs that we will encounter.

2.4.5 Stream Processing

For the purposes of our work we define *stream processing* to be the study of stream transformers. Given that much stream related research is concerned with distributed processing we feel that this definition is appropriate as it encompasses both the theoretical and practical aspects of the field; that is, the theoretical study of STs as abstract functions and also of their (network) implementations. The distinction between abstract STs and their network implementations is made precise in the following section.

2.4.6 Stream Processing Systems

As we have already outlined, within computer science STs are typically implemented and studied as systems composed of a collection of separate, but communicating processes that receive stream data as input and produce stream data as output. In order to emphasize the distinction between STs in abstract and their implementations we will refer to networks comprised of a collection of separate processing elements as *stream processing systems* (SPSs).

A typical SPS with n input streams (*sources*), m output streams (*sinks*), and k processing elements (usually called *filters*) is a distributed processing system θ with functionality

$$\theta : [T \to A]^u \to [T \to A]^v$$

for some $u, v \in S^+$ for some S-sorted algebra A. SPSs are naturally visualized using a directed graph as shown in Figure 2.1 – the reason for the particular annotation on this SPS is made clear in Section 2.5.

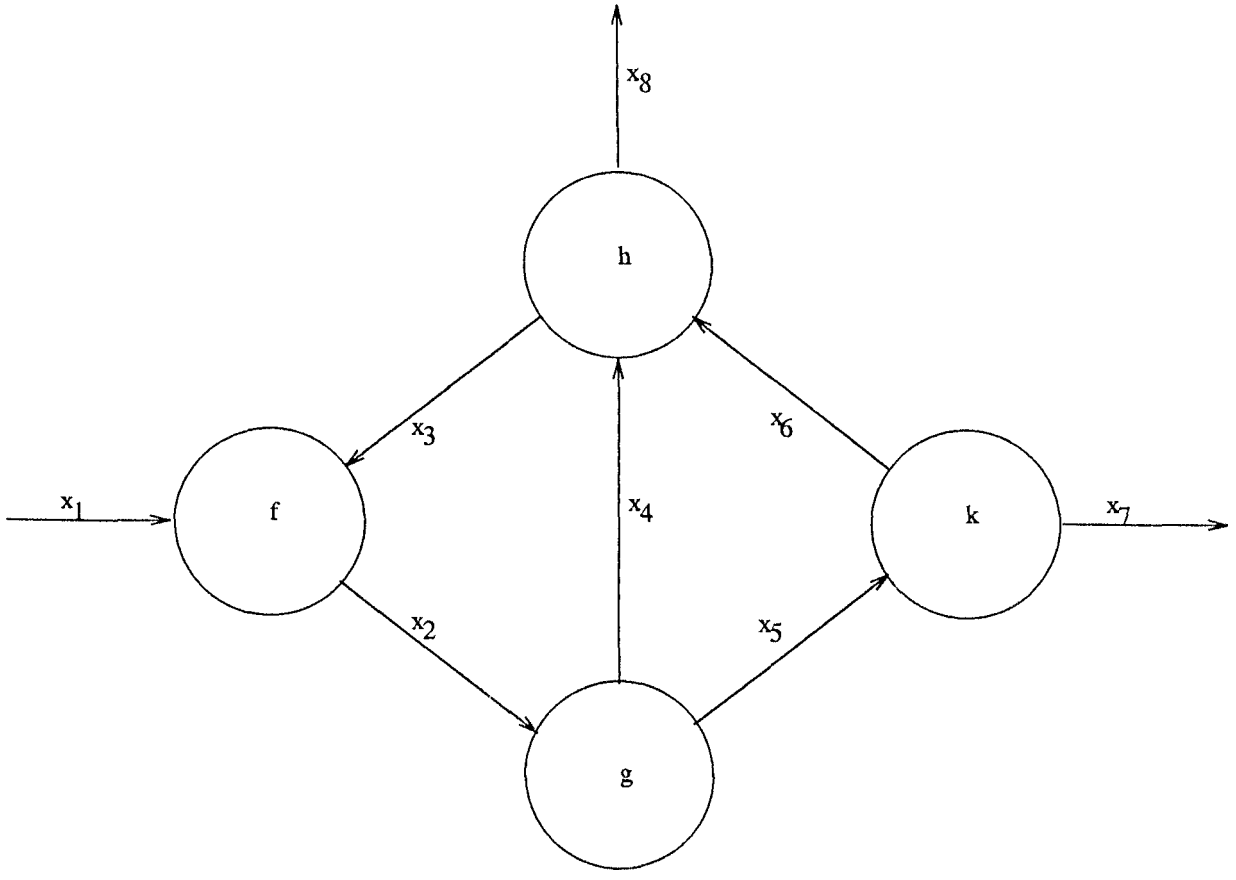

Figure 2.1: A Stream Processing System

2.4.7 Classifying SPSs

Because so much of the stream processing research in the literature is concerned with SPSs it is useful to be able to identify the different types of SPSs concisely. In particular, we will classify SPSs by the following three main characteristics:

(1) Either *synchronous* or *asynchronous* filters.

(2) Either *deterministic* or *non-deterministic* filters.

(3) Either *uni-directional* or *bi-directional* channels.

Furthermore, we will use the following shorthand notation to denote SPSs that are designed to model networks with specific combinations of these properties: $\varsigma\delta\upsilon$-SPS, wherein $\varsigma \in \{S, A\}$, $\delta \in \{D, N\}$, and $\upsilon \in \{U, B\}$. For example, using this classification a synchronous, deterministic SPS with unidirectional channels is denoted SDU-SPS, and an asynchronous, non-deterministic SPS with bidirectional channels is denoted ANB-SPS.

2.5 Kahn's Work

2.5.1 Domain Theory

To understand the traditional approach to formalizing the semantics of SPSs the reader will need to be familiar with the following basic concepts from the theory of *domains*: *complete partial orders* (CPOs), *monotonicity*, *continuity* and *least fixed points*. In particular, the primary theoretical result from domain theory of interest is this section is the following theorem:

Theorem 1. ***Kleene's First Recursion Theorem.*** *Let A and B be any CPOs. If the functional*

$$F : (A \rightsquigarrow B) \rightarrow (A \rightsquigarrow B)$$

is monotone then F has a least fixed point.

For the reader not conversant with these ideas an introduction to domain theory can be found in Stoy [1977], Scott [1982], Plotkin [1983], and Stoltenberg-Hansen *et al.* [1994].

2.5.2 Kahn's Work and the First Recursion Theorem

In 1974 G Kahn published what has become an influential paper in the stream processing literature (see Kahn [1974]), wherein he introduced a simple parallel language for representing SPSs in an ALGOL like style. In particular, it is the generality of the method Kahn introduced to provide a semantics for this language that has enabled his techniques to be used as a language independent denotational semantic model for both SDU-SPSs and some ADU-SPSs (see Section 3.3.3). Indeed, *Kahn's method* has been widely adopted by researchers as a semantic model

for many types of SPSs.

In this section we discuss Kahn's method in some detail in order that we may clarify certain points in later sections when we compare and contrasts existing techniques with Cartesian form specification. Therefore, as we indicated previously, this section can be omitted on a first reading.

Kahn's Motivation. The language that Kahn described in Kahn [1974] was designed to represent fixed, idealized, asynchronous networks of processes communicating by what Kahn called 'FIFO queues'. Kahn invited the reader to visualize such networks as a set of generalized Turing machines (see for example Hopcroft and Ullman [1979]) each with its own work tape, and connected via one-way 'communication tapes'.

Kahn's interest in such a language was not motivated by the development of a user-friendly programming methodology for describing ADU-SPSs. Rather, Kahn was interested in how to prove formally properties of programmes written in such a language. In particular, properties of the networks that this language could describe including computational properties such as termination and non-termination.

A Syntax for Kahn SPSs. Before it was possible to formalize the semantics of Kahn's language he needed to formalize both the syntax and the semantics of the abstract networks he wished to describe, and indeed it is this element of Kahn's work that has been widely adopted by other researchers.

Specifically, Kahn formalised a SPS as a directed graph using the following basic assumptions: (1) arcs are divided into two sets: input and output arcs that are not allowed to either branch or merge (see Section 3.10); and (2) the ith output arc comes from the ith node. However, for the purposes of our explanation of Kahn's work we will adopt a slightly more general framework, although for ease of exposition we will still assume that nodes only have one output arc. In particular, following the style of Kahn, if a node N_i representing a 'process' P_i has $n > 1$ outputs then we will assume that $N_i = (N_1^i, \ldots, N_n^i)$ wherein N_j^i for $j = 1, \ldots, n$ is a sub-node occupying the same position as N^i.

A Semantics for Kahn SPSs. Kahn observed that the (possibly infinite) sequences of data from some data type D passing along the arcs of SPSs (referred to as either queues or *histories*), could be formalized mathematically as the set

$$D^\omega = [T \to D] \bigcup_{n \in \mathbb{N}} D^n;$$

that is, the union of all infinite and finite sequences. (This is the generalized notion of streams that we refer to in the sequel.) Moreover, Kahn also noticed that by associating the usual partial ordering with D^ω that D^ω is a CPO. More specifically, Kahn observed that it is possible to view a network N with $n \in \mathbb{N}$ input arcs and $m \in \mathbb{N}$ output arcs as computing a functional

$$F^N : (D^\omega)^n \to (D^\omega)^m.$$

In addition, with the assumption that each node in our graph representing a processes computes a continuous function the functional F is itself continuous and hence it is possible to apply the First Recursion Theorem to show that the least fixed point of the functional F^N is the required semantics of the network N.

In fact Kahn showed that this technique can also be applied to formalize the semantics of the language that he introduced for describing SPSs. He achieved this as follows.

Kahn SPSs as Systems of Equations. Let P be a programme over some continuous S-sorted algebra A implicitly describing some ADU-SPS N. Furthermore, let the network N consist of $P = (P_1, \ldots, P_k)$ processing elements, $I = (I_1, \ldots, I_l)$ inputs arcs of sort $s_j^I \in S$ for $j = 1, \ldots, l$ and $O = (O_1, \ldots, O_k)$ other arcs of sort $s_j^O \in S$ (either output or internal) for $j' = 1, \ldots, k$ for some $k, l \in \mathbb{N}^+$; wherein associated with each P_i for $1 = 1, \ldots, k$ we have:

(1) A function

$$f_i^A : A^\omega_{s_{i,1}} \times \cdots \times A^\omega_{s_{i,n_i}} \to A^\omega_{s_i}$$

for some $s_i \in S$ and for some $s_{i,p} \in S$ for $p = 1, \ldots, n_i \in \mathbb{N}^+$ and

(2) Three architecture indicators:

$$\tau^i = (\tau^i_1, \ldots, \tau^i_{n_i}) \in \{I, O\}^{n_i},$$

$$\eta^i = (\eta^i_1, \ldots, \eta^i_{n_i}) \in \{1, \ldots, max(k, l)\}^{n_i},$$

and

$$\chi : \{1, \ldots, k\} \to \{1, \ldots, k\}$$

defined such that for $q = 1, \ldots, n_i \in \mathbb{N}^+$

$$\tau^i_q = \begin{cases} \mathrm{I} & \text{if the } q\text{th input to process } i \text{ is from an input arc, and} \\ \mathrm{O} & \text{if the } q\text{th input to process } i \text{ is from an internal arc;} \end{cases}$$

$$\eta^i_q = \begin{cases} \mathrm{v} & \text{if the } q\text{th input to process } i \text{ is from } I_v \text{ for some } v \in \{1, \ldots, l\}, \text{ and} \\ \mathrm{w} & \text{if the } q\text{th input to process } i \text{ is from } O_w \text{ for some } w \in \{1, \ldots, k\}; \end{cases}$$

and for each $n \in \{1, \ldots, k\}$

$$\chi(n) = m \iff P_m\text{'s output is arc } O_n.$$

Finally, let $a = (a_1, \ldots, a_l) \in (A^{s_1^I \cdots s_l^I})^\omega$ be the input to programme P.

Constructing Equations to Formalize P. Kahn's method is now to construct a system of equations $E_P(\Sigma \cup \mathbb{F}, X_P)$ wherein $\mathbb{F} = \{f_1, \ldots, f_k\}$ such that $(\bar{b}_1, \ldots, \bar{b}_m)$ is a solution to the

system of equations $E_P(\Sigma \cup \mathbb{F}, X_P)$ if an only if $F^N(a_1, \ldots, a_l) = (b_1, \ldots, b_m)$. To achieve this he defined $E_P(\Sigma \cup \mathbb{F}, X_P)$ as follows:

$$
\begin{aligned}
E_P(\Sigma \cup \mathbb{F}, X_P) = \{ & \\
& x_1^I = \overline{a}_1, \\
& \vdots \\
& x_l^I = \overline{a}_l, \\
& x_1^O = f_{\chi(1)}(Y_{1,1}, \ldots, Y_{1,n_1}), \\
& \vdots \\
& x_k^O = f_{\chi(k)}(Y_{k,1}, \ldots, Y_{1,n_k}) \quad \}
\end{aligned}
$$

wherein for $i = 1, \ldots, k$ and for $j = 1, \ldots, n_i$

$$Y_{i,j} = x_\eta^\tau$$

wherein

$$\tau = \tau_j^{\chi(i)}$$

and

$$\eta = \eta_j^{\chi(i)}.$$

Example 1. Following the example given in Kahn [1974], from the program describing the SPS shown in Figure 2.1 we derive the following:

$$P = (P_1, \ldots, P_7) = (P_1^1, P_1^2, P_2^2, P_1^3, P_2^3, P_1^4, P_2^4),$$

$$I = (I_1),$$

$$O = (O_1, O_2),$$

$$f_1^A = f : A \times A \to A,$$

$$f_2^A = g_1 : A \to A,$$

$$f_3^A = g_2 : A \to A,$$

$$f_4^A = h_1 : A \times A \to A,$$

$$f_5^A = h_2 : A \times A \to A,$$

$$f_6^A = k_1 : A \to A,$$

$$f_7^A = k_2 : A \to A,$$

$$\tau^1 = (\tau_1^1, \tau_2^1) = (I, O),$$

$$\eta^1 = (\eta_1^1, \eta_2^1) = (1, 2),$$

$$\tau^2 = (\tau_1^2) = \tau^3 = (\tau_1^3) = (I),$$

$$\eta^2 = (\eta_1^2) = \eta^3 = (\eta_1^3) = (1),$$

$$\tau^4 = (\tau_1^4, \tau_2^4) = \tau^5 = (\tau_1^5, \tau_2^5) = (O, O),$$
$$\eta^4 = (\eta_1^4, \eta_2^4) = \eta^5 = (\eta_1^5, \eta_2^5) = (3, 5),$$
$$\tau^6 = (\tau_1^6) = \tau^7 = (\tau_1^7) = (I),$$
$$\eta^6 = (\eta_1^6) = \eta^7 = (\eta_1^7) = (1),$$

and

$$\chi = \{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 7, 6 \mapsto 6, 7 \mapsto 5\};$$

and

$$\begin{aligned} E_P(\Sigma \cup \{f_1, \ldots, f_7\}, X_P) = \{ \quad & \\ & x_1^I = \overline{a}, \\ & x_1^O = f(x_1^I, x_2^O), \\ & x_2^O = h_1(x_3^O, x_5^O), \\ & x_3^O = g_2(x_1^O), \\ & x_4^O = g_1(x_1^O), \\ & x_5^O = k_2(x_4^O), \\ & x_6^O = k_1(x_4^O), \\ & x_7^O = h_2(x_3^O, x_5^O) \quad \}. \end{aligned}$$

As Kahn now observed it is a well-known mathematical result (Kahn cited Milner [1973], but a more recent reference is Stoltenberg-Hansen *et al.* [1994]) that such a system of equations over CPOs admits a unique minimal solution; that is, a least fixed point, and in particular this least fixed point is the value

$$F^N(a_1, \ldots, a_l) = (b_1, \ldots, b_k)$$

that we require. (For a proof of this fact Kahn directed the reader to Cadiou [1972].)

Furthermore, Kahn observed that by adopting a fixed-point semantic approach that Scott's Induction Rule (Kahn cited Manna *et al.* [1973]) and several techniques for proving properties of recursive programs found in Vuillemin [1973] are now available including structural induction and recursion induction.

Conclusion. While we admire Kahn's work we have several specific objections to the methods he developed from the perspective of the development of software tools and from the perspective of the automated verification of SPSs. We return to these points in Section 3.10.

Chapter 3

A Survey of Stream Processing

Comparisons are odourous.

Shakespeare

3.1 Introduction

The origins of stream processing research can be traced back at least as far as the early 1960s, although not always in a form that is immediately recognizable as such today. Since that time individual research has typically been applications driven and has concentrated on developing specialized stream processing systems rather than developing a theory concerned with the general issues arising out of stream-based computation. For example, *dataflow* is considered to be a canonical example of stream processing research, but dataflow is predominately concerned with the development of parallel processing techniques. In particular, this is highlighted by the fact that dataflow is often considered to be a specialized implementation method for functional programming rather than a separate area of research. Indeed, in general we believe it is fair to observe that with a few exceptions stream processing systems have essentially been used as a convenient tool in research that has been concerned with other issues. Consequently, a coherent theory of stream processing based on the study of abstract stream transformers has not emerged in the literature.

In light of the rather eclectic nature of stream processing research, especially from the perspective of semantics, we believe it is particularly important to examine the existing stream processing literature in some detail. Therefore, before we present an overview of the main topics that we will address in this thesis we will use a literature survey to motivate the theoretical and practical issues that we wish to address.

In Section 3.2 we begin our analysis of the literature with a brief historical perspective of the development of stream processing from the early 1960s to the present day.

The three sections following this overview are devoted to a more detailed analysis of some of the individual approaches to stream processing: *dataflow* (Section 3.3); specialized functional and logic programming (Section 3.4); and reactive systems and signal processing (Section 3.5). In each case we discuss the basic motivations and ideas underlying each paradigm. However, for convenience in order to clarify certain issues relating to computability theory and language design we have deferred the topic of stream processing primitives and languages arising in this research until Sections 3.7 and 3.8. The first part of our literature survey is concluded in Section 3.6 where we mention briefly some topics related to stream processing.

For emphasis, in Section 3.9 we discuss a topic of particular interest in this thesis: the use of streams in the design and verification of hardware. This leads into our discussion in Section 3.10 that gives a detailed summary of the research presented in this thesis and how it relates to the existing stream processing research that we have discussed. In particular, we motivate what we believe are the the advantages of the use of algebraic techniques in stream processing including an overview of SCA theory that is the starting point for much of our own research.

3.2 A Brief History of Stream Processing

In this section we present a brief historical perspective of the development of stream processing over the last four decades. Of course given our general definition of stream processing it is impractical to survey every paper that is in some way related to the use of streams as either a theoretical or practical tool. Therefore, we mention only either well-known research or research

that we believe is representative of a particular topic within stream processing, and that provides a useful starting point for any further reading the reader may wish to undertake. Research topics that we believe are most important are analysed in more depth in the following sections.

3.2.1 The 1960s

Within computer science the term *stream* has been attributed to P J Landin (see Burge [1975]) formulated during his work on the correspondence between ALGOL 60 and the λ-calculus (see Landin [1965a] and Landin [1965b]).

The first type of SPSs that can be identified within the literature are *dataflow systems* that have certainly existed, although not always under the name 'dataflow', as early as the late 1960s (see for example McIlroy [1968] and Adams [1969]). The term dataflow originates from the term *data flow analysis* (see Ackerman [1979]) used to evaluate potential concurrency in computations.

3.2.2 The 1970s

The first dataflow language, and probably still the most famous, is LUCID (see Wadge and Ashcroft [1985]) that was conceived in 1974. LUCID is based in part on the language POP-2 (see Burstall *et al.* [1971]), that allowed a limited use of streams. Other relevant dataflow references from the 1970s are Adams [1970], Kosiniski [1973], Dennis [1974], Weng [1975] and Arvind *et al.* [1979]).

As discussed in Section 2.5 in 1974 G Kahn published his well-known work outlining a simple parallel programming language designed for representing SPSs using a fixed-point semantics. The use of a fixed-point semantics for SPSs in the style of Kahn's work is common in the literature, and for this reason SPSs are sometimes referred to as Kahn networks.

In 1975 W Burge (see Burge [1975]) discussed the use of streams as a method for structured programming and introduced a set of functional stream primitives for this purpose.

In 1976 P Henderson and J H Morris (see Henderson and Morris [1976]) and D P Friedman and D S Wise (see Friedman and Wise [1976]) published their work on lazy evaluation techniques that are useful for computing with infinite data types of which streams are an example.

In 1977 G Kahn made another contribution to the field with his joint paper with D MacQueen (see Kahn and MacQueen [1977]) wherein they introduced a language designed to model distributed process interaction using ideas from Kahn [1974].

3.2.3 The 1980s

Dataflow continued to be an area of widespread research during the 1980s and several additional semantic models for dataflow were introduced, for example, Faustini [1982], Bergstra and Klop [1983], Staples and Nguyen [1985], Stefanescu [1987a], Stefanescu [1987b], Kok [1987a] and Jonsson [1988].

Logic programming languages also began to be used to model SPSs. A modification of PROLOG used to model what have become termed *perpetual processes* (see Lloyd [1984]) within logic programming was introduced in Bellia *et al.* [1982].

Functional programming languages were also used widely to model SPSs. Notable in this area is the work of M Broy and his use of functional languages to study stream based distributed processing (see for example Broy [1983], Broy [1986], Broy [1987a], and Broy [1987b]).

In 1985 the first paper on the subject of *synchronous concurrent algorithms* (SCAs) was released. Conceived by B C Thompson and J V Tucker, SCAs have been the stimulation for much of our own work into stream processing (see Section 3.10 where we give a full list of references on the subject of SCAs).

The year 1985 also saw the publication of a paper by D Harel and A Pneuli (see Harel and Pnueli [1985]) on the subject of *reactive systems*. Reactive systems, together with *signal processing networks* and *synchronous dataflow networks* – that can be considered as special cases of reactive systems – have been the stimulation for a large body of stream processing research (see for example Guatier *et al.* [1987], Caspi *et al.* [1987] and Berry *et al.* [1988]).

During the 1980s streams and STs have also been used extensively for hardware description, for example, Sheeran [1983], Sheeran [1986], Kloos [1987a], Kloos [1987b], Harman and Tucker [1988c] and Harman [1989] (also see Section 3.9).

3.2.4 The 1990s

As with the 1980s semantic models for dataflow are still being developed, for example, Kearney and Staples [1991], Bartha [1992a], Bartha [1992b], France [1992], and Lee and Tan [1992], although the work in Bartha [1992a], Bartha [1992b] is concerned with *flowchart schemes* (see Leiserson and Saxe [1983]) that has applications to the study of dataflow schemes. A useful overview of the concept of dataflow with an extensive bibliography can be found in Sharp [1991].

SCAs continue to be an intensive area of research (again see Section 3.10 for references) as does research into reactive systems (see for example Beveniste and Berry [1991], Halbwachs *et al.* [1991], Guernic *et al.* [1991], Ratel *et al.* [1991] and Halbwachs *et al.* [1992]).

The 1990s have also produced a body of work concerned with the theoretical foundations of stream processing. In 1992 J V Tucker and J I Zucker (see Tucker and Zucker [1992]) released the first in a series of generalizations of computability theoretic results from the natural numbers to algebras with streams. This work is continued in Tucker and Zucker [1994]. In addition, Stephens and Thompson [1992] presented a theoretical study of the compositional properties of STs in *Cartesian form* (Chapter 4).

The theoretical work of K Meinke has applications to the specification, verification and parameterization of STs (see Meinke [1990], Meinke [1991a], Meinke [1991b], Meinke [1992a], Meinke [1992b]) as does the work of K Meinke with L J Steggles and B M Hearn (see Meinke and Steggles [1992] and Hearn and Meinke [1993] respectively).

Finally, M Broy continues his functional study of distributed processing over streams (see for example Broy [1990], Broy and Dendorfer [1992], and in particular Broy *et al.* [1993]).

3.3 Dataflow

As dataflow networks were the first type of SPSs to appear in the literature we begin our more detailed survey with an examination of the research aims of dataflow and an analysis of the semantic models and implementation techniques that have been developed. An more detailed introduction to the concept of dataflow can be found in Sharp [1991].

3.3.1 Origins

As we have mentioned dataflow research began as far back as the 1960s and continues to be an area of widespread research. One of the continuing aims of the dataflow approach has been to avoid the so-called 'von Neumann bottleneck' (see Backus [1978] and Backus [1981]) and exploit the parallelism offered by VLSI technology. As part of this research many experiments with specialized architectures have been undertaken (the interested reader can consult the bibliography of Sharp [1991] for a list of references).

We note in passing at this point that it has been observed that the link between J von Neumann and sequential computing methods is historically inaccurate, as he was one of the early advocates of parallel computing methodologies (Kiliminster [1993]). However, we use this phrase as it can be found in the dataflow literature.

3.3.2 Dataflow Networks

A classical dataflow network is an ADU-SPS, although dataflow computation based on ANU-SPSs has also been studied, and more recently dataflow computation based on SDU-SPSs has been of particular interest. The filters within a dataflow network (sometimes referred to as *coroutines* – see McIlroy [1968]– and also *agents*) compute over streams; that is, $[T \rightarrow A]$ wherein A is usually restricted to *int*, *bool*, *real* and lists of these types.

3.3.3 Dataflow Computation and Semantics

The dataflow model of computation can be divided into two basic forms: *data driven* wherein filters compute depending upon the availability of data at their inputs; and *demand driven* wherein filters request data on the input lines when they wish to compute.

Both of these approaches to dataflow computation can be formalized denotationally in the style of Kahn. Indeed, as we have discussed (see Section 2.5) a domain-theoretic semantics is common for dataflow languages. However, as discussed in Kok [1987b] Kahn's method is not sufficient for some more general classes of dataflow network, for example non-deterministic models of dataflow computation. Furthermore, 'straightforward' extensions to the Kahn semantic model to cope with non-determinism can fail to be compositional (see Brock and Ackerman [1981] and also Rabinovich [1993]). Consequently for this and other reasons many other semantic models have been formulated for dataflow. For example, some recent references include Faustini [1982], Bergstra and Klop [1983], Staples and Nguyen [1985], Kok [1987a], Kearney and Staples [1991], Lee and Tan [1992], and France [1992].

Despite the many semantic models for dataflow none seems to have been widely adopted. In

addition, the formal relationship between these different approaches is poorly addressed in the literature as is the correspondence between the model of computation provided by dataflow and formal models of computation. Indeed, it is interesting that despite the fact that the dataflow approach is in many senses closely related to CCS (see Milner [1989]) that (as far as we are aware) dataflow has not been formalized using this well-developed formalism. It has been suggested that this is due to the 'value' passing nature of dataflow networks that CCS does not handle concisely.

3.3.4 The Uptake of Dataflow

Despite the extensive body of dataflow research the dataflow approach appears to have had little impact on the traditional approach to 'von Neumann computing'. Indeed, even the most well-known dataflow language LUCID (see Section 3.8.4) has been described as 'a language looking for an application'.

The reasons for the poor uptake of the dataflow approach may stem from two different areas: on the practical side, implementation of the dataflow approach on conventional architecture leads to inefficiencies, including large and wasteful memory usage, that has required the development of specialized architectures; and on the theoretical side, as we have already mentioned, the lack of a clear and straightforward semantics.

Perhaps the recent move toward more parallel architectures will change the current attitude toward dataflow and the general usage of dataflow techniques.

3.3.5 Synchronous Dataflow

The asynchronous nature of dataflow can lead to problems with non-determinism and associated anomalous behaviour (see Broy [1990]); and cyclic networks can suffer from deadlock (see Wadge [1981] and Misunas [1975]). Synchronous dataflow has been developed to avoid these problems. While each filter in a synchronous dataflow network still has its own clock, rather than a global clock as the name might suggest, the interplay between these clocks is restricted and ensures synchronous (and hence deterministic) behaviour.

We discuss synchronous dataflow more fully in Section 3.8.5 when we examine the language LUSTRE that is used to describe synchronous dataflow networks.

3.4 Specialized Functional and Logic Programming

Functional and logic programming comprises a very broad and diverse area of past and current research within computer science. However, the theoretical approaches used to incorporate streams into functional and logic programming languages are in many cases closely related, and are essentially that of a domain-theoretic approach. Indeed, it is for this reason that we have grouped these two areas of research together into a separate section of our literature survey. A detailed discussion of the domain-theoretic relationship between functional and logic programming languages can be found in Silbermann and Jayaraman [1992].

In Section 3.4.1 we examine the functional approach to stream processing and in particular

the work of M Broy.

In Section 3.4.2 we look and logic programming with streams, and in detail at a modification of PROLOG that can be used for stream processing.

3.4.1 Functional Approaches to Stream Processing

The use of the function abstraction operator (λ-abstraction) provides a mechanism for the representation of STs in functional languages in both second-order and higher-order forms. Indeed, most dataflow languages are functional languages, and some researchers regard dataflow as a particular implementation technique for the functional paradigm (see the bibliography of Sharp [1991] for a list of references on this subject). In particular, within functional programming, STs are often referred to as being in *data passing form* and higher-order STs (third-order or above) are referred to as being in *agent passing form.*

As dataflow languages and functional languages are closely related, in some sense any functional programming language can be considered suitable for general purpose stream programming. For example, the well-known functional languages LISP, ML and MIRANDA (see Milner [1984] and Turner [1985]) can all be used to represent STs. However, whether such languages provide a natural and straightforward mechanism for the specification of STs is less clear and for this reason several specialized stream orientated functional languages have been developed including ARTIC (see Dannenberg [1984]), HOPE (see Burstall *et al.* [1980]) and RUTH (see Harrison [1987]) designed to meet more specific needs such as real-time programming over streams.

Functional SPSs and Semantics. Typically ADU-SPS and ANU-SPS are studied using the functional paradigm and as with dataflow languages the work of G Kahn has been widely adopted as a semantic approach for functional stream processing. However, other (sometimes related) approaches are also used including *greatest fixed points* (see de Roever [1978] and Gordon *et al.* [1979]) and *Aczel's logical theory of constructions* (see Dyber [1985] and Dyber and Sander [1988]). In addition, the work of Friedman and Wise [1976] and Henderson and Morris [1976] on *lazy evaluation* has provided an implementation technique for functional stream processing that has been widely adopted.

Applications. The verification of functionally specified STs has been explored in the literature. In particular, operating systems have been an area of quite extensive research (see Jones and Sinclair [1989] for an overview) as the swapping of processes can be modelled using agent passing stream transformers. An example of operating system specification can be found in Broy and Dendorfer [1992] and in addition this paper provides an example of how ANU-SPS can be specified using *classes* of functions.

As a more detailed example of functional stream processing research we now discuss the work of M Broy who has made a significant contribution to the development of techniques for functionally based stream processing. In particular, we discuss the FOCUS project that provides a functional framework for the specification of distributed systems based on stream communication.

The FOCUS Project. FOCUS (see Broy *et al.* [1993]) is based on the work developed in Broy [1986], Broy [1988a], Broy [1988b], Broy [1989], Broy [1990], Broy and Lengauer [1991], Broy [1992a], and Broy [1992b].

FOCUS is not a language, but rather a collection of tools and modelling concepts that provide a framework for the description of parallel distributed systems as concurrent asynchronous processing elements. Within such networks data is exchanged via unbounded FIFO channels that are modelled as streams.

FOCUS aims to provide a theory of stepwise refinement and modular development of parallel systems and includes verification calculii that are intended to provide a formal system to reason about the correctness of system implementations at various level of abstraction. However, it is not the intention of FOCUS to provide a theory of stream based distributed processing (see Broy *et al.* [1993]).

Despite the fact that FOCUS is a paradigm and not a language it does provide two concrete representations for expressing STs (as SPS). The first (and most abstract in the sense of specification) is the language AL based on AMPL (see Broy [1986]) and the second is the language PL based on the work in Broy and Lengauer [1991] and Dederichs [1992]. We discuss the languages AL and PL in Sections 3.8.9 and 3.8.10 respectively.

Given the specification of an ST in AL the FOCUS paradigm provides transformational rules (refinements) towards more concrete representations (in the sense of specification). Indeed within FOCUS a representation is considered to be in its most concrete form (an implementation) if no further refinements and no further re-writings to another formalism (representation) are possible. Given this definition the implementation language of FOCUS can be consider to be PL, although one can imagine that these techniques could be extended to additional languages.

3.4.2 Logic Programming Languages with Streams

As logic programming provides a high-level and useful method of specification for some classes of systems it is natural that some researchers have explored the use of logic programming languages for the specification of SPSs. Indeed, there are several examples of modifications of relational languages for stream processing that can be found in the literature. In Parker [1990] these languages are divided into three groups:

(1) *Committed choice* parallel programming systems, for example, *PARLOG* (see Clark and Gregory [1985]).

(2) Extension of PROLOG to include either the *parallel and* or *parallel or* operators (see for example Li and Martin [1986]).

(3) Extension to PROLOG to include functional constructs, for example, Kahn [1984], Lindstrom and Panangaden [1984], Subrahmanyam and You [1984], Naish [1985], Bellia and Levi [1986], and DeGroot and Lindstrom [1986].

However, a different classification can be found in Bellia and Levi [1986] wherein logical languages for programming with streams are divided into two groups:

(A) Languages based on *static input-output mode variable declarations* for example the languages of Clark and Gregory [1981] and van Emden and de Lucena Filho [1982].

(B) Languages based on *dynamic variable annotations* for example Clark and Gregory [1983], Shapiro [1983] and Subrahmanyam and You [1984].

While we are not aware of any work in the literature that describes the relationship between these two classifications, it is possible to make the following general comments on the methods used to incorporate the use of streams in logic programming.

Describing SPSs as Relations. The use of the term 'coroutine' in relational languages does not directly imply the use of streams (see Parker [1990]). However, typically logic programming languages modified for stream programming are designed to represent ADU-SPSs, although the particular description of ADU-SPSs will of course depend on the stream processing operations and types of concurrency allowed in the particular language.

The Use of Streams. As with the functional approach streams are treated as the union of finite and infinite sequences. In particular, streams are typically implemented as finite lists, although the declaration and manipulation of infinite lists (and hence streams) may be permitted. However, the use of infinite lists in some relational languages may be non-terminating as they tend to use eager evaluation.

Specialized logic programming languages extended with non-strict processes and lazy evaluation to cope with stream programming are sometimes termed *perpetual processes* (see Lloyd [1984]) and have many similarities with functional languages. (A survey of the relationship between logical and functional languages can be found in DeGroot and Lindstrom [1986] and Bellia and Levi [1986].) Alternatively, relational languages can be modified to cope with streams by eliminating the *occurs check*, although this can lead to 'unsound inferences' (see Parker [1990]).

Semantics. Several semantic approaches have been adopted for dealing with perpetual processes including a fixed-point semantics in the style of Kahn. A discussion and comparison of these approaches can be found in Levi and Palamidessi [1988].

Languages. In addition to the specialized logic programming languages we have already mentioned in Section 3.4.2 we look in detail at a modification of PROLOG to cope with the use of streams.

3.5 Reactive Systems and Signal Processing Networks

The *reactive system* paradigm (see Harel and Pnueli [1985]) and *signal processing* paradigm are conceptually closely related. The essential difference between the two approaches is that reactive system research is concerned with SDB-SPSs and signal processing is concerned with SDU-SPSs; that is, in reactive systems channels are bidirectional. Consequently, from this point we will use the term 'reactive systems' to mean both reactive systems and signal processing networks.

Reactive systems are designed to model real-time systems such as operating systems and process control programs that 'repeatedly respond to inputs from their environment by producing outputs'. Stream communication provides a natural method for the specification of real-time systems. However, real-time system specification is not limited to this technique and is the reason that in general real-time system theory is less related to stream processing than the specialized real-time system research explored in reactive system theory. Therefore in this section we discuss reactive systems as a separate topic.

3.5.1 Streams, Signals and Sensors

Reactive systems and signal processing systems communicate via signals that are related to our concept of streams. Signals are divided into two types: *pure signals* that are un-typed and simply communicate an 'event' that can be used for synchronization; and typed signals that communicate data. Within the reactive system paradigm signals may be used for both input and output, but we note that typed signals are only used for input and are referred to as *sensors*. Given this informal definition signals in signal processing networks are all sensors. A comparison of typed signals and streams can be found in Section 3.8.7.

3.5.2 The Strong Synchrony Hypothesis and Multiform Time

The reactive system paradigm is based on what is referred to as either the *strong* or *perfect synchrony hypothesis* (see Berry and Gonthier [1988]) that requires all filters within a network to react instantly to input producing a corresponding output in zero time. As a consequence the whole computation performed by a reactive system is 'instantaneous'. In addition, reactive systems also use what is referred to as a *multiform* notion of time (see Berry and Gonthier [1988]) wherein signals (streams) may be used as a time unit. As such co-operation of sub-tasks (processes) defines new temporal relations that are used to define the global ordering of the data (compare Harman [1989] – see Section 3.9).

Semantics. The semantics of reactive systems have been formalized using *temporal logic* (see Pnueli [1986]). In addition, Pnueli [1986] also includes a comparison of several different semantic approaches to general concurrent systems and how these approaches can be applied to reactive systems.

Languages. In Sections 3.8.5, 3.8.7, and 3.8.8 we describe three languages for programming reactive systems, respectively LUSTRE, SIGNAL, and ESTEREL, and contrast the different approaches that they take.

3.6 Other Stream Processing Formalisms

3.6.1 ALPHA

While the language ALPHA (see Dezan *et al.* [1992]) is not specifically a stream processing language we mention ALPHA here as it is described by its authors as '...a grandson of LUCID...'

and is used for the design and synthesis of systolic VLSI (see Verge *et al.* [1991]). In particular, ALPHA is an equational language that involves a generalization of stream variables that can be used to represent a 'spatial domain'; that is, a (possibly infinite) matrix indexed by a sub-set of $\mathbb{Z}^n$. For example, if variable X is declared on the domain D defined by

$$D = \{(i,j) \mid i > 0, 1 \leq j \leq 2\}$$

then X represents a matrix

$$\begin{bmatrix} x_{1,1}, x_{1,2}, x_{1,3}, \ldots \\ x_{2,1}, x_{2,2}, x_{2,3}, \ldots \end{bmatrix}.$$

Using this methodology if variable Y was declared over domain D' defined by

$$D' = \{i \mid i > 0\}$$

then Y would essentially be a stream

$$y_0, y_1, y_2, y_3, \ldots.$$

To compute over spatial domains ALPHA uses a generalization of point-wise extensions called 'motionless operators' whose semantics is formalized denotationally in the style of Kahn.

3.6.2 Stream X-Machines

Stream X-machines (see Holcombe and Ipate [1994]) are based on the X-machine model of computation (a generalization of the Turing Machine – see Holcombe [1988]) that allow streams as both input and output. Stream X-machines have been used in the study of system testing and verification for which the authors claim they offer significant advantages.

3.7 Stream Processing Primitives and Constructs

3.7.1 Introduction

As we have already discussed one of the main motivations of our reserach is the development of a mathematically well-founded, high-level language that is suitable for the formal specification and formal verification of systems that compute over streams. In particular, we require a formalism for representing STs that is independent of implementation issues; that is, a specification language for STs that does not either require or imply that any implementation will use a specific type of SPS. Indeed, we require that the representation of a ST does not imply an implementation as a SPS at all. Furthermore, we also require that the specification language does not imply the use of any particular language for the representation of a suitable implementation of the abstract ST.

Of course it is fair to observe that these pre-conditions for our stream processing language seem rather obvious. Moreover, it is correct to observe that these requirements are necessary of any specification language that can be classed as abstract whether it specifies either stream

based computation or not. Despite this fact, as we will show in this section, given the preoccupation of stream processing research with distributed processing we believe none of the specialized languages that exists in the literature satisfy these requirements. Indeed, we believe that this fact becomes self-evident from even an informal examination of these languages' syntax and stream processing constructs. In contrast, we believe that these languages should be considered as useful specialized implementation languages.

However, we still believe that a detailed analysis of these languages is appropriate as in Chapter 6 it will enable us to demonstrate that our specification language ASTRAL is sufficiently expressive to represent abstract specifications of implementations written in many of the existing stream processing languages that we discuss. Furthermore, this analysis will enable us to discuss computability theoretic issues relating to the use of each stream processing primitives.

Therefore, in this section we analyse in some detail the abstract stream processing primitives and constructs that can be found in the literature; and in the following section we look at specific languages that can be found to specify the particular classes of stream processing systems that we discussed in Sections 3.3, 3.4 and 3.5.

3.7.2 Common Functional Stream Processing Operations

As explained in Section 2.5 it is common for stream processing languages to be formalized using a domain-theoretic semantics. As we have discussed a domain-theoretic semantics requires that all operations be continuous, and further that streams be generalized to include finite sequences with an appropriate partial ordering. (This is true of most functional approaches to stream processing.)

In this section we describe informally the typical functional stream processing primitives that can be found in the literature using the generalized concept of a stream. (However, we note in passing that these primitives are used in other formalisms as well, sometimes under a different name.) This formalization is based on the description given in Broy and Dendorfer [1992].

Functional Stream Processing Primitives Let A be any continuous algebra with an appropriate partial ordering for each carrier. We use $A^\omega = [T \rightarrow A] \bigcup A^*$ to denote the set of all finite and infinite sequences (generalized streams) wherein $<> \in A^\omega$ denotes the empty sequence and $\twoheadrightarrow$ denotes a continuous mapping.

(1) Stream construction operator. We define the *stream construction operator*, denoted :, with functionality $: : A \rightarrow A^\omega \twoheadrightarrow A^\omega$ by (in infix notation)

$$(\forall a \in A)\,(\forall s \in A^\omega) \quad a{:}s = s'$$

wherein if $|a| < |\mathbb{N}|$ then

$$(\forall t \in \{0, \ldots, |s| + 1\}) \quad s'(t) = \begin{cases} a & \text{if } t = 0; \\ s(t-1) & \text{otherwise} \end{cases}$$

and if $|a| = |\mathbb{N}|$ then $s' = a$.

(2) Concatenation. We define the *concatenation operator*, denoted $\bullet$, with functionality $\bullet : A^\omega \to A^\omega \to A^\omega$ by (in infix notation)

$$(\forall a \in A^\omega) \quad <> \bullet a = a$$

and

$$(\forall (a{:}s), s' \in A^\omega) \quad (a{:}s) \bullet s' = a{:}(s \bullet s').$$

(3) First element selection. We define the *head operator*, denoted $\boldsymbol{hd}$ (and also $\boldsymbol{first}$), with functionality $\boldsymbol{hd} : A^\omega \to A^\perp$ by

$$\boldsymbol{hd}.<> = \perp$$

and

$$(\forall (a{:}s) \in A^\omega) \quad \boldsymbol{hd}.(a{:}s) = a.$$

(4) First element elimination. We define the *tail operator*, denoted $\boldsymbol{tl}$ (and also $\boldsymbol{rest}$), with functionality $\boldsymbol{tl} : A^\omega \to A^\omega$ by

$$\boldsymbol{tl}.<> = <>$$

and

$$(\forall (a{:}s) \in A^\omega) \quad \boldsymbol{tl}.(a{:}s) = s.$$

(6) Last element selection. We define the *last operator*, denoted $\boldsymbol{last}$, with functionality $\boldsymbol{last} : A^\omega \to A^\perp$ by

$$(\forall s \in A^\omega) \quad \boldsymbol{last}.s = \begin{cases} \perp & \text{if } s = <> \text{ or } |s| = |\mathbb{N}|; \\ a & \text{if } s = < a > \text{ for some } a \in A; \\ \boldsymbol{last}.(\boldsymbol{tail}.s) & \text{otherwise.} \end{cases}$$

(7) Filtering. We define the *filter operator*, denoted ©, with functionality $\copyright : \wp(A) \to A^\omega \to A^\omega$ by (in infix notation)

$$(\forall S \in \wp(A)) \quad S \copyright <> = <>$$

and

$$(\forall S \in \wp(A))\,(\forall (a{:}s) \in A^\omega) \quad S \copyright (a{:}s) = \begin{cases} S \copyright s & \text{if } a \notin S; \\ a{:}S \copyright s & \text{otherwise.} \end{cases}$$

(8) **Pointwise change.** We define the *pointwise change operator*, denoted $.[. \mapsto .]$, with functionality $.[. \mapsto .] : A^\omega \to T \to A \to A^\omega$ by (in infix notation)

$$(\forall s \in A^\omega)\,(\forall t, t' \in T)\,(\forall a \in A) \quad s[t \mapsto a](t') = \begin{cases} s(t') & \text{if } t' \neq t; \\ a & \text{otherwise.} \end{cases}$$

It is also common in functional stream processing to use the following two higher-order primitives that act directly on STs themselves.

(A) **After.** We define the *after operation*, denoted $\ll$, with functionality $\ll : (A^\omega \to B^\omega) \to A \to (A^\omega \to B^\omega)$ by (in infix notation)

$$(\forall f \in (A^\omega \to B^\omega))\,(\forall a \in A)\,(\forall s \in A^\omega) \quad (f \ll a).s = f.(a{:}s).$$

(B) **Then.** We define the *then operation*, ambiguously denoted $\ll$, with functionality $\ll : B \to (A^\omega \to B^\omega) \to (A^\omega \to B^\omega)$ by (in infix notation)

$$(\forall b \in B)\,(\forall f \in (A^\omega \to B^\omega))\,(\forall s \in A^\omega) \quad (b \ll f).s = b{:}f.s.$$

3.7.3 Stream Processing Primitives in Logic Programming

In this section we identify four generic stream processing primitives that can be found in the logic programming literature. We conclude the section with some concrete examples of these types of stream processing primitives based on the list given in Becker and Chambers [1984].

Generic Relational Stream Processing Primitives. In Parker [1990] stream processing primitives in logic programming languages are referred to as *transducers* (see Abelson and Sussman [1985]) and are divided into four groups. However, as pointed out in Parker [1990] this list of transducer types is not exhaustive, although no indication is given as to why this is the case.

(1) **Enumerators (Generators).** Enumerators produce a stream derived from some initial values. A generic enumerator definition is as follows:

```
enumerate(Stream) :-
    initial_state(State),
    enumerate(State,Stream).
```

```
enumerate(S,[X | Xs]) :-
    next_state_and_value(S,NS,X),
    !,
    enumerate(NS,Xs).

enumerate(_,[]).
```

(2) Maps. Maps produce an output stream by applying a function to an input stream. A generic map definition is as follows:

```
map_f([X | Xs],[Y | Ys]) :-
    f(X, Y),
    map_f(Xs, Ys).

map_f([],[]).
```

(3) Filters. Filters produce part of their input stream as output, the elements selected being based on defined criteria. A generic filter definition is as follows:

```
filter([X | Xs], Ys) :-
    inadmissible(X),
    !,
    filter(Xs, Ys).

filter([X | Xs],[X | Ys]) :-
    filter(Xs, Ys).

filter([],[]).
```

(4) Accumulators. Accumulators produce an 'aggregate' of input values as output. A generic enumerator definition is as follows:

```
accumulate(Stream, Value) :-
    initial_state(State),
    accumulate(Stream,State, Value).

accumulate([X | Xs],S, Value) :-
    next_state(X,S,NS),
    accumulate(Xs,NS, Value).

accumulate([],S, Value) :-
    final_state_value(S, Value).
```

Notice that accumulators are strictly first-order primitives.

Examples of Relational Stream Processing Primitives. We now list the examples of second-order stream processing primitives (in functional form) presented informally in Parker [1990] based on the list given in Becker and Chambers [1984].
Let A be any standard S-sorted Σ-algebra.

(A) For each constant $c \in A_s$ for some $s \in S$ we define

$$ConStr^c :\rightarrow [T \rightarrow A]_s$$

by

$$(\forall t \in T) \quad ConStr^c(t) = c.$$

(B) For each constant $i \in \mathbb{Z}$ we define

$$IntFrom^i :\rightarrow [T \rightarrow \mathbb{Z}]$$

by

$$(\forall t \in T) \quad IntFrom^i(t) = i + (t-1).$$

(C) For each binary operator $\sigma : A_s \times A_s \rightarrow A_s$ for some $s \in S$ we define

$$Agg^\sigma : [T \rightarrow A]_s \rightarrow [T \rightarrow A]_s$$

by

$$(\forall a \in [T \rightarrow A]_s)\,(\forall t \in T) \quad Agg^\sigma(a)(t) = \begin{cases} a(0) & \text{if } t = 0, \text{ and} \\ \sigma(Agg^\sigma(a)(t-1), a(t)) & \text{otherwise.} \end{cases}$$

(D) For each unary operator $\sigma : A_s \rightarrow A_s$ for some $s \in S$ we define

$$Map^\sigma : [T \rightarrow A]_s \rightarrow [T \rightarrow A]_s$$

by

$$(\forall a \in [T \rightarrow A]_s)\,(\forall t \in T) \quad Map^\sigma(a)(t) = \sigma(a(t)).$$

(E) For each binary relation $\rho \subseteq A_s \times A_s$ for some $s \in S$ we define

$$Com^\rho : [T \rightarrow A]_s \times [T \rightarrow A]_s \rightarrow [T \rightarrow \mathbb{B}]$$

by

$$(\forall a_1, a_2 \in [T \rightarrow A]_s)\,(\forall t \in T) \quad Com^\rho(a_1, a_2)(t) = \begin{cases} tt & \text{if } \rho(a_1(t), a_2(t)), \text{ and} \\ ff & \text{otherwise.} \end{cases}$$

(F) For each $n \in \mathbb{N}$ and for each $s \in S$ we define

$$Rep_s^n : [T \rightarrow A]_s \rightarrow [T \rightarrow A]_s$$

by

$$(\forall a \in [T \rightarrow A]_s)\,(\forall t \in T) \quad Rep_s^n(a)(t) = a(t \; div \; n).$$

(G) For each $s \in S$ and for each $n, x \in A_s$ we define

$$Lag_s^{n,x} : [T \to A]_s \to [T \to A]_s$$

by

$$(\forall a \in [T \to \mathbb{N}])(\forall t \in T) \quad Lag_s^{n,x}(a)(t) = \begin{cases} x & \text{if } t < n, \text{ and} \\ a(t-n) & \text{otherwise.} \end{cases}$$

(H) For each $s \in S$ we define

$$Merge_s : [T \to A]_s \times [T \to A]_s \to [T \to \mathbb{B}]$$

by

$$(\forall a_1, a_2 \in [T \to A]_s)(\forall t \in T) \quad Merge_s(a_1, a_2)(t) \begin{cases} a_1(t) & \text{if } t \text{ is even, and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

3.8 Stream Processing Languages

As promised we now examine some examples of stream processing languages designed to represent the particular classes of SPSs we have identified in the literature.

In Section 3.8.4 and Section 3.8.5 we discuss the languages LUCID and LUSTRE designed to programme asynchronous and synchronous dataflow SPSs respectively. Also, in Section 3.8.6 we briefly discuss the so-called Manchester Languages and mention some other dataflow languages that can be found in the literature.

In Section 3.8.7 and Section 3.8.8 we discuss the related languages SIGNAL and ESTEREL that are used for programming signal processing networks and reactive systems respectively.

In Section 3.8.9 and Section 3.8.10 we discuss the functional languages AL and PL.

In Section 3.8.11 we examine a modification of PROLOG designed for stream programming.

Finally, in Section 3.8.12 we look at the language STREAM used in the design and verification of hardware.

However, we begin this section with a discussion of the RS-Flip-Flop, that we will use as a running example for presenting and hence comparing the syntax of the existing stream processing languages that we discuss.

3.8.1 A Running Example: the RS-Flip-Flop

The RS-Flip-Flop (that sometimes for convenience we will simply call the Flip-Flop) is a widely occurring device found in computer hardware. The Flip-Flop is designed to output a stream of 'true' (*tt*) and 'false' (*ff*) signals (high and low signals) controlled by two input streams of true and false *control signals*.

Valid control signals consist of one of three simultaneous input pairs:

- 'Reset' – (*tt*,*ff*). This indicates that the Flip-Flop's next output should be a *ff*.

- 'Set' – (*ff*,*tt*). This indicates that the Flip-Flop's next output should be a *tt*.

- 'Hold' – (*ff*,*ff*). This indicates that the Flip-Flop should repeat its previous output.

However, while the pair (*tt*,*tt*) is considered to be illegal input, in general a practical implementation of the Flip-Flop should be able to cope with this input.

3.8.2 Formalization of the Flip-Flop as a ST

This informal description of the Flip-Flop's operation can be made more precise by defining the Flip-Flop as an abstract ST as follows:

$$\text{Flip-Flop} : [T \to \mathbb{B}]^2 \to [T \to \mathbb{B}]$$

defined by

$$(\forall b_1, b_2 \in [T \to \mathbb{B}]) \quad \text{Flip-Flop}(b_1, b_2)(0) = tt$$

and

$$(\forall b_1, b_2 \in [T \to \mathbb{B}])\ (\forall t \in T)$$

$$\text{Flip-Flop}(b_1, b_2)(t+1) = \begin{cases} ff & \text{if } b_1(t) = tt \text{ and } b_2(t) = ff; \\ tt & \text{if } b_1(t) = ff \text{ and } b_2(t) = tt; \text{ and} \\ \text{Flip-Flop}(b_1, b_2)(t) & \text{otherwise.} \end{cases}$$

In particular, notice that this specification outputs its previous output if the illegal control signal (*tt*,*tt*) is supplied as input.

3.8.3 An Implementation of the Flip-Flop as a SPS

A typical implementation of the Flip-Flop can be visualized at the conceptual level as a SDU-SPS comprising two input streams, two modules, and two output streams wherein both modules compute the 'nor' function; that is,

$$\text{nor} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$

defined by

$$(\forall b_1, b_2 \in \mathbb{B}) \quad \text{nor}(b_1, b_2) = \text{not}(\text{or}(b_1, b_2)).$$

To reconcile this implementation with the functionality of the specification only one stream is considered as 'proper output' (the first module's output), with the other stream used only as 'feedback' to compute the Flip-Flop's next output.

The Flip-Flop SPS's Computation. The SPS representing the Flip-Flop is shown in Figure 3.1. Initially the modules of the SPS representing the Flip-Flop will output some initial values (see Sections 3.10 and 8.3) that for convenience we will assume is the pair (*tt*,*ff*).

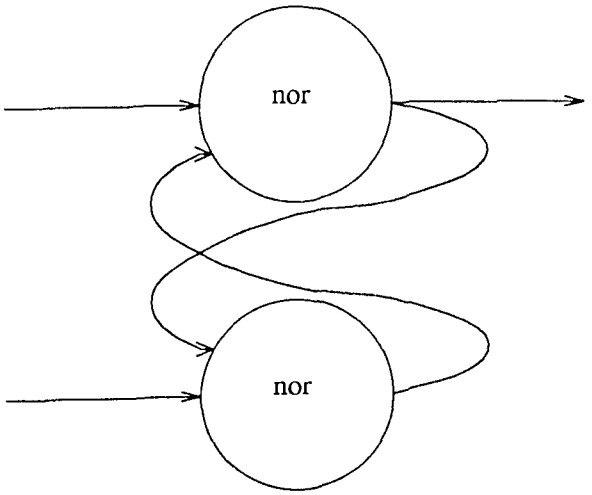

Figure 3.1: The RS-Flip-Flop as a SPS

After the Flip-Flop's initial output each module computes (synchronously) on the streams of control signals and the previous output of the other module to produce the next output.

Properties of the Flip-Flop. Despite the relative simplicity of the Flip-Flop, the device has many subtleties and has been the subject of extensive study (see for example Thompson and Tucker [1991]). Indeed, in order for the Flip-Flop SPS's implementation to meet the requirements of the specification the Flip-Flop requires pre- and post-processing of its input and output respectively. Indeed, in Chapter 7 we show that the formal verification of the implementation of the Flip-Flop we have presented is non-trivial. However, the device is simple to specify at the conceptual level and relies on mutual recursion and therefore provides a useful small example to illustrate the syntax of the stream processing languages that we discuss in the following sections.

3.8.4 LUCID

LUCID (Wadge and Ashcroft [1985]) is perhaps the best known of all the dataflow languages that have been developed. A LUCID programme is essentially a system of recursion equations, although LUCID is described by its authors as a 'functional dataflow programming language'. The term 'dataflow' is chosen because each LUCID programme is semantically equivalent to a dataflow network; and 'functional' because the output of each filter is a function of its inputs. (Note that the term 'functional' used here does not imply a computation without side-effects as in the mathematical sense.) LUCID is also described by its authors as a 'typeless' language as there is no declaration section. However, a more formal description would be to say that LUCID operators are overloaded and their type is inferred from their context.

LUCID was conceived by its authors in 1974 with what they claim to be quite modest aims; that is, to show that real-life programmes could be written in a purely declarative style so that

programme verification would be possible. The authors felt that a purely functional language was not creditable for this purpose for reason of efficiency, and so LUCID contains iterative constructs so that (the authors claim) when writing LUCID programmes the programmer may make use of algorithms used in real 'everyday' programming. It was also (later?) intended that LUCID could exploit the new highly-parallel, multiprocessor dataflow machines.

Constructs and Primitives. Each LUCID programme is an expression structured using the 'where' clause taken from *ISWIM* (see Landin [1966]) over simple 'data types', for example: integers; reals; Booleans; words; character strings; and finite lists. LUCID also uses the *if... then... else* construct.

LUCID has the 'usual' operators over the data types just mentioned and treats them as point-wise extensions over time and hence can be used to manipulate streams directly. In addition LUCID uses six explicit stream processing primitives with the following semantics.

Let A be any standard S-sorted Σ-algebra.

(1) First. For each $\underline{u} \in \underline{S}^+$ we define

$$First_{\underline{u}}^{A} : [T \to A]^u \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall t \in T) \quad First_{\underline{u}}^{A}(a)(t) = a(0).$$

(2) Next. For each $\underline{u} \in \underline{S}^+$ we define

$$Next_{\underline{u}}^{A} : [T \to A]^u \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall t \in T) \quad Next_{\underline{u}}^{A}(a)(t) = a(t+1).$$

(3) Followed By. For each $\underline{u} \in \underline{S}^+$ we define

$$Fby_{\underline{u}}^{A} : [T \to A]^u \times [T \to A]^u \to [T \to A]^u$$

by

$$(\forall a_1, a_2 \in [T \to A]^u)\,(\forall t \in T) \quad Fby_{\underline{u}}^{A}(a_1, a_2)(t) = \begin{cases} a_1(t) & \text{if } t = 0, \text{ and} \\ a_2(t-1) & \text{otherwise.} \end{cases}$$

(4) At Time. For each $\underline{u} \in \underline{S}^+$ we define

$$AtTime^{A}_{\underline{u}} : [T \to A]^u \times [T \to \mathrm{N}] \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall n \in [T \to \mathrm{N}])\,(\forall t \in T) \quad AtTime^{A}_{\underline{u}}(a, n)(t) = a(n(t)).$$

(5) Whenever. For each $\underline{u} \in \underline{S}^+$ we define

$$Whenever^{A}_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Whenever^{A}_{\underline{u}}(a, b)(t) = \begin{cases} a(t) & \text{if } b(t) = tt, \text{ and} \\ Whenever^{A}_{\underline{u}}(a, b)(t+1) & \text{otherwise.} \end{cases}$$

(6) As Soon As. For each $\underline{u} \in \underline{S}^+$ we define

$$Asa^{A}_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \rightsquigarrow [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T) \quad Asa^{A}_{\underline{u}}(a, b)(t) = a(\mu\, k.[b(k) = tt]).$$

(7) Upon. For each $\underline{u} \in \underline{S}^+$ we define

$$Upon^{A}_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Upon_{\underline{u}}(a, b)(t) = \begin{cases} a(0) & \text{if } t = 0, \\ a(\mathrm{NumOfTrues}(Next^{A}_{\underline{b}}(b))(t)) & \text{otherwise} \end{cases}$$

wherein $Next^{A}_{\underline{b}}$ is defined as above and $\mathrm{NumOfTrues} : [T \to \mathbb{B}] \to [T \to \mathrm{N}]$ is defined by

$$(\forall b \in [T \to \mathbb{B}])$$

$$\mathrm{NumOfTrues}(b)(0) = \begin{cases} 1 & \text{if } b(0) = tt, \text{ and} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\text{NumOfTrues}(b)(t+1) = \begin{cases} 1 + \text{NumOfTrues}(b)(t) & \text{if } b(t+1) = tt, \text{ and} \\ \text{NumOfTrues}(b)(t) & \text{otherwise.} \end{cases}$$

The use of Streams. As with many of the other languages we will discuss streams are represented as variables. In the particular case of LUCID any free variables (not explicitly declared) are treated as input streams.

Language Development and Current Uses. Since its conception various implementations of LUCID have been written (see Farah [1977] and Sargeant [1982]) and one such implementation *pLUCID* – LUCID over the algebra of POP-2 taken from Burstall *et al.* [1971] – has been used experimentally for software design (see Wadge [1984]).

Lucid Syntax. The RS-Flip-Flop can be described in LUCID as follows:

flipflop(In1, In2) = (Out1, Out2)
where
Out1 = true ***fby*** *(In1 nor Out2)*
Out2 = false ***fby*** *(In2 nor Out1)*

3.8.5 LUSTRE

LUSTRE (see Caspi *et al.* [1987]) is a synchronous dataflow language related to LUCID. Like LUCID it is based on the description of a SPS as a system of equations. However, unlike LUCID, LUSTRE requires that the output at time t of the functions defined by such a set of equations depends only on input received either before or at time t. This property is referred to by the authors of LUSTRE as *causality*.

We note in passing that intuitively *causality* appears to restrict LUSTRE to expressing the class of *course-of-values recursive functions* (see Tucker and Zucker [1988]). However, the authors do not discuss the issue of computability in this respect.

In common with languages for describing reactive systems LUSTRE is based on the *strong synchrony hypothesis* and has a *multiform* notion of time (see Section 3.5). Furthermore, in common with the language ESTEREL (see Section 3.8.8) LUSTRE programs are implemented via compilation into finite automata.

The authors state that LUSTRE programs are subject to a strict analysis for *deadlock* based on a domain theoretic analysis of the various clocks defined using the *When* operator, rather than by the *cycle sum test* that is applied to LUCID programmes (see Wadge [1981]). However, the authors concede that while this approach does detect any potential deadlock it also rejects some valid programmes. It is this strict approach to the interplay between the various clocks over which the various filters compute within a programme that ensures the synchronous nature of LUSTRE.

Primitives and Constructs. In common with LUCID underlying operations are treated as point-wise extensions over time in LUSTRE and can be directly applied to streams.

Any LUSTRE program, that is correct with respect to the various static-semantic tests that are applied to it, is compiled into a simplified basic abstract syntax. Compilation into this restricted syntax eliminates separate node (filter) definitions, used to employ a modular programming technique. In particular, stream operators are compiled into a restricted subset of stream operators that form a functionally complete set. This functionally complete set consists of the following four operations that we now define informally.

Let A be any standard S-sorted Σ-algebra wherein $S = \{s_1, \ldots, s_n\}$ for some $n \in \mathbb{N}$. Also, let $\mathbb{U} =< \mathbb{U}_{s_1}, \ldots, \mathbb{U}_{s_n} >$ be some collection of distinct values such that $\mathbb{U}_{s_i} \notin A_{s_i}$ for $i = 1, \ldots, n$, and let $A^{\mathbb{U}} = A \cup \mathbb{U}$.

(1) Previous. For each $\underline{u} \in \underline{S}$ we define

$$Pre_{\underline{u}}^{A} : [T \to A]^u \to [T \to A^{\mathbb{U}}]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall t \in T) \quad Pre_{\underline{u}}^{A}(a)(t) = \begin{cases} (a)(t-1) & \text{if } t > 0, \text{ and} \\ (\mathbb{U}_{u_1}, \ldots, \mathbb{U}_{u_{|u|}}) & \text{otherwise.} \end{cases}$$

(2) Followed By. For each $\underline{u} \in \underline{S}$ we define

$$FBy_{\underline{u}}^{A} : [T \to A]^u \times [T \to A]^u \to [T \to A]^u$$

by

$$(\forall a_1, a_2 \in [T \to A]^u)\,(\forall t \in T) \quad FBy_{\underline{u}}^{A}(a_1, a_2)(t) =$$

$$\begin{cases} a_1(0) & \text{if } t = 0, \text{ and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

Notice that this is different from the LUCID operator *Fby*.

(3) When. For each $\underline{u} \in \underline{S}$ we define

$$When_{\underline{u}}^{A} : [T \to A]^u \times [T \to \mathbb{B}] \rightsquigarrow [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T) \quad When_{\underline{u}}^{A}(a, b)(t) = a(\mu\, k.[b(k) = tt \wedge k \geq t]).$$

(4) **Current**. For each $\underline{u} \in \underline{S}$ we define

$$Current^{A}_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Current^{A}_{\underline{u}}(a,b)(t) = \begin{cases} a(t) & \text{if } b(t) = tt \\ Current^{A}_{\underline{u}}(a,b)(t-1) & \text{if } b(t) = ff \wedge t > 0 \\ When^{A}_{\underline{u}}(a,b)(0) & \text{otherwise.} \end{cases}$$

The use of Streams. As is common in equational stream processing languages undefined variables are treated as input streams in LUSTRE programmes. Indeed the notion of a stream in LUSTRE is the same as in standard dataflow and is not the same as in the reactive system paradigm. It is for this reason that we choose to classify LUSTRE as a dataflow language.

Semantics. Two separate approaches to the semantics of LUSTRE have been applied. The first is a domain-theoretic approach in the style of Khan's work. The second approach is an operational semantics based on the work of Plotkin (see Plotkin [1981]). This operational semantics can been used for proofs of equivalence of different LUSTRE programs, and is the semantic models that has been used to analyse the properties of the compilation of LUSTRE programmes into finite automata.

Language Development and Current Uses. LUSTRE has been used for such diverse applications as music synthesis description (see Amblard and Charles [1989]) and for verification of real-time systems (see Halbwachs *et al.* [1992]) and appears to have superseded its parent language LUCID as the the language of choice for dataflow systems.

Syntax. The RS-Flip-Flop can be expressed in LUSTRE as follows:

```
node flipflop(In1, In2 : bool)
    returns(Out1, Out2 : bool);
let
    Out1 = tt : FBy pre(In1) nor pre(Out2);
    Out2 = ff : FBy pre(In2) nor pre(Out1);
tel
```

3.8.6 Other Dataflow Languages

The 'Manchester Languages'. There are several so-called 'Manchester Languages' (see Herath *et al.* [1986]) including SASL, SISAL, LAPSE and MAD that have been used on the Manchester Dataflow Machine. In this Section we very briefly discuss these languages. The reader interested in the topic of specialized dataflow architecture can consult Gurd *et al.* [1981]

and more recently Sharp [1991].

SASL. The language SASL (see Herath *et al.* [1986]) is a functional language. SASL derives its name from the fact that only single assignment functions (one argument) are permitted. Multiple argument functions are achieved with *Currying*.

SISAL. The language SISAL (see McGraw *et al.* [1985]) is a typed 'value orientated' functional language designed for dataflow computing machines. The name SISAL is derived from Streams and Iteration in a Single Assignment Language. SISAL allows recursive constructs and looping. In addition to being implemented on the Manchester Machine, SISAL has also been implemented on the VAX, CRAY and HP dataflow machines (see Sharp [1991]).

VALID. The language VALID (see Amamiya *et al.* [1984]) is a higher-order functional language designed to achieve very high-level parallelism. VALID derives its name from Value Identification Language and has a mix of ALGOL- and LISP-like syntax, including block-structuring and case statements.

DCBL. The language DCBL (pronounced 'decibel' – see Herath *et al.* [1986]) is a high-level dataflow language designed to define the operational semantics for dataflow computing languages. In particular, DCBL is designed to enable users to express programmes with many forms of concurrency, at a high-level of abstraction without any machine dependent characteristics.

General Dataflow Languages.

VAL. The language VAL (see Dennis [1974] and Brock [1987]) is a synchronous functional language with implicit concurrency. The name VAL is derived from the languages 'value orientated' rather than 'variable orientation' nature; that is, new values can be derived, but cannot be modified. This principle is used in the language so that values can be assigned to identifiers, but identifiers cannot be used as variables in order to address certain issues arising from the automatic generation of concurrent implementations.

ID. The language ID (see Arvind and Gostelow [1978]) is an un-typed, functional, block-structured language that supports non-determinism and the use of streams. A programme in ID consists of a list of expressions wherein each expression is either a 'loop', a 'conditional', a 'block' or a 'procedure application'.

3.8.7 SIGNAL

SIGNAL (see Guatier *et al.* [1987]) is an applicative language designed to programme real-time systems using synchronous dataflow. The authors claim that a SIGNAL representation is very close to the specification of a system, either mathematical or graphical, and leads to an elegant formal 'synchronization calculus'.

SIGNAL uses two concepts of time: *logical time* and an associated timing calculus based on the strong synchrony hypothesis (see Section 3.5); and *physical time*. Using this system temporal references are determined entirely by the sequence of communication events and not (as the authors claim) by the input events as in either ESTEREL or the dataflow approach.

Individual processing elements in a SPSs described by SIGNAL are *not* synchronized by a single global clock $T = \{0, 1, 2, \ldots\}$, rather SIGNAL has a 'multiform' notion of time (see Section 3.5).

The use of Streams. The name SIGNAL is derived from the infinite sequences called *signals* over which all processes in a SIGNAL system compute (see Section 3.5.1). Each signal is a map $a : T \rightarrow A$ for some data set A and some *clock* $T = \{1, 2, \ldots\}$. (Notice that the clock starts at 1 and not 0.) It would appear from this description that signals are streams. However, the individual values of a signal may be 'sampled' at continuous points rather than simply at the discrete division indicated by the signal's clock. In addition, the values are not persistent and as such may only be sampled in order; that is, once the value of a signal a has been sampled at time $t \in T$ it may henceforth only be sampled at some time t' wherein $t' > t$. (Also see the following section on further operators.) Notice that this interpretation of a signal is related to Kahn's visualization of streams as asynchronous FIFO queues (see Section 2.5).

Constructs and Primitives. SIGNAL operators are divided into two classes: 'S-operators' that define signals and 'P-operators' that are used to create interconnections between processes. We will only consider signal definition operators here.

(1) Basic Operations. The syntax

$$a := b + 1$$

for some signals $a \in [T \rightarrow A]$ and $b \in [T' \rightarrow A]$ for some data set A wherein 1 is a constant signal creates a process with the following semantics:

$$(\forall t \in T) \quad a(t) = b(t) + 1;$$

that is, it creates a process that takes a single signal input b and produces a single signal output a that at every time cycle t is precisely the value of $a(t)$ plus one.

Notice here that because of the nature of the process specified the two clocks T and T' are synchronized and hence considered to be the same. This is not a property of signal processes in general.

(2) Delays. The syntax

$$a \; init \; c$$
$$a := b \, \$1$$

for some signals $a \in [T \rightarrow A]$ and $b \in [T' \rightarrow A]$ for some data set A wherein c is a constant signal and creates a process with the following semantics:

$$(\forall t \in T) \quad a(t) = \begin{cases} c & \text{if } t = 0 \text{ and} \\ b(t-1) & \text{otherwise;} \end{cases}$$

that is, the statement creates a process with a single input that delays its output by one time cycle and outputs a constant at time $t = 0$.

Notice here that a delay is defined by two separate processes (statements) and hence if the first statement is omitted (as in some of the reference examples) then the signal described by its process is undefined at time $t = 0$. Also, there is an inconsistency in the reference examined in that the signal's underlying clocks are given as $T = \{1, 2, \ldots\}$, but the *init* statement define values of streams at time $t = 0$.

(3) Composition. The syntax

$$(|a \; init \; c|a := b + 1|b := a \; \$1|)$$

denotes the process formed by the composition of the processes $a \; init \; c$, $a := b + 1$ and $b := a \; \$1$ specified in the previous examples. The ordering of the sub-processes within a composition is unimportant; that is, it is associative and commutative, and communication is implied between processes wherein an output signal of one process (an identifier on the left of an ':=') has the same name as an input signal (an identifier on the right of an ':=') from a *different* process. So our example has the intended semantics

$$(\forall t \in T) \quad a(t) = \begin{cases} c & \text{if } t = 0 \text{ and} \\ a(t-1) + 1 & \text{otherwise;} \end{cases}$$

(4) Further operators. SIGNAL also uses the operators *when*, *event* and *synchro* with the following syntax

$$a := b \; when \; c,$$

$$a := event \; b$$

and

$$synchro \; a, b$$

respectively. Because the semantics of these statements is 'formalized' using a *clock calculus*, that we will not discuss, we will only give the intuitive meanings of these statements: *when* is a so-called *undersampling* operator that, in the context of our example, produces the input signal b if it is defined at the same time the Boolean signal c is defined and 'true'; *event* delivers an always 'true' Boolean signal whenever (in the context of our example) signal b is defined; and *synchro* (again in the context of our example) explicitly synchronizes the signal's a and bs clocks.

Because of the lack of a global clock and the definition of a signal, when examining the current value of a particular signal $a : T \to A$ we have two possible result: it may either be undefined or will have some value in the data set A. Because of this definition of signals the authors use a *clock calculus* to give and check the semantics of SIGNAL definitions. As Boolean signals are used to define clocks (via the *event* operator) this

clock calculus requires two data sets (and is the reason, the authors claim, that a Boolean calculus is insufficient); that is, $C = \{-1, 0, 1\}$ for Boolean signals, wherein 0 denotes the absence of a value, -1 denotes 'false', and 1 denotes 'true'; and $C' = \{0, 1\}$ for all other signals, wherein 0 denotes the absence of a value and 1 the presence of a value. Within this calculus the data set C is given the structure of a commutative field onto which all SIGNAL processes can be mapped. This 'mapping' of a process is used to analyse the relationship of any sub-modules clocks and to detect incorrectly defined processes. For example, the compositional process

$$(|x := a \; when \; (a > 0)|y := a \; when \; (not(a > 0))|z := x + y|)$$

gives rise to the following equations in the clock calculus (using c to represent the Boolean expression $a > 0$)

$$\begin{aligned} x^2 &= a^2(-c - c^2) \\ y^2 &= a^2(c - c^2) \\ z^2 &= x^2 = y^2 \end{aligned}$$

that gives $-c = c$. As this has a single solution ($c = 0$) the process defined by this composition is undefined. This is intuitively clear from the example as the clocks over which x and y are defined are mutually exclusive.

Semantics. SIGNALS semantics is based on the *clock calculus* described above that we will not discuss further.

Syntax. The RS-Flip-Flop can be expressed as follows in SIGNAL:

```
(| Out1 init tt | Out2 init ff |
   In1' := In1$1 | In2' := In2$1 |
   Out1' := Out1$1 | Out2' := Out2$1 |
   Out1 := In1' nor Out2' | Out2 := In2' nor Out1'
|)
```

3.8.8 ESTEREL

ESTEREL (see Berry and Cosserat [1984], Berry *et al.* [1988], Berry and Gonthier [1988] and Boussinot and de Simone [1991]) is a real-time imperative concurrent language for describing reactive systems. However, ESTEREL is designed for describing ADB-SPSs rather than ADU-SPSs as in the case of the languages LUSTRE and SIGNAL. (See the following section on the use of streams in ESTEREL.)

The authors state that the aim of ESTEREL is to develop a rigorous formal model of real-time computation with an operational semantics that can be used for tasks where programming using conventional languages is difficult.

Constructs and Primitives. The basic structuring device in an ESTEREL programme is the *module* with input and output signals for broadcast communication and internal signals for internal broadcast communication.

The body of a module that describes its operation can include the following basic primitives and constructs:

(1) Null process. The command

nothing

creates a process that does nothing in zero time.

(2) Local variable declaration. The command

var X : *type* in *i* end

creates a local variable X for process *i*.

(3) Variable assignment. The command

X := *exp*

assigns variable X with the value of the expression *exp*.

(4) Signal Transmission. The command

emit s(*exp*)

emits the value of *exp* on signal s.

(5) Conditional execution. The command

do *i* upto s(*exp*)

repeatedly execute process *i* until the value *exp* is broadcast onto signal s and

do *i* uptonext s(*exp*)

repeatedly execute process *i* until the value *exp* is broadcast onto signal 's' twice.

(6) Sequential Composition. The command

$$i_1; i_2$$

invokes process i_2 immediately upon completion of process i_1.

(7) Parallel Composition. The command

$$i_1 || i_2$$

simultaneously invokes processes i_1 and i_2 sharing the same local variables and local signals.

(8) Iteration. The command

loop i end

executes process i in a continuous loop. However, processes like

X := 0; loop X := X + 1 end; loop emit s(X) end

have no semantics, due to the strong synchrony hypothesis, and are checked for during static semantic evaluation.

(10) If Then Else. The command

if *boolexp* then i_1 else i_2 fi

has the usual semantics, but because of the strong synchrony hypothesis, we assume here that *boolexp* is evaluated in zero time so control is passed immediately to either i_1 or i_2.

(11) Process termination. The command

tag T in i end
exit T

executes process i until 'exit T' is executed (in i) whereupon process i is terminated.

From these basic primitives many 'higher-level' construct are formed. However, these are just for convenience during programming and to not occur in the semantic model.

The use of Streams. ESTEREL uses the same notion of stream processing as SIGNAL; that is, signals and a multiform notion of time. However, unlike SIGNAL in ESTEREL some

signals are used for both input and output from processes and information is broadcast in the sense that complete connectivity is assumed between processes. A commutative operator '*' is explicitly associated with each signal to deal with simultaneous transmission (see Milner [1983]) such that if the values $v_1, v_2, \ldots, v_n$ for $n > 1$ are broadcast simultaneously onto a signal s then the value on s is $v_1 * v_2 * \cdots * v_n$.

Semantics. ESTEREL has a complicated semantic model with three different levels:

(1) Static Semantics. Used to establish temporal relations between processes and check for any temporal paradoxes.

(2) Behavioural Semantics. Used to define the temporal behaviour with respect to the static semantics.

(3) Computational Semantics. Used to establish exactly what a program computes.

Once the computational semantics has been established any concurrency is eliminated by compiling into a sequential programme that is implemented as an automaton in *C* (for example) by a similar method used in parser generators (see for example Sun [1988]). The authors are confident that this technique leads to an efficient implementation.

Language Development and Current Uses. ESTEREL has been used for HCI and for programming communication protocols and real-time controllers (see Clement and Incerpi [1989], Murakami and Sethi [1990] and Berry and Gonthier [1991] respectively). An ESTEREL environment exists (see Boudol *et al.* [1990]) that includes simulators, debugging tools and a compiler to hardware, based on the techniques discussed in Berry [1991]. One current research aim is to implement existing ESTEREL programmes directly in hardware.

Syntax. The RS-Flip-Flop can be described in ESTEREL as follows:

```
var L1,L2 : bool in flipflop ;
module flipflop:
    input In1, In2 : bool ;
    output Out1, Out2 : bool ;
    L1 := true ;
    L2 := false ;
    emit Out1(L1) ;
    emit Out2(L2) ;
    loop
        L1 := In1 nor L2 ;
        L2 := In2 nor L1 ;
        emit Out1(L1) ;
        emit Out2(L2) ;
    end.
```

3.8.9 AL

AL is a typed equational language that provides a specification formalism for (potentially) recursive stream operations. Implicit concurrency is expressed by the juxtaposition of equational definitions within both programme and agent definitions.

Constructs and Primitives. AL uses a block structure and includes constructs such as *if...then...else...fi.* It also includes the finite choice operator □, and hence is AL is able to define non-deterministic behaviour.

AL has all of the basic stream processing primitives as described in Section 3.7.2 as built in operators. In addition, *functions* mapping data to data and *components* mapping data and streams of data to streams of data can be defined by the user.

The use of Streams. The declaration of input and output streams is explicit in AL and streams may occur at most once on the left-hand-side of an equation. In particular output streams must occur exactly once as a left-hand-side and input streams may not occur as a left-hand-side.

Semantics. AL is restricted to second-order definitions and has a fixed-point semantics in the style of Kahn.

Language Development and Current Uses. For an introduction to the use of AL see Section 3.4.1 on the FOCUS project.

A prototype of AL has been implemented on a SUN workstation (see Nueckel [1988]) and experiments to implement AL on an INTEL hyper-cube are in progress (see Gorlatch [1992]).

Syntax. The RS-Flip-Flop can be represented in AL as follows:

```
programme flipflop ≡ chan bool In1, In2 → chan bool Out1, Out2:
    funct nor ≡ bool b1, b2 → bool:
        not(b1 or b2),
    agent streamnor ≡ chan bool sb1, sb2 → chan bool sbout:
        sbout ≡ nor(ft.sb1, ft.sb2)
    end,
    agent leftbs ≡ chan bool lbs1, rbs1 → chan bool lbs
        lbs ≡ lbs1
    end,
    agent rightbs ≡ chan bool lbs1, rbs1 → chan bool rbs
        rbs ≡ rbs1
    end,
    Out1 ≡ true & streamnor(In1, rightbs.flipflop(In1, In2)) & leftbs.rt.flipflop(rt.In1, rt.In2)
    Out2 ≡ false & streamnor(In2, leftbs.flipflop(In1, In2)) & rightbs.rt.flipflop(rt.In1, rt.In2)
end flipflop.
```

3.8.10 PL

PL is a imperative, parallel procedurally language designed for stream programming.

Constructs and Primitives. In some sense PL can be considered to be a classical language containing assignment statements and while loops. However, in addition PL also has the non-terminating loop construct *loop...pool.* PL is syntactically very similar to AL and allows the definition of *functions* and *components* (see Section 3.8.9) and also has all the stream processing functions described in Section 3.7.2 as basic operations.

The use of Streams. As with many stream programming languages variables are used to represent input, but in addition as with AL variables are also used to explicitly represent output. In contrast to AL there are two explicit operators in PL for 'reading' and 'writing' values to and from streams (channels) denoted '?' and '!' respectively that can be defined informally as follows.

If c is a channel identifier and x is a variable of appropriate type then the command

$$c?x$$

is interpreted informally as 'remove the first value from channel c and assign this value to variable x' If c is empty then execution of this command is delayed (possible infinitely). Similarly if c is again a channel identifier and E is an expression of appropriate type then the command

$$c!E$$

is interpreted informally as 'evaluate E and then write this value to channel c.' Again if E cannot be evaluated, as it may depend on some input evaluation, then this command may also be delayed (possible infinitely).

The use of these two operations provides a model of asynchronous communication and it is pointed out in Broy *et al.* [1993] that they should not be confused with the operators '?' and '!' in *CSP* (see Hoare [1985]) that provide synchronous communication.

In PL equations are further restricted in that channel identifiers may only occur once (at most) in the right hand side. Also, new channels may be introduced dynamically within PL via recursion and hence dynamic networks may be modelled. For this reason the use of the word channel is less related to the concept of a stream in PL than it is in AL.

Semantics. PL is based on an operational state transformer semantics derived from work in Broy and Lengauer [1991] and Dederichs [1992]. It is intended that this semantics can be related to an equivalent abstract (denotational) semantics as a ST and hence PL can be related formally to an AL specification.

Language Development and Current Use. For an introduction to the use of PL see Section 3.4.1 on the FOCUS project.

Syntax. The RS-Flip-Flop can be represented in PL as follows:

```
programme flipflop ≡ chan bool In1, In2 → chan bool Out1, Out2:
    var bool i1, i2, l1, l2;
    var bool o1 := true, o2 := false;
    var nat time := 0;
    loop
        if time > 0 then
            In1?i1;
            In2?i2;
            o1 := i1 nor l2;
            o2 := i2 nor l1;
        fi
        Out1!o1;
        Out2!o2;
        l1 := o1;
        l2 := o2;
        time := time + 1;
    pool
end flip-flop.
```

3.8.11 PROLOG with streams

Bellia *et al.* [1982] and Bellia *et al.* [1984] describe a modification of PROLOG (see Kowalski [1974]) (that for convenience we will denote PROLOG) to provide an applicative language for the specification of a class of ADU-SPSs.

Constructs and Primitives. In PROLOG a network of agents is specified by a set of *Horn clauses* wherein each clause corresponds to a particular agent. The structure of the language is essentially that of PROLOG and the stream processing primitives available are those used in the functional approach (see Section 3.7.2).

The use of Streams. The approach to streams in PROLOG is the same as that in functional languages. In particular, in PROLOG uni-directional channels are modelled by shared syntactically distinguished input and output variables within each atomic clause and hence the expressive power of PROLOG is limited compared to conventional PROLOG, as invertibility is limited. However, the authors claim that this is not a problem in practice.

Semantics. PROLOG is formalized using a standard fixed-point semantics (see van Emden and Kowalski [1976]) and makes a explicit distinction between *data constructors* and functions (see Levi and Pegna [1983]) to modify the semantic model to deal with infinite terms.

Language Development and Current Uses. It is intended that PROLOG is viewed as a proper extension of a term re-writing system, wherein each Horn clause is interpreted as an extended re-write rule. It is also the authors' intention that completion algorithms such as

Knuth-Bendix (see Knuth and Bendix [1970]) can be generalized to generate confluent systems from PROLOG network descriptions. However, we are not aware of any subsequent work by the authors in this field.

Syntax. The RS-Flip-Flop can be represented in PROLOG as follows:

type BOOL is tt, ff ;
type STREAM-OF-BOOL is nil, cons(BOOL, STREAM-OF-BOOL) ;

fflop1 : STREAM-OF-BOOL × STREAM-OF-BOOL → STREAM-OF-BOOL ;
fflop2 : STREAM-OF-BOOL × STREAM-OF-BOOL → STREAM-OF-BOOL ;
Nor : BOOL × STREAM-OF-BOOL → BOOL ;
not : BOOL → BOOL ;

fflop1(cons(b1,sb1),cons(b2,sb2)) = cons(cons(tt,o1),o2) →
o1 = Nor(b1,fflop2(cons(b1,sb1),cons(b2,sb2))) ;
o2 = fflop1(sb1,sb2) ;

fflop2(cons(b1,sb1),cons(b2,sb2)) = cons(cons(ff,o3),o4) →
o3 = Nor(b2,fflop1(cons(b1,sb1),cons(b2,sb2))) ;
o4 = fflop2(sb1,sb2) ;

Nor(ff,cons(b,sb)) = not(b) → ;
Nor(tt,cons(b,sb)) = ff → ;

not(tt) = ff → ;
not(ff) = tt → ;

3.8.12 STREAM

STREAM (see Kloos [1987a] and Kloos [1987b]) is a concurrent scheme language designed for formally specifying, reasoning about and transforming hardware designs at the conceptual, register and gate level. Furthermore, STREAM is intended to address description features associated with each level in a single formalism. The approach is rather like a single programming language that includes formal, high-level and machine-code descriptions as primitives, and is referred to as *almost hierarchical* approach (see Sussman and Steele [1980]).

STREAM is an acronym for STandard REpresentation of Algorithms for Micro-electronics. However, the name STREAM is also intended to reflect the stream processing nature of the language. Indeed, in addition to its role as hardware description language STREAM can also be directly interpreted as a dataflow language, resembling the language of Dennis [1974]. However, the formal equivalence of STREAM and Dennis's language is not addressed.

Constructs and Primitives. STREAM uses the following stream processing primitives

that are referred to as *agents*.

Let A be any standard S-sorted Σ-algebra.

(1) Append. For each $s \in S$ we define the *append agent*

$$\&_s : A_s \times [T \to A]_s \to [T \to A]_s$$

(ambiguously denoted $\&$) by

$$(\forall a \in A_s)\,(\forall a' \in [T \to A]_s)\,(\forall t \in T) \quad (a \,\&\, a')(t) = \begin{cases} a & \text{if } t = 0, \text{ and} \\ a'(t-1) & \text{otherwise.} \end{cases}$$

(2) Lifting. For each $\sigma \in \Sigma_{w,s}$ for each $w \in S^+$ and for each $s \in S$ we define the *lifting agent*

$$*_{w,s} : (A^w \to A_s) \to ([T \to A]^w \to [T \to A]_s)$$

(ambiguously denoted $*$) by

$$(\forall a \in A^w)\,(\forall t \in T) \quad \sigma^*(a)(t) = \sigma(a(t)).$$

(3) Distribution. For each $s \in S$ we define the *distribution agent*

$$\mathrm{distr}_s : [T \to \mathbb{B}] \times [T \to A]_s \rightsquigarrow [T \to A]_s \times [T \to A]_s$$

(ambiguously denoted distr) by

$$(\forall b \in [T \to \mathbb{B}])\,(\forall a \in [T \to A]_s)\,(\forall t \in T) \quad \mathrm{distr}(b,a)(t) = (x_1, x_2)$$

wherein

$$x_1 = a(\mu\, k \geq t.[b(k) = tt])$$

and

$$x_2 = a(\mu\, k' \geq t.[b(k') = ff]).$$

(4) Selection. For each $s \in S$ we define the *selection agent*

$$\mathrm{selec}_s : [T \to \mathbb{B}] \times [T \to A]_s^2 \to [T \to A]_s$$

(ambiguously denoted selec) by

$$(\forall b \in [T \to \mathbb{B}])(\forall a_1, a_2 \in [T \to A]_s)(\forall t \in T) \quad \mathrm{selec}(b, a_1, a_2)(t) = \begin{cases} a_1(t) & \text{if } b(t) = tt, \text{ and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

In addition STREAM also uses the following functional constructs for building SPSs from more primitive SPSs:

(A) Parallel Composition. For each $u, u', v, v' \in S^+$ we define the *parallel composition constructor* ambiguously denoted $\Downarrow$ with functionality

$$\Downarrow : ([T \to A]^u \to [T \to A]^v) \times ([T \to A]^{u'} \to [T \to A]^{v'}) \to ([T \to A]^{u\,u'} \to [T \to A]^{v\,v'})$$

by

$$(\forall \mathbb{S} \in [T \to A]^u \to [T \to A]^v)\,(\forall \mathbb{S}' \in [T \to A]^{u'} \to [T \to A]^{v'})$$
$$(\forall a = (a_1, \ldots, a_{|u\,u'|}) \in [T \to A]^{u\,u'})\,(\forall t \in T)$$
$$(\mathbb{S} \Downarrow \mathbb{S}')(a)(t) = (x_1, \ldots, x_{|v\,v'|})$$

wherein

$$x_i = \begin{cases} (\mathbb{S}(a_1, \ldots, a_{|u|})(t))_i & \text{if } i \leq |u|, \text{ and} \\ (\mathbb{S}'(a_{|u|+1}, \ldots, a_{|u\,u'|})(t))_i & \text{otherwise.} \end{cases}$$

(B) Sequential Composition. For each $u, v, w \in S^+$ we define the *sequential composition constructor* ambiguously denoted $\Rightarrow$ with functionality

$$\Rightarrow : ([T \to A]^u \to [T \to A]^v) \times ([T \to A]^v \to [T \to A]^w) \to ([T \to A]^u \to [T \to A]^v)$$

by

$$(\forall \mathbb{S} \in [[T \to A]^u \to [T \to A]^v])\,(\forall \mathbb{S}' \in [[T \to A]^v \to [T \to A]^w])\,(\forall a \in [T \to A]^u)\,(\forall t \in T)$$
$$(\mathbb{S} \Rightarrow \mathbb{S}')(a)(t) = \mathbb{S}'(\mathbb{S}(a))(t).$$

(C) Feedback. For each $s \in S$ and for each $u, v \in S^+$ we define the *feedback constructor* ambiguously denoted C with functionality

$$\mathrm{C} : ([T \to A]^{s\,u} \to [T \to A]^{s\,v}) \to ([T \to A]^u \to [T \to A]^v)$$

by

$$(\forall \mathbb{S} \in [T \to A]^{s\,u} \to [T \to A]^{s\,v})\,(\forall a \in [T \to A]^u)\,(\forall t \in T) \quad (\mathrm{C}^{s,u,v}\mathbb{S})(a)(t) = \mathbb{S}(x, a)(t)$$

wherein

$$x = (\mathbb{S}(x, a))_1.$$

Notice here that as x is defined recursively in terms of itself whether $\mathrm{C}(\mathbb{S})$ is computable will depend on the definition of $\mathbb{S}$.

(D) Forking. For each $s \in S$ we define the *fork constructor* ambiguously denoted fork with functionality

$$\mathrm{fork} : [T \to A]_s \to [T \to A]_s \times [T \to A]_s$$

by

$$(\forall a \in [T \to A]_s)\,(\forall t \in T) \quad \mathrm{fork}_s(a)(t) = (a(t), a(t)).$$

(E) Permuting. For each $s \in S$ we define the *permutation constructor* ambiguously denoted perm with functionality

$$\text{perm} : [T \rightharpoonup A]_s \times [T \rightharpoonup A]_s \rightharpoonup [T \rightharpoonup A]_s \times [T \rightharpoonup A]_s$$

by

$$(\forall a_1, a_2 \in [T \rightharpoonup A]_s)\,(\forall t \in T) \quad \text{perm}_s(a_1, a_2)(t) = (a_2(t), a_1(t)).$$

(F) Sinks. For each $s \in S$ and for each $u \in S^*$ we define the *sink constructor* ambiguously denoted sink with functionality

$$\text{sink} : [T \rightharpoonup A]^{s\,u} \rightharpoonup [T \rightharpoonup A]^u$$

by

$$(\forall a = (a_1, a_2, \ldots, a_{|u|+1}) \in [T \rightharpoonup A]^{s\,u})\,(\forall t \in T) \quad \text{sink}^{s\,u}(a)(t) = (a_2(t), \ldots, a_{|u|+1}(t)).$$

The use of streams. Again in common with the functional approach to stream programing, STREAM adopts the generalized concept of stream as the union of finite and infinite sequences.

Semantics. Both a denotational and algebraic semantics have been derived for STREAM (see Kloos [1987a] and Kloos *et al.* [1986] respectively). The denotational semantics is used in Kloos [1987a] to demonstrate the equivalence of STREAM with a procedural language for stream processing.

Language Development and Current Uses. We are not aware of the development of the use of STREAM in hardware design.

Syntax. SIGNAL uses two syntactic styles to reflect the different requirements of hardware description at different levels of abstraction: an applicative style and a functional style. The RS-Flip-Flop can be represented in the two styles as follows:

Applicative.

agent *RS-flipflop* =
  in r, s ***ni***
    $t :\equiv nor^*(r, s')$,
    $r' :\equiv tt \,\&\, t$,
    $u :\equiv nor^*(s, r')$,
    $s' :\equiv ff \,\&\, u$
  out r', s' ***tou***

Functional.

$$
\begin{array}{l}
\textit{agent RS-flipflop} = \\
\quad C(\\
\qquad C(\\
\qquad\quad (perm \Downarrow Id^* \Downarrow Id^*) \Rightarrow \\
\qquad\quad (Id^* \Downarrow perm^* \Downarrow Id^*) \Rightarrow \\
\qquad\quad (nor \Downarrow nor) \Rightarrow \\
\qquad\quad (ff\& \Downarrow tt\&) \Rightarrow \\
\qquad\quad (fork \Downarrow fork) \Rightarrow \\
\qquad\quad (Id^* \Downarrow perm \Downarrow Id^*) \\
\qquad) \\
\quad)
\end{array}
$$

3.9 Stream Processing in the Design and Verification of Hardware

As our research places a strong emphasis on the formal specification and verification of hardware, in this section we discuss the use of streams as a method for formal hardware description. We begin with a brief general overview of the topic of hardware specification followed by a more detailed discussion of how STs and SPSs provide a natural and general purpose mechanisms for the formal specification and implementation respectively of many types of hardware devices. However, for emphasis we leave a discussion of the theory of *synchronous concurrent algorithms* (SCAs) that is the basis of the techniques that we advocate to the following section.

3.9.1 Abstraction Levels and Formalized Hardware Description

Several levels of abstraction can be identified for the description of hardware. These are (in descending order of abstraction from the actual physical device)

(1) **The Conceptual Level.** A high-abstraction level characterized by the use of graphs and high-level algorithmic descriptions.

(2) **The Architectural Level.** Description by block diagrams.

(3) **The Register Transfer Level.** Architectural entities are identified that are synchronized by a global clock and the transfer of information between them is modelled by binary words.

(4) **The Logic Level.** Combinational and state-preserving elements are identified characterized by the laws of boolean algebra and additional mechanisms.

(5) **The Circuit (Gate and Switch) Level.** Gates and devices are identified, characterized by a simple description of power sources, transistors, resistors, etc.

(6) **The Geometry Level.** The geometry level is the lowest level of abstraction of hardware description and is subdivided into two further levels:

The Flexible Geometry Level. Characterized by the description of relative orientation of primitives (stick diagrams).

The Mask Geometry Level. Characterized by the description of physical sizes and absolute locations.

As pointed out in Stavridou [1993] constructing a formalism that encompasses the details and requirements of hardware at different levels of abstraction is a non-trivial task, as is the formal translation of a description at one level of abstraction into another level of abstraction. As such, it is common for hardware description languages to address a single level of abstraction, although this is not always the case (see Section 3.8.12). Indeed, there have been many experiments with the formal specification and verification of hardware using a variety of different languages and logics. However, a discussion of most of these approaches lies outside the scope of our research as they are not directly related to the subject of stream processing. The interested reader can consult the following general references on the subject of hardware specification and verification: Goguen [1987], Melham [1988], Milne [1989], Cohn and Gordon [1990], McEvoy and Tucker [1990], Weijland [1990], Johnson and Zhu [1991], Zhu and Johnson [1991] and Hanna and Daeche [1993].

3.9.2 The Advantages of Formal Hardware Description

One of the main advantages of a formal (syntactic) description of a hardware device is that when such a description is combined with an appropriate formal calculus it makes possible either an automated or a machine-assisted proof of device correctness. Moreover, the application of a mechanical theorem prover for the verification of hardware is particularly appropriate as correctness proofs tend to be straightforward, but long and tedious and hence error prone.

Despite this fact the application of formal methods to the study of hardware is by no means straightforward and as we mentioned in our introduction has had limited 'real-world' practical success. In particular, in Stavridou [1993] it is observed that at present the value of the specification of hardware at low levels of abstraction is limited in the sense that with current technology only devices comprised of a limited number of components can be tractibly modelled and formally verified with a sufficient degree of accuracy (see Section 8.5). Moreover, this is especially the case at the moment as current technology is based on transistors and the accurate modelling of devices based on transistors is difficult (see Gorden [1981]).

Furthermore, in Stavridou [1993] it is also pointed out that the improvements in the correctness of device fabrication that formal specification techniques provide can be difficult to quantify. More specifically, the benefits offered by formal techniques are typically observed through practical experience rather than by any mathematical argument to justify their effectiveness.

While these objections are justified we believe that there are significant advantages offered by formal techniques at high levels of hardware abstraction, particularly at the four most abstract levels of hardware description that we have identified. We justify this statement by observing that:

(1) Experimental evidence suggests that a significant proportion of design errors are introduced at high levels of abstraction and hence a formally-verified, high-level implementation of the

required hardware device is in general beneficial, especially in the context of safety-critical applications.

(2) With particular relevance to our reserch we believe that the use of streams at high levels of abstraction of hardware description provides a particularly natural mechanism for the formal description of many types of (safety-critical) devices. Indeed, this fact is demonstrated by the successful modelling and verification of hardware as SCAs.

In order to motivate these important points more fully in the remaining part of this section we explain in some detail how streams may be used for general purpose hardware description. We begin this discussion by examining a very straightforward and useful theoretical tool for hardware description based on the changes of a device's *state*. We continue by showing how this description method can be naturally considered as a special case of a more general stream-based description technique.

3.9.3 Streams and Hardware Description

At many levels of abstractions of hardware description the role of clocks (see Section 2.4.1) is an important one. The so-called *state transformer* formalization of hardware (see Harman and Tucker [1993]) relies on the use of an abstract clock $T = \{0, 1, 2, \ldots\}$ to provide a discrete measure of the evolution within a device of the values of (for example) the registers and memory from some initial values to the values at some time $t \in T$ – referred to as the evolution of the device's *state*

Given an algebra A coding all possible states, the state transformer view of hardware allows us to formalize a device in two related ways:

(A) As a function

$$F : A \to [T \to A]$$

defined by

$$F(a)(0) = a$$

and for each $t \in T$

$$F(a)(t+1) = \mathrm{NS}(t, a, F(a)(t))$$

wherein $a \in A$ is some initial state and $\mathrm{NS} : T \times A \times A \to A$ is referred to as the *next state* function for F that in general depends on the current time, the initial state value a, and the previous state of F; and

(B) A refinement of Method (A) based on the observation that it is typical, indeed desirable, for a piece of hardware to be independent of any particular initial state. Consequently, we may fix the initial state to be either a 'don't care' value or some fixed initial value $x \in A$. This provides a further more abstract definition of the hardware in question by the function

$$G^x : T \to A$$

defined relative to the initial state-dependent representation F by

$$(\forall t \in T) \quad G^x(t) = F(x)(t).$$

Generalizing the State Transformer Description. While the description of hardware as a pure state transformer is a useful and accurate model it is in some sense rather unrealistic as the evolution of a piece of hardware's state is also usually influenced by some external input. Moreover, typically any useful hardware will also produce some output and indeed the correctness of a piece of hardware will usually be stated in terms of the output generated as a result of its input. However, this fact is not problematic as we will now show that we can incorporate the state transformer description into a more general description of hardware based on abstract stream transformers.

The Stream Transformer Description Method. It is natural to imagine that in addition to its state any input and output to and from a piece of hardware will also change at discrete time intervals described by two (further) abstract clocks C and C', representing respectively the rate at which the device receives and produces output. Furthermore, typically the input received by a piece of hardware at any particular time $c \in C$ will be stored as some sub-set of its internal memory and register values; that is, as a sub-set of its state. Similarly, it is accurate to assume that the device's output at some time $c' \in C'$ can be derived in the reverse manner by selecting a particular sub-set of its memory as representing the 'result' of a computation.

Therefore, formalizing this idea: if the abstract data types B and B' are appropriate to code all possible input and output values respectively; the function

$$\chi : B \to A$$

models the appropriate change to the overall state given some input; and

$$\psi : A \to B'$$

models the appropriate output given the current state then our most general and abstract description of a piece of hardware (that we refer to as the *stream transformer description*, of which the state transformer description can be thought of as a special case) can be formalized as follows:

$$H^x : [C \to B] \to [C' \to B']$$

defined by

$$(\forall b \in [C \to B])\,(\forall c' \in C') \quad H^x(b)(c') = \psi(G^x(\chi(b(r(c')))))$$

wherein G^x is some state-independent state transformer specification of the hardware device we wish to model and

$$r : C' \to C$$

is a so-called *re-timing* (see Harman and Tucker [1988b], Harman [1989], Harman and Tucker [1990], and Harman and Tucker [1992]) that relates the 'ticks' of clock C' to the 'ticks' of clock C.

At this point the reader unfamiliar with these ideas may be surprised that we model a hardware device using several clocks; that is, as by definition a clock is simply an isomorphic copy of the natural numbers, why should we need more than one clock? Essentially, the reason

that we may need more than one clock is that the individual 'ticks' of each clock need not denote the same amount of 'real time' relative to either the actual device's system clock or the 'user's' notion of time. For example, as the general stream transformer description technique shows a device may consume stream input at a different rate than it produces stream output. Hence, multiple clocks are an essential tool for modelling hardware devices.

3.9.4 Hardware as Stream Transformers

The general method we have just discussed for describing (electronic) devices at the conceptual level provides the foundation for a theory of hardware based on the formal analysis of the computability of abstract functions (STs) that compute over streams, and the analysis of their algorithmic implementations (SPSs). Moreover, as we will show given a suitably abstract formalization of stream transformers much of this theory will have applications outside of the theory of hardware (see Section 7.5).

Indeed, the basis of just such a theory of stream transformers has already been developed and because it is suitably abstract makes use of many well-understood techniques from computability theory. This theory is referred to as the theory of *synchronous concurrent algorithms* and has provided the stimulation for our own work on stream processing. One of the main advantages of the SCA approach is that it relies on an essentially first-order semantic model and hence avoids many of the complications of a higher-order semantics that is a more typical approach when computing with streams. In the following section we discuss SCA theory in some detail and in particular the advantages that an algebraic approach to stream processing provides as the basis for a general purpose theory of stream processing.

3.10 Discussion: an Algebraic Approach to Stream Processing and SCAs

As our examination of the literature has shown, stream processing is a diverse subject without any clear overall objective that is based on specialized rather than general purpose theory. Moreover, rather than an abstract study of STs stream processing has typically been applications driven and has concentrated on either the study of special purpose languages for representing particular classes of SPSs or modifications of general purpose languages for the description of SPSs. Indeed, even the specialized stream processing languages we have identified are general purpose in the sense that these languages are intended for the specification, verification, and implementation (animation) of STs as SPSs in a single formalism. Therefore, as we discussed in Section 3.7 we believe such languages are not sufficiently abstract as a tool to develop the general theory of stream processing that we require.

In addition, from the perspective of the automated verification (of hardware) we have the following four additional specific objections to the approaches to stream processing that are found in the literature:

(1) Higher-order semantic models are used to reason about ST specifications and implementations. In particular, while in principle we have no objection to higher-order semantic

models, typically stream processing languages are based on a domain-theoretic semantics in the style of Kahn. Therefore, as the generation of an implementation from a least fixed-point semantics is non-trivial (see Cai and Paige [1989]) such languages typically require an 'equivalent' operational semantics that is used to effectively study specific computational properties.

However, the equivalence of the denotational (Kahn style) and operational semantics of stream processing languages is, as we have observed before, poorly addressed. Hence, as this means that essentially an operational semantics is used to reason about many stream processing formalisms based on a higher-order semantics, we argue that languages based on this approach are inappropriate as abstract specification languages for STs.

(2) General purpose languages mean that either invalid or incorrect or incomplete STs can be specified leading to non-determinism and deadlock (see Broy [1990] and Wadge [1981]).

(3) The expressive power of general purpose languages means that if automatic tools for the verification of STs are to be used then in general completion algorithms such as *Knuth-Bendix* (see Knuth and Bendix [1970]) must be applied to specifications (see Bellia *et al.* [1982] and Stavridou [1993]).

(4) The infinite size of streams means that traditional methods for the evaluation and representation of data structures for finite objects are not suitable and 'new' strategies must be used to implement and animate STs (see Henderson and Morris [1976] and Friedman and Wise [1976]).

In contrast to the existing approaches to stream processing we have identified, in our research we intend to develop a general and abstract theory of stream processing that specifically addresses these and other problems associated with current stream processing techniques. This theory is based on existing theoretical tools from recursive function theory and universal algebra and provides a mathematically neutral and well-understood approach to stream processing that is more appropriate for the abstract specification of STs.

In particular, the theory that we develop in the following chapters is based on the well-developed theory of *synchronous concurrent algorithms* (SCAs). As such we now discuss how we may abstract the basis of a general theory of stream processing from the techniques used in the formal specification and verification of SCAs.

3.10.1 SCAs

The concept of a *synchronous concurrent algorithm* (SCA) was developed by B C Thompson and J V Tucker in the early 1980s (see Thompson and Tucker [1985], Thompson [1987] and Thompson and Tucker [1991]). Informally, a SCA can be visualized as a particular class of dataflow SDU-SPS; that is, a SCA is as a fixed, synchronous, deterministic dataflow network wherein *modules* (filters) compute and communicate in parallel via *channels* synchronized by a

discrete global clock T.

In more detail, the most basic component of an SCA is a module that computes some predefined total function. Modules receive and supply data via fixed *channels* that communicate individual data elements between modules, and also communicate external input and output respectively. The parallel operation of the modules in a SCA is synchronized by the 'ticks' of a global clock. As such all modules receive and produce data deterministically and hence the SCA as a whole also computes a total function. An typical SCA is shown in Figure 3.2. We

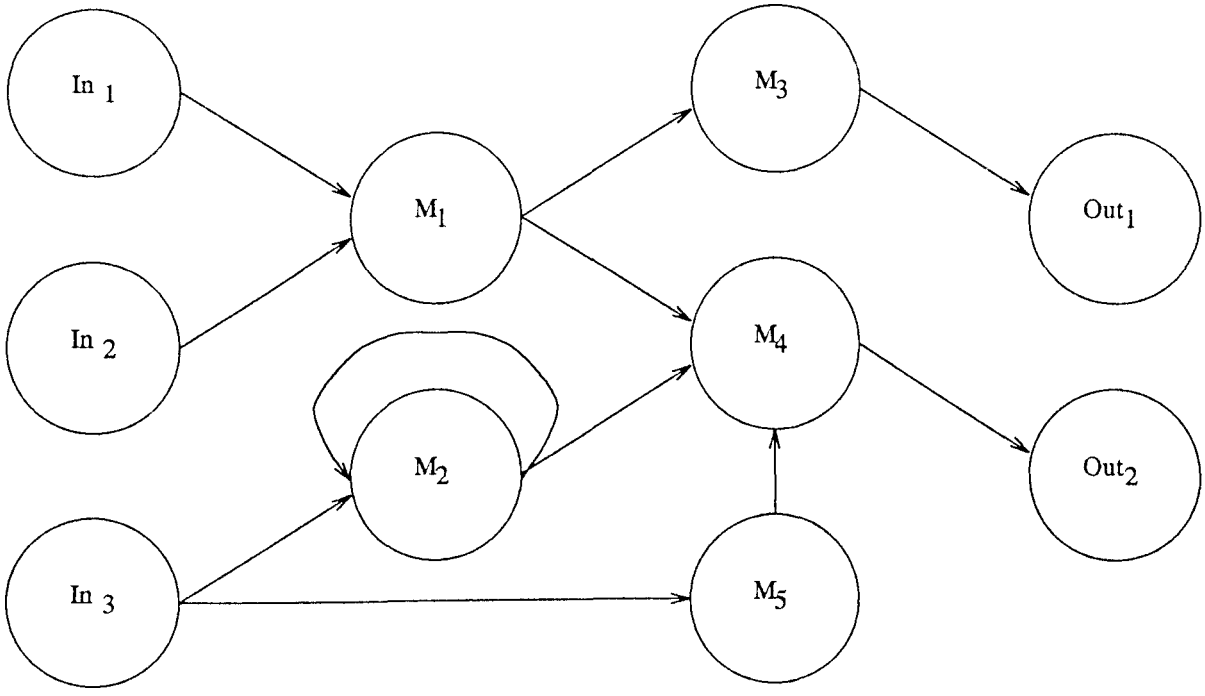

Figure 3.2: A Typical SCA

now discuss each of these features of an SCA more rigorously.

Modules. The set of all modules that make up a SCA are divided into three subsets: *sources* that receive the algorithms input; *internal modules* that perform some computation; and *sinks* that produce output.

Sources. Each source in a SCA is distinguished by the fact that it has no input and produces a single stream on its output channel. However, as with other channels this stream output may branch to pass multiple copies of the data to more than one internal module.

Internal Modules. Internal modules can have any finite number of inputs from either sources or other internal modules (including themselves) and must compute a total function. However, the function computed by an internal module need not be continuous with respect to any partial ordering (see Section 2.5) of the data they receive – as would be required if SCAs

were formalized by a domain-theoretic semantics. Indeed, two unique characteristics of SCA networks compared to other classes of dataflow SPSs are that:

(1) Associated with each internal module is an *initial value* that is output before the module begins to compute; that is, at time $t = 0$ determined by the global clock each module outputs a pre-defined value that is independent of any input that module will receive; and

(2) While internal modules may receive stream input from a source they are not viewed as producing stream output. Rather, each module produces *data* at each 'tick' of the global clock. As we will show this apparently trivial distinction is extremely important and is essentially the basis of the Cartesian form specification method that we mentioned in our introduction. We will return to this point in the following section.

Sinks. Sinks are characterized by the fact that they have no outputs and receive a single input from an internal module. In particular, sinks may not receive input from sources.

Channels. As we have indicated the channels in a SCA network are unidirectional and in addition may branch finitely allowing the copying of data. However, channels may not merge.

We now give a description of SCA computation and how the concept of an SCA and its operation can be formalized as a ST.

Architecture. Let n, m and k be the number of sources, sinks and modules respectively of some SCA wherein for $l = 1, \ldots, k$ each module has $n_l \in \mathbf{N}$ inputs. The architecture of the SCA is described formally by three partial functions α, β and *out* called *wiring functions* with the following functionality:

$$\alpha : \{1, \ldots, k\} \times \mathbf{N} \rightsquigarrow \{1, \ldots, max(n, k)\},$$

$$\beta : \{1, \ldots, k\} \times \mathbf{N} \rightsquigarrow \{S, M\}$$

and

$$out : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, k\}$$

respectively. For each module $i \in \{1, \ldots, k\}$ and for each input $j \in \{1, \ldots, n_i\}$ the meaning of these functions is defined as follows: if the jth input to module i is from module $x \in \{1, \ldots, k\}$ then

$$\alpha(i, j) = x$$

and

$$\beta(i, j) = M.$$

Otherwise, if the jth input to module i is from source $x \in \{1, \ldots, n\}$ then

$$\alpha(i, j) = x$$

and

$$\beta(i, j) = S.$$

Finally, if module $x \in \{1, \ldots, k\}$ is the input to sink $o \in \{1, \ldots, m\}$ then

$$out(o) = x.$$

Example 2. The architecture of the Flip-Flop as described in Section 3.8.1 can be formalized as an SCA as follows:

$$\alpha : \{1,2\} \times \mathbb{N} \rightsquigarrow \{1,2\}$$

is defined by

$$\alpha = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,1) \mapsto 1, (2,2) \mapsto 2\};$$

$$\beta : \{1,2\} \times \mathbb{N} \rightsquigarrow \{S, M\}$$

is defined by

$$\beta = \{(1,1) \mapsto S, (1,2) \mapsto M, (2,1) \mapsto M, (2,2) \mapsto S\}$$

and $out : \{1\} \rightsquigarrow \{1,2\}$ is defined by

$$out = \{1 \mapsto 1\}.$$

An Informal Description of SCA Computation. The initial values that are output by each module at the first clock cycle of the network's operation are formally represented by a vector $z = (z_1, \ldots, z_k) \in A^k$ wherein A is the data type over which the functions associated with each module compute; that is, the output at time $t = 0$ of module $i \in \{1, \ldots, k\}$ is z_i. In particular, the output of a SCA's initial values takes place independently of any subsequent computation that is performed. More specifically, each module in the SCA network computes in a synchronous output, read, compute, store cycle that is governed by the global clock T. At each clock tick $t \in T$ each module i first outputs the result it computed at the previous clock cycle (with the exception that at time $t = 0$ each module outputs its initial value z_i) and simultaneously reads in the value(s) at its input channel(s) $x_1, \ldots, x_{n_i}$. Each module now computes the value $y = f_i(x_1, \ldots, x_{n_i})$ wherein f_i is the total function associated with module i and stores this value ready to be output at the next clock cycle $t + 1$.

The synchronicity of the SCA network is achieved by the combination of the totality of each module function and the following assumption (compare Sections 3.3.5 and 3.8.5):

The Unit Delay Assumption U. The value on each output channel of each internal module at time $t + 1$ is uniquely determined by its input at time t.

Essentially, this means that at each clock tick $0, 1, 2, \ldots$ the modules compute and concurrently exchange data wherein any module that takes less than one time interval to compute 'waits' for any slower modules to complete their computation.

3.10.2 A Formal Algebraic Model of SCA Computation

Despite the informal similarity between SCAs and dataflow networks there are many technical reasons, including the approach to computing with streams and the formalization of a denotational semantics, that separate SCAs from dataflow computation. In particular, SCAs have been designed to provide a formal model of computation suitable for the rigorous study of the properties of hardware. Consequently, as we will discuss in the following sections, the properties

of SCAs are understood from the perspective of *recursive function theory*, *specification theory* and *verification theory* in the sense that the mathematical properties of the languages and logics needed to specify and reason about SCAs are understood in precise detail.

As we discussed previously, dataflow was conceived with very different aims and objectives from that of the formal specification of hardware, although some researchers have tried to adapt dataflow to this use. For this reason we believe that SCA theory has significant advantages over dataflow computing (and all the other stream processing formalisms that we have discussed) as the basis for a formal stream processing paradigm.

We now discuss these ideas in more detail beginning with a formal algebraic description of SCA computation.

Formalizing the Components of an SCA. Recall that an SCA network N is comprised of k modules computing functions $f_1, \ldots, f_k$ respectively, a global clock T and an underlying algebra A from which the values over which the network computes are taken. To begin our formalization of a SCA network N's computation, firstly we gather together some of these basic constituents into what is referred to as the *underlying algebra*, denoted U_N. While U_N will vary depending on the particular network N the minimum requirement that we place on U_N is that it is an enrichment of a standard algebra; that is, that U_N will always have the following standard algebra as a reduct:

$$(A, T, \mathbb{B}; 0, Succ, tt, ff, not, and, or).$$

This assumption is both necessary and convenient as we must have the natural numbers to count the successive 'ticks' of the network's clock, and we also have some basic functions available for specification purposes (see Section 6.7.1).

Secondly, in order to formalize the type of stream input that network N receives and also to fix the basic level of abstraction over which we can formalize the computation performed by network N, we enrich the underlying algebra to form the *component algebra*, denoted $\underline{A}_N$, as follows:

$$\underline{A}_N = (A, T, \mathbb{B}, [T \to A]; f_1, \ldots, f_k, 0, Succ, tt, ff, not, and, or, eval).$$

Again, notice that $\underline{A}_N$ is a standard algebra, although this time a stream algebra (see Definition 2.4.2), enriched with the basic functions computed by each module in the network.

The component algebra enables us to rigorously define the computation of an SCA as a ST as follows.

Formalizing SCA Computation: Value Functions. Let N be any n source, m sink, k module SCA with global clock T, wiring functions α, β and *out*, and component algebra $\underline{A}_N$. If $a = (a_1, \ldots, a_n) \in [T \to A]^n$ is the stream input received by network N's sources and $z = (z_1, \ldots, z_k)$ are network N's initial values then we formalize the computation of network N with the *value functions*

$$V_i : T \times [T \to A]^n \times A^k \to A$$

for each $i = 1, \ldots, k$ as follows:

$$V_i(0, a, z) = z_i$$

and for each $t \in T$

$$V_i(Succ(t), a, z) = f_i(x_1, \ldots, x_{n_i})$$

wherein for $j = 1, \ldots, n_i$

$$x_j = \begin{cases} a_{\alpha(i,j)}(t) & \text{if } \beta(i,j) = S; \text{ and} \\ V_{\alpha(i,j)}(t, a, z) & \text{if } \beta(i,j) = M. \end{cases}$$

The intention here is that the ith value function V_i is defined such that $V_i(t, a, z)$ denotes the value output by module i at time t. Therefore, if we define

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

by

$$(\forall t \in T)\,(\forall a \in [T \to A]^n)\,(\forall z \in A^k) \quad V_N(t, a, z) = (V_1(t, a, z), \ldots, V_k(t, a, z))$$

then V_N tells us the output of every module at every clock cycle. The function V_N is referred to as the *global state function*, although as in general we are only interested in the networks output at its sinks, we also define the network N's *output specification*

$$V_{out} : T \times [T \to A]^n \times A^k \to A^m$$

by

$$(\forall t \in T)\,(\forall a \in [T \to A]^n)\,(\forall z \in A^k) \quad V_{out}(t, a, z) = (V_{out(1)}(t, a, z), \ldots, V_{out(m)}(t, a, z))$$

to represent the output of only certain specific modules of interest.

Example 3. The RS-Flip-Flop is described as an ST by the following value functions:

$$V_1, V_2 : T \times [T \to \mathbb{B}]^2 \times \mathbb{B}^2 \to \mathbb{B}$$

defined by

$$(\forall b_1, b_2 \in [T \to \mathbb{B}])\,(\forall z_1, z_2 \in \mathbb{B})$$

$$V_1(0, b_1, b_2, z_1, z_2) = z_1;$$

$$V_2(0, b_1, b_2, z_1, z_2) = z_2;$$

and

$$(\forall t \in T)\,(\forall b_1, b_2 \in [T \to \mathbb{B}])\,(\forall z_1, z_2 \in \mathbb{B})$$

$$V_1(Succ(t), b_1, b_2, z_1, z_2) = Nor(b_1(t), V_2(t, b_1, b_2, z_1, z_2));$$

and

$$V_2(Succ(t), b_1, b_2, z_1, z_2) = Nor(V_1(t, b_1, b_2, z_1, z_2), b_2(t)).$$

In particular, notice that the wiring functions α, β and *out* of Example 2 can be effectively derived from the value functions coding a particular network. Therefore, not only do value functions encode the semantics of a network, but also the necessary syntactic information to reconstruct the network itself. Indeed, in general, value functions provide a simple recursive method for calculating (either symbolically or otherwise) the output of an SCA at any time $t \in T$. This is why we refer to this class of systems as *synchronous concurrent algorithms* reflecting not only the nature of computation performed by each network (that is, synchronous parallelism), but also that each network has a precise, straightforward and denotational description wherein both the underlying algebra and the function computed by the network are explicit. We believe this is a significant advantage over Kahn's method (see Section 2.5) that is more complicated at the syntactic level of network description, and also relies on a semantics that in practice is neither explicit nor necessarily effective.

In more detail, by using Kahn's method for formalizing complex SPSs it it not obvious at all what the least fixed point of the function that the network specifies may be, and hence if the network specifies a function at all! For example, the network might specify the everywhere undefined function, but in general there is no effective procedure for determining this fact! Moreover, Cai and Paige [1989] shows that the construction of a function from a least fixed-point semantics can be extremely difficult. Therefore, even if the network does specify a function then it is still not clear how in practice we can effectively construct the networks least fixed point and hence formally simulate the networks operation.

In addition, SCAs have an extra advantage as a specification technique in that it is obvious from the restricted equational structure of value functions and the totality of each function computed by the individual modules that value functions are a special case of *primitive recursive functions* (see Chapter 4). We return to this point in the following section and in more detail in later chapters where we will show that the explicit primitive recursiveness of equational definitions has many useful theoretical and practical implications.

The second important observation that we can make at this point about the general form of value functions is concerned with their functionality as stream transformers. Specifically, notice that our general definition of a stream transformer as described in our preliminaries is essentially a function of the form

$$F : [T \to A]^n \to [T \to A]^m$$

for some data type A and for some $n, m \in \mathbb{N}$. However, both the individual value functions V_i for $i = 1, \ldots, k$, the global state function and the output specification V_N and V_{out} respectively are functions of the form

$$G : T \times [T \to A]^n \times A^l \to A^m$$

for some $n, m, l \in \mathbb{N}$ and hence are not strictly speaking stream transformers. In particular, this difference in functionality raises the question: if we wish to exploit the advantages of the SCA specification technique as the basis of a general stream processing theory then can we reconcile this difference in functionality with the more usual approach to stream transformer specification? As we will show in detail the answers to this question is yes! Indeed, based on the following two observations, naively it appears that the reconciliation of these two methods is trivial:

(1) With respect to the elimination of non-stream input in the domain of G we can apply the same technique that we discussed in Section 3.9.3 to derive a family of functions indexed by each $x \in A^l$ with functionality

$$G^x : T \times [T \to A]^n \to A^m$$

defined relative to G by

$$(\forall t \in T)\,(\forall a \in [T \to A]^n) \quad G^x(t, a) = G(t, a, x)$$

(2) We can now observe that the family of functions G^x are essentially nothing more that the *un-Curried form* of F – what we call a *Cartesian form stream transformer* (CFST) rather than what we refer to as either a *Curried* or *applicative form stream transformer* (AFST) that we have described so far.

Therefore, the distinction between the CFST specification method and the AFST specification method appears to be of little significance in the sense that for each CFST

$$H^* : T \times [T \to A]^n \to A^m$$

we can define an equivalent AFST

$$H : [T \to A]^n \to [T \to A]^m$$

by

$$(\forall t \in T)\,(\forall a \in [T \to A]^n) \quad H^*(t, a) = H(a)(t).$$

However, as we will show this apparently unimportant difference in specification technique is subtle in its implications. Specifically: from the perspective of computability the reconciliation of these two techniques is by no means straightforward; and from the perspective of automated verification the use of Cartesian forms has significant advantages in that it will allow us to apply first-order techniques to establish the correctness of stream transformers. Indeed, after we conclude this section with an SCA bibliography we motivate the issues that we must address in our research to exploit the use of both explicit primitive recursive definition and Cartesian form specification.

An SCA Bibliography. SCAs have been studied extensively as a formalism for the specification and verification of hardware including several case studies (see Harman and Tucker [1988b], Harman and Tucker [1988a], Harman [1989], Eker and Tucker [1989], Harman and Tucker [1990], Eker *et al.* [1990] and Harman and Tucker [1992].)

Furthermore, the SCA model is also appropriate for the study of many specialized hardware devices and specialized models of computation including: *systolic arrays* (see Thompson and Tucker [1985], Thompson [1987], Hobley *et al.* [1988], Derrick *et al.* [1989] and Hobley [1990]); *neural networks* (see Holden *et al.* [1991a], Holden *et al.* [1991b], Holden *et al.* [1992b], Yates [1993] and Thompson *et al.* [1992]); and *cellular automata and coupled map lattice dynamical systems* (see Marshall [1991], Blom [1992], Holden *et al.* [1992a], Holden *et al.* [1993] and Blom *et al.* [1993]). For general introductions to the topics of systolic architectures; neural networks; and cellular automata and coupled map lattice dynamical systems see respectively Mead and

Conway [1980], Kung [1982]; McCulloch and Pitts [1943], Widrow and Hoff [1960], Minsky and Papert [1969], Kohonen [1972], Kohonen [1978], Rumelhart *et al.* [1986], Rumelhart and McClelland [1986a], Rumelhart and McClelland [1986b]; Kamp and Hasler [1990] and Hansson [1993]; and von Neumann [1966] and Wolfram [1986].

Finally, the SCA computational model has been generalized and formalized in several ways: *graph theoretical models* (see Meinke [1988] and Meinke and Tucker [1988]); *process theoretic models* (see Tofts [1993]); *operational semantic models* (see Thompson [1987], Martin and Tucker [1988], Martin [1989] and Poole [1994]); and *infinite SCAs* (see McConnell and Tucker [1993] and McConnell [1993]).

3.10.3 The Advantages of the SCA Methodology as a Basis for Formal Stream Processing

SCA research has shown that the algebraic specification of hardware as primitive recursive Cartesian form stream transformers offers significant advantages. In particular, from the perspective of the formal verification of hardware, SCA theory is extremely useful. In our research we generalize the SCA approach and use this as (an alternative) basis for a formal approach to stream processing, although, the verification of hardware as SCAs remains of significant importance and interest.

In this section we motivate what we believe are the advantages of this approach, and in the following section we set a research agenda for the rest of this thesis so that we may begin to properly explore the theoretical and practical implications of primitive recursive Cartesian form equational specification in the broader setting of general purpose stream processing.

Cartesian Forms. A *Cartesian form stream transformer* (CFST) is a function of the form

$$G : T \times [T \to A]^n \to A^m$$

for some $n, m \in \mathbb{N}$. This is an alternative form of specification for a stream processing system in the sense that for every AFST F with functionality

$$F : [T \to A]^n \to [T \to A]^m$$

we can define a map

$$F^* : T \times [T \to A]^n \to A^m$$

by

$$(\forall t \in T)\,(\forall a \in [T \to A]^n) \quad F^*(t, a) = F(a)(t).$$

We call F^* the *Cartesian form* of F (and sometimes the *weak second-order form* of F) and as far as we are aware, as a specific specification methodology for modelling second-order specifications the use of Cartesian forms is unique outside of SCA theory.

Notice that CFSTs return data and not functions (streams). Consequently, we may use an extremely weak second-order model (essentially first-order) to formalize the specification of STs that cannot be used when using AFSTs to specify stream transforming systems. This fact gives us two distinct advantages:

(1) We have a straightforward theoretical account of STs based on first-order methods. In particular, we will show that we may use methods based on first-order equational logic to reason about CFSTs. This fact is particularly useful as most, indeed if not all, of the underlying data types used in computer science (and hence used in stream processing) have equational theories. We will return to this point in more detail in the following chapter.

(2) CFSTs eliminate the need for the specialized evaluation of the particular infinite data structures that are used to represent the stream data generated by an AFST. We justify this statement by observing that as CFSTs simply return data we need now only evaluate (external) stream input. Since for any stream input a the availability of the individual data values $a(t)$ and $a(t')$ for some $t \neq t'$ will (from the perspective of the specification) be mutually exclusive, we obviate the need for any partial evaluation strategy. Therefore, in principle we may use any general purpose language for the animation and implementation of CFSTs.

Primitive Recursive CFSTs. Finally, since each primitive recursive function is total, any primitive recursive ST specification will be effectively testable against a corresponding primitive recursive implementation. Moreover, we will show that primitive recursive specifications when presented in the language PR (see Chapter 4) that is used for the specification of SCA can be compiled directly into equivalent PREQ specifications that are *complete* when considered as left-to-right re-write rules (see Chapter 5).

Therefore not only does PREQ provide the basis of a high-level equational specification language, but its theoretical properties can also be used as the basis of automated verification tools that we present in Chapter 8.

3.10.4 Developing an Algebraic Approach To Stream Processing

We believe the advantages of primitive recursive Cartesian form specification of STs are clear in the context of hardware verification. Our objective now is to develop this methodology for general purpose stream programming. In particular, we must address the following:

(1) Using CFSTs Specification as a Practical Tool: Cartesian Composition.

One important aspect of CFSTs that we have not discussed so far that the reader may have noticed is that they do not appear to provide a compositional model of stream processing; that is, as we have already pointed out CFSTs take streams as input, but return data as output. One of the appealing aspects of AFSTs is that they allow a modular and hierarchical approach to stream processing. In particular, given two AFSTs H and G (that represent two systems S_1 and S_2 respectively) such that H and G have functionality

$$H : [T \rightarrow A]^n \rightarrow [T \rightarrow A]^m$$

and

$$G : [T \rightarrow A]^m \rightarrow [T \rightarrow A]^p$$

for some $n, m, p \in \mathbb{N}$ we may define a AFST F with functionality

$$F : [T \rightarrow A]^n \rightarrow [T \rightarrow A]^p$$

(to give a composite system S) as follows

$$(\forall t \in T)\,(\forall a \in [T \rightarrow A]^n) \quad F(a)(t) = G(H(a))(t).$$

However, given the equivalent Cartesian forms of H and G:

$$H^* : T \times [T \rightarrow A]^n \rightarrow A^m$$

and

$$G^* : T \times [T \rightarrow A]^m \rightarrow A^p$$

respectively we can no longer directly define the composite system represented by

$$F^* : T \times [T \rightarrow A]^n \rightarrow A^p$$

as the 'definition'

$$(\forall t \in T)\,(\forall a \in [T \rightarrow A]^n) \quad F^*(t, a) = G^*(t, H^*(t, a))$$

is *not* well-typed.

Naively, adopting a standard functional-programming technique, we could simply define F^* from H^* and G^* using λ-abstraction as follows:

$$(\forall t \in T)\,(\forall a \in [T \rightarrow A]^n) \quad F^*(t, a) = G^*(t, \lambda t.H^*(t, a)).$$

However, this would require adding λ-abstraction as a primitive operation to our specification language PREQ (see Point (2) to follow). The subtle problem with this approach is that existing theory has shown that λ-abstraction when combined with the language PR (that captures the class of primitive recursive functions) provides a stronger model of computation than PR alone (see Tucker and Zucker [1992]); that is, it is not *computationally conservative*. Consequently, as PREQ is formally equivalent to PR in its expressive power if we add λ-abstraction as a primitive to PREQ then we will lose the theoretical properties that PREQ enjoys by virtue of its primitive recursiveness.

This complication means that if we are to use the SCA methodology as the basis of a stream processing theory then we must show that for any CFSTs H^* and G^* (defined as above) we can construct F^* (defined as above) representing a composite system *without* the use of full λ-abstraction. Furthermore, since one of our aims in this thesis is that our theory can provide the basis of software tools, clearly we require a *constructive* proof that CFSTs are compositional in this sense.

To this end in Chapter 4 we show constructively that:

(A) The class of primitive recursive CFSTs is closed under Cartesian composition.

(B) The class of μPR computable CFSTs (see Section 4.4.2) is closed under Cartesian composition; that is, we are able to show that the SCA methodology is appropriate for the most general class of computable STs and hence our first-order theory of stream processing is indeed general purpose.

In more detail, Properties (A) and (B) above of CFSTs are stated formally in Theorem 7. We prove Theorem 7 constructively using Theorem 9 that concerns the properties of a formal compiler $\mathbb{C}$ such that given two schemes $\alpha_{H^\bullet}$ and $\alpha_{G^\bullet}$ representing the Cartesian forms of the STs H and G respectively, $\mathbb{C}(\alpha_{H^\bullet}, \alpha_{G^\bullet})$ produces a scheme $\alpha_{F^\bullet}$ such that the semantics of $\alpha_{F^\bullet}$ is F^* the Cartesian form of $F = G \circ H$.

In addition, in Chapter 8 we also discuss the implementation of a slightly generalized version of the Cartesian composition compiler denoted $\mathcal{C}$ that is defined in Chapter 7. The compiler $\mathcal{C}$ is tailored for specific practical requirements arising from the use of Cartesian composition in the formulation of the semantics of ASTRAL. Specifically, the compiler $\mathcal{C}$ will allow us to use CFSTs for specification in the same way that AFSTs can be used, and hence will allows us to apply modular specification techniques (see Point (3) to follow).

(2) Developing a Denotational Semantics for Stream Processing: PREQ. As our discussion of SCA theory has shown there is a strong theoretical motivation for using an equational specification methodology. In addition, from a more general perspective an equational language provides a high-level of mathematical abstraction that is well-suited to implementation independent representation. In particular, for the specification of CFSTs an equational language is particularly appropriate and satisfies the criteria for the abstract representation of STs that we set down in Section 3.7.1. Furthermore, as mentioned previously, primitive recursive equational specifications give rise to equivalent *complete* term re-writing systems. This fact combined with the observation that typical data types in Computer Science have equational theories means that not only may we use equational specifications for CFSTs, but we may also use equational logic as the basis of a calculus for reasoning about their correctness (see Chapter 7).

In order to exploit the advantages that the equational specification of CFSTs provide in Section 6 we develop the specification language ASTRAL. In particular, in order that we maintain a sufficient level of mathematical abstraction, and hence avoid the problems that we have highlighted with existing approaches to stream processing, we will provide a denotational semantics for ASTRAL by compiling our ST specifications into a first-order equational specification language PREQ. Furthermore, in order to maintain precise control over issues relating to computability if we restrict the syntax of PREQ so that it captures the class of primitive recursive functions. This fact is particularly important if we wish to maintain the advantages of the SCA methodology for hardware specification. Specifically, should we wish to relax the requirement that a specification should be primitive recursive, as is sometimes required outside of the context of hardware specification, then using PREQ we may do this in a controlled way. If we do not take this approach to the formulation of a denotational semantics then we will have to face the problems associated with general purpose specification languages that we discussed at the beginning of this section – especially as unrestricted equations provide an extremely powerful specification technique in the sense of the class of functions they may specify.

In more detail, PREQ provides a straightforward syntax for the equational representation of primitive recursive functions and hence can be given a semantics in a rigourous way. Therefore, by compiling ASTRAL specifications into PREQ to derive a semantics, we can develop an implementation of ASTRAL that is suitable as a structured, modular high-level programming

language, but which can still be given a formal semantics. In particular, ASTRAL can be given a formal semantics using this technique without encountering the complications in developing a semantics for a high-level language that are typically observed when a top-down approach to language design is adopted. For example, languages such as C and PASCAL have usable implementations, but the direct formulation of a rigourous semantics for these languages is not straightforward as a result of their complex syntax.

As such, we continue the development of our 'first-order' stream processing theory in Chapter 5 as follows:

(A) First, we design the syntax and semantics of the language PREQ.

(B) Secondly, we show formally that PREQ is *sound* and *complete* with respect to the class of primitive recursive functions (Theorem 10), and hence that PREQ provides an appropriate, neutral specification tool to develop a theory specific to stream processing without any loss of mathematical abstraction. We do this by designing two formal compilers: $\mathbb{C}^{\mathrm{PREQ}}$ and $\mathbb{C}^{\mathrm{PR}}$ that compile PR into PREQ and PREQ into PR respectively. In addition, we pay particular attention to the design of the compiler $\mathbb{C}^{\mathrm{PREQ}}$ from the perspective of the number of equations that are produced from a PR scheme, so that $\mathbb{C}^{\mathrm{PREQ}}$ can be used as the basis of efficient software tools.

(C) Finally, we establish formally the important fact that we wish to carry over from the SCA methodology: that primitive recursive specifications give rise to equivalent complete term re-writing systems when expressed directly in PREQ (Theorem 11).

(3) Developing an Abstract Specification Language for STs: ASTRAL.

One of the aims in the development of ASTRAL is to provide a usable, user-friendly specification language for STs. As previously discussed, while CFSTs have mathematical advantages from the perspective of formal verification, AFST's provide a more natural and directly modular specification technique. Therefore, while designing ASTRAL, as its semantics is derived using CFSTs, we have been careful to develop the ASTRAL syntax without losing the mathematical abstraction we have carefully preserved. In order to achieve these aims we proceed as follows:

(A) In Chapter 6 we begin by defining an abstract mathematical formalization of ASTRAL based on applicative form specification techniques, and hence that provides the basis for the hierarchical (modular) implementation techniques that are useful in systems design and implementation.

(B) Secondly, we define a formal compiler that maps abstract (applicative) ASTRAL specifications into equivalent Cartesian form PREQ specifications and hence provide a formal (first-order) denotational semantics for ASTRAL.

(C) Finally, we present a prototype BNF for an implementation of ASTRAL based on the abstract mathematical formalization and comment on the underlying design criteria.

In this way we avoid the complications with languages such as C and PASCAL that we discussed at the beginning of this section.

(4) Developing a Theory of the (Automated) Verification of STs. The formalization of ASTRAL's semantics using PREQ means that the theoretical properties that PREQ enjoys with respect to the construction of equivalent TRSs (Theorem 11) are also enjoyed by ASTRAL. As we have discussed in Section 3.9.1 typically the formal verification of hardware requires routine, but long and hence error prone proofs. Therefore it is important to fully utilize the advantages offered by using the properties that PREQ specifications possess when interpreted as left-to-right re-write rules; that is, completeness.

Typically, the intended semantics of any data type is its *initial algebra semantics* (see for example Meinke and Tucker [1992]) unique up to isomorphism that captures the essential characteristic of any acceptable implementation of the operations that the data type describes. However, while complete TRSs are useful in the sense that they provide a decidable equational theory, this decidability is with respect to truth in all models of the data type (sometimes referred to as *loose semantics* – see Goguen [1988] and Goguen [1990]) that is more general than initial truth. In particular, loose validity implies initial validity, but initial validity *does not* imply loose validity (see Section 7.1.3). Of course in general we cannot hope to have a complete and decidable theory with respect to initial algebra semantics by Gödel's famous incompleteness result. Furthermore, SCA theory has shown that in general deciding the correctness of an implementation of a hardware specification when expressed as a ST is equivalent to deciding the membership of a co-recursively enumerable set, and hence is co-semi-decidable (see Thompson and Tucker [1994] and Davis *et al.* [1976]). However, as we are interested in automated theorem proving one important question we must answer is: how much does Theorem 11 gain us in terms of the decidability of ST verification?

We will show that in order to answer this question we can make use of the following two facts:

- Over ground terms (variable free terms) loose validity is equivalent to initial validity, and
- Free variable induction and equational logic are sound with respect to initial validity.

Specifically, we will show that these two fact will allow us to to the following:

(A) First, given an ASTRAL representation of a ST specification and a corresponding ASTRAL representation of an implementation we can characterize the decidability of the correctness of the implementation relative to the syntactic complexity of an equational correctness statement (Theorem 18). Essentially, this means that we can identify a useful syntactic sub-class of all decidable equational correctness statements relating STs whose correctness can be verified fully automatically.

(B) Secondly, we may use this syntactic characterization to implement an automated theorem prover designed around first-order term re-writing and free variable induction to reason about ST specifications. This provides precisely the basis for a theory of the automated verification of STs that we require and as we will show has important implications for the formal automated verification of (safety-critical) hardware when expressed as SCAs.

Concluding Remarks. The realization of topics discussed in Points (1), (2), (3) and (4) of this subsection are the theoretical and practical agenda around which the development of the rest of this thesis is based. In particular, in Chapter 8 we demonstrate the effectiveness of our theory of stream processing by using the automated verification of the RS-Flip-Flop as a small case study. We also discuss practical ideas to improve the efficiency of the implementation of our software tools to make them appropriate for complex hardware devices such as microprocessors.

Therefore as promised we begin our research agenda by developing PREQ and constructively demonstrating that it has the theoretical properties that we require.

Chapter 4

Primitive Recursion

Cultivate simplicity...

Charles Lamb

4.1 Introduction

The class of primitive recursive functions ($\mathbb{PR}$) was first formally identified as a specific functional class late last century by R Dedekind in Dedekind [1888] with the intention that they could be used as a foundational structure for mathematics in the reductionist style of Hamilton (Kiliminster [1993]). Indeed, since that time primitive recursion has played a fundamental role in classical computability theory as the most basic class of total functions formulated by recursive definitions. Moreover, it was as late as the 1930s that S Kleene suggested that partial recursive functions (primitive recursion with least number search – see Section 4.4.2) should be considered as the most general class of functions in the study of abstract computation. While this fact may now seem surprising, the use of $\mathbb{PR}$ as a general model of computation is in some sense an obvious choice as most 'everyday' functions (in what has become Computer Science) are primitive recursive.

Typically in classical computability theory primitive recursive functions are formulated over the natural numbers. However, for our purposes we require a more abstract definition that can be characterized informally as follows:

Definition 9. For any standard S-sorted Σ-algebra A if $f : T \times A^u \to A^v$ is defined by

$$(\forall a \in A^u) \quad f(0, a) = g(a)$$

and

$$(\forall t \in T)\,(\forall a \in A^u) \quad f(t+1, a) = h(t, a, f(t, a))$$

for some functions $g : A^u \to A^v$ and $h : T \times A^u \times A^v \to A^v$, for some $u, v \in S^+$ then we say f *is defined by an immediate application of primitive recursion.*

Early this century T Skolem (Skolem [1923] – work of 1919) developed the idea of providing a foundation for elementary arithmetic by combining primitive recursive function and predicate definitions with induction as a proof technique. This system, known as *primitive recursive arithmetic*, does not allow either unbounded existential quantification or unbounded negated existential quantification and as Skolem himself observed is similar to an intuisionistic approach to arithmetic, although it is actually more restrictive.

Since its development primitive recursive arithmetic has been studied by several researchers see for example Hilbert and Bernays [1934], Curry [1941], Goodstein [1941], Church [1954], Church [1957a], Church [1957b], Goodstein [1957], Rose [1961] and Rose [1962]. This work includes a consistency proof, some incompleteness results developed from a formalized metatheory and the study of primitive recursive arithmetic as a logic-free calculus. As pointed out in Curry [1941] much of the impetus of this work was the use of primitive recursive arithmetic by K Gödel in the formulation of his famous incompleteness theorems.

Our interest in primitive recursive functions, and what essentially amounts to primitive recursive arithmetic, is stimulated by their advantages as a specification formalism, and as a formal calculus respectively, to study the properties of hardware and certain specialized abstract computational models that can be formalized as STs. Indeed, as we have indicated in Section 3.10, the next chapter will concentrate on the development of an equational formalization of the class

$\mathbb{PR}$ (the language PREQ) and later chapters will examine the automation of primitive recursive arithmetic using techniques based on term re-writing.

In the next two Sections we motivate our choice of using the class $\mathbb{PR}$ as a specification formalism, and in Section 4.1.3 we overview the material presented in the rest of this chapter.

4.1.1 $\mathbb{PR}$ – A Secure Base

While our work is more specialized than classical studies of $\mathbb{PR}$ and primitive recursive arithmetic in that we are specifically concerned with their applications to stream processing, it is important to observe that much of the existing theory of primitive recursion carries over to any standard abstract algebra. Consequently, as stream algebras are standard algebras, by developing the semantics of ASTRAL using PREQ much of the theory of stream processing using ASTRAL is already implicitly known. Moreover, this fact means that we have an intellectually accessible and trusted tool as the basis of the theory of stream processing that we wish to develop – as T Skolem observed 'a secure base'.

Indeed, the generalization of the theory of computability over the natural numbers to algebras with streams has already been considered in Tucker and Zucker [1992] and Tucker and Zucker [1994] that includes the formalization of a Church-Turing thesis for stream computation. Therefore, not only is much of the algebraic theory of stream processing already developed, but we also have a mathematical 'yard-stick' against which we may measure certain aspects of the development of the theory of stream processing in this thesis (see Section 4.4).

4.1.2 A Formal Language for the Class $\mathbb{PR}$

As part of its role in computability theory many formal characterizations of the primitive recursive functions have been developed (see for example Péter [1950], Goodstein [1961], Cutland [1980], Simmons [1988], Tucker and Zucker [1992] and Tucker and Zucker [1994]). In particular, in SCA theory the languages *FPIT* (see Thompson [1987]), *CARESS* (see Martin [1989] and Poole [1994]) – both based on the concurrent assignment statement (see Welch [1983]) – and the functional language PR (see Thompson [1987]) are of interest to us. Indeed, the fact that so many languages already exist to express primitive recursive functions raises the question: why should we need to formulate another? We argue that the reason we need a further equational characterization of PR arises from the following two facts: (1) several of the existing languages are based on an operational semantics and hence are not appropriate as specification languages; and (2) the existing formalisms with a denotational semantics are too low-level in the sense that they are not suitable for high-level, user-friendly specification.

Despite having made these observations about existing languages as we will need to show that our equational language PREQ does indeed capture the class $\mathbb{PR}$ the most obvious (and constructive) way to do this is to show the following:

$$(\forall \Phi \in \mathrm{PREQ})\,(\exists \alpha \in \mathrm{L}_{\mathbb{PR}}) \quad [\![\alpha]\!] = [\![\Phi]\!]$$

and the converse

$$(\forall \alpha \in \mathrm{L}_{\mathbb{PR}})\,(\exists \Phi \in \mathrm{PREQ}) \quad [\![\Phi]\!] = [\![\alpha]\!];$$

that is, to show *soundness* and *adequacy* with respect to some existing language $L_{\mathbb{PR}}$ that captures $\mathbb{PR}$. In fact, this is precisely what we will do (see Section 5.3.3). Furthermore, in so doing if we make an appropriate choice of language then we may directly and constructively exploit some existing results that we will require in the development of our stream processing theory.

The language PR is a rigorously formulated functional specification language based on simultaneous primitive recursion that has been used successfully for the specification and verification of hardware as SCAs. Therefore, a formal, semantically sound compiler from PREQ to PR together with another from PR to PREQ will not only establish that PREQ is sound and adequate with respect to the class $\mathbb{PR}$, but it will enable us to use constructively existing theory that has been developed using PR. Specifically, it will enable us to develop a constructive solution to the problem of Cartesian composition that is more conveniently expressed at the level of function schemes.

4.1.3 Chapter Overview

The predominantly technical material in this chapter is devoted to the formal introduction of the language PR and to a discussion of the use of Cartesian forms as a general purpose specification methodology. In particular, this chapter is concerned with the development of the necessary theoretical results for the use of Cartesian form specification in the context of primitive recursive functions. The following chapter deals with the equivalence of PR and our equational specification language PREQ.

Section 4.2 is concerned with the formal definition of the language PR, and the definition of some useful PR computable functions that we will require in later sections.

So that we may demonstrate the generality of our results, in Section 4.3 we discuss using PR (and hence PREQ) as a method for specifying STs and set ourselves the task of identifying the scope and limits of Cartesian form computation.

In order to answer this important question rigorously in Section 4.4 we introduce the languages μPR, λPR and $\lambda\mu$PR and show that μPR provides a general model of Cartesian form stream computation:

Theorem 2. *Let MC be any effective model of computation and let $MC(\underline{A})$ be the class of functions computed by MC over the $\underline{S}$-sorted $\underline{\Sigma}$-algebra $\underline{A}$.*

If $F : \underline{A}^x \to [T \to A]^v \in MC(\underline{A})$, for some $x \in \underline{S}^+$, and for some $v \in S^+$ then there exists a scheme $\alpha_{F^*} \in \mu PR(\underline{\Sigma})_{t\,x,v}$ such that

$$(\forall t \in T)\,(\forall a \in \underline{A}^x) \quad F(a)(t) = [\![\alpha_{F^*}]\!]_{\underline{A}}(t, a);$$

that is, for every computable function F that returns either one or more streams as output there exists a computable function that computes the Cartesian form F^* of F.

In the final section of this chapter (Section 4.5) we address formally the problem of composing CFSTs that we discussed in Section 3.10.4; that is, we show the following:

Theorem 3. *Let G and H be any functions of type*

$$G : [T \to A]^n \to [T \to A]^p$$

and

$$H : [T \to A]^p \to [T \to A]^m$$

respectively for some $m, n, p \in \mathbb{N}^+$ *and let*

$$F : [T \to A]^n \to [T \to A]^m$$

be defined by

$$F = H \circ G.$$

(1) If $G^*, H^* \in PR(\underline{A})$ *then* $F^* \in PR(\underline{A})$.
(2) If $G^*, H^* \in \mu PR(\underline{A})$ *then* $F^* \in \mu PR(\underline{A})$.
(3) Furthermore, in both cases above, given schema α_{G^*} *and* α_{H^*} *representing* G^* *and* H^* *respectively we can effectively construct a scheme* α_{F^*} *representing* F^* *from* α_{G^*} *and* α_{H^*}*; that is, the composition of CFSTs is uniform in* A.

Despite the straightforward nature of the statement of Theorem 3 (that is set in a simplified form as an exercise in Goodstein [1961]) a full constructive proof is surprisingly technical and requires several intermediate results. Indeed, Section A.2 and Section A.3 in Appendix A are given over to the proof of our main technical result (Theorem 9) that we use to prove Theorem 3 (via Theorems 7 and 8).

Finally, notice that the implication of Theorems 2 and 3 is that Cartesian form specification is indeed a general purpose specification technique in that: for every AFST there exists an equivalent CFST; and given any two appropriately typed CFSTs we may compose them to give a single equivalent CFSTs. These facts are precisely what we required to demonstrate that the SCA specification methodology (see Section 3.10) generalizes to provide the basis of general purpose theory of stream processing. We return to this point in the following sections.

4.2 The Abstract Syntax and Semantics of PR

We now introduce formally the language PR that we will use as a convenient mathematical tool to establish the compositional properties of STs in Cartesian form when specified in PREQ.

Given a standard Σ-structure A we can build-up functions (function definitions) from the constants and operations of A using sequential and parallel composition, and primitive recursion. Thus, formal (syntactic) function definitions will use Σ-symbols as ground terms to denote basic functions, and use named function-constructors to build larger terms denoting composite functions. The set of all well-structured syntactic function definitions is denoted PR(Σ), and a member of PR(Σ) may be thought of as a program in a low-level, strongly-typed functional programming language whose semantics is a function on Σ-algebra A.

4.2.1 The Abstract Syntax of PR

The following definition of the language PR is based on the account presented in Thompson [1987] although we include definition-by-cases as a primitive operation rather than a function building tool. This definition is essentially a generalization of *bounded Kleene schemes* that can

be found in (for example) Cutland [1980].

Let Σ be a standard S-sorted signature. We define

$$\mathrm{PR}(\Sigma) = < \mathrm{PR}(\Sigma)_{u,v} \mid u, v \in S^+ >$$

wherein each set $\mathrm{PR}(\Sigma)_{u,v}$ of *schema of type* (u, v) is defined uniformly in u and v by induction as follows:

Basis Schema.

(1) **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 $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$.

(2) **Algebraic Operations.** If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$ then $\alpha \in \mathrm{PR}(\Sigma)_{w,s}$.

(3) **Projection Functions.** If $\alpha = \mathrm{U}_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$ then $\alpha \in \mathrm{PR}(\Sigma)_{w,w_i}$.

(4) **Definition-by-Cases.** If $\alpha = dc_s$ for some $s \in S$ then $\alpha \in \mathrm{PR}(\Sigma)_{\mathbf{b}\,s\,s,s}$.

Induction: Function Building Tools.

(5) **Vectorisation.** If $\alpha =< \alpha_1, \ldots, \alpha_m >$ wherein $m > 0$ and for $i = 1, \ldots, m$, $\alpha_i \in \mathrm{PR}(\Sigma)_{u,s_i}$ for some $s_i \in S$ then $\alpha \in \mathrm{PR}(\Sigma)_{u,s_1 \cdots s_m}$.

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

(7) **Primitive Recursion.** If $\alpha = *(\alpha_1, \alpha_2)$ where $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\mathbf{n}\,u\,v,v}$ for some $u, v \in S^+$ then $\alpha \in \mathrm{PR}(\Sigma)_{\mathbf{n}u,v}$.

4.2.2 The Semantics of PR

Let A be a standard Σ-algebra. For each $\alpha \in \mathrm{PR}(\Sigma)$ the *meaning of* α over A is $[\![\alpha]\!]_A$ where $[\![.]\!]_A$ is the $S^+ \times S^+$-indexed family

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

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

Basis Schema.

(1) **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 $[\![\alpha]\!]_A : A^w \to A_s$ is defined by

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

(2) Algebraic Operations. If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$, then $[\![\alpha]\!]_A : A^w \to A_s$ is defined by

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

(3) Projection Functions. If $\alpha = U_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$, then $[\![\alpha]\!]_A : A^w \to A_{w_i}$ is defined by

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

(4) Definition-by-Cases. If $\alpha = dc_s$ for some $s \in S$ then $[\![\alpha]\!]_A : A^{\mathbb{b}\,s\,s} \to A_s$ is defined by

$$(\forall b \in \mathbb{B})\ (\forall a_1, a_2 \in A_s) \quad [\![\alpha]\!]_A(b, a_1, a_2) = \begin{cases} a_1 & \text{if } b = tt \\ a_2 & \text{if } b = ff. \end{cases}$$

Induction: Function Building Tools.

(5) Vectorisation. If $\alpha = < \alpha_1, \ldots, \alpha_m >$ for $i = 1, \ldots, m > 0$, $\alpha_i \in \mathrm{PR}(\Sigma)_{u,s_i}$ for some $s_i \in S$, then $[\![\alpha]\!]_A : A^u \to A^{s_1 \cdots s_m}$ is defined by

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

(6) Composition. If $\alpha = \alpha_2 \circ \alpha_1$ where $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$ for some $u, v, w \in S^+$, then $[\![\alpha]\!]_A : A^u \to A^v$ is defined by

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

(7) Primitive Recursion. If $\alpha = *(\alpha_1, \alpha_2)$ where $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\mathbb{n}\,u\,v,v}$ for some $u, v \in S^+$, then $[\![\alpha]\!]_A : T \times A^u \to A^v$ is defined by

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

and

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

4.2.3 Notes

(1) Formally we require a proof to justify the existence of functions defined by Clause (4.2.1) of the preceding definition. The interested reader can consult Tucker and Zucker [1988] for such a proof. Also, notice that Clause (4.2.1) allows us to capture the class of *simultaneous* primitive recursive functions that is strictly larger than the class of primitive recursive functions over an arbitrary abstract standard algebra. However, in this thesis the simultaneity of the function definitions does not play a central role and hence we will simply refer to this class as the primitive recursive functions.

(2) There is a sense in which the Boolean type and its standard operations (see Section 2.3.5)

are not necessary in a standard algebra as they can be readily coded using the natural numbers and primitive recursion (see Chapter 8 of Thompson [1987]). However, from the perspective of efficiency of specification, definition-by-cases is useful as a primitive and will therefore be included (see Section 6.7.1). Although, as it is included for convenience and not as a mathematical necessity, we will omit definition-by-cases in the definitions of the formal compilers in the following sections and leave the construction of the appropriate schemes and the proof of their correctness to the reader.

To complete the functional formalization of the primitive recursive functions on a Σ-algebra A we make the following definition:

Definition 10. Let A be a standard Σ-algebra. We define $\mathrm{PR}(A)$ the class of *primitive recursive functions* by

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

4.2.4 Further Preliminaries

In this section we define further notation and functions either specific to this chapter or that we will be used throughout the rest of this thesis. In particular, with respect to this chapter we introduce operations on S^* that will be used for replacing individual sorts by words. The reader not interested in the details of how we establish the compositionality of Cartesian forms can omit these definitions (Definition 13 onward) and move directly to Section 4.3.

Throughout the rest of this chapter unless specifically stated otherwise Σ will denote any standard S-sorted signature and A will denote any standard S-sorted Σ-algebra. However, sometimes for emphasis we will re-state this assumption. In addition X will always denote any S-sorted collection of variable symbols. Moreover, to avoid any confusion of symbols we will always assume that Σ and X pairwise disjoint, and in addition that neither contain the distinguished symbol f nor any of the distinguished symbols from the set $\{f_{n,n'} \mid n, n' \in \mathbb{N}\}$.

Notation 1. Given a vector $a \in A^w$ for some $w \in S^+$ it is natural to write $a = (a_1, \ldots, a_{|w|})$ to name the individual components of a. We will use a similar notation for the individual components of a vector-valued function: if function f has functionality $f : A^u \to A^v$ for some $u, v \in S^+$ we write '$f = (f_1, \ldots, f_{|v|}) : A^u \to A^v$' to mean that $f_i : A^u \to A_{v_i}$ for $i = 1, \ldots, |v|$ are those functions such that for each $a \in A^u$

$$f(a) = (f_1(a), \ldots, f_{|v|}(a)).$$

We call $f_1, \ldots, f_{|v|}$ the *co-ordinate functions of* f.

Finally, if $f_A : A^u \to A_s$, for some $u \in S^+$, and for some $s \in S$ is some function defined using the operations of A then we use A_f to denote the extension of algebra A that includes f_A as a basic function wherein A_f is an S-sorted Σ_f-algebra such that $\Sigma_f = \Sigma \cup \{f\}$. For convenience if $f_A : A^u \to A^v$, for some $u, v \in S^+$ is a vector-valued function then we also write A_f to denote A extended with f_A as a basic operation noting that formally this is an abbreviation for the algebra $(((A_{f_1})_{f_2}) \cdots)_{f_{|v|}}$ wherein f_i for $i = 1, \ldots, |v|$ are the co-ordinate functions of f.

Definition 11. Given some scheme $\alpha \in \mathrm{PR}(\Sigma)$ in the sequel we will write $\beta \subseteq \alpha$ to indicate that β is a *sub-scheme* of α; that is, a scheme from which α has been constructed by zero or more applications of the function building tools of PR. Furthermore, when $\beta \subseteq \alpha$ and $\beta \neq \alpha$; that is, when β is a *proper sub-scheme* of α we will write $\beta \subset \alpha$.

Definition 12. For each $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$, for some $u, v \in S^+$, we refer to u and v as being the *domain* and *range* of α respectively and we write $\mathrm{dom}(\alpha)$ to mean u and $\mathrm{ran}(\alpha)$ to mean v.

Operations on Strings

Definition 13. For any $w \in S^+$, for any $z \in S^*$ and for any $i \in \{1, \ldots |w|\}$ we define

$$w\{i/z\} \in S^{|w|+|z|-1}$$

by

$$w\{i/z\} = w_1 \cdots w_{i-1}\, z\, w_{i+1} \cdots w_{|w|};$$

that is,

$$w\{i/z\} = w_1 \cdots w_{i-1}\, z_1 \cdots z_{|z|}\, w_{i+1} \cdots w_{|w|}.$$

Notice that it is possible that $z = \lambda$ in which case $w\{i/z\}$ is w with w_i deleted.

Finally, for later convenience, for any $w \in S^+$, for any $z \in S^*$ and for any $i \notin \{1, \ldots, |w|\}$ we define $w\{i/z\} \in S^{|w|}$ by $w\{i/z\} = w$.

Example 4. If $S = \{\mathbf{n}, \mathbf{t}, \mathbf{b}, a, b, c\}$, $w = c\, c\, \mathbf{b}\, b\, a\, b$ and $z = \mathbf{n}\, \mathbf{b}$ then $w\{4/z\} = c\, c\, \mathbf{b}\, \mathbf{n}\, \mathbf{b}\, a\, b$.

Notation 2. Given $w \in S^+$, $i \in \{1, \ldots, |w|\}$ and $z \in S^*$ a typical member x of $A^{w\{i/z\}}$ is a vector

$$x = (a_1, \ldots, a_{i-1}, b_1, \ldots, b_{|z|}, a_{i+1}, \ldots, a_{|w|})$$

wherein $a_j \in A_{w_j}$ for $j = 1, \ldots, i-1, i+1, \ldots, |w|$ and $b_k \in A_{z_k}$ for $k = 1, \ldots, |z|$. Of course we can quantify over all such elements by writing "$\forall x \in A^{w\{i/z\}}$", but in practice we will want to name the elements of $A_{z_1}, \ldots, A_{z_{|z|}}$ that occur in x. To do this we can write

$$\forall x = (a_1, \ldots, a_{i-1}, b_1, \ldots, b_{|z|}, a_{i+1}, \ldots, a_{|w|}) \in A^{w\{i/z\}}$$

or just

$$\forall x = (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_{|w|}) \in A^{w\{i/z\}}$$

provided it is clear that b is a vector of $|z|$ elements. However, this notation is too cumbersome for our needs. For this reason we write "$a\{i/b\}$" for a typical element of $A^{w\{i/z\}}$; that is, $a\{i/b\}$ denotes a vector $(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_{|w|})$ for some $a_1 \in A_{w_1}, \ldots, a_{i-1} \in A_{w_{i-1}}, a_{i+1} \in A_{w_{i+1}}, \ldots, a_{|w|} \in A_{w_{|w|}}$ and some $b \in A^z$, and we use

$$(\forall a\{i/b\} \in A^{w\{i/z\}})$$

to mean

$$(\forall a_1 \in A_{w_1}) \cdots (\forall a_{i-1} \in A_{w_{i-1}})\, (\forall b \in A^z)\, (\forall a_{i+1} \in A_{w_{i+1}}) \cdots (\forall a_{|w|} \in A_{w_{|w|}}).$$

For example, in the context of Example 4,

$$(\forall a\{4/b\} \in A^{w\{4/z\}})$$

means

$$(\forall a_1 \in A_c)(\forall a_2 \in A_c)(\forall a_3 \in A_{\mathbf{b}})(\forall b \in A^{\mathbf{n}\,\mathbf{b}})(\forall a_4 \in A_a)(\forall a_5 \in A_b);$$

that is,

$$(\forall a_1 \in A_c)(\forall a_2 \in A_c)(\forall a_3 \in A_{\mathbf{b}})(\forall b_1 \in A_{\mathbf{n}})(\forall b_2 \in A_{\mathbf{b}})(\forall a_4 \in A_a)(\forall a_5 \in A_b).$$

Definition 14. Let S be any sort set. For any $s \in S$, and for any $w \in S^+$, we define $I^{s,w} \subseteq \{1, \ldots, |w|\}$ by

$$(\forall j \in \{1, \ldots, |w|\}) \quad j \in I^{s,w} \iff w_j = s.$$

Thus, given a sort $s \in S$, and a word $w \in S^+$, $I^{s,w}$ tells us which elements (if any) of w are exactly s. The size of $I^{s,w}$ is denoted by $|I^{s,w}|$.

Example 5. If w is defined as in Example 4 then $I^{\mathbf{b},w} = \{4, 6\}$.

The remainder of this section is devoted to an extremely technical definition of a function *Init* whose purpose will not be clear until Section 4.5.4. Therefore, we suggest the reader omit this material until that time.

Definition 15. For any $s \in S$, and for any $w \in S^+$, we define $\lambda_1^{s,w} : \{1, \ldots, |w|\} \to \{0, \ldots, |I^{s,w}|\}$ by

$$(\forall n \in \{1, \ldots, |w|\}) \quad \lambda_1^{s,w}(n) = \begin{cases} 0 & \text{if } w_n \neq s; \\ \lambda_0^{s,w}(n) & \text{otherwise,} \end{cases}$$

wherein $\lambda_0^{s,w} : \{1, \ldots, |w|\} \to \{0, \ldots, |I^{s,w}|\}$ is defined uniformly in w by

$$\begin{aligned} (\forall n \in \{1, \ldots, |w|\}) \quad \lambda_0^{s,\lambda}(n) &= 0; \\ \lambda_0^{s,w}(0) &= 0; \\ (\forall s' \in S)(\forall n \in \{1, \ldots, |w| - 1\}) \quad \lambda_0^{s,s'w}(n+1) &= \begin{cases} 1 + \lambda_0^{s,w}(n) & \text{if } s' = s; \\ \lambda_0^{s,w}(n) & \text{otherwise.} \end{cases} \end{aligned}$$

Thus, given a sort $s \in S$, a word $w \in S^+$, and a number n such that $1 \leq n \leq |w|$, $\lambda_1^{s,w}(n) = m > 0$ if and only if $w_n = s$ and w_n is the mth occurrence of s in w (reading left-to-right). If $\lambda_1^{s,w}(n) = 0$ then $w_n \neq s$.

Example 6. Let S be defined as in Example 4. If $w = \underline{a}\,\underline{\mathbf{b}}\,c\,\underline{a}\,c\,\underline{\mathbf{b}}\,\underline{a} \in \underline{S}$ then

(1) $\lambda_1^{\mathbf{b},w}(6) = 2$,

(2) $\lambda_1^{a,w}(7) = 3$, and

(3) $\lambda_1^{c,w}(5) = 2$.

Definition 16. For any $s \in S$, and for any $w \in S^+$, we define $\lambda_2^{s,w} : \{0,\ldots,|I^{s,w}|\} \to \{1,\ldots,|w|\}$ uniformly in w by

$$\begin{aligned}
(\forall m \in \{0,\ldots,|I^{s,w}|\}) \quad \lambda_2^{s,\lambda}(m) &= 0;\\
\lambda_2^{s,w}(0) &= 0;\\
(\forall s' \in S)(\forall m \in \{1,\ldots,|I^{s,w}|-1\}) \quad \lambda_2^{s,s'w}(m+1) &= \begin{cases} 1+\lambda_2^{s,w}(m) & \text{if } s' = s;\\ 1+\lambda_2^{s,w}(m+1) & \text{otherwise.}\end{cases}
\end{aligned}$$

Thus, given a sort $s \in S$, a word $w \in S^+$, and a number m such that $1 \le m \le |I^{s,w}|$, $\lambda_2^{s,w}(m) = n$ if and only if w_n is the mth occurrence of s in w (reading left-to-right), otherwise $\lambda_2^{s,w}(m) = 0$. (Notice $\lambda_1^{s,w}$ and $\lambda_2^{s,w}$ are essentially inverses; see Lemma 2).

Example 7. If $\underline{S}$ and w are defined as in Example 6 then

(1) $\lambda_2^{\underline{b},w}(2) = 6$,

(2) $\lambda_2^{c,w}(1) = 3$, and

(3) $\lambda_2^{\underline{a},w}(1) = 1$.

Lemma 1. *For any $s \in S$ and for any $w \in S$, if $I^{s,w} \neq \varnothing$ then*

$$I^{s,w} = \{\lambda_2^{s,w}(1),\ldots,\lambda_2^{s,w}(|I^{s,w}|)\}.$$

□

Lemma 2. *For any $s \in S$, and for any $w \in S^+$,*

(1) $(\forall n \in I^{s,w}) \quad \lambda_2^{s,w}(\lambda_1^{s,w}(n)) = n$, *and*

(2) $(\forall m \in \{1,\ldots,|I^{s,w}|\}) \quad \lambda_1^{s,w}(\lambda_2^{s,w}(m)) = m$.

□

Definition 17. Let S be any standard sort set. For any $s \in S$, we define $\pi^s : S^+ \to S^*$ by

$$(\forall w \in S^+) \quad \pi^s(w) = \underbrace{s\cdots s}_{|I^{s,w}| \text{ times}}.$$

Thus, given a sort $s \in S$ and a word $w \in S^+$, $\pi^s(w)$ is a word w', of length $|I^{s,w}|$, where for $i = 1,\ldots,|I^{s,w}|$, $w'_i = s$. This $\pi^s(w)$ is used to type $\Pi^{s,w}$ in Definition 18.

Definition 18. Let A be any standard S-sorted Σ-algebra. For any $s \in S$, and for any $w \in S^+$ such that $I^{s,w} \neq \varnothing$, we define $\Pi^{s,w} : \underline{A}^w \to \underline{A}^{\pi^s(w)}$ by

$$(\forall a \in \underline{A}^w) \quad \Pi^{s,w}(a) = (a_{\lambda_2^{s,w}(1)}, \ldots, a_{\lambda_2^{s,w}(|I^{s,w}|)}).$$

Thus given a sort $s \in S$, a word $w \in S^+$, an S-sorted algebra $\underline{A}$, and a vector $a \in \underline{A}^w$, $\Pi^{s,w}$ selects all elements from vector a that are of sort s.

(Notice that the co-domain of $\Pi^{s,w}$ is given by $\pi^s(w)$, and the indices of the co-ordinates of a that are selected by $\Pi^{s,w}$, are members of the set $I^{s,w}$; see Lemma 1).

Definition 19. For any $s \in S$, for any $w \in S^+$, and for any $i \in I^{s,w}$, we define

$$\Delta^{s,w,i} : S^* \times S^+ \to S^+$$

by

$$(\forall u \in S^+)\,(\forall z \in S^+) \quad \Delta^{s,w,i}(u,z) = \begin{cases} \lambda & \text{if } u = \lambda; \text{ and} \\ c_1 \cdots c_{|u|} & \text{otherwise,} \end{cases}$$

wherein for $j = 1, \ldots, |u|$,

$$c_j = \begin{cases} u_j & \text{if } j \notin I^{s,u}; \\ \delta^{s,w,i}(z) & \text{otherwise} \end{cases}$$

wherein for any $s \in S$, for any $w \in S^+$, and for any $i \in I^{s,w}$, $\delta^{s,w,i} : S^+ \to S^+$ is defined by

$$(\forall z \in S^+) \quad \delta^{s,w,i}(z) = \mathbf{n}\, w'$$

wherein $w' = \pi^s(w)\{\lambda_1^{s,w}(i)/z\}$.

Given a sort $s \in S$, words $u, w, z \in S^+$, and an $i \in I^{s,w}$, $\Delta^{s,w,i}(u,z)$ is a new word u' that is exactly u except that every occurrence of s is replaced by $\delta^{s,w,i}(z)$. An example of the use of Δ is given at the end of this section.

Definition 20. Let A be any standard S-sorted Σ-algebra. For any $s \in S$, for any $w, z \in S^+$, and for any $i, j \in I^{s,w}$ we define

$$\theta^{s,w,i,z,j} : A^u \times A^v \times A^z \to A^{\delta^{s,w,i}(z)}$$

wherein $u = w_1 \cdots w_{i-1}$ and $v = w_{i+1} \cdots w_{|w|}$, by

$$(\forall a \in A^u)\,(\forall a' \in A^v)\,(\forall b \in A^z) \quad \theta^{s,w,i,z,j}(a,a',b) = (\lambda_1^{s,w}(j), \Pi^{s,u}(a), b, \Pi^{s,v}(a')).$$

An example of the use of θ is given at the end of this section.

Definition 21. Let S be any sort set. For any $w, u \in S^+$ we write $w \supseteq u$ if, and only if $|I^{u_i,w}| \geq |I^{u_i,u}|$ for $i = 1, \ldots, |u|$; that is, $w \supseteq u$ when every sort w_i of w occurs at least as many times in w as it does in u.

Example 8. If $w = a\,b\,c\,d\,a$, $u = d\,c\,b\,a$ and $v = a\,a$ then $w \supseteq u$ and $w \supseteq v$, but $u \not\supseteq v$.

Definition 22. Let S be any sort set and let $w, u \in S^+$ such that $w \supseteq u$. If $\phi : \{1, \ldots, |u|\} \rightarrow \{1, \ldots, |w|\}$ is an injection such that $u_i = w_{\phi(i)}$ for $i = 1, \ldots, |u|$ then we say that (w, u, ϕ) is a *w/u-replacement.* We will often write ϕ for (w, u, ϕ) if w and u are understood or unimportant.

For each w/u-replacement ϕ we define $\overline{\phi} : \{1, \ldots, |w|\} \rightarrow \{0, \ldots, |u|\}$ by

$$(\forall i \in \{1, \ldots, |w|\}) \quad \overline{\phi}(i) = \begin{cases} 0 & \text{if } \sim (\exists j)\, \phi(j) = i; \text{ and} \\ k & \text{if } \phi(k) = i. \end{cases}$$

Example 9. If w, u, v are defined as in Example 8 then $\chi = \{1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto 5\}$ is a w/u-permutation and $\psi = \{1 \mapsto 1, 2 \mapsto 5\}$ is a w/v-permutation.

Lemma 3. *Let S be any sort set and let $w, u \in S^+$, and let (w, u, ϕ) be a w/u-permutation.*

$$(\forall i \in Im(\phi)) \quad \phi(\overline{\phi}(i)) = i.$$

Definition 23. Let A be any standard S-sorted Σ algebra. For each w/u-replacement ϕ, for each $s \in S$ such that $I^{s,w} \neq \varnothing$, for each $p \in I^{s,w}$, and for each $z \in S^*$ we define

$$Init^{\phi,z,s,p} : A^{w\{p/z\}} \rightarrow A^{\Delta^{s,w,p}(u,z)}$$

by

$$(\forall a\{p/b\} \in A^{w\{p/z\}}) \quad Init^{\phi,z,s,p}(a\{p/b\}) = (y_1, \ldots, y_{|u|})$$

wherein for $k = 1, \ldots, |u|$

$$y_k = \begin{cases} a_{\phi(k)} & \text{if } k \notin I^{s,u}, \\ \theta^{s,w,p,z,\phi(k)}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{otherwise.} \end{cases}$$

Example 10. Let $S = \{r, s, s', c, d\}$ and let A be any S-sorted algebra. If $w = s\ s'\ c\ s\ r\ s$, $u = s\ r\ s\ s'$, $\phi = \{1 \mapsto 6, 2 \mapsto 5, 3 \mapsto 4, 4 \mapsto 2\}$, $z = d\ d$ and $p = 4$ then

(1)

$$\overline{\phi} = \{1 \mapsto 0, 2 \mapsto 4, 3 \mapsto 0, 4 \mapsto 3, 5 \mapsto 2, 6 \mapsto 1\}$$

and

(2) The function $Init^{\phi,z,s,p}$ has functionality $Init^{\phi,z,s,p} : A^{w\{p/z\}} \rightarrow A^{\Delta^{s,w,p}(u,z)}$ wherein

$$w\{p/z\} = s\ s'\ c\ s\ r\ s\{4/d\ d\} = s\ s'\ c\ d\ d\ r\ s$$

and

$$\begin{aligned} \Delta^{s,w,p}(u,z) &= \Delta^{s,s\ s'\ c\ s\ r\ s,4}(s\ r\ s\ s', d\ d) \\ &= \delta^{s,w,p}(z)\ r\ \delta^{s,w,p}(z)\ s' \\ &= x\ r\ x\ s' \end{aligned}$$

wherein $x = \mathbf{n}\ s\ d\ d\ s$ since

$$\begin{aligned}\delta^{s,w,p}(z) &= \delta^{s,s\,s'\,c\,s\,r\,s,4}(d\,d)\\ &= \mathbf{n}\,\pi^{s}(s\,s'\,c\,s\,r\,s)\{\lambda_1^{s,s\,s'\,c\,s\,r\,s}(4)/d\,d\}\\ &= \mathbf{n}\,s\,s\,s\{2/d\,d\}\\ &= \mathbf{n}\,s\,d\,d\,s;\end{aligned}$$

and

$$(\forall a\{p/b\} \in A^{w\{p/z\}}) \quad Init^{\phi,z,s,p}(a\{p/b\}) = (\theta^{s,w,p,z,6}(a',a'',b), a_5, \theta^{s,w,p,z,4}(a',a'',b), a_2)$$

wherein $a' = a_1, a_2, a_3$ and $a'' = a_5, a_6$. It now follows from the definitions that

$$\theta^{s,w,p,z,6}(a',a'',b) = \theta^{s,s\,s'\,c\,s\,r\,s,4,d\,d,6}(a',a'',b_1,b_2) = (3,a_1,b_1,b_2,a_6);$$

and

$$\theta^{s,w,p,z,4}(a',a'',b) = \theta^{s,s\,s'\,c\,s\,r\,s,4,d\,d,4}(a',a'',b_1,b_2) = (2,a_1,b_1,b_2,a_6);$$

that is,

$$(\forall a\{p/b\} \in A^{w\{p/z\}}) \quad Init^{\phi,z,s,p}(a\{p/b\}) = (3,a_1,b_1,b_2,a_6,a_5,2,a_1,b_1,b_2,a_6,a_2).$$

4.2.5 Simple PR Computable Functions

Each of the following facts states that a certain function $f : A^u \to A^v$ on an S-sorted Σ-algebra A for some $u, v \in S^+$ is PR computable. To prove this fact we must show that there exists a scheme $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ such that $[\![\alpha]\!]_A = f$. However, as for each of the following functions the construction of such a scheme is straightforward they are omitted.

Lemma 4. *For any $n \in \mathbb{N}$ and for any $w \in S^+$, if $Copy_A^{n,w} : A^w \to (A^w)^n$ is defined by*

$$(\forall a \in A^w) \quad Copy_A^{n,w}(a) = (\overbrace{a,\ldots,a}^{n\,times})$$

then $Copy^{n,w} \in PR(A)$.

Lemma 5. *For any $n \in \mathbb{N}$ for any $s \in S$ and for any $1 \le i \le n$ if $Switch_A^{n,s,i} : A^{s^n} \to A_s$ is defined by*

$$(\forall m \in \mathbb{N})\,(\forall a_1,\ldots,a_n \in A_s) \quad Switch_A^{n,s,i}(m,a_1,\ldots,a_n) = \begin{cases} a_m & \text{if } 1 \le m \le n;\\ a_i & \text{otherwise,}\end{cases}$$

then $Switch_A^{n,s,i} \in PR(A)$.

Finally, the function $Init$ as defined in the previous section is $\mathrm{PR}(A)$ computable.

Lemma 6. *For each w/u-replacement ϕ, for each $s \in S$ such that $I^{s,w} \neq \varnothing$, for each $p \in I^{s,w}$, and for each $z \in S^*$*

$$Init^{\phi,z,s,p} \in PR(A)$$

wherein $Init^{\phi,z,s,p}$ is defined as in Definition 23.

4.3 Using PR to Represent Stream Transformers

Recall the definition of a standard stream algebra $\underline{A}$ from Section 2.4.2 and consider the class of functions that may be computed by PR($\underline{A}$). In particular, notice that there are only two types of primitive operations that may take a stream as an argument: *eval*$_s$ for each $s \in S$ - that simply evaluate a stream at a particular clock cycle; and projection functions (from the language PR) - that simply select a stream argument as output. Furthermore, notice that there are no mechanisms such as λ-abstraction (see Section 4.4.2) for making streams from non-stream data. As a consequence, it would appear that PR($\underline{A}$) provides a very weak model of stream computation. In more detail: if $\underline{A}$ is any standard $\underline{S}$-sorted $\underline{\Sigma}$-algebra then any function $F \in \mathrm{PR}(\underline{A})$ of the form

$$F : \underline{A}^x \to [T \to A]^v$$

for some $x \in \underline{S}^+$ and for some $v \in S^+$ must use projection functions to produce the stream output. Thus, x must include stream sorts (otherwise $F \notin \mathrm{PR}(\underline{A})$) and F must simply copy its stream inputs to its stream outputs; that is, F must essentially be of the form

$$F : [T \to A]^u \times A^y \to [T \to A]^v$$

for some $u \in S^+$ and for some $y \in S^*$ and be defined such that for all $a = (a_1, \ldots, a_{|u|}) \in [T \to A]^u$ and for all $b \in A^y$ if $F(a,b) = (a'_1, \ldots, a'_{|v|})$ then for each $i \in \{1, \ldots, |v|\}$ there exists a $j \in \{1, \ldots, |u|\}$ such that $a'_i = a_j$. (Although, notice that this j may depend on a and b.)

The following lemma formalizes these observations:

Lemma 7. *Let $\underline{A}$ be any standard $\underline{S}$-sorted $\underline{\Sigma}$-algebra. If*

$$F = (F_1, \ldots, F_{|v|}) : [T \to A]^u \times A^y \to [T \to A]^v \in PR(\underline{A})$$

for some $u, v \in S^+$ and for some $y \in S^*$ then:
(1) For each $i \in \{1, \ldots, |v|\}$
(A)

$$(\exists j \in \{1, \ldots, |u|\}) \quad u_j = v_i,$$

(B)

$$(\forall a \in [T \to A]^u)\,(\forall b \in A^y)\,(\exists j \in \{1, \ldots, |u|\}) \quad F_i(a,b) = a_j$$

and
(2) For each such F there exists

$$f = (f_1, \ldots, f_{|v|}) : [T \to A]^u \times A^y \to \mathbb{N}^{|v|} \in PR(\underline{A})$$

such that

$$(\forall a \in [T \to A]^u) \quad f_i(a,b) = j \iff F_i(a,b) = a_k.$$

for some $k \in \{1, \ldots, |u|\}$ such that $a_k = a_j$.

Proof. Property (1) follows by induction on the structure of a scheme α such that $[\![\alpha]\!]_{\underline{A}} = F$. Property (2) follows as a Corollary to Theorem 9 (see Section A.2).

□

Lemma 7 appears to place severe restrictions upon the class of STs that we may represent in PR. In particluar, Lemma 7 states that STs in PR may only permute or copy their stream arguments. For example, by Lemma 7 even a simple ST such as

$$F : [T \to A]^2 \to [T \to A]$$

defined by

$$(\forall a_1, a_2 \in [T \to A])\,(\forall t \in T) \quad F(a_1, a_2)(t) = \begin{cases} a_1(t) & \text{if } t = 0, \\ a_2(t) & \text{otherwise} \end{cases}$$

cannot be primitive recursive over $\underline{A}$ since F neither returns exactly a_1 nor exactly a_2.

Despite this fact we will show in the following section that a large and useful class of stream transformers *can* be admitted to PR($\underline{A}$) provided we work with Cartesian forms.

4.4 Cartesian Form Computability

The language ASTRAL that we present in Chapter 6 derives its semantics from the language PREQ. Therefore, as in the following chapter we show formally that PREQ is equivalent to PR in its expressive power, Lemma 7 also appears to place severe restrictions upon the class of STs that may be represented in ASTRAL. In order that we may show formally that ASTRAL *is* a general purpose specification tool in the context of primitive recursive STs, we now discuss the use of Cartesian form specification from the perspective of computability. In particular, we introduce four extensions to the language PR and discuss the classes of STs that these languages can represent.

4.4.1 Cartesian Forms

The informal definition of Cartesian forms from Section 4.1 can be formalized as follows:

Definition 24. Let $f : \underline{A} \to [T \to A]^v$ for any $u \in \underline{S}^*$ and for any $v \in S^+$. We define the *Cartesian form* $f^* : T \times \underline{A}^u \to A^v$ *of* f (also written $cart(f)$) by

$$(\forall t \in T)\,(\forall a \in \underline{A}^u) \quad f^*(t, a) = f(a)(t).$$

Notice that conceptually the Cartesian form of a function f is similar to the *un-Curried form* of f, although in general the un-Curried form of a function and its Cartesian form are not the same. For example, if g is the un-Curried form of f then g has functionality

$$g : \overbrace{T \times \cdots \times T}^{|v| \text{ times}} \times \underline{A}^u \to A^v$$

as we must un-Curry each co-ordinate of f's stream output individually.

It is also appropriate at this point to give a formal definition of the dual concept to that of a Cartesian form; that is, the applicative form, and two simple lemmata.

Definition 25. Let $g : T \times \underline{A}^u \to A^v$ for some $u \in \underline{S}^*$ and for some $v \in S^+$. We define the *applicative form* $\widehat{g} : \underline{A}^u \to [T \to A]^v$ *of* g (also written $app(g)$) by

$$(\forall a \in \underline{A}^u)\,(\forall t \in T) \quad \widehat{g}(a)(t) = g(t, a).$$

Lemma 8. *Let f and g be as above.*

(1) $app(cart(f)) = f$.

(2) $cart(app(g)) = g$.

Lemma 9. *If $f = (f_1, \ldots, f_n) : T \times \underline{A}^u \to A^v$, for some $u \in \underline{S}^*$ and for some $v \in S^+$ then*

$$\widehat{f} = (\widehat{f_1}, \ldots, \widehat{f_n}).$$

Notice that the Cartesian form of F as defined in the previous section is a function $F^* : T \times [T \to A]^2 \to A$ and can be defined by

$$(\forall t \in T)\,(\forall a_1, a_2 \in [T \to A]) \quad F^*(t, a_1, a_2) = \begin{cases} a_1(t) & \text{if } t = 0, \\ a_2(t) & \text{otherwise.} \end{cases}$$

Also notice that while $F \notin \mathrm{PR}(\underline{A})$ the Cartesian form of F is readily seen to be a member of $\mathrm{PR}(\underline{A})$ and therefore it would appear that at least informally that some useful STs may be specified in PR in Cartesian form. However, as we are developing the basis of a stream processing theory using Cartesian forms as a specification methodology it is important to be precise about the class of functions that may be specified in Cartesian form; that is, (1) what are the scope and limits of Cartesian form computation in the context of stream algebras?; and more specifically (2) what are the scope and limits of Cartesian form computation in the context of $\mathrm{PR}(\underline{A})$? In order that that we may answer these questions rigorously in the following sections we introduce three extension to the language PR.

4.4.2 μPR

It is a well-known result that the class of primitive recursive functions do not provide a general model of computation in the sense of the Church-Turing Thesis (see for example Cutland [1980]). However, the language μPR is a generalization of PR (see Tucker and Zucker [1988]), that includes Kleene's least number search operator, and in the context of algebras with lists containing only finite or countable infinite carriers does provide a general model of computation. We can define μPR by extending PR with the additional induction clause:

(8a) Minimalization. If $\alpha = \mu(\alpha')$ wherein $\alpha' \in \mu\mathrm{PR}(\Sigma)_{\mathbf{n}\,u,\mathbf{b}}$ for some $u \in S^*$ then $\alpha \in \mu\mathrm{PR}(\Sigma)_{u,\mathbf{n}}$.

The semantics of this additional clause is formalized as follows:

(8a) Minimalization. If $\alpha = \mu(\alpha')$ wherein $\alpha' \in \mu\mathrm{PR}(\Sigma)_{\mathbf{n}\,u,\mathbf{b}}$ for some $u \in S^*$ then $[\![\alpha]\!]_A : A^u \to \mathbb{N}$ is defined by

$$(\forall a \in A^u) \quad [\![\alpha]\!]_A(a) = \mathit{least}\, n.([\![\alpha']\!]_A(n, a) = tt).$$

The class of functions computed by $\mu\mathrm{PR}(\Sigma)$ is formalized in the usual way:

Definition 26. Let A be any standard Σ-algebra. We define $\mu\mathrm{PR}(\Sigma)$ the class of *primitive recursive functions with least number search over* A by

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

Notice that similarly to the language PR the formulation of μPR is essentially a generalization of *Kleene schemes* (also see Cutland [1980]).

4.4.3 λPR

One operation that is often (implicitly) included in functional languages is λ-abstraction. Indeed, languages such as ML (see for example Milner [1984] and Wilkström [1987]) based on the λ-calculus often have this facility. The class of functions that can be computed when PR is extended with a limited form of λ-abstraction, referred to as 'stream abstraction', has been considered in Tucker and Zucker [1992]. The language λPR is PR extended with an additional induction clause as follows:

(8b) Stream Abstraction. If $\alpha = \lambda(\alpha')$ wherein $\alpha' \in \lambda\mathrm{PR}(\underline{\Sigma})_{\mathbf{n}\,u,v}$ for some $u \in S^*$ and for some $v \in S^+$ then $\alpha \in \lambda\mathrm{PR}(\underline{\Sigma})_{u,\underline{v}}$. (Note that if $u = \lambda$ then the type of $\lambda(\alpha')$ is $\underline{u}$ by definition.)

The semantics of this additional clause is formalized as follows:

(8b) Stream Abstraction. If $\alpha = \lambda(\alpha')$ wherein $\alpha' \in \lambda\mathrm{PR}(\underline{\Sigma})_{\mathbf{n}\,u,v}$ for some $u \in S^*$ and for some $v \in S^+$ then $[\![\alpha]\!]_A : A^u \to [T \to A]^v$ is defined by

$$(\forall a \in A^u) \quad [\![\alpha]\!]_A(a) = \lambda.n([\![\alpha']\!]_A(n, a)).$$

Again the class of functions computed by $\lambda\mathrm{PR}(\underline{\Sigma})$ is formalized in the usual way:

Definition 27. Let $\underline{A}$ be any standard $\underline{\Sigma}$-algebra. We define $\lambda\mathrm{PR}(\underline{A})$ the class of *primitive recursive functions with stream abstraction over* $\underline{A}$ by

$$\lambda\mathrm{PR}(\underline{A}) = \{[\![\alpha]\!]_{\underline{A}} \mid \alpha \in \lambda\mathrm{PR}(\underline{\Sigma})\}.$$

4.4.4 $\lambda\mu$PR

Having defined $\mu\mathrm{PR}(\underline{\Sigma})$ and $\lambda\mathrm{PR}(\underline{\Sigma})$ one further obvious extension to PR, that is also considered in Tucker and Zucker [1992], is to define $\lambda\mu\mathrm{PR}(\underline{\Sigma})$; that is, PR with both additional induction clauses ((8a) and (8b)). As before the class of functions computed by $\lambda\mu\mathrm{PR}(\underline{\Sigma})$ is formalized as follows:

Definition 28. Let A be any standard Σ-algebra. We define $\lambda\mu\mathrm{PR}(\underline{A})$ the class of *primitive recursive functions with stream abstraction and least number search over* A by

$$\lambda\mu\mathrm{PR}(\underline{A}) = \{[\![\alpha]\!]_{\underline{A}} \mid \alpha \in \lambda\mu\mathrm{PR}(\underline{\Sigma})\}.$$

4.4.5 The Scope and Limits of Cartesian Form Computability

The formalization of the three extensions to the language PR above enables us to characterize precisely the nature of Cartesian form computability by drawing on two results from Tucker and Zucker [1992] and Tucker and Zucker [1994].

Theorem 4. *Let MC be any effective model of computation and let* $MC(\underline{A})$ *be the class of functions computed by MC over the* $\underline{S}$*-sorted algebra* $\underline{A}$.

If $F : \underline{A}^x \to [T \to A]^v \in MC(\underline{A})$, *for some* $x \in \underline{S}^+$ *and for some* $v \in S^+$ *then there exists a scheme* $\alpha_F \in \lambda\mu PR(\underline{\Sigma})_{x,\underline{v}}$ *such that*

$$F = [\![\alpha_F]\!]_{\underline{A}}.$$

Theorem 5. *If* $F : \underline{A}^x \to [T \to A]^v \in \lambda\mu PR(\underline{A})$, *for some* $x \in \underline{S}^+$ *and for some* $v \in S^+$ *then there exists a scheme* $\alpha_{F^\bullet} \in \mu PR(\underline{\Sigma})_{t\,x,v}$ *such that*

$$(\forall t \in T)\,(\forall a \in \underline{A}^x) \quad F(a)(t) = [\![\alpha_{F^\bullet}]\!]_{\underline{A}}(t, a).$$

Notice that clearly Theorem 2 (Page 88) follows as a simply corollary to Theorems 4 and 5 and they provide a precise answer (and from our perspective positive answer) to the first part of our question concerning the scope and limits of Cartesian form computation that we set at the end of Section 4.4.1: that from the perspective of effective computability Cartesian form specification is a general purpose technique. However, what does this result tell us about the second question concerning the scope and limits of Cartesian from computability in $\mathrm{PR}(\underline{A})$ and in particular the computability of STs in ASTRAL?

In fact the answer to this second question follows from an examination of the proof of Theorem 5 that is by induction on the structural complexity of the scheme α_F such that

$$[\![\alpha_F]\!]_{\underline{A}} = F.$$

In particular, we can observe from this proof that the use of least number search does not play a significant role other than in the obvious sense of increasing the class of functions that may

be computed; that is, as a straightforward corollary to the proof of Theorem 5 we have the following:

Corollary 1. *If* $F : \underline{A}^x \to [T \to A]^v \in PR(\underline{A})$ *for some* $x \in \underline{S}^+$ *and for some* $v \in S^+$ *then there exists a scheme* $\alpha_{F^\bullet} \in PR(\underline{\Sigma})_{\mathbf{t}\,x,v}$ *such that*

$$(\forall t \in T)\,(\forall a \in \underline{A}^x) \quad F(a)(t) = [\![\alpha_{F^\bullet}]\!]_{\underline{A}}(t, a);$$

that is, Cartesian form computability is also a general purpose technique in the context of primitive recursive STs.

Indeed, as by definition PR captures the class $\mathbb{PR}$ we may infer from this result that the class of *all* primitive recursive CFSTs can be specified in Cartesian form in PR. More specifically, every SCA (see Section 3.10) can be specified in PR by means of CFSTs. Therefore, as PREQ is equivalent to PR (Theorem 10) the language ASTRAL is also an appropriate and general purpose tool for the specification of SCAs and hence hardware.

4.4.6 The Role of μPR in this Thesis

As Corollary 1 shows that the explicit inclusion of least number search plays no significant role in the relationship between STs and the formulation of an equivalent Cartesian form, for generality, in the following sections we will formulate our proofs concerning Cartesian composition in terms of μPR computability. In particular, Theorems 7, 8 and 9 are concerned with μPR computability and in a similar fashion to the method that we deduced Corollary 1 from Theorem 5 we will show that the composition of primitive recursive STs (Part (1) of Theorem 3) follows directly from Theorem 7.

One slight complication that arises from the use of $\mu\mathrm{PR}(\Sigma)$ in our main theorems is that we are now dealing with partial and not total functions. As such, formally in order to demonstrate that two functions are equivalent we must show *Kleene equality* (see for example Cutland [1980]). However, as in the case of Theorems 7, 8 and 9 it is trivial to deduce Kleene equality from a demonstration of the equivalence of the functions under the assumption that they are both defined we leave the details of the completion of our proofs in this respect to the reader.

4.4.7 PR with Cartesian composition as a Primitive

In contrast to the duality of the expressibility of full STs and their Cartesian forms in $\mathrm{PR}(\underline{A})$ and $\lambda\mu\mathrm{PR}(\underline{A})$, Tucker and Zucker [1992] shows that this duality breaks down in the case of $\lambda\mathrm{PR}(\underline{A})$ computable stream transformers. In particular, Tucker and Zucker [1992] demonstrate the following:

Theorem 6. *There exists an* $F : \underline{A}^x \to [T \to A]^v \in \lambda PR(\underline{A})$, *for some* $x \in \underline{S}^+$ *and for some* $v \in S^+$ *such that there does not exist a scheme* $\alpha_{F^\bullet} \in PR(\underline{A})_{\mathbf{t}\,x,v}$ *satisfying*

$$(\forall t \in T)\,(\forall a \in \underline{A}^x) \quad F(a)(t) = [\![\alpha_{F^\bullet}]\!]_{\underline{A}}(t, a).$$

The proof of Theorem 6 is by the counterexample of showing that Ackermann's function is $\lambda\mathrm{PR}(\underline{A})$ computable.

What makes this result even more interesting from the perspective of this thesis is that, from a computability-theoretic perspective, Part (1) of Theorem 3 shows that combining stream abstraction with composition in a single operation *does* maintain the duality of Cartesian form and full ST expressibility in $\mathrm{PR}(\underline{A})$. This point can be clarified by considering a further extension to the language PR.

The language $\widehat{\mathrm{PR}}(\underline{\Sigma})$ is $\mathrm{PR}(\underline{\Sigma})$ extended with an additional induction clause as follows:

(8c) Cartesian Composition. If $\alpha = \hat{\circ}(\alpha_2, \alpha_1)$ where $\alpha_1 \in \widehat{\mathrm{PR}}(\underline{\Sigma})_{t\,z,u}$ and $\alpha_2 \in \widehat{\mathrm{PR}}(\underline{\Sigma})_{\underline{u},v}$ for some $u, v \in S^+$ and for some $z \in \underline{S}^*$ then $\alpha \in \widehat{\mathrm{PR}}(\underline{\Sigma})_{t\,z,v}$.

The semantics of this additional clause is formalized as follows:

(8c) Cartesian Composition. If $\alpha = \hat{\circ}(\alpha_2, \alpha_1)$ where $\alpha_1 \in \widehat{\mathrm{PR}}(\underline{\Sigma})_{t\,z,u}$ and $\alpha_2 \in \widehat{\mathrm{PR}}(\underline{\Sigma})_{\underline{u},v}$ for some $u, v \in S^+$ and for some $z \in \underline{S}^*$ then $[\![\alpha]\!]_{\underline{A}} : T \times \underline{A}^z \to A^v$ is defined by

$$(\forall t \in T)\,(\forall a \in \underline{A}^z) \quad [\![\alpha]\!]_{\underline{A}}(t, a) = [\![\alpha_2]\!]_{\underline{A}}(t, \lambda.n([\![\alpha_1]\!]_{\underline{A}})(n, a)).$$

As with the other three extensions to PR we make the usual formulation of the class of functions computed by $\widehat{\mathrm{PR}}(\underline{\Sigma})$.

Definition 29. Let $\underline{A}$ be any standard $\underline{\Sigma}$-algebra. We define $\widehat{\mathrm{PR}}(\underline{A})$ the class of *primitive recursive functions with Cartesian composition over* A by

$$\widehat{\mathrm{PR}}(\underline{A}) = \{[\![\alpha]\!]_{\underline{A}} \mid \alpha \in \widehat{\mathrm{PR}}(\underline{\Sigma})\}.$$

We can now use Part (1) of Theorem 3 to make a formal statement concerning $\widehat{\mathrm{PR}}(\underline{A})$ computability that is in contrast to Theorem 6 concerning $\lambda\mathrm{PR}(\underline{A})$ computability.

Corollary 2. *Let $\underline{A}$ be any $\underline{\Sigma}$-algebra. If $\alpha' \in \widehat{PR}(\underline{\Sigma})_{u,v}$, for some $u, v \in \underline{S}$ then there exists a scheme $\alpha \in PR(\underline{\Sigma})_{u,v}$ such that*

$$(\forall a \in \underline{A}^u) \quad [\![\alpha']\!]_{\underline{A}}(a) = [\![\alpha]\!]_{\underline{A}}(a);$$

that is,

$$PR(\underline{A}) = \widehat{PR}(\underline{A}).$$

Proof. It is sufficient to show that if

$$\alpha' = \hat{\circ}(\alpha'_2, \alpha'_1)$$

for some $\alpha'_1 \in \mathrm{PR}(\underline{\Sigma})_{u,w}$, for some $\alpha'_2 \in \mathrm{PR}(\underline{\Sigma})_{w,u}$ and for some $w \in \underline{S}^+$ then there exists a scheme $\alpha \in \mathrm{PR}(\underline{\Sigma})$ such that

$$(\forall a \in \underline{A}^u) \quad [\![\alpha']\!]_{\underline{A}}(a) = [\![\alpha]\!]_{\underline{A}}(a).$$

This follows immediately by Part (1) of Theorem 3.

□

Implications of Corollary 2 to this Thesis. Corollary 2 states that adding Cartesian composition to PR; that is, adding a limited form of stream abstraction with composition as a single operation does not increase the class of functions that may be computed. Therefore, we may conclude that it is the specific combination of the primitive recursion operator and stream abstraction that enable us to define functions that are not computable in the absence of stream abstraction on its own. From the perspective of providing a modular approach to the specification of primitive recursive STs this is an important result, as it shows that $\widehat{\text{PR}}$ is the most general expansion of the language PR that we can use. We return to this point in Chapter 6 when via the definition of a specialized compiler we provide what essentially amounts to an implementation of the language of $\widehat{\text{PR}}$ that we use to compile ASTRAL specifications into PREQ.

Having discussed the scope and limits of Cartesian form primitive recursive computability we now move on to the proof of Theorem 3. The reader not interested in the technical details of the proof can now move directly to Chapter 5 on Page 120.

4.5 The Effective Composition of STs in Cartesian Form

Note that before continuing at this point if the reader has not already done so then they should read the comments in Section 4.4.6 concerning the demonstration of the equality of two $\mu\text{PR}(\Sigma)$ computable functions in the following theorems.

4.5.1 Composition of Cartesian Form Stream Transformers in PR

For technical reasons that will become clear later in order to formally prove Theorem 3 it is necessary for us to reformulate and then generalize the theorem in several specific ways. First, we make a modest generalization of Theorem 3 in the form of Theorem 7 that makes explicit the types of the functions G and H and more specifically allows the domains of G and H to contain both stream and non-stream inputs. In particular, Part (2) of Theorem 3 is readily seen to be a special case of Theorem 7. Part (1) of Theorem 3 also follows from Theorem 7, but in addition also requires Lemma 10. Secondly, we state Theorem 8 which is a restricted form of Theorem 7 in the sense that G may not be vector-valued. We can now show that Theorem 7 follows directly from Theorem 8 by the repeated composition of the co-ordinates of a vector-valued functions and hence to prove Theorem 3 we may now concentrate on a proof of Theorem 8. Finally, to prove Theorem 8 we use Theorem 9 that concerns the properties of a formal compiler $\mathbb{C}$. However, in Sections 4.5.5, 4.5.6 and 4.5.7 before we state Theorem 9 we motivate its precise technical form using a high-level algorithmic description of $\mathbb{C}$ and in particular show how Theorem 8 can be considered as a special case of the compiler's formal properties.

4.5.2 Proof of Theorem 3

Definition 30. For any $u^1, u^2 \in \underline{S}^*$, for any $v, y \in S^+$ and for any $z \in \underline{S}^*$ if $u = u^1 \underline{y} u^2$ and h and g have functionality $h : \underline{A}^u \to A^v$ and $g : T \times \underline{A}^z \to A^y$ respectively then we define the *Cartesian composition* $f : \underline{A}^{u^1 z u^2} \to A^v$ *of* h *and* g, in symbols $f = h \,\hat{\circ}\, g$, by

$$(\forall a = (a_1, b, a_2) \in \underline{A}^{u^1 z u^2}) \quad f(a_1, b, a_2) = h(a_1, \hat{g}(b), a_2).$$

We now use Definition 3 to formulate Theorem 7 from which we prove Theorem 3.

Theorem 7. *For any* $u^1, u^2 \in \underline{S}^*$ *for any* $v, y \in S^+$ *and for any* $z \in \underline{S}^*$ *if* $u = u^1 \underline{y} u^2$ *and* $h : \underline{A}^u \to A^v \in \mu PR(\underline{A})$ *and* $g : T \times \underline{A}^z \to A^y \in \mu PR(\underline{A})$ *then* $f = h \,\hat{\circ}\, g \in \mu PR(\underline{A})$.

For convenience we first re-state the theorem:

Theorem 3. *Let* G *and* H *be any functions of type*

$$G : [T \to A]^n \to [T \to A]^p$$

and

$$H : [T \to A]^p \to [T \to A]^m$$

respectively for some $m, n, p \in \mathrm{N}^+$ *and let*

$$F : [T \to A]^n \to [T \to A]^m$$

be defined by

$$F = H \circ G.$$

(1) If $G^*, H^* \in PR(\underline{A})$ *then* $F^* \in PR(\underline{A})$.
(2) If $G^*, H^* \in \mu PR(\underline{A})$ *then* $F^* \in \mu PR(\underline{A})$.
(3) Furthermore, in both cases above given schema α_{G^*} *and* α_{H^*} *representing* G^* *and* H^* *respectively we can effectively construct a scheme* α_{F^*} *representing* F^* *from* α_{G^*} *and* α_{H^*}*; that is, the composition of CFSTs is uniform in* A.

Proof. We prove Part (2) of Theorem 3 and leave the reader to formally deduce Part (1) using Lemma 10. Notice that if we take $\underline{A}^z = [T \to A]^n$, $\underline{A}^u = \underline{A}^y = [T \to A]^p$, $\underline{A}^v = [T \to A]^m$, $g = G^*$ and $h = H^*$ then by Theorem 7 we have $f = H^* \,\hat{\circ}\, G^* = F^* \in \mu\mathrm{PR}(A)$ as required.

□

4.5.3 Proof of Theorem 7

Notice that Theorem 7 concerns the Cartesian composition of a vector-valued function g. Our method of proving Theorem 7 is to repeatedly compose h with the co-ordinate functions of g one at a time. In order to discuss this idea more rigorously we introduce the following notation for composing a single-valued function g at a selected argument position of a function h:

Definition 31. For any $s \in S$, for any $d \in \underline{S}^+$ such that $I^{s,d} \neq \emptyset$, for any $p \in I^{s,d}$, for any $x \in S^+$ and for any $q \in \underline{S}^*$ if $d\{p/q\} = d^1\, q\, d^2$ and h and g have functionality $h : \underline{A}^d \to A^x$ and $g : T \times \underline{A}^q \to A^s$ respectively then we define the *co-ordinate p Cartesian composition* $f : \underline{A}^{d^1\, q\, d^2} \to A^x$ *of h and g*, in symbols $f = h \,\hat{\circ}_p\, g$, by

$$(1) \qquad (\forall a = (a_1, b, a_2) \in \underline{A}^{d^1\, q\, d^2}) \quad f(a_1, b, a_2) = h(a_1, \hat{g}(b), a_2);$$

that is,

$$(\forall a\{p/b\} \in \underline{A}^{d\{p/q\}}) \quad f(a\{p/b\}) = h(a\{p/\hat{g}(b)\}).$$

As with Definition 30 we can use Definition 31 to formulate a further more convenient form of Theorem 3, but now wherein g is single-valued.

Theorem 8. *For any $s \in S$, for any $d \in \underline{S}^+$ such that $I^{s,d} \neq \emptyset$, for any $p \in I^{s,d}$, for any $x \in S^+$ and for any $q \in \underline{S}^*$ if h and g have functionality $h : \underline{A}^d \to A^x$ and $g : T \times \underline{A}^q \to A^s$ respectively then $f = h \,\hat{\circ}_p\, g \in \mu PR(\underline{A})$.*

Discussion. Notice from Equation (1) that f is computed by replacing the pth argument of h with $\hat{g}(b)$. Therefore for any $g = (g_1, \ldots, g_{|y|}) : T \times \underline{A}^z \to A^y$ we can compute $h \,\hat{\circ}\, g$ by repeatedly replacing the arguments of h with the appropriate co-ordinate functions of g; that is, by replacing the pth argument of h with $\hat{g}_p(b)$ for $i = 1, \ldots, |y|$. This is the basic idea behind the following proof of Theorem 7 from Theorem 8. (We note that these semantic ideas are also formalized in the definition of the compiler $\hat{\circ}$ in Section 6.1.1.)

Proof of Theorem 7. By induction on the value $n = |y| \in \mathbf{N}$.
Basis. If $n = 1$ then y is a single sort and hence this follows immediately from Theorem 8 if we take $d = u$, $x = v$, $q = z$, $p = |u^1| + 1$ and $s = y$.
Induction Hypothesis. Assume for any $y' \in S^+$ such that $|y'| = k$ for some fixed $k \geq 1$ and for any $u^3, u^4, z' \in \underline{S}^*$ and for any $v' \in \underline{S}^+$ if $u' = u^3\, \underline{y'}\, u^4$ and $h' : \underline{A}^{u'} \to A^{v'} \in \mu\mathrm{PR}(\underline{A})$ and $g' = (g'_1, \ldots, g'_k) : T \times \underline{A}^{z'} \to A^{y'} \in \mu\mathrm{PR}(\underline{A})$ then $f' = h' \,\hat{\circ}\, g' \in \mu\mathrm{PR}(\underline{A})$.
Induction Step. We must show that for any $y'' \in S^+$ such that $|y''| = k + 1$ and for any $u^5, u^6, z'' \in \underline{S}^*$ and for any $v'' \in \underline{S}^+$ if $u'' = u^5\, \underline{y''}\, u^6$ and $h'' : \underline{A}^{u''} \to A^{v''} \in \mu\mathrm{PR}(\underline{A})$ and $g'' = (g''_1, \ldots, g''_{k+1}) : T \times \underline{A}^{z''} \to A^{y''} \in \mu\mathrm{PR}(\underline{A})$ then $f'' = h'' \,\hat{\circ}\, g'' \in \mu\mathrm{PR}(\underline{A})$.

Notice that by the Induction Hypothesis if we take $u^3 = u^5$, $y' = y''_1 \cdots y''_k$, $u^4 = \underline{y''_{k+1}}\, u^6$, $v' = v''$, $z' = z''$, $h' = h''$, and $g' = G = (g''_1, \ldots, g''_k)$ then there exists

$$f' : \underline{A}^{u^5\, z''\, \underline{y''}\, u^6} \to A^{v''} \in \mu\mathrm{PR}(\underline{A})$$

such that

$$(\forall a = (a_1, b, a_2) \in \underline{A}^{u^5\, z''\, \underline{y''}\, u^6}) \quad f'(a) = h'' \,\hat{\circ}\, G(a).$$

Therefore, by Theorem 8 if we take $d = u^5\, z''\, \underline{y''_{k+1}}\, u^6$, $p = |u^5\, z''| + 1$, $x = v''$, $q = z''$, $h = f'$, and $g = g''_{k+1}$ then there exists

$$f : \underline{A}^{u^5\, z''\, z''\, u^6} \to A^{v''} \in \mu\mathrm{PR}(\underline{A})$$

that satisfies

$$(\forall a = (a_1, b, a_2) \in \underline{A}^{u^5 z'' u^6}) \quad f(a_1, b, b, a_2) = (h'' \,\widehat{\circ}\, G) \,\widehat{\circ}_i\, g''_{k+1}(a_1, b, b, a_2);$$

that is,

$$f(a_1, b, b, a_2) = h''(a_1, \widehat{G}(b), \widehat{g''}_{k+1}(b), a_2)$$

by the definition of $.\,\widehat{\circ}\,.$ and $.\,\widehat{\circ}_i\,.$

$$= h''(a_1, (\widehat{g''_1}, \ldots, \widehat{g''_k})(b), \widehat{g''}_{k+1}(b), a_2)$$

by the definition of G and by Lemma 9

$$= h''(a_1, \widehat{g''}(b), a_2)$$

by the definition of g'' and by Lemma 9

$$= h'' \,\widehat{\circ}\, g''(a_1, b, a_2)$$

by definition.

Consequently if we define $f'' : \underline{A}^{u^5 z'' u^6} \to A^{v''}$ by

$$(\forall a = (a_1, b, a_2) \in \underline{A}^{u^5 z'' u^6}) \quad f''(a_1, b, a_2) = f(a_1, Copy^{2, z''}(b), a_2)$$

then as $f \in \mu\mathrm{PR}(\underline{A})$ by hypothesis and $Copy^{2,z''} \in \mathrm{PR}(\underline{A}) \subset \mu\mathrm{PR}(\underline{A})$ by Lemma 4 we have

$$f'' = h'' \,\widehat{\circ}\, g'' \in \mu\mathrm{PR}(\underline{A})$$

as required.

4.5.4 A Cartesian Composition Compiler $\mathbb{C}$

As we indicated previously to prove Theorem 8 we will use Theorem 9. In particular, Theorem 9 concerns the properties of a compiler $\mathbb{C} : \mu\mathrm{PR}(\underline{\Sigma}) \to \mu\mathrm{PR}(\underline{\Sigma})$. While the definition of $\mathbb{C}$ is based on a straightforward idea, the full proof of Theorem 9 is highly technical and long and is therefore banished to Appendix A.2. Consequently, in the following sections we informally, but rigorously discuss the construction of the compiler and take great care to explain *why* the constructions made are the correct ones. In particular, we will show by means of a high-level algorithm that if g and h are primitive recursive over $\underline{A}$ then so is $f = h\widehat{\circ}_p g$ by showing how to construct a scheme $\alpha_f = \mathbb{C}(\alpha_g, \alpha_h) \in PR(\underline{\Sigma})$ such that $[\![\alpha_f]\!]_{\underline{A}} = f$ for any schemes $\alpha_g, \alpha_h \in PR(\underline{\Sigma})$ such that $[\![\alpha_g]\!]_{\underline{A}} = g$ and $[\![\alpha_h]\!]_{\underline{A}} = h$ respectively. In essence, the basic algorithm underlying the construction of α_f from α_g and α_h is to replace occurrences of $eval_s$ in α_h by α_g in such a way that whenever α_h 'executes' $eval_s$ on arguments (t, a_p), α_f executes α_g on (t, b). We will now explain this idea in some detail based on the account given in Stephens and Thompson [1992].

4.5.5 An Algorithm for Performing Cartesian Composition in PR

Recall Section 4.2.1 where the denotational semantics of a PR scheme α is defined. This definition can be informally viewed as providing an operational semantics in the sense that it defines a virtual machine which 'executes' a PR scheme α on some suitable arguments $a = (a_1, \ldots, a_n)$ by starting with the expression '$[\![\alpha]\!]_{\underline{A}}(a)$', and then repeatedly expanding and simplifying this expression using the appropriate clauses of the definition of $[\![.]\!]_{\underline{A}}$ until no further expansion or simplification is possible. For example, consider what happens when this virtual machine is executed on α_h with arguments $a_1, \ldots, a_n$. In particular, consider what happens to argument a_p as the execution proceeds. Since a_p is a stream of sort $\underline{s}$ and the only operation of $\underline{A}$ that processes such streams is the algebraic operation $eval_{\underline{s}}$, the only stage at which the value of a_p can influence the execution is when a_p is passed to $eval_{\underline{s}}$ together with some natural number t so that the value $[\![eval_{\underline{s}}]\!]_{\underline{A}}(t, a_p) = a_p(t)$ is computed (and is presumably used in some subsequent computation).

Now imagine a second machine executing α_h on the same arguments $a = (a_1, \ldots, a_n)$, but in the case of this second machine we first compute $\hat{g}(b)$ (for some $b \in \underline{A}^{\underline{s}}$) and substitute this for a_p. (Notice that in general this preliminary computation of $\hat{g}(b)$ must be done externally since if $\hat{g}$ is a non-trivial ST then $\hat{g}$ cannot be primitive recursive by Lemma 7.)

If the two machines are executed on their arguments in parallel, then the machines will perform identical computations at the same times until the first time that the pth argument is used. At this time, the first machine is about to compute

$$[\![eval]\!]_{\underline{A}}(t, a_p) = a_p(t)$$

whereas the second machine is about to compute

$$[\![eval_{\underline{s}}]\!]_{\underline{A}}(t, \hat{g}(b)) = \hat{g}(b)(t).$$

However, $\hat{g}(b)(t) = g(t, b) = [\![\alpha_g]\!]_{\underline{A}}(t, b)$. Thus, if we could modify the *first* machine so that at this critical stage in the execution sequence it executes α_g on arguments t and b instead of $eval_{\underline{s}}$ on t and a_p, then both machines will compute the same value ($g(t, b)$) at this point in the execution sequence.

In essence then, we have forced the first machine to simulate the second by replacing an occurrence of $eval_{\underline{s}}$ in α_h with α_g. If we can consistently modify α_h so that whenever it would normally execute $eval_{\underline{s}}$ on some t' and the special stream a_p we replace this with an execution of α_g on t' and b, then both machines will compute the same result: $f(a\{p/b\})$. The difference between the two computations is that the first machine computes $f(a\{p/b\})$ using only primitive recursive computations. This suggests that f is indeed primitive recursive, and the technique of substituting α_g for relevant occurrences of $eval_{\underline{s}}$ is at least the basis of an effective procedure for performing Cartesian composition.

4.5.6 The Effectiveness of our Algorithm: Code Vectors

The difficulty with the above procedure is that we need an effective mechanism for determining whether a stream argument to $eval_{\underline{s}}$ is the special stream a_p or not. However, in general this

test is impossible to perform effectively since streams are infinite objects and hence equality on $[T \to A_s]$ is only co-semi-decidable. In order to describe how we avoid this problem it will be helpful if we first establish some notation.

Recall that h is a function of n arguments $a = (a_1, \ldots, a_n)$ of which the streams of sorts s are $a_{i_1}, \ldots, a_{i_m}$. Thus, for $k = 1, \ldots, n$, if $a_k \in [T \to A_s]$ then there exists some $j = j(k) \in \{1, \ldots, m\}$ such that $a_k = a_{i_{j(k)}}$. In particular, there must be some $r = j(p) \in \{1, \ldots, m\}$ such that $a_p = a_{i_r}$.

Now consider executing α_h on $a = (a_1, \ldots, a_n)$ again. At any stage in the calculation where $eval_s$ is to be executed on some stream, it must be the case that the stream in question is a_{i_j} for some $j \in \{1, \ldots, m\}$ by part (a) of Lemma 7. However, $a_{i_j} = a_p$ when $i_j = p$ (or, equivalently, when $j = r$). Thus, if we can modify α_h so that it computes with the indices of streams rather than the streams themselves, then we can effectively test whether or not a stream is a_p by comparing the indices of the two streams in question.

We use the indices of streams to compute f by executing a new scheme α'_h obtained from α_h in the following way. (For simplicity we will assume in what follows that $|q| = 1$ so that a typical argument $b \in \underline{A}^z$ to $\widehat{g}$ is a scalar rather than a vector.)

Given arguments $a = (a_1, \ldots, a_n)$, we replace each argument a_k that is a stream of sort s with what we call its *code*

$$code(k) = (j(k), a_{i_1}, \ldots, a_{i_{r-1}}, b, a_{i_{r+1}}, \ldots, a_{i_m}) \tag{2}$$

for $k = 1, \ldots, n$ wherein r is such that $a_{i_r} = a_p$ as above. The idea here is that first component of each code is the index of the stream it represents. (The code also contains copies of b and all the streams of sort s as these will be needed later.)

The crucial idea is to construct α'_h so that in each place that α_h uses a stream of sort s, α'_h uses a code instead. In particular, wherever $eval_s$ occurs in α_h and is passed a number t and a stream a to evaluate, at the corresponding place in α'_h the available arguments will be t and a code $c = code(k)$ for some $k \in \{1, \ldots, n\}$. Thus α_h can now inspect the first component of c and decide what action to take. Let the first component be some $j \in \{1, \ldots, m\}$. If $j = r$ then c represents a_p so α'_h needs to execute α_g on t and b — this is straightforward to arrange since t is already available and b is the $(r+1)$th component of c. Alternatively, if $j \neq r$ then c represents a_{i_j} so α'_h needs to execute $eval_s$ on t and this stream. Again, this is easy to arrange since a_{i_j} is the $(j+1)$th component of c.

More formally, the scheme α'_h is obtained from α_h in the following way. First note that a code has type $\mathbf{n}e$ where e comprises m copies of $\underline{s}$ with the rth occurrence of $\underline{s}$ replaced with q. Thus codes are members of the set

$$\underline{A}^{\mathbf{n}e} = \mathbf{N} \times [T \to A_s]^r \times \underline{A}^z \times [T \to A_s]^{m-r}.$$

We now construct α_h so that it computes in the way described above by replacing all occurrences in α'_h of $eval_s$ in α_h by a scheme $Eval$ of type

$$[\![Eval]\!]_{\underline{A}} : T \times \underline{A}^{\mathbf{n}e} \to A_s$$

that is,

$$[\![Eval]\!]_{\underline{A}} : T \times \mathbf{N} \times [T \to A_s]^r \times \underline{A}^z \times [T \to A_s]^{m-r} \to A_s$$

and is defined for each $t \in T$ and each $c = (c_1, \ldots, c_{m+1}) \in \mathbb{N} \times \underline{A}^e$ by

$$(3) \qquad [\![Eval]\!]_{\underline{A}}(t, c) = \begin{cases} [\![\alpha_g]\!]_{\underline{A}}(t, c_{r+1}) & \text{if } c_1 = r \\ [\![eval_s]\!]_{\underline{A}}(t, c_{c_1+1}) & \text{if } c_1 \neq r. \end{cases}$$

4.5.7 Remarks

The transformation of α_h into α'_h obtained by substituting $Eval$ for $eval_s$ is the essence of our method of compiling α_g and α_h into α_f. However, there are a number of matters arising:

(1) Some readers may have asked themselves why we do not use part (b) of Lemma 7 in our construction to compute the index of a stream at the point at which that stream is about to be used as an argument to $eval_s$. The reason for this is simply that we have been unable to prove Lemma 7 without using results based on the composition of CFSTs(!)

(2) Simply substituting $Eval$ for $eval_s$ will create 'type clashes'.

(3) Although it is clear that α'_h can be used to compute f, α'_h is not the required scheme α_f since $[\![\alpha'_h]\!]_{\underline{A}}$ does not have the same domain as f.

(4) This description of the compilation process does not consider the case where h returns streams.

(5) The presentation of $Eval$ contains a subtle flaw.

Each of points (2)–(5) has an impact on the precise formulation of the compilation of α_g and α_h into α_f.

With respect to (2), it is not sufficient to modify α_h by simply replacing occurrences of $eval_s$ with $Eval$ as we must also modify all of α_h's other basis schemes so that they have the right types. For example, if α_h involves a constant scheme d^x for some constant symbol d and some $x \in \underline{S}^+$, then in α'_h this needs to be changed to $d^{x'}$ where x' is x with any occurrence of $\underline{s}$ is replaced by the type **ne** of a code. Projection schemes and definition-by-cases schemes require similar modifications, but we will leave the details to the formal definitions in Section 4.5.3.

With respect to (3), the constructions discussed above result in a scheme α'_h such that

$$[\![\alpha'_h]\!]_{\underline{A}} : \underline{A}^{d'} \to \underline{A}^x$$

where d' is d with every occurrence of $\underline{s}$ replaced by **ne**. Notice that if $\vec{a}$ denotes $a = (a_1, \ldots, a_n)$ with each a_k that is a stream of sort s replaced by $code(k)$ for $k \in \{1, \ldots, n\}$, then it follows from the above discussion of codes and $Eval$ that

$$[\![\alpha'_h]\!]_{\underline{A}}(\vec{a}) = f(a\{p/b\}).$$

Thus α'_h is the required characterisation α_f of $f = h\hat{\circ}_p g$ except that α'_h has the wrong domain. However, it is easy to see that $\vec{a}$ can be computed from $a\{p/b\}$ with a PR scheme $Init$ such that if we define α_f by

$$\alpha_f = \alpha'_h \circ Init$$

then

$$\begin{aligned}
[\![\alpha_f]\!]_{\underline{A}}(a\{p/b\}) &= [\![\alpha'_h \circ Init]\!]_{\underline{A}}(a\{p/b\}) \\
&= [\![\alpha'_h]\!]_{\underline{A}}([\![Init]\!]_{\underline{A}}(a\{p/b\})) \\
&= [\![\alpha'_h]\!]_{\underline{A}}(\vec{a}) \\
&= f(a\{p/b\}).
\end{aligned}$$

(Recall that *Init* was defined in Definition 23.)

With respect to (4), the hypothesis on the function h is that it does not return streams. (Recall that the co-domain of h is $\underline{A}^x$ for $x \in S^+$ and not $x \in \underline{S}^+$). However, notice that it is possible for α_h to be of the form $\alpha_h = \alpha_2 \circ \alpha_1$ where α_1 returns streams. This means that a theorem stating the primitive recursiveness of $f = h\hat{\circ}_p g$ cannot be proved by an argument that proceeds by induction on the structure of α_h since the induction hypothesis will not be strong enough. (We note in passing that g and h must have the general form given in Definition 30 for the same reason.)

This matter is easily resolved: if we allow the co-domain of α_h to be $\underline{A}^x$ for $x \in \underline{S}^+$ then the construction will generate a scheme α'_h with co-domain $\underline{A}^{x'}$ where x' is x with every occurrence of $\underline{s}$ replaced with $\mathbf{n}e$. Since this is not the co-domain of f, the theorem to prove in this case is one stating that for each co-ordinate function h_i of h: (a) if the co-domain of h_i is not $[T \to A_s]$ then the ith co-ordinate of $[\![\alpha'_h \circ Init]\!]_{\underline{A}}$ returns the ith co-ordinate of $f\hat{\circ}_p g$, and (b) if the co-domain of h_i is $[T \to A_s]$ then $[\![\alpha'_h \circ Init]\!]_{\underline{A}}$ returns the *code of* the stream returned by $(f\hat{\circ}_p g)_i$. (Theorem 9 that states the correctness of the compilation process has exactly this form.)

With respect to (5), the discussion of *Eval* was simplified in Section 4.5.6 for ease of presentation. The difficulty is that the generalisation of h described immediately above is still not sufficient for a proof by structural induction to succeed if we use *Eval* naively.

To see the difficulty, recall the semantics of *Eval* from equation (3) above. Notice how this definition is implicitly dependent on d (the domain type of h); that is, '*Eval*' is properly '$Eval_d$'. For example, suppose $\alpha_h = eval_s$. In this case d is $\mathbf{n}\underline{s}$, and since the second argument to h is the only stream, we deduce that $p = 2$ is the only possibility. This means that there is only one code: $code(2)$, and it must be that $[\![Init]\!]_{\underline{A}}$ always returns $code(2) = (1, b)$ since the second argument is the first and only stream. This in turn means that the definition of $Eval = Eval_{\mathbf{n}\underline{s}}$ collapses to $Eval = \alpha_g$. While this is correct in the case where α_h is $eval_s$, in general $eval_s$ will only be a sub-scheme of α_h. Thus in an argument that proceeds by induction on the structure of α_h we need to replace $eval_s$ by $Eval_d$, but the induction hypothesis will only cover the case where $u = \mathbf{n}\underline{s}$. This is not what is required, and as with the situation above, we must generalise the construction again to give us a stronger induction hypothesis.

We do this in the following way. When $eval_s$ occurs as a sub-scheme of α_h, the only streams on which $eval_s$ could ever be executed must be inputs to α_h. In other words, the (stream) arguments to $eval_s$ are contained in the (stream) arguments to α_h. To reflect this situation in an induction hypothesis we consider the substitution of $\hat{g}(b)$ for the pth argument in any list $a' = (a'_1, \ldots, a'_{n'})$ which contains the arguments $a = (a_1, \ldots, a_n)$ to α_h. More precisely, in terms of the notation and terminology of Section 4.2.4, our approach is to consider the effect of substituting $\hat{g}(b)$ for the pth component of a vector a' of type w where $w \supseteq d$ and there exists some

w/d-replacement ϕ. (This was the reason for defining sort replacements in Definition 22.)

In an argument based on the structure of α_h in this context, the basis case where $\alpha_h = eval_s$ does not require the domain type d of α_h to be exactly $\mathbf{n}\underline{s}$ as it is now sufficient to have $d \supseteq \underline{s}$. This reflects the idea that the stream argument to $eval_s$ when it is a sub-scheme of α_h must be one of α_h's arguments, and leads to a construction wherein $eval_s$ is replaced by $Eval_d$ as required.

Finally, to recover the theorem we set out to prove, it is a simple matter of considering the special case where $w = d$ and ϕ is the identity function.

4.5.8 A Formal Definition of the Compiler $\mathcal{C}$

In the previous section we explained informally the structure of a compiler $\mathcal{C}$ such that

$$\mathcal{C}(\alpha_g, \alpha_h) = \alpha'_h \circ Init$$

and

$$[\![\mathcal{C}(\alpha_g, \alpha_h)]\!]_{\underline{A}} = [\![\alpha_h]\!]_{\underline{A}} \,\hat{\circ}_p\, [\![\alpha_g]\!]_{\underline{A}}$$

(Recall that α_h is of type (d, x) wherein $d_p = \underline{s}$ and α_g is of type $(\mathbf{t}\ q, s)$.) In this section we present a complete definition of this compiler and make a formal statement of its properties that we use to prove Theorem 7.

For technical convenience the formal definition of $\mathcal{C}$ is given by two separate functions. For example, the construction of the scheme α'_h is formalized by the definition of the compiler $\Diamond$ wherein for additional technical reasons concerning the functionality of $\Diamond$ the scheme α_g is given as an index rather than an argument. For example, the formal construction of the scheme α'_h as above is given by

$$\alpha'_h = \Diamond_{d,x}^{\alpha_g, d, p}(\alpha_h).$$

The formal construction of a scheme to represent the function $Init$ was given in the latter part of Section 4.2.4 to which the reader may now wish to refer before continuing. Specifically, notice that the function $Init$ is used to compute the appropriate input to α'_h where each stream is replaced by a code vector, and the function θ is used to compute the individual code for each particular stream. The function Δ computes the correct type for the domain and co-domain of α'_h where streams are replaced by codes and the function δ is used to compute the type of each individual code. In particular, in the case of α'_h as above $\Delta^{\underline{s}, d, p}(d, q)$, and $\Delta^{\underline{s}, x, p}(d, q)$ are the domain and co-domain of α'_h respectively wherein the type of each code vector replacing each stream of sort $\underline{s}$ is $\delta^{\underline{s}, d, p}(q)$; and if id is the identity function on $\{1, \ldots, |d|\}$ then for each $a = (a', b, a'') \in A^{d\{p/q\}}$ the input to α'_h is $Init^{id, q, \underline{s}, p}(a)$ wherein $Init^{id, q, \underline{s}, p}$ has functionality $Init^{id, q, \underline{s}, p} : A^{d\{p/q\}} \rightarrow A^{\Delta^{\underline{s}, d, p}(d, q)}$ and replaces each stream of type $\underline{s}$ occurring as the kth co-ordinate of d with the code $\theta^{\underline{s}, d, p, q, k}(a', a'', b)$.

The Compiler $\Diamond$. We now present the formal definition of the compiler $\Diamond$. Recall that for technical reasons concerned with the formal proof of the properties of $\mathcal{C}$ (stated as Theorem 9) that the definition of $\Diamond$ involves a further domain type w such that $w \supseteq u$ (wherein u

is now the domain of the function we are performing Cartesian composition on) and a w/u-replacement ϕ. However, to deduce Theorem 7 from Theorem 9 it is sufficient to have $w = u$ and ϕ as the identity function (as above).

Definition 32. Let $\underline{\Sigma}$ be any standard $\underline{S}$-sorted signature. For each $s \in S$, for each $w, u, v \in \underline{S}^+$ such that $w \supseteq u$ and $I^{\underline{s},w} \neq \varnothing$ for each $p \in I^{s,w}$, and for each $\beta \in \mu PR(\underline{\Sigma})_{t\,z,s}$ for any $z \in \underline{S}^*$ we define

$$\Diamond_{u,v}^{\beta,w,p} : \mu PR(\underline{\Sigma})_{u,v} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(v,z)}$$

(ambiguously denoted $\Diamond^{\beta,w,p}$ or just $\Diamond$) by induction on the structural complexity of a scheme $\alpha \in \mu PR(\underline{\Sigma})_{u,v}$ uniformly in (u, v) as follows:

Basis Cases.

(1) Constant Functions. If $\alpha = c^u$ for some $c \in \Sigma_{\lambda,s'}$ for any $s' \in S$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,s'} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(s',z)}$$

is defined by

$$\Diamond(\alpha) = c^{\Delta^{\underline{s},w,p}(u,z)}.$$

Well-Definedness. Since $\Delta^{\underline{s},w,p}(s', z) = s'$ by definition it is clear that

$$\Diamond(\alpha) \in \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(v,z)}$$

as required.

(2) Algebraic Operations. If $\alpha = \sigma$ for some $\sigma \in \Sigma_{u,s'}$ for any $s' \in S$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,s'} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(s',z)}$$

is defined by

$$\Diamond(\alpha) = \begin{cases} \sigma & \text{if } \sigma \neq eval_s, \\ Eval_s^{w,z,p} & \text{if } \sigma = eval_s \end{cases}$$

wherein for each $w \in \underline{S}^+$, for each $z \in \underline{S}^*$ and for each $p \in I^{\underline{s},w}$

$$Eval_s^{w,z,p} = switch^{|I^{\underline{s},w}|,s,\lambda_1^{\underline{s},w}(p)} \circ < U_2^{t\,x}, \beta_1, \ldots, \beta_{|I^{\underline{s},w}|} >$$

where $switch^{I^{\underline{s},w},s,\lambda_1^{\underline{s},w}(p)}$ is as in Lemma 5, $x = \delta^{\underline{s},w,p}(z)$, and for $j = 1, \ldots, |I^{\underline{s},w}|$,

$$\beta_j = \begin{cases} eval_s \circ < U_1^{t\,x}, U_{j+2}^{t\,x} > & \text{if } 1 \leq j < \lambda_1^{\underline{s},w}(p); \\ \beta \circ < U_1^{t\,x}, U_{j+2}^{t\,x}, \ldots, U_{j+|z|+1}^{t\,x} > & \text{if } j = \lambda_1^{\underline{s},w}(p); \\ eval_s \circ < U_1^{t\,x}, U_{j+|z|+1}^{t\,x} > & \text{if } \lambda_1^{\underline{s},w}(p) < j \leq |I^{\underline{s},w}|. \end{cases}$$

Note, when defining β_j above, in the case that $j = \lambda_1^{\underline{s},w}(p)$ if $z = \varepsilon$ and hence $|z| = 0$ then the sequence of projections $U_{j+2}^{t\,x}, \ldots, U_{j+|z|+1}^{t\,x}$ is empty and we operate the convention that $\beta_j = \beta \circ < U_1^{t\,x} >$.

Well-Definedness. We have two cases to consider:

(a) $\sigma \neq eval_s$. Notice that since $\sigma \neq eval_s$ we have $\Delta^{\underline{s},w,p}(u,z) = u$ and $\Delta^{\underline{s},w,p}(s',z) = s'$ by definition and therefore $\Diamond(\alpha) \in \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\ \Delta^{\underline{s},w,p}(s',z)}$ as required.

(b) $\sigma = eval_s$. First notice that if $1 \leq j < \lambda_1^{\underline{s},w}(p)$ then $x_{j+2} = \underline{s}$ by definition. Similarly if $\lambda_1^{\underline{s},w}(p) < j \leq |I^{\underline{s},w}|$ then $x_{j+|z|+1} = \underline{s}$. Thus, $eval_s \circ < U_1^{\mathbf{t}\,x}, U_{j+2}^{\mathbf{t}\,x} >$ and $eval_s \circ < U_1^{\mathbf{t}\,x}, U_{j+|z|+1}^{\mathbf{t}\,x} >$ are well-defined as compositions since the domain of $eval_s$ is $\mathbf{t}\ \underline{s}$. Second, notice that if $j = \lambda_1^{\underline{s},w}(p)$ then $U_{j+2}^{\mathbf{t}\,x}, \ldots, U_{j+|z|+1}^{\mathbf{t}\,x}$ is of type $(\mathbf{t}\,x, z)$ by definition and therefore $\beta \circ < U_1^{\mathbf{t}\,x}, U_{j+2}^{\mathbf{t}\,x}, \ldots, U_{j+|z|+1}^{\mathbf{t}\,x} >$ is well-defined since by hypothesis the domain of β is $\mathbf{t}\,z$. Also, notice that $< U_2^{\mathbf{t}\,x}, \beta_1, \ldots, \beta_{|I^{\underline{s},w}|} >$ is well-defined since $\beta_j \in PR(\underline{\Sigma})_{\mathbf{t}\,x,s}$ for $j = 1, \ldots, |I^{\underline{s},w}|$ and $\mathrm{U}_2^{\mathbf{t}\,x}$ is of type $(\mathbf{t}\,x, \mathbf{n})$ by the definition of x and by hypothesis $switch^{|I^{\underline{s},w}|,s,\lambda_1^{\underline{s},w}(p)} \in PR(\underline{\Sigma})_{\mathbf{n}\,x',s}$ wherein $x' = s_1 \cdots s_{|I^{\underline{s},w}|}$ such that $s_q = s$ for $q = 1, \ldots, |I^{\underline{s},w}|$. Therefore $switch^{|I^{\underline{s},w}|,s,\lambda_1^{\underline{s},w}(p)} \circ < U_2^{\mathbf{t}\,x}, \beta_1, \ldots, \beta_{|I^{\underline{s},w}|} >$ is well-defined as a composition. Finally, since $\Delta^{\underline{s},w,p}(\mathbf{t}\ \underline{s}, z) = \mathbf{t}\ \delta^{\underline{s},w,p}(z)$ by definition and $\Delta^{\underline{s},w,p}(s,z) = s$ by definition, we have $\Diamond(\alpha) \in \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\ \Delta^{\underline{s},w,p}(s,z)}$ as required.

(3) Projection Functions. If $\alpha = \mathrm{U}_j^u$ for some $j \in \{1, \ldots, |u|\}$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,u_j} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(u_j,z)}$$

is defined by

$$\Diamond(\alpha) = \begin{cases} \mathrm{U}_{j'}^{\Delta^{\underline{s},w,p}(u,z)} & \text{if } j \notin I^{\underline{s},u}, \\ < \mathrm{U}_{j'}^{\Delta^{\underline{s},w,p}(u,z)}, \ldots, \mathrm{U}_{j'+|\delta^{\underline{s},w,p}(z)|-1}^{\Delta^{\underline{s},w,p}(u,z)} > & \text{if } j \in I^{\underline{s},u} \end{cases}$$

wherein $j' = |\Delta^{\underline{s},w,p}(u_1 \cdots u_{j-1}, z)| + 1$.

Well-Definedness. We have two cases to consider:

(a) $j \notin I^{\underline{s},u}$. Notice that since $j \notin I^{\underline{s},u}$ we have $u_j \neq \underline{s}$ and hence $\Delta^{\underline{s},w,p}(u_j, z) = u_j$ by definition. Therefore as $j' = |\Delta^{\underline{s},w,p}(u_1 \cdots u_{j-1}, z)| + 1$ we have $(\Delta^{\underline{s},w,p}(u,z))_{j'} = u_j$ and therefore $\Diamond(\alpha) \in \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(u_j,z)}$ as required.

(b) $j \in I^{\underline{s},u}$. Notice that since $j \in I^{\underline{s},u}$ we have $u_j = \underline{s}$ and hence $\Delta^{\underline{s},w,p}(u_j, z) = \delta^{\underline{s},w,p}(z)$ by definition. Therefore as $j' = |\Delta^{\underline{s},w,p}(u_1 \cdots u_{j-1}, z)| + 1$ we have

$$((\Delta^{\underline{s},w,p}(u,z))_{j'} \cdots (\Delta^{\underline{s},w,p}(u,z))_{j'+|\delta^{\underline{s},w,p}(z)|-1}) = \delta^{\underline{s},w,p}(z)$$

and therefore $\Diamond(\alpha) \in \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\ \Delta^{\underline{s},w,p}(u_j,z)}$ as required.

Induction Hypothesis. Assume for any scheme $\alpha' \in \mu\mathrm{PR}(\underline{\Sigma})_{u',v'}$ for some $u', v' \in \underline{S}^+$ of less structural complexity than α that

$$\Diamond_{u',v'}^{\beta,w,p}(\alpha') \in \mu\mathrm{PR}(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u',z),\Delta^{\underline{s},w,p}(v',z)}.$$

Induction: Function Building Tools.

Note that in each of the cases below the well-definedness of $\Diamond(\alpha)$ follows immediately from the Induction Hypothesis. Consequently we leave the details to the reader.

(4) **Vectorisation.** If $\alpha =< \alpha_1, \ldots, \alpha_m >$ for some $\alpha_i \in \mu PR(\underline{\Sigma})_{u,s_i}$ for any $s_i \in S$ for $i = 1, \ldots, m > 0$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,s_1 \cdots s_m} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(s_1 \cdots s_m,z)}$$

is defined by

$$\Diamond(\alpha) =< \Diamond(\alpha_1), \ldots, \Diamond(\alpha_m) > .$$

(5) **Composition.** If $\alpha = \alpha_2 \circ \alpha_1$ for some $\alpha_1 \in \mu PR(\underline{\Sigma})_{u,u'}$ and for some $\alpha_2 \in \mu PR(\underline{\Sigma})_{u',v}$ for any $u' \in \underline{S}^+$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,v} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(v,z)}$$

is defined by

$$\Diamond(\alpha) = \Diamond(\alpha_2) \circ \Diamond(\alpha_1).$$

(6) **Simultaneous Primitive Recursion.** If $\alpha = *(\alpha_1, \alpha_2)$ for some $\alpha_1 \in \mu PR(\Sigma)_{u,v}$ and for some $\alpha_2 \in \mu PR(\Sigma)_{\mathbf{t}\,u\,v,v}$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{\mathbf{t}\,u,v} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(\mathbf{t}\,u,z),\Delta^{\underline{s},w,p}(v,z)}$$

is defined by

$$\Diamond(\alpha) = *(\Diamond(\alpha_1), \Diamond(\alpha_2)).$$

(7) **Minimalization.** If $\alpha = \mu(\alpha')$ for some $\alpha' \in \mu PR(\underline{\Sigma})_{\mathbf{n}\,u,\mathbf{b}}$ then

$$\Diamond : \mu PR(\underline{\Sigma})_{u,\mathbf{n}} \rightarrow \mu PR(\underline{\Sigma})_{\Delta^{\underline{s},w,p}(u,z),\Delta^{\underline{s},w,p}(\mathbf{n},z)}$$

is defined by

$$\Diamond(\alpha) = \mu(\Diamond(\alpha')).$$

Lemma 10. *Let $\underline{\Sigma}$ be any standard $\underline{S}$-sorted signature. For each $s \in S$, for each $w, u, v \in \underline{S}^+$ such that $w \supseteq u$ and $I^{\underline{s},w} \neq \varnothing$ for each $p \in I^{s,w}$, for each $\beta \in PR(\underline{\Sigma})_{\mathbf{t}\,z,s}$ for any $z \in \underline{S}^*$ and for any $\alpha \in PR(\underline{\Sigma})_{u,v}$*

$$\Diamond^{\beta,w,p}(\alpha) \in PR(\underline{\Sigma}).$$

Proof. Immediate from the definition of $\Diamond$ since $\Diamond$ never introduces any new instances of minimalization. This fact is used to deduce Part (1) of Theorem 7.

□

The Formal Definition of the compiler $\mathcal{C}$.

Definition 33. For each $s \in S$, for each $w, u \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \varnothing$, for each w/u-permutation ϕ, for each $p \in I^{\underline{s},w}$ for each $\alpha \in \mu\mathrm{PR}(\underline{\Sigma})_{u,v}$ and for each $\beta \in \mu PR(\underline{\Sigma})_{t\,z,s}$ for any $z \in \underline{S}^*$ we define

$$\mathcal{C}^{\phi,w,p}_{u,v,z,s} : \mu\mathrm{PR}(\underline{\Sigma})_{u,v} \times \mu\mathrm{PR}(\underline{\Sigma})_{t\,z,s} \to \mu\mathrm{PR}(\underline{\Sigma})_{w\{p/z\},\Delta^{\underline{s},w,p}(v,z)}$$

(ambiguously denoted $\mathcal{C}^{\phi,w,p}$) by

$$\mathcal{C}^{\phi,w,p} = \Diamond^{\beta,w,p}_{u,v}(\alpha) \circ Init^{\phi,z,\underline{s},p}.$$

Notice that by Lemma 6 we have $Init^{\phi,z,\underline{s},p} \in \mathrm{PR}(\underline{\Sigma})$. Therefore if $\alpha, \beta \in \mathrm{PR}(\underline{\Sigma})$ then by Lemma 10 we have $\Diamond^{\beta,w,p} \in \mathrm{PR}(\underline{\Sigma})$ and hence $\mathcal{C}^{\phi,w,p}(\alpha,\beta) \in \mathrm{PR}(\underline{\Sigma})$. Otherwise, if $\alpha, \beta \in \mu\mathrm{PR}(\underline{\Sigma})$ then $\mathcal{C}^{\phi,w,p}(\alpha,\beta) \in \mu\mathrm{PR}(\Sigma)$.

4.5.9 Proof of Theorem 8

We now complete this section with a formal statement of the properties of the compiler $\mathcal{C}$ and use this to prove Theorem 8 and hence Theorem 3. Notice that the statement of the theorem concerns the particular co-ordinates of the function created by the compiler $\mathcal{C}$. In particular, in the context of our example notice that if sort x_i of the co-domain of α_h is not a stream of type $\underline{s}$ then the output produced by the compiler is precisely as required by Theorem 7. However, as discussed under Point (4) in Section 4.5.7 if sort x_i of the co-domain of α_h is a stream of type $\underline{s}$ then the compiler produces a scheme that returns some code vector $\theta^{\underline{s},d,p,q,r}(a',a'',b)$ for some $r \in I^{\underline{s},d}$ of type $\delta^{\underline{s},d,p}(q)$ that was produced by $Init$.

Theorem 9. *Let A be any standard S-sorted Σ-algebra. For each $s \in S$, for each $w, u \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \varnothing$, for each w/u-permutation ϕ, for each $p \in I^{\underline{s},w}$, for each $\alpha \in \mu PR(\underline{\Sigma})_{u,v}$ for some $v \in \underline{S}^+$, for each $\beta \in \mu PR(\underline{\Sigma})_{t\,z,s}$ for some $z \in \underline{S}^*$ and for each $i = 1, \ldots, |v|$:*
(1) if $i \notin I^{\underline{s},v}$ then if we define

$$F_i^{\alpha,\beta,\phi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}_{v_i}$$

(ambiguously denoted $F_i^{\alpha,\beta}$) by

$$F_i^{\alpha,\beta,\phi,w,p} = (\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_i}$$

wherein $j_i = (i - |I^{\underline{s},v_1 \cdots v_{i-1}}|) + (|I^{\underline{s},v_1 \cdots v_{i-1}}| * |\delta^{\underline{s},w,p}(z)|) + 1$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha,\beta}(a\{p/b\}) = \Big(\llbracket \alpha \rrbracket_A(P(a\{p/b\}))\Big)_i$$

wherein

$$P : \underline{A}^{w\{p/z\}} \to \underline{A}^{u\{\overline{p}/z\}}$$

is defined by

$$P(a\{p/b\}) = (a_{\phi(1)}, \ldots, a_{\phi(\overline{p}-1)}, \widehat{\llbracket \beta \rrbracket_A}(b), a_{\phi(\overline{p}+1)}, \ldots, a_{\phi(|u|)});$$

wherein $\overline{p} = \overline{\phi}(p)$*; otherwise*
(2) if $i \in I^{\underline{s},v}$ *then if we define*

$$F_i^{\alpha,\beta,\phi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}^{\delta^{\underline{s},w,p}(z)}$$

(again ambiguously denoted $F_i^{\alpha,\beta}$*) by*

$$F_i^{\alpha,\beta,\phi,w,p} = ((\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_{i,1}}, \ldots, (\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_{i,|\delta^{\underline{s},w,p}(z)|}})$$

wherein $j_{i,1} = j_i$ *as defined above and* $j_{i,k} = j_{i,k-1} + 1$ *for* $k = 2, \ldots, |\delta^{\underline{s},w,p}(z)|$ *then*

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha,\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ *such that*

$$(P(a\{p/b\}))_{\overline{\phi}(r)} = (\llbracket \alpha \rrbracket_{\underline{A}}(P(a\{p/b\})))_i.$$

We now show how it is straightforward to deduce Theorem 8 from Theorem 9.

Proof of Theorem 8. Notice that as by hypothesis $h, g \in \mu\mathrm{PR}(\underline{A})$ there exist schemes $\alpha_h, \alpha_g \in \mu\mathrm{PR}(\underline{\Sigma})$ such that $\llbracket \alpha_h \rrbracket_{\underline{A}} = h$ and $\llbracket \alpha_g \rrbracket_{\underline{A}} = g$. We claim that by Theorem 9 if we take $u = w = d$, $\phi(j) = j$ for $j = 1, \ldots, |d|$, $v = x$, $z = q$, $\alpha = \alpha_h$, $\beta = \alpha_g$ and $p = i$ then

$$F^{\alpha_h,\alpha_g} = (F_1^{\alpha_h,\alpha_g}, \ldots, F_{|x|}^{\alpha_h,\alpha_g}) = h \,\widehat{\circ}_p\, g.$$

To see this first notice that $F_j^{\alpha_h,\alpha_g} \in \mu\mathrm{PR}(\underline{A})$ for $j = 1, \ldots, |x|$ and as $x \in S^+$ we have $\Delta^{\underline{s},d,i}(x,q) = x$ and hence F^{α_h,α_g} is of the same type as $h \,\widehat{\circ}_p\, g$. Also, again since $x \in S^+$, by Theorem 9 we may calculate as follows for each $a\{p/b\} \in \underline{A}^{d\{p/q\}}$

$$\begin{aligned}
F^{\alpha_h,\alpha_g}(a\{p/b\}) &= (F_1^{\alpha_h,\alpha_g}, \ldots, F_{|x|}^{\alpha_h,\alpha_g}) \\
&= (\llbracket \alpha_h \rrbracket_{\underline{A}}(G(a\{p/b\}))_1, \ldots, \llbracket \alpha_h \rrbracket_{\underline{A}}(G(a\{p/b\}))_{|x|}) \\
&= \llbracket \alpha_h \rrbracket_{\underline{A}}(G(a\{p/b\})) \\
&= \llbracket \alpha_h \rrbracket_{\underline{A}}(a_1, \ldots, a_{p-1}, \widehat{\llbracket \alpha_g \rrbracket_{\underline{A}}}(b), a_{p+1}, \ldots, a_{|d|})
\end{aligned}$$

by the definition of G with the hypothesis that $\phi(j) = j$

$$= h(a_1, \ldots, a_{p-1}, \widehat{g}(b), a_{p+1}, \ldots, a_{|d|})$$

by hypothesis

$$= h \,\widehat{\circ}_p\, g(a\{p/b\})$$

by the definition of $h \,\widehat{\circ}_p\, g$ as required.

Chapter 5

Primitive Recursive Equational Specification

A little inaccuracy sometimes saves tons of explanation.

Saki

5.1 Introduction

In this chapter we define the language PREQ that we will use to give a formal semantics to our stream processing language ASTRAL that is presented in the following chapter. Our motivation for the development of PREQ can be found in Sections 3.7.1 and 4.1 and in particular Section 4.1.2.

5.1.1 Overview

Essentially we have two main aims in this chapter: (1) to show formally that PR and PREQ are equivalent in their expressive power; and (2) to establish some formal properties of PREQ that demonstrate its effectiveness as a specification language from the perspective of automated verification. In a similar fashion to Chapter 4 wherein we used the formal compiler $\mathcal{C}$ to establish a theorem constructively, in order to prove Property (1) we will employ the use of two formal compilers: $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$. As with the compiler $\mathbb{C}$ both of the compilers $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$ are also highly technical in nature and so the formal proofs of their properties (Lemmata 27 and 28) are again banished to an appendix (Appendix B). Therefore, as before we concentrate on an informal, but rigorous algorithmic description of our compilation techniques.

In more detail, this chapter is structured as follows: in Section 5.2 we introduce the important concept of a normal form representation for PR schemes. The use of normal forms will enable us to reduce the number of equations produced by PR schemes when they are compiled into PREQ to a theoretical minimum. As a motivation for this normal form representation we present the high-level algorithm for compiling PR schemes into equations that is the basis for the compiler definitions that we use to show the adequacy of the language PREQ.

In Section 5.3 we concentrate on the development of the language PREQ. First, in Section 5.3.2 we define the syntax and semantics of PREQ. In Section 5.3.3 we are now in a position to define the two formal compilers that we will use to show that PREQ captures the class $\mathbb{PR}$:

Theorem 10. *There exist compilers*

$$\mathbb{C}^{PR} : PREQ(\Sigma, X) \rightarrow PR(\Sigma)$$

and

$$\mathbb{C}^{PREQ} : PR(\Sigma) \rightarrow PREQ(\Sigma, X)$$

such that

$$(\forall \Phi \in PREQ(\Sigma, X)) \quad [\![\Phi]\!]_A = [\![\mathbb{C}^{PR}(\Phi)]\!]_A$$

and

$$(\forall \alpha \in PR(\Sigma)) \quad [\![\alpha]\!]_A = [\![\mathbb{C}^{PREQ}(\alpha)]\!]_A$$

respectively from which we deduce that
(1)

$$(\forall \Phi \in PREQ(\Sigma, X))(\exists \alpha \in PR(\Sigma)) \quad [\![\alpha]\!]_A = [\![\Phi]\!]_A$$

and
(2)

$$(\forall \alpha \in PR(\Sigma))(\exists \Phi \in PREQ(\Sigma, X)) \quad [\![\Phi]\!]_A = [\![\alpha]\!]_A.$$

To conclude this chapter (Section 5.4) we concentrate on the properties of PREQ from the perspective of automated verification; that is, we show that any PREQ specification when interpreted as a TRS is *complete*:

Theorem 11. *If $\Phi \in PREQ(\Sigma, X)$ and $\mathcal{R} = TRCON(\Phi) \subseteq TRS(\Sigma, X)$; that is, if $\mathcal{R}$ is the term re-writing system formed from Φ by orienting each equation in Φ as a left-to-right re-write rule then $\mathcal{R}$ is complete.*

As with the previous chapter this chapter is also predominantly concerned with the technical development of our stream processing theory. Therefore as before for the reader not interested in these details we will indicate which sections can be omitted.

5.2 Compiling PR into Equations: PR Normal Forms

In order to establish the adequacy of the language PREQ in the sequel we will define a compiler $\mathbb{C}^{\text{PREQ}}$ that maps PR schemes into a semantically equivalent equational representation in the PREQ syntax. A compiler that maps PR schemes into equivalent equational representations can already be found in Thompson and Tucker [1991]. However, while this compiler is mathematically concise in its construction, it is highly inefficient from the perspective of the number of equations produced from a PR scheme and this in turn leads to impractically large TRSs (see Chapter 8).

In order to produce compact TRSs the compiler $\mathbb{C}^{\text{PREQ}}$ in this thesis relies on a scheme $\alpha \in \text{PR}(\Sigma)$ being first converted into an equivalent 'normal form' α' before it is finally converted into equations. (In the sequel we will denote the class of all such normal forms PR_E.) This intermediate compilation into a normal form means that the number of equations created from a PR scheme α can be reduced to what in general constitutes a theoretical minimum: if k is the number of applications of primitive recursion in a scheme α then $\mathbb{C}^{\text{PREQ}}$ produces either $2k$ equations if $\alpha = *(\alpha_1, \alpha_2)$ or $2k - 1$ equations otherwise.

The formalization of this intermediate compilation process relies on the construction of a compiler

$$\mathbb{C}^{\text{PR}_E} : \text{PR}(\Sigma) \to \text{PR}_E(\Sigma \cup \mathcal{F})$$

(wherein $\mathcal{F}$ is a set of additional function symbols dependent on the scheme α that we are compiling) that essentially performs the following two operations: (1) the replacement of occurrences of primitive recursion by new function symbols (Definition 44); and (2) the elimination of occurrences of vector-valued compositions (Definition 34).

While at an intuitive level $\mathbb{C}^{\text{PR}_E}$ is straightforward in its operation, a detailed description of its precise functional behaviour is highly technical. Therefore, we motivate the form of the systems of equations that it produces with a high-level algorithmic explanation of the recursive

operation of the compiler $\mathbb{C}^{\text{PREQ}}$. As such, we suggest the reader not interested in the precise nature of our compilation technique limit their attention to this informal algorithmic description of the operation of $\mathbb{C}^{\text{PREQ}}$ presented in Section 5.2.2. For the reader who is interested in the technical details of the compiler $\mathbb{C}^{\text{PREQ}}$ we suggest they limit their attention to Section 5.2.2 on a first reading, and return to Sections 5.2.3 to 5.2.6 when they are familiar with the intuitive ideas underlying our technical definitions.

Finally, despite the fact that the elimination of vector-valued composition from a scheme relies on first replacing occurrences of the primitive recursion operator with new function symbols we explain vector-valued composition elimination first as it is intuitively more straightforward and is only performed for technical convenience.

5.2.1 Eliminating Vector-Valued Compositions

The elimination of vector-valued compositions from a scheme α relies on the fact that either α contains no application of the primitive recursion operator or α contains one application wherein α is of the form $\alpha = *(\alpha_1, \alpha_2)$. The class of all PR schemes that satisfy this property (see the following section) are denoted $\text{PR}_C(\Sigma)$, and we call the process of eliminating vector-valued compositions from a scheme $\alpha \in \text{PR}_C(\Sigma)$ *thinning*.

Essentially, the compiler that we define to perform the thinning operation:

$$\text{Thin} : \text{PR}_C(\Sigma) \rightarrow \text{PR}_E(\Sigma)$$

is nothing more than a homomorphism that converts any sub-schemes of a scheme $\alpha \in \text{PR}_C(\Sigma)$ of the form

$$< \alpha_{2,1}, \alpha_{2,2}, \ldots, \alpha_{2,n} > \circ \alpha_1$$

into a semantically equivalent scheme of the form

$$< \alpha_{2,1} \circ \alpha_1, \alpha_{2,2} \circ \alpha_1, \ldots, \alpha_{2,n} \circ \alpha_1 > .$$

The operation of Thin is formalized as follows:

Definition 34. For each $u \in S^+$ and for each $v \in S^+$ we define

$$\text{Thin}_{u,v} : \text{PR}_C(\Sigma) \rightarrow \text{PR}_E(\Sigma)$$

(ambiguously denoted Thin) by induction on the structural complexity of an argument $\alpha \in \text{PR}_C(\Sigma)_{u,v}$ as follows:

Basis Schema.

(1) Constant Functions, (2) Algebraic Operations, (3) Projections and (4) Definition-by-Cases. If either $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $w \in S^+$ and for some $s \in S$ or $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$, for some $w \in S^+$ and for some $s \in S$ or $\alpha = \text{U}_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$ or $\alpha = \text{dc}_s$ for some $s \in S$ then

$$\text{Thin}(\alpha) = \alpha.$$

Induction.

(5) Vectorization. If $\alpha = < \alpha_1, \ldots, \alpha_m >$ wherein $\alpha_i \in \mathrm{PR}_C(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1, \ldots, m$ then

$$\mathrm{Thin}(\alpha) = < \mathrm{Thin}(\alpha_1), \ldots, \mathrm{Thin}(\alpha_m) > .$$

(6) Composition. If $\alpha = \alpha_2 \circ \alpha_1$ wherein $\alpha_1 \in \mathrm{PR}_C(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}_C(\Sigma)_{w,v}$ for some $u, v, w \in S^+$ then

$$\mathrm{Thin}(\alpha) = \begin{cases} \alpha_2 \circ \mathrm{Thin}(\alpha_1) & \text{if } |\mathrm{ran}(\alpha_2)| = 1, \\ < \mathrm{Thin}(\alpha_{2,1} \circ \alpha_1), \ldots, \mathrm{Thin}(\alpha_{2,|v|} \circ \alpha_1) > & \text{if } \alpha_2 = < \alpha_{2,1}, \ldots, \alpha_{2,|v|} >, \text{ and} \\ \mathrm{Thin}(\alpha_{2,2} \circ (\alpha_{2,1} \circ \alpha_1)) & \text{if } \alpha_2 = \alpha_{2,2} \circ \alpha_{2,1}. \end{cases}$$

Notice that these three cases are exhaustive by the hypothesis that $\alpha \in \mathrm{PR}_C(\Sigma)$.

(7) Simultaneous Primitive Recursion. If $\alpha = *(\alpha_1, \alpha_2)$ wherein $\alpha_1 \in \mathrm{PR}_C(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}_C(\Sigma)_{\mathbf{n}\,u\,v,v}$ for some $u, v \in S^+$ then

$$\mathrm{Thin}(\alpha) = *(\mathrm{Thin}(\alpha_1), \mathrm{Thin}(\alpha_2)).$$

Notice that from a close examination of Case (6) of the preceding definition it is not immediately obvious that Thin is terminating. However, the compiler can be shown to terminate by observing that Thin reduces the number of sub-schemes of α such that

$$\alpha = \alpha_2 \circ \alpha_{1,2} \circ \alpha_{1,1}$$

wherein $|\mathrm{ran}(\alpha_2)| > 1$. However, we leave the details of a formal proof of this fact to the reader.

The main property of Thin that we require can now be stated as follows:

Lemma 11. *If $\alpha \in PR_C(\Sigma)_{u,v}$ for some $u, v \in S^+$ then*

$$[\![\alpha]\!]_A = [\![\mathit{Thin}(\alpha)]\!]_A.$$

Proof. By induction on the structural complexity of α and by sub-induction on the number of sub-schema $\beta \subseteq \alpha$ such that $\beta = \beta_2 \circ \beta_1$ wherein $|\mathrm{ran}(\gamma_2)| > 1$.

□

As demonstrated by Lemma 11 the correctness of the compiler Thin relies on the fact that a scheme α satisfies $\alpha \in \mathrm{PR}_C(\Sigma)$. As such, we now informally define the process of converting a scheme into an equivalent representation in $\mathrm{PR}_C(\Sigma)$.

5.2.2 An Informal Algorithmic Description of Compiling PR into Equations

We first explain the intuitive visualization technique on which the operation of the compilers $\mathbb{C}^{\mathrm{PR}_E}$ and $\mathbb{C}^{\mathrm{PREQ}}$ are based and a straightforward theoretical result that provides the basis for a recursively defined compilation procedure.

Visualizing PR Schemes. Exploiting a technique that is common in computer science we can visualize a PR scheme as an inverted tree wherein *branches* represent applications of the function building tools – vectorization, composition and primitive recursion – and *leaves* represent constants, projections and algebraic operations. In particular, according to some predefined ordering we may index the *nodes* (branches and leaves) of a tree to enable us to talk about concepts such as 'the ith node of scheme α' and 'the jth primitive recursive node of scheme α'. For example, the scheme

$$\alpha = *(*(*(.,.),.),\nabla(*(.,*(.,.)),*(.,.)))$$

wherein '.' represents some basis scheme and ∇ represents some function building tool other than primitive recursion can be visualized as the tree shown in Figure 5.1. Specifically, notice that each node is annotated with a tuple (x, y, z) that represents three separate indexes as per a top-down, breath-first traversal of the tree: the number x counts each node; ignoring the root node, the number y counts only branches that represent an application of the primitive recursion operator – what we refer to as 'proper' primitive recursions; and ignoring the root node again, the number z counts only branches that represent applications of the primitive recursion operator that *do not* occur below any other applications of primitive recursion – what we shall refer to as 'top-level' primitive recursions.

In particular, observe that using this indexing: the second node is the first proper primitive recursive node and also the first top-level primitive recursive node; the tenth node is the fourth proper primitive recursive node, but is not a top-level primitive recursion; and the thirteenth node is the fifth proper primitive recursive node and also the third top-level primitive recursive node.

Eliminating Top-Level Primitive Recursions. Recall the meaning of A_f from Section 4.2.4. In general, as part of our normalization process we wish to eliminate top-level primitive recursions by replacing them with (new) function identifiers. For example, in the case of α as defined in Figure 5.1 we can replace nodes 2, 8 and 13 with the function symbols f_2, f_4 and f_6 respectively (the reason for this choice of numbering is explained in the sequel) to give scheme

$$\alpha' = *(f_2, \nabla(f_4, f_6)).$$

Of course now $\alpha' \in \mathrm{PR}(\Sigma \cup \{f_2, f_4, f_6\})$, but if we define f_2, f_4 and f_6 by the semantics of the sub-scheme of α that they have replaced; that is, if we define

$$f_2^{A_{f_2,f_4,f_6}} = [\![\alpha^2]\!]_A,$$

$$f_4^{A_{f_2,f_4,f_6}} = [\![\alpha^8]\!]_A$$

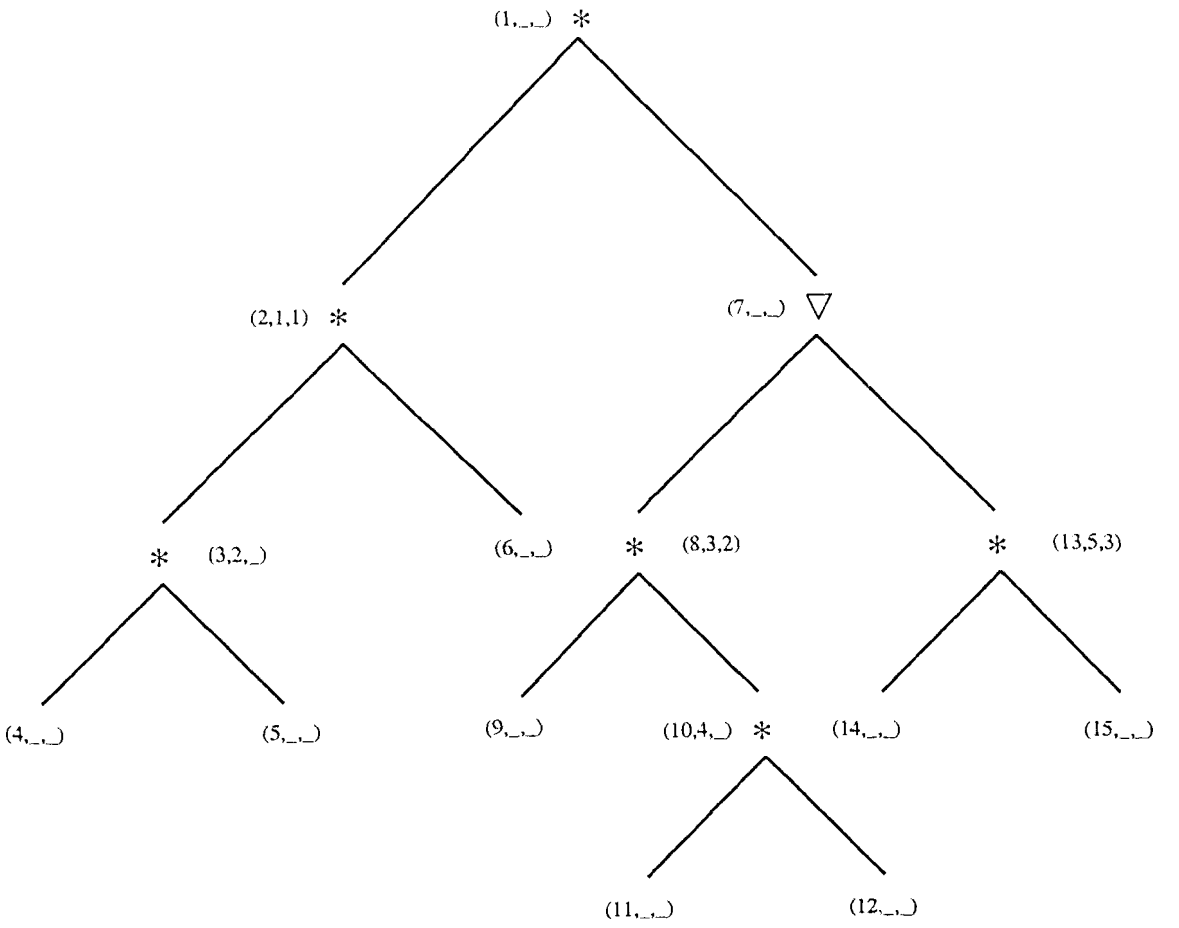

Figure 5.1: An Example Tree Representation of a PR Scheme with Three Indexes

and

$$f_6^{A_{f_2,f_4,f_5}} = [\![\alpha^{13}]\!]_A$$

wherein α^2, α^8 and α^{13} are the sub-schema of α with nodes 2, 8 and 13 respectively as their root node then

$$[\![\alpha']\!]_{A_{f_2,f_4,f_5}} = [\![\alpha]\!]_A.$$

More generally, the result on which the correctness of this phase of normalization is based can be stated as follows: given any scheme $\alpha \in \mathrm{PR}(\Sigma \cup \{f\})$ if f represents a function that is primitive recursive over algebra A then there exists a scheme $\alpha' \in \mathrm{PR}(\Sigma)$ such that semantically α and α' are equivalent:

Lemma 12. *Let $f_A : A^{u'} \to A^{v'}$ for some $u', v' \in S^+$ be any function that is primitive recursive over A. If $\alpha \in PR(\Sigma \cup \{f\})_{u,v}$ for some $u, v \in S^+$ then there exists a scheme $\alpha' \in PR(\Sigma)_{u,v}$ such that $[\![\alpha']\!]_A = [\![\alpha]\!]_{A_f}$.*

Proof. By induction on the structural complexity of α. By hypothesis there exists a scheme $\beta \in \mathrm{PR}(\Sigma)_{u,v}$ such that $[\![\beta]\!]_A = f_A$. Therefore, if we replace each basis scheme 'f' of α with β

to give α' then $[\![\alpha']\!]_A = [\![\alpha]\!]_{A_f}$ as required.

□

Compiling PR Schemes into Equations. As we will now show, by combining Lemma 12 with the technique of visualizing a PR scheme as a tree we have the basis for a recursive algorithm to compile PR schemes into equations. For example, notice that to equationally specify scheme α of our running example, it is sufficient to specify α' as defined above. Moreover, to do this we can firstly specify α' by a system of equations E_0 wherein f_2, f_4 and f_6 are basic operations; secondly specify f_2, f_4 and f_6 themselves with separate specifications E_1, E_2 and E_3 respectively; and finally derive the required specification E of α by merging E_0, E_1, E_2 and E_3. We now explain this 'divide-and-conquer' technique in more detail, by discussing each of its four constituent phases of operations individually. This is followed by a high-level algorithmic description of the whole compilation process.

First, given a scheme α we replace all top-level primitive recursions in α with new function symbols. The function that we define to formalize this procedure in the sequel is denoted ElimSubPR. One important feature of ElimSubPR is that as it is used recursively we wish to avoid clashes in the use of function symbol names that are used to replace applications of primitive recursion. For example, we would not want E_0 through E_3 to all specify the function symbol f_1. Therefore, throughout the compilation process we use a natural number index e, that is passed to ElimSubPR on each recursive application of $\mathbb{C}^{\mathrm{PREQ}}$, to enable us to avoid any potential name clashes.

Secondly, after we have applied ElimSubPR we use the main sub-function of $\mathbb{C}^{\mathrm{PREQ}}$ (denoted $\mathbb{C}^{\circ\mathrm{PREQ}}$) to either generate one or two equations depending on the structure of α. In particular, if the scheme α to be compiled into equations does not contain a primitive recursion after the top-level primitive recursions have been replaced then

$$\mathbb{C}_e^{\circ\mathrm{PREQ}}(\alpha) \stackrel{def}{=} f_e(x_1, \ldots, x_n) = \tau$$

wherein x_i are some variable symbols for $i = 1, \ldots, n \in \mathbb{N}$ and $\tau \in \mathrm{T}(\Sigma, \{x_1, \ldots, x_n\})$; otherwise if α does contain a primitive recursion after ElimSubPR has been applied then

$$\mathbb{C}_e^{\circ\mathrm{PREQ}}(\alpha) \stackrel{def}{=} \begin{array}{rcl} f_e(0, x_1, \ldots, x_n) & = & \tau \\ f_e(t+1, x_1, \ldots, x_n) & = & \tau' \end{array}$$

wherein x_i are some variable symbols for $i = 1, \ldots, n \in \mathbb{N}$, t is a distinguished variable symbol of type $\mathbf{n}$, $\tau \in \mathrm{T}(\Sigma, \{x_1, \ldots, x_n\})$ and $\tau' \in \mathrm{T}(\Sigma, \{t, x_1, \ldots, x_n, Y\})$. (The significance of the variable Y is explained in Section 5.3.2.)

Thirdly, $\mathbb{C}^{\mathrm{PREQ}}$ is applied recursively to the sub-schemes of α that have been replaced by ElimSubPR.

Finally, all the sets of equations produced are joined into one single specification. These four phases can be explained more rigorously using the following high-level algorithm.

Let α contain $k \in \mathbb{N}$ top-level primitive recursions and let $\alpha_i \subset \alpha$ such that scheme α_i has the ith top-level primitive recursion of α as its root node for $i = 1, \ldots, k$. Finally, let r^α be the function defined such that for each $i \in \{1, \ldots, k\}$ $r^\alpha(i) = j$ if and only if the ith top-level primitive recursion of α is the jth proper primitive recursion of α, and let the index value $e = 1$.

BEGIN

(1) For $i = 1, \ldots, k$ replace each sub-scheme α_i with function symbol $f_{r^\alpha(i)+e}$ and 'thin' all occurrences of compositions to derive scheme $\alpha' \in \mathrm{PR}_E(\Sigma)$. (Notice that if $k = 0$ then $\alpha = \alpha'$.)

(2) Let $E = \mathbb{C}_e^{\mathrm{PREQ}}(\alpha')$.

(3) Recursively repeat the compilation process on α_i for $i = 1, \ldots, k$ with $e_i = r^\alpha(i)$ as an index value to give the sets of equations E_i.

(4) Join E with $E_1, \ldots, E_k$.

END

Example 11. Using α as defined in Figure 5.1: at Step (1) we derive

$$\alpha' = *(f_{r^\alpha(1)+e}, \nabla(f_{r^\alpha(2)+e}, f_{r^\alpha(3)+e})) = *(f_2, \nabla(f_4, f_6)).$$

At Step (2) α' is compiled into the following set of equations E (say):

$$f_1(0, x) = f_2(x)$$

and

$$f_1(t+1, x) = f_4(f_6(t, x)).$$

At Step (3) we recursively repeat the procedure on schemes $\alpha_1 = \alpha^2$, $\alpha_2 = \alpha^8$ and $\alpha_3 = \alpha^{13}$ with indexes $e_1 = 2$, $e_2 = 4$ and $e_3 = 6$ respectively to give the sets of equations E_1, E_2 and E_3 that give specifications of the function symbols f_2, f_4 and f_6 respectively.

At Step (4) we join E with E_1, E_2 and E_3 to produce a single equational specification of α defined over function and constant symbols taken from Σ and the function symbols $f_1, \ldots, f_6$ representing each occurrence of an application of primitive recursion in α.

Discussion. The recursive structure of the algorithm defined above offers two advantages: firstly, in a uniform way we are able to compile PR scheme into minimal sets of equations in the sense defined previously; and secondly, the recursive structure of the algorithm enables us to verify its correctness using an inductive argument based on the number of occurrences of proper primitive recursions in a scheme.

The following three technical sections formalize the intuitive ideas behind the high-level algorithm that we have just described. The reader not interested in these technical definitions can now move directly to Section 5.3 on Page 137.

5.2.3 Section Overview

The first section formalizes the methods of counting particular occurrences of applications of primitive recursions in a scheme α and identifies particular classes of PR schemes based on these countings.

The second section formalizes the functions necessary to relate the indices of a given node in a scheme under different countings.

The final section gathers together these definitions to present the main lemma of this section (Lemma 22) that formalizes the intuitive idea of the construction of a normal form representation in PR.

5.2.4 Counting Primitive Recursions

In this section, as many of the lemmata are simple exercises in using the definitions, we will either omit proofs or sketch proofs and leave the details to the reader. Also, as a concession to conserving space we will present the definition of some functions defined by structural induction on the complexity of PR schema simultaneously. We also omit the proofs of the well-definedness of these constructions.

Definition 35. For each $\alpha \in \mathrm{PR}(\Sigma)$ we define

$$|.| : \mathrm{PR}(\Sigma) \to \mathbb{N}$$

by

$$|\alpha| = |\{\beta \mid \beta \subseteq \alpha\}|;$$

that is, $|\alpha|$ is the number of nodes in α.

We now present four functions that tell us the number of primitive recursive sub-schemes and proper primitive recursive sub-schemes of a scheme α.

Definition 36. We define the function

$$\mathrm{NPRSS} : \mathrm{PR}(\Sigma) \to \mathbb{N}$$

that counts the *N*umber of *P*rimitive *R*ecursive *S*ub-*S*chemes in a scheme α. However, for convenience we will also define the functions

$$\mathrm{NPRSS}', \mathrm{NPPRSS}, \mathrm{NPPRSS}' : \mathrm{PR}(\Sigma) \to \mathbb{N}$$

that we will require in the sequel. For each $\alpha \in \mathrm{PR}(\Sigma)$

$$\mathrm{NPRSS}(\alpha) = |\{\beta \mid \beta = *(\beta_1, \beta_2) \subseteq \alpha\}|,$$

$$\mathrm{NPRSS}'(\alpha) = \mathrm{NPRSS}(\alpha) - |\{\gamma \mid \gamma = *(\gamma_1, \gamma_2) \subset *(\delta_1, \delta_2) \subset \alpha\}|,$$

$$\mathrm{NPPRSS}(\alpha) = |\{\beta \mid \beta = *(\beta_1, \beta_2) \subset \alpha\}|,$$

and

$$\mathrm{NPPRSS}'(\alpha) = \mathrm{NPPRSS}(\alpha) - |\{\gamma \mid \gamma = *(\gamma_1, \gamma_2) \subset *(\delta_1, \delta_2) \subset \alpha\}|$$

respectively.

Thus: $\mathrm{NPRSS}(\alpha)$ is the number of sub-schemes of α that are primitive recursions; $\mathrm{NPRSS}'(\alpha)$ is the number of primitive recursive sub-schemes of α counting only the root of α and top-level primitive recursions; $\mathrm{NPPRSS}(\alpha)$ is the number of proper primitive recursive sub-schemes of α; and $\mathrm{NPPRSS}'(\alpha)$ is the number of top-level proper primitive recursions.

Lemma 13. *If $\alpha \in PR(\Sigma)$ then*

$$NPPRSS'(\alpha) \leq NPPRSS(\alpha).$$

In particular, if $NPPRSS(\alpha) = 0$ then $NPPRSS'(\alpha) = 0$ and if $NPPRSS(\alpha) = 1$ then $NPPRSS'(\alpha) = 1$.

Using the function

$$\mathrm{NPRSS} : \mathrm{PR}(\Sigma) \rightharpoonup \mathbb{N}$$

we can now define our class of normal forms $\mathrm{PR}_E(\Sigma)$. We do this via four intermediate subclasses of PR schemes as follows:

Definition 37. We define $\mathrm{PR}_A(\Sigma), \mathrm{PR}_B(\Sigma), \mathrm{PR}_C(\Sigma), \mathrm{PR}_D(\Sigma), \mathrm{PR}_E(\Sigma) \subset \mathrm{PR}(\Sigma)$ to be the restricted forms of PR schemes as follows: for each $\alpha \in \mathrm{PR}(\Sigma)$

$$\alpha \in \mathrm{PR}_A(\Sigma) \iff \mathrm{NPRSS}(\alpha) = 0,$$

$$\alpha \in \mathrm{PR}_B(\Sigma) \iff \alpha = *(\alpha_1, \alpha_2) \wedge (\mathrm{NPRSS}(\alpha_1) = \mathrm{NPRSS}(\alpha_2) = 0),$$

$$\alpha \in \mathrm{PR}_C(\Sigma) \iff \alpha \in \mathrm{PR}_A(\Sigma) \vee \alpha \in \mathrm{PR}_B(\Sigma),$$

$$\alpha \in \mathrm{PR}_D(\Sigma) \iff \alpha \in \mathrm{PR}_A(\Sigma) \wedge [(\forall \alpha' \subseteq \alpha) \quad \alpha' = \beta_2 \circ \beta_1 \iff |\mathrm{ran}(\beta_2)| = 1]$$

and

$$\alpha \in \mathrm{PR}_E(\Sigma) \iff \alpha \in \mathrm{PR}_C(\Sigma) \wedge [(\forall \alpha' \subseteq \alpha) \quad \alpha' = \beta_2 \circ \beta_1 \iff |\mathrm{ran}(\beta_2)| = 1].$$

Thus: $\mathrm{PR}_A(\Sigma)$ is the set of all *polynomial functions* over Σ; that is, PR schemes without an application of the primitive recursion operator; $\mathrm{PR}_B(\Sigma)$ is the set of all schemes over Σ with a single primitive recursion at the root node; $\mathrm{PR}_C(\Sigma)$ is the union of $\mathrm{PR}_A(\Sigma)$ and $\mathrm{PR}_B(\Sigma)$; $\mathrm{PR}_D(\Sigma)$ is $\mathrm{PR}_A(\Sigma)$ with the additional restriction that compositions cannot return vector types; and $\mathrm{PR}_E(\Sigma)$ is $\mathrm{PR}_C(\Sigma)$ with the additional restriction that compositions cannot return vector types.

Lemma 14. *For each $\alpha \in PR_E(\Sigma)$ if $NPRSS(\alpha) = 0$ then $\alpha \in PR_D(\Sigma)$. In particular, $PR_D(\Sigma) \subset PR_E(\Sigma)$ (see Definition 62 on Page 163).*

5.2.5 Indexing Nodes in PR Schemes

We now simultaneously define several functions that formalize the intuitive concepts discussed in the introduction to this section relating to the indexing of nodes in a scheme when it is visualized as a tree. The informal description of these functions is as follows.

Let $\alpha \in \mathrm{PR}(\Sigma)$ be any scheme: $\zeta_1^{e,p}(\alpha) = x + p - 1$ if and only if the xth node of α is the eth primitive recursive node of α; $\zeta_2^{e,p}(\alpha) = x + p - 1$ if and only if the xth node of α is the eth top-level primitive recursive node of α; $\xi_1^{b,p}(\alpha)$ replaces all top-level primitive recursions in α with function symbols $f_p, f_{p+1}, \ldots$; and $\xi_2^e(\alpha)$ returns the sub-scheme of α that has the eth node of α as it root.

Definition 38. For each $e \in \mathbb{N}$, for each $p \in \mathbb{N}^+$, for each $b \in \mathbb{B}$, for each $\mathcal{F}$ such that $\{f_{1,1}\} \subseteq \mathcal{F} \subset \{f_{n,n'} \mid n, n' \in \mathbb{N}\}$, for each $u \in S^+$ and for each $v \in S^+$ we define

$$\zeta_{1,u,v}^{e,p} : \mathrm{PR}(\Sigma)_{u,v} \rightarrow \mathbb{N},$$

$$\zeta_{2,u,v}^{e,p} : \mathrm{PR}(\Sigma)_{u,v} \rightarrow \mathbb{N},$$

$$\xi_{1,u,v}^{\mathcal{F},b,e} : \mathrm{PR}(\Sigma)_{u,v} \rightarrow \mathrm{PR}(\Sigma \cup \mathcal{F})$$

$$\xi_{2,u,v}^{e} : \mathrm{PR}(\Sigma)_{u,v} \rightarrow \mathrm{PR}(\Sigma)$$

(ambiguously denoted $\zeta_1^{e,p}$, $\zeta_2^{e,p}$, $\xi_1^{b,e}$ and ξ_2^e respectively) uniformly in (u, v) by induction on the structural complexity of an argument $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ as follows:

Basis Schema.

(1) Constant Functions, (2) Algebraic Operations, (3) Projections and (4) Definition-by-Cases. If either $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$, for some $w \in S^+$ and for some $s \in S$ or $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$, for some $w \in S^+$ and for some $s \in S$ or $\alpha = \mathrm{U}_i^w$, for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$ or $\alpha = \mathrm{dc}_s$ for some $s \in S$ then

$$\zeta_{1,w,w_+}^{e,p}(\alpha) = \zeta_{2,w,w_+}^{e,p}(\alpha) = 0;$$

and

$$\xi_{1,w,w_+}^{\mathcal{F},b,e}(\alpha) = \xi_{2,w,w_+}^{e}(\alpha) = \alpha.$$

Induction.

(5) Vectorization. If $\alpha = < \alpha_1, \ldots, \alpha_m >$ wherein $\alpha_i \in \mathrm{PR}(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1, \ldots, m$ then

$$\zeta_{1,u,s_1 \cdots s_m}^{e,p}(\alpha) = \zeta_1^{k,k'}(\alpha_i)$$

wherein

$$i = \begin{cases} \mu(l \leq m)[e \leq \sum_{j=1}^{j=l} \mathrm{NPRSS}(\alpha_j)] & \text{if } \sum_{j=1}^{j=m} \mathrm{NPRSS}(\alpha_j) \geq e, \\ m & \text{otherwise;} \end{cases}$$

$$k = e - \sum_{j=1}^{j=i-1} \mathrm{NPRSS}(\alpha_j)$$

and

$$k' = p + 1 + \sum_{j=1}^{j=i-1} |\alpha_j|,$$

$$\zeta_{2,u,s_1 \cdots s_m}^{e,p}(\alpha) = \zeta_2^{k,k'}(\alpha_i)$$

wherein

$$i = \begin{cases} \mu(l \leq m)[e \leq \sum_{j=1}^{j=l} \mathrm{NPRSS}'(\alpha_j)] & \text{if } \sum_{j=1}^{j=m} \mathrm{NPRSS}'(\alpha_j) \geq e, \\ m & \text{otherwise;} \end{cases}$$

$$k = e - \sum_{j=1}^{j=i-1} \mathrm{NPRSS}'(\alpha_j)$$

and

$$k' = p + 1 + \sum_{j=1}^{j=i-1} |\alpha_j|,$$

$$\xi^{\mathcal{F},b,e}_{1,u,s_1\cdots s_m}(\alpha) = \begin{cases} < \xi_1^{ff,k_1+1}(\alpha_1), \ldots, \xi_m^{ff,k_m+1}(\alpha_m) > & \text{if } b = tt, \\ < \xi_1^{ff,k_1}(\alpha_1), \ldots, \xi_m^{ff,k_m}(\alpha_m) > & \text{otherwise} \end{cases}$$

wherein for $i = 1, \ldots, m$, $k_i = e + \sum_{j=1}^{j=i-1} \mathrm{NPRSS}(\alpha_j)$; and

$$\xi^{e}_{2,u,s_1\cdots s_m}(\alpha) = \begin{cases} \alpha & \text{if } e \leq 1, \\ \xi_2^k(\alpha_i) & \text{otherwise} \end{cases}$$

wherein

$$i = \begin{cases} \mu(l < m)[\sum_{j=1}^{j=l} |\alpha_j| \geq e - 1] & \text{if } \sum_{j=1}^{j=m} |\alpha_j| \geq e - 1, \\ m & \text{otherwise} \end{cases}$$

and

$$k = e - \sum_{j=1}^{j=i-1} |\alpha_j| - 1$$

respectively.

(6) Composition. If $\alpha = \alpha_2 \circ \alpha_1$ wherein $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,v}$ for some $u, v, w \in S^+$ then

$$\zeta^{e,p}_{1,u,v}(\alpha) = \begin{cases} \zeta_1^{e,p+1}(\alpha_2) & \text{if } \mathrm{NPRSS}(\alpha_1) \geq e, \\ \zeta_1^{k,l}(\alpha_1) & \text{otherwise} \end{cases}$$

wherein $k = e - \mathrm{NPRSS}(\alpha_1)$ and $l = p + 1 + |\alpha_2|$;

$$\zeta^{e,p}_{2,u,v}(\alpha) = \begin{cases} \zeta_2^{e,p+1}(\alpha_2) & \text{if } \mathrm{NPRSS}'(\alpha_1) \geq e, \\ \zeta_2^{k,l}(\alpha_1) & \text{otherwise} \end{cases}$$

wherein $k = e - \mathrm{NPRSS}'(\alpha_1)$ and $l = p + 1 + |\alpha_2|$;

$$\xi^{\mathcal{F},b,e}_{1,u,v}(\alpha) = \begin{cases} \xi_1^{ff,e+1}(\alpha_2) \circ \xi_m^{ff,k+1}(\alpha_1) & \text{if } b = tt \\ \xi_1^{ff,e}(\alpha_2) \circ \xi_m^{ff,k}(\alpha_1) & \text{otherwise} \end{cases}$$

wherein $k = e + \mathrm{NPRSS}(\alpha_2)$; and

$$\xi^{e}_{2,u,v}(\alpha) = \begin{cases} \alpha & \text{if } e \leq 1, \\ \xi_2^{e-1}(\alpha_2) & \text{if } 1 < e \leq |\alpha_2|, \\ \xi_2^{e-|\alpha_2|-1}(\alpha_1) & \text{otherwise} \end{cases}$$

respectively.

(7) Simultaneous Primitive Recursion. If $\alpha = *(\alpha_1, \alpha_2)$ where $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{\mathrm{n}\,u\,v,v}$ for some $u, v \in S^+$ then

$$\zeta_{1,u,v}^{e,p}(\alpha) = \begin{cases} p & \text{if } e \leq 1, \\ \zeta_1^{e-1,p+1}(\alpha_1) & \text{if } e > 1 \wedge \mathrm{NPRSS}(\alpha_1) \geq e-1, \\ \zeta_1^{k,l}(\alpha_2) & \text{otherwise} \end{cases}$$

wherein $k = e - 1 - \mathrm{NPRSS}(\alpha_1)$ and $l = p + 1 + |\alpha_1|$;

$$\zeta_{2,u,v}^{e,p}(\alpha) = \begin{cases} p & \text{if } e \leq 1, \\ \zeta_2^{e-1,p+1}(\alpha_1) & \text{if } e > 1 \wedge \mathrm{NPRSS}'(\alpha_1) \geq e-1, \\ \zeta_2^{k,l}(\alpha_2) & \text{otherwise} \end{cases}$$

wherein $k = e - 1 - \mathrm{NPRSS}'(\alpha_1)$ and $l = p + 1 + |\alpha_1|$;

$$\xi_{1,u,v}^{\mathcal{F},b,e}(\alpha) = \begin{cases} *(\xi_1^{ff,e+1}(\alpha_1), \xi_1^{ff,k}(\alpha_2)) & \text{if } b = tt, \\ f_{e,1} & \text{if } b = ff,\ f_{e,1} \in \mathcal{F} \text{ and } |v| = 1, \\ < f_{e,1}, \ldots, f_{e,|v|} > & \text{if } b = ff,\ f_{e,1}, \ldots, f_{e,|v|} \in \mathcal{F} \text{ and } |v| > 1, \\ f_{1,1} & \text{otherwise} \end{cases}$$

wherein $k = e + \mathrm{NPRSS}(\alpha_1) + 1$; and

$$\xi_{2,u,v}^{e}(\alpha) = \begin{cases} \alpha & \text{if } e \leq 1, \\ \xi_2^{e-1}(\alpha_1) & \text{if } 1 < e \leq |\alpha_1|, \\ \xi_2^{e-|\alpha_1|-1}(\alpha_2) & \text{otherwise} \end{cases}$$

respectively.

5.2.6 The Formal Definition and Correctness of the Normal Form Compiler

We now gather together the definitions of the previous two sections to formalize the remaining ideas presented in the introduction to this section.

We first formalize the idea of identifying a particular sub-scheme of a scheme α relative to some index value.

Definition 39. For each $u, v \in S^+$ we define

$$\mathrm{SubSch}_{u,v} : \mathbb{N} \times \mathrm{PR}(\Sigma)_{u,v} \rightarrow \mathrm{PR}(\Sigma)$$

(ambiguously denoted SubSch) by

$$(\forall n \in \mathbb{N})(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v}) \quad \mathrm{SubSch}(n, \alpha) = \begin{cases} \alpha & \text{if } n = 0 \vee n > |\alpha| \\ \xi_2^n(\alpha) & \text{otherwise.} \end{cases}$$

Thus, $\mathrm{SubSch}(n, \alpha)$ returns α if $n = 0$ or $n > |\alpha|$ and the sub-scheme of α that has the nth node of α as its root node otherwise.

Definition 40. For each $u, v \in S^+$ we define

$$\mathrm{PRSS}_{u,v} : \mathrm{PR}(\Sigma)_{u,v} \times \mathbb{N} \rightharpoonup \mathrm{PR}(\Sigma)$$

(ambiguously denoted PRSS) by

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v})\,(\forall n \in \mathbb{N})$$

$$\mathrm{PRSS}(\alpha, n) = \begin{cases} \mathrm{SubSch}(\zeta_1^{n,1}(\alpha), \alpha) & \text{if } n < k; \text{ and} \\ \mathrm{SubSch}(\zeta_1^{k,1}(\alpha), \alpha) & \text{otherwise} \end{cases}$$

wherein $k = \mathrm{NPRSS}(\alpha)$. Thus, $\mathrm{PRSS}(n, \alpha)$ is the sub-scheme of α with the nth primitive recursive node of α as its root node.

Definition 41. For each $u, v \in S^+$ we define

$$\mathrm{PPRSS}_{u,v} : \mathrm{PR}(\Sigma)_{u,v} \times \mathbb{N} \rightharpoonup \mathrm{PR}(\Sigma)$$

(ambiguously denoted PPRSS) by

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v})\,(\forall n \in \mathbb{N})$$

$$\mathrm{PPRSS}(\alpha, n) = \begin{cases} \mathrm{SubSch}(\zeta_1^{n+1,1}(\alpha), \alpha) & \text{if } \alpha = *(\alpha_1, \alpha_2) \text{ and } n \leq k; \\ \mathrm{SubSch}(\zeta_1^{n,1}(\alpha), \alpha) & \text{if } \alpha \neq *(\alpha_1, \alpha_2) \text{ and } n \leq k; \text{ and} \\ \mathrm{SubSch}(\zeta_1^{k,1}(\alpha), \alpha) & \text{otherwise} \end{cases}$$

wherein $k = \mathrm{NPPRSS}(\alpha)$. Thus, $\mathrm{PPRSS}(n, \alpha)$ is the sub-scheme of α with the nth proper primitive recursive node of α as its root node.

Definition 42. For each $u, v \in S^+$ we define

$$\mathrm{PPRSS}'_{u,v} : \mathrm{PR}(\Sigma)_{u,v} \times \mathbb{N} \rightharpoonup \mathrm{PR}(\Sigma)$$

(ambiguously denoted PPRSS′) by

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v})\,(\forall n \in \mathbb{N})$$

$$\mathrm{PPRSS}'(\alpha, n) = \begin{cases} \mathrm{SubSch}(\zeta_2^{n+1,1}(\alpha), \alpha) & \text{if } \alpha = *(\alpha_1, \alpha_2) \text{ and } n \leq k; \\ \mathrm{SubSch}(\zeta_2^{n,1}(\alpha), \alpha) & \text{if } \alpha \neq *(\alpha_1, \alpha_2) \text{ and } n \leq k; \text{ and} \\ \mathrm{SubSch}(\zeta_2^{k,1}(\alpha), \alpha) & \text{otherwise} \end{cases}$$

wherein $k = \mathrm{NPPRSS}'(\alpha)$. Thus, $\mathrm{PPRSS}'(n, \alpha)$ is the sub-scheme of α that has the nth top-level primitive recursive node of α as its root node.

Lemma 15. *Let $\alpha \in PR(\Sigma)$.*
(1) If $i \in \{1, \ldots, NPPRSS(\alpha)\}$ then

$$NPPRSS(PPRSS(\alpha, i)) < NPPRSS(\alpha).$$

(2) If $i \in \{1, \ldots, NPPRSS'(\alpha)\}$ then

$$NPPRSS(PPRSS'(\alpha, i)) < NPPRSS(\alpha).$$

Lemma 16. *Let $\alpha \in PR(\Sigma)$ such that $NPPRSS(\alpha) \geq 1$. If $i \in \{1, \ldots, NPPRSS(\alpha)\}$ then for $j = 1, \ldots, NPPRSS(PPRSS(\alpha, i))$*

$$PPRSS(\alpha, i + j) = PPRSS(PPRSS(\alpha, i), j).$$

We now formalize the definition that relates the values of the index of a sub-scheme according to a counting of proper primitive recursions and top-level primitive recursions.

Definition 43. For each $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ such that $\mathrm{NPPRSS}'(\alpha) \geq 1$ we define

$$r^\alpha : \{1, \ldots, \mathrm{NPPRSS}'(\alpha)\} \rightarrow \{1, \ldots, \mathrm{NPPRSS}(\alpha)\}$$

(ambiguously denoted r) by

$$r^\alpha = \bigcup_{i=1}^{i=\mathrm{NPPRSS}'(\alpha)} \begin{cases} \{i \mapsto \mu\, n.[\zeta_2^{i+1,1}(\alpha) = \zeta_1^{n+1,1}(\alpha)]\} & \text{if } \alpha = *(\alpha_1, \alpha_2), \text{ and} \\ \{i \mapsto \mu\, n.[\zeta_2^{i,1}(\alpha) = \zeta_1^{n,1}(\alpha)]\} & \text{otherwise.} \end{cases}$$

Thus, for each scheme $\alpha \in \mathrm{PR}(\Sigma)$ if $\alpha' = \mathrm{PPRSS}'(\alpha, i)$ for some $i \in \{1, \ldots, \mathrm{NPPRSS}'(\alpha)\}$ then $\alpha' = \mathrm{PPRSS}(\alpha, r(i))$.

Example 12. If

$$\alpha = *(*(*(\beta_1, \beta_2), \beta_3), *(\beta_4, *(\beta_5, \beta_6))) \circ *(\beta_7, \beta_8)$$

for some schema β_i for $i = 1, \ldots, 8$ (see Figure 5.1) then

$$r^\alpha : \{1, 2, 3\} \rightarrow \{1, 2, 3, 4, 5\}$$

is defined by

$$r^\alpha = \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}.$$

Lemma 17. *Let $\alpha \in PR(\Sigma)$. If $NPPRSS'(\alpha) \geq 1$ then*

$$r^\alpha(1) = 1.$$

Lemma 18. *Let $\alpha \in PR(\Sigma)$. If $n = NPPRSS'(\alpha) \geq 2$ then*
(1) For each $i \in \{1, \ldots, n-1\}$

$$r^\alpha(i) + NPPRSS(PPRSS'(\alpha, i)) + 1 = r^\alpha(i + 1).$$

(2)

$$r^\alpha(n) + NPPRSS(PPRSS'(\alpha, n)) = NPPRSS(\alpha).$$

Finally, we are now in a position to make a formal definition of the function ElimSubPR that eliminates top-level primitive recursion from a scheme and hence to make the formal definition of the compiler $\mathbb{C}^{\mathrm{PR}_E}$. In particular, notice that as the number of additional function symbols that will be required to replace the top-level primitive recursions in a scheme $\alpha \in \mathrm{PR}(\Sigma)$ cannot be determined in advance we are forced to define the co-domain of our formal compiler ElimSubPR to be the union of all PR schemes defined over Σ extended with all possible sub-sets of the set $\{f_{i,j} \mid i,j \in \mathbb{N}\}$. However, notice that from the perspective of developing software tools based on ElimSubPR, by defining ElimSubPR using the function ξ once given a scheme α we can determine precisely which finite sub-set of $\{f_{i,j} \mid i,j \in \mathbb{N}\}$ we require using the functions PPRSS′ and r^{α}. This idea is made precise in Lemma 19.

Definition 44. For each $u, v \in S^+$ we define

$$\mathrm{ElimSubPR}_{u,v} : \mathrm{PR}(\Sigma)_{u,v} \times \mathbb{N} \rightarrow \bigcup_{\Sigma' \subset \mathbb{G}} \mathrm{PR}(\Sigma \cup \Sigma')$$

wherein

$$\mathbb{G} = \{f_{i,j} \mid i,j \in \mathbb{N}\}$$

(ambiguously denoted ElimSubPR) by

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v})\,(\forall n \in \mathbb{N}) \quad \mathrm{ElimSubPR}(\alpha, n) = \xi_1^{\mathcal{F},tt,n}(\alpha)$$

wherein

$$\mathcal{F} = \bigcup_{i=1}^{i=\mathrm{NPPRSS}'(\alpha)} \{f_{r^{\alpha}(i)+n,1}, \ldots, f_{r^{\alpha}(i)+n,|v^i|}\}$$

and $v^i = \mathrm{ran}(\mathrm{PPRSS}'(\alpha, i))$ for $i = 1, \ldots, \mathrm{NPPRSS}'(\alpha)$.

Lemma 19. *Let $\alpha \in PR(\Sigma)_{u,v}$ for some $u, v \in S^+$. For any $e \in \mathbb{N}$*

$$ElimSubPR(\alpha, n) \in PR(\Sigma \cup \mathcal{F})$$

wherein

$$\mathcal{F} = \bigcup_{i=1}^{i=NPPRSS'(\alpha)} \{f_{r^{\alpha}(i)+e,1}, \ldots, f_{r^{\alpha}(i)+e,|v^i|}\}.$$

This lemma is used implicitly in the following results.

Lemma 20. *Let $\alpha \in PR(\Sigma)_{u,v}$ for some $u, v \in S^+$. For any $e \in \mathbb{N}$ if either $NPPRSS'(\alpha) = 0$ or for each $i = 1, \ldots, NPPRSS'(\alpha)$ and for each $j = 1, \ldots, |ran(PPRSS(\alpha, i))|$ the function $f_{r^{\alpha}(i)+e,j}$ is defined over A by $f^A_{r^{\alpha}(i)+e,j} = \left([\![PPRSS'(\alpha, i))]\!]_A\right)_j$ then*

$$[\![\alpha]\!]_A = [\![ElimSubPR(\alpha, e)]\!]_{A_{\mathcal{F}}}$$

wherein $\mathcal{F}$ is defined as in Lemma 19.

Proof. By induction on the structural complexity of α using Lemma 12 and the definition of ElimSubPR.

□

Definition 45. For each $u, v \in S^+$ and for each $e \in \mathbb{N}$ we define

$$\mathbb{C}^{\mathrm{PR}_E}_{e,u,v} : \mathrm{PR}(\Sigma)_{u,v} \to \bigcup_{\Sigma' \subset \mathbb{G}} \mathrm{PR}_E(\Sigma \cup \Sigma')_{u,v}$$

wherein

$$\mathbb{G} = \{f_{i,j} \mid i, j \in \mathbb{N}\}$$

(ambiguously denoted $\mathbb{C}^{\mathrm{PR}_E}_e$) by

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v}) \quad \mathbb{C}^{\mathrm{PR}_E}_e(\alpha) = \mathrm{Thin}(\mathrm{ElimSubPR}(\alpha, e)).$$

Lemma 21. *If* $\alpha = *(\alpha_1, \alpha_2) \in PR(\Sigma)$ *and* $\alpha' = *(\alpha'_1, \alpha'_2) = \mathbb{C}^{PR_E}_e(\alpha)$ *for some* $e \in \mathbb{N}$ *then* $NPRSS(\alpha'_1) = NPRSS(\alpha'_2) = 0$.

Proof. By definition of $\mathbb{C}^{\mathrm{PR}_E}$.

□

Lemma 22. *Let* $\alpha \in PR(\Sigma)_{u,v}$ *for some* $u, v \in S^+$. *For any* $e \in \mathbb{N}$ *if either* $NPPRSS'(\alpha) = 0$ *or for each* $i = 1, \ldots, NPPRSS'(\alpha)$ *and for each* $j = 1, \ldots, |ran(PPRSS(\alpha, i))|$ *the function* $f_{r^\alpha(i)+e,j}$ *is defined over* A *by* $f^A_{r^\alpha(i)+e,j} = \left([\![PPRSS'(\alpha, i))]\!]_A\right)_j$ *then*

$$[\![\alpha]\!]_A = [\![\mathbb{C}^{PR_E}_e(\alpha)]\!]_{A_{\mathcal{F}}}$$

wherein

$$\mathcal{F} = \bigcup_{i=1}^{i=NPPRSS'(\alpha)} \{f_{r^\alpha(i)+n,1}, \ldots, f_{r^\alpha(i)+n,|v'|}\}.$$

Proof. By the definition of $\mathbb{C}^{\mathrm{PR}_E}$, Lemma 20 and Lemma 11.

□

5.3 Primitive Recursive Equational Specification

We can now begin the first stage of the development of our abstract specification language for STs. As we have indicated, in order that we preserve a sufficient level of mathematical abstraction ASTRAL's semantics is formulated denotationally and will be derived using essentially nothing more than a first-order equational formalism for making primitive recursive definitions.

The language PREQ that we develop for this purpose is distinct from existing equational specification languages such as OBJ (see Goguen [1987], Goguen and Winkler [1988], and Goguen *et al.* [1992]) in that it is deliberately and syntactically restricted to capture the class $\mathbb{PR}$; that is, it does not as a matter of course provide a general model of computation (see Section 4.4.2).

We will show that PREQ is sound and adequate with respect to the language PR by defining two appropriate compilers. As a consequence we may constructively apply Theorem 7 (see Section 4.5.1) in order that we may compose CFSTs when specified in PREQ (see Section 3.10).

5.3.1 Overview

We begin by defining the abstract syntax of the equational specification language *RPREQ* that can represent a restricted class of primitive recursive functions. Secondly, we define the abstract syntax of PREQ in terms of RPREQ. Thirdly, we define the formal semantics of RPREQ and PREQ. Finally, we make the formal compiler definitions that map between PREQ and PR and hence provide a constructive proof of Theorem 10. However, for convenience of presentation, the formal proofs of correctness of our compilers are given in Appendix B.

5.3.2 The Syntax and Semantics of RPREQ and PREQ

We now introduce the formal abstract syntax and semantics of RPREQ and PREQ.

Abstract Syntax of RPREQ. Let Σ be any standard S-sorted signature and let X be any S-indexed collection of variables satisfying the conditions set out in Section 4.2.4. We define the $S^+ \times S^+$-indexed family of *restricted primitive recursive equational specifications*

$$\mathrm{RPREQ}(\Sigma, X) = < \mathrm{RPREQ}(\Sigma, X)_{u,v} | u, v \in S^+ >$$

wherein for each $u, v \in S^+$ the set $\mathrm{RPREQ}(\Sigma, X)_{u,v}$ is defined uniformly in (u,v) by one of the following cases: (throught the following definitions we use $\mathbb{X}$ to denote the set $\{x_1, \ldots, x_n\}$ wherein $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, n \geq 1$)

(1) Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma, \mathbb{X})_s$ for some $s \in S$ then

$$\phi \in \mathrm{RPREQ}(\Sigma, X)_{s_1 \cdots s_n, s}.$$

(2) Vector-Valued Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = < \tau_1, \ldots, \tau_m >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma, \mathbb{X})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ then

$$\phi \in \mathrm{RPREQ}(\Sigma, X)_{s_1 \cdots s_n, s'_1 \cdots s'_m}.$$

(3) Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_1 \in T(\Sigma, \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma, \mathbb{X}')_s$, for some $s \in S$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y\}$ such that $t \in X_{\mathbf{n}}$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\phi \in \mathrm{RPREQ}(\Sigma, X)_{t\, s_1 \cdots s_n, s}.$$

(4) **Vector-Valued Primitive Recursive Specifications.** If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = < \tau_{1,1}, \ldots, \tau_{1,m} >;$$
$$f(t+1, x_1, \ldots, x_n) = < \tau_{2,1}, \ldots, \tau_{2,m} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_{1,j} \in T(\Sigma, \mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma, \mathbb{X}')_{s'_j}$, for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1, \ldots, Y_m\}$ such that $t \in X_{\mathbf{n}}$ and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, m$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\phi \in \mathrm{RPREQ}(\Sigma, X)_{t\, s_1 \cdots s_n, s'_1 \cdots s'_m}.$$

Discussion. It is our intention to formalize the semantics of a specification $\phi \in \mathrm{RPREQ}(\Sigma, X)_{u,v}$ for some $u, v \in S^+$ in some algebra A to give a primitive recursive function

$$f^A = (f_1^A, \ldots, f_{|v|}^A) : A^u \to A^v.$$

In order to ensure that function f^A is indeed primitive recursive when ϕ is defined by Case (3) and Case (4) we restrict the interpretation of ϕ as follows: in the context of Case (3) (Case (4) is similar) for each $a = (a_1, \ldots, a_{|u|}) \in A^u$ if $a_1 = 0$ then the value of $[\![\phi]\!]_A(a)$ is the value of τ_1 in A under the assignment of $a_1, a_2, \ldots, a_{|u|}$ to the variables $t, x_1, \ldots, x_n$ respectively; otherwise if $a_1 > 0$ then the value of $[\![\phi]\!]_A(a)$ is the value of τ_2 in A under the assignment of $a_1 - 1, a_2, \ldots, a_{|u|}$ to the variables $t, x_1, \ldots, x_n$ respectively and variable Y has the value $[\![\phi]\!]_A(a_1 - 1, a_2, \ldots, a_{|u|})$. We illustrate this idea in the following example:

Example 13. Let $X \supseteq X_{\mathbf{n}} \supseteq \{x_1\}$. The RPREQ scheme:

$$\phi \stackrel{def}{=} f(0, x_1) = x_1;$$
$$f(t+1, x_1) = Succ(Y)$$

represents the addition function on the natural numbers.

The use of the variable Y to represent $f(t, x_1)$ (and more generally Y_i to represent $f_i(t, x_2, \ldots, x_n)$) in this way may at first sight seem rather unnatural. However, it is important to remember at this stage that we are designing an abstract theoretical tool not an implementation language. As such the use of the distinguished variable Y in this way allows us to avoid the possible complication with unrestricted equations that $f(t+1, a_1, \ldots, a_n)$ may be defined (either directly or indirectly) in terms of $f(t+d, a'_1, \ldots, a'_n)$ for some $d \geq 1$ and for some a'_i such that $a'_i \neq a_i$ for $i = 1, \ldots, n$. From this perspective the use of the variable Y provides a concise syntactic method to control the class of functions that we may specify. Indeed, as we will show this choice of syntax greatly simplifies the formulation of a denotational semantics for PREQ without adversely affecting the design of a user-friendly syntax for the implementation of ASTRAL (see Definition 77 on Page 191).

The Abstract Syntax of PREQ. We continue our equational formalization of the primitive recursive functions by introducing the idea of a family of mutually dependent RPREQ specifications; that is, a PREQ specification. Essentially, the formulation of PREQ is based on the definition of the class $\mathbb{PR}$ as the union of all classes of functions in the Gregorczyk Hierarchy (see Grzegorczyk [1953] and also Röddling [1964], Marchenkov [1969] and Kozmidiadi and Marchenkov [1969]); that is,

$$\mathbb{PR} = \bigcup_{n \in \mathbb{N}} \mathcal{E}^n$$

wherein $\mathcal{E}^n$ is the class of all n-ary functions closed under n nested applications of bounded (primitive) recursion. Informally, we can think of the Gregorczyk Hierarchy as being the successive classes of functions that are computable by increasing the number of nested 'for loops' in an imperative programming language.

Based on this idea a PREQ specification is comprised of a number $l \geq 1$ of RPREQ specifications defined over a common signature Σ extended with extra function symbols $f_{i,1}, \ldots, f_{i,n_i}$ for some $n_i \in \mathbb{N}^+$ for $i = 1, \ldots, m \geq l$. Each extra function symbol $f_{i,j}$ is interpreted as the semantics of co-ordinate j of some RPREQ scheme $\phi_{\iota(i)}$ wherein $\iota : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, l\}$ is some given injection such that $\iota(i)\downarrow$, that tells us which function symbol is to be interpreted by which RPREQ specification. A semantics is given to the PREQ specification itself by choosing a particular RPREQ scheme ϕ_ς for some $\varsigma \in \{1, \ldots, l\}$ as representing the 'main function' that is being specified.

This method of formalizing the PREQ syntax provides a sound basis for the equational formalization of the class $\mathbb{PR}$. However, unfortunately by itself this method is still not sufficient to ensure that a PREQ specification does indeed define a primitive recursive function. In order to deal with this complication we first define the class of *pre-PREQ specifications* (Definition 46) denoted PREQ_0, based on the informal explanation given above, and use the function InTermsOf (Definition 49) to syntactically test for a PREQ_0 specifications 'primitive recursiveness'. Specifically, given a PREQ_0 specification comprised of l component RPREQ specifications and a number $j \in \{1, \ldots, l\}$ the function InTermsOf tells us which other RPREQ specifications the jth RPREQ specification's definition depends upon. To do this the function InTermsOf uses the function DefOver (Definition 48) to tell which additional functions symbols from the set $\{f_{i,1}, \ldots, f_{i,n_i}\}$ for some $n_i \in \mathbb{N}^+$ for $i = 1, \ldots, m \geq l$ occur in the term(s) over which the jth RPREQ specification is defined. If for each $j \in \{1, \ldots, l\}$ the RPREQ specification j is not defined in terms of itself by the use of the additional function symbols then the PREQ_0 specification potentially represents a primitive recursive function and is admitted as a member of the class of *partial* PREQ denoted PREQ_1 (Definition 50). We use the phrase 'partial' here as in order to be a *total* PREQ specification we must further ensure that each additional function symbol is indeed interpreted by a particular RPREQ specification.

The class of total PREQ specifications is presented in Definition 51. For convenience we also identify a further sub-class of total PREQ specifications that we refer to as *standard* PREQ specifications in Definition 52.

These ideas are formalized as follows:

Pre-PREQ Specifications.

Definition 46. For some $l, m \in \mathbb{N}^+$, for some $\varsigma \in \{1, \ldots, l\}$, for some injection $\iota : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, l\}$ and for some mapping $\eta = (\eta^D, \eta^R) : \{1, \ldots, m\} \rightarrow S^+ \times S^+$ if $\phi_i \in \mathrm{RPREQ}(\Sigma', \mathbb{X})_{u^i, v^i}$ for some $u^i, v^i \in S^+$ for $i = 1, \ldots, l$ wherein $\Sigma' = \Sigma \cup \mathcal{F}$ and $\mathcal{F}$ is defined by

$$\mathcal{F} = \bigcup_{i=1}^{i=m} \bigcup_{j=1}^{j=|\eta^R(i)|} \cup_{\eta^D(i),(\eta^R(i))_j} \{f_{i,j}\}$$

then we say that

$$\Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma >$$

is a *pre-PREQ specification* of *size* l, *scope* m and *type* $(u^\varsigma, v^\varsigma)$ denoted $\Phi \in \mathrm{PREQ}_0(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u^\varsigma, v^\varsigma}$ (and sometimes ambiguously just $\Phi \in \mathrm{PREQ}_0(\Sigma, X)_{u^\varsigma, v^\varsigma}$).

Checking Pre-PREQ Specifications for Primitive Recursiveness. The function DefOver, that determines which function symbols from the family $\mathcal{F}$ a particular RPREQ specification is defined over, uses a further sub-function TermsDefOver defined as follows:

Definition 47. For each $s \in S$ and for each $\mathcal{F} \subset \{f_{n,n'} \mid n, n' \in \mathbb{N}\}$

$$\mathrm{TermsDefOver}^{\mathcal{F}}_s : T(\Sigma \cup \mathcal{F}, X)_s \times \wp(\{f_{n,n'} \mid n, n' \in \mathbb{N}\}) \rightarrow \wp(\mathbb{N})$$

(ambiguously denoted TermsDefOver) is defined for each $\mathbb{F} \in \wp(\{f_{n,n'} \mid n, n' \in \mathbb{N}\})$ uniformly in s by induction on the structural complexity of a term $\tau \in T(\Sigma \cup \mathcal{F}, X)_s$ as follows:
Basis.

(1) Constants. If $\tau = c$ for some $c \in \Sigma_{\lambda,s}$ for any $s \in S$ then

$$\mathrm{TermsDefOver}_s(\tau, \mathbb{F}) = \varnothing.$$

(2) Variables. If $\tau = x$ for some $x \in X_s$ for any $s \in S$ then

$$\mathrm{TermsDefOver}_s(\tau, \mathbb{F}) = \varnothing.$$

Induction.

(3) Algebraic Operations. If $\tau = \sigma(\tau_1, \ldots, \tau_{|w|})$ for some $\sigma \in \{\Sigma \cup \mathcal{F}\}_{w,s}$ for any $w \in S^+$, and for any $s \in S$, and for some $\tau_i \in T(\Sigma \cup \mathcal{F}, X)_w$, for $i = 1, \ldots, |w|$ then

$$\mathrm{TermsDefOver}_s(\tau, \mathbb{F}) = \begin{cases} \{k\} \bigcup_{i=1}^{i=|w|} \mathrm{TermsDefOver}(\tau_i, \mathbb{F}) & \text{if } \sigma = f_{k,l} \in \mathbb{F} \text{ for some } k, l \in \mathbb{N} \\ \bigcup_{i=|w|}^{i=1} \mathrm{TermsDefOver}(\tau_i, \mathbb{F}) & \text{otherwise.} \end{cases}$$

Thus, given a term $\tau \in T(\Sigma \cup \mathcal{F}, X)_s$ and an $\mathbb{F} \in \wp(\{f_{n,n'} \mid n, n' \in \mathbb{N}\})$, the result of TermsDefOver$(\tau, \mathbb{F})$ is the first index of all the function symbols $f_{i,j} \in \mathcal{F}$ such that $f_{i,j} \in \mathbb{F}$ and $f_{i,j}$ occurs in τ.

Example 14. If

$$\mathcal{F} = \{f_{1,1}, \ldots, f_{1,5}, f_{2,1}, \ldots, f_{2,3}\},$$

$$\tau = Add(Succ(f_{1,4}(x_1)), f_{2,3}(x_2, x_3))$$

and

$$\mathbb{F} = \{f_{1,1}, \ldots, f_{1,5}\}$$

then

$$\text{TermsDefOver}(\tau, \mathbb{F}) = \{1\}.$$

Using TermsDefOver it is straightforward to define DefOver:

Definition 48. For each $u, v \in S^+$ and for each $\mathcal{F} \subset \{f_{n,n'} \mid n, n' \in \mathbb{N}\}$

$$\text{DefOver}^{\mathcal{F}}_{u,v} : \text{RPREQ}(\Sigma \cup \mathcal{F}, X)_{u,v} \times \wp(\{f_{n,n'} \mid n, n' \in \mathbb{N}\}) \to \wp(\mathbb{N})$$

(ambiguously denoted DefOver) is defined for each $\mathbb{F} \in \wp(\{f_{n,n'} \mid n, n' \in \mathbb{N}\})$ by the structural complexity of a specification $\phi \in \text{RPREQ}(\Sigma \cup \mathcal{F}, X)_{u,v}$ as follows: (throught the following definitions we use $\mathbb{X}$ to denote the set $\{x_1, \ldots, x_n\}$ wherein $x_i \in X_s$, for some $s_i \in S$ for $i = 1, \ldots, n \geq 1$)

(1) Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma \cup \mathcal{F}, \mathbb{X})_s$ for some $s \in S$ then

$$\text{DefOver}_{s_1 \cdots s_n, s}(\phi, \mathbb{F}) = \text{TermsDefOver}(\tau, \mathbb{F}).$$

(2) Vector-Valued Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = < \tau_1, \ldots, \tau_m >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma \cup \mathcal{F}, \mathbb{X})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ then

$$\text{DefOver}_{s_1 \cdots s_n, s'_1 \cdots s'_m}(\phi, \mathbb{F}) = \bigcup_{j=1}^{j=m} \text{TermsDefOver}(\tau_j, \mathbb{F}).$$

(3) Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_1 \in T(\Sigma \cup \mathcal{F}, \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma \cup \mathcal{F}, \mathbb{X}')_s$ for some $s \in S$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y\}$ such that $t \in X_{\mathbf{n}}$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathrm{DefOver}_{\mathbf{t}\, s_1 \cdots s_n, s}(\phi, \mathbb{F}) = \mathrm{TermsDefOver}(\tau_1, \mathbb{F}) \cup \mathrm{TermsDefOver}(\tau_2, \mathbb{F}).$$

(4) Vector-Valued Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = < \tau_{1,1}, \ldots, \tau_{1,m} >;$$
$$f(t+1, x_1, \ldots, x_n) = < \tau_{2,1}, \ldots, \tau_{2,m} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_{1,j} \in T(\Sigma \cup \mathcal{F}, \mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma \cup \mathcal{F}, \mathbb{X}')_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1, \ldots, Y_m\}$ such that $t \in X_{\mathbf{n}}$ and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, m$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathrm{DefOver}_{\mathbf{t}\, s_1 \cdots s_n, s'_1 \cdots s'_m}(\phi, \mathbb{F}) = \bigcup_{j=1}^{j=m} (\mathrm{TermsDefOver}(\tau_{1,j}, \mathbb{F}) \cup \mathrm{TermsDefOver}(\tau_{2,j}, \mathbb{F})).$$

Given a pre-PREQ specification Φ to test Φ's 'primitive recursiveness' our strategy is to repeatedly test each constituent RPREQ specification ϕ_i for $i = 1, \ldots, l$ to see which function symbols from the set $\mathcal{F}$ are being used in their definitions. From this information we construct a set $\mathcal{F}_i \subseteq \mathcal{F}$ associated with each ϕ_i and a set $\mathcal{F}_i^T$ representing the transitive closure of $\mathcal{F}_i$ relative to $\mathcal{F}_1, \ldots, \mathcal{F}_{i-1}, \mathcal{F}_{i+1}, \ldots, \mathcal{F}_l$. Now *mutato mutandis* the 're-numbering' ι, if $\mathcal{F}_i^T \not\supseteq \{i\}$ then we can conclude that if every function symbol from $\mathcal{F}$ is interpreted by a RPREQ specification then the pre-PREQ specification does indeed represent a primitive recursive function.

Essentially, the definition of InTermsOf that follows is a formalization of the informal algorithm that we have just described. Notice in particular the role of the sets $\mathbb{S}$ and $\mathbb{S}'$ in the following definition that we use to ensure that InTermsOf is terminating.

Definition 49. For each l, m, ι, η and ς defined as in Definition 46 and for each $u, v \in S^+$ we define

$$\mathrm{InTermsOf}_l : \mathrm{PREQ}_0(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma} \times \{1, \ldots, l\} \times \wp(\{f_{n,n'} \mid n, n' \in \mathbf{N}\}) \rightarrow \wp(\{1, \ldots, l\})$$

(ambiguously denoted InTermsOf) by

$$(\forall \Phi \in \mathrm{PREQ}_0(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma})\,(\forall i \in \{1, \ldots, l\})\,(\forall \mathbb{F} \in \wp(\{f_{n,n'} \mid n, n' \in \mathbf{N}\}))$$
$$\mathrm{InTermsOf}(\Phi, i, \mathbb{F}) = \mathrm{InTermsOf}'(\Phi, i, \varnothing, \mathbb{F})$$

wherein for each l, m, ι, η and ς and for each $u, v \in S^+$ defined as before

$$\mathrm{InTermsOf}'_l : \mathrm{PREQ}_0(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma} \times \{1, \ldots, l\} \times \wp(\mathbf{N}) \times \wp(\{f_{n,n'} \mid n, n' \in \mathbf{N}\}) \rightarrow \wp(\{1, \ldots, l\})$$

(ambiguously denoted InTermsOf') is defined by

$$(\forall \Phi \in \mathrm{PREQ}_0(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma})\,(\forall i \in \{1, \ldots, l\})\,(\forall \mathbb{S} \in \wp(\mathbf{N}))\,(\forall \mathbb{F} \in \wp(\{f_{n,n'} \mid n, n' \in \mathbf{N}\}))$$

$$\text{InTermsOf}'(\Phi, i, \mathbb{S}, \mathbb{F}) = \mathcal{F}_i \cup \mathcal{F}_i^T$$

wherein

$$\mathcal{F}_i = \{\iota(j) \mid j \in \text{DefOver}(\phi_i, \mathbb{F}) \wedge \iota(j) \downarrow\}$$

and

$$\mathcal{F}_i^T = \bigcup_{k \in \mathcal{F}_i - \mathbb{S}'} \text{InTermsOf}'(\Phi, k, \mathbb{S}', \mathbb{F})$$

wherein

$$\mathbb{S}' = \mathbb{S} \cup \{i\}.$$

Thus, for example if $\Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in \text{PREQ}_0(\Sigma, X)$ wherein for $i = 1, \ldots, l$ we have $\phi_i \in \text{RPREQ}(\Sigma', X)$ defined using Case (1); that is, if ϕ_i is defined using a single term τ_i then

$$\text{InTermsOf}(< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma >, i, \Sigma' - \Sigma) = \mathbb{P}$$

for some $\mathbb{P} \subseteq \{1, \ldots, l\}$ if and only if for each $j \in \mathbb{P}$ *either* the function symbol $f_{k,p}$ for some $k, p \in \mathbb{N}$ appears in τ_i and $\iota(k) = j$ *or* the function symbol $f_{k,p}$ appears in τ_i and $j \in \text{InTermsOf}(< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma >, \iota(k), \Sigma' - \Sigma)$.

Lemma 23. *For each* $\Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ_1(\Sigma, X)_{u,v}$, *for some* $u, v \in S^+$, *for each* $j \in \{1, \ldots, l\}$ *and for each* $\mathbb{F} \subseteq \{f_{n,n'} \mid n, n' \in \mathbb{N}\}$ *if*

$$|InTermsOf(\Phi, j, \mathbb{F})| = \mathbb{P}$$

and

$$DefOver(\phi_j, \mathbb{F}) \supseteq \{k\}$$

for some $k \in \{1, \ldots, m\}$ *then for each* $e \in \{1, \ldots, j-1, j+1, \ldots, l\}$ *such that*

$$InTermsOf(\Phi, j, \mathbb{F}) \supseteq \{e\}$$

we have

$$|InTermsOf(\Phi, e, \mathbb{F})| < |\mathbb{P}|.$$

Proof. Immediate from the definition of InTermsOf and DefOver.

□

PREQ_1 Specifications. As we indicated we can now use InTermsOf to identify the subset of all pre-PREQ specifications that define a primitive recursive function; that is, the class $\text{PREQ}_1 \subset \text{PREQ}_0$. Finally, we identify the class of total PREQ specifications (Definition 51). The role of PREQ_1 specifications is explained prior to the formalization of the semantics of RPREQ and PREQ at the end of this section.

Definition 50. We define $\mathrm{PREQ}_1(\Sigma, X) \subset \mathrm{PREQ}_0(\Sigma, X)$ such that

$$\Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in \mathrm{PREQ}_1(\Sigma, X)$$

if and only if

$$(1) \qquad \begin{array}{c} (\forall i \in \{1, \ldots, l\}) \\ i \notin \mathit{InTermsOf}(< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma >, i, \mathcal{F}) \end{array}$$

wherein $\mathcal{F}$ is defined as in Definition 46 and

$$(2) \qquad (\forall i \in \{1, \ldots, m\}) \quad \iota(i)\downarrow \Longrightarrow (u^{\iota(i)} = \eta^D(i) \wedge v^{\iota(i)} = \eta^R(i)).$$

Example 15. Let Σ and X be defined as in Example 13. If

$$\begin{aligned} \phi_1 &\stackrel{def}{=} f(0, x_1) = x_1, \\ &\quad f(t+1, x_1) = Succ(Y); \\ \phi_2 &\stackrel{def}{=} f(0, x_1) = 0, \\ &\quad f(t+1, x_1) = f_{1,1}(x_1, Y); \\ \phi_3 &\stackrel{def}{=} f(0, x_1) = Succ(0), \\ &\quad f(t+1, x_1) = f_{2,1}(x_1, Y); \end{aligned}$$

and $\iota = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$, $\eta^D = \{1 \mapsto \mathbf{n}^2, 2 \mapsto \mathbf{n}^2, 3 \mapsto \mathbf{n}^2\}$, and $\eta^R = \{1 \mapsto \mathbf{n}, 2 \mapsto \mathbf{n}, 3 \mapsto \mathbf{n}\}$ then $\Phi = < \phi_1, \phi_2, \phi_3; \iota; \eta; 3 > \in \mathrm{PREQ}_1(\Sigma, X)^{3,3,\iota,\eta,3}_{\mathbf{n}\,\mathbf{n},\mathbf{n}}$.

Given the semantic interpretation we intend Φ represents the exponential function over the natural numbers.

Using the class PREQ_1 of partial PREQ specification we can now identify the class of total PREQ specifications that do indeed specify a primitive recursive function.

Definition 51. For any $\Phi \in \mathrm{PREQ}_1(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v} \in \mathrm{PREQ}_1$ for some $l, m, \iota, \eta, \varsigma, u$ and v defined as in Definition 50 if for each

$$i \in \bigcup_{j=1}^{j=l} \mathit{InTermsOf}(\Phi, j, \{f_{1,1}, \ldots, f_{1,|\eta^R(1)|}, \ldots, f_{m,1}, \ldots, f_{m,|\eta^R(m)|}\})$$

we have $\iota(i)\downarrow$ then we say that either Φ is a *totally defined primitive recursive equational specification* or just *totally defined* denoted $\Phi \in \mathrm{PREQ}(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v}$.

In common with other formal language definitions in this thesis we gather together all totally defined PREQ specifications into an $S^+ \times S^+$-indexed family

$$\mathrm{PREQ}(\Sigma, X) = < \mathrm{PREQ}(\Sigma, X)_{u,v} \mid u, v \in S^+ > .$$

Of the class of all total PREQ specifications one particular further sub-class that we wish to identify are those wherein each function symbol is interpreted directly by the RPREQ specification indicated by its index; that is, wherein function $f_{i,j}$ is interpreted by co-ordinate j of the ith RPREQ specification:

Definition 52. For any $\Phi \in \mathrm{PREQ}(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v}$ for some $l, m, \iota, \eta, \varsigma, u$ and v defined as in Definition 50 if
(1) $l = m$;
(2) $\iota(i) = i$ for $i = 1, \ldots, m$; and
(3) $\varsigma = 1$
then we say that Φ is *standard.*

In the sequel if Φ is standard then we will write '$< \phi_1, \ldots, \phi_m >$' for Φ omitting l, ι, η and ς. Similarly, if $\Phi \in \mathrm{PREQ}(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v}$ satisfies (1) and (2), but not (3) then we will write '$< \phi_1, \ldots, \phi_m; \varsigma >$' for Φ.

See Example 16 on Page 150 for an example of a standard PREQ specification.

Lemma 24. *If* $\Phi = < \phi_1, \ldots, \phi_l > \in PREQ(\Sigma, X)_{u,v}$ *then for each* $i \in \{1, \ldots, l\}$

$$< \phi_1, \ldots, \phi_l; i > \in PREQ(\Sigma, X)_{u',v'}.$$

Modularity: Joining PREQ Specifications. Rather than directly define the class of total PREQ specifications as a sub-class of PREQ_0 we have identified the intermediate sub-class PREQ_1 to address one of the issues that is important in the development of ASTRAL: the use of modular specification techniques. In particular, it is useful at the abstract language level to allow several partially defined specifications to be joined together. At the 'front end' this enables a user of ASTRAL to define a complete specification either 'bottom up' or 'top down' via the definition of sub-functions that may be incompletely specified. Using a common programming technique these sub-functions can then be brought together to give a final complete specification wherein each function is specified by either some particular equation or equations from within the specifications constituent parts.

In order to support the use of this technique it is sufficient to define a function that joins two PREQ_1 specifications into a single specification. The following function $\uplus$ designed for this purpose sets out formally the conditions under which two partial PREQ specifications may be joined.

Informally, we will allow two partial PREQ specifications Φ_1 and Φ_2 to be joined if the resulting specification is itself at least a partial PREQ specification (it does not need to be total). To ensure that this is the case we place certain constraints upon the functions η_1 and ι_1 and η_2 and ι_2 from Φ_1 and Φ_2 respectively: that (1) η_1 and η_2 must agree on the functionality of any function symbols for which both functions are defined; and (2) the sub-domains of the domains of ι_1 and ι_2 for which each function is defined must be disjoint.

Definition 53. For each $l_1, l_2, m_1, m_2 \in \mathbb{N}^+$; for each

$$\eta_1 : \{1, \ldots, m_1\} \rightarrow S^+ \times S^+$$

$$\eta_2 : \{1, \ldots, m_2\} \to S^+ \times S^+$$

such that

$$(\eta_1 \supseteq \eta_2) \vee (\eta_2 \supseteq \eta_1) \vee (\eta_1 \bigcap \eta_2 = \varnothing);$$

for each injection

$$\iota_1 : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, l_1\}$$

and for each injection

$$\iota_2 : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, l_2\}$$

such that

$$Dom\downarrow(\iota_1) \bigcap Dom\downarrow(\iota_2) = \varnothing;$$

for each $\varsigma_1 \in \{1, \ldots, l_1\}$ and for each $\varsigma_2 \in \{1, \ldots, l_2\}$; and for each $n \in \{1, 2\}$ and for each $u^1, u^2, v^1, v^2 \in S^+$ we define

$$\biguplus^{l_1,l_2,m_1,m_2,\iota_1,\iota_2}_{\varsigma_1,\varsigma_2,n,u^1,u^2,v^1,v^2} : \mathrm{PREQ}_1(\Sigma, X)^{l_1,m,\iota_1,\eta_1,\varsigma_1}_{u^{\varsigma_1},v^{\varsigma_1}} \times \mathrm{PREQ}_1(\Sigma, X)^{l_2,m,\iota_2,\eta_2,\varsigma_2}_{u^{\varsigma_2},v^{\varsigma_2}} \to \mathrm{PREQ}_1(\Sigma, X)^{l_1+l_2,m,\iota,\eta,k}_{u^{\varsigma^n},u^{\varsigma^n}}$$

(ambiguously denoted $\biguplus_n$) by

$$(\forall \Phi_1 =< \phi^1_1, \ldots, \phi^1_{l_1}; \iota_1; \eta_1; \varsigma_1 > \in \mathrm{PREQ}_1(\Sigma, X)_{u^{\varsigma_1},v^{\varsigma_1}})$$

$$(\forall \Phi_2 =< \phi^2_1, \ldots, \phi^2_{l_2}; \iota_2; \eta_2; \varsigma_2 > \in \mathrm{PREQ}_1(\Sigma, X)_{u^{\varsigma_2},v^{\varsigma_2}})$$

$$\Phi_1 \biguplus_n \Phi_2 = \Phi' =< \phi'_1, \ldots, \phi'_{l_1+l_2}; \iota; \eta; k >$$

wherein $\phi'_i = \phi^1_i$ for $i = 1, \ldots, l_1$ and $\phi'_i = \phi^2_{i-l_1}$ for $i = l_1 + 1, \ldots, l_1 + l_2$; $\iota : \{1, \ldots, m\} \rightsquigarrow \{1, \ldots, l_1 + l_2\}$ is defined by

$$(\forall i \in \{1, \ldots, m\}) \quad \iota(i) = \begin{cases} \iota_1(i) & \text{if } \iota_1(i)\downarrow, \text{ and} \\ \iota_2(i) + l_1 & \text{if } \iota_2(i)\downarrow; \end{cases}$$

$$\eta = \begin{cases} \eta_1 & \text{if } \eta_1 \supseteq \eta_2, \\ \eta_2 & \text{if } \eta_2 \supseteq \eta_1, \\ \eta_1 \cup \eta_2 & \text{otherwise;} \end{cases}$$

and

$$k = \begin{cases} \varsigma_1 & \text{if } n = 1 \\ \varsigma_2 + l_1 & \text{otherwise} \end{cases}$$

Well-Definedness. We consider the case wherein $n = 1$ and leave the case wherein $n = 2$ (that is similar to the case wherein $n = 1$) to the reader.

To show that $\biguplus_1$ is well-defined it is sufficient to show that

(A) $$(\forall i \in \{1, \ldots, l_1 + l_2\}) \quad i \notin \mathrm{InTermsOf}(\Phi', i, \mathbb{F})$$

wherein $\mathbb{F} = \{f_{n,n'} \mid n, n' \in \mathrm{N}\}$ and for $i = 1, \ldots, m$

(B) $$\iota(i)\downarrow \Longrightarrow (u^{\iota(i)} = \eta^D(i) \wedge v^{\iota(i)} = \eta^D(i)).$$

To prove property (A) we will use induction. In the basis case we will show that the assumption that there exists a $k \in \{1, \ldots, l_1 + l_2\}$ such that $\text{InTermsOf}(\Phi', k, \mathbb{F}) \supseteq \{k\}$ leads to a contradiction.

Proof of (A). By induction on $q = |\text{InTermsOf}(\Phi', k, \mathbb{F})|$. We consider two basis cases:

(1) $|\text{InTermsOf}(\Phi', k, \mathbb{F})| = 0$.

(2) $|\text{InTermsOf}(\Phi', k, \mathbb{F})| = 1$.

Basis Case (1) $q = 0$. This is obvious and is omitted.

Basis Case (2) $q = 1$. In this case $\text{InTermsOf}(\Phi', k, \mathbb{F}) = \{k\}$. We consider two sub-cases:

(a) $1 \leq k \leq l_1$,

(b) $l_1 < k \leq l_1 + l_2$.

Sub-case (a) $1 \leq k \leq l_1$. This assumption implies that there exists a $k' \in \{1, \ldots, m\}$ such that

$$(l_1(k') = k) \wedge DefinedOver(\phi_1^k, \mathbb{F}) \supseteq \{k'\}$$

contrary to the assumption that $\Phi_1 \in \text{PREQ}_1(\Sigma, X)$.

Sub-case (b) $l_1 < k \leq l_1 + l_2$. This assumption implies that there exists a $k' \in \{1, \ldots, m\}$ such that

$$(l_2(k') = p = k - l_1) \wedge DefinedOver(\phi_2^p, \mathbb{F}) \supseteq \{k'\}$$

contrary to the assumption that $\Phi_2 \in \text{PREQ}_1(\Sigma, X)$.

Induction Hypothesis. Assume that for some fixed $q' \in \mathbb{N}^+$ if $i \in \{1, \ldots, l_1 + l_2\}$ and $|\text{InTermsOf}(\Phi', i, \mathbb{F})| \leq q'$ then $i \notin \text{InTermsOf}(\Phi', i, \mathbb{F})$.

Induction. We must show that if there exists $d \in \{1, \ldots, l_1 + l_2\}$ such that

$$|\text{InTermsOf}(\Phi', d, \mathbb{F})| = q' + 1$$

then $d \notin \text{InTermsOf}(\Phi', d, \mathbb{F})$.

Notice by definition that

$$\text{InTermsOf}(\Phi', d, \mathbb{F}) = \{\iota(j) \mid j \in T \wedge \iota(j)\downarrow\} \cup T'$$

wherein

$$T = DefinedOver(\phi'_d, \mathbb{F}),$$

$$T' = \bigcup_{j \in R} \text{InTermsOf}^*(\Phi', j, \{d\}, \mathbb{F}),$$

and

$$R = \{\iota(k) \mid (k \in T - \{d\}) \wedge \iota(k)\downarrow\}.$$

We have two sub-cases to consider:

(a) $d \in \{\iota(j) \mid j \in T \wedge \iota(j)\downarrow\}$.

(b) $d \in T'$.

Sub-case (a) $d \in \{\iota(j) \mid j \in T \wedge \iota(j)\downarrow\}$. This implies that there exist a $k' \in T$ such that $\iota(k) = d$ and hence that either

$$\Phi_1 \notin \mathrm{PREQ}_1(\Sigma, X)$$

if $\iota(k') = \iota_1(k')$ or

$$\Phi_2 \notin \mathrm{PREQ}_1(\Sigma, X)$$

if $\iota(k') = \iota_2(k') + l_1$ contrary to hypothesis.
Sub-Case (b) $d \in T'$. Notice in this case that as $d \notin S = \{\iota(j) \mid j \in T \wedge \iota(j)\downarrow\}$ it must be the case that $T \supseteq \{q\}$ for some $q \in \{1, \ldots, m\}$ such that $\iota(q) \neq d$ otherwise by definition we would have $R = \varnothing$ and hence that $|T'| = 0$ contrary to the hypothesis that $|\mathrm{InTermsOf}(\Phi', d, \mathbb{F})| \geq 1$. Therefore, $|S| > 1$ and consequently $|T'| \leq q'$ and so by the induction hypothesis $d \notin T'$ as required.

□

Proof of Property (B). Notice that if we let (u^i, v^i) be the type of ϕ'_i for $i = 1, \ldots, l_1 + l_2$, (w^j, x^j) be the type of ϕ^1_j for $j = 1, \ldots, l_1$ and let (y^k, z^k) be the type of ϕ^2_k for $k = 1, \ldots, l_2$ then by definition of $\biguplus_1$

$$\phi'_i = \phi^1_i$$

for $i = 1, \ldots, l_1$ and

$$\iota_1(p)\downarrow \Longrightarrow \iota(p) = \iota_1(p)$$

for $p = 1, \ldots, m$ we have

$$u^{\iota(p)} = w^{\iota_1(p)}$$

and

$$v^{\iota(p)} = x^{\iota_1(p)}.$$

Similarly, as by the definition of $\biguplus_1$

$$\phi'_i = \phi^2_i$$

for $i = l_1 + 1, \ldots, l_1 + l_2$ and

$$\iota_2(p)\downarrow \Longrightarrow \iota(p) = \iota_2(p) + l_1$$

for $p = 1, \ldots, m$ (the fact that $\iota_1(p)\downarrow$ and $\iota_2(p)\downarrow$ are mutually exclusive is guaranteed by hypothesis) we have

$$u^{\iota(p)} = y^{\iota_2(p)}$$

and

$$v^{\iota(p)} = z^{\iota_1(p)}.$$

Consequently notice that by hypothesis for each $p \in \{1, \ldots, m\}$ we have

$$\iota_1(p)\downarrow \Longrightarrow (w^{\iota_1(p)} = \eta^D(p) \wedge x^{\iota_1(p)} = \eta^D(p))$$

and

$$\iota_2(p)\downarrow \Longrightarrow (y^{\iota_2(p)} = \eta^D(p) \wedge z^{\iota_2(p)} = \eta^D(p))$$

as required.

□

Example 16. If Φ is defined as in Example 15 and

$$\Phi' = <\phi, \iota', \eta, 1> \in \mathrm{PREQ}_1(\Sigma, X)_{\mathbf{n,n}}$$

wherein

$$\phi \stackrel{def}{=} f(x_1) = f_{3,1}(f_{3,1}(x_1, 2), 2)$$

and

$$\iota' = \varnothing$$

then

$$\Phi \biguplus_2 \Phi' = \Phi'' = <\phi_1, \ldots, \phi_3, \phi; \iota'; \eta; 4> .$$

Furthermore, given the semantic interpretation we intend Φ'' is the equational specification of the function 2^{2^n}.

Also notice that Φ is totally defined and standard, Φ'' is totally defined, and Φ' is partial. In particular, notice that we may make totally defined PREQ specifications by joining partial PREQ specifications.

The Semantics of RPREQ and PREQ. We begin by defining a variable evaluation map ν (see Section 2.3.10) that we require to give a denotational semantics to RPREQ specifications.

Definition 54. Let X be any S-indexed collection of variable symbols. For each $\mathbb{X} = \{x_1, \ldots, x_n\}$ for some $x_i \in X$ of type s_i for $i = 1, \ldots, n$ we define

$$\nu^{X,\mathbb{X}} : A^{s_1 \cdots s_n} \to X \rightsquigarrow A$$

(ambiguously denoted $\nu^{\mathbb{X}}$) by

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n})\,(\forall x \in X) \quad \nu^{\mathbb{X}}(a)(x) = \begin{cases} a_i & \text{if } x = x_i, \text{ and} \\ \uparrow & \text{otherwise.} \end{cases}$$

Lemma 25. *Let* $\mathbb{X} = \{x_1, \ldots, x_n\}$ *for some* $x_i \in X_{s_i}$ *for some* $s_i \in S$ *for* $i = 1, \ldots, n$. *If* $\tau \in T(\Sigma, \mathbb{X})$ *then*

$$(\forall a \in A^{s_1 \cdots s_n}) \quad V_{\nu^{\mathbb{X}}(a)}(\tau)\downarrow .$$

This fact is used implicitly in the following definitions.

Definition 55. For each $\phi \in \mathrm{RPREQ}(\Sigma, X)$ we define the *meaning of* ϕ *over* A by the $S^+ \times S^+$-indexed family of mappings

$$[\![\cdot]\!]_A = < [\![\cdot]\!]^A_{u,v} : \mathrm{RPREQ}(\Sigma, X)_{u,v} \to [A^u \to A^v] | u, v \in S^+ >$$

wherein for each $u, v \in S^+$ the map $[\![\cdot]\!]^A_{u,v} : \mathrm{RPREQ}(\Sigma, X)_{u,v} \to [A^u \to A^v]$ (ambiguously denoted $[\![\cdot]\!]_A$) is defined uniformly in (u, v) by the structural complexity of the specification $\phi \in \mathrm{RPREQ}(\Sigma, X)_{u,v}$ as follows: (throughout the following definitions we use $\mathbb{X}$ to denote the set $\{x_1, \ldots, x_n\}$ wherein $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, n \geq 1$)

(1) Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma, \mathbb{X})_s$ for some $s \in S$ then $[\![\phi]\!]_A : A^{s_1 \cdots s_n} \to A_s$ is defined by

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad [\![\phi]\!]_A(a) = V_{\nu_{\mathbb{X}(a)}}(\tau).$$

(2) Vector-Valued Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = < \tau_1, \ldots, \tau_m >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma, \mathbb{X})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ then $[\![\phi]\!]_A : A^{s_1 \cdots s_n} \to A^{s'_1 \cdots s'_m}$ is defined by

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad [\![\phi]\!]_A(a) = (V_{\nu_{\mathbb{X}(a)}}(\tau_1), \ldots, V_{\nu_{\mathbb{X}(a)}}(\tau_m)).$$

(3) Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_1 \in T(\Sigma, \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma, \mathbb{X}')_s$ for some $s \in S$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y\}$ such that $t \in X_n$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then $[\![\phi]\!]_A : T \times A^{s_1 \cdots s_n} \to A_s$ is defined by

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad [\![\phi]\!]_A(0, a) = V_{\nu_{\mathbb{X}}(a)}(\tau_1)$$

and

$$(\forall t' \in T)(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad [\![\phi]\!]_A(t'+1, a) = V_{\nu_{\mathbb{X}'(a, t', [\![\phi]\!]_A(t', a))}}(\tau_2).$$

(4) Vector-Valued Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = < \tau_{1,1}, \ldots, \tau_{1,m} >;$$
$$f(t+1, x_1, \ldots, x_n) = < \tau_{2,1}, \ldots, \tau_{2,m} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_{1,j} \in T(\Sigma, \mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma, \mathbb{X}')_{s'_j}$, for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1, \ldots, Y_m\}$ such that $t \in X_{\mathbf{n}}$ and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, m$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then $[\![\phi]\!]_A : T \times A^{s_1 \cdots s_n} \to A^{s'_1 \cdots s'_m}$ is defined by

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad [\![\phi]\!]_A(0, a) = (V_{\nu x_{(a)}}(\tau_{1,1}), \ldots, V_{\nu x_{(a)}}(\tau_{1,m}))$$

and

$$(\forall t' \in T)\,(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n})$$

$$[\![\phi]\!]_A(t'+1, a) = (V_{\nu x'_{(a,t',[\![\phi]\!]_A(t',a))}}(\tau_{2,1}), \ldots, V_{\nu x'_{(a,t',[\![\phi]\!]_A(t',a))}}(\tau_{2,m})).$$

Using the formal definition of the semantics of RPREQ specifications we now finally complete this section with the formal semantics of PREQ specifications.

Definition 56. For each $\Phi \in \mathrm{PREQ}(\Sigma, X)$ we define the *meaning of* Φ *over* A by the $S^+ \times S^+$-indexed family of mappings

$$[\![.]\!]_A = < [\![.]\!]_A^{u,v} : \mathrm{PREQ}(\Sigma, X)_{u,v} \to [A^u \to A^v] \mid u, v \in S^+ >$$

wherein for each $u, v \in S^+$ the mapping

$$[\![.]\!]_A^{u,v} : \mathrm{PREQ}(\Sigma, X)_{u,v} \to [A^u \to A^v]$$

(ambiguously denoted $[\![.]\!]_A$) is defined for each $\Phi \in \mathrm{PREQ}(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma}$ as follows:

$$(\forall a \in A^u) \quad [\![\Phi]\!]_A(a) = [\![\phi_\varsigma]\!]_A(a)$$

wherein each $f_{i,j} \in \mathcal{F}$ for $i = 1, \ldots, m$ and for $j = 1, \ldots, |\eta^R(i)|$; and $\mathcal{F}$ as defined in Definition 50 satisfies

$$f_{i,j}^{A,\tau} = ([\![< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >]\!]_A)_j.$$

Notice that the well-definedness of $< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >$ is addressed in Lemma 24.

5.3.3 The Soundness and Adequacy of PREQ

In this section we turn our attention to the proof of Theorem 10; that is, we show that PREQ is both sound and adequate with respect to the class of primitive recursive functions. In order to prove Theorem 10 we define two compilers $\mathbb{C}^{\mathrm{PR}} : \mathrm{PREQ}(\Sigma, X) \to \mathrm{PR}(\Sigma)$ and $\mathbb{C}^{\mathrm{PREQ}} : \mathrm{PR}(\Sigma) \to \mathrm{PREQ}(\Sigma, X)$ respectively, with the intention that a proof of Statement (1) of Theorem 10 follows as a corollary from a proof of the correctness of $\mathbb{C}^{\mathrm{PR}}$ and similarly the proof of Statement (2) of Theorem 10 follows as a corollary from a proof of the correctness of $\mathbb{C}^{\mathrm{PREQ}}$. However, for convenience we leave the formal proofs of correctness of the compilers $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$ to Appendix A.

An informal algorithm that describes the basic intuition behind the operation of the compiler $\mathbb{C}^{\mathrm{PREQ}}$ can be found in Section 5.2. The compiler $\mathbb{C}^{\mathrm{PR}}$ is structured so as far as possible it behaves

as the inverse of $\mathbb{C}^{\mathrm{PREQ}}$. The reader wishing to omit the technical material concerned with the construction of both these compilers can move directly to Section 5.4 on Page 170.

The Soundness of PREQ. In order to define $\mathbb{C}^{\mathrm{PR}}$ we will require three sub-compilers: $\mathbb{C}^{\Gamma}$ and $\mathbb{C}^{\mathfrak{T}}$ that compile terms into PR, and $\mathbb{C}^{\star\mathrm{PR}}$ that compiles RPREQ specifications into PR. As such we begin with the definitions of these compilers.

Discussion: Compiling Terms into PR. Recall that due to the particular form of the definition of a PREQ specification $\Phi = < \phi_1, \ldots, \phi_l > \in \mathrm{PREQ}(\Sigma, X)$ it is the terms that occur on the left-hand-side of each RPREQ specification ϕ_i for $i = 1, \ldots, l$ that play the central role in the description of the function that Φ represents. Also recall that all of these terms fall into two categories: (1) the terms used in simple RPREQ specifications and in the basis case of primitive recursive RPREQ specifications – that are defined over $\mathrm{T}(\Sigma \cup \mathcal{F}, \mathbb{X})$ (see Definition 50); and (2) the terms in the induction case of primitive recursive RPREQ specifications – that are defined over $\mathrm{T}(\Sigma \cup \mathcal{F}, \mathbb{X} \cup \{t, Y_1, \ldots, Y_n\})$ for some $n \in \mathbb{N}^+$ wherein the variables t and Y_j for $j = 1, \ldots, n$ are given the particular interpretation described in the discussion at the beginning of Section 5.3.2.

First, we remark that in the context of compilation into PR it is useful to consider these two classes of terms separately as the special interpretation of variable symbols $Y_1, \ldots, Y_n$ slightly complicates the compilation process. In particular, we require terms from the second class of equations to be mapped into PR schemes of type $(\mathbf{t}\, u, v)$ for some $u, v \in S^+$. However, in general this requirement is not true of the first class of terms. Therefore, while both compilers $\mathbb{C}^{\Gamma}$ and $\mathbb{C}^{\mathfrak{T}}$ perform essentially the same compilation process, using two separate compilers is convenient as it simplifies the overall complexity of our constructions.

Secondly, while the similarity of terms and function schemes make the process of compiling terms into PR basically routine, it is slightly more difficult in the context of terms occurring in PREQ specifications due to the intended interpretation of each additional function symbol from the signature $\mathcal{F}$. In more detail, recall that (ignoring for the moment the role of the function ι) each of the additional function symbols $f_{i,j} \in \mathcal{F}$ for some $i, j \in \mathbb{N}^+$ is interpreted by co-ordinate j of the ith RPREQ specification ϕ_i. Therefore, if symbol $f_{i,j}$ occurs in term τ (say) and we wish to compile τ into an equivalent scheme $\alpha_\tau \in \mathrm{PR}(\Sigma)$ then as part of this process we must construct the sub-scheme $\alpha_{f_{i,j}} \subseteq \alpha_\tau$ equivalent to $f_{i,j}$. However, the structure of the required scheme $\alpha_{f_{i,j}}$ must by definition be dependent on the structure of ϕ_i; that is, more specifically it must be dependent on the terms in ϕ_i that are not part of the structure of τ itself. Moreover, as we cannot derive the structure of ϕ_i from τ alone and we cannot predict in advance the particular value of $i \in \{1, \ldots, l\}$ we must in general have the whole of $\Phi =< \phi_1, \ldots, \phi_l >$ available when we compile τ into a function scheme. For this reason when compiling a term τ it is necessary to index both compilers $\mathbb{C}^{\Gamma}$ and $\mathbb{C}^{\mathfrak{T}}$ by the particular PREQ specification Φ in which τ occurs. As such if function symbol $f_{i,j} \in \mathcal{F}$ occurs in τ then the appropriate PR scheme can be created by recursively using the compiler $\mathbb{C}^{\star\mathrm{PR}}$ that compiles RPREQ specification into PR to generate the scheme $\alpha_{f_{i,j}}$.

The only potential problem with this method is that $\mathbb{C}^{\star\mathrm{PR}}$ is itself defined in terms of $\mathbb{C}^{\Gamma}$

and $\mathbb{C}^{\mathrm{T}}$ and therefore we must consider the termination properties of this recursive process; that is, if τ occurs in RPREQ specification ϕ_i then we must show that $\mathbb{C}^{\mathrm{T}}_{\Phi}(\tau)$ cannot give rise to a recursive call of $\mathbb{C}^{\mathrm{PR}}_{\Phi}(\phi_i)$. Fortunately, it is not difficult to demonstrate this fact as in order for Φ to be a total PREQ specification it must satisfy Property (1) of Definition 50.

Finally, we note that we must also index $\mathbb{C}^{\mathrm{T}}$ and $\mathbb{C}^{\mathrm{T}}$ with the particular variables $\mathbb{X} = \{x_1, \ldots, x_n\}$ wherein $x_i \in X_{s_i}$ for some $s_i \in S$ over which the term τ to be compiled is defined. This enables us to determine the type of the function scheme that will be required (as variables are used as input) and reduces the construction of the necessary sub-scheme to represent a variable to an analysis of the particular index $i \in \{1, \ldots, n\}$. In particular, if $u = s_1 \ldots s_n$ then we can straightforwardly define $\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X}}(x) = \mathrm{U}^u_i$ that clearly has the required semantics as it represents a function that has n inputs of the correct type and selects the ith co-ordinate of its input (which is variable x_i) as output. These ideas are formalized as follows:

The Compiler $\mathbb{C}^{\mathrm{T}}$.

Definition 57. For each $\Phi \in \mathrm{PREQ}(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{w,v}$, for some $l,m,\iota,\eta,\varsigma,w$ and v as defined in Definition 50; for each $u = s_1 \cdots s_n \in S^+$; for each $s \in S$; for each $\mathbb{X} = \{x_1, \ldots, x_n\} \subseteq X$ such that $x_i \in X_{s_i}$ for $i = 1, \ldots, n$ are distinct; and for each $s \in S$ we define

$$\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},u,s} : T(\Sigma', \mathbb{X})_s \rightarrow \mathrm{PR}(\Sigma)_{u,s}$$

(ambiguously denoted either $\mathbb{C}^{\mathrm{T}}_{u,s}$ or just $\mathbb{C}^{\mathrm{T}}$) wherein $\Sigma' = \Sigma \cup \mathcal{F}$ is the extended signature of Φ as defined in Definition 50 uniformly in u and s by induction on the structural complexity of a term $\tau \in T(\Sigma', \mathbb{X})_s$ as follows:

Basis Cases.

(1) Constants. If $\tau = c_s$ for some $c \in \Sigma'_{\lambda,s}$ for some $s \in S$ then

$$\mathbb{C}^{\mathrm{T}}(\tau) = c^u_s.$$

Well-definedness. It is clear here from the simplicity of the scheme that $\mathbb{C}^{\mathrm{T}}(\tau) \in \mathrm{PR}(\Sigma)_{u,s}$ as required.

(2) Variables. If $\tau = x_i$ for some $x_i \in \mathbb{X}_s$ for some $s \in S$ then

$$\mathbb{C}^{\mathrm{T}}(\tau) = \mathrm{U}^u_i.$$

Well-definedness. By hypothesis on x_i we have $u_i = s_i = s$ and therefore $\mathbb{C}^{\mathrm{T}}(\tau) \in \mathrm{PR}(\Sigma)_{u,s}$ as required.

Induction Hypothesis. Assume for each $\tau \in T(\Sigma', \mathbb{X})_s$ for some $s \in S$ that for any term $\tau' \in T(\Sigma', \mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that $\mathbb{C}^{\mathrm{T}}(\tau') \in \mathrm{PR}(\Sigma)_{u,s'}$.

Induction.

(3) Algebraic operations. If $\tau = \sigma(\tau_1, \ldots, \tau_k)$ for some $\sigma \in \Sigma'_{w,s}$ for some $w \in S^+$ and for some $s \in S$, and for some $\tau_i \in T(\Sigma', \mathbb{X})_{s'_i}$ for $i = 1, \ldots, k = |w|$ wherein $w = s'_1 \cdots s'_k$ then

$$\mathbb{C}^{\mathrm{T}}(\tau) = \sigma \circ \langle \mathbb{C}^{\mathrm{T}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{T}}(\tau_k) \rangle$$

if $\sigma \in \Sigma$ and

$$\mathbb{C}^{\Gamma}(\tau) = U_q^{\iota(p)} \circ \mathbb{C}_{\Phi}^{\star \mathrm{PR}}(\phi_{\iota(p)}) \circ < \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) >$$

if $\sigma = f_{p,q} \in \mathcal{F}$ wherein $\mathbb{C}^{\star \mathrm{PR}}$ is defined as in Definition 59.

Well-definedness. We have two cases to consider:

(A) $\sigma \in \Sigma$.

(B) $\sigma = f_{p,q} \in \mathcal{F} \subseteq \{f_{1,1}, \ldots, f_{1,|\eta^R(1)|}, \ldots, f_{m,1}, \ldots, f_{m,|\eta^R(m)|}\}$.

However, for convenience we first consider the sub-scheme common to both cases

$$< \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) > .$$

Notice that by the induction hypothesis $\mathbb{C}^{\Gamma}(\tau_i)$ is of type (u, s_i') for $i = 1, \ldots, k$ and hence $< \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) >$ is well-defined as a vector-valued scheme with type (u, w).

Case (A) $\sigma \in \Sigma$. As by hypothesis $\sigma \in \Sigma_{w,s}$ it is clear that

$$\sigma \circ < \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) >$$

is well-defined as a composition and $\mathbb{C}^{\Gamma}(\tau) \in \mathrm{PR}(\Sigma)_{u,s}$ as required.

Case (B) $\sigma = f_{p,q} \in \mathcal{F}$. Notice that, by hypothesis on Φ that $\mathbb{C}_{\Phi}^{\star \mathrm{PR}}(\phi_{\iota(p)})$ is terminating and well-defined with type $\eta(p) = (\eta^D(p), \eta^R(p)) = (w, \eta^R(p))$ and therefore

$$\mathbb{C}^{\star \mathrm{PR}}(\phi_{\iota(p)}) \circ < \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) >$$

is well-defined as a composition with type $(u, \eta^R(p))$. Finally, notice that by hypothesis $f_{p,q}$ is of type $(\eta^D(p), (\eta^R(p))_q) = (w, s)$ and therefore

$$U_q^{\eta^R(p)} \circ \mathbb{C}^{\star \mathrm{PR}}(\phi_{\iota(p)}) \circ < \mathbb{C}^{\Gamma}(\tau_1), \ldots, \mathbb{C}^{\Gamma}(\tau_k) >$$

is well-defined with type (u, s) and therefore $\mathbb{C}^{\Gamma}(\tau) \in \mathrm{PR}(\Sigma)_{u,s}$ as required.

The Compiler $\mathbb{C}^{\mathbf{T}}$. As we have already remarked the definition of $\mathbb{C}^{\mathbf{T}}$ is essentially the same as $\mathbb{C}^{\Gamma}$ – the only difference being that $\mathbb{C}^{\mathbf{T}}$ must deal with the special interpretation of the variable symbols t and $Y_1, \ldots, Y_n$. To motivate the construction of the necessary function schemes for these variables recall the types of the sub-schemes α_1 and α_2 in a scheme $\alpha = *(\alpha_1, \alpha_2)$ defined by primitive recursion. In particular, recall that if the overall type of α is $(\mathbf{t}\ u, v)$ for some $u, v \in S^+$ then α_1 and α_2 must be of type (u, v) and $(\mathbf{t}\ u\ v, v)$ respectively. Therefore, in general if we consider that in the case of a primitive recursive RPREQ specification our aim is the construction of some scheme α as defined above then essentially we use $\mathbb{C}^{\mathbf{T}}$ to construct that part of α defined by scheme α_2. Also, recall that form the perspective of semantics if $a = (a_1, a_2, \ldots, a_{|u|+1}) \in A^{\mathbf{t}\ u}$ is the input supplied to F_α wherein F_α is the function computed by scheme α then by definition

$$a' = (a_1 - 1, a_2, \ldots, a_{|u|+1}, F_\alpha(a_1 - 1, a_2, \ldots, a_{|u|+1}))$$

is the input supplied to F_{α_2} wherein F_{α_2} is the sub-function of F_α computed by scheme α_2. Therefore, if we also index $\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X}}$ with the particular variables t and $Y_1,\ldots,Y_m$ then as with the set $\mathbb{X}$ this enables us to determine the type of the function scheme that we will derive and again reduces the construction of the necessary sub-schemes $\alpha_x \subseteq \alpha_\tau$ to represent a variable x to an analysis of the particular index $i \in \{1,\ldots,m\}$. In particular, if $x = t$ then we can straightforwardly define $\mathbb{C}^{\mathbb{T}}(x) = \mathrm{U}_1^{\mathbf{t}\,u\,v}$; if $x = x_i$ then we define $\mathbb{C}^{\mathbb{T}}(x) = \mathrm{U}_{i+1}^{\mathbf{t}\,u\,v}$; and if $x = Y_j$ then we define $\mathbb{C}^{\mathbb{T}}(x) = \mathrm{U}_{j+|u|+1}^{\mathbf{t}\,u\,v}$ wherein $j + |u| + 1$ will select the particular co-ordinate of $F_\alpha(a_1 - 1, a_2, \ldots, a_{|u|+1})$ from a' as required.

Definition 58. For each $\Phi \in \mathrm{PREQ}(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{w,w'}$ for some $l, m, \iota, \eta, \varsigma, w$ and w' defined as in Definition 50; for each $u = s_1 \cdots s_n \in S^+$; for each $v = s'_1 \cdots s'_{n'} \in S^+$; for each $s \in S$; for each $\mathbb{X} = \{x_1,\ldots,x_n,t,Y_1,\ldots,Y_{n'}\} \subseteq X$ such that $x_i \in X_{s_i}$ for $i = 1,\ldots,n$ are distinct, $t \in X_n$ is a distinguished variable symbol distinct from each x_i, and $Y_j \in X_{s'_j}$ for $j = 1,\ldots,n'$ are distinguished variable symbols distinct from each x_i and t; and for each $s \in S$ we define

$$\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s} : T(\Sigma',\mathbb{X})_s \to \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,s}$$

(ambiguously denoted either $\mathbb{C}^{\mathbb{T}}_{u,v,s}$ or just $\mathbb{C}^{\mathbb{T}}$) wherein $\Sigma' = \Sigma \cup \mathcal{F}$ is the extended signature of Φ as defined in Definition 50 by induction on the structural complexity of a term $\tau \in T(\Sigma',\mathbb{X})_s$ uniformly in u, v and s as follows:

Basis Cases.

(1) Constants. If $\tau = c_s$ for some $c \in \Sigma'_{\lambda,s}$ for some $s \in S$ then

$$\mathbb{C}^{\mathbb{T}}(\tau) = c_s^{\mathbf{t}\,u\,v}.$$

Well-definedness. It is clear here from the simplicity of the scheme that $\mathbb{C}^{\mathbb{T}}(\tau) \in \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,s}$ as required.

(2) Variables. If $\tau = x$ for some $x \in \mathbb{X}_s$ for some $s \in S$ then

$$\mathbb{C}^{\mathbb{T}}(\tau) = \begin{cases} \mathrm{U}_1^{\mathbf{t}\,u\,v} & \text{if } x = t; \\ \mathrm{U}_{i+1}^{\mathbf{t}\,u\,v} & \text{if } x = x_i \text{ for some } i \in \{1,\ldots,n\}; \\ \mathrm{U}_{|u|+j+1}^{\mathbf{t}\,u\,v} & \text{if } x = Y_j \text{ for some } j \in \{1,\ldots,n'\}. \end{cases}$$

Well-definedness. We have three cases to consider:

Case (A) $x = t$. By hypothesis $t \in X_n$ and therefore it is clear here that $\mathbb{C}^{\mathbb{T}}(\tau) \in \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,\mathbf{t}}$ as required.

Case (B) $x = x_i$. By hypothesis on x_i we have $u_i = s_i = s$ therefore as $(\mathbf{t}\,u\,v)_{i+1} = u_i = s$ we have $\mathbb{C}^{\mathbb{T}}(\tau) \in \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,s}$ as required.

Case (C) $x = Y_j$. By hypothesis on Y_j we have $v_j = s_j = s$ therefore as $(\mathbf{t}\,u\,v)_{|u|+j+1} = v_j = s$ we have $\mathbb{C}^{\mathbb{T}}(\tau) \in \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,s}$ as required.

Induction Hypothesis. Assume for each $\tau \in T(\Sigma',\mathbb{X})_s$ for some $s \in S$ that for any term $\tau' \in T(\Sigma',\mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that $\mathbb{C}^{\mathbb{T}}(\tau') \in \mathrm{PR}(\Sigma)_{\mathbf{t}\,u\,v,s'}$.

Induction.

(3) Algebraic operations. If $\tau = \sigma(\tau_1, \ldots, \tau_k)$ for some $\sigma \in \Sigma'_{w,s}$ for some $w \in S^+$, for some $s \in S$, and for some $\tau_i \in T(\Sigma', \mathbb{X})_{s'_i}$ for $i = 1, \ldots, k = |w|$ wherein $w = s'_1 \cdots s'_k$ then

$$\mathbb{C}^{\mathrm{T}}(\tau) = \sigma \circ < \mathbb{C}^{\mathrm{T}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{T}}(\tau_k) >$$

if $\sigma \in \Sigma$ and

$$\mathbb{C}^{\mathrm{T}}(\tau) = \mathrm{U}_q^{\iota(p)} \circ \mathbb{C}_{\Phi}^{\star\mathrm{PR}}(\phi_{\iota(p)}) \circ < \mathbb{C}^{\mathrm{T}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{T}}(\tau_k) >$$

if $\sigma = f_{p,q} \in \{f_{1,1}, \ldots, f_{1,|\eta^R(1)|}, \ldots, f_{m,1}, \ldots, f_{m,|\eta^R(m)|}\}$ wherein $\mathbb{C}^{\star\mathrm{PR}}$ is defined as in Definition 59.

Well-definedness. The well-definedness of both cases follows by essentially the same argument as the well-definedness of $\mathbb{C}^{\mathrm{T}}$ in the same case and is omitted.

The Compiler $\mathbb{C}^{\star\mathrm{PR}}$. Using $\mathbb{C}^{\mathrm{T}}$ and $\mathbb{C}^{\mathrm{T}}$ we continue by defining the compiler $\mathbb{C}^{\star\mathrm{PR}}$ that maps RPREQ specifications into PR

Definition 59. For each $\Phi \in \mathrm{PREQ}(\Sigma, X)_{w,w'}^{l,m,\iota,\eta,\varsigma}$ for some $l, m, \iota, \eta, \varsigma, w$ and w' as defined in Definition 50 and for each $u, v \in S^+$ we define

$$\mathbb{C}_{\Phi,u,v}^{\star\mathrm{PR}} : \mathrm{RPREQ}(\Sigma', X)_{u,v} \to \mathrm{PR}(\Sigma)_{u,v}$$

(ambiguously denoted either $\mathbb{C}_{u,v}^{\star\mathrm{PR}}$ or just $\mathbb{C}^{\star\mathrm{PR}}$) wherein $\Sigma' = \Sigma \cup \mathcal{F}$ is the extended signature of Φ as defined in Definition 50 by analysis of the structural complexity of the specification $\phi \in \mathrm{RPREQ}(\Sigma, X)_{u,v}$ as follows: (Note that the termination properties and the well-definedness of this construction are considered at the end of the definition.)

(1) Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma, \mathbb{X})_s$ for some $s \in S$ then

$$\mathbb{C}^{\star\mathrm{PR}}(\phi) = \mathbb{C}_{\Phi,\mathbb{X},u,s}^{\mathrm{T}}(\tau)$$

wherein $u = s_1 \cdots s_n$ and $\mathbb{X} = \{x_1, \ldots, x_n\}$.

(2) Vector-Valued Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = < \tau_1, \ldots, \tau_m >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma, \mathbb{X})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ then

$$\mathbb{C}^{\star\mathrm{PR}}(\phi) = < \mathbb{C}_{\Phi,\mathbb{X},u,s'_1}^{\mathrm{T}}(\tau_1), \ldots, \mathbb{C}_{\Phi,\mathbb{X},u,s'_m}^{\mathrm{T}}(\tau_m) >$$

wherein $u = s_1 \cdots s_n$ and $\mathbb{X} = \{x_1, \ldots, x_n\}$.

(3) Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_1 \in T(\Sigma, \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma, \mathbb{X}')_s$, for some $s \in S$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y\}$ such that $t \in X_n$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathbb{C}^{\star \mathrm{PR}}(\phi) = *(\mathbb{C}^{\mathrm{T}}_{\Phi, \mathbb{X}, u, s}(\tau_1), \mathbb{C}^{\mathbb{T}}_{\Phi, \mathbb{X}', u, s, s}(\tau_2))$$

wherein $u = s_1, \ldots, s_n$, $\mathbb{X} = \{x_1, \ldots, x_n\}$, and $\mathbb{X}' = \{x_1, \ldots, x_n, t, Y\}$.

(4) Vector-Valued Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = < \tau_{1,1}, \ldots, \tau_{1,m} >;$$
$$f(t+1, x_1, \ldots, x_n) = < \tau_{2,1}, \ldots, \tau_{2,m} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_{1,j} \in T(\Sigma, \mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma, \mathbb{X}')_{s'_j}$, for some $s'_j \in S$ for $j = 1, \ldots, m > 1$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1, \ldots, Y_m\}$ such that $t \in X_n$ and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, m$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathbb{C}^{\star \mathrm{PR}}(\phi) =$$

$$*(< \mathbb{C}^{\mathrm{T}}_{\Phi, \mathbb{X}, u, s'_1}(\tau_{1,1}), \ldots, \mathbb{C}^{\mathrm{T}}_{\Phi, \mathbb{X}, u, s'_m}(\tau_{1,m}) >,$$
$$< \mathbb{C}^{\mathbb{T}}_{\Phi, \mathbb{X}', u, v, s'_1}(\tau_{2,1}), \ldots, \mathbb{C}^{\mathbb{T}}_{\Phi, \mathbb{X}', u, v, s'_m}(\tau_{2,m}) >)$$

wherein $u = s_1, \ldots, s_n$, $\mathbb{X} = \{x_1, \ldots, x_n\}$, $v = s'_1, \ldots, s'_m$, and $\mathbb{X}' = \{x_1, \ldots, x_n, t, Y_1, \ldots, Y_m\}$.

Termination. Notice that as previously discussed because both $\mathbb{C}^{\mathrm{T}}$ and $\mathbb{C}^{\star \mathrm{PR}}$, and $\mathbb{C}^{\mathbb{T}}$ and $\mathbb{C}^{\star \mathrm{PR}}$ are mutually recursive before we can show that $\mathbb{C}^{\star \mathrm{PR}}$ is well-defined we must first show that for any Φ and ϕ that $\mathbb{C}^{\star \mathrm{PR}}_{\Phi}(\phi)$ cannot give rise to a recursive call of $\mathbb{C}^{\star \mathrm{PR}}_{\Phi}(\phi)$. To see this notice that $\Phi \in \mathrm{PREQ}(\Sigma, X)$ and hence that $i \notin \mathrm{InTermsOf}(< \phi_1, \ldots, \phi_l; \iota; \eta; i >, i, \mathcal{F})$ for each $i \in \{1, \ldots, l\}$. Therefore, $\mathbb{C}^{\star \mathrm{PR}}(\phi_j)$ for each $j \in \{1, \ldots, l\}$ cannot give rise to a recursive call of $\mathbb{C}^{\star \mathrm{PR}}(\phi_j)$ and clearly as a consequence $\mathbb{C}^{\star \mathrm{PR}}(\phi)$ cannot give rise to a call of $\mathbb{C}^{\star \mathrm{PR}}(\phi_j)$ that in turn could give rise to a recursive call of $\mathbb{C}^{\star \mathrm{PR}}(\phi_j)$. Hence termination is guaranteed.
Well-Definedness. Since we are guaranteed termination the well-definedness of $\mathbb{C}^{\star \mathrm{PR}}$ follows directly from the hypotheses on Φ and the well-definedness of $\mathbb{C}^{\mathrm{T}}$ and $\mathbb{C}^{\mathbb{T}}$ and is omitted.

The Compiler $\mathbb{C}^{\mathrm{PR}}$. Finally we are now in a position to define the compiler $\mathbb{C}^{\mathrm{PR}} : \mathrm{PREQ}(\Sigma, X) \rightarrow \mathrm{PR}(\Sigma)$. Indeed, this is now straightforward using $\mathbb{C}^{\star \mathrm{PR}}$ as by definition for each PREQ specification $\Phi = < \phi_1, \ldots, \phi_l; \varsigma >$ it is $\phi_\varsigma \in \mathrm{RPREQ}(\Sigma', X)$ that provides the semantics for Φ.

Definition 60. We define the $S^+ \times S^+$-indexed family of compilers

$$\mathbb{C}^{\mathrm{PR}} = < \mathbb{C}^{\mathrm{PR}}_{u,v} : \mathrm{PREQ}(\Sigma, X)_{u,v} \to \mathrm{PR}(\Sigma)_{u,v} \mid u, v \in S^+ >$$

wherein for each $\Phi \in \mathrm{PREQ}(\Sigma, X)_{u,v}$ for any $u, v \in S^+$ the mapping

$$\mathbb{C}^{\mathrm{PR}}_{u,v} : \mathrm{PREQ}(\Sigma, X)_{u,v} \to \mathrm{PR}(\Sigma)_{u,v}$$

is defined as follows:

$$\mathbb{C}^{\mathrm{PR}}_{u,v}(\Phi) = \mathbb{C}^{\star \mathrm{PR}}_{\Phi,u,v}(\phi_c).$$

Well-definedness. As by hypothesis $\Phi \in \mathrm{PREQ}(\Sigma, X)$ by the well-definedness of $\mathbb{C}^{\star \mathrm{PR}}(\phi_c)$ we have $\mathbb{C}^{\mathrm{PR}}_{u,v}(\Phi) \in \mathrm{PR}(\Sigma)_{u,v}$ as required.

The Adequacy of PREQ. We now define the compiler $\mathbb{C}^{\mathrm{PREQ}} : \mathrm{PR}(\Sigma) \to \mathrm{PREQ}(\Sigma, X)$ that we gave an informal algorithmic description of in Section 5.2.2. As with the definition of the compiler $\mathbb{C}^{\mathrm{PR}}$ the compiler $\mathbb{C}^{\mathrm{PREQ}}$ is also defined in terms of a number of sub-compilers: $\mathbb{C}^{\star \mathrm{PREQ}}$, $\mathbb{C}^{\circ \mathrm{PREQ}}$, $\mathbb{C}^{\bullet \mathrm{PREQ}}$, $\mathbb{C}^{\nabla \mathrm{PREQ}}$ and the compiler $\mathbb{C}^{\mathrm{PR}_E}$. Of these sub-compilers it is $\mathbb{C}^{\circ \mathrm{PREQ}}$ and $\mathbb{C}^{\mathrm{PR}_E}$ that perform the most complex aspects of the compilation procedure.

In order to formally define $\mathbb{C}^{\circ \mathrm{PREQ}}$ that maps PR schemes into RPREQ specifications we employ the use of the normal form representation PR_E (see Definition 37). Indeed, the use of PR_E in essence enables us to reduce the compilation of PR schemes into RPREQ into the construction of either one or two appropriate terms. More precisely, recall that if $\alpha \in \mathrm{PR}_E(\Sigma)$ then either (1) α contains no applications of primitive recursion or (2) α contains one primitive recursion and is of the form $\alpha = *(\alpha_1, \alpha_2)$; that is, either (1) $\alpha \in \mathrm{PR}_D \subseteq \mathrm{PR}_E$ and represents a polynomial function or (2) $\alpha \in \mathrm{PR}_E$ and represents a primitive recursive function defined using two polynomial functions $\alpha_1 \in \mathrm{PR}_D$ and $\alpha_2 \in \mathrm{PR}_D$. Also recall that $\mathrm{PR}_E(A) = \mathrm{PR}_A(A)$ is precisely the class of functions that can be represented in RPREQ over algebra A. Therefore, (ignoring vector-valued functions that we discuss in a moment) in Case (1) the process of converting α into RPREQ can essentially be reduced to converting α to a single term τ – used in a simple RPREQ specification; and in Case (2) the process of converting α into RPREQ can essentially be reduced to converting α_1 and α_2 to two terms τ_1 and τ_2 respectively – both used in a primitive recursive RPREQ specification.

The compilation of PR_D schemes into terms is performed by the compiler $\mathbb{C}^{\star \mathrm{PREQ}}$. Indeed, as far as possible $\mathbb{C}^{\star \mathrm{PREQ}}$ is structured to be the inverse of the compilers $\mathbb{C}^{\mathrm{T}}$ and $\mathbb{C}^{\mathbb{T}}$ described in the previous section. In particular, $\mathbb{C}^{\star \mathrm{PREQ}}$ deals with the generation of the appropriate variables to represent each input by making the order of the sets $\mathbb{X} = \{x_1, \ldots, x_n\}$ and $\mathbb{X}' = \{t, x_1, \ldots, x_n, Y_1, \ldots, Y_m\}$ significant (see Definition 5.3.2). Thus, if $\alpha = \mathrm{U}^w_i$ then $\mathbb{C}^{\star \mathrm{PREQ}}(\alpha) = x$ wherein $x = t$ if $i = 1$; $x = x_{i-1}$ if $1 < i \leq n + 1$; and $x = Y_{i-n-1}$ if $n + 1 < i \leq 1 + n + m$.

Finally, returning to the question of vector-valued functions notice that in the following definition of $\mathbb{C}^{\star \mathrm{PREQ}}$ that (ignoring other indexing) in fact $\mathbb{C}^{\star \mathrm{PREQ}}$ is properly a family of compilers $\mathbb{C}^{\star \mathrm{PREQ}}_e$ for each $e \in \mathbb{N}^+$. The index e is designed to deal with the slight complication created with vector-valued functions in that regardless of the size of the co-domain of the PR scheme α,

we require the result of the compilation $\mathbb{C}^{*\mathrm{PREQ}}(\alpha)$ to produce a term τ that must by definition be single-valued. In more detail, to cope with this problem, essentially each compiler $\mathbb{C}_e^{*\mathrm{PREQ}}$ is only well-defined on a PR scheme α if α defines a function $F^\alpha = (F_1^\alpha, \ldots, F_k^\alpha)$ for some $k \geq e$. In particular, $\mathbb{C}_e^{*\mathrm{PREQ}}(\alpha)$ produces a term τ such that the semantics of τ is equivalent to F_e^α. Therefore, to produce a RPREQ scheme for the vector-valued function α as above, if $\mathrm{NPRSS}(\alpha) = 0$ then we simply compute the terms $\tau_i = \mathbb{C}_i^{*\mathrm{PREQ}}(\alpha)$ for $i = 1, \ldots, k$ and use these to make a vector-valued simple RPREQ specification; otherwise if $\alpha = *(\alpha_1, \alpha_2)$ then we compute the terms $\tau_{1,i} = \mathbb{C}_i^{*\mathrm{PREQ}}(\alpha_1)$ and $\tau_{2,i} = \mathbb{C}_i^{*\mathrm{PREQ}}(\alpha_2)$ for $i = 1, \ldots, k$ and use these to make a vector-valued primitive recursive RPREQ specification.

The Compiler $\mathbb{C}^{*\mathrm{PREQ}}$.

Definition 61. For each $\mathbb{X} = \{x_1, \ldots, x_{|u|}\} \subseteq X$ such that $x_i \in X_{u_i}$ for $i = 1, \ldots, |u|$ we define the $S^+ \times S^+ \times \mathbb{N}$-indexed family of mappings

$$\mathbb{C}^{*\mathrm{PREQ}} = < \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,n} : \mathrm{PR}_D(\Sigma) \to T(\Sigma, \mathbb{X})_{v_n} >$$

wherein for each $u, v \in S^+$, for each $n \in \{1, \ldots, |v|\}$

$$\mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,n} : \mathrm{PR}_D(\Sigma) \to T(\Sigma, \mathbb{X})_{v_n}$$

(ambiguously denoted $\mathbb{C}_n^{*\mathrm{PREQ}}$) is defined uniformly in (u, v) by induction on the structural complexity of a scheme $\alpha \in \mathrm{PR}_D(\Sigma)_{u,v}$ as follows:

(1) 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

$$\mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\alpha) = c^w.$$

Well-definedness. It is clear here that since by hypothesis $c \in \Sigma_{\lambda,s}$ and $v = v_1 = s$ we have $\mathbb{C}_1^{*\mathrm{PREQ}}(\alpha) \in T(\Sigma, \mathbb{X})_s$ as required.

(2) Algebraic Operations. If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$ then

$$\mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\alpha) = \sigma(x_1, \ldots, x_{|w|}).$$

Well-definedness. Again as by hypothesis $v = v_1 = s$ and also as by hypothesis we have $x_i \in X_{w_i}$ for $i = 1, \ldots, |w = u|$ and $\sigma \in \Sigma_{w,s}$ it is clear that $\mathbb{C}_1^{*\mathrm{PREQ}}(\alpha) \in T(\Sigma, \mathbb{X})_s$ as required.

(3) Projection Functions. If $\alpha = \mathrm{U}_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$ then

$$\mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\alpha) = x_i.$$

Well-definedness. It is clear here that since by hypothesis $x_i \in X_{w_i}$ for $i = 1, \ldots, |w = u|$ and $v = w_i = s$ we have $\mathbb{C}_1^{*\mathrm{PREQ}}(\alpha) \in T(\Sigma, \mathbb{X})_{w_i}$ as required.

Induction. Assume that for each $\alpha \in \mathrm{PR}_D(\Sigma)_{u,v}$ for some $u, v \in S^+$ that for any scheme $\alpha' \in \mathrm{PR}_D(\Sigma)_{u',v'}$ for some $u', v' \in S^+$ of less structural complexity than α if $\mathbb{X}' = \{x'_1, \ldots, x'_{|u'|}\}$ wherein $x'_j \in X_{u'_j}$ for $j = 1, \ldots, |u'|$ then for each $i \in \{1, \ldots, |v'|\}$

$$\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X}',u',v',i}(\alpha) \in T(\Sigma, \mathbb{X})_{v'_i}.$$

(4) **Vectorization.** If $\alpha = < \alpha_1, \ldots, \alpha_{|m|} >$ for some $m > 0$, for some $\alpha_i \in \mathrm{PR}_D(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1, \ldots, |u|$ then

$$\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,s_1\cdots s_m,n}(\alpha) = \mathbb{C}^{\star\mathrm{PREQ}}_{u,s_n,1}(\alpha_n).$$

Well-definedness. By the induction hypothesis $\mathbb{C}^{\star\mathrm{PREQ}}_1(\alpha_n) \in T(\Sigma, \mathbb{X})_{s_n}$. Therefore, it is clear that $\mathbb{C}^{\star\mathrm{PREQ}}_n(\alpha) \in T(\Sigma, \mathbb{X})_{s_n}$ as required.

(5) **Composition.** In this case $\alpha = \alpha_2 \circ \alpha_1$ for some $\alpha_1 \in \mathrm{PR}_D(\Sigma)_{u,w}$ and for some $\alpha_2 \in \mathrm{PR}_D(\Sigma)_{w,s}$ for some $u, w \in S^+$ and for some $s \in S$. As $\alpha \in \mathrm{PR}_D(\Sigma)$ we now consider two sub-cases:

(A) $w = s' \in S$.

(B) $w = s_1 \cdots s_{|w|} \in S^+$ **wherein** $|w| \geq 2$.

Sub-case (A) $w = s' \in S$. In this case

$$\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,s,1}(\alpha) = \begin{cases} c^w & \text{if } \alpha_2 = c^w, \\ \sigma(\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w,1}(\alpha_1)) & \text{if } \alpha_2 = \sigma, \text{ and} \\ \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w,1}(\alpha_1) & \text{if } \alpha_2 = \mathrm{U}^w_1. \end{cases}$$

Notice that by the hypothesis that $|w| = 1$ and the definition of $\mathrm{PR}_D(\Sigma)$ the scheme α must correspond to one of the above three cases.

Sub-Case (B) $w = s_1 \cdots s_{|w|} \in S^+$ **wherein** $|w| \geq 2$. In this case

$$\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,s,1}(\alpha) = \begin{cases} c^w & \text{if } \alpha_2 = c^w, \\ \sigma(\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w_1,1}(\alpha_{1,1}), \ldots, \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w_{|w|},1}(\alpha_{1,|w|}))) & \text{if } \alpha_2 = \sigma, \text{ and} \\ & \alpha_1 = < \alpha_{1,1}, \ldots, \alpha_{1,|w|} > \\ \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w_i,1}(\alpha_{1,i}) & \text{if } \alpha_2 = \mathrm{U}^w_i \text{ and} \\ & \alpha_1 = < \alpha_{1,1}, \ldots, \alpha_{1,|w|} >. \end{cases}$$

Notice that by the hypothesis that $|w| \geq 2$ and the definition of $\mathrm{PR}_D(\Sigma)$ the scheme α must correspond to one of the above three cases.

Well-Definedness. We consider the well-definedness of Sub-case (B) and leave Sub-case (A) that is similar to the reader.

We have three sub-sub-cases to consider:

(A) $\alpha_2 = c$. In this case since by hypothesis $c \in \Sigma_{\lambda,s}$ it is clear that $\mathbb{C}^{\star\mathrm{PREQ}}_n(\alpha) \in T(\Sigma, \mathbb{X})_s$ as required.

(B) $\alpha_2 = \sigma$. By the Induction Hypothesis $\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,w_{i,1}}(\alpha_{1,i}) \in T(\Sigma,\mathbb{X})_{w_i}$ for $i = 1,\ldots,|w|$ and by hypothesis $\sigma \in \Sigma_{w,s}$. Therefore, it is clear that $\mathbb{C}^{\star\mathrm{PREQ}}_{n}(\alpha) \in T(\Sigma,\mathbb{X})_s$ as required.

(C) $\alpha_2 = \mathrm{U}^w_i$. This case is similar to Case (B) and is omitted.

The Compiler $\mathbb{C}^{\diamond\mathrm{PREQ}}$.

Definition 62. We define the $\mathbb{N} \times S^+ \times S^+$ -indexed family of compilers

$$\mathbb{C}^{\diamond\mathrm{PREQ}} = < \mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v} : \mathrm{PR}(\Sigma)_{u,v} \to \mathrm{RPREQ}(\Sigma', X)_{u,v} \mid e \in \mathbb{N} \text{ and } u, v \in S^+ >$$

wherein $\Sigma' \subset \Sigma \cup \mathcal{F}$ and $\mathcal{F}$ is defined as in Lemma 19 as follows: for each $e \in \mathbb{N}$ for each $u \in S^+$ and for each $v \in S^+$ the mapping

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v} : \mathrm{PR}(\Sigma)_{u,v} \to \mathrm{RPREQ}(\Sigma', X)_{u,v}$$

(ambiguously denoted $\mathbb{C}^{\diamond\mathrm{PREQ}}_{e}$) is defined by the structural complexity of a scheme $\alpha \in \mathrm{PR}(\Sigma)_{u,v}$ as follows: (as the well-definedness of the various cases follows directly from the well-definedness of $\mathbb{C}^{\star\mathrm{PREQ}}$ and $\mathbb{C}^{\mathrm{PR}_E}$ and by Lemma 14 we leave this to the reader; also, notice that in each of the following cases $\mathbb{X} = \{x_1,\ldots,x_{|u|}\}$ wherein $x_i \in X_{u_i}$ for $i = 1,\ldots,|u|$)

(1) **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

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,w,s}(\alpha) \stackrel{def}{=} f(x_1,\ldots,x_{|w|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)).$$

(2) **Algebraic Operations.** If $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$ then

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,w,s}(\alpha) \stackrel{def}{=} f(x_1,\ldots,x_{|w|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)).$$

(3) **Projection Functions.** If $\alpha = \mathrm{U}^w_i$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$ then

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,w,w_i}(\alpha) \stackrel{def}{=} f(x_1,\ldots,x_{|w|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,w_{i,1}}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)).$$

Induction.

(4) **Vectorization.** If $\alpha = < \alpha_1,\ldots,\alpha_m >$ wherein $\alpha_i \in \mathrm{PR}(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1,\ldots,m$ then

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,s_1\cdots s_m}(\alpha) \stackrel{def}{=} f(x_1,\ldots,x_{|u|}) = < \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,s_{1,1}}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha_1)),\ldots,\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,s_{m,1}}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha_m)) > .$$

(5) **Composition.** If $\alpha = \alpha_2 \circ \alpha_1$ wherein $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,w}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{w,s}$ for some $u, w \in S^+$ and for some $s \in S$ then

$$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v}(\alpha) \stackrel{def}{=}$$

$$\begin{cases} f(x_1,\ldots,x_{|u|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)) & \text{if } |v| = 1, \text{ and} \\ f(x_1,\ldots,x_{|u|}) = < \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)),\ldots,\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,v,|v|}(\mathbb{C}^{\mathrm{PR}_E}_{e}(\alpha)) > & \text{if } |v| > 1. \end{cases}$$

(6) Simultaneous Primitive Recursion. If $\alpha = *(\alpha_1, \alpha_2)$ wherein $\alpha_1 \in \mathrm{PR}(\Sigma)_{u,v}$ and $\alpha_2 \in \mathrm{PR}(\Sigma)_{u\,u\,v,v}$ for some $u, v \in S^+$ then
$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v}(\alpha) \stackrel{def}{=}$

$$f(0, x_1, \ldots, x_{|w|}) = \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_1))$$
$$f(t+1, x_1, \ldots, x_{|w|}) = \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_2))$$

if $|v| = 1$ and
$\mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v}(\alpha) =$

$$f(0, x_1, \ldots, x_{|w|}) = < \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_1)), \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,|v|}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_1)) >$$
$$f(t+1, x_1, \ldots, x_{|w|}) = < \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_2)), \ldots, \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,|v|}(\mathbb{C}^{\mathrm{PR}_E}_e(\alpha_2)) >$$

if $|v| > 1$ wherein $\mathbb{X}' = \{t, x_1, \ldots, x_{|u|}, Y_1, \ldots, Y_{|v|}\}$.

Using $\mathbb{C}^{\diamond\mathrm{PREQ}}$ we now define the compiler $\mathbb{C}^{\bullet\mathrm{PREQ}}$ that given a scheme α produces a partial PREQ specification essentially of the form $\Phi = < \mathbb{C}^{\diamond\mathrm{PREQ}}(\alpha) >$.

The Compiler $\mathbb{C}^{\bullet\mathrm{PREQ}}$.

Definition 63. For each $e \in N^+$ and for each $u, v \in S^+$ we define

$$\mathbb{C}^{\bullet\mathrm{PREQ}}_{e,u,v} : \mathrm{PR}(\Sigma)_{u,v} \to \mathrm{PREQ}_1(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v}$$

(ambiguously denoted $\mathbb{C}^{\bullet\mathrm{PREQ}}_e$) as follows:

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v}) \quad \mathbb{C}^{\bullet\mathrm{PREQ}}_{e,u,v}(\alpha) = \Phi = < \phi; \iota; \eta; \varsigma >$$

wherein

$$\phi = \mathbb{C}^{\diamond\mathrm{PREQ}}_{e,u,v}(\alpha);$$
$$\iota = \{e \mapsto 1\};$$
$$\eta = \{e \mapsto (u^1, v^1), e + r^\alpha(1) \mapsto (u^2, v^2), \ldots, e + r^\alpha(k) \mapsto (u^{k+1}, v^{k+1})\}$$

wherein $u^1 = u$, $v^1 = v$, and for $i = 1, \ldots k = \mathrm{NPPRSS}'(\alpha)$ we have $u^{i+1} = Dom(\mathrm{PPRSS}'(\alpha, i))$ and $v^{i+1} = Ran(\mathrm{PPRSS}'(\alpha, i))$;

$$\varsigma = 1;$$

and

$$m = e + k.$$

Well-definedness. First, notice that by the well-definedness of $\mathbb{C}^{\diamond\mathrm{PREQ}}$ we immediately have $\phi \in \mathrm{RPREQ}(\Sigma', X)_{u,v}$ wherein

$$\begin{aligned}\Sigma' = \Sigma &\bigcup \{f_{e,1}\} \bigcup \cdots \bigcup \{f_{e,|v^1|}\} \\ &\bigcup \{f_{e+r^\alpha(1),1}\} \bigcup \cdots \bigcup \{f_{e+r^\alpha(1),|v^2|}\} \\ &\vdots \\ &\bigcup \{f_{e+r^\alpha(k),1}\} \bigcup \cdots \bigcup \{f_{e+r^\alpha(k),|v^{k+1}|}\}.\end{aligned}$$

Secondly, again by the well-definedness of $\mathbb{C}^{\diamond\text{PREQ}}$ we have

$$1 \notin \text{InTermsOf}(\Phi, 1, \Sigma' - \Sigma).$$

Finally, as

$$Dom\downarrow(\iota) = \{e\}$$

and

$$u = u^{\iota(e)} = u^1 = \eta^D(e)$$

and

$$v = v^{\iota(e)} = v^1 = \eta^R(e)$$

by definition it is clear that $\Phi \in \text{PREQ}_1(\Sigma, X)_{u,v}$ as required.

The Compiler $\mathbb{C}^{\nabla\text{PREQ}}$. Following the structure of the informal algorithmic description presented in Section 5.2.2 our strategy is to now define the compiler $\mathbb{C}^{\nabla\text{PREQ}}$ to use $\mathbb{C}^{\bullet\text{PREQ}}$ to produce partial PREQ specifications for each top-level primitive recursion in α and join these specifications together using the function $\biguplus$ (see Definition 53) to make a single (total defined) PREQ specification with the same semantics as α. As we indicated during the discussion of the high-level algorithm to perform this procedure the main technical problem is to ensure that each additional function symbol from the signature $\mathcal{F}$ is given an appropriate index. To achieve this formally we use the function r (Definition 43). The correctness of this indexing procedure is address in Lemma 26.

Definition 64. We define the $\mathbb{N}^+ \times S^+ \times S^+$-indexed family of compilers

$$\mathbb{C}^{\nabla\text{PREQ}} = < \mathbb{C}^{\nabla\text{PREQ}}_{e,u,v} : \text{PR}(\Sigma)_{u,v} \rightarrow \text{PREQ}(\Sigma, X)_{u,v} \mid e \in \mathbb{N}^+ \text{ and } u, v \in S^+ >$$

wherein for each $e \in \mathbb{N}^+$ and for each $u, v \in S^+$ the mapping

$$\mathbb{C}^{\nabla\text{PREQ}}_{e,u,v} : \text{PR}(\Sigma)_{u,v} \rightarrow \text{PREQ}(\Sigma, X)_{u,v}$$

ambiguously denoted $\mathbb{C}^{\nabla\text{PREQ}}_e$ is defined as follows:
$(\forall \alpha \in \text{PR}(\Sigma)_{u,v}) \quad \mathbb{C}^{\nabla\text{PREQ}}_e(\alpha) =$

$$\begin{aligned}
&\mathbb{C}^{\bullet\text{PREQ}}_e(\alpha) \biguplus_1 \mathbb{C}^{\nabla\text{PREQ}}_{r^\alpha(1)+e}(\text{PPRSS}'(\alpha, 1)) \\
&\qquad \biguplus_1 \mathbb{C}^{\nabla\text{PREQ}}_{r^\alpha(2)+e}(\text{PPRSS}'(\alpha, 2)) \\
&\qquad \vdots \\
&\qquad \biguplus_1 \mathbb{C}^{\nabla\text{PREQ}}_{r^\alpha(k)+e}(\text{PPRSS}'(\alpha, k))
\end{aligned}$$

wherein $k = \text{NPPRSS}'(\alpha)$.

Lemma 26. *If $\alpha \in PR(\Sigma)_{u,v}$ for some $u, v \in S^+$ then for each $e \in \mathbb{N}^+$*

$$\mathbb{C}^{\nabla PREQ}_{e,u,v}(\alpha) \stackrel{def}{=} \Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)^{l,m,\iota,\eta,\varsigma}_{u,v}$$

wherein

$$\phi_1 = \mathbb{C}^{\diamond PREQ}_{e,u^1,v^1}(\alpha);$$

$$\phi_i = \mathbb{C}^{\diamond PREQ}_{e+i-1,u^i,v^i}(PPRSS(\alpha, i-1))$$

for $i = 2, \ldots, l = (r^\alpha(k) = K = NPPRSS(\alpha)) + 1$ *wherein* $k = NPPRSS'(\alpha)$;

$$\iota = \{e \mapsto 1, e+1 \mapsto 2, \ldots, e+K \mapsto l\};$$

$$\eta = \{e \mapsto (u^1, v^1), e+1 \mapsto (u^2, v^2), \ldots, e+K \mapsto (u^{K+1}, v^{K+1})\}$$

wherein $u^1 = u$, $v^1 = v$, *and for* $i = 2, \ldots, K$ *we have* $u^i = Dom(PPRSS(\alpha, i-1))$ *and* $v^i = Ran(PPRSS(\alpha, i-1))$;

$$\varsigma = 1;$$

and

$$m = e + K.$$

Notice that if $e = 1$ *then* Φ *is standard.*

Proof. By induction on $K = \mathrm{NPPRSS}(\alpha)$. We consider two basis cases:
(a) $K = 0$, and
(b) $K = 1$.

(a) $K = 0$. In this case by definition

$$\mathbb{C}^{\nabla \mathrm{PREQ}}_e(\alpha) = \mathbb{C}^{\bullet \mathrm{PREQ}}_e(\alpha)$$

and hence the proof is immediate from Lemma 71 and the definition of $\mathbb{C}^{\bullet \mathrm{PREQ}}$.

(b) $K = 1$. In this case as by hypothesis $\mathrm{NPPRSS}(\alpha) = 1$ by Lemma 13 we have $\mathrm{NPPRSS}'(\alpha) = 1$ and therefore by Lemma 17 we have $r^\alpha = \{1 \mapsto 1\}$ and $\mathrm{PPRSS}'(\alpha, 1) = \mathrm{PPRSS}(\alpha, 1)$ and hence

$$\Phi = \Phi_1 \uplus_1 \Phi_2$$

wherein

$$\Phi_1 = \mathbb{C}^{\bullet \mathrm{PREQ}}_e(\alpha) = < \phi_1; \iota_1; \eta_1; 1 >$$

and

$$\Phi_2 = \mathbb{C}^{\nabla \mathrm{PREQ}}_{e+1}(\mathrm{PPRSS}(\alpha, 1)) = \mathbb{C}^{\bullet \mathrm{PREQ}}_{e+1}(\mathrm{PPRSS}(\alpha, 1)) = < \phi_2; \iota_2; \eta_2; 1 >$$

wherein

$$\phi_1 = \mathbb{C}^{\diamond \mathrm{PREQ}}_1(\alpha)$$
$$\phi_2 = \mathbb{C}^{\diamond \mathrm{PREQ}}_{e+1}(\mathrm{PPRSS}(\alpha, 1))$$
$$\iota_1 = \{e \mapsto 1\}$$
$$\iota_2 = \{e+1 \mapsto 1\}$$
$$\eta_1 = \{e \mapsto (u^1, v^1), e+1 \mapsto (u^2, v^2)\}$$

and

$$\eta_2 = \{e+1 \mapsto (u^2, v^2)\}.$$

Notice, that by the well-definedness of $\mathbb{C}^{\bullet\text{PREQ}}$ we immediately have $\Phi_1 \in \text{PREQ}_1(\Sigma, X)_{u^1,v^1}$ and $\Phi_2 \in \text{PREQ}_1(\Sigma, X)_{u^2,v^2}$. Therefore as $Dom\downarrow(\iota_1) \bigcap Dom\downarrow(\iota_2) = \varnothing$ and $\eta_1 \supseteq \eta_2$ by the well-definedness of $\biguplus_1$ we have

$$\Phi \in \text{PREQ}_1(\Sigma, X)^{2,e+1,\iota,\eta_1,1}_{u,v}$$

wherein

$$\iota = \{e \mapsto 1, e+1 \mapsto 2\};$$

that is,

$$\Phi =< \phi_1, \phi_2; \iota; \eta_1; 1 > .$$

Therefore to complete our proof in this case it remains to show that Φ is totally-defined; that is, for each

$$i \in \bigcup_{j=1}^{j=2} \text{InTermsOf}(\Phi, j, \Sigma', -\Sigma)$$

we have $\iota(i)\downarrow$. This is obvious as by the definition of $\mathbb{C}^{\diamond\text{PREQ}}$ we have

$$\bigcup_{j=1}^{j=2} \text{InTermsOf}(\Phi, j, \Sigma', -\Sigma) = e+1$$

and by the definition of ι we have $\iota(e+1) = 2$.

Induction Hypothesis. Assume for any scheme $\alpha' \in \text{PR}(\Sigma)_{u',v'}$ for some $u', v' \in S^+$ that if $\text{NPPRSS}(\alpha') \leq K'$ for some $K' \in \mathbb{N}$ then for each $e \in \mathbb{N}^+$

$$\mathbb{C}^{\nabla\text{PREQ}}_{e,u',v'}(\alpha') = \Phi' =< \phi'_1, \ldots, \phi'_{l'}; \iota'; \eta'; \varsigma' >$$

wherein

$$\phi'_1 = \mathbb{C}^{\diamond\text{PREQ}}_{e,x^1,y^1}(\alpha');$$

$$\phi'_i = \mathbb{C}^{\diamond\text{PREQ}}_{e+i-1,x^i,y^i}(\text{PPRSS}(\alpha', i-1))$$

for $i = 2, \ldots, l' = (r^{\alpha'}(k') = \text{NPPRSS}(\alpha') = K') + 1$ wherein $k' = \text{NPPRSS}'(\alpha')$;

$$\iota' = \{e \mapsto 1, e+1 \mapsto 2, \ldots, e+K' \mapsto l'\};$$

$$\eta = \{e \mapsto (x^1, y^1), e+1 \mapsto (x^2, y^2), \ldots, e+K' \mapsto (x^{K'+1}, y^{K'+1})\}$$

wherein $x^1 = u'$, $y^1 = v'$, and for $i = 2, \ldots, K'$ we have $x^i = Dom(\text{PPRSS}(\alpha', i-1))$ and $y^i = Ran(\text{PPRSS}(\alpha', i-1))$;

$$\varsigma' = 1;$$

and

$$m' = e + K'.$$

Induction Step. We must show for any scheme $\alpha'' \in \mathrm{PR}(\Sigma)_{u'',v''}$ for some $u'', v'' \in S^+$ that if $\mathrm{NPPRSS}(\alpha'') = K'' = K' + 2$ then for each $e \in \mathbb{N}^+$

$$\mathbb{C}^{\nabla\mathrm{PREQ}}_{e,u'',v''}(\alpha'') = \Phi'' = < \phi''_1, \ldots, \phi''_{l''}; \iota''; \eta''; \varsigma'' >$$

wherein

$$\phi''_1 = \mathbb{C}^{\diamond\mathrm{PREQ}}_{e,\overline{x}^1,\overline{y}^1}(\alpha'');$$

$$\phi''_i = \mathbb{C}^{\diamond\mathrm{PREQ}}_{e+i-1,\overline{x}^1,\overline{y}^1}(\mathrm{PPRSS}(\alpha'', i-1))$$

for $i = 2, \ldots, l'' = (r^{\alpha''}(k'') = \mathrm{NPPRSS}(\alpha'') = K'') + 1$ wherein $k'' = \mathrm{NPPRSS}'(\alpha'')$;

$$\iota'' = \{e \mapsto 1, e+1 \mapsto 2, \ldots, e + K'' \mapsto l''\};$$

$$\eta = \{e \mapsto (\overline{x}^1, \overline{y}^1), e+1 \mapsto (\overline{x}^2, \overline{y}^2), \ldots, e + K'' \mapsto (\overline{x}^{K''+1}, \overline{y}^{K''+1})\}$$

wherein $\overline{x}^1 = u''$, $\overline{y}^1 = v''$, and for $i = 2, \ldots K''$ we have $\overline{x}^i = \mathit{Dom}(\mathrm{PPRSS}(\alpha'', i-1))$ and $\overline{y}^i = \mathit{Ran}(\mathrm{PPRSS}(\alpha'', i-1))$;

$$\varsigma'' = 1;$$

and

$$m'' = e + K''.$$

Notice that by definition

$$\Phi'' = \mathbb{C}^{\nabla\mathrm{PREQ}}_{e}(\alpha'') = \mathbb{C}^{\bullet\mathrm{PREQ}}_{e}(\alpha'') \uplus_1 \mathbb{C}^{\nabla\mathrm{PREQ}}_{e+r^{\alpha''}(1)}(\mathrm{PPRSS}'(\alpha'', 1))$$

$$\vdots$$

$$\uplus_1 \mathbb{C}^{\nabla\mathrm{PREQ}}_{e+r^{\alpha''}(k'')}(\mathrm{PPRSS}'(\alpha'', k''))$$

wherein for each $i \in \{1, \ldots, k''\}$ by Lemma 15 $\mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', i)) \leq K''$ and therefore by the Induction Hypothesis we have

$$\mathbb{C}^{\nabla\mathrm{PREQ}}_{e+r^{\alpha''}(i)}(\mathrm{PPRSS}'(\alpha'', i)) \in \mathrm{PREQ}(\Sigma, X)^{l',m',\iota',\eta',1}_{\overline{x}^i,\overline{y}^i}$$

wherein

$$l^i = k^i + 1 = \mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', i)) + 1,$$
$$m^i = e + r^{\alpha''}(i) + k^i,$$
$$\iota^i = \{e + r^{\alpha''}(i) \mapsto 1, e + r^{\alpha''} + 1 \mapsto 2, \ldots, e + r^{\alpha''}(i) + k^i \mapsto l^i\},$$

and

$$\eta^i = \{e + r^{\alpha''}(i) \mapsto (\overline{u}^i, \overline{v}^i), e + r^{\alpha''}(1) + 1 \mapsto (\overline{u}^{i+1}, \overline{v}^{i+1}), \ldots,$$
$$e + r^{\alpha''}(i) + k^i \mapsto (\overline{u}^{k^i}, \overline{v}^{k^i})\}$$

wherein $\overline{u}^1 = Dom(\mathrm{PPRSS}'(\alpha'', i))$, $\overline{v}^1 = Ran(\mathrm{PPRSS}'(\alpha'', i))$, $\overline{w}^{j-1} = Dom(\mathrm{PPRSS}(\mathrm{PPRSS}'(\alpha'', i)))$ and $\overline{v}^{j-1} = Ran(\mathrm{PPRSS}(\mathrm{PPRSS}'(\alpha'', i)))$ for $j = 2, \ldots, (k^i = \mathrm{NPPRSS}(\mathrm{PPRSS}(\alpha'', i))) + 1$. Furthermore, notice that as by hypothesis $\mathrm{NPPRSS}'(\alpha'') \geq 2$ that by Lemma 18 for each $i \in \{1, \ldots, \mathrm{NPPRSS}'(\alpha'') - 1\}$ we have

$$r^{\alpha''}(i) + \mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', i)) + 1 = r^{\alpha''}(i+1)$$

and

$$r^{\alpha''}(\mathrm{NPPRSS}'(\alpha'')) + \mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', \mathrm{NPPRSS}'(\alpha''))) = \mathrm{NPPRSS}(\alpha'').$$

Therefore for each $i, j \in \{1, \ldots, r^{\alpha''}(k'')\}$ we have

$$i < j \Longrightarrow ((Dom \downarrow \iota^i \bigcap Dom \downarrow \iota^j = \varnothing) \wedge (\mathrm{dom}(\eta^i) \bigcap \mathrm{dom}(\eta^j) = \varnothing))$$

and hence it is clear that

$$\begin{aligned} \Psi = \mathbb{C}^{\nabla \mathrm{PREQ}}_{r^{\alpha''}(1)+e}(\mathrm{PPRSS}'(\alpha'', 1)) \uplus_1 & \mathbb{C}^{\nabla \mathrm{PREQ}}_{e+r^{\alpha''}(1)+e}(\mathrm{PPRSS}'(\alpha'', 2)) \\ & \vdots \\ \uplus_1 & \mathbb{C}^{\nabla \mathrm{PREQ}}_{r^{\alpha''}(k'')+e}(\mathrm{PPRSS}'(\alpha'', k'')) \end{aligned}$$

is well-defined.

In addition, notice now that as $K'' \geq 2$ by Lemma 13 we have $\mathrm{NPPRSS}'(\alpha'') \geq 2$ and hence by Lemma 17

$$r^{\alpha''}(1) = 1$$

and by Lemma 18 for each $i \in \{2, \ldots, k'' - 1\}$ we have

$$r^{\alpha''}(i) + \mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', i)) + 1 = r^{\alpha''}(i+1)$$

and

$$r^{\alpha''}(k'') + \mathrm{NPPRSS}(\mathrm{PPRSS}'(\alpha'', k'')) = \mathrm{NPPRSS}(\alpha'') = K''$$

and therefore

$$\Psi = < \phi''_1, \ldots, \phi''_{K''}; \iota''; \eta''; \varsigma'' >$$

wherein

$$\phi''_i = \mathbb{C}^{\diamond \mathrm{PREQ}}_{e+i}(\mathrm{PPRSS}(\alpha'', i))$$

for $i = 1, \ldots, l'' = K''$;

$$\iota'' = \{e+1 \mapsto 1, e+2 \mapsto 2, \ldots, e+K'' \mapsto l''\};$$

$$\eta'' = \{e+1 \mapsto (\overline{x}^2, \overline{y}^2), e+2 \mapsto (\overline{x}^3, \overline{y}^3), \ldots, e+K'' \mapsto (\overline{x}^{k''+1}, \overline{y}^{k''+1})\}.$$

Finally, notice by definition that

$$\mathbb{C}^{\bullet \mathrm{PREQ}}_e(\alpha'') = < \phi; \iota; \eta; 1 > \in \mathrm{PREQ}_1(\Sigma, X)^{l, m, \iota, \eta, 1}_{u'', v''}$$

wherein

$$\begin{aligned}\phi &= \mathbb{C}_e^{\diamond \mathrm{PREQ}}(\alpha''),\\ l &= 1,\\ m &= e + k'',\\ \iota &= \{e \mapsto 1\},\end{aligned}$$

and

$$\eta = \{e \mapsto (\overline{x}^1, \overline{y}^1), e + r^{\alpha''}(1) \mapsto (\overline{x}^2, \overline{y}^2), \ldots, e + r^{\alpha''}(k'') \mapsto (\overline{x}^{k''+1}, \overline{y}^{k''+1})\}.$$

Consequently, as by the definition of $r^{\alpha''}$ we have $\mathrm{PPRSS}'(\alpha'', i) = \mathrm{PPRSS}(\alpha'', r^{\alpha''}(i))$ for $i = 1, \ldots, k''$ it is clear that

$$Dom \downarrow \iota \bigcap Dom \downarrow \iota'' = \varnothing$$

and

$$\eta'' \supseteq \eta$$

and hence that

$$\Phi'' = \mathbb{C}_e^{\nabla \mathrm{PREQ}}(\alpha'') = \mathbb{C}_e^{\bullet \mathrm{PREQ}}(\alpha'') \uplus_1 \Psi$$

is well-defined and furthermore that $\Phi'' \in \mathrm{PREQ}(\Sigma, X)_{u'',v''}^{K'', e+K'', \iota'', \eta'', 1}$ as required.

□

Finally, we are now in a position to define the required compiler $\mathbb{C}^{\mathrm{PREQ}}$ that is essentially $\mathbb{C}^{\nabla \mathrm{PREQ}}$, but ensures that the resulting PREQ specification is standard (see Definition 52).

The Compiler $\mathbb{C}^{\mathrm{PREQ}}$.

Definition 65. We define the $S^+ \times S^+$-indexed family of compilers

$$\mathbb{C}^{\mathrm{PREQ}} = < \mathbb{C}_{u,v}^{\mathrm{PREQ}} : \mathrm{PR}(\Sigma)_{u,v} \to \mathrm{PREQ}_1(\Sigma, X)_{u,v} \mid u, v \in S^+ >$$

wherein for each $u \in S^+$ and for each $v \in S^+$ the mapping $\mathbb{C}_{u,v}^{\mathrm{PREQ}} : \mathrm{PR}(\Sigma)_{u,v} \to \mathrm{PREQ}(\Sigma, X)_{u,v}$ (ambiguously denoted $\mathbb{C}^{\mathrm{PREQ}}$) is defined as follows:

$$(\forall \alpha \in \mathrm{PR}(\Sigma)_{u,v}) \quad \mathbb{C}^{\mathrm{PREQ}}(\alpha) = \mathbb{C}_1^{\nabla \mathrm{PREQ}}(\alpha).$$

This completes the definitions of our formal compilers. In the following section we state two lemmata regarding the formal properties of $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$ that we can use to deduce Theorem 5.3.3. The proofs of these lemmata can be found in Appendix B.

5.3.4 Proof of Theorem 10

Lemma 27. *If*

$$\Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)_{u,v}$$

for some $u, v \in S^+$ *then*

$$(\forall a \in A^u) \quad [\![\Phi]\!]_A(a) = [\![\mathbb{C}^{PR}(\Phi)]\!]_A(a).$$

Lemma 28. *For each* $\alpha \in PR(\Sigma)_{u,v}$ *for some* $u, v \in S^+$ *if* $\Phi = \mathbb{C}^{PREQ}(\alpha)$ *then*

$$(\forall a \in A^u) \quad [\![\alpha]\!]_A = [\![\Phi]\!]_A.$$

For convenience we first re-state Theorem 10.

Theorem 10. *If* A *is some standard algebra then*

(1)

$$(\forall \Phi \in PREQ(\Sigma, X))(\exists \alpha \in PR(\Sigma)) \quad [\![\alpha]\!]_A = [\![\Phi]\!]_A.$$

(2)

$$(\forall \alpha \in PR(\Sigma))(\exists \Phi \in PREQ(\Sigma, X)) \quad [\![\Phi]\!]_A = [\![\alpha]\!]_A.$$

Proof of Statement (1) of Theorem 10. This now follows as a simple corollary to Lemma 27.

For each $\phi \in \mathrm{PREQ}(\Sigma, X)$ if we define $\alpha = \mathbb{C}^{\mathrm{PR}}(\phi) \in \mathrm{PR}(\Sigma)$ then by Lemma 27 we have $[\![\phi]\!]_A = [\![\alpha]\!]_A$ as required, and hence PREQ is sound with respect to $\mathbb{PR}$.

Proof of Statement (2) of Theorem 10. This now follows as a simple corollary to Lemma 28.

For each $\alpha \in \mathrm{PR}(\Sigma)$ if we define $\Phi = \mathbb{C}^{\mathrm{PREQ}}(\alpha) \in \mathrm{PREQ}(\Sigma, X)$ then by Lemma 28 we have $[\![\alpha]\!]_A = [\![\Phi]\!]_A$ as required, and hence PREQ is adequate with respect to $\mathbb{PR}$.

5.4 The Properties of PREQ Specifications as TRSs

To conclude this chapter we complete the technical development of PREQ by turning our attention to the proof of Theorem 11 that is concerned with the properties of PREQ specifications when interpreted as left-to-right term re-writing systems. Theorem 11 plays a significant role in our formulation of techniques for the automated verification of STs in Chapter 7.

5.4.1 Overview

While essentially PREQ specifications are a restricted class of first-order systems of equations we cannot directly convert a PREQ specification into a TRS (by orienting the equations as left-to-right re-write rules) because of the following two technical problems: (1) PREQ specifications may be vector-valued; and (2) (R)PREQ specification involve the use of the special variable symbols Y_i.

As we will require the use of PREQ specification as both proper 'first-order' systems of equations and as equivalent TRSs in Section 5.4.2 we begin by defining two functions: the function EQCON that converts a PREQ specification into a proper equational specification; and the function TRCON that converts a PREQ specifications into TRS. In particular, TRCON is defined to be exactly EQCON except that it orients each equation as a left-to-right re-write rule.

5.4.2 Interpreting PREQ Specifications as TRSs

Definition 66. For each $u, v \in S^+$, for each $l, m \in \mathbb{N}$, for each $\iota : \{1, \ldots, m\} \rightarrow \{1, \ldots, l\}$, for each $\eta : \{1, \ldots, m\} \rightarrow S^+ \times S^+$ and for each $\varsigma \in \{1, \ldots, l\}$ as defined in Definition 50 we define

$$\mathrm{EQCON}_{u,v}^{l,m,\iota,\eta,\varsigma} : \mathrm{PREQ}_1(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma} \rightarrow \mathrm{EQ}(\Sigma'', X)$$

(ambiguously denoted EQCON) wherein

$$\Sigma'' = \{f_{\iota(i),j} \mid f_{i,j} \in \mathcal{F} \wedge \iota(i)\downarrow\}$$

wherein $\mathcal{F}$ is also defined as in Definition 50 by

$$(\forall \Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in \mathrm{PREQ}(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma}) \quad \mathrm{EQCON}(\Phi) = \bigcup_{i=1}^{i=l} \mathrm{EQCON}'_{l,m,i,\iota,\eta}(\phi_i)$$

wherein for each $l, m \in \mathbb{N}$, for each $i \in \{1, \ldots, l\}$, for each $\iota : \{1, \ldots, m\} \rightarrow \{1, \ldots, l\}$ and for each $\eta : \{1, \ldots, m\} \rightarrow S^+ \times S^+$ as defined above we define

$$\mathrm{EQCON}'_{l,m,i,\iota,\eta} : \mathrm{RPREQ}(\Sigma', X) \rightarrow \mathrm{EQ}(\Sigma'', X)$$

(ambiguously denoted EQCON') by the structural complexity of a scheme $\phi \in \mathrm{RPREQ}(\Sigma', X)$ as follows: (in the following definition $\Sigma' = \Sigma \cup \mathcal{F}$)

(1) Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma', \mathbb{X})_s$ for some $s \in S$ then

$$\mathrm{EQCON}'(\phi) = \{f_{i,1}(x_1, \ldots, x_n) = \overline{\tau}\}$$

wherein

$$\overline{\tau} = \tau \bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}].$$

(2) Vector-Valued Simple Specifications. If

$$\phi \stackrel{def}{=} f(x_1, \ldots, x_n) = << \tau_1, \ldots, \tau_{n'} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma', \mathbb{X})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, n' > 1$ wherein $\mathbb{X} = \{x_1, \ldots, x_n\}$ then

$$\mathrm{EQCON}'(\phi) = \bigcup_{j=1}^{j=n'} \{f_{i,j}(x_1, \ldots, x_n) = \overline{\tau}_j\}$$

wherein for $j = 1, \ldots, n'$

$$\overline{\tau}_j = \tau_j \bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}].$$

(3) Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ for some $\tau_1 \in T(\Sigma', \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma, \mathbb{X}')_s$ for some $s \in S$ wherein $\mathbb{X}' = \mathbb{X} \cup \{t, Y\}$ wherein $t \in X_{\mathbf{n}}$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathrm{EQCON}'(\phi) = \{f_{i,1}(0, x_1, \ldots, x_n) = \overline{\tau}_1, f_{i,1}(Succ(t), x_1, \ldots, x_n) = \overline{\tau}_2\}$$

wherein

$$\overline{\tau}_1 = \tau_1 \bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}]$$

and

$$\overline{\tau}_2 = \tau_2 (\bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}])[Y/f_{i,1}(t, x_1, \ldots, x_n)].$$

(4) Vector-Valued Primitive Recursive Specifications. If

$$\phi \stackrel{def}{=} f(0, x_1, \ldots, x_n) = < \tau_{1,1}, \ldots, \tau_{1,n'} >;$$
$$f(t+1, x_1, \ldots, x_n) = < \tau_{2,1}, \ldots, \tau_{2,n'} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$, for some $\tau_{1,j} \in T(\Sigma', \mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma', \mathbb{X}')_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, n' > 1$ wherein $\mathbb{X} = \{x_1, \ldots, x_n\}$ and $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1, \ldots, Y_{n'}\}$ wherein $t \in X_{\mathbf{n}}$ and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, m$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$ then

$$\mathrm{EQCON}'(\phi) = \bigcup_{j=1}^{j=n'} \{f_{i,j}(0, x_1, \ldots, x_n) = \overline{\tau}_{1,j}, f_{i,j}(Succ(t), x_1, \ldots, x_n) = \overline{\tau}_{2,j}\}$$

wherein

$$\overline{\tau}_{1,j} = \tau_{1,j} \bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}]$$

and

$$\overline{\tau}_{2,j} = \tau_2(\bigcup_{p=1}^{p=m} \bigcup_{j=1}^{j=|\eta_R(p)|} [f_{p,j}/f_{\iota(p),j}]) \bigcup_{q=1}^{q=n'} [Y_q/f_{i,q}(t, x_1, \ldots, x_n)].$$

Well-Definedness. Essentially, EQCON and EQCON′ eliminate the use of the function ι in a PREQ specification Φ (see Section 5.3.2) and replace the special variable symbol Y (or $Y_1, \ldots, Y_{n'}$ depending on the form of Φ) from each constituent primitive recursive RPREQ making-up Φ. However, we leave it as an exercise for the reader to deduce formally that both EQCON and EQCON′ give a well-defined equivalent equational specification.

We now use EQCON to define the function TRCON that converts PREQ specifications into equivalent TRSs by orienting the equations generated by EQCON as left-to-right re-write rules. Formally:

Definition 67. For each $u, v \in S^+$, for each $l, m \in \mathbb{N}$, for each $\iota : \{1, \ldots, m\} \to \{1, \ldots, l\}$, for each $\eta : \{1, \ldots, m\} \to S^+ \times S^+$ and for each $\varsigma \in \{1, \ldots, l\}$ as defined in Definition 50 we define

$$\mathrm{TRCON}_{u,v}^{l,m,\iota,\eta,\varsigma} : \mathrm{PREQ}_1(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma} \to \mathrm{TRS}(\Sigma'', X)$$

(ambiguously denoted EQCON) wherein Σ'' is defined as in the previous definition by

$$(\forall \Phi = < \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in \mathrm{PREQ}_1(\Sigma, X)_{u,v}^{l,m,\iota,\eta,\varsigma})$$

$$\mathrm{TRCON}(\Phi) = \{\rho = (l \mapsto r) \mid e = (l = r) \in \mathrm{EQCON}(\Phi)\}.$$

Using TRCON we now set about establishing that for each $\Phi \in \mathrm{PREQ}_1(\Sigma, X)$ the TRS $R = \mathrm{TRCON}(\Phi)$ is complete.

5.4.3 TRSs as Constructor Systems

Before we proceed we recall one further concept form the theory of term re-writing: *constructor systems* (also see Section 2.3.12).

Definition 68. Let $R \subseteq \mathrm{TRS}(\Sigma, X)$ be any TRS. If there exists a rule $\rho = (r \mapsto l) \in R$ such that $r = f(\tau_1, \ldots, \tau_n)$ for some terms τ_i for $i = 1, \ldots, n \in \mathbb{N}^+$ and for some algebraic operation $f \in \Sigma$ then we say that f is either a *defined symbol of* Σ or simply just a *defined symbol* if Σ is either understood or unimportant.

If $f \in \Sigma$ is *not* a defined symbol then we say that f is either a *constructor symbol of* Σ or simply just a *constructor symbol* if Σ is either understood or unimportant.

A signature that has been partitioned into defined symbols and constructor symbols is referred to in the literature as a *constructor system.* Indeed, constructor systems are used widely for the analysis of the particular properties of TRSs including most notably *modular properties.*

The list of references on this subject is extensive including: Toyama [1987a], Toyama [1987b], Middeldorp and Toyama [1991], Kurihara and Ohuchi [1992b], Kurihara and Ohuchi [1992a], and Toyama *et al.* [1989]; and in general is a subtle and complicated problem. Moreover, as pointed out in Fernández and Jouannaud [1993] the subject at present lacks an overall coherent theory, although Fernández and Jouannaud [1993] goes some way to resolving this problem.

In the sequel we will use constructor systems to analyse the termination and confluence properties of particular classes of TRSs.

Transitive Closure and Separability. Given a collection of signatures $\Sigma_i \supseteq \Sigma$ for $i = 1, \ldots, n \in \mathbb{N}^+$ we now present a formal mechanism for establishing which symbols from $\Sigma_j - \Sigma$ for some $j \in \{1, \ldots, n\}$ are shared by some Σ_l for some $l \in \{1, \ldots, n\}$ such that $l \neq j$. In particular, we define a formal mechanism for establishing the transitive closure of this property.

Definition 69. Let Σ be any S-sorted signature and let Σ_i for $i = 1, \ldots, n^+$ be some signature such that

$$\Sigma_i \subseteq \{f_1, \ldots, f_k\} \bigcup \Sigma$$

for some function symbols f_i for $i = 1, \ldots, k \in \mathbb{N}^+$. For each $n, m \in \{1, \ldots, k\}$ we define the extended signature

$$\Sigma_m^n \subseteq \bigcup_{j=1}^{j=n} \Sigma_j$$

by case analysis on the value of n as follows:

$$\Sigma_m^1 = \Sigma_m - (\Sigma \bigcup \{f_m\});$$

and for each $n \in \{2, \ldots, k\}$

$$\Sigma_m^n = \Sigma_m^1 \bigcup_{j \in \mathbb{F}_m} \Sigma_j^{n-1}$$

wherein $\mathbb{F}_m = \{x \mid f_x \in \Sigma_m\}$.

Example 17. If

$$\Sigma_1 = \{f_1, f_2\} \bigcup \Sigma,$$

$$\Sigma_2 = \{f_2, f_4\} \bigcup \Sigma,$$

$$\Sigma_3 = \{f_3\} \bigcup \Sigma,$$

$$\Sigma_4 = \{f_4, f_5\} \bigcup \Sigma$$

and

$$\Sigma_5 = \{f_5, f_3\} \bigcup \Sigma$$

then

$$\Sigma_1^1 = \{f_2\},$$

$$\Sigma_1^2 = \{f_2, f_4\},$$

$$\Sigma_1^3 = \{f_2, f_4, f_5\}$$

and

$$\Sigma_1^4 = \Sigma_1^5 = \{f_2, f_3, f_4, f_5\}.$$

By analysing the transitive closure of collections of signatures based on the occurrence of particular function symbols we are now able to define an important property that we refer to as *separability*.

Definition 70. Let Σ be any S-sorted signature and let Σ_i for $i = 1, \ldots, n^+$ be some signature such that

$$\Sigma_i \subseteq \{f_1, \ldots, f_k\} \bigcup \Sigma$$

for some function symbols f_i for $i = 1, \ldots, k \in \mathbb{N}^+$. If for each $i = 1, \ldots, k$

$$\Sigma_i^k \bigcap \{f_i\} = \varnothing$$

then we say that $\Sigma_1, \ldots, \Sigma_k$ are *separable*.

Example 18. If $\Sigma_1, \ldots, \Sigma_5$ are defined as in the previous example then they are separable. However, if Σ_3 were defined by

$$\Sigma_3 = \{f_1, f_3\} \bigcup \Sigma$$

then $\Sigma_1, \ldots, \Sigma_5$ would not be separable as

$$\Sigma_1^5 = \{f_1, f_2, f_3, f_4, f_5\};$$

that is $f_1 \in \Sigma_1^5$.

Our strategy now is to first show that by considering a TRS $R \subseteq \mathrm{TRS}(\Sigma, X)$ as an indexed collection of one-rule TRSs $R_i \subseteq \mathrm{TRS}(\Sigma_i, X)$, for some $\Sigma_i \subseteq \Sigma$ for $i = 1, \ldots, n \in \mathbb{N}^+$, we can analyse the transitive closure properties of the signatures $\Sigma_1, \ldots, \Sigma_n$, based on the occurrence of defined symbols, to identify classes of TRSs that are complete. To prove Theorem 11 we will use the fact that TRSs created from PREQ specifications satisfy precisely these properties.

In order to do this we require one final function that eliminates all immediate applications of primitive recursion from a TRS.

Definition 71. Let $\mathcal{R} \subseteq \mathrm{TRS}(\Sigma', X)$ wherein Σ' is a constructor system with $\{f_1, \ldots, f_k\}$ as its defined symbols such that each $f_i : s_i, s_{i,1}, \ldots, s_{i,n_i} \rightarrow s_i'$, for some $s_{i,j} \in S$ for $i = 1, \ldots, k$, for $j = 1, \ldots, n_i \geq 1$, and for some $s_i, s_i' \in S$. Furthermore, let $\mathcal{R}$ be defined by

$$\begin{aligned} \mathcal{R} = \{ & f_1(t_1, x_{1,1}, \ldots, x_{1,n_1}) \mapsto \tau_1, \\ & \vdots \\ & f_k(t_k, x_{k,1}, \ldots, x_{k,n_k}) \mapsto \tau_k \} \end{aligned}$$

wherein $t_i \in \mathrm{T}(\Sigma, X)_{s_i}$ for $i = 1, \ldots, k$, $x_{i,j} \in X_{s_{i,j}}$ for $i = 1, \ldots, k$ and for $j = 1, \ldots, n_i$, and

$$\tau_i \in \mathrm{T}(\Sigma_i, X)_{s'}$$

for some $\Sigma_i \subseteq \Sigma'$. Also, let $o : \Sigma' \to \mathbb{N}$ be any injection.

For each o and for each $\mathcal{R}$ as above we define

$$\mathrm{PR\_STRIP}^o : \mathrm{TRS}(\Sigma', X) \to \mathrm{TRS}(\Sigma', X)$$

(ambiguously $\mathrm{PR\_STRIP}(\mathcal{R})$ when o is either understood or unimportant) as follows:

$$\begin{aligned}\mathrm{PR\_STRIP}(\mathcal{R}) = \{&f_1(t_1, x_{1,1}, \ldots, x_{1,n_1}) \to \tau'_1,\\ &\vdots\\ &f_k(t_k, x_{k,1}, \ldots, x_{k,n_k}) \to \tau'_k\}\end{aligned}$$

wherein for $i = 1, \ldots, k$

$$\tau'_i \in \mathrm{T}(\Sigma', X)_{s'}$$

is defined by

$$\tau'_i = \begin{cases} \tau_i & \text{if } t_i \neq Succ(x) \text{ for some } x \in X_{\mathbf{n}}, \text{ and} \\ \tau_i \bigcup_{j=1}^{j=k} [f_j(x, x_{j,1}, \ldots, x_{j,n_k})/c_j] & \text{otherwise} \end{cases}$$

wherein c_j is some constant of type (λ, s'_j) from Σ' satisfying $o(c_j) < o(c)$ for each $c \in \Sigma_{\lambda, s'_j}$ such that $c \neq c_j$. Notice that c_j is guaranteed to exists by the hypothesis that in this thesis we only consider non-void signatures.

In the sequel we will refer to $\mathrm{PR\_STRIP}(\mathcal{R})$ as the *PR-stripped version of* $\mathcal{R}$.

We now begin presenting the preliminary lemmata that we will require to establish our main result. Recall that in this thesis we assume that Σ is any S-sorted signature, X is any S-indexed collection of variable symbols such that Σ and X are pairwise disjoint and that neither Σ nor X contain the distinguished symbol f and any of the distinguished symbols $f_i, f_{i,j}$ for each $i, j \in \mathbb{N}^+$.

Lemma 29. *If $r = (f(x_1, \ldots, x_n) \to \tau)$ wherein the function symbol f is of type $(s_1, \ldots, s_n, s)$ for some $s_i \in S$ for $i = 1, \ldots, n \geq 1$ and for some $s \in S$, and $x_i \in X_{s_i}$ for $i = 1, \ldots, n$ are distinct variables and $\tau \in T(\Sigma, \{x_1, \ldots, x_n\})_s$ then the one rule TRS $\mathcal{R} = \{r\} \in TRS(\Sigma \bigcup \{f\}, X)$ is orthogonal and strongly terminating.*

Lemma 30. *If $\mathcal{R} \subseteq TRS(\Sigma \bigcup \{f_1, \ldots, f_k\}, X)$ wherein the distinguished function symbol $f_i : s_{i,1}, \ldots, s_{i,n_i} \to s_i$ for some $s_{i,j} \in S$ for $i = 1, \ldots, k$ and for $j = 1, \ldots, n_i \geq 1$, and for some $s_i \in S$ for $i = 1, \ldots, k$ and is defined by*

$$\begin{aligned}\mathcal{R} = \{&f_1(x_{1,1}, \ldots, x_{1,n_1}) \to \tau_1,\\ &\vdots\\ &f_k(x_{k,1}, \ldots, x_{k,n_k}) \to \tau_k\}\end{aligned}$$

wherein $x_{i,j} \in X_{s_{i,j}}$ for $i = 1, \ldots, k$ and for $j = 1, \ldots, n_i$, and

$$\tau_i \in T(\Sigma_i, \{x_{i,1}, \ldots, x_{i,n_i}\})$$

for some $\Sigma_i \subseteq \Sigma \bigcup \{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_k\}$ for $i = 1, \ldots, k$ such that $\Sigma_1, \ldots, \Sigma_k$ are separable then $\mathcal{R}$ is orthogonal and strongly terminating.

Proof. By induction on the number $k \in \mathrm{N}^{+}$. We leave the details to the reader. (Notice that the basis case follows immediately by Lemma 29.)

□

Lemma 31. *If* $\mathcal{R} \subseteq TRS(\Sigma \bigcup \{f\}, X)$ *wherein the distinguished function symbol* $f : \mathbf{n}\, s_1, \ldots, s_n \rightarrow s'$ *for some* $s_i \in S$ *for* $i = 1, \ldots, n \geq 1$, *and for some* $s' \in S$ *and is defined by*

$$\mathcal{R} = \{f(0, x_1, \ldots, x_n) \rightarrow \tau_1, f(Succ(x), x_1, \ldots, x_k) \rightarrow \tau_2\}$$

wherein $x_i \in X_{s_i}$ *for* $i = 1, \ldots, n$, $x \in X_{\mathbf{n}}$ *is some variable distinct from* x_i *for* $i = 1, \ldots, n$,

$$\tau_1 \in T(\Sigma, \{x_1, \ldots, x_n\}),$$

and

$$\tau_2 \in T(\Sigma', \{x, x_1, \ldots, x_n\})$$

for some $\Sigma' \subseteq \Sigma \bigcup \{f\}$ *such that any term* $\tau' \subseteq \tau_2$ *with* f *as the outermost function symbol is of the form* $\tau' = f(x, x_1, \ldots, x_n)$ *then* $\mathcal{R}$ *is orthogonal and strongly terminating.*

Discussion. We now use these results to establish our main technical lemma of this section: Lemma 32. In particular, Lemma 32 details the necessary conditions under which particular classes of TRSs are orthogonal and strongly terminating and essentially identifies precisely the properties of PREQ specifications when they are converted into TRSs. Specifically, notice that Conditions (1) to (4) of the lemma simply restrict the definitions of the defined symbols occurring in the signature of $\mathcal{R}$ to be either one rule – defining a polynomial function – or to a pair of rules – defining a function by an immediate application of primitive recursion. Condition (5) restricts any recursive definitions to be strictly those defined by primitive recursion; that is, Condition (5) eliminates the possibility of any non-terminating reduction sequences.

Lemma 32. *If* $\mathcal{R} \subseteq TRS(\Sigma \bigcup \{f_1, \ldots, f_n\}, X)$ *wherein the distinguished function* $f_i : s_i, s_{i,1}, \ldots, s_{i,n_i} \rightarrow s_i'$) *for some* $s_{i,j} \in S$ *for* $i = 1, \ldots, n$ *and for* $j = 1, \ldots, n_i \geq 1$, *and for some* $s_i, s_i' \in S$ *for* $i = 1, \ldots, n$ *and is defined by*

$$\begin{aligned} \mathcal{R} = \{ & f_{l_1}(t_1, x_{1,1}, \ldots, x_{1,n_1}) \rightarrow \tau_1, \\ & \vdots \\ & f_{l_k}(t_k, x_{k,1}, \ldots, x_{k,n_k}) \rightarrow \tau_k\} \end{aligned}$$

wherein

(1) $x_{i,j} \in X_{s_{i,j}}$ *for* $i = 1, \ldots, k$ *and for* $j = 1, \ldots, n_i$;

(2) for $i = 1, \ldots, k$ *we have either* $t_i = x_i \in X_{s_i}$ *wherein either* x_i *is distinct from those variables defined in Case (1) above or* $t_i = 0 \in T(\Sigma)_{\mathbf{n}}$ *or* $t_i = Succ(x_i) \in T(\Sigma)_{\mathbf{n}}$;

(3) $l_i \in \{1, \ldots, k\}$ *for* $i = 1, \ldots, k$ *are defined such that for each* $p, q \in \{1, \ldots, k\}$ *we have*

$$l_p = l_q \Longleftrightarrow p = q \vee (t_p = 0 \wedge t_q = Succ(x_q)) \vee (t_q = 0 \wedge t_p = Succ(x_p));$$

(4) for $i = 1, \ldots, k$

(A) if $t_i = x_i$ *then*

$$\tau_i \in T(\Sigma_i, \{x_i, x_{i,1}, \ldots, x_{i,n_i}\})_{s'_i}$$

wherein $\Sigma_i \subseteq \Sigma \cup \{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_k\}$;

(B) if $t_i = 0$ *then*

$$\tau_i \in T(\Sigma_i, \{x_{i,1}, \ldots, x_{i,n_i}\})_{s'_i}$$

wherein $\Sigma_i \subseteq \Sigma \cup \{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_k\}$;

(C) if $t_i = Succ(x_i)$ *then*

$$\tau_i \in T(\Sigma_i, \{x_i, x_{i,1}, \ldots, x_{i,n_i}\})_{s'_i}$$

wherein $\Sigma_i \subseteq \Sigma \cup \{f_1, \ldots, f_k\}$ *and any term* $\tau' \subseteq \tau_i$ *with* f_i *as the outermost function symbol* τ' *is of the form* $\tau' = f_i(x_i, x_{i,1}, \ldots, x_{i,n_i})$;

(5) if $\Sigma'_1, \ldots, \Sigma'_k$ *are the signatures from PR_STRIP*$(\mathcal{R})$ *as defined in Definition 71 then* $\Sigma'_1, \ldots, \Sigma'_k$ *are separable*

then $\mathcal{R}$ *is orthogonal and strongly terminating.*

Based on these five conditions of the two properties that such specifications have (orthogonality and strong termination), only strong termination is non-trivial to establish. However, this property is itself straightforward to prove by induction on the number of equations occurring in the TRSs of interest $\mathcal{R}$. Consequently as a concession to conserving space will only present a proof sketch of how to establish strong termination and leave the remaining details to the reader.

Proof of Lemma 32. By induction on $k \in N^+$.
Basis. We consider two basis cases:
(A) $k = 1$, **and**
(B) $k = 2$.
Case (A) $k = 1$. This follows immediately by Lemma 29.
Case (B) $k = 2$. In this case we must consider two sub-cases:
(a) $l_1 \neq l_2$, **and**
(b) $l_1 = l_2$.
Sub-Case (a) $l_1 \neq l_2$. This follows immediately by Lemma 30.
Sub-Case (b) $l_1 = l_2$. This follows immediately by Lemma 31.
Induction Step. Notice now that as the only mutually recursive rules in $\mathcal{R}$ are defined by primitive recursion (this is guaranteed by the separability of any PR-stripped version of $\mathcal{R}$). Consequently, for any reduction sequence under $\mathcal{R}$ wherein $\mathcal{R}$ contains $k+1$ rules and for any redex t it must be the case that t is always re-written in a finite number of steps to a term t' such that for all possible remaining reduction sequences only $k' \leq k$ rules from $\mathcal{R}$ can be applied to t'. If we let $\mathcal{R}'$ denote this subset of $\mathcal{R}$ with only these k' rules then we can observe that the normal form of t' under $\mathcal{R}$ and $\mathcal{R}'$ must be the same and furthermore by the Induction Hypothesis $\mathcal{R}'$

is strongly terminating. Therefore, $\mathcal{R}$ must be strongly terminating as required.

□

To conclude this chapter we can now prove Theorem 11 using Lemma 32 and the following result from Bergstra and Tucker [1992].

Lemma 33. *(Lemma 2.4.3 of Bergstra and Tucker [1992]) If $\mathcal{R}$ is an orthogonal and weakly terminating TRS then $\mathcal{R}$ is complete.*

For convenience we first re-state Theorem 11.

Theorem 11. *If $\Phi \in PREQ(\Sigma, X)$ and $\mathcal{R} = TRCON(\Phi) \subseteq TRS(\Sigma, X)$; that is, if $\mathcal{R}$ is the term re-writing system formed from Φ by orienting each equation in Φ as a left-to-right re-write rule then $\mathcal{R}$ is complete.*

Proof of Theorem 11. Notice that for any PREQ specification Φ the equivalent term re-writing system $\mathcal{R}$ = TRCON(Φ) is of precisely the form required to satisfy Conditions (1) to (5) of Lemma 32. In particular, notice that we can satisfy Condition (5) by the fact that $\Phi \in \text{PREQ}(\Sigma, X) \Longrightarrow \Phi \in \text{PREQ}_1(\Sigma, X)$ and hence

$$(\forall i \in \{1, \ldots, l\})$$

$$i \notin InTermsOf(< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma >, i, \mathcal{F})$$

wherein $\mathcal{F}$ is defined as in Definition 46 and is precisely the defined symbols of $\mathcal{R}$. Therefore, since for any TRS strong termination implies weak termination by Lemma 33 we have that $\mathcal{R}$ is complete as required.

□

Chapter 6

ASTRAL

Language is a kind of human reason, which has its own internal logic of which man knows nothing.

Claude Lévi-Strauss

6.1 Introduction

As we indicated in Section 3.10.4 the main theoretical consideration that we face in the design of the abstract syntax and semantics of ASTRAL is to reconcile the mathematical advantages of Cartesian form specification with the more natural style of applicative form specification that is associated with stream processing. In particular, we will require that a user of ASTRAL may specify applicative form definitions, while a formal denotational semantics is achieved using an equivalent Cartesian form definition in PREQ. However, in addition to these theoretical considerations we also wish to design a user-friendly implementation of ASTRAL based on the abstract mathematical syntax, and address the practical issues that arise in the development of this syntax. In order to accommodate these two aims we proceed as follows:

First, in Section 6.2 we define an abstract mathematical formalization of ASTRAL that is similar to PREQ, but is syntactically tailored to the representation of STs. Specifically, ASTRAL allows the specification and composition of non-trivial AFSTs and hence is appropriate for the modular and hierarchical specification techniques discussed in Section 3.10.4.

Secondly, in Section 6.3 we use this formal syntactic definition of ASTRAL to define a compiler that maps ASTRAL specifications into equivalent equational representation in PREQ. This compiler, denoted $\mathbb{C}^{\text{ASTRAL}}$, makes use of the compilers $\mathbb{C}$, $\mathbb{C}^{\text{PR}}$ and $\mathbb{C}^{\text{PREQ}}$ (see Definitions 33, 60 and 65 respectively) and as such is able to eliminate any (composite) definitions made using AFSTs to derive a single, equivalent Cartesian form definition. As a consequence in Section 6.4 we are able to use the resulting Cartesian form PREQ schema created by $\mathbb{C}^{\text{ASTRAL}}$ as the formal semantics of our original ASTRAL specifications.

Finally, in Sections 6.5 and 6.7 to complete the development of ASTRAL we present a high-level prototype BNF based on the abstract syntax presented in Section 6.2 and comment on the features that an actual implementation of ASTRAL will incorporate. In particular, we use the stream processing primitives and constructs that we examined in Chapter 3 as examples to demonstrate that ASTRAL provides a general purpose and natural specification formalism for a broad class of stream processing systems.

6.1.1 Preliminary Notation and Definitions

Before we proceed with the development of the technical material in this chapter we require some further preliminaries. Specifically, we require a further modification of the language PR, denoted PR^{v}, that has two additional vectorization construction. However, while we will define the abstract syntax of this languages we will not specify an independent semantics. Rather, as PR^{v} is a computationally conservative expansion of PR we will have a compiler:

$$\backslash^{\text{PR}^{\text{v}}} : \text{PR}^{\text{v}}(\Sigma) \rightarrow \text{PR}(\Sigma)$$

to define the semantics of $\text{PR}^{\text{v}}(\Sigma)$ schemes.

Furthermore, in order to eliminate instances of Cartesian composition that are created when ASTRAL is compiled into PREQ we will define a compiler

$$\mathcal{C} : \text{PR}(\underline{\Sigma}) \rightarrow \text{PR}(\underline{\Sigma})$$

that is a more general formulation of the compiler $\mathcal{C}$. In particular, $\mathcal{C}'$ is more general than $\mathcal{C}$ in that it will perform vector-valued Cartesian composition and also combine vector-valued composition and ordinary composition in a single operation.

Our final preliminaries for this chapter concern the use of higher-order signatures and terms.

The Language PR$^{\vee}$. In order to simplify the compilation of ASTRAL into PREQ it is convenient to define one further computationally conservative expansion of PR. The language PR$^{\vee}$ includes two additional and more general formulations of vectorization as function-building tools: '$\ll \ldots \gg$' that allows vector-valued schemes to be vectorized; and '$\lll \ldots \ggg$' that allows vector-valued schemes with different domains to be vectorized. We can define PR$^{\vee}$ formally by extending PR with the following additional induction clauses:

(5a) Type One Extended Vectorization. If $\alpha = \ll \alpha_1, \ldots, \alpha_m \gg$ for some $\alpha_i \in \mathrm{PR}^{\vee}(\Sigma)_{u,v^i}$ for some $u, v^i \in S^+$ for $i = 1, \ldots, m \geq 1$ then $\alpha \in \mathrm{PR}^{\vee}(\Sigma)_{u,v^1 \cdots v^m}$.

(5b) Type Two Extended Vectorization. If $\alpha = \lll \alpha_1, \ldots, \alpha_m \ggg$ for some $\alpha_i \in \mathrm{PR}^{\vee}(\Sigma)_{u^i,v^i}$ for some $u^i, v^i \in S^+$ for $i = 1, \ldots, m \geq 1$ then $\alpha \in \mathrm{PR}^{\vee}(\Sigma)_{u^1 \cdots u^m, v^1 \cdots v^m}$.

Notice here that in Case (5b) as each α_i may have a different domain, in contrast with standard vectorization, the domain of α is the concatenation of the words u^i for $1 = 1, \ldots, m$.

We now define the compiler $\backslash^{\mathrm{PR}^{\vee}}$ that maps PR$^{\vee}$ schemes into standard PR schemes, although, we only formally define the compilation of schemes defined by extended vectorization. We leave the other cases and the well-definedness of this compiler to the reader. However, as the definition of $\backslash^{\mathrm{PR}^{\vee}}$ is quite technical in the case of type two extended vectorization we begin by motivating the structure of the PR schemes the compiler produces (Definition 72). The PR schemes constructed in the case of type one extended vectorization are similar.

First, notice that as each α_i of type (u^i, v^i) may be vector-valued, in general we replace a PR$^{\vee}$ scheme $\lll \alpha_1, \ldots, \alpha_m \ggg$ with a scheme $< \beta_1, \ldots, \beta_n >$ wherein $n = |v^1 \cdots v^m|$; that is, we replace each α_i with $|v^i|$ schemes each representing an individual co-ordinate of α_i. Secondly, notice that in each of the four cases the scheme $< \mathrm{U}^u_{k_1}, \ldots, \mathrm{U}^u_{k_{|u|}} >$ selects the correct co-ordinates of $u = u^1, \ldots, u^m$ as input to each β_j; that is, if β_j represents a particular co-ordinate of α_i then $< \mathrm{U}^u_{k_1}, \ldots, \mathrm{U}^u_{k_{|u|}} >$ selects precisely u^i from u. Thirdly, notice that if $|\alpha_i| > 1$ then the scheme $\mathrm{U}^u_{k_1}$ selects the appropriate co-ordinate as output for each β_j. Finally, notice that we only inductively apply the compiler $\backslash^{\mathrm{PR}^{\vee}}$ to a scheme α_i if it itself contains further occurrences of extended vectorization.

Definition 72. We define

$$\backslash^{\mathrm{PR}^{\vee}} = < \backslash^{\mathrm{PR}^{\vee}}_{u,v} : \mathrm{PR}^{\vee}(\Sigma)_{u,v} \to \mathrm{PR}(\Sigma)_{u,v} \mid u, v \in S^+ >$$

wherein each $\backslash^{\mathrm{PR}^{\vee}}_{u,v} : \mathrm{PR}^{\vee}(\Sigma)_{u,v} \to \mathrm{PR}(\Sigma)_{u,v}$ (ambiguously denoted $\backslash^{\mathrm{PR}^{\vee}}$) is defined uniformly in (u, v) by induction on the structural complexity of a scheme $\alpha \in \mathrm{PR}^{\vee}(\Sigma)$. In particular, the extended vectorization induction cases are defined as follows:

(5a) Type One Extended Vectorization. If $\alpha = \ll \alpha_1, \ldots, \alpha_m \gg$ for some $\alpha_i \in \mathrm{PR}^{\vee}(\Sigma)_{u,v^i}$ for some $u, v^i \in S^+$ for $i = 1, \ldots, m \geq 1$ then

$$\backslash^{\mathrm{PR}^{\vee}}(\alpha) = < \beta_1, \ldots, \beta_n >$$

wherein $n = |v^1 \cdots v^m|$ and for $i = 1, \ldots, n$

$$\beta_i = \begin{cases} \alpha_j & \text{if } |v^j| = 1 \text{ and } \alpha_j \in \mathrm{PR}(\Sigma); \\ \lambda^{\mathrm{PR}^*}_{u^j,v^j}(\alpha_j) & \text{if } |v^j| = 1 \text{ and } \alpha_j \notin \mathrm{PR}(\Sigma); \\ U^{v^j}_k \circ \lambda^{\mathrm{PR}^*}_{u^j,v^j}(\alpha_j) & \text{if } |v^j| > 1 \text{ and } \alpha_j \in \mathrm{PR}(\Sigma); \text{ and} \\ U^{v^j}_k \circ \alpha_j & \text{otherwise} \end{cases}$$

wherein $j = \mu\, l.[|v^1 \cdots v^l| \geq i]$ and $k = i - |v^1 \cdots v^{j-1}|$.

(5b) Type Two Extended Vectorization. If $\alpha = \lll\, \alpha_1, \ldots, \alpha_m \,\ggg$ for some $\alpha_i \in \mathrm{PR}^*(\Sigma)_{u^i,v^i}$ for some $u^i, v^i \in S^+$ for $i = 1, \ldots, m \geq 1$ then

$$\lambda^{\mathrm{PR}^*}(\alpha) = < \beta_1, \ldots, \beta_n >$$

wherein $n = |v^1 \cdots v^m|$ and for $i = 1, \ldots, n$

$$\beta_i = \begin{cases} \alpha_j \circ < U^u_{k_1}, \ldots, U^u_{|u^j|} > & \text{if } |v^j| = 1 \text{ and } \alpha_j \in \mathrm{PR}(\Sigma); \\ \lambda^{\mathrm{PR}^*}_{u^j,v^j}(\alpha_j) \circ < U^u_{k_1}, \ldots, U^u_{k_{|u^j|}} > & \text{if } |v^j| = 1 \text{ and } \alpha_j \notin \mathrm{PR}(\Sigma); \\ U^{v^j}_k \circ \lambda^{\mathrm{PR}^*}_{u^j,v^j}(\alpha_j) \circ < U^u_{k_1}, \ldots, U^u_{|u^j|} > & \text{if } |v^j| > 1 \text{ and } \alpha_j \in \mathrm{PR}(\Sigma); \text{ and} \\ U^{v^j}_k \circ \alpha_j \circ < U^u_{k_1}, \ldots, U^u_{|u^j|} > & \text{otherwise} \end{cases}$$

wherein $j = \mu l.[|v^1 \cdots v^l| \geq i]$, $k = i - |v^1 \cdots v^{j-1}|$ and $k_p = |u^1 \cdots u^j| + p$ for $p = 1, \ldots, |u^j|$.

An Extended Cartesian Composition Compiler. We now define an extended Cartesian composition compiler, ambiguously denoted $\mathcal{C}$, based on the Cartesian composition compiler $\mathcal{C}$ of Chapter 4, that we will use in the formalization of a denotational semantics for ASTRAL. The reader should consult Section 4.5.4 for an explanation of the operation of the compiler $\mathcal{C}$. This extended compiler is precisely what we need to eliminate the instances of Cartesian composition created by the composition of AFSTs in ASTRAL.

The difference between the compiler $\mathcal{C}$ and the compiler $\mathcal{C}$ is that the latter allows us to combine the actions of ordinary composition and Cartesian composition into one 'primitive' by selecting the particular type of composition required for each co-ordinate of the co-domain of a particular function. For example, if $\alpha \in \mathrm{PR}(\underline{\Sigma})$ is of type (u, v) wherein $u = \underline{s_1}\, s_2\, \underline{s_3}\, s_4$ for some $s_i \in S$ for $i = 1, \ldots, 4$ and $\beta_1 \in \mathrm{PR}(\underline{\Sigma})_{z_1,s_2}$ and $\beta_2 \in \mathrm{PR}(\underline{\Sigma})_{t\, z_2, s_3}$ for some $z_1, z_2 \in \mathrm{PR}(\underline{\Sigma})$ then

$$\alpha' = \mathcal{C}^{\{\beta_1,\beta_2\},\{o,c\}}_{u,v}(\alpha) \in \mathrm{PR}(\underline{\Sigma})_{\underline{s_1}\, z_1\, z_2\, s_4, v}$$

is the scheme such that

$$(\forall a = (a_1, a_2, a_3, a_4) \in \underline{A}^{\underline{s_1}\, z_1\, z_2\, s_4}) \quad [\![\alpha']\!]_{\underline{A}}(a) = [\![\alpha]\!]_{\underline{A}}(a_1, [\![\beta_1]\!]_{\underline{A}}(a_2), \lambda t.[\![\beta_2]\!]_{\underline{A}}(t, a_3), a_4).$$

Thus, $\mathcal{C}$ has allowed us simultaneously compose α with β_1 at the second co-ordinate of α's domain and to perform Cartesian composition on α and β_2 at the third co-ordinate of α's domain. Indeed, (while for simplicity we have not demonstrated the fact in our example) $\mathcal{C}$ also allows us to perform vector-valued Cartesian composition formalizing the idea behind the semantic

proof of Theorem 7 from Theorem 8 in Section 4.5.1. However, for convenience we formalize vector-valued Cartesian composition into an effective procedure using an intermediate compiler denoted $\hat{\circ}$. As such we begin with the definition of $\hat{\circ}$.

Formalizing Vector-Valued Cartesian Composition. The operation of the compiler $\hat{\circ}$ is essentially straightforward in that it formalizes the process of repeated Cartesian composition using the successive co-ordinates of a vector-valued function. However, the compiler is highly technical as we must be precise about the types of the co-ordinates of the domain of the resulting scheme that is created as the process is iterated. This is reflected in the establishment of the well-definedness of $\hat{\circ}$ that requires a proof by induction. Therefore, before we present the formal definition, we motivate the formulation of $\hat{\circ}$ with a few comments on the structure of the schemes it produces (Definition 73).

Notice in the induction step that $\hat{\circ}^{n'',m''}_{\tilde{u},v'',z''}(\alpha'',\beta'')$ is defined inductively using essentially three new schemes derived from the original structure of α'' and β''; that is, the schemes γ, $\beta''_{l''+1}$ and κ. Of these schemes γ is the vectorization of the first to the l''th co-ordinate functions of the function computed by β'', and $\beta''_{l''+1}$ represents the $l''+1$th co-ordinate function of the function computed by β''. As such by two applications of the Induction Hypothesis

$$\hat{\circ}^{p,p}_{\tilde{u}_1\cdots\tilde{u}_{n''-1}z''\tilde{u}_{m''}\tilde{u}_{m''+1}\cdots\tilde{u}_k,v'',z''}(\hat{\circ}^{n'',m''-1}_{\tilde{u},v'',z''}(\alpha'',\gamma),\beta''_{l''+1})$$

must represent the composition of each co-ordinate of β'' with α''. The slight complication with this inductive technique is that (similarly to our semantic proof of Theorem 7 from Theorem 8 in Section 4.5.1) as we have composed the co-ordinate functions of β'' individually, the domain of the scheme generated is not what is required. In particular, rather than the required domain $\tilde{u}_1\cdots\tilde{u}_{n''-1}z''\tilde{u}_{m''+1}\cdots\tilde{u}_k$ we have $\tilde{u}_1\cdots\tilde{u}_{n''-1}z''z''\tilde{u}_{m''+1}\cdots\tilde{u}_k$. This is dealt with by the scheme κ that duplicates the necessary co-ordinates of any input.

Definition 73. For each $u=(u_1\cdots u_k)\in\underline{S}^+$ such that there exists a $u'=(u_n\cdots u_m)\in S^+$ for some $1\leq n\leq m\leq k$, for each $z\in\underline{S}^+$, and for each $v\in S^+$ we define

$$\hat{\circ}^{n,m}_{u,v,z}:\mathrm{PR}(\underline{\Sigma})_{u,v}\times\mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,z,u'}\rightarrow\mathrm{PR}(\underline{\Sigma})_{u_1\cdots u_{n-1}\,z\,u_{m+1}\cdots u_k,v}$$

by induction on the value $l=m-n$ as follows:

Basis $l=0$. First, notice in this case that $n=m$ and hence we have $u'=u_n$. We define $\hat{\circ}^{n,m}_{u,v,z}$ as follows:

$$(\forall\alpha\in\mathrm{PR}(\underline{\Sigma})_{u,v})(\forall\beta\in\mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,z,u_n})\quad\hat{\circ}^{n,m}_{u,v,z}(\alpha,\beta)=\mathcal{C}^{Id^u,u,n}_{u,v,z,u_n}(\alpha,\beta)$$

wherein $Id^u:\{1,\ldots,|u|\}\rightarrow\{1,\ldots,|u|\}$ is the identity function on each $i\in\{1,\ldots,|u|\}$.

Well-Definedness. Notice that by the well-definedness of $\mathcal{C}$ we have

$$\mathcal{C}^{Id^u,u,n}_{u,v,z,u_n}(\alpha,\beta)\in\mathrm{PR}(\underline{\Sigma})_{u\{n/z\},\Delta^{u_n,u,n}(v,z)}$$

Therefore to complete our argument it suffices to show that $\Delta^{u_n,u,n}(v,z)=v$. This follows immediately from the fact that $v\in S^+$.

Induction Hypothesis. Assume for each $\tilde{u}=(\tilde{u}_1\cdots\tilde{u}_k)\in\underline{S}^+$ such that there exists a

$u'' = (\tilde{u}'_{n'} \cdots \tilde{u}'_{m'}) \in S^+$ for some $1 \leq n' \leq m' \leq k$ such that $m' - n' \leq l'$ for some fixed $l' \in \mathbb{N}$, for each $z' \in \underline{S}^+$, and for each $v' \in S^+$ that if $\alpha' \in \mathrm{PR}(\underline{\Sigma})_{\tilde{u},v'}$ and $\beta' \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, z',u''}$ then

$$\hat{\circ}^{n',m'}_{\tilde{u},v',z'}(\alpha',\beta') \in \mathrm{PR}(\underline{\Sigma})_{\tilde{u}_1 \cdots \tilde{u}_{n'-1}\, z'\, \tilde{u}_{m'+1} \cdots \tilde{u}_k, v'}.$$

Induction Step. For each $\underline{\tilde{u}} = (\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_k) \in \underline{S}^+$ such that there exists a $u''' = (\tilde{\tilde{u}}_{n''} \cdots \tilde{\tilde{u}}_{m''}) \in S^+$ for some $1 \leq n'' \leq m'' \leq k$ such that $m'' - n'' = l' + 1$, for each $z'' \in \underline{S}^+$, and for each $v'' \in S^+$ we define $\hat{\circ}^{n'',m''}_{\underline{\tilde{u}},v'',z''}$ as follows:

$$(\forall \alpha'' \in \mathrm{PR}(\underline{\Sigma})_{\underline{\tilde{u}},v''})\,(\forall \beta'' \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, z'',u''})$$

$$\hat{\circ}^{n'',m''}_{\underline{\tilde{u}},v'',z''}(\alpha'',\beta'') = \hat{\circ}^{p,p}_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''}\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, v'', z''}(\hat{\circ}^{n'',m''-1}_{\underline{\tilde{u}},v'',z''}(\alpha'',\gamma), \beta''_{l'+1}) \circ \kappa$$

wherein $p = m'' + (|z''| - 1)$, $\gamma = <\mathrm{U}^{\underline{\tilde{u}}}_1 \circ \beta'', \ldots, \mathrm{U}^{\underline{\tilde{u}}}_{l'} \circ \beta'' >$, $\beta''_{l'+1} = \mathrm{U}^{\underline{\tilde{u}}}_{m''} \circ \beta''$, and

$$\begin{aligned}\kappa = <&\mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_1, \ldots, \mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{n''-1},\\ &\mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{n''}, \ldots, \mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{(n''+|z''|)-1},\\ &\mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{n''}, \ldots, \mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{(n''+|z''|)-1},\\ &\mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{m''+|z''|}, \ldots, \mathrm{U}^{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}_{(k+|z''|)-1} > .\end{aligned}$$

Well-Definedness. First, notice that as $\gamma \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, z'', \tilde{\tilde{u}}_{n''} \cdots \tilde{\tilde{u}}_{m''-1}}$ and $m'' - 1 - n'' = l'$, by the Induction Hypothesis with $\alpha' = \alpha''$, $\beta' = \gamma$, $\tilde{u} = \underline{\tilde{\tilde{u}}}$, $u' = \tilde{\tilde{u}}_{n''} \cdots \tilde{\tilde{u}}_{m''-1}$, $n' = n''$, $m' = m'' - 1$, $z' = z''$, and $v' = v''$ we have $\delta = \hat{\circ}^{n'',m''-1}_{\underline{\tilde{u}},v'',z''}(\alpha'',\gamma) \in \mathrm{PR}(\underline{\Sigma})_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''} \cdots \tilde{\tilde{u}}_k, v''}$. Also, notice that as $\beta'' \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, z'', \tilde{\tilde{u}}_{m''}}$ and $p - p = 0 \leq l'$ by the Induction Hypothesis with $\alpha' = \delta$, $\beta' = \beta''_{l'+1}$, $\tilde{u} = \tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''} \cdots \tilde{\tilde{u}}_k$, $u' = \tilde{\tilde{u}}_{m''}$, $n' = p$, $m' = p$, $z' = z''$, and $v' = v''$ we have

$$\hat{\circ}^{p,p}_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''}\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, v'', z''}(\hat{\circ}^{n'',m''-1}_{\underline{\tilde{u}},v'',z''}(\alpha'',\gamma), \beta''_{l'+1}) \in \mathrm{PR}(\underline{\Sigma})_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, v''}.$$

Therefore as $\kappa \in \mathrm{PR}(\underline{\Sigma})_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, \tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k}$ it is clear that

$$\hat{\circ}^{p,p}_{\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''}\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, v'', z''}(\hat{\circ}^{n'',m''-1}_{\underline{\tilde{u}},v'',z''}(\alpha'',\gamma), \beta''_{l'+1}) \circ \kappa \in \mathrm{PR}(\underline{\Sigma})$$

with type $(\tilde{\tilde{u}}_1 \cdots \tilde{\tilde{u}}_{n''-1}\, z''\, \tilde{\tilde{u}}_{m''+1} \cdots \tilde{\tilde{u}}_k, v'')$ as required.

Combining Ordinary and Cartesian Composition. As with the compiler $\hat{\circ}$ the intention behind $\mathcal{C}$ is essentially straightforward in that it is based on an iterative process. The difference now is that, rather than acting on the successive co-ordinates of a vector-valued scheme, this time the composition process is iterated on successive schemes that may themselves be vector-valued. In particular, the compilation process is iterated on each member of the set $\mathbb{S} = \{\beta_1, \ldots, \beta_k\}$ for some $\beta_i \in \mathrm{PR}(\underline{\Sigma})_{u^i,v^i}$ for some $u^i, v^i \in S^+$ for $i = 1, \ldots, k \geq 1$. Moreover, each successive composition may be either a normal composition or a Cartesian composition and hence the type of composition required must also be indicated. Specifically, the type of each composition is indicated by the set $\mathbb{T} = \{d_1, \ldots, d_k\}$ wherein for each $i \in \{1, \ldots, k\}$ $d_i = o$ indicates a normal

composition and $d_i = c$ indicates a Cartesian composition. In order for this process to be well-defined, if $d_i = c$ then (1) β_i must be an appropriate scheme; that is, a Cartesian form ST; and (2) the particular co-ordinates of the domain of α to which β_i is to be composed must be a stream of an appropriate type.

Again the technical nature of this compiler means that in order to establish that it is indeed well-defined we require a proof by induction on the number k.

Definition 74. For each $\mathbb{S} = \{\beta_1, \ldots, \beta_k\}$ wherein for $i = 1, \ldots, k \geq 1$ either $\beta_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z^i, w^i}$ for some $z^i \in \underline{S}^*$ and for some $w^i \in S^+$, or $\beta_i \in \mathrm{PR}(\underline{\Sigma})_{z^i, w^i}$ for some $z^i \in \underline{S}^*$ and for some $w^i \in S^+$; and for each $\mathbb{T} = \{d_1, \ldots, d_k\}$ such that $d_i \in \{c, o\}$ for $i = 1, \ldots, k$ satisfies $d_i = c \Longrightarrow \beta_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z^i, w^i}$; and for each $u = x\, w^1 \cdots w^k\, y \in S^+$ for some $x, y \in S^*$, and for each $v \in S^+$ we define

$$\mathcal{C}^{\mathbb{S},\mathbb{T}}_{u,v} : \mathrm{PR}(\underline{\Sigma})_{u,v} \rightarrow \mathrm{PR}(\underline{\Sigma})_{x\, z^1 \cdots z^k\, y, v}$$

by induction on the value $k \in \mathbb{N}^+$ as follows:

Basis. $k = 1$. Notice in this case that $\mathbb{S} = \{\beta_1\}$, $\mathbb{T} = \{d_1\}$, and $u = x\, w^1\, y$. We consider two sub-cases:

(1) $d_1 = c$.

(2) $d_1 = o$.

Sub-Case (1) $d_1 = c$. We define $\mathcal{C}^{\mathbb{S},\mathbb{T}}_{u,v}$ as follows: (notice in this case that $\beta_1 \in \mathrm{PR}(\underline{\Sigma})_{t\, z^1, w^1}$)

$$(\forall \alpha \in \mathrm{PR}(\underline{\Sigma})_{u,v}) \quad \mathcal{C}^{\mathbb{S},\mathbb{T}}_{u,v}(\alpha) = \hat{o}^{|x|+1,|x\,w^1|}_{u,v,z^1}(\alpha, \beta_1).$$

Well-Definedness. Notice that in this case by the well-definedness of $\hat{o}^{|x|+1,|x\,w^1|}_{u,v,z^1}$ we have

$$\hat{o}^{|x|+1,|x\,w^1|}_{u,v,z^1}(\alpha, \beta_1) \in \mathrm{PR}(\underline{\Sigma})_{x\, z^1\, y, v}$$

as required.

Sub-Case (2) $d_1 = o$. In this case we define $\mathcal{C}^{\mathbb{S},\mathbb{T}}_{u,v}$ as follows:

$$(\forall \alpha \in \mathrm{PR}(\underline{\Sigma})_{u,v})$$

$$\mathcal{C}^{\mathbb{S},\mathbb{T}}_{u,v}(\alpha) = \begin{cases} \alpha \circ \backslash^{\mathrm{PR}^\vee}(\ll \mathrm{U}^x_1, \ldots, \mathrm{U}^x_{|x|}, \beta_1, \mathrm{U}^y_1, \ldots, \mathrm{U}^y_{|y|} \gg) & \text{if } |x| > 0 \text{ and } |y| > 0, \\ \alpha \circ \backslash^{\mathrm{PR}^\vee}(\ll \beta_1, \mathrm{U}^y_1, \ldots, \mathrm{U}^y_{|y|} \gg) & \text{if } |x| = 0 \text{ and } |y| > 0, \\ \alpha \circ \backslash^{\mathrm{PR}^\vee}(\ll \mathrm{U}^x_1, \ldots, \mathrm{U}^x_{|x|}, \beta_1 \gg) & \text{if } |x| > 0 \text{ and } |y| = 0, \text{ and} \\ \alpha \circ \beta_1 & \text{if } |x| = 0 \text{ and } |y| = 0. \end{cases}$$

Well-Definedness. In order to reduce the complexity of the schemes created we have used the language $\mathrm{PR}^\vee$. In particular, this enables us to have vector-valued compositions wherein the domains of the functions that are combined need not be the same. Notice, that these extended schemes are reduced to standard PR by the use of the compiler $\chi^{\mathrm{PR}^\vee}$. Using this compiler the proof of the well-definedness of each of the schemes in the four cases above is straightforward and is left to the reader.

Induction Hypothesis. Assume for each $\mathbb{S}' = \{\beta'_1, \ldots, \beta'_{k'}\}$ wherein either $\beta'_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z'^i, w'^i}$ for some $z'^i \in \underline{S}^*$ and for some $w'^i \in S^+$ or $\beta'_i \in \mathrm{PR}(\underline{\Sigma})_{z'^i, w'^i}$ for some $z'^i \in \underline{S}^+$ and for some

$w'^i \in S^+$ for $i = 1, \ldots, k'$ for some fixed $k' \in \mathbb{N}^+$; for each $\mathbb{T}' = \{d'_1, \ldots, d'_{k'}\}$ such that $d'_i \in \{b, o\}$, for $i = 1, \ldots, k'$, is defined such that $d'_i = b \Longrightarrow \beta'_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z'^1 w'^1}$; and for each $u' = x'\, w'^1 \cdots w'^{k'}\, y' \in S^+$ for some $x', y' \in S^*$, and for each $v' \in S^+$ that

$$(\forall \alpha' \in \mathrm{PR}(\underline{\Sigma})_{u',v'}) \quad \mathcal{C}^{\mathbb{S}',\mathbb{T}'}_{u',v'}(\alpha') \in \mathrm{PR}(\underline{\Sigma})_{x'\, z'^1 \ldots z'^{k'}\, y', v'}.$$

Induction. For each $\mathbb{S}'' = \{\beta''_1, \ldots, \beta''_{k''+1}\}$ wherein either $\beta''_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z''^i w''^i}$ for some $z''^i \in \underline{S}^+$ and for some $w''^i \in S^+$ for $i = 1, \ldots, k'' + 1$; for each $\mathbb{T}'' = \{d''_1, \ldots, d''_{k''+1}\}$ such that $d''_i \in \{b, o\}$, for $i = 1, \ldots, k'' + 1$, is defined such that $d''_i = b \Longleftrightarrow \beta''_i \in \mathrm{PR}(\underline{\Sigma})_{t\, z''^i w''^i}$; for each $u'' = x''\, w''^1 \cdots w''^{k''+1}\, y'' \in S^+$ for some $x'', y'' \in S^*$; and for each $v'' \in S^+$ we define $\mathcal{C}^{\mathbb{S}'',\mathbb{T}''}_{u'',v''}$ as follows:

$$(\forall \alpha'' \in \mathrm{PR}(\underline{\Sigma})_{u'',v''}) \quad \mathcal{C}^{\mathbb{S}'',\mathbb{T}''}_{u'',v''}(\alpha'') = \mathcal{C}^{\{\beta''_{k''+1}\},\{d''_{k''+1}\}}_{x''\, z''^1 \ldots z''^{k''}\, w''^{k''+1}\, y'', v''}(\gamma)$$

wherein

$$\gamma = \mathcal{C}^{\{\beta''_1, \ldots, \beta''_{k''}\},\{d''_1, \ldots, d''_{k''}\}}_{u'',v''}(\alpha).$$

Well-Definedness. First, notice that by the Induction Hypothesis with $\mathbb{S}' = \{\beta''_1, \ldots, \beta''_{k''}\}$, $\mathbb{T}' = \{d''_1, \ldots, d''_{k''}\}$, $u' = u''$, $x' = x''$, $y' = w''^{k''+1}\, y''$, and $v' = v''$ we have

$$\gamma = \mathcal{C}^{\{\beta''_1, \ldots, \beta''_{k''}\},\{d''_1, \ldots, d''_{k''}\}}_{u'',v''}(\alpha) \in \mathrm{PR}(\underline{\Sigma})_{x''\, z''^1 \ldots z''^{k''}\, w''^{k''+1}\, y'', v''}.$$

Therefore by the Induction Hypothesis with $\mathbb{S}' = \{\beta''_{k''+1}\}$, $\mathbb{T}' = \{d''_{k''+1}\}$, $u' = x''\, z''^1 \cdots z''^{k''}\, w''^{k''+1}\, y''$, $x' = x''\, z''^1 \cdots z''^{k''}$, $y' = y''$, and $v' = v''$ we have

$$\mathcal{C}^{\{\beta''_{k''+1}\},\{d''_{k''+1}\}}_{x''\, z''^1 \ldots z''^{k''}\, w''^{k''+1}\, y'', v''}(\gamma) \in \mathrm{PR}(\underline{\Sigma})_{x''\, z''^1 \ldots z''^{k''+1}\, y'', v''}$$

as required.

Higher-Order Signatures and Terms. In the following sections we will require the use of signatures containing full second-order function symbols rather than the restricted weak second-order signatures that we have used in the previous chapters. However, as the use of such functions is restricted to the purely syntactic level we will not require the full development of this generalized theory of universal algebra. The interested reader in directed to Meinke [1992b] for work on this extended theory.

Definition 75. As our use of higher-order functions is so limited, for our purposes it is sufficient to define an $\underline{S}^+$-sorted second-order signature $\overline{\Sigma}$ to be the $\underline{S}^* \times (\underline{S} - S)^+$-indexed collection of sets wherein any $c \in \overline{\Sigma}_{\lambda,\underline{v}}$ for some $\underline{v} \in \underline{S}^+$ is referred to as a (second-order) constant of type $\underline{v}$; and any $f \in \overline{\Sigma}_{u,\underline{v}}$ for each $u, \underline{v} \in \underline{S}^+$ is referred to as either a (second-order) function or *functional* of type $(u, \underline{v})$.

Notice in particular that a functional may be vector-valued and may have non-streams sorts in their domains. However, their co-domain types are restricted in the sense that they must return stream output and hence we cannot have $f \in \overline{\Sigma}$ if f is of type (z, w) for some $z \in \underline{S}^*$ and for some $w \in S^+$.

Discussion. In making Definition 75 our higher-order signatures are limited in that they may only contain second-order function symbols with restricted types. However, as we are developing an essentially first-order theory of stream processing (that eliminates the use of full second-order functions) this limited definition is sufficient for our purposes. Moreover, we have not directly extended the definition of $\underline{S}$-sorted signatures $\underline{\Sigma}$ to include full second-order function symbols so that in the sequel it is straightforward to be precise about which function symbols are either first-order or weak second-order and are taken from a standard signature. Consequently, this clarifies which function symbols may derive their semantics using the basic techniques of universal algebra we have already developed, and which function symbols are properly second-order and must be eliminated to derive a first-order denotational semantics via PREQ. This fact is reflected in the following notation.

Notation 3. When defining a term τ that may contain a second-order function symbol we will write

$$\tau \in \mathrm{T}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X}),$$

to indicate that τ is term of sort $s \in S$ formed over symbols taken from the standard $\underline{S}$-sorted signature $\underline{\Sigma}$, the $\underline{S}^* \times (\underline{S} - S)^+$-indexed second-order signature $\overline{\Sigma'}$ and the $\underline{S}$-indexed collection of variable symbols $\underline{X}$. In particular, notice as a consequence of restricting higher-order signature definitions, $\underline{\Sigma}$ and $\overline{\Sigma}$ must be disjoint. Indeed, continuing the assumptions of previous chapters in the sequel we will always assume that $\underline{\Sigma}$ is some $\underline{S}$-sorted signature, $\overline{\Sigma'}$ is some $\underline{S}^* \times (\underline{S} - S)^+$-indexed second-order signature and $\underline{X}$ is some $\underline{S}$-indexed collection of variable symbols such that $\underline{\Sigma}$, $\overline{\Sigma'}$ and $\underline{X}$ are all pair-wise disjoint and that none of them contain the distinguished function symbols F and f, and f_i and $f_{i,j}$ for $i, j \in \mathsf{N}^+$.

Also, throughout this chapter if it is not explicitly stated otherwise we make the assumption that any collections of variable symbols always contain the distinguished symbol 't' of type $\mathbf{n}$ and *do not* contain the distinguished symbols 'Y_i' for each $i \in \mathsf{N}$.

Finally, as a concession to conserving space, as with previous chapters we will only include the well-definedness arguments for our constructions when they are not straightforward.

6.2 The Abstract Syntax of ASTRAL

We begin the development of the abstract mathematical syntax of ASTRAL with the definition of the particular classes of terms that may be used in a specification. Example specification using a prototype BNF based on this abstract syntax can be found in Section 6.7.

As we have indicated the abstract syntax of ASTRAL specifications is very similar in form to that of PREQ specifications. In particular, a full ASTRAL specification is constructed from several restricted specification in the same way that a PREQ specification is constructed from RPREQ specifications. However, unlike PREQ full ASTRAL specifications are constructed from two distinct types of restricted specifications: ASTRAL_1 specifications and ASTRAL_2 specifications.

As with RPREQ specifications the formalization of these restricted classes of ASTRAL

specifications is based on the use of particular classes of terms. As such we begin with these definitions.

6.2.1 ASTRAL Terms

As Theorem 6 shows (see Section 4.4), in order to maintain control over the class of functions that we can specify in PREQ, it is necessary to control the combined use of primitive recursion and full λ-abstraction. Therefore, as we will allow the use of AFSTs in ASTRAL, we must be careful to ensure that each ASTRAL specification only represents a primitive recursive ST, so that it can be compiled into PREQ to derive its semantics. In order to maintain this precise control we will define three classes of ASTRAL terms. Specifically, we will begin by defining two restricted classes of terms: *type one ASTRAL terms* and *extended type one ASTRAL terms*, that we will use to define *type one ASTRAL specifications* (denoted ASTRAL_1). Type one ASTRAL specifications are used to define full ASTRAL specifications in essentially the same way as RPREQ specifications are used to construct PREQ specifications. In particular, type one ASTRAL terms are used in 'simple specifications' and the basis case of 'primitive recursive (type one ASTRAL) specifications', and extended type one ASTRAL terms are used in the induction case of 'primitive recursive (type one ASTRAL) specifications'.

Type one ASTRAL terms are also used in a third more general class of terms: *type two ASTRAL terms*. This third class of terms are used as the basic mechanism to provide a user-friendly syntax for the composition of AFSTs without expanding the class of STs that can be specified. Specifically, we use type one ASTRAL specifications – defined using type one and extended type one ASTRAL terms – together with type two ASTRAL specifications – defined using type two ASTRAL terms – to define *type three ASTRAL specification*. In particular, type three ASTRAL specifications are sufficient from a computability theoretic perspective to define every AFST that can be compiled into an equivalent primitive recursive CFST in PREQ. However, as a further consideration to the user, while full ASTRAL specification are based on type three ASTRAL specifications, we provide an extended syntax that is more suited to modular specification techniques. These ideas are now presented in more detail.

Type One ASTRAL Terms. *Type one ASTRAL terms* are essentially first-order terms except that we include an additional induction case (Case (5)) that also allows second-order function symbols to be used in terms provided that that they appear in evaluated form. For example, if F is a second-order function symbol of type $(u, \underline{v})$, for some $u \in S^+$ and for some $\underline{v} \in \underline{S}^+$ then for appropriate terms τ_i of type u_i for $i = 1, \ldots, |u|$

$$F(\tau_1, \ldots, \tau_{|u|})(0)$$

is permissible as a type one ASTRAL term.

Definition 76. We define the S-indexed collection of *type one ASTRAL terms*

$$A^{T_1}(\underline{\Sigma}, \overline{\underline{\Sigma}'}, \underline{X}) = < A^{T_1}(\underline{\Sigma}, \overline{\underline{\Sigma}'}, \underline{X})_s \mid s \in S >$$

wherein each $A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$ is defined uniformly in $s \in S$ by induction on the structural complexity of a term $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$ as follows:

Basis Cases.

(1) Constants. If

$$\tau = c$$

for some $c \in \Sigma_{\lambda,s}$, for some $s \in S$ then $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$.

(2) Variables. If

$$\tau = x$$

for some $x \in X_s$, for some $s \in S$ then $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$.

Induction.

(3) Algebraic Operations. If

$$\tau = \sigma(\tau_1, \ldots, \tau_n)$$

for some $\sigma \in \Sigma_{w,s}$, for some $w = (s_1 \cdots s_n) \in S^+$ and for some $s \in S$, and for some $\tau_i \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_i}$ for $i = 1, \ldots, n$ then $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$.

(4) Stream Variables. If

$$\tau = x(\tau')$$

for some $x \in \underline{X}_{\underline{s}}$, for some $s \in S$ and for some $\tau' \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_t$ then $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$.

(5) Type One Restricted Functional Use. If

$$\tau = H_i(\tau_1, \ldots, \tau_n)(\tau')$$

for some $H \in \overline{\Sigma'}_{u,\underline{v}}$, for some $u = (s_1 \cdots s_n) \in \underline{S}^+$ and for some $v \in S^+$; for some $i \in \{1, \ldots, |\underline{v}|\}$; for some τ_j such that for $j = 1, \ldots, n$ either $\tau_j = x_j \in \underline{X}_{s_j}$ if $s_j \in (\underline{S} - S)$ or $\tau_j \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_j}$ otherwise; and for some $\tau' \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_t$ then $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{v_i}$. Notice, that in our definition H_i is properly $(H)_i$; that is, i is considered as a separate syntactic token indicating which co-ordinate of H we wish to select and does *not* indicate that $H_i \in \overline{\Sigma'}$.

Extended Type One ASTRAL Terms. We now defining two further classes of terms. First, we define the class of *extended type one ASTRAL terms*. Essentially, the additional clause that is included in these terms is equivalent to the use of the special variable symbol Y_i in RPREQ specifications (see Definition 84); that is, it is the mechanism by which we specify primitive recursive AFSTs definitions.

Definition 77. For each $\mathbb{X} = \{y_1, \ldots, y_n\} \subseteq \underline{X}$ and for each $\underline{w} \in \underline{S}^+$ we define the S-indexed collection of *extended type one ASTRAL terms*

$$A^{T_1,\mathbb{X},\underline{w}}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X}) = < A^{T_1,\mathbb{X},\underline{w}}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s \mid s \in S >$$

wherein each $A^{\Gamma_1,\mathbb{X},\underline{w}}(\underline{\Sigma},\overline{\Sigma'},\underline{X})_s$ is defined uniformly in $s \in S$ by induction on the structural complexity of a term $\tau \in A^{\Gamma_1,\mathbb{X},\underline{w}}(\underline{\Sigma},\overline{\Sigma'},\underline{X})_s$.

Essentially, for each $\mathbb{X} = \{y_1,\ldots,y_n\} \subseteq \underline{X}$, for each $\underline{w} \in \underline{S}^+$ and for each $s \in S$ we define $A^{\Gamma_1,\mathbb{X},\underline{w}}(\underline{\Sigma},\overline{\Sigma'},\underline{X})_s$ to be identical to $A^{\Gamma_1}(\underline{\Sigma},\overline{\Sigma'},\underline{X})_s$ with the exception of an additional induction clause defined as follows:

(6) Type Two Restricted Functional Use. If

$$\tau = F_i(y_1,\ldots,y_n)(t)$$

for some $i \in \{1,\ldots,|\underline{w}|\}$ then $\tau \in A^{\Gamma_1,\mathbb{X},\underline{w}}(\underline{\Sigma},\overline{\Sigma'},\underline{X})_{w_i}$.

Thus, extended type one ASTRAL terms allow us to use second-order stream transformers in terms, but they may only take variables as input and are evaluated at a fixed time t.

Type Two ASTRAL Terms. Finally, our third class of terms – *type two ASTRAL terms* – provide the mechanism for the most general use of AFSTs that can be permitted without expanding the class of functions that may be specified. In particular, the induction case of type two ASTRAL terms allows us to apply a second-order function symbol to both type one and extended type one ASTRAL terms and hence provides a mechanism for the composition of AFSTs. For example, if G is of type $(\underline{u},\underline{v})$, for some $u,v \in S^+$; $x_i \in \underline{X}_{\underline{u}_i}$ for $i = 1,\ldots,|\underline{u}|$; F is of type $(s_1\ s_2\ \underline{v},\underline{v}')$, for some $s_1,s_2 \in S$ and for some $\underline{v}' \in \underline{S}^+$; and τ_1 and τ_2 are type one ASTRAL terms of sort s_1 and s_2 respectively then

$$F(\tau_1,\tau_2,G(x_1,\ldots,x_{|\underline{u}|}))$$

is a well-defined type two ASTRAL term.

Notice that this syntax provides a user-friendly method for working with AFSTs, in that AFSTs can be composed directly in vector-valued form. Indeed, the desire to work with terms of this form is the reason that we need the generality provided by the compiler $\mathcal{C}$. In particular, in order to construct a PREQ specification that provides the necessary semantics for the term above we must compose τ_1 and τ_2 with F, but we must use vector-valued Cartesian composition to compose G with F as we must give F and G their semantics in Cartesian form.

The use of terms in the form above is especially useful in the context of formal hardware description, as in addition to allowing the specification of devices from smaller components, it also allows devices to be dependent on initial non-stream values (see Section 3.10 and for an example see Section 3.8.1 and Section 6.7.2).

Definition 78. Let $\mathbb{X} = \{x_1,\ldots,x_m\} \subseteq \underline{X}$ such that $x_i \in \underline{X}_{s_i}$ for some $s_i \in \underline{S}$ for $i = 1,\ldots,m \geq 1$. We define the $\underline{S}^* \times (\underline{S}-S)^+$-indexed collection of *type two ASTRAL terms*

$$A^{\Gamma_2}(\underline{\Sigma},\overline{\Sigma'},\mathbb{X}) = < A^{\Gamma_2}(\underline{\Sigma},\overline{\Sigma'},\mathbb{X})_{u,\underline{v}} \mid u \in \underline{S}^*, \underline{v} \in \underline{S}^+ >$$

wherein each $A^{\Gamma_2}(\underline{\Sigma},\overline{\Sigma'},\mathbb{X})_{u,\underline{v}}$ is defined uniformly in $(u,\underline{v})$ by induction on the structural complexity of a term $\tau \in A^{\Gamma_2}(\underline{\Sigma},\overline{\Sigma'},\mathbb{X})_{u,\underline{v}}$ as follows:

Basis.

(1) Stream Variables. If

$$\tau = x$$

for some $x \in \mathbb{X}$ of type $\underline{s} \in \underline{S}$ then $\tau \in A^{T_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{\underline{s},\underline{s}}$.

Induction.

(2) Composition. If

$$\tau = H(\tau_1, \ldots, \tau_n)$$

for some $H \in \overline{\Sigma'}_{w,\underline{v}}$, for some $w = y^1 \cdots y^n \in \underline{S}^+$, for some $n \leq |w|$, for some $\underline{v} \in \underline{S}^+$ and for some τ_i for $i = 1, \ldots, n$ such that either

$$\tau_i \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{s'_i}$$

for some $s'_i = y^i \in S$ or

$$\tau_i \in A^{T_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^i,\underline{z}^i}$$

for some $u^i \in \underline{S}^*$ and for some $\underline{z}^i = y^i \in \underline{S}^+$ then $\tau \in A^{T_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{d^1 \cdots d^n,\underline{v}}$ wherein for $i = 1, \ldots, n$ we have

$$d^i = \begin{cases} s_1 \cdots s_m & \text{if } \tau_i \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{s'_i}, \text{ and} \\ u^i & \text{if } \tau_i \in A^{T_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^i,\underline{z}^i} \text{ otherwise.} \end{cases}$$

6.2.2 ASTRAL Specifications

Using the classes of terms defined in the previous section we now define three forms of restricted ASTRAL specifications that we use to define the full abstract ASTRAL syntax. We begin with *type one ASTRAL specifications* that essentially provide a more natural syntax for the representation of RPREQ specifications in the context of stream based specification.

Type One ASTRAL Specifications

Definition 79. We define the $\underline{S}^* \times (\underline{S} - S)^+$-indexed family of *type one ASTRAL specifications*

$$\text{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X}) = < \text{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}} \mid u, \underline{v} \in \underline{S}^+ >$$

wherein for each $u \in \underline{S}^+$, and for each $\underline{v} \in \underline{S}^+$ we define

$$\text{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}}$$

(ambiguously denoted $\text{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})$) by case analysis as follows:

(1) Simple Specifications. If

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_n)(t) = \tau$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n \in \mathbb{N}$ and for some $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})$, for some $s \in S$ then $\Phi \in \text{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_1 \cdots s_n,\underline{s}}$.

(2) Vector-Valued Simple Specifications. If

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_n)(t) = (\tau_1, \ldots, \tau_m)$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n \in \mathbb{N}$ and for some $\tau_j \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$, for some $s'_j \in S$ for $j = 1, \ldots, m \geq 1$ then $\Phi \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_1 \cdots s_n, s'_1 \cdots s'_m}$.

(3) Primitive Recursive Specifications. If

$$\Phi \stackrel{def}{=} \begin{array}{lcl} F(x_1, \ldots, x_n)(0) & = & \tau \\ F(x_1, \ldots, x_n)(t+1) & = & \tau' \end{array}$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n \in \mathbb{N}^+$; for some $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_s$, for some $s \in S$; and for some $\tau' \in A^{T_1\{x_1, \ldots, x_n\}, s}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_s$ then $\Phi \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_1 \cdots s_n, s}$.

(4) Vector-Valued Primitive Recursive Specifications. If

$$\Phi \stackrel{def}{=} \begin{array}{lcl} F(x_1, \ldots, x_n)(0) & = & (\tau_1, \ldots, \tau_m) \\ F(x_1, \ldots, x_n)(t+1) & = & (\tau'_1, \ldots, \tau'_m) \end{array}$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n \in \mathbb{N}^+$; for some $\tau_j \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$, for some $s'_j \in S$; and for some $\tau'_j \in A^{T_1\{x_1, \ldots, x_n\}, s'_1 \cdots s'_m}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$ for $j = 1, \ldots, m \geq 1$ then $\Phi \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_1 \cdots s_n, s'_1 \cdots s'_m}$.

Type Two ASTRAL Specifications. To avoid the complications that the unrestricted combination of λ-abstraction and primitive recursion can create, we cannot directly incorporate the composition of AFSTs into type one ASTRAL terms. As such we define two further classes of restricted ASTRAL specifications that we will use to construct full ASTRAL specifications. The first of these is *type two ASTRAL specifications* that allow the use of type two ASTRAL terms and hence the composition of STs.

Definition 80. We define the $\underline{S}^+ \times (\underline{S} - S)^+$-indexed family of *type two ASTRAL specifications*

$$\mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X}) = < \mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}} | u, \underline{v} \in \underline{S}^+ >$$

wherein for each $u \in \underline{S}^*$ and for each $\underline{v} \in \underline{S}^+$ $\mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}}$ (ambiguously denoted $\mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})$) is defined as follows:
if

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_n) = \tau$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n \in \mathbb{N}^+$; and for some $\tau \in A^{T_2}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})$ of type $(w, \underline{v})$ for some $w \in \underline{S}^*$ and for some $\underline{v} \in \underline{S}^+$ then $\Phi \in \mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_1 \cdots s_n, \underline{v}}$.

Type Three ASTRAL Specifications. In the same way that we used total PREQ specifications as an intermediate stage between partial PREQ specifications and standard PREQ specifications, the definition of type three ASTRAL specifications and full ASTRAL specifications is again simply a convenience in that it allows a more modular approach to be used in

system specification. In particular, type three ASTRAL specifications are constructed from a single type two ASTRAL specification and a number of type one ASTRAL specifications. The role that the type two specification plays is to choose a particular type one specifications as the semantics of the whole specification. This technique mirrors the way that we choose a particular RPREQ specification to represent the semantics of a whole PREQ specification. However, as the abstract syntax of ASTRAL only provides the basis of an implementation language, for mathematical convenience this role of choosing a particular type one specification is also combined with the mechanism for composing AFSTs. Specifically, the type two ASTRAL specification within a type three ASTRAL specification either selects a single type one ASTRAL specification to be the semantics of the whole specification or selects a number of type one ASTRAL specifications composed together to be the semantics of the whole specification.

While from the perspective of computability a type three ASTRAL specification is a general purpose tool for the specification of primitive recursive STs, from the perspective of user-friendly specification it is rather limited. Therefore, in full ASTRAL specifications we allow several type two specifications to be used wherein only one determines the whole specification's semantics. For example, suppose we wished to specify a system F that was naturally visualized as two subsystems F_1 and F_2 wherein it is also natural to sub-divide F_1 into two further sub-sub-systems $F_{1,1}$ and $F_{1,2}$. If we restricted the user to type three ASTRAL specifications then they would have to specify F as follows:

$$\Phi =< \psi, \phi_{1,1}, \phi_{1,2}, \phi_2 >$$

wherein the type one ASTRAL specifications $\phi_{1,1}$, $\phi_{1,2}$ and ϕ_2 represent sub-systems $F_{1,1}$, $F_{1,2}$ and F_2 respectively and the type two ASTRAL specification ψ would be of the form

$$\psi = F(x_1, \ldots, x_n) = F_2(F_{1,2}(F_{1,1}(x_1, \ldots, x_n)))$$

for some approriate variables x_i for $i = 1, \ldots, n \geq 1$. However, using a full ASTRAL specification we can specify F as follows:

$$\Phi =< \psi_1, \psi_2, \phi_{1,1}, \phi_{1,2}, \phi_2 >$$

wherein the type two ASTRAL specifications ψ_1 and ψ_2 would be of the form

$$\psi_1 = F(x_1, \ldots, x_n) = F_2(F_1(x_1, \ldots, x_n))$$

and

$$\psi_2 = F_1(x_1, \ldots, x_n) = F_{1,2}(F_{1,1}(x_1, \ldots, x_n))$$

respectively. This more naturally reflects a modular approach to the specification based on the systems intuitive structure.

As we will show in the sequel the additional type two specifications in a full ASTRAL specification can be eliminated (using an inductive technique) by compiling full ASTRAL specifications into a type three ASTRAL specification. To complete the compilation procedure the resulting type three ASTRAL specification is compiled into PREQ to derive a semantics.

Finally, notice in the following definition we use a version of the function InTermsOf (see Definition 49) adapted for use with ASTRAL specifications, although we leave the details of a formal definition of the necessary modifications to the reader.

Definition 81. We define the $\underline{S}^* \times (\underline{S} - S)^+$-indexed family of *type three ASTRAL specifications*

$$\mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X}) = < \mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})_{u,\underline{v}} \mid u \in \underline{S}^*, \underline{v} \in \underline{S}^+ >$$

wherein for each $u, \underline{v} \in \underline{S}^+$ we define $\mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})_{u,\underline{v}}$ (ambiguously denoted $\mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})$) as follows:
if

$$\Phi = < \psi, \phi_1, \ldots, \phi_n >$$

for some

$$\psi \in \mathrm{ASTRAL}_2(\underline{\Sigma}, \{F_1, \ldots, F_n\}, \underline{X})_{u,\underline{v}}$$

for some $u \in \underline{S}^*$, for some $\underline{v} \in \underline{S}^+$ such that F_i is of type $(u^i, \underline{v}^i)$ for some $u^i \in \underline{S}^*$ and for some $\underline{v}^i \in \underline{S}^+$ for $i = 1, \ldots, n \geq 1$; and for some

$$\phi_i \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \{F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n\}, \underline{X})_{u^i,\underline{v}^i}$$

for $i = 1, \ldots, n$ such that

$$i \notin \mathrm{InTermsOf}(< \phi_1, \ldots, \phi_n >, i, \{F_1, \ldots, F_n\})$$

for $i = 1, \ldots, n$ then $\Phi \in \mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})_{u,\underline{v}}$.

ASTRAL Specifications

Definition 82. We define the $\underline{S}^* \times (\underline{S} - S)^+$-indexed family of *ASTRAL specifications*

$$\mathrm{ASTRAL}(\underline{\Sigma}, \underline{X}) = < \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,\underline{v}} \mid u \in \underline{S}^*, \underline{v} \in \underline{S}^+ >$$

wherein for each $u \in \underline{S}^*$ and for each $\underline{v} \in S^+$ we define $\mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,\underline{v}}$ (ambiguously denoted $\mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})$) as follows:
if

$$\Phi = < \psi, \psi_1, \ldots, \psi_m, \phi_1, \ldots, \phi_n >$$

for some

$$\psi \in \mathrm{ASTRAL}_2(\underline{\Sigma}, \{F_1, \ldots, F_{m+n}\}, \underline{X})_{u,\underline{v}}$$

for some $u \in \underline{S}^*$, for some $\underline{v} \in \underline{S}^+$ such that F_i is of type $(u^i, \underline{v}^i)$ for some $u^i \in \underline{S}^*$ and for some $\underline{v}^i \in \underline{S}^+$ for $i = 1, \ldots, m + n \geq 1$; for some

$$\psi_i \in \mathrm{ASTRAL}_2(\underline{\Sigma}, \{F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_m\}, \underline{X})_{u^i,\underline{v}^i}$$

for $i = 1, \ldots, m$; and for some

$$\phi_i \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \{F_{m+1}, \ldots, F_{i-1}, F_{i+1}, \ldots, F_{n+m}\}, \underline{X})_{u^i,\underline{v}^i}$$

for $i = m + 1, \ldots, m + n$ such that

$$i \notin \mathrm{InTermsOf}(< \psi_1, \ldots, \psi_m, \phi_1, \ldots, \phi_n >, i, \{F_1, \ldots, F_{m+n}\})$$

for $i = 1, \ldots, m + n$ then $\Phi \in \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,\underline{v}}$.

The definition of ASTRAL specifications finalizes the first stage of our abstract specification language formulation. In order to complete the formulation, by defining a denotational semantics for ASTRAL specifications, we require the compiler definitions that we make in the following section.

6.3 Compiling ASTRAL into PREQ Specifications

As with the compiler that maps PREQ into PR, the compiler that maps ASTRAL into PREQ is defined in terms of several sub-compilers that map the various classes of terms and restricted classes of ASTRAL specifications we have defined.

6.3.1 Compiling ASTRAL Terms

First, we show how to compile type one ASTRAL terms into strictly first-order terms that will be used in the construction of individual RPREQ specifications. Indeed, given the restricted form of type one ASTRAL terms this process is relatively straightforward. The main operation of this compiler is the replacement of terms involving functionals by equivalent terms in Cartesian form comprised of new weak-second order function symbols.

Definition 83. Let $\overline{\Sigma'} \subseteq \{F_1, \ldots, F_n\}$, for some $n \in \mathbb{N}$ such that F_i is of type (u^i, v^i), for some $u^i \in \underline{S}^+$ and for some $\underline{v}^i \in \underline{S}^+$ for $i = 1, \ldots, n$. Furthermore, let $\tilde{\underline{\Sigma}}$ be defined such that for each $w \in \underline{S}^*$ and for each $s \in S$

$$\tilde{\underline{\Sigma}}_{w,s} = \begin{cases} \underline{\Sigma}_{w,s} \bigcup \{f_{i,j}\} & \text{if } w = \mathbf{t}\ u \text{ and there exists an } F_i \in \overline{\Sigma}'_{u^i,\underline{v}^i} \\ & \text{for some } i \in \{1, \ldots, n\}, \text{ for some } v^i = (s_1 \cdots s_m) \in \underline{S}^+ \\ & \text{such that } \underline{s} = s_j, \text{ for some } j \in \{1, \ldots m\}, \text{ and} \\ \underline{\Sigma}_{w,s} & \text{otherwise.} \end{cases}$$

For each $s \in S$ we define

$$\chi_s^{\mathrm{A}^{\mathrm{T}_1}} : \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s \rightarrow \mathrm{T}(\tilde{\underline{\Sigma}}, \underline{X})_s$$

(ambiguously denoted $\chi^{\mathrm{A}^{\mathrm{T}_1}}$) uniformly in $s \in S$ by induction on the structural complexity of a term $\tau \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$ as follows:

Basis Cases.

(1) Constants. If $\tau = c$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau) = c.$$

(2) Variables. If $\tau = x$ for some $x \in X_s$ for some $s \in S$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau) = x.$$

Induction.

(3) Algebraic Operations. If $\tau = \sigma(\tau_1, \ldots, \tau_m)$ for some $\sigma \in \Sigma_{w,s}$, for some $w = (s_1 \cdots s_m) \in S^+$, for some $s \in S$ and for some $\tau_i \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_i}$ for $i = 1, \ldots, m$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau) = \sigma(\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau_1), \ldots, \chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau_m)).$$

(4) Stream Variables. If $\tau = x(\tau')$ for some $x \in \underline{X}_{\underline{s}}$, for some $s \in S$ and for some $\tau' \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{\mathrm{t}}$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau) = eval_s(\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau'), x).$$

Well-Definedness. First, notice that by definition that $\tilde{\underline{\Sigma}}$ is standard and therefore $eval_s \in \tilde{\underline{\Sigma}}_{\mathrm{t}\,\underline{s},s}$. Also notice that by the unstated Induction Hypothesis that $\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau') \in \mathrm{T}(\tilde{\underline{\Sigma}}, \underline{X})_{\mathrm{t}}$. Therefore as by hypothesis $x \in \underline{X}_{\underline{s}}$ it is clear that $eval_s(\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau'), x) \in \mathrm{T}(\tilde{\underline{\Sigma}}, \underline{X})_s$ as required.

(5) Type One Restricted Functional Use. If $\tau = (F_i)_j(\tau_1, \ldots, \tau_m)(\tau')$, for some $F_i \in \overline{\Sigma'}_{u,\underline{v}}$ for some $u = (s_1 \cdots s_l) \in \underline{S}^+$ and for some $v \in S^+$; for some $i \in \{1, \ldots, n\}$; for some $j \in \{1, \ldots, |\underline{v}|\}$, for some τ_k such that for $k = 1, \ldots, m$ either $\tau_k = x_k \in \underline{X}_{s_k}$ wherein $s_k \in (\underline{S} - S)$ or $\tau_k \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_k}$ otherwise; and for some $\tau' \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{\mathrm{t}}$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau) = f_{i,j}(\chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau'), y_1, \ldots, y_m)$$

wherein for $k = 1, \ldots, m$ we have

$$y_k = \begin{cases} x_k & \text{if } \tau_k = x_k; \text{ and} \\ \chi^{\mathrm{A}^{\mathrm{T}_1}}(\tau_k) & \text{otherwise.} \end{cases}$$

Using the compiler $\chi^{\mathrm{A}^{\mathrm{T}_1}}$ it is now also straightforward to define a compiler that maps extended type one ASTRAL terms. The only additional mechanism that we need is to convert terms defined using type two restricted functional use. This is achieved by replacing these terms with the distinguished variable symbols Y_i that has the required semantics when the resulting term is used in the induction case of a primitive recursive RPREQ specification.

Definition 84. Let $\overline{\Sigma'}$ and $\tilde{\underline{\Sigma}}$ be defined as in the previous definition. For each $\mathbb{X} = \{y_1, \ldots, y_n\} \subseteq \underline{X}$, for each $\underline{w} \in \underline{S}^+$ and for each $s \in S$ we define

$$\chi_s^{\mathrm{A}^{\mathrm{T}_1}\mathbb{X},\underline{w}} : \mathrm{A}^{\mathrm{T}_1,\mathbb{X},\underline{w}}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s \to \mathrm{T}(\tilde{\underline{\Sigma}}, \underline{X'})_s$$

(ambiguously denoted $\chi^{\mathrm{A}^{\mathrm{T}_1}\mathbb{X},\underline{w}}$), wherein for each $s' \in \underline{S}$

$$\underline{X'} = \begin{cases} \underline{X}_{s'} \cup \{Y_i\} & \text{if } s' = \underline{w_i} \text{ for some } i \in \{1, \ldots, |\underline{w}|\}, \text{ and} \\ \underline{X}_{s'} & \text{otherwise,} \end{cases}$$

uniformly in $s \in S$ by induction on the structural complexity of a term $\tau \in \mathrm{A}^{\mathrm{T}_1,\mathbb{X},\underline{w}}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_s$. Essentially $\chi_s^{\mathrm{A}^{\mathrm{T}_1}\mathbb{X},\underline{w}}$ is defined identically to $\chi_s^{\mathrm{A}^{\mathrm{T}_1}}$ with the additional induction case as follows:

(6) Type Two Restricted Functional Use. If $\tau = F_i(y_1, \ldots, y_n)(t)$ for some $i \in \{1, \ldots, |\underline{w}|\}$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_1}\mathbb{X},\underline{w}}(\tau) = Y_i.$$

As Cartesian composition is formalized at the level of PR schemes it is also necessary to compile type one ASTRAL terms and type two ASTRAL terms directly into PR. As such the schemes created by this compilation process can be combined using the compiler $\mathcal{C}$ into a single scheme and converted back into PREQ using the compiler $\mathbb{C}^{\text{PREQ}}$. In more detail, recall from the previous section that from the perspective of computability type three ASTRAL specifications are sufficient to specify any AFST with an equivalent primitive recursive CFST. Also recall that a type three ASTRAL specification Ψ is comprised of a single type two ASTRAL specification and $k \geq 1$ type one ASTRAL specifications; that is,

$$\Psi = < \psi, \phi_1, \ldots, \phi_k > .$$

Therefore to compile a type three ASTRAL specification into PREQ our strategy is as follows: (1) independently of ψ (that either selects a particular ϕ_j as the semantics of Ψ or a number $\phi_{j_1}, \ldots, \phi_{j_m}$ for some $j_l \in \{1, \ldots, k\}$ for $l = 1, \ldots, m \in \mathbb{N}^+$ composed together as the semantics of Ψ) compile Ψ into k Cartesian form PREQ specifications Φ_i wherein each ϕ_i for $i = 1, \ldots, k$ respectively is selected in turn as the semantics of Ψ; (2) using the compiler $\mathbb{C}^{\text{PR}}$ compile each Φ_i into an equivalent PR scheme α_i; (3) using the schemes $\alpha_1, \ldots, \alpha_k$ compile Ψ into a PREQ scheme using the compiler $\mathcal{C}$. As such if ψ simply selects a particular type two specification ϕ_j then we use Φ_j as the semantics of Ψ; otherwise if ψ dictates that $\phi_{j_1}, \ldots, \phi_{j_m}$ composed together is the semantics of Ψ then (essentially) we make

$$\Phi = \mathbb{C}^{\text{PREQ}}(\mathcal{C}^{\{\alpha_{j_2}, \ldots, \alpha_{j_m}\}}(\alpha_{j_1}))$$

the semantics of Ψ wherein by definition the schemes $\alpha_{j_1}, \ldots, \alpha_{j_m}$ are equivalent to the type one ASTRAL specifications $\phi_{j_1}, \ldots, \phi_{j_k}$ respectively and hence using $\mathcal{C}$ and $\mathbb{C}^{\text{PREQ}}$ is appropriate to construct the single PREQ scheme Φ to represent Ψ.

With this strategy in mind we now define the two compilers that we need to compile a type two ASTRAL specification into PR.

Definition 85. Let $\overline{\Sigma'}$ be defined as in Definition 83 and let $\mathbb{X} = \{x_1, \ldots, x_m\} \subseteq \underline{X}$ for some $x_i \in \underline{X}_{s_i}$ for some $s_i \in \underline{S}$ for $i = 1, \ldots, m \geq 1$. For each $\mathbb{S} = \{\alpha_1, \ldots, \alpha_n\}$ wherein $\alpha_i \in \text{PR}(\underline{\Sigma})_{t_{u^i,v}}$ for $i = 1, \ldots, n$ and for each $s \in S$ we define

$$\mathbb{C}_s^{\text{A}^{\text{T}_1}\mathbb{S}} : \text{A}^{\text{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_s \rightarrow \text{PR}(\underline{\Sigma})_{s_1 \cdots s_m, s}$$

(ambiguously denoted $\mathbb{C}^{\text{A}^{\text{T}_1}\mathbb{S}}$) uniformly in $s \in S$ by induction on the structural complexity of a term $\tau \in \text{A}^{\text{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_s$ as follows:

Basis Cases.

(1) Constants. If $\tau = c$, for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$ then

$$\mathbb{C}^{\text{A}^{\text{T}_1}\mathbb{S}}(\tau) = c^{s_1 \cdots s_m}.$$

(2) Variables. If $\tau = x_i$, for some $x_i \in \mathbb{X}$ of type $s_i \in S$ then

$$\mathbb{C}^{\text{A}^{\text{T}_1}\mathbb{S}}(\tau) = \text{U}_i^{s_1 \cdots s_m}.$$

Induction.

(3) Algebraic Operations. If $\tau = \sigma(\tau_1, \ldots, \tau_m)$, for some $\sigma \in \Sigma_{w,s}$, for some $w = (s'_1 \cdots s'_m) \in S^+$ and for some $s \in S$; and for some $\tau_i \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s'_i}$ for $i = 1, \ldots, m$ then

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau) = \sigma \circ < \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_m) > .$$

(4) Stream Variables. If $\tau = x_i(\tau')$, for some $x_i \in \mathbb{X}$ of type $s_i \in (\underline{S} - S)$ and for some $\tau' \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_t$ then

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau) = eval_s \circ < \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau'), \mathrm{U}_i^{s_1 \cdots s_m} > .$$

(5) Type One Restricted Functional Use. If $\tau = (F_i)_j(\tau_1, \ldots, \tau_m)(\tau')$ for some $F_i \in \overline{\Sigma'}_{u^i, \underline{v}^i}$, for some $u^i = (s_{i,1} \cdots s_{i,m}) \in \underline{S}^*$ and for some $\underline{v}^i \in \underline{S}^+$; for some $i \in \{1, \ldots, n\}$; for some $j \in \{1, \ldots, |\underline{v}^i|\}$; for some τ_l such that for $l = 1, \ldots, m$ either $\tau_l = x_l \in \underline{X}_{s_{i,l}}$, wherein $s_{i,l} \in (\underline{S} - S)$ or $\tau_l \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{s_{i,l}}$ otherwise; and for some $\tau' \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_t$ then

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau) = \mathrm{U}_j^{\underline{v}^i} \circ \alpha_i \circ < \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau'), \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_m) > .$$

Definition 86. Let $\overline{\Sigma'}$ be defined as in Definition 83 and let $\mathbb{X} = \{x_1, \ldots, x_l\} \subseteq \underline{X}$ such that $x_i \in \underline{X}_{r_i}$ for some $r_i \in \underline{S}$ for $i = 1, \ldots, l \geq 1$. For each $\mathbb{S} = \{\alpha_1, \ldots, \alpha_n\}$ wherein $\alpha_i \in \mathrm{PR}(\underline{\Sigma})_{\mathrm{t}\, u^i, v^i}$ for $i = 1, \ldots, n$ and for each $u \in \underline{S}^*$ and for each $\underline{v} \in \underline{S}^+$ we define

$$\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}_{u,\underline{v}} : \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u,\underline{v}} \to \mathrm{PR}(\underline{\Sigma})_{\mathrm{t}\, u,v}$$

(ambiguously denoted $\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}$) uniformly in $(u, \underline{v})$ by induction on the structural complexity of a term $\tau \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u,\underline{v}}$ as follows:

Basis Cases.

(1) Stream Variables. If $\tau = x$, for some $x \in \mathbb{X}$ of type $\underline{s} \in S$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}_{\underline{s},\underline{s}}(\tau) = eval_{\underline{s}}.$$

Induction.

(2) Composition. If $\tau = F_i(\tau_1, \ldots, \tau_m)$ for some $F_i \in \overline{\Sigma'}_{u^i, \underline{v}^i}$, for some $u^i = y^1 \cdots y^n \in \underline{S}^+$, for some $m \leq |u^i|$ and for some $\underline{v}^i \in \underline{S}^+$; and for some τ_j for $j = 1, \ldots, m$ such that either

$$\tau_j \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{s_j}$$

for some $s_j = y^j \in S$ or

$$\tau_j \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{w^j, \underline{z}^j}$$

for some $w^j \in \underline{S}^*$ and for some $\underline{z}^j = y^j \in \underline{S}^+$ then

$$\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau) = \mathcal{C}^{\mathbb{R},\mathbb{T}}_{\mathrm{t}\, u^i, \underline{v}^i}(\alpha_i)$$

wherein

$$\mathbb{R} = \{\gamma_1, \ldots, \gamma_m\}$$

and

$$\mathbb{T} = \{d_1, \ldots, d_m\}$$

wherein for $j = 1, \ldots, m$ we have

$$\gamma_j = \begin{cases} \chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau_j) & \text{if } \tau_j \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^j, \underline{z}^j}, \text{ and} \\ \mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_j) & \text{otherwise} \end{cases}$$

and

$$d_j = \begin{cases} c & \text{if } \tau_j \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^j, \underline{z}^j}, \text{ and} \\ o & \text{otherwise.} \end{cases}$$

Well-Definedness. First, notice that we have $\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau_j) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, u^j, v^j}$ such that $\tau_j \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^j, \underline{z}^j}$ for each $j \in \{1, \ldots, m\}$. Secondly, notice that by the well-definedness of $\mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}$ we have $\mathbb{C}^{\mathrm{A}^{\mathrm{T}_1}\mathbb{S}}(\tau_j) \in \mathrm{PR}(\underline{\Sigma})_{r_1, \ldots, r_l, s_j}$. Therefore as by definition $d_j = c \Longrightarrow \beta_j = \chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau_j)$ is of type $(\mathbf{t}\, u^j, v^j)$ for $j = 1, \ldots, m$ by the well-definedness of $\mathcal{C}$ we have

$$\chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau) = \mathcal{C}^{\mathbb{R}, \mathbb{T}}_{\mathbf{t}\, u^1, \underline{v}^1}(\alpha_i) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, e^1, \ldots, e^m, v^i}$$

wherein for $j = 1, \ldots, m$ we have

$$e_j = \begin{cases} r_1 \cdots r_l & \text{if } \tau_j \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{s_j}, \text{ and} \\ u^j & \text{if } \tau_j \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^j, \underline{z}^j} \text{ otherwise} \end{cases}$$

as required.

While the two previous compilers are sufficient to map the terms that occur on the right-hand-side of a equation in a type two ASTRAL specification into PR. To complete this compilation process we must also take into account the role played by the term on the left-hand-side of the equation and in particular the variables that occur in this term that are used to name particular co-ordinates of input.

In order to make sure that the PR scheme created from the term on the right-hand-side of an equation in a type two ASTRAL specification receives the correct input we use the compiler $\mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}$ that follows. For example, if a type two ASTRAL specification is of the form

$$F(x_1, \ldots, x_n) = \tau$$

for some $x_i \in \underline{X}_{r_i}$ for some $r_i \in \underline{S}$ then we use the set $\mathbb{X} = \{x_1, \ldots, x_n\}$ as an index when we compile τ to create a scheme $\beta_\tau = \mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}(\tau)$ that can be composed with $\alpha_\tau = \chi^{\mathrm{A}^{\mathrm{T}_2}\mathbb{S}}(\tau)$ as follows:

$$\alpha = \alpha_\tau \circ \beta_\tau$$

to achieve a scheme with the desired semantics.

Definition 87. For each $\mathbb{X} = \{x_1, \ldots, x_m\}$ for some $x_i \in \underline{X}_{r_i}$, for some $r_i \in \underline{S}$ for $i = 1, \ldots, m \geq 1$, for each $u \in \underline{S}^+$ and for each $\underline{v} \in \underline{S}^+$ we define

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}_{u,\underline{v}} : \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u,\underline{v}} \rightarrow \mathrm{PR}(\underline{\Sigma})_{r_1 \cdots r_n, u}$$

(ambiguously denoted $\mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}$) uniformly in $(u, \underline{v})$ by induction on the structural complexity of a term $\tau \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u,\underline{v}}$ as follows:

Basis Cases.

(1) Stream Variables. If $\tau = x_i$ for some $x_i \in \mathbb{X}$ of sort $r_i \in \underline{S}$ then

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}(\tau) = \mathrm{U}_i^{r_1 \cdots r_m}.$$

Induction.

(2) Composition. If

$$\tau = H(\tau_1, \ldots, \tau_n)$$

for some $H \in \overline{\Sigma'}_{w,\underline{v}}$, for some $w = y^1 \cdots y^n \in \underline{S}^+$ for some $n \leq |w|$ and for some $\underline{v} \in \underline{S}^+$; and for some τ_i for $i = 1, \ldots, n$ such that either

$$\tau_i \in \mathrm{A}^{\mathrm{T}_1}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{s_i}$$

for some $s_i = y^i \in S$ or

$$\tau_i \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^i,\underline{z}^i}$$

for some $u^i \in \underline{S}^+$ and for some $\underline{z}^i = y^i \in \underline{S}^+$ then

$$\mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}(\tau) = \chi^{\mathrm{PR}^{\vee}}(\ll \gamma_1, \ldots, \gamma_n \gg)$$

wherein for $i = 1, \ldots, n$ we have

$$\gamma_i = \begin{cases} \mathbb{C}^{\mathrm{A}^{\mathrm{T}_2}}(\tau_i) & \text{if } \tau_i \in \mathrm{A}^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{u^i,\underline{z}^i}, \text{ and} \\ < \mathrm{U}_1^{r_1 \cdots r_m}, \ldots, \mathrm{U}_m^{r_1 \cdots r_m} > & \text{otherwise.} \end{cases}$$

6.3.2 Compiling ASTRAL Specifications

Using the compilers we have defined to compile ASTRAL terms we now define a number of compilers that map ASTRAL specifications into PR and PREQ, to realise our strategy for compiling full ASTRAL specifications (via type three ASTRAL specifications) in PREQ.

Type One ASTRAL Specifications. We begin by compiling type one ASTRAL specifications into RPREQ specifications.

Definition 88. Let $\overline{\Sigma'}$ and $\underline{\tilde{\Sigma}}$ be defined as in Definition 83. Also, let $\underline{X'} = \underline{X} \cup \{Y_1, \ldots, Y_{|v|}\}$ wherein Y_i is of type v_i of $i = 1, \ldots, |v|$. For each $u, \underline{v} \in \underline{S}^+$ we define

$$\chi^{\mathrm{ASTRAL}_1}_{u,\underline{v}} : \mathrm{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}} \to \mathrm{RPREQ}(\underline{\tilde{\Sigma}}, \underline{X'})_{t\,u,v}$$

(ambiguously denoted χ^{ASTRAL_1}) by case analysis on the structure of a specification

$$\Phi \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}}$$

as follows:

(1) Simple Specifications. If

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_n)(t) = \tau$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n$ and for some $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_s$ for some $s \in S$ then

$$\chi^{\mathrm{ASTRAL}_1}(\Phi) \stackrel{def}{=} f(t, x_1, \ldots, x_n) = \chi^{A^{T_1}}(\tau).$$

(2) Vector-Valued Simple Specifications. If

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_n)(t) = (\tau_1, \ldots, \tau_m)$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n$ and for some $\tau_j \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$ for some $s'_j \in S$ for $j = 1, \ldots, m \geq 1$ then

$$\chi^{\mathrm{ASTRAL}_1}(\Phi) \stackrel{def}{=} f(t, x_1, \ldots, x_n) = < \chi^{A^{T_1}}(\tau_1), \ldots, \chi^{A^{T_1}}(\tau_m) > .$$

(3) Primitive Recursive Specifications. If

$$\Phi \stackrel{def}{=} \begin{array}{lcl} F(x_1, \ldots, x_n)(0) & = & \tau \\ F(x_1, \ldots, x_n)(t+1) & = & \tau' \end{array}$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n$; for some $\tau \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_s$ for some $s \in S$; and for some $\tau' \in A^{T_1\{x_1, \ldots, x_n\}, \underline{s}}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_s$ then

$$\chi^{\mathrm{ASTRAL}_1}(\Phi) \stackrel{def}{=} \begin{array}{lcl} f(0, x_1, \ldots, x_n) & = & \chi^{A^{T_1}}(\tau) \\ f(t+1, x_1, \ldots, x_n) & = & \chi^{A^{T_1}\{x_1, \ldots, x_n\}, \underline{s}}(\tau'). \end{array}$$

(4) Vector-Valued Primitive Recursive Specifications. If

$$\Phi \stackrel{def}{=} \begin{array}{lcl} F(x_1, \ldots, x_n)(0) & = & (\tau_1, \ldots, \tau_m) \\ F(x_1, \ldots, x_n)(t+1) & = & (\tau'_1, \ldots, \tau'_m) \end{array}$$

for some $x_i \in \underline{X}_{s_i}$, for some $s_i \in \underline{S}$ for $i = 1, \ldots, n$; for some $\tau_j \in A^{T_1}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$ for some $s'_j \in S$; and for some $\tau'_j \in A^{T_1\{x_1, \ldots, x_n\}, \underline{s}'_1 \cdots \underline{s}'_m}(\underline{\Sigma}, \overline{\Sigma'}, \{x_1, \ldots, x_n\})_{s'_j}$ for $j = 1, \ldots, m \geq 1$ then

$$\chi^{\mathrm{ASTRAL}_1}(\Phi) \stackrel{def}{=}$$

$$\begin{array}{lcl} f(0, x_1, \ldots, x_n) & = & < \chi^{A^{T_1}}(\tau_1), \ldots, \chi^{A^{T_1}}(\tau_m) > \\ f(t+1, x_1, \ldots, x_n) & = & < \chi^{A^{T_1}\{x_1, \ldots, x_n\}, \underline{s}'_1}(\tau'_1), \ldots, \chi^{A^{T_1}\{x_1, \ldots, x_n\}, \underline{s}'_m}(\tau'_m) > . \end{array}$$

Type Two ASTRAL Specifications. We now show how to compile type two ASTRAL specifications into PREQ.

Definition 89. Let $\overline{\Sigma'}$ be defined as in Definition 83. For each $\mathbb{S} = \{\alpha_1, \ldots, \alpha_n\}$ wherein $\alpha_i \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,u',v'}$ for $i = 1, \ldots, n$ and for each $u, \underline{v} \in \underline{S}^+$ we define

$$\chi^{\mathrm{ASTRAL}_2^{\mathbb{S}}}_{u,\underline{v}} : \mathrm{ASTRAL}_2(\underline{\Sigma}, \overline{\Sigma'}, \underline{X})_{u,\underline{v}} \rightarrow \mathrm{PREQ}(\underline{\Sigma}, \underline{X})_{\mathbf{t}\,u,v}$$

(ambiguously denoted $\chi^{\mathrm{ASTRAL}_2^{\mathbb{S}}}$) as follows: if

$$\Phi \stackrel{def}{=} F(x_1, \ldots, x_m) = \tau$$

for some $x_i \in \underline{X}_{u_i}$ for $i = 1, \ldots, m \geq 1$ and for some $\tau \in A^{\mathrm{T}_2}(\underline{\Sigma}, \overline{\Sigma'}, \mathbb{X})_{w,\underline{v}}$, for some $w \in \underline{S}^*$ and for some $\underline{v} \in \underline{S}^+$ wherein $\mathbb{X} = \{x_1, \ldots, x_m\}$ then

$$\chi^{\mathrm{ASTRAL}_2^{\mathbb{S}}}(\Phi) = \begin{cases} \mathbb{C}^{\mathrm{PREQ}}(\chi^{A^{\mathrm{T}_2}\mathbb{S}}(\tau) \circ \chi^{\mathrm{PR}^{\mathbf{v}}}(\ll \mathrm{U}_1^{\mathbf{t}}, \mathbb{C}^{A^{\mathrm{T}_2}}(\tau) \gg)) & \text{if } |w| > 0; \text{ and} \\ \mathbb{C}^{\mathrm{PREQ}}(\chi^{A^{\mathrm{T}_2}\mathbb{S}}(\tau)) & \text{otherwise.} \end{cases}$$

Well-Definedness. We consider the case where $|w| > 0$ and leave the case where $|w| = 0$ to the reader. First, notice that by the well-definedness of $\mathbb{C}^{A^{\mathrm{T}_2}}$ we have

$$\mathbb{C}^{A^{\mathrm{T}_2}}(\tau) \in \mathrm{PR}(\underline{\Sigma})_{u,w}$$

and hence by the well-defineness of $\chi^{\mathrm{PR}^{\mathbf{v}}}$ we have

$$\gamma = \chi^{\mathrm{PR}^{\mathbf{v}}}(\ll \mathrm{U}_1^{\mathbf{t}}, \mathbb{C}^{A^{\mathrm{T}_2}}(\tau) \gg) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,u,\mathbf{t}\,w}.$$

Also notice that by the well-definedness of $\chi^{A^{\mathrm{T}_2}\mathbb{S}}$ we have

$$\chi^{A^{\mathrm{T}_2}\mathbb{S}}(\tau) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,w,v}$$

and hence clearly $\chi^{A^{\mathrm{T}_2}\mathbb{S}}(\tau) \circ \gamma \in \mathrm{PR}(\underline{\Sigma})$ with type $(\mathbf{t}u, v)$. Therefore finally by the well-definedness of $\mathbb{C}^{\mathrm{PREQ}}$ we have

$$\chi^{\mathrm{ASTRAL}_2^{\mathbb{S}}}(\Phi) = \mathbb{C}^{\mathrm{PREQ}}(\chi^{A^{\mathrm{T}_2}\mathbb{S}}(\tau) \circ \chi^{\mathrm{PR}^{\mathbf{v}}}(\ll \mathrm{U}_1^{\mathbf{t}}, \mathbb{C}^{A^{\mathrm{T}_2}}(\tau) \gg)) \in \mathrm{PREQ}(\Sigma, X)_{\mathbf{t}\,u,v}$$

as required.

Type Three ASTRAL Specifications. Using the three previous compilers we can now compile type three ASTRAL specifications into PREQ.

Definition 90. For each $u \in \underline{S}^+$ and for each $\underline{v} \in \underline{S}^+$ we define

$$\chi^{\mathrm{ASTRAL}_3}_{u,\underline{v}} : \mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})_{u,\underline{v}} \rightarrow \mathrm{PREQ}(\Sigma, X)_{\mathbf{t}\,u,v}$$

as follows: for each

$$\Phi = < \psi, \phi_1, \ldots, \phi_n > \in \mathrm{ASTRAL}_3(\underline{\Sigma}, \underline{X})_{u,\underline{v}}$$

for some

$$\psi \in \mathrm{ASTRAL}_2(\underline{\Sigma}, \{F_1, \ldots, F_n\}, \underline{X})_{u,\underline{v}}$$

for some $u, \underline{v} \in \underline{S}^+$ such that F_i is of type $(u^i, \underline{v}^i)$ for some $u^i \in \underline{S}^+$, for some $\underline{v}^i \in \underline{S}^+$ for $i = 1, \ldots, n \geq 1$; and for some

$$\phi_i \in \mathrm{ASTRAL}_1(\underline{\Sigma}, \{F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n\}, \underline{X})_{u^i, \underline{v}^i}$$

for $i = 1, \ldots, n$ such that

$$i \notin \mathrm{InTermsOf}(< \phi_1, \ldots, \phi_n >, i, \{F_1, \ldots, F_n\})$$

for $i = 1, \ldots, n$

$$\chi^{\mathrm{ASTRAL}_3}_{u, \underline{v}}(\Phi) = \chi^{\mathrm{ASTRAL}^3_2}(\psi)$$

wherein $\mathbb{S} = \{\alpha_1, \ldots, \alpha_n\}$ such that for $i = 1, \ldots, n$

$$\alpha_i \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, u^i, v^i}$$

is defined by

$$\alpha_i = \mathbb{C}^{\mathrm{PR}}(< \chi^{\mathrm{ASTRAL}_1}(\phi_1), \ldots, \chi^{\mathrm{ASTRAL}_1}(\phi_n), \iota, \eta, i >)$$

wherein $\iota : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ is the identity function on each $k \in \{1, \ldots, n\}$ and $\eta : \{1, \ldots, n\} \rightarrow \underline{S}^+ \times S^+$ is defined by

$$(\forall k \in \{1, \ldots, n\}) \quad \eta(k) = (\mathbf{t}\, u^k, v^k).$$

Well-Definedness. Notice that by the well-definedness of $\chi^{\mathrm{ASTRAL}^3_2}$ it suffices to show that

$$\alpha_i = \mathbb{C}^{\mathrm{PR}}(< \chi^{\mathrm{ASTRAL}_1}(\phi_1), \ldots, \chi^{\mathrm{ASTRAL}_1}(\phi_n), \iota, \eta, i >) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, u^i, v^i}$$

for $i = 1, \ldots, n$. In particular, notice by the well-definedness of χ^{ASTRAL_1} for each $k \in \{1, \ldots, n\}$ we have

$$\chi^{\mathrm{ASTRAL}_1}(\phi_k) \in \mathrm{RPREQ}(\tilde{\underline{\Sigma}}^k, \underline{X}')_{\mathbf{t}\, u^k, v^k}$$

wherein $\tilde{\underline{\Sigma}}^k$ is defined such that for each $w \in \underline{S}^*$ and for each $s \in S$

$$\tilde{\underline{\Sigma}}^k_{w,s} = \begin{cases} \underline{\Sigma}_{w,s} \bigcup \{f_{i,j}\} & \text{if } w = \mathbf{t}\, u^i \text{ and there exists an } F_i \in \overline{\Sigma'}_{u^i, v^i} \\ & \text{for some } i \in \{1, \ldots, k-1, k+1, \ldots, n\}, \\ & \text{for some } v^i = (s_1 \cdots s_m) \in \underline{S}^+ \text{ such that } \underline{s} = s_j \\ & \text{for some } j \in \{1, \ldots m\}, \text{ and} \\ \underline{\Sigma}_{w,s} & \text{otherwise} \end{cases}$$

and $\underline{X}' = \underline{X} \cup \{Y_1, \ldots, Y_{|v^k|}\}$ wherein Y_j is of type v_j of $j = 1, \ldots, |v^k|$. Therefore if for $i = 1, \ldots, n$ we can show that

$$\Phi_i = < \chi^{\mathrm{ASTRAL}_1}(\phi_1), \ldots, \chi^{\mathrm{ASTRAL}_1}(\phi_n), \iota, \eta, i > \in \mathrm{PREQ}(\Sigma, X)_{\mathbf{t}\, u^i, v^i}$$

then by the well-definedness of $\mathbb{C}^{\mathrm{PR}}$ we will have

$$\alpha_i = \mathbb{C}^{\mathrm{PR}}(\Phi_i) \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, u^i, v^i}$$

as required.

First, notice that by the definition of ι and η that Φ_i is standard. Therefore to complete our well-definedness argument it sufficies to show that

(1)

$$(\forall i \in \{1,\ldots,n\}) \quad i \notin \text{InTermsOf}(\Phi_i, i, \mathbb{F})$$

wherein

$$\mathbb{F} = \bigcup_{j=1}^{j=n} \underline{\Sigma}^j - \underline{\Sigma}$$

and

(2)

$$(\forall i \in \{1,\ldots,n\}) \quad \iota(i)\downarrow \Longrightarrow (u^{\iota(i)}\eta^D(i)) \wedge (v^{\iota(i)} = \eta^R(i)).$$

The fact that (1) holds follows from the hypothesis that

$$i \notin \text{InTermsOf}(< \phi_1,\ldots,\phi_n >, i, \{F_1,\ldots,F_n\}).$$

The fact that (2) holds follows from the fact that $\iota(i) = i$ for $i = 1,\ldots,n$ and the fact that $(\mathbf{t}\ u^i, v^i) = \eta(i)$ for $i = 1,\ldots,n$ by definition.

Compiling Full ASTRAL Specifications. As promised to complete our compilation process we now show how the compilation of full ASTRAL specifications can be achived using the compiler that maps type three ASTRAL specifications into PREQ. In particular, the compiler definition is based on an iterative process that reduces the complexity of the full ASTRAL specification until it involves only one type two ASTRAL specification that can then be compiled directly into PREQ using χ^{ASTRAL_3}.

Definition 91. For each $u \in \underline{S}^+$, for each $\underline{v} \in \underline{S}^+$ and for each

$$\Phi =< \psi, \psi_1,\ldots,\psi_m,\phi_1,\ldots,\phi_n >$$

for some

$$\psi \in \text{ASTRAL}_2(\underline{\Sigma}, \{F_1,\ldots,F_{m+n}\}, \underline{X})_{u,\underline{v}}$$

for some $u \in \underline{S}^*$ and for some $\underline{v} \in \underline{S}^+$, such that F_i is of type $(u^i, \underline{v}^i)$ for some $u^i \in \underline{S}^*$ and for some $\underline{v}^i \in \underline{S}^+$ for $i = 1,\ldots,m+n \geq 1$; for some

$$\psi_i \in \text{ASTRAL}_2(\underline{\Sigma}, \{F_1,\ldots,F_{i-1},F_{i+1},\ldots,F_m\}, \underline{X})_{u^i,\underline{v}^i}$$

for $i = 1,\ldots,m$; and for some

$$\phi_i \in \text{ASTRAL}_1(\underline{\Sigma}, \{F_{m+1},\ldots,F_{i-1},F_{i+1},\ldots,F_{n+m}\}, \underline{X})_{u^i,\underline{v}^i}$$

for $i = m+1,\ldots,m+n$ such that

$$i \notin \text{InTermsOf}(< \psi_1,\ldots,\psi_m,\phi_1,\ldots,\phi_n >, i, \{F_1,\ldots,F_{m+n}\})$$

for $i = 1,\ldots,m+n$ we define

$$\mathbb{C}^{\text{ASTRAL}}_{u,\underline{v}} : \text{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,\underline{v}} \rightarrow \text{PREQ}(\Sigma, X)_{\mathbf{t}\,u,v}$$

(ambiguously denoted $\mathbb{C}^{\text{ASTRAL}}$) by induction on the value $m \in \mathbb{N}$.
Basis $m = 0$. Notice in this case that

$$\Phi \in \text{ASTRAL}_3(\underline{\Sigma}, \underline{X})$$

and therefore we define $\mathbb{C}^{\text{ASTRAL}}$ by

$$\mathbb{C}^{\text{ASTRAL}}(\Phi) = \chi^{\text{ASTRAL}_3}(\Phi).$$

Well-Definedness. This follows immediately by the well-definedness of χ^{ASTRAL_3}.
Induction $m = k + 1$ for some fixed $k \in \mathbb{N}$. In this case we define $\mathbb{C}^{\text{ASTRAL}}$ by

$$\mathbb{C}^{\text{ASTRAL}}(< \psi, \psi_1, \ldots, \psi_{k+1}, \phi_1, \ldots, \phi_n >) = \chi^{\text{ASTRAL}_2^3}(\psi)$$

wherein

$$\mathbb{S} = \{\alpha_1, \ldots, \alpha_{k+n+1}\}$$

such that for $i = 1, \ldots, k+1$ we have

$$\alpha_i = \mathbb{C}^{\text{PR}}(\mathbb{C}^{\text{ASTRAL}}(< \psi_i, \psi_1, \ldots, \psi_{i-1}, \psi_{i+1}, \ldots, \psi_{k+1}, \phi_1, \ldots, \phi_n >));$$

and for $i = k+2, \ldots, k+n+1$ we have

$$\alpha_i = \mathbb{C}^{\text{PR}}(< \chi^{\text{ASTRAL}_1}(\phi_1), \ldots, \chi^{\text{ASTRAL}_1}(\phi_n), \iota, \eta, i >)$$

wherein $\iota : \{1, \ldots, n+m\} \rightsquigarrow \{1, \ldots, n\}$ is defined by

$$(\forall j \in \{1, \ldots, n+m\}) \quad \iota(j) = j - m$$

and $\eta : \{1, \ldots, n+m\} \rightarrow \underline{S}^+ \times S^+$ is defined by

$$(\forall j \in \{1, \ldots, n+m\}) \quad \eta(j) = (\mathbf{t}\, u^j, v^j).$$

Well-Definedness. The well-definedness in this case follows from the fact that by the Induction Hypothesis we have

$$\mathbb{C}^{\text{ASTRAL}}(< \psi_i, \psi_1, \ldots, \psi_{i-1}, \psi_{i+1}, \ldots, \psi_{k+1}, \phi_1, \ldots, \phi_n >) \in \text{PREQ}(\Sigma, X)_{\mathbf{t}\, u^i, v^i}$$

for $i = m+1, \ldots, m+n$, and a similar argument to the well-definedness of χ^{ASTRAL_3}, and is therefore left to the reader.

6.4 The Semantics of ASTRAL

We are now in a position to define the denotational semantics of ASTRAL specifications using PREQ. Notice that because ASTRAL has no independent semantics we cannot verify semantically the correctness of the compiler $\mathbb{C}^{\text{ASTRAL}}$ on which the semantics of ASTRAL relies as there is nothing to prove. Rather, we simply observe that this is by definition our intended semantics and that it is well-defined as the compiler $\mathbb{C}^{\text{ASTRAL}}$ is well-defined

Definition 92. Let $\underline{A}$ be any standard $\underline{S}$-sorted $\underline{\Sigma}$-algebra. We define the $\underline{S}^+ \times (\underline{S} - S)^+$-indexed family of maps

$$[\![.]\!]_{\underline{A}} = < [\![.]\!]_{\underline{A}}^{u,v} : \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,v} \rightarrow [\underline{A}^u \rightarrow \underline{A}^v] \mid u \in \underline{S}^+, v \in \underline{S}^+ >$$

wherein for each $u \in \underline{S}^+$ and for each $v \in \underline{S}^+$

$$[\![.]\!]_{\underline{A}}^{u,v} : \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,v} \rightarrow [\underline{A}^u \rightarrow \underline{A}^v]$$

(ambiguously denoted $[\![.]\!]_{\underline{A}}$) is defined for each $\Phi \in \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{u,v}$ as follows:

$$(\forall a \in \underline{A}^u)\,(\forall t \in T) \quad [\![\Phi]\!]_{\underline{A}}(a)(t) = [\![\mathbb{C}^{\mathrm{ASTRAL}}(\Phi)]\!]_{\underline{A}}(t, a).$$

This definition of the semantics of ASTRAL specifications completes the abstract formulation of our specification language for STs. In the following section we discuss the construction of a concrete syntax for an implementation of ASTRAL and present several example programmes.

6.5 Implementing ASTRAL

6.5.1 Introduction

As we have discussed in previous chapters, the development of the abstract syntax of ASTRAL has to a large extent been influenced by the underlying theoretical issues that we wished to address:

(1) Using primitive recursive functions (represented equationally) as a specification methodology.

(2) The reconciliation of a Cartesian form semantics with the need for more natural applicative form specifications.

With these aims now achieved, in this section we turn our attention to some of the practical issues that arise in the use of the abstract syntax and semantics of ASTRAL as a mathematically well-founded basis for a high-level specification and programming language. In particular, we present several examples of ASTRAL specification using a prototype BNF.

In the following section we begin by using ASTRAL to specify the stream processing primitives that we identified in our literature survey in Chapter 3, including a discussion of the computability theoretic issues that this raises. In particular, we will show that while ASTRAL relies on primitive recursive functions for the formalization of its semantics, from a practical perspective it is still possible to specify systems that require least number search (see Section 4.4) by simulating their behaviour. In more detail, any actual implementation will have finite constraints placed on the amount of memory that is available. Therefore, while some specifications may at the abstract level use least number search, when an actual programme representing such a specification is implemented this use of least number search can be effectively simulated by *bounded least number search* that is a computationally conservative expansion of the primitive

recursive functions. We explain this idea more fully in the sequel.

After the stream processing primitives we specify the RS-Flip-Flop (see Section 3.8.1) that we also use as a case study in the following chapter. However, we begin this chapter by motivating the prototype ASTRAL syntax. We do this by summarizing the underlying requirements that have shaped its development, and presenting the basic programming primtitives and their syntactic structure.

6.5.2 Developing an ASTRAL Implementation

From the practical perspective their are several requirements that we wish to address in the design of an implementation of ASTRAL:

(1) As far as possible ASTRAL should provide a general purpose programming methodology with a syntax that is in keeping with the style of modern languages such as C, PASCAL and C++. In particular, we require an implementation of ASTRAL to incorporate the following:

(A) ASTRAL syntax will include features for the specification of hardware as stream transformers.

(B) ASTRAL will provide support for the definition of user-defined data-types and will allow the definition and use of library function and project files to support software re-use and modular programming techniques.

Indeed, as a specific example, ASTRAL will provide library functions to support the use of the stream processing primitives and constructs that we identified in our literature survey in Chapter 3. In principle, this will enable specifications and implementations written in existing stream processing languages to be compiled into ASTRAL, and hence to be formally verified.

(2) While the abstract syntax of ASTRAL is declarative in style the implementation of ASTRAL will incorporate features from imperative languages to aid in the design of certain systems and algorithms where an imperative approach is more natural (compare with Section 3.8.4). For example, ASTRAL will incorporate iteration in the form of 'for' and 'while' loops (see Section 6.7).

(3) ASTRAL will provide strong support for the animation of specifications by efficient compilation into a suitable existing high-level language. In addition to the usefulness of this feature with respect to hardware verification – as animation is often helpful to establish what it is we wish to prove about a piece of hardware – efficient compilation of ASTRAL code is also required to make ASTRAL a viable general purpose programming language as per Requirement (1).

(4) The ASTRAL language will be incorporated into an ASTRAL environment to provide support for the formal verification of systems via their compilation into term re-writing systems (see Section 5.4).

In addition to the practical problems that the development of such an implementation of ASTRAL will present, many of the issues that arise themselves create further theoretical problems that must be overcome. This is particularly the case with respect to incorporating user-defined data types and parameterization issues that arises out of the specification of families of hardware. Indeed, the development of a full implementation of ASTRAL goes beyond the scope of this thesis, although it has been the subject of collaborative work between the author and M N Summerfield (see Summerfield [1994]). However, at the time of writing this thesis, research into these practical issues is only at a stage where a prototype BNF for ASTRAL has been developed. Therefore, rather than present this BNF formally, as there is a strong similarity between the implementation of ASTRAL and abstract ASTRAL in Section 6.6 we prefer to: (1) briefly discuss the general structure of ASTRAL; and (2) in Section 6.7 demonstrate ASTRAL's effectiveness as a specification language using some of the stream processing primitives we identified in Chapter 3 and the RS-Flip-Flop as examples.

6.6 The General Structure of ASTRAL Programmes

Our implementation of ASTRAL is based directly on full abstract ASTRAL specifications. Hence, ASTRAL is declarative wherein each programme is essentially a collection of type one and type two ASTRAL specifications. However, as our implementation is intended to be a general purpose high-level programming language, in addition ASTRAL programmes can also include: declarations of non-STs; and user-defined data types and user-defined constants; and 'abbreviations' to reduce the size and syntactic complexity of programmes.

To provide the reader with an overall perspective of the structure of our implementation we now briefly discuss some of ASTRAL's key features. Indeed, we re-iterate many of these points in more detail in the following section with the aid of our examples.

(1) **Signatures, Variables and Pre-Defined Data Types.** It is not necessary to make an explicit definition of the signature and variables required to make definitions in the ASTRAL implementation. Rather, this information is derived implicitly from each individual ST and function definition, although the user may explicitly define additional constants. Furthermore, reflecting the emphasis we place on the specification of hardware, the standard constants and operations associated with the following pre-defined data types are always available to the user without the need for their explicit inclusion: *bit*, *byte*, *bool*, *char*, *nat* and *int*. In addition, for each of these data type (and for each user-defined data type – see below), the associated array type, set type, stream type, stream of array type, stream of set type and the data type extended with the undefined element u are also available to the user without their explicit definition. For example: *bitArray*, *bitSet*, *bitStream*, *bitArrayStream*, *u_bitArray*, *u_bitSet*, *u_bitStream* and *u_bitArrayStream* are always available to the user. Notice, in particular that the real numbers are not supported as a pre-defined data type in ASTRAL, although they will be supported via a pre-defined library. This distinction is made for technical reasons concerning the formal verification of ASTRAL programmes as the real numbers are not *finitely generatable* (see Chapter 7).

(2) Definitions. There are six basic types of definition that may be made within each ASTRAL programme: two forms of AFST definition; function definitions; user-defined data types; user-defined constants; and abbreviations. We discuss each of these definitions in more detail.

(A) Evaluated AFST Definitions. These definitions have the following basic structure:

AFST_name(var_1 : d_type_1,...,var_n : d_type_n) r_type_1,...,r_type_m (t)
=
definition_body.

Evaluated AFST definitions are the the concrete representation of a type one ASTRAL specification (specifically indicated by the token '(t)') and hence *AFST_name* can only be used in the definition body in evaluated form. (The structure of the body of AFST and function definitions is discussed below.) Sort names in the range of an AFST definition must be stream sorts.

(B) Un-Evaluated AFST Definitions. These definitions have the following basic structure:

AFST_name(var_1 : d_type_1,...,var_n : d_type_n) r_type_1,...,r_type_m
=
definition_body.

Un-evaluated AFST definitions are the concrete representation of type two ASTRAL specification and hence *AFST_name cannot* be used in the function body. Again, sort names in the range of an AFST definition must be stream sorts.

(C) Function Definitions. These definitions have the following basic structure:

function_name(var_1 : d_type_1,...,var_n : d_type_n) r_type_1,...,r_type_m
=
definition_body.

This syntax is the concrete representation of a RPREQ specification and hence the function may also have stream sorts in the domain and range if the user wishes. Indeed, notice that as far as the user is concerned the only syntactic distiction between un-evaluated AFST and function definitions is the range type of the defined function.

(D) User-Defined Type Definitions. User defined data types come in three basic forms: restrictions of pre-defined types, compound types and type unions:

(a) Type Restriction Definitions. These definitions must conform to one of the three following forms:

type type_name isa pre-defined_type_name .
type type_name isa pre-defined_type_name (constant_value to constant_value).
type type_name isa pre-defined_array_type_name [constant_value].

Examples:

type totals isa natArray .
type register isa nat (0 to 32) .
type lowercasechar isa char ("a" to "z") .
type smallintarray isa intArray [5] .

(b) Compound Type Definitions. These definitions have the following form:

type type_name compsises
    component_type_list.

wherein a *component_type_list* is comprised of any one of the three basic type definitions except that the keyword *type* is omitted and a variable name replaces the user-defined type name. Example:

type employeetype comprises
    Name isa charArray [50].
    Number isa nat (1 to 100).
    PartTime isa u_bool.

Thus *employeetype* is a compound type with three fields called *Name*, *Number* and *PartTime* respectively. In particular, if as part of an AFST header definition we were to define the variable '*Employee : employeetype*' as an input then the values of the components are accessed as follows: *Employee.Name*, *Employee.Number* and *Employee.PartTime* respectively.

(c) Union Type Definitions. These definitions have the following form:

typeunion type_name isa type_name_list.

Example:

typeunion intchar isa int, char.

(D) User-Defined Constants Definitions. User-defined constants may be declared over both pre-defined and user-defined data types. These definitions have the following basic structure:

const type const_name is const_val_list .

Examples: (using the definitions of the user-defined types above)

const nat MAX_STRING_LEN is 50 .
const smallintarray ZERO_SMALL_INT_ARRAY is 0, 0, 0, 0, 0 .
const employeetype NULL_EMPLOYEE is "", 0, u .

(E) Abbreviation Definitions. Practical experience with the specification of hardware has shown that there is a need for a mechanism for the definition of tokens that can be used as abbreviations for complex expressions that occur frequently. However, the specific mechanism in which this feature will be included into ASTRAL is at present under-developed, although we do include an simple example in the following section of how we envisage abbreviations will be used.

(3) AFST and Function Body Definitions. There are four basic types of compound expressions that can be used in the body of AFST and function definitions: *case statements*, *ifmatch statement*, *for ... statements* and *for ... while ... statements*. These are explained in the following section using our examples.

There are several further features of ASTRAL that we have not presented and discussed above as they are not easily motivated by a simple presentation of a BNF definition and some examples. These include *template functions* that can be used to define families of functions, input and output mechanisms, and libraries. Summerfield [1994] gives a more detailed presentation of the features of our prototype ASTRAL implementation and discusses these addition features.

6.7 Example ASTRAL Specifications

6.7.1 Existing Stream Processing Primitives

We begin our example ASTRAL specifications by representing the various classes of stream processing primitives we identified in our literature survey in Chapter 3. Where appropriate we also include the corresponding abstract ASTRAL definition of a primitive. However, we will not formally define a compiler that generates abstract ASTRAL definitions from an ASTRAL programme (implementation) as, while this is essentially straightforward, their are still several subtle practical issues that must be resolved that go beyond the scope of this thesis. We mention each of this points in the sequel where appropriate.

In particular, one of the points that we aim to emphasise during our discussions, is that, while ASTRAL is restricted to the specification of primitive recursive functions, *in practice* ASTRAL is not limited in the sense of the class of actual systems, either hardware or software, that we can specify.

Functional Stream Processing Primitives. The reader should refer back to Section 3.4.1 for a definition of the following primitives. Also, note that in the following definitions we will use the word 'sort' to indicate any valid basic type from our underlying signature Σ. Furthermore, we assume that any operations that are not explicitly defined are part of the underlying signature.

In particular, we always assume that equality denoted *eq* is always available on any basic (non-stream) type.

(1) Stream Construction Operator. This can be defined in ASTRAL as follows:

cons(a : sort, s : sortStream) sortStream (t)
=
a if t = 0;
s('t).

Discussion. First, notice that as we mentioned in the previous section, in the implementation of ASTRAL there is no explicit information concerning the underlying signature and (local) variable set. Rather, this information is defined implicitly by the header associated with each function definition. In particular, in this case we have one function *cons* with type (*sort* $\underline{sort}$, $\underline{sort}$) and our variable set is comprised of two variables a and s of type *sort* and $\underline{sort}$ respectively.

Secondly, notice that the typing information at the end of the function header is post-fixed with '*(t)*'; that is, the functions range is post-fixed with '*(t)*'. This is to indicate that while in this case the function *cons* returns a stream we are actually specifying the function at some time t; that is, we are specifying *cons* in evaluated form. Indeed, this concrete syntax corresponds to an abstract definition of a type one ASTRAL specification. The use of '*(t)*' also implicitly indicates that that our (local) variable set is extended with the symbol t of type *nat*.

In general a function definition body (that follows the '=') is comprised of a group of expressions structured using various constructs and is terminated by a '.'. In the particular instance of *cons* we have a two-way *case statement* (*if ... then ... else* clause), although multi-way case statements are also permitted. (The symbol ';' can be used as a shorthand for 'else'.)

The symbol '" is also a shorthand that may be pre- and post-fixed to any numerical expression to indicate the predecessor and successor respectively of the expressions value. For example, *'t* is shorthand for *pred(t)*, *t'* is shorthand for *succ(t)*, *t"* is shorthand for *succ(succ(t))* and so forth.

An Equivalent Abstract ASTRAL Specification. This concrete representation of *cons* corresponds to the following abstract ASTRAL definition: let $X \supseteq \{a, s, t\}$ such that a, s and t are of type r, $\underline{r}$ and **n** respectively for some $r \in (\underline{S} - S)$. We define

$$\Phi =< \psi, \phi > \in \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{r\,\underline{r},\underline{r}}$$

wherein

$$\psi \stackrel{def}{=} F(a, s) = cons(a, s)$$

and

$$\phi \stackrel{def}{=} cons(a, s)(t) = dc_r(eq_{\mathbf{n}}(t, 0), a, s(pred(t))).$$

Notice in particular that *dc* (*definition-by-cases*) has allowed us to eliminate the cases statement and hence to convert a limited form of conditional equations to a 'pure' equational representation (see Section 4.2.3). Also, notice that as we indicated in Section 6.2.2 the role of the type two ASTRAL specification in this definition is simply to indicate that the functional *cons* is to be used as the semantics of the whole specification.

(2) Concatenation. This can be defined in ASTRAL as follows:

typeunion sortGenStream isa sortArray, sortStream.

concat(s1 : sortGenStream, s2 : sortGenStream) sortGenStream (t)
=
s1(t) if size(s1) < t;
s2(t - size(s1)).

size(s : sortGenStream) nat
=
arraysize(s) if isa_array(s);
MAX_NAT.

Discussion. First, notice that because the stream concatenation operator relies on the more general notion of a stream (see Chapter 2), in order to simulate the operation of *concat* in ASTRAL we have defined a user-defined type *sortGenStream* that is the union of the array and stream type for the type *sort*. This is achieved by the definition

typeunion sortGenStream isa sortArray, sortStream.

(wherein ',' is an abbreviation for the word 'and') that allows us to explicitly describe information about the underlying signature over which the function definitions that we make are defined. However, the order of definitions in ASTRAL is unimportant and hence this declaration could have appeared anywhere in the specification.

One difficulty that arises from the use of type unions for this purpose, is that we now require the operation *isa_array* that essentially allows us access to information about an expressions type; that is, that allows us to tell which basic type an expression has. While the use of such operations can be eliminated during the compilation into abstract ASTRAL using a similar coding technique to that discussed in Section 4.5.4 a detailed discussion of this process goes beyond the scope of this thesis.

Also, notice that the implementation of ASTRAL allows non-STs to be defined; that is, specifically in this case the function *size*. In practice, this is not a problem as the function *size* is (R)PREQ definable and hence can be considered to be part of the underlying signature. Again the details of compiling ASTRAL specifications that incorporate non-STs goes beyond the scope of this thesis (although, Chapter 7 and in particular the definition of the function AV (Definition 109) goes some way to showing how this process can be formalized).

(3) First Element Selection. This can be defined in ASTRAL as follows:

hd(s : sortStream) sort
=
s(0).

Discussion. As with the function *size* in the previous definition, the function *hd* is not strictly a stream transformer and hence cannot be defined in abstract ASTRAL. However, again in practice this is not a problem as *hd* is PREQ definable.

(4) First Element Elimination. This can be defined in ASTRAL as follows:

tl(s : sortStream) sortStream (t)
=
s(t').

An Equivalent Abstract ASTRAL Specification. Let $X \supseteq \{s\}$ such that s is of type $\underline{r}$ for some $r \in (\underline{S} - S)$. We define

$$\Phi =< \psi, \phi > \in \text{ASTRAL}(\underline{\Sigma}, \underline{X})_{\underline{r},\underline{r}}$$

wherein

$$\psi \stackrel{def}{=} F(s) = tl(s)$$

and

$$\phi \stackrel{def}{=} tail(s) = s(succ(t)).$$

(5) Last Element Selection. This can be defined in ASTRAL as follows:

typeunion sortGenStream isa sortArray, sortStream.

last(s : sortGenStream) u_sort
=
u if isa_stream(s);
s(size(s)).

Discussion. As with the operation *concat* the definition of *last* in ASTRAL requires the more general notion of a stream. Also, notice that to mirror the conceptual relationship between a finite sequence (a one-dimensional array) and a stream (an infinite sequence) that array evaluation is also represented as '*a(t)*' for some array *a*. Finally, notice that ASTRAL has an *overloaded* constant *u* associated with each type to represent an undefined value. This is why the range of the function *last* is *u_sort* wherein *u_sort* $\supset$ *sort* is *sort* extended with the additional element *u*.

(7) Filtering. This can be defined in ASTRAL as follows:

filt(S : sortSet, s : sortStream) u_sortStream (t)

=

s(e) for e = t to MAX_NAT

while not s(e) in S;

u.

Discussion. First, and most importantly, notice that the mathematical definition of *filt* as presented in Section 3.7.2 implicitly requires the use of least number search. Indeed, it is a feature of unrestricted second-order equations that it is straightforward to implicitly incorporate least number search and hence to specify partial functions. Therefore as ASTRAL's semantics is derived using primitive recursive functions strictly speaking we cannot define filt using abstract ASTRAL.

However, by considering the fact that we are dealing with an implementation of ASTRAL and that in turn any functional language must be implemented on a machine with finite constraints on memory it is possible in practice to define a simulation of the filtering operation. Moreover, we argue that this simulation is essentially indistinguishable from any implementation of a functional language representation of *filt*.

In more detail, we can specify a simulation of *filt* using the *for ... while ...* construct that is a feature of the ASTRAL implementation. The intended semantics of the *filt* simulation using the *for ... while ...* construct is as follows:

$$filt(S,s)(t) = \begin{cases} s(e) & \text{wherein } e \text{ is the least value from the set} \\ & \{t,\ldots,MAX\_NAT\} \text{ such that } s(e) \in S, \text{ and} \\ u & \text{if no such } e \text{ exists.} \end{cases}$$

Essentially, the *for ... while ...* construct is a concrete representation of bounded least number search that is a computationally conservative expansion of the primitive recursive functions (see for example Cutland [1980] and also Section 4.4). Indeed, this simulation of *filt* (that is primitive recursive) can be represented in the abstract syntax of ASTRAL as follows:

An Equivalent Abstract ASTRAL Specification. Let $MAX\_NAT \in \Sigma_{\lambda,\mathbf{n}}$ and $X \supseteq \{S,s,t,x\}$ such that S, s, t, x are of type $\wp(r)$, $\underline{r}$, $\mathbf{n}$ and $\mathbf{n}$ respectively for some $r \in (\underline{S}-S)$. We define

$$\Phi =< \psi, \phi_1, \phi_2 > \in \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{r\,\underline{r},\underline{r}}$$

wherein

$$\psi \stackrel{def}{=} F(S,s) = filt(S,s),$$

$$\phi_1 \stackrel{def}{=} filt(S,s)(t) = lst(t,S,s)(MAX\_NAT)$$

and

$$\phi_2 \stackrel{def}{=}$$

$$
\begin{array}{lcll}
lst(x,S,s)(0) & = & dc(& in(s(x),S),x,u) \\
lst(x,S,s)(t+1) & = & dc(& and(in(s(t+x+1),S),eq_r(lst(x,S,s)(t),u)), \\
 & & & t+x+1, \\
 & & & lst(x,S,s)(t) \\
 & &) &
\end{array}
$$

Notice that the function *lst* simulates the bounded least number search that we require, although in this particular example at the abstract syntax level it does this inefficiently. A discussion of compilation techniques using more efficient methods again goes beyond the scope of this thesis.

(8) Pointwise Change. This can be defined in ASTRAL as follows:

pc(s : sortStream, n : nat, x : type) typeStream (t)
=
    s(t) if t <> n;
    x.

Relational Stream Processing Primitives. The definition of all the relation stream processing primitives identified in Section 3.7.3 are straightforward and are left to the reader.

LUCID Primitives. We now specify some of the LUCID stream processing primitives we identified in Section 3.8.4. In particular, we specify the operators *whenever*, *asa* and *upon* and leave the operators *first*, *next*, *fby* and *attime* to the reader.

(5) Whenever. This can be defined in ASTRAL as follows:

whenever(s : sortStream, b : boolStream) u_sortStream (t)
=
    s(t) if b(t) = true;
    whenever(s,b)(n) for n = t' to MAX_NAT
        while not b(n);
    u.

Discussion. As with the functional primitive *filt* the operator *whenever* implicitly requires the use of least number search and therefore the above definition is a primitive recursive simulation of *whenever*. However, as with *filt* we argue that this simulation of *whenever* is indistinguishable from any implementation of the partial version of *whenever*.

We leave the definition of the abstract ASTRAL specification of *whenever* to the reader.

(6) As Soon As. This can be defined in ASTRAL as follows:

asa(s : sortStream, b : boolStream) u_sortStream (t)
=

```
s(n) for n = 0 to MAX_NAT
    while not b(n);
u.
```

Discussion. Agian we are restricted to defining a primitive recursive approximation of *asa*. The definition of the abstract ASTRAL specification of *asa* is again left to the reader.

(7) Upon. This can be defined in ASTRAL as follows:

```
upon(s : sortStream, b : boolStream) sortStream (t)
=
    s(0) ifmatch (_,_,0);
    s(numofts(b)(t)-1) ifmatch (_,(0)true,<>0);
    s(numofts(b)(t)).

numofts(b : boolStream) natStream (t)
=
    1 ifmatch ((0)true,0);
    0 ifmatch ((0)false,0);
    1 + numofts(b)('t) ifmatch ((t)true,<>0);
    numofts(b)('t).
```

Discussion. The operator *upon* provides an example of the use of the *ifmatch* construct in ASTRAL that allows case statements to be presented in a more concise format.

The *ifmatch* construct can be used in place of the keyword *if* in a cases statement, and must be followed by a bracket-enclosed, comma separated list of the same length as the number of arguments in the domain of the function in which the *ifmatch* statement appears. For example, in the case of the definition of *upon* the list must be of length three, and in the case of *numofts* the list must be of length two (as t is considered to be an input). In addition, each elemenent of the list must be a constant expression of the same type as the corresponding co-ordinate of the domain, with the exception of stream elements wherein if the expression is not a 'don't care' then the expression must be of the same type as the evaluated corresponding co-ordinate of the domain (see below).

Within an *ifmatch* statement the tokens '_', '*(.)*' and '<>' wherein '.' denotes any natural number expression, are used with the following meaning: the token '_' may be read as 'don't care' and will match with any value of an appropriate type; the token '*(.)*' may be pre-fixed to any expression wherein the corresponding domain co-ordinate from the function that we are defining is a stream type; that is, this type of expression will match if, and only if the expression before the token '*(.)*' matches the corresponding stream co-ordinate evaluated at time '.' – see below for an example; and the token '<>' pre-fixed before an expression can be read as 'not a' and will cause the statement to match at that argument with any input that does not have that value.

Using the *upon* specification as our example the *ifmatch* statement above is equivalent to the following alternative ASTRAL programme:

upon(s : sortStream, b : boolStream) sortStream (t)

=

s(0) if t = 0;
s(t) if t <> 0 and b(t) = true;
upon(s,b)('t).

numofts(b : boolStream) natStream (t)

=

1 if b(t) = true, t = 0;
0 if b(t) = false, t = 0;
1 + numofts(b)('t) if b(t) = true, t <> 0;
numofts(b)('t).

An Equivalent Abstract ASTRAL Specification. Let $X \supseteq \{s, b, t\}$ such that s, b and t are of type $\underline{r}$, $\underline{\mathbf{b}}$ and $\mathbf{n}$ respectively for some $r \in (\underline{S} - S)$. We define

$$\Phi =< \psi, \phi > \in \mathrm{ASTRAL}(\underline{\Sigma}, \underline{X})_{\underline{r}\,\underline{\mathbf{b}},\underline{r}}$$

wherein

$$\psi \stackrel{def}{=} F(s,b) = upon(s,b)$$

and

$$\psi \stackrel{def}{=}$$

$$\begin{aligned} upon(s,b)(0) &= s(0) \\ upon(s,b)(t+1) &= dc_r(b(Succ(t)), s(Succ(t)), upon(s,b)(t)) \end{aligned}$$

LUSTRE Primitives. These are very similar in form to the LUCID primitives and are again left to the reader.

ESTEREL Primitives. These are omitted.

STREAM Primitives and Constructs. We define all the STREAM primitives with exception of *&*, ***, *selec*, sequential composition, *C*, *fork* and *perm* that as before are straightforward and left to the reader. However, notice with respect to the 'feedback' operator C that ASTRAL is restricted to primitive recursive feeedback. Also, for those primitives and constructs where we do give an ASTRAL specification of a STREAM primitive we leave the construction of an equivalent abstract ASTRAL specification to the reader.

(3) Distribution. This can be defined in ASTRAL as follows:

distr(b : boolStream, s : sortStream) sortStream sortStream (t)
& x1 "leasttrue(b,t)".
& x2 "leastfalse(b,t)".

=

(x1,x2).

```
leasttrue(b : boolStream, n : nat) nat
=
    e for e = n to MAX_NAT
        while not b(e);
    u.

leastfalse(b : boolStream, n : nat) nat
=
    e for e = n to MAX_NAT
        while b(e);
    u.
```

Discussion. Notice that as with some previous primitives we are restricted to defining a primitive recursive approximation of *distr*.

The definition of *distr* provides an example of a vector-valued function definition and also the use of (local) abbreviations. In particular, in ASTRAL the symbol '&' (not to be confused with the STREAM primitive '&') followed by two expressions, defines the first expression to be an abbreviation for the second expression. Indeed, the second expression can be enclosed within quotes to avoid any ambiquity during parsing. Also, unless placed between a function header and the following '=', as in the above definition, abbreviations have global scope. Essentially, if defined correctly then an abbreviation may be used as a variable of an appropriate type.

In addition, we also envisage that a full implementation of ASTRAL will allow more complicated abbreviations including 'indexed' familes of abbreviations and nested abbreviations.

(a) Parallel Composition. This can be defined in ASTRAL as follows:

```
F(s1 : sort1Stream, s2 : sort2Stream) sort3Stream sort4Stream =
    (G(s1),H(s2))

G(s : sort1Stream) sort3Stream(t) =
    ... .

H(s : sort2Stream) sort4Stream(t) =
    ... .
```

Discussion. In the above definition '...' is used in the definitions of the functions G and H to represent any legal ASTRAL expressions. Furthermore, the types of the functions in the specification above are simply examples and could be of any type. Essentially, the use of what amounts to type-two ASTRAL specifications (function F in the above example) allows the possibility of parallel execution as the evaluation of G and H may

be done independently. However, the exploitation of such parallelism will depend on the underlying architecture on which ASTRAL is implemented.

The definition of selected primitives for the STREAM languages concludes our simple ASTRAL specification examples. In the following section we define the RS-Flip-Flop (see Section 3.8.1) that we will use as case study in Chapter 8.

6.7.2 Specifying the RS-Flip-Flop in ASTRAL

We now present a more complex example by defining the specification and an implementation of our running example the RS-Flip-Flop as ASTRAL programmes.

RSFlipFlopSpec(s1,s2 : boolStream) boolStream (t)
=
true if t = 0;
false if t > 0, s1(t) = true, s2(t) = false;
true if t > 0, s1(t) = false, s2(t) = true;
RSFlipFlopSpec(s1,s1)(t).

and

RSFlipFlopImp(s1,s2 : boolStream) boolStream boolStream
=
OutSch(FFlop(true,false,InpSch(s1,s2))).

FFlop(b1,b2 : bool, s1,s2 : boolStream) boolStream boolStream (t)
=
(b1,b2) if t = 0;
(FFlop1(b1,b2,s1,s2)(t) nor s2, s1 nor Fflop2(b1,b2,s1,s2)(t)).

OutSch(s1,s2 : boolStream) boolStream (t)
=
*s1(t * 2).*

InpSch(s1,s2 : boolStream) boolStream boolStream (t)
=
(s1(t div 2), s2(t div 2)).

Discussion. The ASTRAL programme to represent the implementation of the RS-FlipFlop provides the first example of the implicit use of Cartesian composition. In particular, notice that the function body of the definition of *RSFlipFlopImp* is essentially a composition of the three applicative stream transformers: *OutSch* – representing an output scheduling function: *Fflop* – represting the actual RS-FlipFlop device; and *InpSch* – representing an input scheduling function. Therefore, as we can only specify the Cartesian forms of the ASTRAL representation in PREQ the formulation of a PREQ specification of the entire RS-FlipFlop implementation will require the use of the extended Cartesian composition compiler $\mathcal{C}$ (see Section 6.1.1).

Chapter 7

Automated Verification

It is a great advantage of a system of philosophy to be substantially true.

George Santayana

7.1 Introduction

Recall in Section 3.10.4 that the fourth part of our research agenda was concerned with the investigation of automatic software tools for deciding the equality of two ASTRAL programmes under initial algebra semantics. More specifically, we promised that using Theorem 11 of Chapter 5 we can identify a syntactic sub-class of all decidable equational correctness statements relating ASTRAL programmes that can be verified automatically using first-order equational logic. In order to complete this task it is necessary for us to do the following:

(1) Identify a logical calculus that is sound with respect to the equality of STs using initial algebra semantics. For this purpose we define the calculus EQWIL that formalizes equational logic augmented with induction and case analysis as a proof system.

(2) Formulate an effective decision procedure for EQWIL that can be used as the basis of our automated verification tools. For this purpose we define the related functions VER and EVER both based on term re-writing techniques.

(3) Reduce deductions about Cartesian form (weak second-order) equational correctness statements in weak second-order systems of equations to deductions about first-order equations in first-order systems of equations so that we may apply first-order techniques. For this purpose we define the function SubEvals that systematically eliminates occurrences of stream variables and replaces them with first-order terms.

We now discuss and motivate each of these points in more detail.

7.1.1 Overview

In the literature software tools based on automated decision procedures are often synonymously referred to as either *proof assistants* or *proof tools* without (as far as we are aware) any rigorous definition of what these terms imply with respect to the software's expected behaviour. From the perspective of our research, in order to clarify the results that we present, we find it is useful to begin this chapter by formalising these ideas. In particular, we find it useful to identify and classify the properties of four types of abstract device suitable to verify the equality of two STs:

(A) A *proof assistant*: a device that implements a partial function that can perform deductions in a formal calculus, but that may either fail to find a proof of a hypothesis or fail to terminate even if a proof of the hypothesis exists.

(B) A *total proof assistant*: a device that implements a *total* function that can perform deductions in a formal calculus; that is, a device that may fail to find a proof of a hypothesis even if a proof of the hypothesis exists, but always terminates.

(C) A *proof tool*: a device that implements a *partial* function that can perform deductions in a formal calculus and that can find a proof of a hypothesis if and only if such a proof exists. However, a proof tool may still fail to terminate if no proof of a hypothesis exists.

(D) A *total proof tool*: a device that implements a *total* function that can perform deductions in a formal calculus; that is, a proof tool that can find a proof of a hypothesis if and only if such a proof exists and that always terminates.

We note at this point that it is implicit in the above definitions that both proof assistants and proof tools must always be sound with respect to some intended semantics (the initial semantics in the context of this chapter). However, even for total proof tools it should not be inferred from these definitions that we assume that these devices implement a complete proof system – that is, the non-existence of the proof of a hypothesis in the formal calculus that either a proof assistant or proof tool implements does not, in general, imply that the hypothesis is false. Indeed, as we mentioned in Section 3.10.4 unfortunately because of Gödel's incompleteness result and the negative result to Hilbert's tenth problem concerning the solution of Diophantine equations (see Davis *et al.* [1976]), in general it is impossible to design a sound and complete, total proof tool that will verify the equality of two primitive recursive functions under initial algebra semantics. In particular, as we have already pointed out, in the context of SCAs, that are a proper sub-class of the STs that are representable by ASTRAL programmes, this is because the solution to such a problem is equivalent to deciding the membership of a non-recursive, co-recursively enumerable set (see Thompson and Tucker [1994]).

Therefore, with respect to the use of EQWIL, that provides a powerful, but in general incomplete proof system, in this chapter we shall prove the following:

(I) There exists a general purpose total proof assistant that can verify the equality of two primitive recursive STs under initial algebra semantics.

(II) It is possible to identify non-trivial classes of correctness statements relating primitive recursive STs under initial algebra semantics for which there exists a total proof tool.

In particular, we will show that the function VER has these two properties for first-order systems of primitive recursive equations, and the function EVER has these two properties for systems of weak second-order primitive recursive equations.

In more detail, we will show that the functions VER and EVER behave as total proof tools in the context of first-order and weak second-order systems of primitive recursive equations respectively. Moreover, we will also show (Theorems 15 and 17) that we can identify syntactically four classes of correctness statements for which VER and EVER behave as total proof tools.

Informally, in the context of EVER (in increasing order of significance) these four classes of correctness statements are characterized as follows: (in the case of VER it is the same four classes of equations except restricted to strictly first-order equations)

(A) Ground terms equations.

(B) Equations whose variables either range over finite carriers or whose variables range over stream carriers of the form $[T \rightarrow A]$ wherein A is a finite carrier.

(C) Equations of the form

$$f(x, t_1, \ldots, t_n) = c$$

wherein $x \in X_n$, c is some constant and the terms t_i for $i = 1, \ldots, n \in \mathbb{N}$ contain variables that either range over finite carriers or range over stream carriers of the form $[T \rightarrow A]$ wherein A is a finite carrier.

(D) Equations of the form

$$f(x, t_1, \ldots, t_n) = f'(x, t'_1, \ldots, t'_m)$$

for some $f \neq f'$ wherein $x \in X_n$ and for $i = 1, \ldots, n \in \mathbb{N}$ and for $j = 1, \ldots, m \in \mathbb{N}$ respectively the terms t_i and t'_j contain variables that either range over finite carriers or range over stream carriers of the form $[T \rightarrow A]$ wherein A is a finite carrier.

Of these four types of equations notice that Class (D) captures the class of correctness statements that can be used to represent the equality of a very broad class of hardware devices when they are expressed as CFSTs. This is our main result and we will return to this point in Section 7.5.

7.1.2 Reasoning about Weak Second-Order Systems

While Birkhoff's Theorem cannot, in general, be applied to higher-order systems of equations (see Meinke [1992b]), we now discuss how, in the context of weak second-order systems of equations, it is possible to reasoning about the initial truth of equational statements by systematically eliminating occurences of the *eval* operator during a deduction.

First, we show that the provability of the initial truth of certain classes of first-order equational correctness statements, relative to primitive recursive systems of equations, is decidable with respect to the calculus EQWIL (Theorem 15). We prove this using Birkhoff's Theorem and Theorem 11 by demonstrating that VER can simulate deductions in EQWIL, and observing that EQWIL is sound with respect to the initial truth of first-order theories (Theorem 13).

Secondly, we show that EQWIL's soundness is preserved with respect to the initial truth of weak second-order systems of equations (Lemma 50).

Finally, we show that relative to systems of Cartesian form equations, deciding the initial truth of an equational correctness statement $e \in \mathrm{EQ}(\underline{\Sigma}, \underline{X})$ using EQWIL can be reduced to deciding the initial truth of an equation $e' = (t = t') \in \mathrm{EQ}(\underline{\Sigma}, \underline{X})$ wherein any stream variable $x \in X_{\underline{s}}$ for some $s \in S$ that occurs in either term t or term t' must be part of a term of the form $\eta = eval(\tau, x)$ for some $\tau \in T(\underline{\Sigma}, \underline{X})$. In particular, we show that by replacing each occurrence of η in e' with a new variable symbol $y \in X_s$ we can derive a new strictly first-order equation e'' such that the initial truth of e'' implies the initial truth of e. Moreover, we show (Lemma 58) that this proof method can be implemented by combining the function VER (Definition99) with the function SubEvals (Definition 104) to give the function EVER (Definition 105). Therefore, we show that it is possible to generalize Theorem 15 concerning first-order systems of equations to Theorem 17 concerning weak second-order systems of equations. This process is summarized in the following diagram:

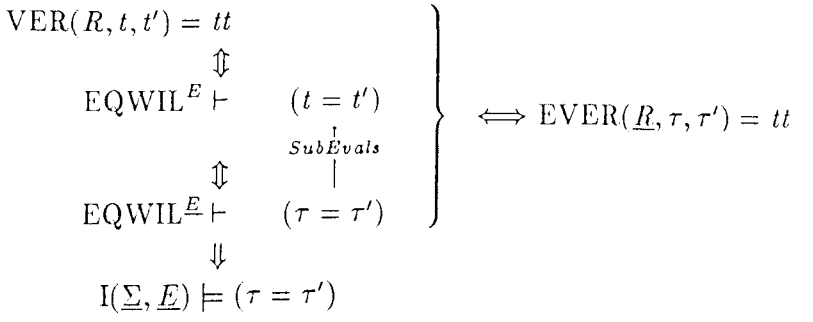

Figure 7.1: A Schematic Representation of our Proof Technique

Thus, we have a total proof tool (with respect to EQWIL) for reasoning about the correctness of a broad and useful class of systems, that is based on strictly first-order term re-writing techniques.

A more detailed overview of the rest of this chapter is as follows.

As the functions VER and EVER are based on term re-writing techniques we begin in Sections 7.1.3 and 7.1.4 by discussing the relationship between equational logic, term re-writing and truth in the initial model of a set of equations in more detail.

This discussion motivates the development of the calculus EQWIL in Section 7.2 that is based on equational logic. In particular, in Section 7.2.2 we define the key idea involved in the formal development of EQWIL; that is, the concept of a *signature of constructors* that can be used to finitely generate all members of the carriers of certain initial algebras, and that enables us to formalize the calculus EQWIL itself in Section 7.2.4.

While our main interest in this chapter is the study of the automated verification of STs, for generality we find it useful to begin exploring the properties of the calculus EQWIL in the context of strictly first-order systems of equations. In Section 7.3.1 we show that EQWIL is sound with respect to truth in the initial models of sets of first-order equations. We also present some limited completeness results. In Section 7.3.2 we begin to explore the automation of EQWIL by defining the function VER that is suitable as the basis of both a general purpose total proof assistant, and total proof tool in the context of the four classes of correctness statements we discussed in the introduction. These results concerning VER are formalized in Section 7.3.2.

Using these general results as a basis, in Section 7.4 we focus our attention on the automation of the calculus EQWIL in the context of the verification of STs. In Section 7.4.1 we first show that EQWIL is sound with respect to the initial semantics of weak second-order systems of equations. In Section 7.4.2 we discuss the limitations of VER with respect to stream algebras, but show how it can be extended to give the function EVER that is more appropriate as the basis of an automated theorem proving tool specifically for STs.

In Section 7.4.3 we complete the fourth part of our research agenda by showing how we may use EVER to specify a function AV that, in the context of STs specified as ASTRAL programmes, is both a general purpose total proof assistant, and a total proof tool in the context of the four classes of correctness statement we have identified.

Finally, in Section 7.5 we discuss the implications of the theoretical results concerning the

functions VER, EVER and AV in the context of SCAs (see Section 3.10) and in particular in the context of certain classes of hardware devices. This discussion clarifies how our technical results satisfy the claims we made in statements (A) and (B) in the introduction to this chapter.

7.1.3 Equational Logic, Term Re-Writing and Initial Truth

Recall our discussion and definitions from Chapter 2 concerning term re-writing and TRSs.

Equational Logic. Equational logic is the simplest and most fundamental fragment of first-order logic. As pointed out in Meinke and Tucker [1992] the importance of equational logic stems from the large number of interesting equational theories that naturally arise in mathematics and computer science. Moreover, with particular reference to (automated) verification equational logic is appealing for the following two reasons:

(1) The Correspondence Theorem shows that any deduction from an equational theory E using the four rules of equational logic: *reflexivity*, *symmetry*, *transitivity* and *substitution* may also be performed using a TRS constructed from E; that is, that equational logic and term re-writing are equivalent in their proof-theoretic power. As a consequence the implementation of equational logic may be achieved via the comparatively straightforward implementation of a term re-writing engine.

(2) Birkhoff's Soundness and Completeness Theorem for equational logic shows that we do not need a more powerful logic to reason about first-order equational theories. This fact follows as by the completeness of the equational calculus any equation that is provable by a more powerful logic must also be provable by purely equational means.

As we will show, the combination of these two facts with some of the previous results of this thesis provides the basis for a straightforward verification methodology for STs.

Indeed, we begin by outlining a 'naive' automated verification technique based on these ideas, and by highlighting its flaw motivate the construction of the calculus EQWIL in Section 7.2.

Using Term Re-Writing to Verify STs. We have shown that a very broad class of STs and SPSs can be specified using essentially nothing more than first-order equations. In particular, we have show that we can programme a ST in ASTRAL and can convert this programme into a complete, essentially first-order TRS. Therefore, given two ASTRAL programmes with one representing a ST specification and the other representing a corresponding implementation (possibly a SPS), if we can construct an equational correctness statement that relates these two programmes then in order to automatically verify this correctness statement it appears at first that we need only do the following:

(A) Initially we must implement the abstract compiler definitions that we have presented in the previous chapters thereby providing a mechanism for the automatic generation of a complete TRS from an ASTRAL programme. Indeed, in the sequel we will show formally that we may combine two ASTRAL programmes to produce a single TRS that captures the intended semantics of the STs represented by each programme.

(B) Having generated the required TRS we can use these re-write rules to reduce both sides of the correctness statement to normal forms. (Notice that by virtue of the fact that the TRS is complete these normal forms are guaranteed to exist and must be unique.)

(C) Finally, on comparing the normal forms of these terms (that represent symbolically the results of the computation performed by the two STs), if the normal forms are the same (syntactically identical) then we may conclude that the implementation is correct.

Initial Truth. While the verification method outlined above is both sound and complete, its completeness is with respect to truth in all models, that does not in general coincide with truth in the initial model. In more detail, as we have already mentioned in Section 3.10.4 the difficulty that we face with this method is that typically we require the verification of the correctness of any implementation of an ST to be in terms of its initial algebra semantics. Theorem 11 show us that the verification of an equational correctness statement that relates two ASTRAL programmes is decidable by virtue of the generation of a complete TRS. However, this decidability is with respect to the loose algebraic semantics (see Goguen [1988] and Goguen [1990]) that does not in general imply decidability with respect to the truth of the correctness statement in the initial model; that is, its *initial truth.* More specifically, validity (truth in all models) and initial truth coincide over closed equations (see Goguen and Meseguer [1982], MacQueen and Sanella [1985] and Heering [1986]), but if we consider open terms then in general the initial semantics of an equational theory and its loose semantics only coincide if the specification is *ω-complete.*

Unfortunately, the property of ω-completeness is only enjoyed by a small sub-set of equational theories (see Heering [1986]). In particular, primitive recursive equational specifications are *not* in general ω-complete, but the equational correctness statements relating ASTRAL programmes *do* in general require the use of open terms. As a consequence, while the verification method for STs that we have outlined is certainly useful, in general it is only appropriate with respect to loose and not initial semantics.

These limitations of equational logic are well-known, and therefore in order to address this problem researchers have considered the following question: is it possible to enrich equational logic with further proof rules to capture the initial semantics of an equational theory? In the following section we discuss two solutions to this problem, and later show how we may adapt one method as the basis of the software tools for the verification of STs that we require.

7.1.4 Equational Logic and Induction

While it is possible to enhance equational logic by the addition of further proof rules to give a sound and complete calculus that captures the initial semantics of an equational theory, unfortunately in general this requires the use of an infinitry deduction rule: the ω-rule (see for example Meinke and Tucker [1992]). We use the word 'unfortunately' again as the ω-rule requires an infinite number of premises to be discharged during a deduction and therefore is not suitable for implementation.

However, one technique that can be used to (partially) address this problem is to augment equational logic with induction. While this does not have the power of the ω-rule in the sense that it will allow us to capture the initial semantics of an arbitrary equational theory with a

sound and complete calculus, we can make use of the fact that equational logic plus induction is sound with respect to the initial semantics (see Goguen [1988] and Goguen [1990]). Moreover, what we will demonstrate in this chapter is that despite the fact that equational logic and induction cannot in general provide a complete deduction system with respect to initial semantics; we can identify a syntactic class of correctness statements relating ASTRAL programmes and their implementations for which equational logic and induction is a decidable calculus and hence provides the basis for a total proof tool.

7.2 Formalizing Primitive Recursive Arithmetic

In this section we present the underlying technical results of this chapter concerning a formalization of primitive recursive arithmetic over arbitrary first-order equations.

7.2.1 General Preliminaries

We begin the section with some basic definitions and some results from the literature that we will require. To conserve space where either a result is straightforward or the result is well-known we will not give a proof, but will include a reference from which the result is taken or in which the interested reader can find a suitable proof.

Notation 4. We make the usual assumption that Σ, Σ' and Σ'' are any non-void S-sorted, S'-sorted and S''-sorted signatures respectively and that X, X' and X'' are any S-indexed S'-indexed and S''-indexed collection of variable symbols respectively such that Σ and X, Σ' and X' and Σ'' and X'' are pair-wise disjoint. In addition, the symbols A, A' and A'' always denote any S-sorted Σ-algebra, any S'-sorted Σ'-algebra and any S''-sorted Σ''-algebra respectively.

Recall the definition of terms from Section 2.3.9. In the sequel in order to make our definitions effective we will need to place an ordering on the sub-terms of a term that share a particular property of interest. More specifically, we will need to identify the *left-most sub-term* of a term that has some property of interest. This idea is made more rigorous in the following definition.

Definition 93. Let $P \subseteq T(\Sigma, X)$ and let $\tau \in T(\Sigma, X)$. We define the *left-most sub-term of τ satisfying P* by induction on on the structural complexity of τ as follows:

If either $\tau = c \in \Sigma_{\lambda,s}$ or $\tau = x \in X_s$, for some $s \in S$ then if $P(\tau)$ then: τ is the left-most sub-term of τ satisfying P; otherwise τ has no left-most sub-term satisfying P.

If $\tau = \sigma(\tau_1, \ldots, \tau_{|w|})$, for some $\sigma \in \Sigma_{w,s}$, for some $w \in S^+$ and for some $s \in S$; and for some $\tau_i \in T(\Sigma, X)_w$, for $i = 1, \ldots, |w|$ then: if $P(\tau)$ then τ is the left-most sub-term of τ satisfying P; otherwise if $P(\tau_i)$ for some $i \in \{1, \ldots, |w|\}$ then the left-most sub-term of τ satisfying P is the smallest value $i \in \{1, \ldots, |w|\}$ such that $P(\tau_i)$; otherwise τ has no left-most sub-term satisfying P.

For example, if $\tau = f(0, x, 1, y)$ then x is the *left most variable of* τ.

Lemma 34. (see Meinke and Tucker [1992].) *Let $E \subseteq EQ(\Sigma, X)$ be any system of equations. If $e \in EQ(\Sigma)$ then*

$$Alg(\Sigma, E) \models e \iff I(\Sigma, E) \models e.$$

Lemma 35. *Let $E \subseteq EQ(\Sigma, X)$ be any system of equations and let R be any complete TRS equivalent to E. If $e = (\tau = \tau') \in EQ(\Sigma, X)$ then*

$$(NF^R(\tau) = NF^R(\tau')) \Longleftrightarrow Alg(\Sigma, E) \models e.$$

In particular, notice that by Lemma 34 if $e \in EQ(\Sigma)$ then

$$(NF^R(\tau) = NF^R(\tau')) \Longleftrightarrow I(\Sigma, E) \models e.$$

Lemma 36. (The Constants Lemma – Goguen [1987].) *Let $E \subseteq EQ(\Sigma, X)$ and let χ be some constant of sort s such that $\chi \notin \Sigma$. Also let $\Sigma' = \Sigma \cup \{\chi\}$ and let $E' \subseteq EQ(\Sigma', X)$ be defined such that $E' = E$. If $(\tau = \tau') \in EQ(\Sigma', X)_{s'}$ for some $s' \in S$ then*

$$E' \vdash (\tau = \tau') \Longleftrightarrow E \vdash (\tau[\chi/x] = \tau'[\chi/x])$$

wherein $x \in X_s$ is some variable that does not occur in τ or τ'.

Lemma 37. (Bergstra and Tucker [1987].) *Let $\Sigma \subseteq \Sigma'$, let $E \subseteq EQ(\Sigma, X)$ and $E' \subseteq EQ(\Sigma', X)$ be defined such that $E \subseteq E'$. If $A \cong I(\Sigma, E)$, $A' \in Alg(\Sigma', E')$, $A'|_\Sigma \cong A$ and $I(\Sigma', E')|_\Sigma \cong I(\Sigma, E)$ then $A' \cong I(\Sigma', E')$.*

7.2.2 Signatures of Constructors

We introduce the idea of a signature that may be used to finitely generate (inductively generate) a representative of each member of the carrier of an algebra. The reader can consult Meinke and Tucker [1992] and Goguen [1987] for more details.

Definition 94. Let A be any algebra such that $A_s \leq |\mathbb{N}|$ for each $s \in S$. Also let $\Gamma \subseteq \Sigma$ be defined for each $s \in S$ as follows: If $|A_s| = n_s$ for some $n_s \in \mathbb{N}^+$ and $A_s = \{a_{s,1}, \ldots, a_{s,n_s}\}$ then $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ wherein $c_{s,i} \in \Sigma_{\lambda,s}$ for $i = 1, \ldots, n_s \in \mathbb{N}^+$ is some constant such that $c^A_{s,i} = a_{s,i}$; otherwise if $|A_s| = |\mathbb{N}|$ and $A_s = \{a_{s,0}, a_{s,1}, a_{s,2}, \ldots\}$ then $\Gamma_s = \{\{b_s\}, \{g_s\}\}$ wherein $b_s \in \Sigma_{\lambda,s}$ and $g_s \in \Sigma_{s,s}$ are some constant and some unary operation respectively such that $b^A_s = a_{s,0}$ and the function $g^i_s : A_s \to A_s$ for each $i \in \mathbb{N}^+$ is defined by $g^i_s(b^A_s) = a_{s,i}$ wherein

$$g^i_s(b^A_s) = \begin{cases} g^A_s(b^A_s) & \text{if } i = 1, \text{ and} \\ g^A_s(g^{i-1}_s(b^A_s)) & \text{otherwise.} \end{cases}$$

If there exists a Γ defined as above then we either say that Γ is *inductive* for A or that Γ is a *signature of constructors for* A or just a *signature of constructors* if A is understood or unimportant.

Example 19. If A is a standard algebra with no carriers other than $\mathbb{N}$ and $\mathbb{B}$ then $\Gamma = \{\Gamma_n = \{\{0\}, \{Succ\}\}, \Gamma_b = \{tt, ff\}\}$ is inductive for A.

Note that in the sequel for convenience we will we write $\Gamma_{\mathbf{n}} = \{0, Succ\}$ to mean $\Gamma_{\mathbf{n}} = \{\{0\}, \{Succ\}\}$ and more generally $\Gamma_s = \{b_s, g_s\}$ to mean $\Gamma_{\mathbf{n}} = \{\{b_s\}, \{g_s\}\}$.

Discussion. We note that this is not the most general definition of a signature of constructors found in the literature (see Goguen [1988]). However, in the context of primitive recursive equations, in particular primitive recursive equations when used to specify hardware, we find it convenient to use this more restrictive definition. Moreover, the results that we present easily generalize to the use of a more general signature of constructors for countably infinite carriers.

It is also important to note that the initial model of every equational theory with either finite or countably infinite carriers has at least one minimal and finite signature of constructors (see Goguen [1988]) and hence inductively generating the members of the carriers of an algebra is a general purpose technique.

Finally, in the sequel we will need to generalize the use of signatures of constructors to algebras wherein some carriers are not countable (stream carriers are not in general countable). In particular, if an algebra has an uncoutable carrier then we will use the convention that for each $s \in S$ such that $|A_s| > |\mathbb{N}|$ we have $\Gamma_s = \emptyset$. Thus, for any stream algebra $\underline{A}$ we have $\Gamma_{\underline{s}} = \emptyset$ for each $s \in S$.

7.2.3 Using Signatures of Constructors to Identify Classes of Equations

We now identify two classes of equations $\mathrm{CEorFC} \subseteq \mathrm{EQ}(\Sigma, X)$ and $\mathrm{ComTRS} \subseteq \mathrm{TRS}(\Sigma, X) \times \mathrm{EQ}(\Sigma, X)$ that will play a significant role in the sequel. The first class is the union of all closed equations extended with equations whose variables range over finite carriers. The second class is the same as the first class, but also extended with equations that when orientated as left-to-right re-write rules form a complete TRS R' when adjoined to some given TRS R.

Definition 95. For each Σ, for each $\Gamma \subseteq \Sigma$ and for each X we define the predicate

$$\mathrm{CEorFC}^{\Sigma,\Gamma,X} \subseteq \mathrm{EQ}(\Sigma, X)$$

(ambiguously denoted CEorFC) as follows: for each $e \in \mathrm{EQ}(\Sigma, X)$ the predicate $\mathrm{CEorFC}(e)$ holds if and only if either

(1) $e \in \mathrm{EQ}(\Sigma)$; that is, e is a closed equation; or

(2) $e \in \mathrm{EQ}(\Sigma, \{x_1, \ldots, x_n\})$ for some $x_i \in X_{s_i}$, for some $s_i \in S$ such that for each $i = 1, \ldots, n \in \mathbb{N}^+$ we have $\Gamma_{s_i} = \{c_{s_i,1}, \ldots, c_{s_i,n_{s_i}}\}$ for some constants $c_{s_i,j} \in \Sigma_{\lambda,s_i}$ for $j = 1, \ldots, n_{s_i} \in \mathbb{N}^+$; that is, e is defined over variables that range over finite carriers.

Definition 96. For each Σ, for each $\Gamma \subseteq \Sigma$ and for each X we define the predicate

$$\mathrm{ComTRS}^{\Sigma,\Gamma,X} \subseteq \mathrm{TRS}(\Sigma, X) \times \mathrm{EQ}(\Sigma, X)$$

(ambiguously denoted ComTRS) as follows: for each $R \subseteq \mathrm{TRS}(\Sigma, X)$ and for each $e = (\tau = \tau') \in \mathrm{EQ}(\Sigma, X)$ the predicate $\mathrm{ComTRS}(R, e)$ holds if and only if either

(1) CEorFC(e) holds; or

(2) $e = (\tau = \tau') \in \mathrm{EQ}(\Sigma, \{x_1, \ldots, x_n\})$, for some $x_i \in X_{s_i}$, for some $s_i \in S$ for $i = 1, \ldots, n \in \mathbb{N}^+$ such that x_1 is the left-most variable of τ and $\Gamma_{s_1} = \{b_{s_1}, g_{s_1}\}$ for some $b_{s_1} \in \Sigma_{\lambda, s_1}$ and for some $g_{s_1} \in \Sigma_{s_1, s_1}$; and the TRS R' defined by

$$R' = R \cup \{r'\}$$

is complete wherein r' is formed by orienting e' as a left-to-right re-write rule; and e' is any equation defined by

$$e' = e[x_1/\chi][x_i/\rho_i]_{i=2}^{i=n}$$

wherein χ is some constant of sort s_1 such that $\chi \notin \Sigma$ and $\rho_i \in T(\Sigma'', X)_{s_i}$ for some $\Sigma'' \subseteq \Sigma \cup \{\chi\}$ are defined such that $\rho_i \not\trianglerighteq \chi$ for $i = 1, \ldots, n$.

Example 20. Let Σ, Γ and A be defined as in Example 19 and let χ be some constant of type $\mathbf{n}$ such that $\chi \notin \Sigma$. If $t_{\mathbf{n}}$ is some closed term of sort $\mathbf{n}$ and $t_{\mathbf{b}}$ is some closed term of sort $\mathbf{b}$ then

(1) CEorFC($not(or(x, y)) = t_{\mathbf{b}}$) holds because both x and y are of type $\mathbf{b}$ and $\Gamma_{\mathbf{b}} = \{tt, ff\}$.

(2) CEorFC($add(x, 0) = x$) does not hold because x is of type $\mathbf{n}$ and $\Gamma_{\mathbf{n}} = \{0, Succ\}$. However, if the TRS R is defined by $R = \{add(0, x) \mapsto x, add(Succ(x), y) \mapsto Succ(add(x, y))\}$ then ComTRS($R, add(x, 0) = x$) holds because $R' = R \cup \{add(\chi, 0) \mapsto \chi\}$ is complete.

We now present two key results concerning the predicate ComTRS and sets of primitive recursive equations. Indeed, while for convenience we present them in 'reverse' order, the first result is the basis of our proof of Statement (B) in the introduction. The second result is the basis of our proof of Statement (A).

Lemma 38. *Let $\Gamma \subseteq \Sigma$ such that $\Gamma_{\mathbf{n}} = \{0, Succ\}$. Also, let $\Psi \in PREQ(\underline{\Sigma}, \underline{X})$, let $R = TRCON(\Psi) \subseteq TRS(\underline{\Sigma'}, \underline{X})$ wherein Σ' is as defined in Definition 66. If f is one of the functions symbols of type $(\mathbf{n}\, u, s)$ appearing in R, for some $u \in \underline{S}^+$ and for some $s \in S$; and $c \in \Sigma_{\lambda, s}$ then*

$$ComTRS(R, f(x, t_1, \ldots, t_{|u|}) = c)$$

holds for any variable $x \in X_{\mathbf{n}}$ and for any terms $t_i \in T(\Sigma'', X)_{u_i}$ such that $t_i \not\trianglerighteq \chi$ for $i = 1, \ldots, |u|$ wherein $\Sigma'' \supseteq \Sigma' \cup \{\chi\}$.

Lemma 39. *Let $\Gamma \subseteq \Sigma$ such that $\Gamma_{\mathbf{n}} = \{0, Succ\}$. Also let $\Psi \in PREQ(\underline{\Sigma}, \underline{X})$, let $R = TRCON(\Psi) \subseteq TRS(\underline{\Sigma'}, \underline{X})$ wherein Σ' is defined as in Definition 66. If f and g are two of the functions symbols of type $(\mathbf{n}\, u, s)$ and $(\mathbf{n}\, u', s)$ respectively for some $u, u' \in \underline{S}^+$ and for some $s \in S$ appearing in R such that $f \neq g$ then*

$$ComTRS(R, f(x, t_1, \ldots, t_{|u|}) = g(x, t'_1, \ldots, t'_{|u'|}))$$

holds for any variable $x \in X_{\mathbf{n}}$ and for any terms $t_i \in T(\Sigma'', X)_{u_i}$ such that $t_i \not\trianglerighteq \chi$ for $i = 1, \ldots, |u|$ and for any terms $t'_j \in T(\Sigma'', X)_{u'_j}$ such that $t'_i \not\trianglerighteq \chi$ for $j = 1, \ldots, |u'|$ wherein $\Sigma'' \supseteq \Sigma' \cup \{\chi\}$.

We prove Lemma 39 using the following result from Knuth and Bendix [1970] and leave the similar proof of Lemma 38 to the reader.

Theorem 12. *Let R be any strongly normalizing TRS. If all the critical pairs of R are convergent then R is complete.*

Proof of Lemma 39. Notice that by hypothesis the left-most variable x of $f(x, t_1, \ldots, t_{|u|})$ is of type $\mathbf{n}$ and $\Gamma_{\mathbf{n}} = \{0, Succ\}$. Therefore we must show that

$$R' = R \cup \{r'\}$$

wherein

$$r' = (f(\chi, \tau_1, \ldots, \tau_{|u|}) \mapsto g(\chi, \tau'_1, \ldots, \tau'_{|u'|}))$$

is complete for any terms $t_i \in T(\Sigma'', X)_{u_i}$ such that $t_i \not\trianglerighteq \chi$ for $i = 1, \ldots, |u|$ and for any terms $t'_j \in T(\Sigma'', X)_{u'_j}$ such that $t'_i \not\trianglerighteq \chi$ for $j = 1, \ldots, |u'|$ wherein $\Sigma'' \supseteq \Sigma' \cup \{\chi\}$.

First, notice that by the definition of TRCON the only rules in R of the form

$$f(\kappa_1, \ldots, \kappa_{|u|+1}) \mapsto \rho$$

and

$$g(\kappa'_1, \ldots, \kappa'_{|u'|+1}) \mapsto \rho'$$

for some $\kappa_i \in T(\underline{\Sigma}', \underline{X})$ for $i = 1, \ldots, |u| + 1$ and for some $\rho \in T(\underline{\Sigma}', \underline{X})$; and for some $\kappa'_j \in T(\underline{\Sigma}', \underline{X})$ for $j = 1, \ldots, |u'| + 1$ and for some $\rho' \in T(\underline{\Sigma}', \underline{X})$ must be of the form

$$r_1 = f(0, x_1, \ldots, x_{|u|}) \mapsto \eta_1$$

and

$$r_2 = f(Succ(x), x_1, \ldots, x_{|u|}) \mapsto \eta_2$$

and

$$r'_1 = f(0, x'_1, \ldots, x'_{|u'|}) \mapsto \eta'_1$$

and

$$r'_2 = f(Succ(x), x'_1, \ldots, x'_{|u'|}) \mapsto \eta'_2$$

for some variables $x_i \in \underline{X}_{u_i}$ for $i = 1, \ldots, |u|$, for some variables $x'_j \in \underline{X}_{u_j}$ for $j = 1, \ldots, |u'|$, and for some terms $\eta_1, \eta_2, \eta'_1, \eta'_2 \in T(\underline{\Sigma}', \underline{X})$. Notice now that R is complete by Theorem 11 and so it is by definition strongly normalizing. As a consequence clearly R' is strongly normalizing as $g(\chi, \rho_1, \ldots, \rho_{|u'|})$ is a normal form under R' for any $\rho_i \in T(\underline{\Sigma''}, \underline{X})$ for $i = 1, \ldots, |u'|$. Therefore as we cannot by definition make a critical pair with divergent normal forms from any of the rules in R (as R is complete) the only potential critical pairs are either r_1 and r' or r_2 and r'. Hence, as by observation it is clear that we cannot make a critical pair from either r_1 and r' or r_2 and r' R' is complete by Theorem 12.

□

Systems of primitive recursive equations also have one further useful property with respect to particular terms:

Lemma 40. *Let* $\Gamma \subseteq \Sigma$ *such that* $\Gamma_{\mathbf{n}} = \{0, Succ\}$. *Also let* $\Psi \in PREQ(\underline{\Sigma}, \underline{X})$, *let* $R = TRCON(\Psi) \subseteq TRS(\underline{\Sigma'}, \underline{X})$ *wherein* Σ' *is as defined in Definition 66. If* f *and* g *are two of the functions symbols of type* $(\mathbf{n}\, u, s)$ *and* $(\mathbf{n}\, u', s)$ *respectively for some* $u, u' \in \underline{S}^+$ *and for some* $s \in S$ *appearing in* R *such that* $f \neq g$ *then*

$$R' = R \cup \{f(0, t_1, \ldots, t_{|u|}) \mapsto g(0, t'_1, \ldots, t'_{|u'|})\}$$

is strongly normalizing for any terms $t_i \in T(\underline{\Sigma''}, \underline{X})_{u_i}$ *for* $i = 1, \ldots, |u|$ *and for any terms* $t'_j \in T(\underline{\Sigma''}, \underline{X})_{u'_j}$ *for* $j = 1, \ldots, |u'|$ *wherein* $\Sigma'' \supseteq \Sigma'$.

Proof. Similar to the proof of Lemma 39

□

7.2.4 Formalizing Primitive Recursive Arithmetic

We now define a formal calculus that extends equational logic with additional rules for induction and cases analysis. In the context of systems of equations that are primitive recursive this calculus is essentially a formalization of primitive recursive arithmetic and, as far as we are aware, in the context of reasoning about STs is new.

Definition 97. For each $\Gamma \subseteq \Sigma$ and for each $E \subseteq \mathrm{EQ}(\Sigma, X)$ we define the calculus $\mathrm{EQWIL}^{\Gamma,E}$ (*EQ*uational calculus *W*ith *I*nductive *L*ogic – pronounced 'equal') uniformly in E to be the four rules of many-sorted equational logic over E:

Rule (i) – Reflexivity. If $\tau \in T(\Sigma, X)_s$ for some $s \in S$ then

$$\frac{}{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau = \tau}$$

Rule (ii) – Symmetry. If $\tau, \tau' \in T(\Sigma, X)_s$ for some $s \in S$ then

$$\frac{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau = \tau'}{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau' = \tau}$$

Rule (iii) – Transitivity. If $\tau, \tau', \tau'' \in T(\Sigma, X)_s$ for some $s \in S$ then

$$\frac{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau = \tau' \quad \mathrm{EQWIL}^{\Gamma,E} \vdash \tau' = \tau''}{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau = \tau''}$$

Rule (iv) – Substitution. For any terms $\tau, \tau' \in T(\Sigma, X)_s$ and $t, t' \in T(\Sigma, X)_{s'}$ for some $s, s' \in S$ if $y \subseteq \tau$ or $y \subseteq \tau'$ for some $y \in X_{s'}$ then

$$\frac{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau = \tau' \quad \mathrm{EQWIL}^{\Gamma,E} \vdash t = t'}{\mathrm{EQWIL}^{\Gamma,E} \vdash \tau[y/t] = \tau'[y/t']}$$

extended with the following two additional rules:

Rule (v) – Case Analysis. For any terms $\tau, \tau' \in T(\Sigma, X)_s$ for some $s \in S$ if $y \subseteq \tau$ or $y \subseteq \tau'$ for some variable $y \in X_{s'}$ for some $s' \in S$; and $\Gamma_{s'} = \{c_1, \ldots, c_{n_{s'}}\}$ for some constants $c_i \in \Sigma_{\lambda,s'}$ for $i = 1, \ldots, n_{s'} \in \mathbb{N}^+$ then

$$\frac{\text{EQWIL}^{\Gamma,E} \vdash \tau[y/c_{s',1}] = \tau'[y/c_{s',1}] \cdots \text{EQWIL}^{\Gamma,E} \vdash \tau[y/c_{s',n_{s'}}] = \tau'[y/c_{s',n_{s'}}]}{\text{EQWIL}^{\Gamma,E} \vdash \tau = \tau'.}$$

Rule (vi) – Induction. For any terms $\tau, \tau' \in T(\Sigma, X)_s$ for some $s \in S$ if $y \subseteq \tau$ or $y \subseteq \tau'$ for some variable $y \in X_{s'}$ for some $s' \in S$ and $\Gamma_{s'} = \{b_{s'}, g_{s'}\}$ for some $b_{s'} \in \Sigma_{\lambda,s'}$ and for some $g_{s'} \in \Sigma_{s',s'}$ then

$$\frac{\text{EQWIL}^{\Gamma,E} \vdash \tau[y/b_{s'}] = \tau'[y/b_{s'}] \quad \text{EQWIL}^{\Gamma,E'} \vdash \tau[y/g_{s'}(\chi)] = \tau'[y/g_{s'}(\chi)]}{\text{EQWIL}^{\Gamma,E} \vdash \tau = \tau'}$$

wherein

$$E' = E \cup \{\tau[y/\chi] = \tau'[y/\chi]\} \in \text{EQ}(\Sigma', X),$$

χ is some constant of sort s' such that $\chi \notin \Sigma$, and $\Sigma' = \Sigma \cup \{\chi\}$.

We conclude this section with some basic definitions and three basic results about the structure of proofs in EQWIL that we will require in the sequel.

Lemma 41. *If $\Gamma \subseteq \Sigma$ and $E \subseteq EQ(\Sigma, X)$ and $e \in EQ(\Sigma)$ then*

$$EQWIL^{\Gamma,E} \vdash e \iff E \vdash e.$$

Definition 98. Let $\Gamma \subseteq \Sigma$, let $E \subseteq \text{EQ}(\Sigma, X)$ and let $e, e' \in \text{EQ}(\Sigma, X)$.

(1) Let $\text{EQWIL}^{\Gamma,E} \vdash e$. We say that e' is a *sub-consequence of* e if e' is one of the premises of one of the applications of Rules (i) through (vi) used at any stage in the proof used to deduce e.

For example, if P was the following proof

$$\frac{\text{EQWIL}^{\Gamma,E} \vdash And(tt, ff) = ff \quad \text{EQWIL}^{\Gamma,E} \vdash And(ff, ff) = ff}{\text{EQWIL}^{\Gamma,E} \vdash And(x, ff) = ff}$$

from some appropriate system of equations E then both $And(tt, ff) = ff$ and $And(ff, ff) = ff$ are sub-consequences of $And(x, ff) = ff$.

(2) Let P be the proof $\text{EQWIL}^{\Gamma,E} \vdash e$ and e' be a sub-consequence of e. We say that P' is a *sub-proof of* P if P' is the part of P used to deduce e'; that is, if P' is the proof $\text{EQWIL}^{\Gamma,E'} \vdash e'$ wherein E' is either E or some system of equations such that $E' \supset E$ as defined in Rule (vi).

(3) Let P be the proof $\text{EQWIL}^{\Gamma,E} \vdash e$. If there exists a sub-consequence e' of e with corresponding sub-proof P' such that $E \vdash e'$, but P' includes either an application of Rule (v) or an application of Rule (vi) then we say that P has a *trivial deduction*. Thus, P has a trivial deduction if e' is provable by using equational logic, but P' has used either case analysis or induction to prove e'.

Lemma 42. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$. If P is the proof $EQWIL^{\Gamma,E} \vdash e$ and P has trivial deductions then there exists a proof P' of $EQWIL^{\Gamma,E} \vdash e$ with no trivial deductions.*

Lemma 43. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$. If P is the proof $EQWIL^{\Gamma,E} \vdash e$ and P contains k non-trivial deductions then there exists a proof P' of $EQWIL^{\Gamma,E} \vdash e$ with these k non-trivial deduction as the last k steps of P'.*

Proof. By induction on the number k with a sub-induction in the basis case ($k = 1$) on the number of rules l applied after the application of either Rule (v) or Rule (vi). In turn the sub-induction basis case ($l = 1$) requires a case analysis on the last rule of P: symmetry; transitivity and substitution.

□

7.3 The Soundness, Completeness and Decidability of EQWIL

7.3.1 Soundness and Completeness

We begin with three results concerning the soundness and completeness of EQWIL. However, notice that as a consequence of the limitations of any formal calculus with respect to initial algebra semantics, this completeness result is only concerned with closed equations and equations whose variables range over finite carriers.

Theorem 13. (EQWIL Soundness.) *Let A be any algebra such that there exists $\Gamma \subseteq \Sigma$ that is inductive for A and let $E \subseteq EQ(\Sigma, X)$. If $A \cong I(\Sigma, E)$ then for any $e \in EQ(\Sigma, X)$, for some $s \in S$*

$$EQWIL^{\Gamma,E} \vdash e \Longrightarrow I(\Sigma, E) \models e.$$

Proof. While equational logic and induction is not typically presented as a formal calculus, the fact that equational logic with induction is sound with respect to the initial algebra semantics of an equational theory is well-known (see for example Goguen [1988]). Indeed, the proof of soundness is straightforward making use of Lemma 36 and therefore is omitted.

□

Theorem 14. (EQWIL Completeness.) *Let A be any algebra such that there exists $\Gamma \subseteq \Sigma$ that is inductive for A and let $E \subseteq EQ(\Sigma, X)$. If $A \cong I(\Sigma, E)$ then for any $e \in EQ(\Sigma, X)$, for some $s \in S$ such that $CEorFC(e)$ holds*

$$I(\Sigma, E) \models e \Longrightarrow EQWIL^{\Gamma,E} \vdash e.$$

Proof. We proceed by induction on the number of variables $k \in \mathbf{N}$ appearing in e.
Basis $n = 0$. First, notice that by hypothesis we have $e \in \mathrm{EQ}(\Sigma)$ and hence

$$\mathrm{I}(\Sigma, E) \models e \Longrightarrow \mathrm{Alg}(\Sigma, E) \models e$$

by Lemma 34. Also notice that by the completeness of equational logic we have

$$\mathrm{Alg}(\Sigma, E) \models e \Longrightarrow E \vdash e.$$

Therefore by Lemma 41 we have

$$E \vdash e \Longrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e$$

as required.

Induction Hypothesis. Let $\Gamma \subseteq \Sigma$. Assume for any system of equations $E \subseteq \mathrm{EQ}(\Sigma, X)$ and for any equation $e' \in \mathrm{EQ}(\Sigma, X)_{s'}$ for some $s' \in S'$ such that e' is defined over k' variables for some fixed $k' \in \mathbb{N}$; and such that $\mathrm{CEorFC}(e')$ holds then

$$\mathrm{I}(\Sigma, E) \models e' \Longrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e'.$$

Induction Step. Let $\Gamma \subseteq \Sigma$. We must show that for any system of equations $E \subseteq \mathrm{EQ}(\Sigma, X)$ and for any equation $e'' \in \mathrm{EQ}(\Sigma, X)_{s''}$ for some $s'' \in S''$ such that e'' is defined over $k' + 1$ variables; and such that $\mathrm{CEorFC}(e'')$ holds then

$$\mathrm{I}(\Sigma, E) \models e'' \Longrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e''.$$

We proceed as follows: choose any variable $y = x_i \in \{x_1, \ldots, x_{k'+1}\}$ wherein $x_j \in X_{s_j}$ for some $s_j \in S''$ for $j = 1, \ldots, k' + 1$ are the variables over which e'' is defined. Notice that as by hypothesis $\mathrm{CEorFC}(e'')$ holds it must be the case that $\Gamma_{s_i} = \{c_{s_i,1}, \ldots, c_{s_i,n_{s_i}}\}$ for some $n_{s_i} \in \mathbb{N}^+$.

Now consider the equations $e_1, \ldots, e_{n_{s_i}}$ defined by

$$e_l = e''[y/c_{s_i,l}]$$

for $l = 1, \ldots, n_{s_i}$. First, notice that clearly by the hypothesis on Γ we have

$$\mathrm{I}(\Sigma, E) \models e'' \Longrightarrow \mathrm{I}(\Sigma, E) \models e_l$$

for $l = 1, \ldots, n_{s_i}$. Secondly, notice that as each equation e_l is defined over k' variables and $\mathrm{CEorFC}(e_l)$ holds by the Induction Hypothesis with $e' = e_l$ for $l = 1, \ldots, n_{s_i}$ that

$$\mathrm{I}(\Sigma, E) \models e_l \Longrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e_l.$$

Therefore, combining these two facts with Rule (v) of $\mathrm{EQWIL}^{\Gamma,E}$ we have that

$$\mathrm{I}(\Sigma, E) \models e'' \Longrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e''$$

as required.

□

Corollary 3. *Let A be any algebra such that there exists $\Gamma \subseteq \Sigma$ that is inductive for A and let $E \subseteq EQ(\Sigma, X)$. If $A \cong I(\Sigma, E)$ then for any $e \in EQ(\Sigma, X)_s$ for some $s \in S$ such that $CEorFC(e)$ holds*

$$I(\Sigma, E) \models e \Longleftrightarrow EQWIL^{\Gamma,E} \vdash e.$$

Proof. Immediate from Theorem 13 and from Theorem 14.

□

7.3.2 Decidability

Recall the four classes of equations that we identified in the introduction. We now consider the automation of the deductions necessary to prove such equations. In particular, in the context of strictly first-order systems of primitive recursive equations we define a function VER that we can use to automate the deduction of such equations using the calculus EQWIL.

Definition 99. For each Σ, for each $\Gamma \subseteq \Sigma$, for each X and for each $s \in S$ we define

$$\mathrm{VER}_s^{\Sigma,\Gamma,X} : \mathrm{TRS}(\Sigma, X) \times T(\Sigma, X)_s \times T(\Sigma, X)_s \rightsquigarrow \mathbb{B}$$

(ambiguously denoted VER^Γ) as follows:

$$(\forall R \in \mathrm{TRS}(\Sigma, X))(\forall \tau, \tau' \in T(\Sigma, X)_s)$$

$$\mathrm{VER}^\Gamma(R, \tau, \tau') = \begin{cases} \mathrm{RED}(R, \tau, \tau') & \text{if } \tau, \tau' \in T(\Sigma) \\ \bigwedge_{i=1}^{i=n_{s_1}} \mathrm{VER}^\Gamma_{s_1}(R, \tau[x_1/c_{s_1,i}], \tau'[x_1/c_{s_1,i}]) & \text{if } \tau, \tau' \in T(\Sigma, \{x_1, \ldots, x_n\})_s \\ & \text{and } \Gamma_{s_1} = \{c_{s_1,1}, \ldots, c_{s_1,n_{s_1}}\}; \text{ and} \\ B_1 \wedge B_2 & \text{if } \tau, \tau' \in T(\Sigma, \{x_1, \ldots, x_n\})_s \\ & \text{and } \Gamma_{s_1} = \{b_{s_1}, g_{s_1}\} \end{cases}$$

wherein

$$B_1 = \mathrm{VER}^\Gamma_{s_1}(R, \tau[x_1/b_{s_1}], \tau'[x_1/b_{s_1}])$$

and

$$B_2 = \mathrm{VER}^\Gamma_{s_1}(R', \tau[x_1/g_{s_1}(\chi)], \tau'[x_1/g_{s_1}(\chi)]),$$

$x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, n \in \mathbb{N}^+$ are the variables over which τ and τ' are defined such that either x_1 is the left-most variable of τ if $\tau \in T(\Sigma, X)$ or x_1 is the left-most variable of τ' otherwise; χ is some constant of type s_1 such that $\chi \notin \Sigma$; $R' = R \cup \{\tau[x_1/\chi] \mapsto \tau'[x_1/\chi]\}$; and

$$\mathrm{RED}_s^{\Sigma,X} : \mathrm{TRS}(\Sigma, X) \times T(\Sigma)_s \times T(\Sigma)_s \rightsquigarrow \mathbb{B}$$

(ambiguously denoted RED) is defined for each $R \subseteq \mathrm{TRS}(\Sigma, X)$ and for each $t, t' \in T(\Sigma)_s$ by

$$\mathrm{RED}(R, t, t') = \begin{cases} tt & \text{If } \mathrm{NF}^R(t) = \mathrm{NF}^R(t'), \\ ff & \text{otherwise.} \end{cases}$$

The following lemma is implicit in our use of VER in the sequel.

Lemma 44. *Let $\Gamma \subseteq \Sigma$ and let $t, t' \in T(\Sigma, X)$. If $R \subseteq TRS(\Sigma, X)$ is complete then*

$$RED(R, t, t') \downarrow .$$

Effectively Simulating Deductions in EQWIL. We now present a number of results concerning the use of VER to simulate deduction using EQWIL.

Definition 100. Let $\Gamma \subseteq \Sigma$ and let $R \subseteq \mathrm{TRS}(\Sigma, X)$. If $e = (\tau = \tau') \in \mathrm{EQ}(\Sigma, X)$ and

$$\mathrm{VER}^{\Gamma}(R, \tau, \tau') \downarrow$$

the we say that either e is *weakly decidable with respect to* R or just *weakly decidable* if R is either understood or unimportant.

Lemma 45. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$. Also let $R \subseteq TRS(\Sigma, X)$ be equivalent to E. If $e = (\tau = \tau') \in EQ(\Sigma, X)$ then*

$$VER^{\Gamma}(R, \tau, \tau') = tt \Longrightarrow EQWIL^{\Gamma, E} \vdash e.$$

Proof. Obvious from the definition of VER.

□

Lemma 46. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$ such that there exists a complete TRS R equivalent to E. If $e = (\tau = \tau') \in EQ(\Sigma)$ then*

$$EQWIL^{\Gamma, E} \vdash e \Longleftrightarrow VER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. Notice that as $e \in \mathrm{EQ}(\Sigma)$ we have

$$\mathrm{EQWIL}^{\Gamma, E} \vdash e \Longleftrightarrow E \vdash e$$

by Lemma 41. Furthermore,

$$\mathrm{VER}^{\Gamma}(R, \tau, \tau') = \mathrm{RED}(R, \tau, \tau') = \begin{cases} tt & \text{if } \mathrm{NF}^{R}(\tau) = \mathrm{NF}^{R}(\tau'), \text{ and} \\ ff & \text{otherwise.} \end{cases}$$

Therefore, it is clear that the lemma holds by the Correspondence Theorem and the hypothesis that R is complete.

□

Lemma 47. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$ such that there exists a complete TRS R equivalent to E. If $e = (\tau = \tau') \in EQ(\Sigma, X)$ is defined such that CEorFC(e) holds then*

$$EQWIL^{\Gamma, E} \vdash e \Longleftrightarrow VER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. By induction on the number of variables $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, k \in \mathbb{N}$ appearing in e. Notice in the following proof that by Lemma 43 we may assume, without loss of generality, that an application of Rule (v) in a proof $\mathrm{EQWIL}^{\Gamma,E} \vdash e$ occurs as the last proof step.

Basis $k = 0$. This case follows immediately by Lemma 46.

Induction Hypothesis. If $e' = (t = t') \in \mathrm{EQ}(\Sigma, X)$ is some equation such that $\mathrm{CEorFC}(e')$ holds and e' is defined over k' variables for some fixed $k' \in \mathbb{N}$ then

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e' \iff \mathrm{VER}^{\Gamma}(R, t, t') = tt.$$

Induction Step. We must show that if $e'' = (r = r') \in \mathrm{EQ}(\Sigma, X)$ is some equation such that $\mathrm{CEorFC}(e'')$ holds and e'' is defined over $k = k' + 1$ variables then

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e'' \iff \mathrm{VER}^{\Gamma}(R, r, r') = tt.$$

First, notice that as by hypothesis $\mathrm{CEorFC}(e'')$ holds in this case $\mathrm{VER}^{\Gamma}(R, r, r')$ is defined by

$$\mathrm{VER}^{\Gamma}(R, r, r') = \bigwedge_{j=1}^{j=n_{s_1}} \mathrm{VER}^{\Gamma}(R, r[x_1/c_{s_1,j}], r'[x_1/c_{s_1,j}])$$

wherein $x_1 \in X_{s_1}$ for some $s_1 \in S$ is either the left-most variable of r if $r \in T(\Sigma, X)$ or x_1 is the left-most variable of r' otherwise. Also notice that as $\mathrm{CEorFC}(e'') \Longrightarrow \mathrm{CEorFC}(r[x_1/c_{s_1,j}] = r'[x_1/c_{s_1,j}])$ for $j = 1, \ldots, n_{s_1}$ and the number of variables occurring in $r[x_1/c_{s_1,j}] = r'[x_1/c_{s_1,j}]$ is k', by the Induction Hypothesis we have

$$\mathrm{EQWIL}^{\Gamma,E} \vdash r[x_1/c_{s_1,j}] = r'[x_1/c_{s_1,j}] \iff \mathrm{VER}^{\Gamma}(R, r[x_1/c_{s_1,j}], r'[x_1/c_{s_1,j}]) = tt$$

for $j = 1, \ldots, n_{s_1}$. Therefore, as $\mathrm{VER}^{\Gamma}(R, r, r')$ is defined in this case by

$$\mathrm{VER}^{\Gamma}(R, r, r') = \bigwedge_{j=1}^{j=n_{s_1}} \mathrm{VER}^{\Gamma}(R, r[x_1/c_{s_1,j}], r'[x_1/c_{s_1,j}])$$

that is clearly equivalent to an application of Rule (v) of $\mathrm{EQWIL}^{\Gamma,E}$ we have

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e'' \iff \mathrm{VER}^{\Gamma}(R, r, r') = tt.$$

as required.

□

Corollary 4. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$ be some system of equations such that there exists a complete TRS R equivalent to E. Also let Γ be inductive for some algebra A such that $A \cong I(\Sigma, E)$. If $e = (\tau = \tau') \in EQ(\Sigma, X)$ is some equation such that CEorFC(e) holds then*

$$I(\Sigma, E) \models e \iff VER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. Immediate from Corollary 3 and Lemma 47.

□

The following result formalizes the idea that we may automate deductions in first-order systems of primitive recursive equations to prove equations that satisfy the ComTRS predicate. However, for later convenience when we use this result we phrase the premises of the lemma in a different, but equivalent form using only the predicate CEorFC.

Lemma 48. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\Sigma, X)$ such that there exists a complete TRS R equivalent to E. Also, let $e = (\tau = \tau') \in EQ(\Sigma, X)$ be defined such that $\tau \in T(\Sigma, X)$ wherein $x \in X_s$ for some $s \in S$ is the left-most variable of τ. If either $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some $c_{s,j} \in \Sigma_{\lambda,s}$ and $CEorFC(e[x/c_{s,j}])$ holds for $j = 1, \ldots, n_s \in \mathbb{N}^+$ or $\Gamma_s = \{b_s, g_s\}$ for some $b_s \in \Sigma_{\lambda,s}$ and for some $g_s \in \Sigma_{s,s}$ and $CEorFC(e[x/b_s])$ holds and $CEorFC(e[x/g_s(\chi)])$ holds and*

$$\mathbb{R} = R \cup \{\tau[x/\chi] \mapsto \tau'[x/\chi]\}$$

wherein χ is some constant of type s such that $\chi \notin \Sigma$ is complete then

$$EQWIL^{\Gamma,E} \vdash e \iff VER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. We consider the two cases:
(1) $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,j} \in \Sigma$ for $j = 1, \ldots, n_s$.
(2) $\Gamma_s = \{b_s, g_s\}$ for some constant $b_s \in \Sigma_{\lambda,s}$ and for some algebraic operation $g_s \in \Sigma_{s,s}$.

Again notice that by Lemma 43 we may assume without loss of generality that any application of either Rule(v) or Rule (vi) occurs as the last step of a deduction.
Sub-Case (1) $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,j} \in \Sigma$ for $j = 1, \ldots, n_s$.. This case follows immediately by Lemma 47.
Sub-Case (2) $\Gamma_s = \{b_s, g_s\}$ for some constant $b_s \in \Sigma_{\lambda,s}$ and for some algebraic operation $g_s \in \Sigma_{s,s}$. In this case

$$\mathrm{VER}^{\Gamma}(R, \tau, \tau') = B_1 \wedge B_2$$

wherein

$$B_1 = \mathrm{VER}^{\Gamma}_s(R, \tau[x/b_s], \tau'[x/b_s])$$

and

$$B_2 = \mathrm{VER}^{\Gamma}_s(R', \tau[x/g_s(\chi)], \tau'[x/g_s(\chi)])$$

wherein

$$R' = R \cup \{\tau[x/\chi] \mapsto \tau'[x/\chi]\}.$$

First, notice that as by hypothesis $\mathrm{CEorFC}(\tau[x/b_s] = \tau'[x/b_s])$ holds and R is complete, by Lemma 47 we have

$$\mathrm{EQWIL}^{\Gamma,E} \vdash \tau[x/b_s] = \tau'[x/b_s] \iff \mathrm{VER}^{\Gamma}(R, \tau[x/b_s] = \tau'[x/b_s]) = tt$$

and hence $B_1 = tt \Longleftrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash \tau[x/b_s] = \tau'[x/b_s]$.

Also notice that as by hypothesis $\mathrm{CEorFC}(\tau[x/g_s(\chi)] = \tau'[x/g_s(\chi)])$ holds and R' is complete by hypothesis we have

$$\mathrm{EQWIL}^{\Gamma,E'} \vdash \tau[x/g_s(\chi)] = \tau'[x/g_s(\chi)] \Longleftrightarrow \mathrm{VER}^{\Gamma}(R', \tau[x/g_s(\chi)] = \tau'[x/g_s(\chi)]) = tt$$

wherein

$$E' = E \cup \{\tau[x/\chi] = \tau'[x/\chi]\}$$

and hence $B_2 = tt \Longleftrightarrow \mathrm{EQWIL}^{\Gamma,E'} \vdash \tau[x/g_s(\chi)] = \tau'[x/g_s(\chi)]$. Therefore as $\mathrm{VER}^{\Gamma,E}(R, \tau, \tau') = B_1 \wedge B_2$ in this case, that is clearly equivalent to an application of Rule (vi) of $\mathrm{EQWIL}^{\Gamma,E}$, we have

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e \Longleftrightarrow \mathrm{VER}^{\Gamma}(R, \tau, \tau') = tt$$

as required.

□

7.3.3 Simulating Deductions in Primitive Recursive Arithmetic

We are now in a position to state formally the first of our main results concerning the use of the calculus EQWIL in the context of primitive recursive arithmetic.

Lemma 49. *Let $\Gamma \subseteq \Sigma'$ be defined by $\Gamma_{\mathbf{n}} = \{0, Succ\}$, $\Gamma_{\mathbf{b}} = \{tt, ff\}$. Also, let $\Phi \in PREQ(\Sigma, X)$, let $R = TRCON(\Phi) \subseteq TRS(\Sigma', X)$ wherein Σ' is defined as in Definition 66. If f and g are two of the functions symbols of type $(\mathbf{n}\, u, s)$ and $(\mathbf{n}\, u', s)$ respectively for some $u, u' \in \underline{S}^+$ and for some $s \in S$ appearing in R such that $f \neq g$ then*

$$f(x, t_1, \ldots, t_{|u|}) = g(x, t'_1, \ldots, t'_{|u'|})$$

is weakly decidable for any variable $x \in X_{\mathbf{n}}$, for any terms $t_i \in T(\Sigma, X)_{u_i}$ for $i = 1, \ldots, |u|$ and for any terms $t'_j \in T(\Sigma, X)_{u'_j}$ for $j = 1, \ldots, |u'|$.

Proof. By the definition of VER using Lemma 39 and Lemma 40

□

Notice in particular that Lemma 49 states that VER is appropriate as a total proof assistant in the context of primitive recursive equations. However, while Lemma 49 is a useful fact from the perspective of automated verification it is limited result in the sense that Lemma 49 *does not* mean that

$$\mathrm{VER}^{\Gamma}(R, f(x, t_1, \ldots, t_{|u|}), g(x, t'_1, \ldots, t'_{|u'|})) = ff$$

implies

$$\mathrm{EQWIL}^{\Gamma,E} \not\vdash f(x, t_1, \ldots, t_{|u|}) = g(x, t'_1, \ldots, t'_{|u'|}))$$

wherein $E = \mathrm{EQCON}(\Phi) \subseteq \mathrm{EQ}(\Sigma', X)$.

As we have already stated, while in general we cannot define a total proof tool for systems of primitive recursive equations, with respect to the four classes of equations we identified in the introduction, initial truth is decidable with respect to EQWIL using VER. This statement is formalized in the following theorem.

Theorem 15. *Let* $\Phi \in PREQ(\Sigma, X)$, *let* $R = TRCON(\Phi) \subseteq TRS(\Sigma', X)$ *and let*

$$E = EQCON(\Phi) \subseteq EQ(\Sigma', X)$$

wherein Σ' *is defined as in Definition 66. Also, let* $\Gamma \subseteq \Sigma'$ *be defined such that* $\Gamma_{\mathbf{n}} = \{0, Succ\}$, $\Gamma_{\mathbf{b}} = \{tt, ff\}$ *and let* $e = (\tau = \tau') \in EQ(\Sigma', X)$. *If* e *satisfies either*

(A) $\tau, \tau' \in T(\Sigma')$; *or*

(B) $\tau, \tau' \in T(\Sigma', X)$ *and if* $x \in X_s$ *for some sort* $s \in S$ *occurs in* e *then* $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ *for some constants* $c_{s,j} \in \Sigma_{\lambda,s}$ *for* $j = 1, \ldots, n_s \in \mathbb{N}^+$; *or*

(C) $\tau = \sigma(x, \tau_1, \ldots, \tau_n)$ *and* $\tau' = c$ *wherein* $\sigma \in \Sigma'$ *is some algebraic operation,* $c \in \Sigma'$ *is any constant,* $x \in X_{\mathbf{n}}$ *and for* $i = 1, \ldots, n$ *if* $y \subseteq \tau_i$ *wherein* $y \in X_s$ *for some* $s \in S$ *is some variable then* $y \neq x$ *and* $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ *for some constants* $c_{s,j} \in \Sigma_{\lambda,s}$ *for* $j = 1, \ldots, n_s \in \mathbb{N}^+$; *or*

(D) $\tau = \sigma(x, \tau_1, \ldots, \tau_n)$ *and* $\tau' = \sigma'(x, \tau'_1, \ldots, \tau'_m)$ *wherein* $\sigma, \sigma' \in \Sigma'$ *are some algebraic operations such that* $\sigma \neq \sigma'$, $x \in X_{\mathbf{n}}$ *and for* $i = 1, \ldots, n \in \mathbb{N}$ *and for* $j = 1, \ldots, m \in \mathbb{N}$ *if either* $y \subseteq \tau_i$ *or* $y \subseteq \tau'_j$ *wherein* $y \in X_s$ *for some* $s \in S$ *is some variable then* $x \neq y$ *and* $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ *for some constants* $c_{s,l} \in \Sigma_{\lambda,s}$ *for* $l = 1, \ldots, n_s \in \mathbb{N}^+$

then $EQWIL^{\Gamma,E} \vdash e$ *is decidable.*

Proof. We consider each particular set of hypothesis in turn.
Case (A) Notice that in this case CEorFC(e) holds. Therefore, by Lemma 47 we have that $\mathrm{EQWIL}^{\Gamma,E} \vdash e$ is decidable by virtue of the fact that $\mathrm{VER}^{\Gamma}(R, \tau, \tau') = tt \Longleftrightarrow \mathrm{EQWIL}^{\Gamma,E} \vdash e$.
Case (B) Again notice that in this case CEorFC(e) holds. Therefore, by the same argument as Case (A) we have that $\mathrm{EQWIL}^{\Gamma,E} \vdash e$ is decidable as required.
Case (C) First, notice that by Lemma 38 we have that ComTRS(R, e) holds and hence

$$R' = R \cup \{\tau[x/\chi] \mapsto c\}$$

is complete for any constant χ of sort s such that $\chi \notin \Sigma'$. Therefore, as by hypothesis for each $y \subseteq \tau$ wherein $y \in X_s$ for some $s \in S$ is some variable such that $y \neq x$ and $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,j} \in \Sigma_{\lambda,s}$ for $j = 1, \ldots, n_s \in \mathbb{N}^+$ by definition CEorFC($e[x/0]$) holds, and CEorFC($e[x/Succ(\chi)]$ holds and hence by Lemma 48

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e \Longleftrightarrow \mathrm{VER}(R, \tau, c) = tt;$$

that is, $\mathrm{EQWIL}^{\Gamma,E} \vdash e$ is decidable as required.
Case (D) This case follows by a similar argument to Case (C) using the fact that ComTRS(R, e) holds by Lemma 39 and hence

$$R' = R \cup \{\tau[x/\chi] \mapsto \tau'[x/\chi]\}$$

is complete for any constant χ of sort s such that $\chi \notin \Sigma'$.

□

Discussion. Notice that for each of the class of terms above Theorem 15 states that VER is suitable as a total proof assistant; that is, with respect to EQWIL the function VER can find a proof of a hypothesis if and only if such a proof exists. While at an abstract level it is relatively straightforward to visual the class of terms that are identified by Cases (A) to (D) of Theorem 15, the practical use of this result is not so clear. Therefore, in Section 7.5 we return to this point when we identify one of the practical implications of this result.

Finally, one fact that it is important to re-iterate at this point is that EQWIL is not complete with respect to initial truth and hence in general even with respect to the classes of equations identified in Theorem 15

$$\mathrm{VER}^{\Gamma}(R, \tau', \tau') = \mathit{ff} \not\Longrightarrow \mathrm{I}(\Sigma, E) \not\models (\tau = \tau');$$

that is, even though VER behaves as a total proof tool for EQWIL for such equations, because EQWIL cannot be complete it may be the case that e is true in the initial model, but e is not provable by EQWIL. However, in the case that CEorFC(e) holds we do have provability if and only if $\mathrm{I}(\Sigma, E) \models e$; that is,

Corollary 5. *Let $\Gamma \subseteq \Sigma$ be defined such that $\Gamma_{\mathrm{n}} = \{0, Succ\}$, $\Gamma_{\mathrm{b}} = \{tt, ff\}$. Also let $\Phi \in PREQ(\Sigma, X)$, let $R = TRCON(\Phi) \subseteq TRS(\Sigma', X)$ and let $E = EQCON(\Phi) \subseteq EQ(\Sigma', X)$ wherein Σ' is defined as in Definition 66, and Γ is inductive for some algebra A such that $A \cong I(\Sigma', E')$. If $e = (\tau = \tau') \in EQ(\Sigma', X)$ is some term such that CEorFC(e) holds then*

$$I(\Sigma', E') \models e \Longleftrightarrow VER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. Immediate by Theorem 15 and by Lemma 47.

□

In the following section we will show how we may use and generalize these results in the context of reasoning about STs, by relaxing our requirement that variables range over finite carrier to the requirement that variables may also range over streams of the form $[T \rightarrow A]$ wherein A is a finite carrier.

7.4 The Automated Verification of STs

Drawing on results developed in previous chapters we now examine the use of the calculus EQWIL to reason about initial truth in stream algebras. In particular we develop a function AV to verify the equality of two STs using an extended version of the function VER (EVER), and identify certain classes of STs, including a broad class of hardware devices, for which AV is a total proof tool.

7.4.1 Using EQWIL to Reason About Stream Algebras

Recall the discussion following Definition 94 concerning the use of signatures of constructors in the context of stream algebras. In particular, recall that for any carrier A such that $|A| \geq 2$ the function space $[T \to A]$ is uncountable, and hence we cannot find a finite Γ to inductively generate the set of all streams over A. However, we can make use of the following positive result.

Lemma 50. *Let $E \subseteq EQ(\Sigma, X)$ and let $\underline{E} \subseteq EQ(\underline{\Sigma}, \underline{X})$. If $\Gamma \subseteq \Sigma$ is inductive for some algebra A such that $A \cong I(\Sigma, E)$ then for any $e \in EQ(\underline{\Sigma}, \underline{X})$*

$$EQWIL^{\Gamma,\underline{E}} \vdash e \Longrightarrow I(\underline{\Sigma}, \underline{E}) \models e.$$

Lemma 50 provides the theoretical basis for the development of the function AV that we will use to verify the correctness of STs specified in ASTRAL.

7.4.2 Extending VER

Notice that Lemma 50 guarantees the soundness of VER with respect to stream algebras. However, while VER remains an useful tool in this context it is not as effective in its capabilities as we would like as the following example demonstrates.

Example 21. Given the following (informally presented) system of primitive recursive equations E defined by

$$And(x, y) = \begin{cases} ff & \text{if } x = ff \text{ and } y = ff, \\ ff & \text{if } x = tt \text{ and } y = ff, \\ ff & \text{if } x = ff \text{ and } y = tt, \text{ and} \\ tt & \text{otherwise,} \end{cases}$$

notice that clearly

$$\text{EQWIL}^{\Gamma,E} \vdash And(eval(0, X), ff) = ff.$$

However, notice that for any $\Gamma \supseteq \Gamma_{\mathfrak{b}} = \{tt, ff\}$ that

$$\text{VER}^{\Gamma}(R, And(eval(0, X), ff), ff) = ff$$

wherein R is the TRS created by orientating E as left-to-right re-write rules and X is some stream variable of sort $\underline{\mathbf{b}}$. Therefore, even by moving from first-order equations to weak second-order equations we lose some of the properties of VER that we identified in the previous section. While in this simple example it is possible to modify E to give E' such that $\text{I}(\Sigma, E') \cong \text{I}(\Sigma, E)$ and to generate a complete TRS R' equivalent to E' so that

$$\text{VER}^{\Gamma}(R', And(eval(0, X), ff), ff) = tt$$

we cannot in general expect this to be the case as the problem of finding such an E' is essentially equivalent to finding an ω-complete specification (see Example 22). Moreover, as in the sequel we are interested in working with equational specifications that are automatically created from ASTRAL programmes, wherever possible we do not wish to place further restrictions upon the form of equations we must work with to guarantee the effectiveness of our verification techniques. As a consequence, in the context of stream algebras and their defining equations we prefer to modify the function VER to give the function EVER that is a more appropriate as the basis of a verification tool.

The formulation of EVER requires some further preliminary definitions. We begin by reformulating the predicates CEorFC and ComTRS in the context of stream algebras.

Definition 101. For each $\underline{\Sigma}$, for each $\Gamma \subseteq \underline{\Sigma}$ and for each $\underline{X}$ we define the predicate

$$\underline{\mathrm{CEorFC}}^{\underline{\Sigma},\Gamma,\underline{X}} \subseteq \mathrm{EQ}(\underline{\Sigma},\underline{X})$$

(ambiguously denoted $\underline{\mathrm{CEorFC}}$) as follows: for each $e \in \mathrm{EQ}(\underline{\Sigma},\underline{X})$ the predicate $\underline{\mathrm{CEorFC}}(e)$ holds if and only if either

(1) $e \in \mathrm{EQ}(\underline{\Sigma})$; that is, e is a closed term (notice that this implies that $e \in \mathrm{EQ}(\Sigma)$); or

(2) $e \in \mathrm{EQ}(\underline{\Sigma},\{x_1,\ldots,x_n\})$ for some $x_i \in \underline{X}_{s_i}$ for some $s_i \in \underline{S}$ such that for each $i = 1,\ldots,n \in \mathbb{N}^+$ either $\Gamma_{s_i} = \{c_{s_i,1},\ldots,c_{s_i,n_{s_i}}\}$ for some constants $c_{s,j} \in \Sigma_{\lambda,s_i}$ for $j = 1,\ldots,n_{s_j} \in \mathbb{N}$ or $\Gamma_{s_i} = \varnothing$ and $\Gamma_{r_i} = \{c_{r_i,1},\ldots,c_{r_i,n_{r_i}}\}$ for some constants $c_{s,j} \in \Sigma_{\lambda,r_i}$ for $j = 1,\ldots,n_{r_j} \in \mathbb{N}$ wherein $r_j = s_j$; that is, e is defined over variables that either range over finite carriers or that range over streams whose co-domains are finite carriers.

Definition 102. For each $\underline{\Sigma}$, for each $\Gamma \subseteq \underline{\Sigma}$ and for each $\underline{X}$ we define the predicate

$$\underline{\mathrm{ComTRS}}^{\underline{\Sigma},\Gamma,\underline{X}} \subseteq \mathrm{TRS}(\underline{\Sigma},\underline{X}) \times \mathrm{EQ}(\underline{\Sigma},\underline{X})$$

(ambiguously denoted $\underline{\mathrm{ComTRS}}$) as follows: for each $R \subseteq \mathrm{TRS}(\underline{\Sigma},\underline{X})$ and for each $e = (\tau = \tau') \in \mathrm{EQ}(\underline{\Sigma},\underline{X})$ the predicate $\mathrm{ComTRS}(R,e)$ holds if and only if either

(1) $\underline{\mathrm{CEorFC}}(e)$ holds; or

(2) $e = (\tau = \tau') \in \mathrm{EQ}(\underline{\Sigma},\{x_1,\ldots,x_n\})$ for some $x_i \in \underline{X}_{s_i}$ for some $s_i \in \underline{S}$ for $i = 1,\ldots,n \in \mathbb{N}^+$ wherein x_1 is the left-most variable of τ such that $\Gamma_{s_1} = \{b_{s_1}, g_{s_1}\}$ for some $b_{s_1} \in \Sigma_{\lambda,s_1}$ and for some $g_{s_1} \in \Sigma_{s_1,s_1}$; and the TRS R' defined by

$$R' = R \cup \{r'\}$$

is complete wherein r' is formed by orienting e' as a left-to-right re-write rule; and $e' \in \mathrm{EQ}(\underline{\Sigma''},\underline{X})$ is any equation defined by

$$e' = e[x_1/\chi][x_i/\rho_i]_{i=2}^{i=n}$$

wherein $\Sigma'' \subseteq \Sigma \cup \{\chi\}$, χ is some constant of sort s_1 such that $\chi \notin \Sigma$ and the terms $\rho_i \in T(\underline{\Sigma''},\underline{X})_{s_i}$ are defined such that $\rho_i \not\supseteq \chi$ for $i = 1,\ldots,n$.

Lemma 51. *Let $\Gamma \subseteq \underline{\Sigma}$, let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ and let $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$. If $\sim \underline{CEorFC}(e)$ holds, but $ComTRS(R, e)$ holds then $\underline{ComTRS}(R, e)$ holds.*

Proof. Immediate from Lemmata 38 and 39.

□

To define EVER we require the following two sub-functions. The first sub-function enables us to count sub-terms of the form $eval(t, x)$ for some $t \in T(\underline{\Sigma}, \underline{X})_n$ and for some $x \in \underline{X}_s$ for some $s \in S$. The second sub-function allows us to replace sub-terms of the form $eval(t, y)$ with a new variable symbol x.

Definition 103. For each Σ and for each X we define

$$\text{AnyEvals}^{\Sigma,X} : T(\underline{\Sigma}, \underline{X}) \to \mathbb{B}$$

(ambiguously denoted AnyEvals) for each $\tau \in T(\underline{\Sigma}, \underline{X})$ by $\text{AnyEvals}(\tau) = tt$ if and only if there exist a term $\eta = eval_s(\tau, y) \subseteq \tau$ for some $s \in S$ and for some $y \in \underline{X}_s$.

Definition 104. For each Σ , for each $\Gamma \subseteq \Sigma$ and for each X we define

$$\text{SubEvals}^{\Sigma,\Gamma,X} : T(\underline{\Sigma}, \underline{X}) \times T(\underline{\Sigma}, \underline{X}) \to T(\underline{\Sigma}, \underline{X}) \times T(\underline{\Sigma}, \underline{X})$$

(ambiguously denoted SubEvals^{Γ}) by

$$(\forall t, t' \in T(\underline{\Sigma}, \underline{X}))$$

$$\text{SubEvals}^{\Gamma}(t, t') = \begin{cases} (t, t') & \text{if AnyEvals}(t) = \text{AnyEvals}(t') = ff \\ \text{SubEvals}^{\Gamma}(\tau, \tau') & \text{otherwise} \end{cases}$$

wherein $\tau = t[\eta/x]$ and $\tau' = t'[\eta/x]$ wherein if $\text{AnyEvals}(t) = tt$ then η is the left-most sub-term of t such that $\eta = eval_s(\rho, y)$ otherwise η is the left-most sub-term of t' such that $\eta = eval_s(\rho, y)$ for some $\rho \in T(\Sigma, X)_n$ and for some $y \in \underline{X}_s$; and $x \in X_s$ is some variable such that $x \not\subseteq t$ and $x \not\subseteq t'$.

Lemma 52. *Let $\Gamma \subseteq \Sigma$ and let $E \subseteq EQ(\underline{\Sigma}, \underline{X})$. For any equation $e = (t = t') \in EQ(\underline{\Sigma}, \underline{X})$*

$$EQWIL^{\Gamma,E} \vdash e' \Longrightarrow EQWIL^{\Gamma,E} \vdash e$$

wherein $e' = (\tau = \tau')$ and

$$(\tau, \tau') = SubEvals^{\Gamma}(t, t').$$

Proof. By induction on the number $k \in \mathbb{N}^+$ of occurrences of a term $\eta = eval_s(\rho, y)$ such that either $\eta \subseteq t$ or $\eta \subseteq t'$. We sketch a proof. The key step is to observe in the case wherein $k = 1$ that as equational logic is a sub-logic of EQWIL if we can prove e' then e is provable by an application of the substitution rule as follows:

$$\frac{t[\eta/x] = t'[\eta/x] \quad \eta = \eta}{t[x/\eta] = t'[x/\eta].}$$

□

Indeed, this fact is the basis for the definition of an extended version of the function VER.

Definition 105. For each Σ, for each $\Gamma \subseteq \Sigma$, for each X and for each $s \in S$ we define

$$\mathrm{EVER}_s^{\Sigma,\Gamma,X} : \mathrm{TRS}(\underline{\Sigma}, \underline{X}) \times T(\underline{\Sigma}, \underline{X})_s \times T(\underline{\Sigma}, \underline{X})_s \rightsquigarrow \mathbb{B}$$

(ambiguously denoted EVER^{Γ}) as follows:

$$(\forall R \in \mathrm{TRS}(\underline{\Sigma}, \underline{X}))\,(\forall \tau, \tau' \in T(\underline{\Sigma}, \underline{X})_s)$$

$$\mathrm{EVER}^{\Gamma}(R, \tau, \tau') = \begin{cases} \mathrm{ERED}(R, \tau, \tau') & \text{if } \tau, \tau' \in T(\Sigma) \text{ or} \\ & \text{for each } x \in X_s \text{ for some } s \in S \\ & \text{such that either } x \subseteq \tau \text{ or } x \subseteq \tau' \\ & \Gamma_s = \varnothing \\ \bigwedge_{i=1}^{i=n_{s_1}} \mathrm{EVER}_{s_1}^{\Gamma}(R, \tau[x_1/c_{s_1,i}], \tau'[x_1/c_{s_1,i}]) & \text{if } \tau, \tau' \in T(\Sigma, \{x_1, \ldots, x_n\})_s \\ & \text{and } \Gamma_{s_1} = \{c_{s_1,1}, \ldots, c_{s_1,n_{s_1}}\}; \text{ and} \\ B_1 \wedge B_2 & \text{if } \tau, \tau' \in T(\Sigma, \{x_1, \ldots, x_n\})_s \\ & \text{and } \Gamma_{s_1} = \{b_{s_1}, g_{s_1}\} \end{cases}$$

wherein

$$B_1 = \mathrm{EVER}_{s_1}^{\Gamma}(R, \tau[x_1/b_{s_1}], \tau'[x_1/b_{s_1}])$$

$$B_2 = \mathrm{EVER}_{s_1}^{\Gamma}(R', \tau[x_1/g_{s_1}(\chi)], \tau'[x_1/g_{s_1}(\chi)])$$

and $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, n \in \mathbb{N}^+$ are the variables over which τ and τ' are defined wherein either x_1 is the left-most variable such that $x_1 \subseteq \tau$ and $\Gamma_{s_1} \neq \varnothing$ if such a variable exists or x_1 is the left-most variable such that $x_1 \subseteq \tau'$ and $\Gamma_{s_1} \neq \varnothing$ otherwise; χ is some constant of type s_1 such that $\chi \notin \Sigma$; $R' = R \cup \{\tau[x_1/\chi] \mapsto \tau'[x_1/\chi]\}$; and for each Σ, for each $\Gamma \subseteq \Sigma$, for each X and for each $s \in S$ we define

$$\mathrm{ERED}^{\Sigma,\Gamma,X} : \mathrm{TRS}(\underline{\Sigma}, \underline{X}) \times T(\underline{\Sigma}, \underline{X})_s \times T(\underline{\Sigma}, \underline{X})_s \rightsquigarrow \mathbb{B}$$

(ambiguously denoted ERED^{Γ}) for each R, τ and τ' as above by

$$\mathrm{ERED}^{\Gamma}(R, \tau, \tau') = \begin{cases} \mathrm{RED}(R, \tau, \tau') & \text{if } \mathrm{AnyEvals}(\mathrm{NF}^R(\tau)) = \mathrm{AnyEvals}(\mathrm{NF}^R(\tau')) = \mathit{ff} \\ \mathrm{EVER}^{\Gamma}(R, t, t') & \text{otherwise} \end{cases}$$

wherein $(t, t') = \mathrm{SubEvals}^{\Gamma}(\mathrm{NF}^R(\tau), \mathrm{NF}^R(\tau'))$.

Discussion. Notice that using this extended definition of VER given E and R as defined in Example 21 that

$$\mathrm{EVER}^{\Gamma}(R, \mathit{And}(\mathit{eval}(0, X), \mathit{ff}), \mathit{ff}) = \mathit{tt}$$

and hence in general the class of weak second-order equations for which EVER can simulate EQWIL is strictly larger than the class of weak second-order equations for which VER can simulate EQWIL.

We now use Lemma 52 to establish formally that EVER is both terminating and sound with respect to the calculus EQWIL.

The Soundness of EVER. We begin by defining two functions that enable us to identify and count the number of non-stream variables and stream variables respectively that occur in equations and terms.

Definition 106. For each $\underline{\Sigma}$ and for each $\underline{X}$ we define $\mathrm{NonStrVars}^{\underline{\Sigma},\underline{X}} : T(\underline{\Sigma},\underline{X}) \to \wp(X)$ and $\mathrm{NumNonStrVars}^{\underline{\Sigma},\underline{X}} : T(\underline{\Sigma},\underline{X}) \to \mathbb{N}$ (ambiguously denoted NonStrVars and NumNonStrVars respectively) by

$$(\forall \tau \in T(\underline{\Sigma},\underline{X})) \quad \mathrm{NonStrVars}(\tau) = \{x \in X \mid x \subseteq \tau\}$$

and

$$(\forall \tau \in T(\underline{\Sigma},\underline{X})) \quad \mathrm{NumNonStrVars}(\tau) = |\mathrm{NonStrVars}(\tau)|.$$

We also ambiguously define $\mathrm{NumNonStrVars} : \mathrm{EQ}(\underline{\Sigma},\underline{X}) \to \mathbb{N}$ by

$$(\forall e = (\tau = \tau') \in \mathrm{EQ}(\underline{\Sigma},\underline{X}))$$

$$\mathrm{NumNonStrVars}(e) = |\mathrm{NonStrVars}(\tau) \cup \mathrm{NonStrVars}(\tau')|.$$

Thus, NumNonStrVars counts the number of distinct non-stream variables that occur in either a term or an equation.

Definition 107. For each $\underline{\Sigma}$ and for each $\underline{X}$ we define $\mathrm{StrVars}^{\underline{\Sigma},\underline{X}} : T(\underline{\Sigma},\underline{X}) \to \wp(\underline{X} - X)$ and $\mathrm{NumStrVars}^{\underline{\Sigma},\underline{X}} : T(\underline{\Sigma},\underline{X}) \to \mathbb{N}$ (ambiguously denoted StrVars and NumStrVars respectively) by

$$(\forall \tau \in T(\underline{\Sigma},\underline{X})) \quad \mathrm{StrVars}(\tau) = \{x \in (\underline{X} - X) \mid x \subseteq \tau\}$$

and

$$(\forall \tau \in T(\underline{\Sigma},\underline{X})) \quad \mathrm{NumStrVars}(\tau) = |\mathrm{StrVars}(\tau)|.$$

We also ambiguously define $\mathrm{NumStrVars} : \mathrm{EQ}(\underline{\Sigma},\underline{X}) \to \mathbb{N}$ by

$$(\forall e = (\tau = \tau') \in \mathrm{EQ}(\underline{\Sigma},\underline{X})) \quad \mathrm{NumStrVars}(e) = |\mathrm{StrVars}(\tau) \cup \mathrm{StrVars}(\tau')|.$$

Thus, NumStrVars counts the number of distinct stream variables that occur in either a term or an equation.

Using the function NumStrVars we now define an important concept that we will require in the sequel.

Definition 108. Let $R \subseteq \mathrm{TRS}(\underline{\Sigma},\underline{X})$. If for each $\tau \in T(\underline{\Sigma},\underline{X})$

$$\mathrm{NumStrVars}(\tau) \geq \mathrm{NumStrVars}(\mathrm{NF}^R(\tau))$$

then we say that R is *stream variable reducing.*

The identification of TRSs that are stream variable reducing allows us to identify certain classes of equations for which VER and EVER have the same behaviour and hence conveniently enable us to make use of some of the results of the previous section.

Lemma 53. *Let $\Gamma \subseteq \Sigma$ and let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ be some TRS that is stream variable reducing. If $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$ and $NumStrVars(e) = 0$ then*

$$EVER^{\Gamma}(R, \tau, \tau') = VER^{\Gamma}(R, \tau, \tau').$$

Proof. Immediate from the definition of EVER^{Γ} and the hypothesis that R is stream variable reducing.

□

Lemma 54. *Let $\Gamma \subseteq \Sigma$ and let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ be some TRS that is stream variable reducing. If $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$ and $NumStrVars(e) = 1$ and $NumNonStrVars(e) = 0$ then*

$$EVER^{\Gamma}(R, \tau, \tau') = VER^{\Gamma}(R, \tau, \tau').$$

Proof. By induction on the number $k \in \mathrm{N}$ of occurrences of $x \in \underline{X}$ in $e' = (\mathrm{NF}^R(\tau) = \mathrm{NF}^R(\tau'))$. The key step is the basis case wherein $k = 0$ that follows by Lemma 53. Again we leave the details of a full proof to the reader.

□

Lemma 55. *Let $\Gamma \subseteq \Sigma$, let $E \subseteq EQ(\underline{\Sigma}, \underline{X})$ and let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ be some TRS that is stream variable reducing and that is equivalent to E. Also let $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$. If e is weakly decidable and $NumStrVars(e) = 1$ then*

$$EVER^{\Gamma}(R, \tau, \tau')\downarrow$$

and

$$EVER^{\Gamma}(R, \tau, \tau') = tt \Longrightarrow EQWIL^{\Gamma, E} \vdash e.$$

Proof. By induction on the number $k = \mathrm{NumNonStrVars}(e)$.
The key step is again the basis case that requires a sub-induction on the number $l \in \mathrm{N}$ of occurrences of sub-terms of the form $eval(\theta, x)$ in the normal forms of τ and τ' produced during the iteration of the process of defining

$$\mathrm{EVER}^{\Gamma}(R, \tau, \tau') = \mathrm{EVER}^{\Gamma}(R, t, t')$$

wherein

$$(t, t') = \mathrm{SubEvals}^{\Gamma}(\mathrm{NF}^R(\tau), \mathrm{NF}^R(\tau')).$$

(Notice that l is guaranteed to be finite by the hypothesis that R is stream variable reducing.) The basis case of this sub-induction follows by Lemma 53 and by Lemma 45. As before we leave the details of a full proof to the reader.

□

We are now in a position to establish the termination and soundness of EVER with respect to systems of primitive recursive equations defined over stream signatures.

Lemma 56. *If $R = TRCON(\Phi)$ for some $\Phi \in PREQ(\underline{\Sigma}, \underline{X})$ then R is stream variable reducing.*

Proof. Immediate by Lemma 7, Theorem 10 and the definition of TRCON.

□

Theorem 16. *Let $\Gamma \subseteq \Sigma$, let $\Phi \in PREQ(\underline{\Sigma}, \underline{X})$ and let $E = EQCON(\Phi)$. Also let $R = TRCON(\Phi)$. If the equation $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$ is weakly decidable then*

$$EVER^{\Gamma}(R, \tau, \tau') \downarrow$$

and

$$EVER^{\Gamma}(R, \tau, \tau') = tt \Longrightarrow EQWIL^{\Gamma, E} \vdash e.$$

Proof. By induction on the number $k = \text{NumNonStrVars}(e) + \text{NumStrVars}(e)$. We consider two basis cases:
Basis $k = 0$. This case follows immediately by Lemma 56, and by Lemma 45.
Basis $k = 1$. There are two sub-cases two consider:
(1) NumNonStrVars$(e) = 1$.
(2) NumStrVars$(e) = 1$.
Sub-Case (1) NumNonStrVars$(e) = 1$. As by hypothesis $\text{NumStrVars}(e) = 0$ again this case follows immediately by Lemma 56 and by Lemma 45.
Sub-Case (2) NumStrVars$(e) = 1$. This case follows by Lemma 56 and by Lemma 55.

If $\text{NumNonStrVars}(e) > 0$ then the rest of the proof now follows by a routine application of the Induction Hypothesis and by Lemma 56. If $\text{NumNonStrVars}(e) = 0$ the the rest of the proof follows by Lemma 45. Again the details are omitted.

□

Decidability. Using the predicates CEorFC and ComTRS we now extend the results of Section 7.3.2.

Lemma 57. *Let $\Gamma \subseteq \Sigma$, let $E \subseteq EQ(\underline{\Sigma}, \underline{X})$ and let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ be a complete TRS that is equivalent to E and that is stream variable reducing.*

If $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$ is some equation such that CEorFC(e) holds then

$$EQWIL^{\Gamma, E} \vdash e \Longrightarrow EVER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. By induction on the number $k = \text{NumNonStrVars}(e) + \text{NumStrVars}(e)$. We consider two basis cases.
Basis Case (1) $k = 0$. This case is obvious as by Lemma 41

$$\text{EQWIL}^{\Gamma, E} \vdash e \Longrightarrow E \vdash e$$

and by definition

$$\text{EVER}^{\Gamma}(R, \tau, \tau') = \text{RED}(R, \tau, \tau').$$

Basis Case (2) $k = 1$. We now consider two sub-cases:
(1) NumStrVars$(e) = 0$.
(2) NumNonStrVars$(e) = 0$.
Sub-Case (1) NumStrVars$(e) = 0$. First, notice that as by hypothesis NumStrVars$(e) = 0$ by Lemma 53

$$\mathrm{EVER}^{\Gamma}(R, \tau, \tau') = \mathrm{VER}^{\Gamma}(R, \tau, \tau').$$

Also notice that if NumStrVars$(e) = 0$ then by definition $\underline{\mathrm{CEorFC}}(e) \Longrightarrow \mathrm{CEorFC}(e)$. Therefore by Lemma 47 we have

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e \Longrightarrow \mathrm{EVER}^{\Gamma}(R, \tau, \tau') = tt$$

as required.
Sub-Case (2) NumNonStrVars$(e) = 0$. Let $x \in \underline{X}_s$ for some $s \in S$ be the variable such that either $x \subseteq \tau$ or $x \subseteq \tau'$. We now proceed by sub-induction on the number $n \in \mathbf{N}$ of the applications of Rule (v) in $\mathrm{EQWIL}^{\Gamma,E} \vdash e$. (Notice that we cannot have used Rule (vi) by the hypothesis that $\underline{\mathrm{CEorFC}}(e)$ holds.)
Sub-Basis $n = 0$. In this case

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e \Longrightarrow E \vdash e$$

and

$$E \vdash e \Longrightarrow \mathrm{NF}^{R}(\tau) = \mathrm{NF}^{R}(\tau')$$

by the hypothesis that R is complete, and

$$\mathrm{NF}^{R}(\tau) = \mathrm{NF}^{R}(\tau') \Longrightarrow \mathrm{RED}(R, \tau, \tau') = tt$$

by the definition of RED. Therefore, as by hypothesis NumNonStrVars$(e) = 0$ we have

$$\mathrm{EVER}^{\Gamma}(R, \tau, \tau') = \mathrm{RED}(R, \tau, \tau')$$

and hence

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e \Longrightarrow \mathrm{EVER}^{\Gamma}(R, \tau, \tau')$$

as required.
Sub-Induction Hypothesis. Assume for any $e' = (t = t') \in \mathrm{EQ}(\underline{\Sigma}, \underline{X})$ that if

$$\mathrm{NumNonStrVars}(e') = 0,$$

$\underline{\mathrm{CEorFC}}(e')$ holds and $\mathrm{EQWIL}^{\Gamma,E} \vdash e'$ using n' applications of Rule (v) for some fixed $n' \in \mathbf{N}$ that

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e' \Longrightarrow \mathrm{EVER}^{\Gamma}(r, t, t') = tt.$$

Sub-Induction Step. Let $e'' = (\theta = \theta') \in \mathrm{EQ}(\underline{\Sigma}, \underline{X})$, let $\underline{\mathrm{CEorFC}}(e'')$ hold and let $\mathrm{EQWIL}^{\Gamma,E} \vdash e''$ using $n = n' + 1$ applications of Rule (v). Notice that as by hypothesis NumNonStrVars$(e'') = 0$ if we have used an application of Rule (v) to show $\mathrm{EQWIL}^{\Gamma,E} \vdash e''$ then it must be the case that we have shown

$$\mathrm{EQWIL}^{\Gamma,E} \vdash \eta = (\rho, \rho')$$

for some equation $\eta \in \mathrm{EQ}(\underline{\Sigma}, \underline{X})$ such that either

$$eval(\kappa, x) \subseteq \rho$$

or

$$eval(\kappa, x) \subseteq \rho';$$

and we have used Rule (v) to deduce that

$$\mathrm{EQWIL}^{\Gamma,E} \vdash \eta[eval(\kappa, x)/y]$$

for some $y \in X$ such that y does not occur in η and deduced e'' using substitution as follows:

$$\frac{\mathrm{EQWIL}^{\Gamma,E} \vdash \eta[eval(\kappa, x)/y] \quad eval(\kappa, x) = eval(\kappa, x)}{\mathrm{EQWIL}^{\Gamma,E} \vdash e''.}$$

Also notice that by Lemma 42 without loss of generality we may assume that in the proof $\mathrm{EQWIL}^{\Gamma,E} \vdash e''$ there are no trivial deduction and hence any applications of Rules (i) to (iv) of EQWIL were applied before Rule (v) was applied. Therefore, as by hypothesis $\mathrm{EQWIL}^{\Gamma,E} \vdash \eta[eval(\kappa, x)/y]$ in n' applications of Rule (v) and by hypothesis R is stream variable reducing x is the only stream variable that can occur in $\eta[eval(\kappa, x)/y]$. Thus, if x does occur in $\eta[eval(\kappa, x)/y]$ then by the Sub-Induction Hypothesis

$$\mathrm{EQWIL}^{\Gamma,E} \vdash \eta[eval(\kappa, x)/y] \Longrightarrow \mathrm{EVER}^{\Gamma}(R, \rho[eval(\kappa, x)/y], \rho'[eval(\kappa, x)/y]) = tt$$

and if x does not occur in $\eta[eval(\kappa, x)/y]$ then by definition $\mathrm{NumStrVars}(\eta[eval(\kappa, x)/y]) = 0$ and by Case (1) we also have that

$$\mathrm{EQWIL}^{\Gamma,E} \vdash \eta[eval(\kappa, x)/y] \Longrightarrow \mathrm{EVER}^{\Gamma}(R, \rho[eval(\kappa, x)/y], \rho'[eval(\kappa, x)/y]) = tt.$$

Consequently, as by definition in this case

$$\mathrm{EVER}^{\Gamma}(R, \theta, \theta') = \mathrm{EVER}^{\Gamma}(R, \delta, \delta')$$

wherein $(\delta, \delta') = \mathrm{SubEvals}^{\Gamma}(\mathrm{NF}^{R}(\theta), \mathrm{NF}^{R}(\theta'))$, since R is complete by hypothesis we have

$$\delta = \delta' = \rho[eval(\kappa, x)/y] = \rho'[eval(\kappa, x)/y]$$

and hence

$$\mathrm{EQWIL}^{\Gamma,E} \vdash e'' \Longrightarrow \mathrm{EVER}^{\Gamma}(R, \theta, \theta') = tt$$

as required.

The rest of the proof now follows by a routine application of the Induction Hypothesis using a similar argument to that in Lemma 47.

□

Using Lemma 57 as promised we can now generalize Lemma 48 and Theorem 15 to the context of stream algebras.

Lemma 58. *Let $\Gamma \subseteq \Sigma$, let $E \subseteq EQ(\underline{\Sigma}, \underline{X})$ and let $R \subseteq TRS(\underline{\Sigma}, \underline{X})$ be a complete TRS that is equivalent to E and that is stream variable reducing. Also, let $e = (\tau = \tau') \in EQ(\underline{\Sigma}, \underline{X})$ be some equation such that $\tau \in T(\underline{\Sigma}, \underline{X})$ wherein $x \in X_s$ for some $s \in S$ is the left-most variable occurring in τ. If either $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some $c_{s,j} \in \Sigma_{\lambda,s}$ and $\underline{CEorFC}(e[x/c_{s,j}])$ holds for $j = 1, \ldots, n_s \in \mathbb{N}^+$ or $\Gamma_s = \{b_s, g_s\}$ for some $b_s \in \Sigma_{\lambda,s}$ and for some $g_s \in \Sigma_{s,s}$ and $\underline{CEorFC}(e[x/b_s])$ holds and $\underline{CEorFC}(e[x/g_s(\chi)])$ holds and*

$$\mathbb{R} = R \cup \{\tau[x/\chi] \rightarrowtail \tau'[x/\chi]\}$$

wherein χ is some constant of type s such that $\chi \notin \Sigma$ is complete then

$$EQWIL^{\Gamma,E} \vdash e \Longleftrightarrow EVER^{\Gamma}(R, \tau, \tau') = tt.$$

Proof. Similar to the proof of Lemma 48.

□

Theorem 17. *Let $\Phi \in PREQ(\underline{\Sigma}, \underline{X})$, let $R = TRCON(\Phi) \subseteq TRS(\underline{\Sigma'}, \underline{X})$ and let*

$$E = EQCON(\Phi) \subseteq EQ(\Sigma', X)$$

wherein Σ' is defined as in Definition 67. Also, let $\Gamma \subseteq \Sigma'$ be defined such that $\Gamma_{\mathbf{n}} = \{0, Succ\}$, $\Gamma_{\mathbf{b}} = \{tt, ff\}$ and let $e = (\tau = \tau') \in EQ(\underline{\Sigma'}, \underline{X})$. If e satisfies one of the following conditions: either

(A) *$\tau, \tau' \in T(\Sigma')$; or*

(B) *$\tau, \tau' \in T(\underline{\Sigma'}, \underline{X})$ and if $x \in X_s$ for some sort $s \in S$ occurs in e then either $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,j} \in \Sigma_{\lambda,s}$ for $j = 1, \ldots, n_s \in \mathbb{N}^+$ or $s = \underline{r}$ for some $r \in S$ and $\Gamma_r = \{c_{r,1}, \ldots, c_{r,n_r}\}$ for some constants $c_{r,j} \in \Sigma_{\lambda,r}$ for $j = 1, \ldots, n_r \in \mathbb{N}^+$; or*

(C) *$\tau = \sigma(x, \tau_1, \ldots, \tau_n)$ and $\tau' = c$ wherein $\sigma \in \Sigma'$ is some algebraic operation, $c \in \Sigma$ is any constant, $x \in X_{\mathbf{n}}$ and for $i = 1, \ldots, n$ if $y \subseteq \tau_i$ wherein $y \in X_s$ for some $s \in S$ is some variable such that $y \neq x$ then either $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,j} \in \Sigma'_{\lambda,s}$ for $j = 1, \ldots, n_s \in \mathbb{N}^+$ or $s = \underline{r}$ for some $r \in S$ and $\Gamma_r = \{c_{r,1}, \ldots, c_{r,n_r}\}$ for some constants $c_{r,j} \in \Sigma'_{\lambda,r}$ for $j = 1, \ldots, n_r \in \mathbb{N}^+$; or*

(D) *$\tau = \sigma(x, \tau_1, \ldots, \tau_n)$ and $\tau' = \sigma'(x, \tau'_1, \ldots, \tau'_m)$ wherein $\sigma, \sigma' \in \Sigma'$ are some algebraic operations such that $\sigma \neq \sigma'$, $x \in X_{\mathbf{n}}$ and for $i = 1, \ldots, n \in \mathbb{N}$ and for $j = 1, \ldots, m \in \mathbb{N}$ if either $y \subseteq \tau_i$ or $y \subseteq \tau'_j$ wherein $y \in X_s$ for some $s \in S$ is some variable such that $x \neq y$ then either $\Gamma_s = \{c_{s,1}, \ldots, c_{s,n_s}\}$ for some constants $c_{s,l} \in \Sigma'_{\lambda,s}$ for $l = 1, \ldots, n_s \in \mathbb{N}^+$ $s = \underline{r}$ for some $r \in S$ and $\Gamma_r = \{c_{r,1}, \ldots, c_{r,n_r}\}$ for some constants $c_{r,j} \in \Sigma'_{\lambda,r}$ for $j = 1, \ldots, n_r \in \mathbb{N}^+$*

then $EQWIL^{\Gamma,E} \vdash e$ is decidable.

Proof. Similar to the proof of Theorem 15 using Lemma 57 in the place of Lemma 47 and Lemma 58 in the place of Lemma 48.

□

In addition to the use of Theorem 17 in the following we also present a discussion of its practical implications in Section 7.5.

7.4.3 A Total Proof Assistant and Total Proof Tool For STs

Using Lemma 58 and Theorem 17 as our theoretical basis we now define the function AV that has very useful properties in the context of the verification of STs.

AV as a Total Proof Assistant. In the following definition the PREQ specification Ψ allows us to deal with the definition of additional algebraic operations not defined in Σ, and Cartesian form stream transformer definitions. The motivation for the inclusion of Ψ was discussed in Section 6.6 and an example can be found in Section 6.7.1 after the definition of the concatenation operator.

Definition 109. Let $\Psi \in \mathrm{PREQ}(\underline{\Sigma}, \underline{X})$ and let $\Sigma' = \Sigma \bigcup \mathcal{F}$ be the extended signature of Ψ as defined in Definition 50. Also let $\Gamma \subseteq \Sigma'$. For each $u, u' \in \underline{S}^+$ and for each $\underline{v} \in \underline{S}^+$ we define

$$\mathrm{AV}^{\Psi,\Gamma}_{u,u',\underline{v}} : \mathrm{ASTRAL}(\underline{\Sigma'}, \underline{X})_{u,\underline{v}} \times \mathrm{ASTRAL}(\underline{\Sigma'}, \underline{X})_{u',\underline{v}} \rightarrow \mathbb{B}$$

(ambiguously denoted $\mathrm{AV}^{\Psi,\Gamma}$) as follows: for each $\Phi \in \mathrm{ASTRAL}(\underline{\Sigma'}, \underline{X})_{u,\underline{v}}$ and for each $\Phi' \in \mathrm{ASTRAL}(\underline{\Sigma'}, \underline{X})_{u',\underline{v}}$ such that Φ and Φ' do not share any symbols other than those in $\underline{\Sigma'}$ and $\underline{X}$

$$\mathrm{AV}^{\Psi,\Gamma}(\Phi, \Phi') = \bigwedge_{i=1}^{i=|v|} \mathrm{EVER}^{\Gamma}(R^{\Phi,\Phi',\Psi}, \tau_i, \tau'_i)$$

wherein

$$R^{\Phi,\Phi',\Psi} = \mathrm{TRCON}(E^{\Phi,\Phi',\Psi} = (E^{\Phi} \uplus_1 (E^{\Phi'} \uplus_1 \Psi)))$$

wherein

$$E^{\Phi} = \chi^{\mathrm{ASTRAL}}(\Phi),$$
$$E^{\Phi'} = \chi^{\mathrm{ASTRAL}}(\Phi')$$

and for $i = 1, \ldots |v|$

$$\tau_i = f_i(x, x_1, \ldots, x_{|u|})$$

and

$$\tau'_i = g_i(x, x'_1, \ldots, x'_{|u'|})$$

wherein f_i and g_i are the symbols from $E^{\Phi,\Phi',\Psi}$ (and hence $R^{\Phi,\Phi',\Psi}$) representing the co-ordinate function of the Cartesian forms of the functions F and G represented by Φ and Φ' respectively, and $x \in X_t$ and $x_j \in \underline{X}_{u_j}$ for $j = 1, \ldots, |u|$ and $x'_l \in \underline{X}_{u'_j}$ for $l = 1, \ldots, |u'|$ are some distinct variable symbols.

Well-Definedness, Termination and Soundness. We first show that AV is well-defined. The termination properties of AV are considered in Lemma 59. Theorem 18 shows that AV is sound with respect to the semantics of ASTRAL programmes.

Well-Definedness. First, notice that by the well-definedness of χ^{ASTRAL} we have $\chi^{\mathrm{ASTRAL}}(\Phi) \in \mathrm{PREQ}(\underline{\Sigma'}, \underline{X})_{t\,u,v}$ and $\chi^{\mathrm{ASTRAL}}(\Phi') \in \mathrm{PREQ}(\underline{\Sigma'}, \underline{X})_{t\,u',v}$. Also, notice that as by hypothesis Φ and Φ' do not share any symbols other than those in $\underline{\Sigma'}$ and $\underline{X}$ and by the fact that χ^{ASTRAL}

does not introduce any function symbols common to $E^{\Phi} = \chi^{\mathrm{ASTRAL}}(\Phi)$ and $E^{\Phi'} = \chi^{\mathrm{ASTRAL}}(\Phi')$, by the well-definedness of $\uplus$ it is clear that $E^{\Phi,\Phi',\Psi} = E^{\Phi} \uplus_1 (E^{\Phi'} \uplus_1 \Psi)$ is well-defined as a $\mathrm{PREQ}(\underline{\Sigma'}, \underline{X})_{t\,u,v}$ specification. Therefore, by the well-definedness of TRCON we have $R^{\Phi,\Phi',\Psi} \subseteq \mathrm{TRS}(\underline{\Sigma'}, \underline{X})$ and hence by the well-definedness of EVER it is clear that $\mathrm{AV}^{\Psi,\Gamma}$ is well-defined as required.

Lemma 59. *If Ψ, Γ, Φ and Φ' are defined as above then*

$$AV^{\Psi,\Gamma}(\Phi,\Phi')\downarrow .$$

Proof. First, notice that by Lemma 49 $e_i = (\tau_i = \tau_i')$ for $j = 1,\ldots,|v|$ is weakly decidable with respect to $R^{\Phi,\Phi',\Psi}$. Also notice that $R^{\Phi,\Phi',\Psi}$ is stream variable reducing by Lemma 56. Therefore by Theorem 16

$$\mathrm{VER}^{\Gamma}(R^{\Phi,\Phi',\Psi}, \tau_i, \tau_i')\downarrow$$

for $j = 1,\ldots,|v|$. Consequently it is clear that $\mathrm{AV}^{\Psi,\Gamma}(\Phi,\Phi')$ is a total function as required.

□

Theorem 18. (The AV Soundness Theorem.) *Let $\Psi \in PREQ(\underline{\Sigma}, \underline{X})$ and let $\Sigma' = \Sigma \bigcup \mathcal{F}$ wherein $\mathcal{F}$ is defined as in Definition 50. Also let $A \cong I(\Sigma', E)$ wherein $E = EQCON(\Psi)|_{\Sigma'}$ and let $\Gamma \subseteq \Sigma'$ be inductive for A.*

If $\Phi \in ASTRAL(\underline{\Sigma'}, \underline{X})_{u,v}$ and $\Phi' \in ASTRAL(\underline{\Sigma'}, \underline{X})_{u',v}$ are defined such that Φ and Φ' do not share any symbols other than those in $\underline{\Sigma'}$ and $\underline{X}$ then

$$AV^{\Psi,\Gamma}(\Phi,\Phi') = tt \Longrightarrow [\![\Phi]\!]_{\underline{A}} = [\![\Phi']\!]_{\underline{A}}.$$

Proof. Let $E^{\Phi,\Phi',\Psi}$, $R^{\Phi,\Phi',\Psi}$, f_i and g_i for $i = 1,\ldots,|v|$ be defined as in Definition 109 and let $\mathbb{E} = \mathrm{EQCON}(E^{\Phi,\Phi',\Psi}) \subseteq \mathrm{EQ}(\underline{\Sigma''}, \underline{X})$. First, notice that (using $\mathbb{R}$ to denote $R^{\Phi,\Phi',\Psi}$) by Lemma 49 $e_i = \tau_i = \tau_i'$ for $j = 1,\ldots,|v|$ is weakly decidable with respect to $\mathbb{R}$. Also notice that $\mathbb{R}$ is stream variable reducing by Lemma 56. Therefore by Theorem 16

$$\mathrm{EVER}^{\Gamma}(\mathbb{R}, \tau_i, \tau_i') = tt \Longrightarrow \mathrm{EQWIL}^{\Gamma,\mathbb{E}} \vdash f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x_1', \ldots, x_{|u'|}').$$

for $j = 1,\ldots,|v|$. Also notice that as by definition $\underline{\Sigma'} \subseteq \underline{\Sigma''}$, $E \subseteq \mathbb{E}$, $\Gamma \subseteq \Sigma'$ is inductive for A and $A \cong \mathrm{I}(\Sigma', E)$ by Lemma 50 we have

$$\mathrm{EQWIL}^{\Gamma,\mathbb{E}} \vdash f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x_1', \ldots, x_{|u'|}')$$

$$\Longrightarrow$$

$$\mathrm{I}(\underline{\Sigma''}, \mathbb{E}) \models f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x_1', \ldots, x_{|u'|}').$$

Moreover, as $\mathrm{I}(\underline{\Sigma''}, \mathbb{E})|_{\Sigma'} \cong \mathrm{I}(\Sigma', E)$ by Lemma 37 we have $\underline{A} \cong \mathrm{I}(\underline{\Sigma''}, \mathbb{E})$ and hence

$$\underline{A} \models f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x_1', \ldots, x_{|u'|}').$$

Recall now that by hypothesis for $j = 1, \ldots, |v|$ the symbols f_i and g_i are the function symbols representing the co-ordinate functions of the Cartesian forms of F and G respectively. Consequently,

$$\underline{A} \models f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x'_1, \ldots, x'_{|u'|}).$$

for $i = 1, \ldots, |v|$ implies that

$$\underline{A} \models F(x_1, \ldots, x_{|u|})(x) = G(x'_1, \ldots, x'_{|u'|})(x);$$

that is,

$$(\forall a \in \underline{A}^u)\,(\forall a' \in \underline{A}^{u'})\,(\forall t \in T) \quad F^{\underline{A}}(a)(t) = G^{\underline{A}}(a)(t).$$

However, by hypothesis $F^{\underline{A}} = [\![\Phi]\!]_{\underline{A}}$ and $G^{\underline{A}} = [\![\Phi']\!]_{\underline{A}}$ and therefore we have

$$\mathrm{AV}^{\Psi,\Gamma}(\Phi, \Phi') = tt \Longrightarrow [\![\Phi]\!]_{\underline{A}} = [\![\Phi']\!]_{\underline{A}}$$

as required.

□

Discussion. Notice that it is the combination of Lemma 59 and Theorem 18 that establish formally that AV is indeed a general purpose proof assistant in the context of primitive recursive STs. However, as we promised we can also show formally that in the context of restricted, but still useful classes of correctness statements AV is also a total proof tool. The implications of these results are discussed in the following section.

AV as a Total Proof Tool.

Notation 5. Let Ψ, $\underline{\Sigma'}$ and Γ be defined as above. Also let $E = \mathrm{EQCON}(\Psi)$ and let

$$E^{\Phi,\Phi',\Psi} = (E^{\Phi} \uplus_1 (E^{\Phi'} \uplus_1 \Psi))).$$

We write

$$\mathrm{EQWIL}^{\Gamma, E^{\Phi,\Phi',\Psi}} \vdash \Phi = \Phi'$$

to mean formally that for $i = 1, \ldots, |\underline{v}|$

$$\mathrm{EQWIL}^{\Gamma, E^{\Phi,\Phi',\Psi}} \vdash f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x'_1, \ldots, x'_{|u'|})$$

wherein f_i and g_i are the symbols from $E^{\Phi,\Phi',\Psi}$ representing the co-ordinate function of the Cartesian forms of the functions F and G represented by Φ and Φ' respectively and $x \in X_{\mathbf{t}}$ and $x_j \in \underline{X}_{u_j}$ for $j = 1, \ldots, |u|$ and $x'_l \in \underline{X}_{u'_j}$ for $l = 1, \ldots, |u'|$ are some distinct variable symbols.

Theorem 19. *Let $\Psi \in PREQ(\Sigma, X)$, let $\Sigma' = \Sigma \bigcup \mathcal{F}$ wherein is defined in Definition 50 and let $\Gamma \subseteq \Sigma'$. Also let $E = EQCON(\Psi)$, let*

$$E^{\Phi,\Phi',\Psi} = (E^{\Phi} \uplus_1 (E^{\Phi'} \uplus_1 \Psi)))$$

and for $i = 1, \ldots, |\underline{v}|$ *let* f_i *and* g_i *be the function symbols from* E *representing the co-ordinate functions of the Cartesian forms of* Φ *and* Φ' *respectively. If* $\Phi \in ASTRAL(\underline{\Sigma'}, \underline{X})_{u,\underline{v}}$ *and* $\Phi' \in ASTRAL(\underline{\Sigma'}, \underline{X})_{u',\underline{v}}$ *are defined such that* Φ *and* Φ' *do not share any symbols other than those in* $\underline{\Sigma'}$ *and* $\underline{X}$ *and* $x_j \in \underline{X}_{u_j}$ *for* $j = 1, \ldots, |u|$ *and* $x'_l \in \underline{X}_{u'_j}$ *for* $l = 1, \ldots, |u'|$ *are defined as in Notation 5 and the equations*

$$f_i(x, x_1, \ldots, x_{|u|}) = g_i(x, x'_1, \ldots, x'_{|u'|})$$

for $i = 1, \ldots, |v|$ *satisfy any of the criteria in Cases (A) to (D) of Theorem 17 then*

$$AV^{\Psi,\Gamma}(\Phi, \Phi') = tt \Longleftrightarrow EQWIL^{\Gamma, E^{\Phi,\Phi',\Psi}} \vdash \Phi = \Phi'$$

Proof. First, notice that by Theorem 18 we immediately have

$$\mathrm{AV}^{\Psi,\Gamma}(\Phi, \Phi') = tt \Longrightarrow \mathrm{EQWIL}^{\Gamma, E^{\Phi,\Phi',\Psi}} \vdash \Phi = \Phi'$$

and therefore it is sufficient to show the converse. This follows immediately by Theorem 17.

□

7.5 Discussion: Verifying SCAs and Hardware

Recall from Section 3.10 that by definition every SCA can be represented by a primitive recursive function. Also, recall that SCAs encompass several broad and useful classes of computational systems that are used in computer science including: artificial neural networks; cellular automata; dynamical systems and of course a large class of hardware that is one of our main interests. As such the implications of the fact that VER, EVER and AV behave (at worst) as total proof assistants in the context of primitive recursive sets of equations is clear. However, the forms of Theorems 15 and 17, and Theorem 19 that depends on Theorem 17 are by necessity technical in nature. Therefore, it is not immediately obvious what the specific practical implications of the abstract functions VER, EVER and AV may be in the context of their use as total proof tools. As such, in this final section we discuss how we may make use of these functions' properties in the context of SCAs and more specifically in the context of hardware devices when expressed as STs.

7.5.1 Hardware: the Practical Implications of Theorem 19

In highlighting the practical implications of Theorem 19 we focus on the case that the ST to be verified satisfies the criteria in Case (D) of Theorem 17.

Essentially what Case (D) states is that if we only consider hardware devices with finite state and that only receive control signals (streams) whose point-wise values are taken from a finite set then, relative to using equational logic and induction as a proof technique, the equivalence of a device's implementation and its specification under initial algebra semantics is decidable.

From a practical perspective the class of hardware devices that satisfy these two criteria

are very broad and include all standard microprocessors and most other basic hardware devices including our running example the RS-Flip-Flop (see Section 8.4.1). Indeed, we would imagine that the only hardware devices that do not satisfy these criteria would be either analogue devices or devices with analogue components. Therefore, it would appear that Theorem 19, implemented using the function AV, provides the most straightforward and general purpose method of verifying hardware represented as an ST that we can reasonably expect. More specifically, if for no other reason then we believe the implications of this result justify the approach to stream processing that we advocate in this thesis.

Of course having made such a strong statement we must be careful to quantify it. In particular, we ourselves admit that even theoretical results specifically tailored for their practical implications do not necessarily guarantee that they will provide usable software tools. Indeed, as we discuss in Section 8.5 it by no means trivial to design a practical implementation of AV that is suitable for use on devices of 'real world' complexity. Rather, what we claim is that the function AV provides a sound theoretical basis for automated verification tools in the sense that in principle it reduces the problem of verifying hardware to developing a usable implementation of the function AV.

7.5.2 The Practical Implications of Theorem 15

To conclude this chapter we highlight one practical implication of Theorem 15 outside of the context of hardware devices.

Cellular Automata. The abstract computational devices known as cellular automata (see von Neumann [1966] and Codd [1968]) are finding increasing use in non-linear science applications (see Farmer *et al.* [1989] and Gutowitz [1990]) including the study of chaotic and biological systems. Of the four types of computational device that we have mentioned that are encompassed by SCAs the class of cellular automata are distinct in the sense that most examples that are found in the literature are based on what is referred to as *closed computation* in Thompson and Tucker [1994]; that is, apart from a 'clock tick' and some preset initial values (initial state) they do not require any input to generate their output.

More formally, a typical cellular automaton can be specified as a function of the form

$$C : T \times A^u \to A^u;$$

that is, as an SCA without stream inputs, wherein $u \in S^+$ codes the device's initial state, and also codes any subsequent states that are computed from the current time and the current state as described in Section 3.9.3. Moreover, and most importantly in the context of this chapter, a cellular automaton's state (A^u) is by definition comprised of a finite number of configurations and hence can be formalized as elements from finite carriers.

Therefore, an equational correctness statement relating the equality of two cellular automata is of precisely the form to satisfy Case (D) of Theorem 15 and so we can deduce the following useful fact.

Corollary 6. *Let A be some standard S-sorted Σ-algebra, let X be some S-indexed collection of variable symbols such that Σ and X are pairwise disjoint, and let $\Gamma \subseteq \Sigma$ be inductive for A.*

Also let $\Phi \in PREQ(\Sigma, X)$ wherein Φ includes the definition of two cellular automata represented by the function symbols C and C' both of type $(\mathbf{t}\, u, u)$ for some $u \in S^+$.

If $E = EQCON(\Phi)$, $A \cong I(\Sigma, E)$, and $x \in X_{\mathbf{t}}$ and $x_i \in X_{u_i}$ for $i = 1, \ldots, |u|$ are some distinct variable symbols then if we wish to show that

$$I(\Sigma, E) \models C(x, x_1, \ldots, x_{|u|}) = C'(x, x_1, \ldots, x_{|u|})$$

then equational logic and induction provide a decidable calculus.

Proof. We prove the case where $|u| = 1$; that is, where $u = s \in S$ and leave the case where $|u| > 1$ to the reader.

First, notice that under the hypothesis that $\Gamma \subseteq \Sigma$ is inductive for A and that $A \cong \mathrm{I}(\Sigma, E)$ by Theorem 13

$$\mathrm{EQWIL}^{\Gamma, E} \vdash C(x, x_1, \ldots, x_{|u|}) = C'(x, x_1, \ldots, x_{|u|})$$

$$\Longrightarrow$$

$$\mathrm{I}(\Sigma, E) \models C(x, x_1, \ldots, x_{|u|}) = C'(x, x_1, \ldots, x_{|u|}).$$

Therefore, as by hypothesis C and C' represent cellular automata the equation $C(x, x_1, \ldots, x_{|u|}) = C'(x, x_1, \ldots, x_{|u|})$ is precisely of the form to satisfy Case (D) of Theorem 15 and hence

$$\mathrm{I}(\Sigma, E) \models C(x, x_1, \ldots, x_{|u|}) = C'(x, x_1, \ldots, x_{|u|})$$

is decidable in the sense defined above as required.

□

Chapter 8

Implementing a Proof Tool for STs: a Case Study

Machines are worshiped because they are beautiful, and valued because they confer power; they are hated because they are hideous, and loathed because they impose slavery.

Bertrand Russell

8.1 Overview

In this chapter we discuss the implementation of the function AV and its application to a small case study: the RS-Flip-Flop, that we have used as our running example. This provides an opportunity to demonstrate AV's effectiveness as a total proof tool.

8.1.1 Simulating a Full Implementation

Recall that the definition of AV requires the development of several large and complex pieces of software that include: an ASTRAL parser, a PREQ parser, and a PR parser. Moreover, as we will discuss later, there are several practical considerations that at present limit the capabilities of a full implementation of AV. As a consequence, what we describe in this chapter essentially amounts to the development of two of the necessary constituents of AV: the function EVER and the compiler $\mathcal{C}$, the combination of which is sufficient to demonstrate the effectiveness of a complete and efficient implementation.

After some general comments regarding the development of the software, in Section 8.2 we begin a more detailed discussion with a description of the input that must be supplied to the implementation of EVER. This is followed in Section 8.3 by a description of the three phases of the software's operation. In Section 8.4 we discuss the implementation of the compiler $\mathcal{C}$.

We conclude the chapter with an indication of the practical difficulties that we face in designing an efficient implementation of AV suitable for the verification of large systems, and suggests some methods that can be used to overcome these problems.

8.1.2 General Comments

The implementation of the function EVER and the compiler $\mathcal{C}$ has been developed on a *SUN SPARC station 2* running *UNIX version 4.1.3* under *Open Windows version 4.1.1*.

This software has been programmed using the *Sun cc* C compiler and the *Berkeley C Shell* programming language. The combination of these two applications has provided a fast prototyping environment and has made possible the re-use of previously developed software as we will describe. A schematic representation of our implementation is shown in Figure 8.1.

8.2 Input

The correct operation of the implementation of EVER relies on the user providing the following input to the system:

(1) An equational specification E that when orientated as left-to-right re-write rules forms a complete TRS.

(2) A description of the signature of constructors Γ.

(3) An equation of the form $f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_m)$ wherein f and g are function symbols occurring in E and t_i and t'_j are some terms of interest for $i = 1, \ldots, n \in \mathbb{N}$ and for $j = 1, \ldots, m \in \mathbb{N}$ respectively.

We now discuss each of these requirements in more detail using our case study the RS-Flip-Flop as an example.

8.2.1 Implementing an Equational Specification Language

The development and implementation of a concrete language for the representation of equational specifications forms an integral part of any implementation of EVER. As at the time of writing no implementation of a PREQ parser exists the particular language that we have used for this purpose is based directly on the language *EQ* that has been implemented by the author as part of a previous project (see Stephens [1991]). The relevant features of EQ with respect to the implementation of EVER are as follows:

(A) EQ provides a general purpose syntax suitable for the presentation of weak second-order equational specification and hence for the specification of primitive recursive equational specifications in Cartesian form. In particular, EQ is suitable as a target language for an implementation of the ASTRAL compiler χ^{ASTRAL}.

(B) We can compile EQ specifications into an equivalent TRS suitable for input into a first-order version of the *ATLAS*system (see Hearn [1994]) that provides an efficient and flexible implementation of a first-order term re-writing engine.

(C) The combination of EQ and ATLAS provides an implementation of essentially all of the constituent functions of EVER with the exception of some very low-level operations. Moreover, these low-level operations performed by EVER that are not implemented by EQ and ATLAS can be readily programmed using C shell script. While this reduces the efficiency of the overall system it is adequate as a demonstration prototype and has considerable reduced development time.

Example EQ Programme. The RS-Flip-Flop specification can be represented in Cartesian form in EQ as follows: (notice in the discussion that follows because we are using the syntax of actual implementation languages that both 'Eval' and 'eval' are used to represent the basic stream operation *eval* and should not be confused with the actual function *Eval* discussed in Section 4.5.6.

```
SIGNATURE OF RSFlipFlop IS
   SORTS
      nat, bool, _bool
   END_SORTS
   OPERATORS
      FFlopSpec : nat * _bool * _bool -> bool ;
   END_OPERATORS
END_SIGNATURE
EQUATIONS
VAR T : nat ;
VARS  S1, S2 : _bool ;
```

```
VARS  X1, X2 : bool ;
   FFlopSpec( 0, S1, S2 ) = False ;
   FFlopSpec( Succ( T ), S1, S2 ) =
      True
         IF And(Eq{bool}(Eval(T,S1),False),Eq{bool}(Eval(T,S2),True)),
      False
         IF And(Eq{bool}(Eval(T,S1),True),Eq{bool}(Eval(T,S2),False)),
      False
         IF And(Eq{bool}(Eval(T,S1),True),Eq{bool}(Eval(T,S2),True)),
      FFlopSpec( T, S1, S2 )
         OTHERWISE ;
END_EQUATIONS
```

Using an example EQ programme we can also highlight three more specific features that are relevant to our discussion.

(D) Aside from its use as part of key words, the underscore ('_') is prefixed to an existing sort name in EQ to denote a stream carrier.

(E) Although we have explicitly included a sort declaration section in our example, EQ provides features to automatically include corresponding stream sorts for each sort declared and also allows the usual constants and operations associated with a particular sort to be included automatically. For example, the operation '*Eval*' is automatically included for each sort as is '*Eq*' (equality), '0', '*Succ*', '*True*' and '*False*'. However, as EQ was not intended to be a direct user-interface notation, notice that some of these operations must be postfixed with sort information '{sort}' as EQ does not allow overloading.

(F) Notice that EQ provides the facility to specify functions using case statements. These case statements are eliminated using definition-by-cases when an EQ specification is compiled into re-write rules (see Sections 4.2.3 and 6.7.1). In particular, notice that each case statement must include an 'otherwise clause' and hence EQ is strictly limited to proper equational specification; that is, this facility is nothing more than a syntactical convenience.

As we will see in later examples, EQ also provides the facility for local variable name declaration that allows the equations section to be divided in a modular fashion. This feature makes EQ a good target language for ASTRAL specifications that also use local variables (see Section 6.7). Indeed, we note in passing that our practical experience suggests that local variable declaration and Features (E) and (F) are very convenient from the perspective of the user and would be included in an implementation of the language PREQ.

8.2.2 Describing Γ and Specifying the Equation to be Verified

In the development of a prototype of EVER in order that we may supply the necessary information concerning the signature of constructors Γ we have found it useful to slightly modify the original implementation of EQ to derive the language *VEQ*. In particular, a VEQ programme is

an EQ programme followed by an additional section containing a description of Γ together with the details of the equation that we wish to verify. For example, in the previous EQ programme if we had also included the equations representing the RS-Flip-Flop implementation in Cartesian form then the VEQ programme necessary to verify the correctness of this implementation is the EQ programme itself followed by this additional section:

```
VERIFY
   SORT_INFO
      nat : INFINITE : {0,Succ} ;
      bool : FINITE : {True,False} ;
   END_SORT_INFO
   VAR T : nat ;
   VARS S1, S2 : _bool ;
   VARS X1, X2 : bool ;
   FFlopImp( T, S1, S2, X1, X2 ) = FFlopSpec( T, S1, S2 ) ;
END_VERIFY
```

As the language VEQ is not intended to be a direct user-interface language and given the straightforward nature of this additional section we simply note at this point that our choice of syntax is sufficient to code both the structure of any Γ and any equation e that are appropriate for use with the abstract function EVER.

8.3 The Operation of the EVER Implementation

With the three necessary user inputs supplied to our software using a VEQ specification, the operation of the EVER implementation is fully automatic and behaves either as a total proof assistant or as a total proof tool depending on the syntactic structure of the equation e to be verified (see Section 7.1).

The implementation's operation is divided into essentially three modes of operation:

(1) Generating the Necessary Input for the ATLAS System.

(2) Simulating an EVER Deduction.

(3) Automatically Typesetting a Proof in TEX.

We describe each of these system modes in more detail:

8.3.1 Generating the Necessary Input for the ATLAS System

The generation of the necessary input to the ATLAS system is carried out by the VEQ compiler and is basically an 'information generation phase' that is further sub-divided into several operations. However, most of these sub-functions are simply to create files to provide typing information and lists of name substitutions to eliminate name-mangled identifiers during proof generation (Mode (3)). Therefore, we will only discuss the most important operations performed during this mode of execution omitting unimportant details.

The VEQ compiler first analyses the structure of the equation e to be verified. More specifically, it identifies the distinct variables occurring in e and produces a number of files as output.

These files hold particular sub-sets of the set of all equations derivable by all possible ground-term substitutions of all variables in e using appropriate members of Γ. For example, in the context of the RS-Flip-Flop the VEQ compiler produces eight equations to be input to ATLAS:

```
fflopimp(0,cs15,cs25,true,true) = fflopspec(0,cs15,cs25)
fflopimp(0,cs15,cs25,false,true) = fflopspec(0,cs15,cs25)
fflopimp(0,cs15,cs25,true,false) = fflopspec(0,cs15,cs25)
fflopimp(0,cs15,cs25,false,false) = fflopspec(0,cs15,cs25)
fflopimp(succ(ct5),cs15,cs25,true,true) = fflopspec(succ(ct5),cs15,cs25)
fflopimp(succ(ct5),cs15,cs25,false,true) = fflopspec(succ(ct5),cs15,cs25)
fflopimp(succ(ct5),cs15,cs25,true,false) = fflopspec(succ(ct5),cs15,cs25)
fflopimp(succ(ct5),cs15,cs25,false,false) = fflopspec(succ(ct5),cs15,cs25)
```

wherein these equations are divided into two files containing the first and last four equations respectively. We now explain the significance of these sub-sets in more detail.

Complimentary Sub-sets. Notice that in the equations above created to verify the RS-Flip-Flop the variable T of sort *nat* has been replaced in the first four equations by 0 and in the last four equations by *succ(ct5)* wherein *ct5* is a name-mangled implementation of the constant χ. (Indeed, we note in passing that the constant *ct5* along with any other new constants is also automatically added to the original signature when the equations from the VEQ specification are converted into a TRS.) These particular substitutions of the variable T are a direct result of the description of the structure of Γ in the verify section of the VEQ programme, that specifies that *nat* is to be interpreted by a countably infinite carrier (N is this particular example).

The consequence of this fact is that (in the context of the RS-Flip-Flop) as there is only one variable of type *nat* (variable T) we derive what we refer to as two *complementary sub-sets* of equations; that is, the first and last four equations form two complimentary sub-sets because equation e_i differs from e_{i+1} for $i = 1, \ldots, 3$ by one substitution that essentially constitute different cases in a proof by case analysis; and e_i differs from e_{i+4} for $i = 1, \ldots, 4$ by one substitution that essentially constitute the basis case and an induction case of a proof by induction. Thus, we have divided the eight equations generated by VEQ into two sub-sets representing the basis case and induction case of a proof by induction respectively, and both steps in this proof require four specific cases to be analysed. Indeed, for this reason in the sequel we will refer to the *basis sub-set* and the *induction sub-set* of two complimentary sub-sets of equations with the obvious meaning. In addition, the specific pair of equations from a basis sub-set and an induction sub-set that differ by only one substitution are referred to as a *specific complement.*

More generally, outside of the context of our case study, if $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, m$ are the number of distinct variables (from left-to-right) occurring in an equation e and $\mathcal{I} = \{l_1, \ldots, l_k\} \subseteq \{1, \ldots, m\}$ is defined such that for each $j \in \{1, \ldots, k\}$ we have $l_j \in \mathcal{I} \iff \Gamma_{s_{l_j}} = \{b_{s_{l_j}}, g_{s_{l_j}}\}$ for some constant $b_{s_{l_j}}$ and for some unary operation $g_{s_{l_j}}$ then VEQ will create $|\mathcal{I}|$ pairs of complimentary sub-sets. In particular, if $C^B_{l_j}$ and $C^I_{l_j}$ are the basis sub-set and inductive sub-set respectively of the l_jth complimentary sub-sets then $C^B_{l_j} \supset C^B_{l_{j+1}}$ and $C^I_{l_j} \supset C^I_{l_{j+1}}$ for $j = 1, \ldots, k-1$. For example, if the equation to be 'verified' were $Add(x, y) = Mult(y, x)$ and Γ is defined as in our case study then VEQ would create 4

equations:

$$e_1 \stackrel{def}{=} Add(0,0) = Mult(0,0),$$

$$e_2 \stackrel{def}{=} Add(0, Succ(\chi_2)) = Mult(Succ(\chi_2), 0),$$

$$e_3 \stackrel{def}{=} Add(Succ(\chi_1), 0) = Mult(0, Succ(\chi_1))$$

and

$$e_4 \stackrel{def}{=} Add(Succ(\chi_1), Succ(\chi_2)) = Mult(Succ(\chi_2), Succ(\chi_1))$$

that are divided into two pairs of complementary sub-sets: $C_1^B = \{e_1, e_2\}$, $C_2^B = \{e_2\}$, $C_1^I = \{e_3, e_4\}$ and $C_2^I = \{e_4\}$.

In general, the creation of the entire set of ground-term instantiations of the variables in the equation e to be verified is equivalent to an 'un-winding' of the recursive abstract definition of EVER, that itself implements the successive applications of Rules (v) and (vi) in the calculus EQWIL. Moreover, the identification the complementary sub-sets of these equations, has several important practical implications:

(1) First, from the perspective of efficiency, if we use ATLAS to re-write both sides of each equation in the smallest basis sub-set $C_{l_k}^B$ first, and no common normal forms are found, then we already know our proof has failed – as the basis case of a necessary induction has failed. We return to this point in our algorithmic description of the operation of our implementation of EVER (see Section 8.3.2).

(2) Secondly, the identification of the individual cases in complementary sub-sets enables us to present a proof based on our our formal deduction that is structured in a more natural semantic style (also see Section 8.3.2).

(3) Thirdly, as we will discuss in detail in Section 8.3.4 we can use complementary sub-sets to infer existential quantification over variables defined over finite carriers.

(4) Finally, we note that the verification of the individual cases in complementary sub-sets can be performed completely independently and hence makes the application of parallel techniques during a verification very straightforward. We return to this point in the final section of this chapter.

The Total Number of Equations Created by VEQ. Returning to our case study, notice that as *bool* is to be interpreted by a finite carrier with two elements ($\mathbb{B}$ in this case), the variables $X1$ and $X2$ have been replaced by combinations of the constants *true* and *false*. However, also notice that throughout the eight equations the stream variables $S1$ and $S2$ of type *_bool* have been replaced with the name-mangled constants *cs15* and *cs25* respectively. This is a consequence of the fact that (mirroring our theoretical assumption that $\Gamma_{\underline{s}} = \varnothing$ for each $s \in S$ – see Sction 7.2.2) the implementation assumes that *_bool* is to be interpreted by an uncountable carrier ($[T \rightarrow \mathbb{B}]$) and hence cannot be finitely generated. In particular, this is the reason that we have $8 = 2^3$ equations to be passed to ATLAS and not $32 = 2^5$ equations as we would have had, for example,

if $S1$ and $S2$ were of type *bool*. Indeed, in general for some quation e the number of equations $n_e \in \mathbb{N}$ to be verified that are passed by VEQ to ATLAS is

$$n_e = |\Gamma_{s_{i_1}}| \times |\Gamma_{s_{i_2}}| \times \cdots |\Gamma_{s_{i_k}}|$$

wherein if as before $x_i \in X_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, m$ are the number of distinct variables occurring in e then $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, m\}$ are the indexes of the variables such that $\Gamma_{s_{i_j}} \neq \emptyset$ for each $j \in \{1, \ldots, k\}$. In particular, the replacement of the stream variables $S1$ and $S2$ with constants in our case study is simply to ensure every equation created is defined using ground terms. This is to avoid possible complications that can arise with open terms during the term re-writing process.

We also mention in passing that the change in case of the function and constant names in the equations generated from the original VEQ specification is to accommodate the naming conventions supported by ATLAS.

Having created all the necessary information concerning the syntactic structure of e and the complementary sub-sets, the implementation begins its second phase of operation that simulates the operation of EVER.

8.3.2 Simulating a EVER Deduction: an Overview

The specific way in which EVER is simulated using ATLAS has been strongly influenced by the desire to automatically generate a readable proof. In particular, the mechanism by which the abstract functions VER and EVER verify the correctness of an equation is quite different from the standard proof techniques that would typically be applied in a verification done 'by hand'. Therefore, the implementation of EVER has been structured so that it reflects more closely the structure of a deduction in the calculus EQWIL with the intention that this will give a more naturally structured output (proof).

In more detail, by generating all possible ground-term instantiations of the equation to be verified and by further identifying all complimentary sub-sets of these equations, it is possible to generate a number of separate proof scores representing particular deductions about individual ground-term equations; that is, within each complementary sub-set each specific equation represents a particular case of a proof by case analysis; and each pair of complementary sub-sets represents the basis case and induction step of a proof by induction. Therefore, each individual proof score that is generated by verifying a particular ground-term equation can be linked together to form a overall proof structured in a more semantic style. Most specifically, a proof that is very close in structure to a proof done 'by hand' in that it mirrors an essentially semantic use of the deduction rules of EQWIL.

We argue that the advantage of this approach is that such a proof is much more readily accessible to human (and machine) verification. Moreover, while the theorems of the previous chapter guarantee the correctness of the abstract functions VER and EVER, there is no such guarantee of correctness for our implementation. Therefore, we can significantly increase our confidence in an automatic verification if we have available a readable proof score of the deduction. Indeed, we argue that the generation of such a readable proof provides the maximum

degree of confidence that can be reasonably expected from any implementation of such tools that have not themselves been formally verified.

8.3.3 An Algorithmic Description of the EVER Implementation

Essentially, the software that implements EVER consists of a number of C shell scripts that take their input from the files created by VEQ: the files containing all the pairs of complimentary sub-sets of the equations created by VEQ; a file containing the TRS created from the original EQ specification that is suitable as input to ATLAS; and several files containing information concerning variable typing and any identifier re-naming that has been necessary. At a very high level the operation of the this software is as follows.

Let $C^B_{l_j}$ and $C^I_{l_j}$ for $j = 1, \ldots, k$ be the pairs of complementary sub-sets as defined in Section 8.3.1, recalling in particular that $C^B_{l_j} \supset C^B_{l_{j+1}}$ and $C^I_{l_j} \supset C^I_{l_{j+1}}$ for $j \in \{1, \ldots, k-1\}$ and hence by removing an equation from $C^I_{l_{j+1}}$ (say) that we are also removing an equation from $C^I_{l_j}$. Also, let R be the TRS created by VEQ in a format that is suitable as input to ATLAS.

BEGIN

(A) Set $j = 2k + 1$.

(B)

- **(a)** Set $j = j - 1$.
- **(b)** If $j = 0$ then typeset a proof and STOP.
- **(c)**
  - **(I)** If $j > k$ then set $C = C^B_{l_{j-k}}$.
  - **(II)** If $j \leq k$ then set $C = C^I_{l_j}$.

(C) Split each equation in C into two terms (the left-hand-side and the right-hand-side) and use ATLAS to reduce both terms to their normal forms using R. Store these normal forms along with the corresponding reduction sequences output by ATLAS.

(D)

- **(a)** If all the normal forms from Step (C) are the same then GOTO Step (B).
- **(b)** For each equation $\eta = (\tau = \tau')$ in C without matching normal forms do:
  - **(I)** If there *does not* exists a term $t = eval_s(\theta, X)$ for some $s \in S$ such that either $t \subseteq \tau$ or $t \subseteq \tau'$ then remove the equation η from C and remove η's specific compliment from C's complimentary set (see the following example).
  - **(II)** If there *does* exists a term $t = eval_s(\theta, X)$ for some $s \in S$ such that either $t \subseteq \tau$ or $t \subseteq \tau'$ then
    - **(i)** Replace each of the terms t with a new variable symbol not occurring in η to make a new equation to be verified η' .
    - **(ii)** Make a copy of the VEQ programme with the verify section modified by changing e to η' adding the information about the new variable symbols as appropriate.

(iii) Recursively call EVER with the new specification as defined above and record whether EVER was successful in proving this new equation.

(iv) For each equation η whose recursive call to EVER was *not* successful remove η from C and remove η's specific compliment from C's complimentary set.

(c) If $C = \varnothing$ then output a diagnostic indicating that the verification has failed and STOP.

(d) If $1 < j < k$ then for each equation in C add an appropriate re-write to R representing the corresponding induction hypothesis for the 'basis case' that has just been proved (again see the following example).

(E) GOTO Step (B).

END

Example 22. We continue our use of the RS-Flip-Flop as a case study. In particular, we use the full VEQ specification representing both the Flip-Flop specification and the Flip-Flop implementation (that unfortunately cannot be included as it is too long). A discussion of how the equations representing the Flip-Flop implementation were generated can be found in the next section.

Recall that in this case the equation e to be verified is

```
FFlopImp( T, S1, S2, X1, X2 ) = FFlopSpec( T, S1, S2 ).
```

Therefore, in this particular case we have $m = 5$, $\{x_1, \ldots, x_m\} = \{T, S1, S2, X1, X2\}$, $k = 1$, $\mathcal{I} = \{j_1\} = \{1\}$, $C_1^B = \{e_1, \ldots, e_4\}$ and $C_1^I = \{e_5, \ldots, e_8\}$ wherein e_i for $i = 1, \ldots, 8$ are the equations generated by VEQ as defined previously in Section 8.3.1. Notice in the following description of the operation of EVER that the system automatically deduces that the Flip-Flop implementation is only correct if the values of variables X1 and X2 are *tt* and *ff* respectively; that is, our implementation automatically deduces an appropriate existential quantification on these variables.

(1) At Step (A) we set $j = 2k + 1 = 3$.

(2) At Step (B.c.I) we set $C = C_1^B = \{e_1, \ldots, e_4\}$.

(3) At Step (C) we generate eight normal forms – the normal forms of the left- and right-hand-sides of equations $e_1, \ldots, e_4$ that we will denote nfl_i and nfr_i for $i = 1, \ldots, 4$:

nfl_1 = false, nfr_1 = true;
nfl_2 = false, nfr_2 = true;
nfl_3 = false, nfr_3 = false
and
nfl_4 = false, nfr_4 = false.

We also generate the eight corresponding reduction sequences.

(4) As $nfl_1 \neq nfr_1$ and $nfl_2 \neq nfr_2$ Step (D.a) fails.

(5) As none of the normal forms contain an occurrences of *eval* during the first and second iteration of Step (D.b) at (I) we set $C = C_1^B = \{e_3, e_4\}$ and $C_1^I = \{e_7, e_8\}$ respectively. Notice that this means that only certain instantiations of variables gave matching normal forms and hence our resulting correctness statement will contain existential quantification on the variables X1 and X2.

(6) At Step (D.e) we add the following re-write rules to R:
`fflopimp(ct5,cs15,cs25,false,true) -> fflopspec(ct5,cs15,cs25)`
and
`fflopimp(ct5,cs15,cs25,false,false) -> fflopspec(ct5,cs15,cs25)`

(7) At Step (B.a) we set $j = 1$.

(8) At Step (B.c.II) we set $C = C_1^I = \{e_7, e_8\}$.

(9) At Step (C) we generate four normal forms – the normal forms of the left- and right-hand-sides of equations e_7, e_8 that we will denote nfl_i and nfr_i for $i \in \{7, 8\}$:

$$
\begin{array}{l}
nfl_7 = nfl_8 = \\
\quad dc(\\
\quad\quad and(eq(eval(ct5, cs15), false), eq(eval(ct5, cs25), true)), \\
\quad\quad true, \\
\quad\quad dc(\\
\quad\quad\quad and(eq(eval(ct5, cs15), true), eq(eval(ct5, cs25), false)), \\
\quad\quad\quad false, \\
\quad\quad\quad dc(\\
\quad\quad\quad\quad and(eq(eval(ct5, cs15), true), eq(eval(ct5, cs25), true)), \\
\quad\quad\quad\quad false, \\
\quad\quad\quad\quad f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true) \\
\quad\quad\quad) \\
\quad\quad) \\
\quad);
\end{array}
$$

$$
\begin{array}{l}
nfr_8 = \\
\quad not(\\
\quad\quad or(\\
\quad\quad\quad eval(ct5, cs15), \\
\quad\quad\quad not(\\
\quad\quad\quad\quad or(\\
\quad\quad\quad\quad\quad f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true), \\
\quad\quad\quad\quad\quad eval(ct5, cs25) \\
\quad\quad\quad\quad) \\
\quad\quad\quad) \\
\quad\quad)
\end{array}
$$

```
    )
  )
```

and

$nfr_7 =$

```
  not(
    or(
      eval(ct5, cs15),
      not(
        or(
          f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, false),
          eval(ct5, cs25)
        )
      )
    )
  )
```

We also generate the four corresponding reduction sequences.

Stepping back from our case study for one moment, at this point we wish to re-emphasize an important point that we made in Section 7.4.2, and more specifically the point we made in Example 21 concerning the relative effectiveness of VER and EVER and ω-complete specifications. Notice that in the terms nfr_7 and nfr_8 there are sub-terms of the form

$$or(f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true), eval(ct5, cs25))$$

and

$$or(f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, false), eval(ct5, cs25))$$

respectively. In particular, notice that even with the boolean operation *or* defined as follows:

$$or(x, y) = \begin{cases} y & \text{if } x = ff, \\ x & \text{if } y = ff, \\ tt & \text{if } x = tt \text{ and} \\ tt & \text{if } y = tt \end{cases}$$

that both the above sub-terms are irreducible *without* the use of the abstract function SubEvals.

Now returning to our case study.

(10) As $nfl_7 \neq nfr_7$ and $nfl_8 \neq nfr_8$ Step (D.a) fails.

(11) As both pairs of normal forms contain occurrences of *eval* at Step (D.b.II) we recursively apply EVER to the two equations η_1 and η_2 defined respectively as follows:

$$
\begin{array}{l}
dc(\\
\quad and(eq(x, false), eq(y, true)), \\
\quad true, \\
\quad dc(\\
\qquad and(eq(x, true), eq(y, false)), \\
\qquad false, \\
\qquad dc(\\
\qquad\quad and(eq(x, true), eq(y, true)), \\
\qquad\quad false, \\
\qquad\quad f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true) \\
\qquad) \\
\quad) \\
) \\
\quad = \\
not(\\
\quad or(\\
\qquad x, \\
\qquad not(\\
\qquad\quad or(f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true), y) \\
\qquad) \\
\quad) \\
)
\end{array}
$$

and

$$
\begin{array}{l}
dc(\\
\quad and(eq(x, false), eq(y, true)), \\
\quad true, \\
\quad dc(\\
\qquad and(eq(x, true), eq(y, false)), \\
\qquad false, \\
\qquad dc(\\
\qquad\quad and(eq(x, true), eq(y, true)), \\
\qquad\quad false, \\
\qquad\quad f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, true) \\
\qquad) \\
\quad) \\
) \\
\quad = \\
not(\\
\quad or(\\
\qquad x, \\
\qquad not(
\end{array}
$$

$$or(f66c1(mult(ct5, succ(succ(0))), cs15, cs25, cs15, cs25, false, false), y)$$

)

)

).

Therefore, we now have two sub-verification to perform: that is, we now need to verify the following two equations:

$$\texttt{nfl}_7\texttt{[eval(ct5,cs15)/}x\texttt{][eval(ct5,cs25)/}y\texttt{]} = \texttt{nfr}_7\texttt{[eval(ct5,cs15)/}x\texttt{][eval(ct5,cs25)/}y\texttt{]}$$

and

$$\texttt{nfl}_8\texttt{[eval(ct5,cs15)/}x\texttt{][eval(ct5,cs25)/}y\texttt{]} = \texttt{nfr}_8\texttt{[eval(ct5,cs15)/}x\texttt{][eval(ct5,cs25)/}y\texttt{]}$$

wherein x and y are new free variables. Both of these sub-verifications essentially require a proof by case analysis wherein there are four cases (as we have two variables x and y of type *bool* in each equation). The first sub-proof is successful the second is not. Notice that this means that if the value of variable T is non-zero then the only instantiation of variables X1 and X2 for which the Flip-Flop implementation is correct is `X1 = true` and `X2 = false`.

(12) Based on failure of the first sub-verification, at Step (D.II.iv) we remove equation e_8 from $C = C_1^I$ and equation e_4 from C_1^B leaving $C = C_1^I = \{e_3\}$ and $C_1^B = \{e_7\}$. This indicates that our proof has been successful in that at least one pair of specific complements from each complimentary sub-set has given matching normal forms.

(13) At Step (B.b) we are now in a position to deduce the appropriate quantification for our correctness statement and hence typeset a proof. This process is explained in the following sections.

Discussion. To conclude this section we wish to re-emphasize at this point that in the context of equational specifications created from AV programmes, the abstract algorithm we have presented above mirrors the theoretical properties of EVER in that its deductive properties are at worst equivalent to a total proof assistant. Moreover, at best it also behaves as a total proof tool, as using the RS-Flip-Flop as a case study demonstrates. This is because the correctness statement relating the Cartesian form of the RS-Flip-Flop's specification and the Cartesian form of its implementation give rise to an equation that satisfies Case (D) of Theorem 17 (see Section 7.5).

8.3.4 Deducing Existential Quantification

Using the method we outlined in Section 8.3.2 the final mode of operation of our implementation (Mode 3) is the automatic generation of a 'semantic style' proof based on a successful formal deduction carried out in Mode (2).

While this aspect of our implementation is quite challenging from a programming perspective there is only one sub-operation performed in this mode that is of any theoretical interest: the mechanism by which we may automatically deduce the appropriate existential quantification on variables ranging over finite carriers. Therefore, we limit our discussion to this aspect of the

final stage of automatic verification. An abridged version of the proof created by our software during the verification of RS-Flip-Flop can be found in Appendix C

Existential Quantification. Taking our case study as a specific example, strictly speaking our implementation of EVER is not correct. More precisely, to achieve a successful proof notice that our abstract algorithm does not require that every equation in a basis and induction subset need be proved. Rather the algorithm requires that at least one equation in each basis and induction sub-set can be proved. Essentially, this relaxation of the manner in which the abstract function EVER makes a deduction amounts to extending the implicit universal quantification on each variable to both universal and existential quantification on variables that range over finite carriers.

In the context of hardware, allowing existential quantification is very useful as it is often appropriate to deduce that a piece of hardware is correct relative to some particular initial values that are preset before the device begins to receive input. Indeed, this is true of the RS-Flip-Flop and of many other pieces of hardware that can be expressed as SCAs (see Section 3.10).

The reason we have not implemented existential quantification at a theoretical level is that if it is allowed as part of the underlying calculus then the four standard rules of equational logic are no longer sound (see Meinke and Tucker [1992]). While this problem can be overcome by appropriately modifying the basic rules of deduction, this complicates the correspondence between the deductions carried out in this calculus and the deductions carried out by term re-writing using a TRS created from our specifications. However, by deducing existential quantification 'externally' we avoid this technical difficulty as our implementation is still based on standard equational logic.

Therefore, it remains for us to explain how we may deduce the appropriate existential quantification when all of the ground-term equations created by VEQ do not give matching normal forms. For convenience in the explanation that follows we will use the term 'quantification' to mean 'existential quantification' where this does not create any ambiguity.

An Algorithm. Recall the definitions and examples of Section 8.2.1. In particular, recall the definition of variables x_i for $i = 1, \ldots, m$ from equation e and the number of combinations of ground-term substitutions n_e that are derivable from e. Now let $\mathbb{X} = \{x_1, \ldots, x_m\}$.

First, the deduction of the correct quantification on a variable $x \in \mathbb{X}$ requires that we place some ordering on the members of $\mathbb{X}$. However, this ordering is not significant from the perspective of the quantification in the correctness statement that will be produced in the sense that changing this ordering may produce a different quantification, but all possible quantifications will be equivalent. More formally, all possible quantifications that can be produced relative to our choice of ordering will share the same *prenex normal form* representation (see for example Mendelson [1987]). Therefore, rather than simply make this ordering relative to each variable's first occurrence left-to-right in e we find it more convenient to use essentially this order, but treat stream variables and non-stream variables separately; that is, each variable is ordered as per its first occurrence left-to-right in e, but we first examine non-stream variables, making stream variables 'least significant'. Now let $(j_1, \ldots, j_m)$ be the permutation of the indexes $(1, \ldots, m)$ of the variables $\mathbb{X}$ as per this ordering. For example, if e is the correctness statement for the

RS-Flip-Flop then $\mathbb{X} = \{x_1, \ldots, x_5\} = \{T, S_1, S_2, X_1, X_2\}$, but the ordering we impose on these variables is T, X_1, X_2, S_1, S_2 and hence $(j_1, \ldots, j_5) = (1, 4, 5, 2, 3)$.

Secondly, we need to know precisely which ground-term substitutions have been made for each variable in $\mathbb{X}$ and which of these substitutions produced matching normal forms. In particular, we need to know all the combinations of ground-term equations that were generated by VEQ and which of these combinations produced matching normal forms. Now let $\mathbb{C}_i = (\varsigma_{i,1}, \ldots, \varsigma_{i,m})$ for $i = 1, \ldots, n_e$ wherein $\varsigma_{i,k}$ is the particular substitution made for variable x_k in equation e_i for $k = 1, \ldots, m$ by the VEQ compiler and let $M = \{m_1, \ldots, m_p\} \subseteq \{1, \ldots, n_e\}$ be the the indexes of these substitutions that gave matching normal forms. For example, again using the correctness statement e of the RS-Flip-Flop as an example, we have $\mathbb{C}_1 =$ (*0,cs15,cs25,true,true*), $\mathbb{C}_2 =$ (*0,cs15,cs25,false,true*), ..., $\mathbb{C}_8 =$ (*succ(ct5),cs15,cs25,false,false*) and $M = \{3, 7\}$.

In general, given $\mathbb{C}_i$ for $i = 1, \ldots, m$, the permutation $(j_1, \ldots, j_k)$ and $M = \{m_1, \ldots, m_{n_e}\}$ the algorithm to deduce the appropriate quantification $Q_i \in \{\forall, \exists\}$ for variable x_{j_i} for $i = 1, \ldots, m$ is as follows:

BEGIN

(A) Let $i = 1$, and let $\mathbb{S} = \bigcup_{j=1}^{j=|M|} \{\varsigma_{m_j, j_1}\}$.

(B)

(a) If $\mathbb{S} = \bigcup_{j=1}^{j=n_e} \{\varsigma_{m_j, j_i}\}$ then $Q_i = \forall$.

(b) If $\mathbb{S} \subset \bigcup_{j=1}^{j=n_e} \{\varsigma_{m_j, j_i}\}$ then $Q_i = \exists$.

(C) Let $\mathbb{S} = \bigcup_{j=1}^{j=|M|} \{\varsigma_{m_j, j_i}\}$.

(D) If $j_i < m$ then

(a) For each $j \in \{1, \ldots, |M|\}$ such that there exists a $\varsigma_{m_j, j_i} \in \mathbb{S}$ let

$$\mathbb{R}_j = \bigcup_{k=1}^{k=|M|} \{\varsigma_{m_k, j_{i+1}} \mid \text{such that } \exists \varsigma_{m_k, j_i} = \varsigma_{m_j, j_i}\}$$

(b) Let $\mathbb{S} = \bigcap_{j=1}^{j=|M|} \mathbb{R}_j$.

(E) Let $i = i + 1$.

(F) If $i \leq m$ then GOTO Step (B).

END

Discussion. Ignoring the special case $j_i = 1$, essentially for each $i = 2, \ldots, m$ the algorithm deduces the appropriate quantification Q_i by checking each substitution of variable $x_{j_{i-1}}$ that gave matching normal forms in the verification to see which substitutions of variable x_{j_i} occurred with that substitution - this is done by constructing the sets $\mathbb{R}_j$. If all of the possible substitutions of variable x_{j_i} occurred with each substitution of variable $x_{j_{i-1}}$ then we conclude that $Q_i = \forall$ - this will be the case if $\bigcap_{j=1}^{j=|M|} \mathbb{R}_j = \bigcup_{j=1}^{j=n_e} \{\varsigma_{m_j, j_i}\}$; otherwise if only some of the possible substitutions of variable x_{j_i} occurred with each substitution of variable $x_{j_{i-1}}$ then we conclude that $Q_i = \exists$.

Example 23. Again using the RS-Flip-Flop we show how the algorithm correctly deduces the quantification $Q_1 = \forall$, $Q_2 = \exists$ and $Q_3 = \exists$ for variables T X_1 and X_2 respectively.

(1) At Step (A) we set $\mathbb{S} = \bigcup_{j=1}^{j=2}\{\varsigma_{m_j,j_1}\} = \{0, succ(ct5)\}$.

(2) At Step (B.b) we set $Q_1 = \forall$.

(3) At Step (C) $\mathbb{S}$ is unchanged.

(4) As $\varsigma_{m_1,j_1} = \varsigma_{3,1} = 0 \in \mathbb{S}$ and $\varsigma_{m_2,j_1} = \varsigma_{7,1} = succ(ct5) \in \mathbb{S}$ at Step (D.a) we create two sets: $\mathbb{R}_1$ and $\mathbb{R}_2$ defined respectively as follows: $\mathbb{R}_1 = true$ because $\varsigma_{m_1,j_2} = \varsigma_{3,4} = true$ and $\varsigma_{m_1,j_1} = \varsigma_{3,1} = 0$ and $\mathbb{R}_2 = true$ because $\varsigma_{m_2,j_2} = \varsigma_{7,4} = true$ and $\varsigma_{m_2,j_1} = \varsigma_{7,1} = succ(ct5)$.

(5) At Step (D.b) we set $\mathbb{S} = \{true\}$.

(6) At Step (B.b) we set $Q_2 = \exists$.

(7) At Step (C) $\mathbb{S}$ is unchanged.

(8) As $\varsigma_{m_1,j_2} = \varsigma_{3,4} = true \in \mathbb{S}$ and $\varsigma_{m_2,j_2} = \varsigma_{7,4} = true \in \mathbb{S}$ at Step (D.a) we create two sets: $\mathbb{R}_1$ and $\mathbb{R}_2$ defined respectively as follows: $\mathbb{R}_1 = false$ because $\varsigma_{m_1,j_3} = \varsigma_{3,5} = false$ and $\varsigma_{m_1,j_2} = \varsigma_{3,4} = true$ and $\mathbb{R}_2 = false$ because $\varsigma_{m_2,j_3} = \varsigma_{7,5} = false$ and $\varsigma_{m_2,j_2} = \varsigma_{7,4} = true$.

(9) At Step (D.b) we set $\mathbb{S} = \{false\}$.

(10) At Step (B.b) we set $Q_3 = \exists$.

Thus, the implementation has deduced that the appropriate quantification for the correctness statement is:

$(\forall$`T`$)(\exists$`X1`$)(\exists$`X2`$)(\forall$`S1`$)(\forall$`S2`$)$ `FFlopImp( T, S1, S2, X1, X2 ) = FFlopSpec( T, S1, S2 ).`

Example 24. As a final example let us also use the RS-Flip-Flop, but assume now that the specification has been changed so that $M = \{3, 4, 7, 8\}$; that is, so that the initial values of the boolean pair (X_1, X_2) = (false, false) also gives a correct implementation. We show how the algorithm correctly deduces the quantification $Q_1 = \forall$ and $Q_2 = \forall$ for variables T and X_i respectively.

(1) At Step (A) we set $\mathbb{S} = \bigcup_{j=1}^{j=4}\{\varsigma_{m_j,j_1}\} = \{0, succ(ct5)\}$.

(2) At Step (B.b) we set $Q_1 = \forall$.

(3) At Step (C) $\mathbb{S}$ is unchanged.

(4) As $\varsigma_{m_1,j_1} = \varsigma_{3,1} = 0 \in \mathbb{S}$, $\varsigma_{m_2,j_1} = \varsigma_{4,1} = 0 \in \mathbb{S}$, $\varsigma_{m_3,j_1} = \varsigma_{7,1} = succ(ct5) \in \mathbb{S}$ $\varsigma_{m_4,j_1} = \varsigma_{8,1} = succ(ct5) \in \mathbb{S}$ at Step (D.a) we create four sets: $\mathbb{R}_1, \ldots, \mathbb{R}_4$ defined respectively as follows: $\mathbb{R}_1 = \mathbb{R}_2 = \{true, false\}$ because $\varsigma_{m_1,j_2} = \varsigma_{3,4} = true$ and $\varsigma_{m_1,j_1} = \varsigma_{m_2,j_1} = 0$ and $\varsigma_{m_2,j_2} = \varsigma_{4,4} = false$ and $\varsigma_{m_1,j_1} = \varsigma_{m_2,j_1} = 0$; and $\mathbb{R}_3 = \mathbb{R}_4 = \{true, false\}$ because $\varsigma_{m_3,j_2} = \varsigma_{7,4} = true$ and $\varsigma_{m_3,j_1} = \varsigma_{m_4,j_1} = succ(ct5)$ and $\varsigma_{m_4,j_2} = \varsigma_{8,4} = false$ and $\varsigma_{m_3,j_1} = \varsigma_{m_4,j_1} = succ(ct5)$.

(5) At Step (D.b) we set $\mathbb{S} = \{true, false\}$.

(6) At Step (B.b) we set $Q_2 = \forall$.

Thus, the implementation has deduced that the appropriate quantification for the correctness statement is:

```
(∀T)(∀X1)(∃X2)(∀S1)(∀S2)  FFlopImp( T, S1, S2, X1, X2 ) = FFlopSpec( T, S1, S2 ).
```

This concludes our explanation of the implementation of EVER. We now discuss the implementation of the Cartesian composition compiler $\mathcal{C}$.

8.4 Implementing the Cartesian Composition Compiler

8.4.1 Overview

Recall that the definition of the compiler χ^{ASTRAL} from ASTRAL into PREQ makes use of the generalized Cartesian composition compiler $\mathcal{C}$. Therefore, from the perspective of implementation, the compiler χ^{ASTRAL} makes use of PR as an intermediate representation. As a consequence and as highlighted by the RS-Flip-Flop as a case study, while an implementation of EVER does not directly rely on the implementation of the compiler $\mathcal{C}$, a demonstration of its effectiveness as a total proof assistant and total proof tool does. In more detail, if we wish to specify the RS-Flip-Flop implementation in EQ in Cartesian form then this requires composing the Cartesian forms of the RS-Flip-Flop itself and its pre- and post-processing schedules (see Section 6.7.2). This raises two practical problems: (1) the theory to effect Cartesian composition is stated in terms of PR schemes and not equations; and (2) at present there is no compiler from EQ into an implementation of PR. In order to overcome this difficulty we proceed as follows.

(A) First, we specify the RS-Flip-Flop and it pre- and post-processing schedules as CFSTs in PR as three separate schemes.

(B) Secondly, we combine these schemes into a single equivalent scheme using the implementation of $\mathcal{C}$.

(C) Thirdly, we convert this single scheme into an equivalent EQ scheme using a PR to EQ compiler that already exists (see Stephens [1991]).

(D) Finally, we combine the EQ programme representing the RS-Flip-Flop and the EQ programme representing the RS-Flip-Flop specification to produce the necessary VEQ programme that we have already discussed and used in the previous sections.

Discussion. One important point that we wish to make at this stage is that this is not an *ad hoc* method used in the context of a single example. Rather, this process mirrors the operations that will eventually be performed automatically as part of the AV implementation in the following sense: that an ASTRAL programme representing the RS-Flip-Flop's implementation

would create the required PR representation (that we have created by hand) automatically before creating the corresponding EQ (PREQ) specification (see Section 6.3.1). Therefore, this technique provides an effective test to demonstrate the implementation of EVER as it mirrors any eventual implementation of AV's use of the function EVER as a sub-programme.

Therefore, to complete this chapter we again use our case study the RS-Flip-Flop to explain each of the four steps above more precisely.

8.4.2 Implementing Cartesian Composition

In a similar fashion to the way we found it convenient to use the implementation of the language EQ to represent PREQ programmes, the implementation of the compiler $\mathcal{C}$ is also based on software previously developed by the author. In particular, the implementation of the compiler $\mathcal{C}$ is based on a PR parser and a PR to EQ compiler that was developed as part of the same project in which EQ was developed (see Stephens [1991]). Indeed, the implementation of the language PR is very similar in some respects to EQ and hence we will not describe all the details of the implementation's syntax and grammar. Rather, as before we will concentrate on the features of the language that are relevant to its use in the context of this thesis.

The programme that is used to represent the three components of the RS-Flip-Flop implementation is essentially three PR programmes presented as a single specification as follows:

```
SORTS
   bool ;
OPERATORS
  2 : -> nat ;
  Nor{bool} : bool bool -> bool ;
PROGRAMS

 <
  Eval{bool} o
    <
      Mult{nat} o < U< nat _bool _bool, 1 >; 2[ nat _bool _bool] >;
      U<nat _bool _bool, 2 >
    > ;
  Eval{bool} o
    <
      Mult{nat} o < U< nat _bool _bool, 1 >; 2[ nat _bool _bool] >;
      U<nat _bool _bool, 3 >
    >
 >

 <-( 2 = 3 )

 *( < U< _bool _bool bool bool, 3 >;
    U< _bool _bool bool bool, 4 >
```

```
 >,
 <
  o(
   Nor{bool},
   <
    o(
     Eval{bool},
     <
      U< nat _bool _bool bool bool bool bool, 1 >;
      U< nat _bool _bool bool bool bool bool, 2 >
     >
    );
    U< nat _bool _bool bool bool bool bool, 7 >
   >
  );
  o(
   Nor{bool},
   <
    U< nat _bool _bool bool bool bool bool, 6 >;
    o( Eval{bool},
       <
         U< nat _bool _bool bool bool bool bool, 1 >;
         U< nat _bool _bool bool bool bool bool, 3 >
       >
    )
   >
  )
 >
)

<-( 2 = 3 )

<
 Eval{bool} o
   <
     Div{nat} o < U< nat _bool _bool, 1 >; 2[ nat _bool _bool] >;
     U<nat _bool _bool, 2 >
   > ;
 Eval{bool} o
   <
     Div{nat} o < U< nat _bool _bool, 1 >; 2[ nat _bool _bool] >;
     U<nat _bool _bool, 3 >
```

```
    >
>


;
```

Discussion.

(A) First, notice that in the Programmes Section each of the basic operations and each of the function building tools are implemented in a straightforward way. However, two small points that we wish to make are that: (a) as formally PR uses an infinite signature, constants are postfixed with an expression of the form '[.]' wherein '.' represents the constants particular domain in the context in which it is being used; and (b) for convenience during parsing the symbol ';' is used to separate schemes in vectorizations and is also used as the programme terminator.

(B) Secondly, notice that as we indicated in this particular case the Programme Section is divided into three schemes: $\alpha_1, \alpha_2, \alpha_3$ that represent the post-scheduling function, the Flip-Flop itself and the pre-scheduling function respectively. In particular, notice that these schemes are separated by two expressions of the form '←(2 = 3)'. This indicates that each of the schemes are to be treated as CFSTs and that we want to generate a single PR scheme representing the result. More specifically, this syntax indicates the vector-valued Cartesian composition of scheme α_3 with co-ordinates two and three of α_2 'simultaneously' with the vector-valued Cartesian composition of α_2 with co-ordinates two and three of scheme α_1. The informal use of the word 'simultaneous' in this context simply means that the order in which the Cartesian compositions are to be performed is not important relative to the semantics of the resulting scheme. However, it can be very important from the perspective of efficiency relative to the size of the scheme that is created, although unfortunately a discussion of this topic goes beyond the scope of this thesis (see Point (C) below).

We also note in passing that for convenience during the design of our software the implementation of $\mathcal{C}$ is also capable of performing Cartesian composition using just specific co-ordinates of schemes that may be 'applied out of sequence'. For example, considering the schemes α_2 and α_3 in isolation for a moment, if we had separated schemes α_2 and α_3 with the expression '←(3, 2)' then this would have created the scheme representing the 'simultaneous' single-valued Cartesian composition of co-ordinate one of α_3 with co-ordinate three of α_2's domain and co-ordinate two of α_3 with co-ordinate two of α_2's domain.

(C) As the operation performed by the compiler $\mathcal{C}$ is highly technical the schemes that it creates are generally very large and highly complex even if the schemes supplied as input are straightforward. This fact combined with the inefficiency of the prototype implementation of $\mathcal{C}$ means that using the general Cartesian composition method of the compiler $\mathcal{C}$ (on which $\mathcal{C}$ is based) we cannot, for reasons of the memory required, generate the single scheme representing the RS-Flip-Flop from the programme above.

However, we can observe that the generality, and hence the complexity, of the compiler $\mathcal{C}$ is not necessary for any of the examples (of hardware) that we have encountered as part of our work. More specifically, we have designed a formal function that can test PR schema to see if the full generality of the compiler $\mathcal{C}$ is required to perform the necessary Cartesian composition, and incorporated this function into our implementation of $\mathcal{C}$. As such if the full generality of the compiler $\mathcal{C}$ is is not required to perform the Cartesian composition of two schemes then we can apply a much more efficient version of Cartesian composition that can be more readily implemented.

Unfortunately, due to considerations of the space needed to present these ideas formally, they will not be included in this thesis. However, as a practical indication of the increase in efficiency gained relative to the size of PR scheme created by the efficient $\mathcal{C}$ compiler (and hence the number of equations created when this scheme is compiled into EQ) we note that: the single PR scheme created by this efficient compilation method to represent the RS-Flip-flop when it is only composed with its pre-scheduling function (the Cartesian composition of schemes α_2 and α_3 above) is a scheme composed of 4 of constants, 10 of algebraic operations, 27 of projections, 15 compositions, 17 vectorizations and 1 primitive recursion; whereas the direct implementation of $\mathcal{C}$ creates a scheme composed of 21 of constants, 50 of algebraic operations, 233 of projections, 77 compositions, 79 vectorizations and 1 primitive recursion. Hence, in this specific example the efficient implementation of $\mathcal{C}$ is capable of creating a scheme that requires only 8% of the memory required by the general method.

8.4.3 Compiling PR into EQ

The PR to EQ compiler that we have made use of is based on the compilation technique presented in Thompson and Tucker [1991] (see Section 5.2) and hence also suffers from some limitations from the perspective of efficiency. However, in the context of the RS-Flip-Flop this is not a major problem.

We conclude this chapter with some further comments on the development of a full implementation of AV, but this time from the perspective of the overall efficiency of the software relative to its speed and memory use.

8.5 Designing an Effective Implementation of AV

In the particular context of this chapter we have chosen the RS-Flip-Flop as a case study as our experimentation with larger case studies, including the PDP8 (see Harman and Tucker [1993]), has shown that while our implementation of AV can in principle verify the correctness of many classes of hardware device automatically, developing an implementation of practical use is by no means straightforward. For example, our work with the PDP8 has shown that there are in excess of 402,000,000 individual cases that need to be checked to complete a verification! Moreover, based on our smaller case studies we estimate that the time and memory necessary to complete this verification using the current implementation of AV is approximately 3,000 years and 5,000 giga-bytes of storage respectively!

While this may at first appear to make the theoretical verification methods that we have presented in the previous chapter intractable for all but trivial examples, we note that experimentation with executing our current implementation on alternative hardware has already shown that an immediate increase of between one and two orders-of-magnitude in the speed with which a formal verification may be completed is possible. We also state that it is reasonable to expect a further increase of one order-of-magnitude in speed by improving the efficiency of the implementation of our abstract algorithms. In particular, it is reasonable to expect an increase of one order-of-magnitude in speed by eliminating the use of interpreted code (see Figure 8.1). Therefore, as in principle we can imagine that an amount of memory in the order of 5,000 giga-bytes could be made available with current technology, let us explore hypothetically the practical steps necessary to make our implementation of AV usable. However, to be realistic in our assumptions let us first take into consideration that a typical modern microprocessor will be of significantly greater complexity than the PDP8, perhaps requiring between ten times and one hundred times as many cases need to be tested (say). Hence, let us examine hypothetically the steps necessary to achieve an increase in speed of between five and six orders-of-magnitude in the combined performance of the current software and underlying hardware to derive an effective implementation of AV.

First, recall that by incorporating the two improvements to our software we have suggested above we concluded that we will immediately obtain between two and three orders-of-magnitude increase in the speed with which a verification is performed. Secondly, also recall that every case within a complimentary sub-set can be verified completely separately and hence we may readily perform each mode of operation of our software in parallel (see Section 8.3.1). Consequently, consider the situation where we perform our verification on the latest version of the *connection machine* (see Hillis [1985]) that has already been constructed and is essentially equivalent to a parallel connection of in excess of 64,000 machines that are each between one and two orders-of-magnitude faster than the machine on which our current implementation of AV was developed (see Section 8.1.2). Based on our conjectures the connection machine would make the formal verification of a current 'real-world' device possible in approximately *four hours*. Moreover, even if this speed-up was slowed by two orders-of-magnitude due to data transfer overheads and other considerations the verification of a 'real-world' device would still be possible in *two-and-a-half weeks*. Therefore, since this period of time is significantly less than the design phase of a modern microprocessor (see Stavridou [1993]) we conclude that the development of a practical implementation of AV is certainly not out of the question within the near future.

This chapter completes the development of the research agenda that we set in Chapter 3. As such we now conclude this thesis with some general observations on the work we have presented.

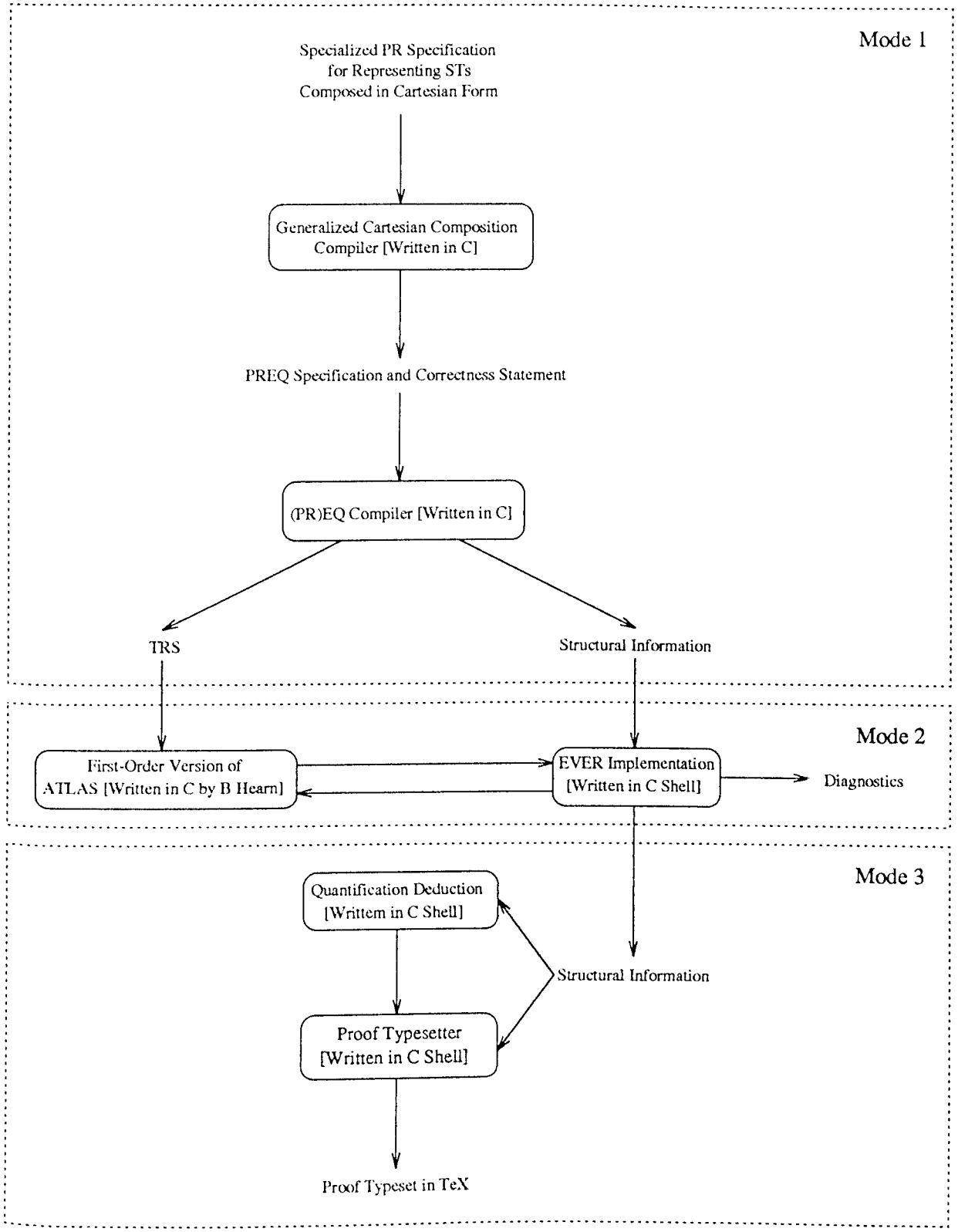

Figure 8.1: A Schematic Representation of the AV Implementation

Chapter 9

Concluding Remarks

Accuracy is the enemy of pedagogy.

Richard Feynman

Obviousness is the enemy of correctness.

Bertrand Russell

9.1 Thesis Overview

This thesis has presented the basis of an alternative and essentially first-order theory of stream processing based on algebraic techniques. In particular, this theory has developed and extended ideas taken from the theory of synchronous concurrent algorithms and generalized recursion theory based on the work of B C Thompson, J V Tucker and J I Zucker. Our main motivation has been: (1) the development of a user-friendly specification language for STs with a straightforward denotational semantics; and (2) the development of theoretical and practical tools for the automated verification of STs when expressed in this language. More specifically, the development of tools that are suitable for the automated verification of safety-critical hardware, but not limited to this application.

In Chapter 3 we identified what we believe are the weaknesses of existing approaches to stream processing. In particular, we highlighted weaknesses from the perspective of the application of automated verification techniques for STs. Moreover, we set an agenda of research that was sufficient to address each of the problems we identified.

In Chapter 4 we presented an effective solution to the first of our research problems that demonstrated formally the practical applicability of our alternative method of specifying STs in Cartesian form. We also analysed the scope and limits of Cartesian form computation.

In Chapter 5 we designed an abstract equational specification language PREQ based on the class of primitive recursive functions, and showed formally that it has desirable properties from both a practical and theoretical perspective. In more detail, we showed that PREQ can be given a straightforward denotational semantics using algebraic techniques; and that PREQ specifications can be easily converted into complete term re-writing systems.

In Chapter 6 we presented our formal specification language ASTRAL designed specifically for STs and showed that the semantics of ASTRAL can be derived using Cartesian form specification in PREQ. We also presented a prototype implementation of ASTRAL and discussed its features as a high-level programming language.

In Chapter 7 we demonstrated that by using equational logic as the basis of our formal verification techniques it is possible to identify non-trivial classes of STs that can be automatically proved correct using term re-writing techniques. In particular, we showed that in principle the correctness (in the initial model) of a broad class of hardware devices can be verified completely automatically.

Finally, in Chapter 8 we discussed the implementation of some of the theoretical tools that we have presented. We concluded the chapter by discussing certain techniques that are appropriate to increase the efficiency of our prototype software to make it suitable for the verification of complex modern electronic devices such a micro-processors.

9.2 Further Work

There are two obvious areas of the work that we have presented that require further research. These are: (in order of presentation)

9.2.1 ASTRAL

We believe the language ASTRAL has much potential as the basis of a usable high-level programming language. However, in order for ASTRAL to reach its full potential in this respect the following aspects of the implementation of ASTRAL need to be investigated more fully:

(1) From the perspective of the specification of hardware ASTRAL's BNF can be improved using further case studies as examples. In particular, we need to identify any additional primitives that would be useful for the specification of hardware devices. Some specific areas where ASTRAL is underdeveloped in this respect where mentioned in Section 6.7.

(2) The specification of a formal compiler from the BNF into abstract ASTRAL specifications is required. In particular, from the perspective of automated verification, to avoid the need to check every case individually an investigation of efficient techniques to simulate least number search with bounded least number search is required. One possible method might be partial automatic ω-enrichment of the resulting specification (see Heering [1986]).

(3) The formulation of a compiler from abstract ASTRAL specifications into a high-level programming language is required, with an emphasis on the generation of efficient code to allow the non-symbolic simulation and testing of programmes.

9.2.2 Automated Verification

The investigation of techniques to provide a usable implementation of our automated proof tool AV based on the suggestions in Section 8.5 needs further investigation. Three specific areas of research in this respect are:

(1) The investigation of parallel programming techniques for which AV is particularly suited.

(2) Techniques for identifying which abstract functions in an ASTRAL specification have caused a formal verification to fail. In general, this is by no means obvious as the automated verification is achieved using equations which in some cases may have no straightforward resemblance to the specification from which they were generated.

(3) An investigation of *term graph re-writing* techniques (see Sleep [1994]). These appear to be particularly appropriate for primitive recursive specifications and could significantly improve the efficiency of our verification techniques.

Appendix A

Proof of Theorem 9

In this first appendix we prove formally Theorem 9, concerning the correctness of the compiler $\mathcal{C}$, that we used to prove Theorem 8 in Chapter 4.

A.1 Intermediate Lemmata

To prove Theorem 9 we use the following results whose proofs we defer until Section A.3.

Lemma 60. *Let A be any standard S-sorted Σ-algebra, let $s \in S$, let ϕ be some w/u-permutation for some $w, u \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \emptyset$, let $p, r \in I^{\underline{s},w}$, and let $\beta \in \mu PR(\underline{\Sigma})_{\underline{t}\,z,s}$ for some $z \in \underline{S}^*$.*

(1) If $r = p$ then

$$(\forall t \in T)\,(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}})$$

$$[\![Eval^{w,z,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)) = [\![\beta]\!]_{\underline{A}}(t, b)$$

and

(2) If $r \neq p$ then

$$(\forall t \in T)\,(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}})$$

$$[\![Eval^{w,z,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)) = eval^{\underline{A}}_{\underline{s}}(t, a_r).$$

Lemma 61. *Let A be any standard S-sorted Σ-algebra. For each $s \in S$, for each w/y-permutation ψ for some $y, x \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \emptyset$, for each $p \in I^{\underline{s},w}$, for each $\gamma \in \mu PR(\underline{\Sigma})_{x,x'}$ for some $x, x' \in \underline{S}^+$ for each $\beta \in \mu PR(\underline{\Sigma})_{\underline{t}\,z,s}$ for some $z \in \underline{S}^*$ and for each $\gamma' \in \mu PR(\underline{\Sigma})_{y,x}$ if*

(A) for each $j \notin I^{\underline{s},x}$ we have that

$$F_j^{\gamma',\beta,\psi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}_x,$$

(ambiguously denoted $F_i^{\alpha,\beta}$) satisfies

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_j^{\gamma',\beta}(a\{p/b\}) = \left([\![\gamma']\!]_A(L(a\{p/b\}))\right)_j$$

wherein

$$L(a\{p/b\}) = (a_{\psi(1)}, \ldots, a_{\psi(\overline{\psi}(p)-1)}, \widehat{[\![\beta]\!]_A}(b), a_{\psi(\overline{\psi}(p)+1)}, \ldots, a_{\psi(|y|)});$$

and

(B) for each $j \in I^{\underline{s},x'}$ we have that

$$F_j^{\gamma',\beta,\psi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}^{\delta^{\underline{s},w,p}(z)}$$

(again ambiguously denoted $F_i^{\alpha,\beta}$) satisfies

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_j^{\gamma',\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ such that

$$(L(a\{p/b\}))_{\overline{\psi}(r)} = ([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_j$$

then if we define

$$F^{\gamma',\beta} = (F_1^{\gamma',\beta}, \ldots, F_{|x|}^{\gamma',\beta})$$

then

(1) if $i \notin I^{\underline{s},x'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad \left([\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))\right)_i = \left([\![\gamma \circ \gamma']\!]_A(L(a\{p/b\}))\right)_i$$

and

(2) if $i \in I^{\underline{s},x'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}})$$

$$\left([\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))\right)_i = \theta^{\underline{s},w,p,z,r'}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r' \in I^{\underline{s},w}$ such that

$$(L(a\{p/b\}))_{\overline{\psi}(r')} = \left([\![\gamma \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\}))\right)_i.$$

Lemma 62. *Let $\mathcal{A}$ be any standard S-sorted Σ-algebra, let $s \in S$, let ϕ be a w/w'-sort permutation over $\underline{S}$ for some w such that $I^{\underline{s},w} \neq \emptyset$ and let $p \in I^{\underline{s},w}$. Also, let $\delta \in PR(\underline{\Sigma})_{w',v'}$ for some $v' \in \underline{S}^+$, let $\beta \in PR(\underline{\Sigma})_{t\,z,s}$ for some $z \in \underline{S}^+$ and let $P : \underline{A}^{w\{p/z\}} \to \underline{A}^{w'}$ be defined by*

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad P(a\{p/b\}) = (a_{\psi(1)}, \ldots, a_{\psi(\overline{\psi}(p)-1)}, \widehat{[\![\beta]\!]}_{\mathcal{A}}(b), a_{\psi(\overline{\psi}(p)+1)}, \ldots, a_{\psi(|y|)}).$$

For each $a\{p/b\} \in \underline{A}^{w\{p/z\}}$ and for each $a' = (a'_1, \ldots, a'_{|w'|}) \in \underline{A}^{w'}$ if

$$\chi = (\chi_1, \ldots, \chi_{|w'|}) \in \underline{A}^{\Delta^{\underline{s},w,p}(w',z)}$$

is defined by

$$\chi_i = \begin{cases} a'_i & \text{if } i \notin I^{\underline{s},w'}, \text{ and} \\ \theta^{\underline{s},w,p,z,r_i}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{otherwise} \end{cases}$$

for some $r_i \in I^{\underline{s},w}$ for $i = 1, \ldots, |w'|$ and

$$\rho = (\rho_1, \ldots, \rho_{|w'|}) \in \underline{A}^{|w'|}$$

is defined by

$$\rho_j = \begin{cases} a'_j & \text{if } j \notin I^{\underline{s},w'}, \text{ and} \\ x_j & \text{otherwise} \end{cases}$$

for some $x_j \in \underline{A}^{\underline{s}}$ such that $x_j = (P(a\{p/b\}))_{\overline{\phi}(r_j)}$ for $j = 1, \ldots, |w'|$ then

(1) for each $l \notin I^{\underline{s},v'}$

$$\left([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{\mathcal{A}}}(\chi)\right)_l = \left([\![\delta]\!]_{\underline{\mathcal{A}}}(\rho)\right)_l$$

and

(2) for each $l \in I^{\underline{s},v'}$

$$\left([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{\mathcal{A}}}(\chi)\right)_l = \theta^{\underline{s},w,p,z,q_l}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $q_l \in I^{\underline{s},w}$ for $i = 1, \ldots, |w'|$ such that $(P(a\{p/b\}))_{\overline{\phi}(q_l)} = (P(a\{p/b\}))_l$ for $j = 1, \ldots, |w'|$.

For convenience we now re-state the theorem.

Theorem 9. *Let $\mathcal{A}$ be any standard S-sorted Σ-algebra. For each $s \in S$, for each $w, u \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \emptyset$, for each w/u-permutation ϕ, for each $p \in I^{\underline{s},w}$, for each $\alpha \in \mu PR(\underline{\Sigma})_{u,v}$ for some $v \in \underline{S}^+$, for each $\beta \in \mu PR(\underline{\Sigma})_{t\,z,s}$ for some $z \in \underline{S}^*$ and for each $i = 1, \ldots, |v|$:*
(1) if $i \notin I^{\underline{s},v}$ then if we define

$$F_i^{\alpha,\beta,\phi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}_{v_i}$$

(ambiguously denoted $F_i^{\alpha,\beta}$) by

$$F_i^{\alpha,\beta,\phi,w,p} = (\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_i}$$

wherein $j_i = (i - |I^{\underline{s},v_1\cdots v_{i-1}}|) + (|I^{\underline{s},v_1\cdots v_{i-1}}| * |\delta^{\underline{s},w,p}(z)|) + 1$ *then*

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha,\beta}(a\{p/b\}) = \left(\llbracket \alpha \rrbracket_A(P(a\{p/b\}))\right)_i$$

wherein

$$P : \underline{A}^{w\{p/z\}} \to \underline{A}^{u\{\overline{p}/z\}}$$

is defined by

$$P(a\{p/b\}) = (a_{\phi(1)}, \ldots, a_{\phi(\overline{p}-1)}, \widehat{\llbracket \beta \rrbracket}_A(b), a_{\phi(\overline{p}+1)}, \ldots, a_{\phi(|u|)});$$

wherein $\overline{p} = \overline{\phi}(p)$*; otherwise*
(2) if $i \in I^{\underline{s},v}$ *then if we define*

$$F_i^{\alpha,\beta,\phi,w,p} : \underline{A}^{w\{p/z\}} \to \underline{A}^{\delta^{\underline{s},w,p}(z)}$$

(again ambiguously denoted $F_i^{\alpha,\beta}$) by

$$F_i^{\alpha,\beta,\phi,w,p} = ((\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_{i,1}}, \ldots, (\llbracket \mathbb{C}(\alpha,\beta) \rrbracket_{\underline{A}})_{j_{i,|\delta^{\underline{s},w,p}(z)|}})$$

wherein $j_{i,1} = j_i$ *as defined above and* $j_{i,k} = j_{i,k-1} + 1$ *for* $k = 2, \ldots, |\delta^{\underline{s},w,p}(z)|$ *then*

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha,\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ *such that*

$$(P(a\{p/b\}))_{\overline{\phi}(r)} = (\llbracket \alpha \rrbracket_{\underline{A}}(P(a\{p/b\})))_i.$$

A.2 Proof of Theorem 9

Proof of Theorem 9. By induction on the structural complexity of the scheme $\alpha \in \mu PR(\underline{\Sigma})_{u,v}$ uniformly in (u, v).

Basis Cases. We have three cases to consider:

(1) **Constant Functions.** In this case $\alpha = c^u$ for some $c \in \Sigma_{\lambda,s'}$ for any $s' \in S$.
Since in this case $s' \in S$ it is sufficient to show that

$$F_1^{\alpha,\beta}(a\{p/b\}) = \llbracket \alpha \rrbracket_{\underline{A}}(P(a\{p/b\})).$$

This is obvious as $\Diamond^{\beta,w,p}(c^u) = c^{\Delta^{\underline{s},w,p}(u,z)}$.

(2) Algebraic Operations. In this case $\alpha = \sigma$ for some $\sigma \in \Sigma_{u,s'}$ for any $s' \in S$. Again since $s' \in S$ as with the previous case it is sufficient to show that

$$F_1^{\alpha,\beta}(a\{p/b\}) = [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})).$$

We have two sub-cases to consider:
(a) $\sigma \neq eval_s$ and
(b) $\sigma = eval_s$.

Sub-case (a) $\sigma \neq eval_s$. We calculate as follows:
L.H.S.

$$F_1^{\alpha,\beta}(a\{p/b\}) = [\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\})$$

by the definition of $F_1^{\alpha,\beta}$ with the hypothesis that $v \in S$

$$= [\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of $\mathcal{C}$

$$= [\![\sigma]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by hypothesis on α and the definition of $\Diamond$

$$= [\![\sigma]\!]_{\underline{A}}(a_{\phi(1)}, \ldots, a_{\phi(|u|)})$$

by the definition of $Init$ with the hypothesis that $\sigma \neq eval_s$ and

hence that $I^{\underline{s},u} = \varnothing$

$$= [\![\sigma]\!]_{\underline{A}}(P(a\{p/b\}))$$

by the definition of P since as $I^{\underline{s},u} = \varnothing$ it must be the case that $\overline{\phi}(p) = 0$.

Sub-case (b) $\sigma = eval_s$. We calculate as follows
L.H.S.

$$F_1^{\alpha,\beta}(a\{p/b\}) = [\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\})$$

by the definition of $F_1^{\alpha,\beta}$ with the hypothesis that $|v| = 1$

$$= [\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of $\mathcal{C}$

$$= [\![Eval_s^{w,z,p}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by hypothesis on α and the definition of $\Diamond$

$$= [\![Eval_s^{w,z,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b))$$

by the definition of $Init$ for some $t = a_{\phi(1)}$ and for some $r = \phi(2) \in I^{\underline{s},w}$. We now have two sub-sub-cases to consider:

(I) $r = p$, and

(II) $r \neq p$.

Sub-sub-Case (I) $r = p$.

$$[\![Eval_s^{w,z,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)) = [\![\beta]\!]_{\underline{A}}(t, b)$$

by Lemma 60

$$= \widehat{[\![\beta]\!]_{\underline{A}}}(b)(t)$$

by the definition of $\widehat{\ }$

$$= [\![eval_s]\!]_{\underline{A}}(t, \widehat{[\![\beta]\!]_{\underline{A}}}(b))$$

by definition

$$= [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\}))$$

by hypothesis on α, the fact that $t = a_{\phi(1)}$, $r = p = \phi(2)$ and by the definition of P.

Sub-Sub-case (II) $r \neq p$.

$$[\![Eval_s^{w,z,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)) = [\![eval_s]\!]_{\underline{A}}(t, a_r)$$

by Lemma 60

$$= [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\}))$$

by hypothesis on α, the fact that $t = a_{\phi(1)}$, $r = \phi(2)$ and by the definition of P.

(3) Projection Functions. In this case $\alpha = U_k^u$ for any $u \in \underline{S}^+$ and for any $k \in \{1, \ldots, |u|\}$, and consequently we have two sub-cases to consider:

(a) $k \notin I^{\underline{s},u}$ and

(b) $k \in I^{\underline{s},u}$.

Sub-case (a) $k \notin I^{s,u}$. Since $k \notin I^{s,u}$ and $|v| = |u_k| = 1$ it is sufficient to show that

$$F_1^{\alpha,\beta}(a\{p/b\}) = [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})).$$

We calculate as follows:

L.H.S.

$$F_1^{\alpha,\beta}(a\{p/b\}) = [\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\})$$

by the definition of $F_1^{\alpha,\beta}$ with the hypothesis that $v \in S$

$$= [\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of $\mathcal{C}$

$$= [\![\mathrm{U}_{k'}^{\Delta^{\underline{s},w,p}(u,z)}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by hypothesis on α and the definition of $\Diamond$

$$= [\![\mathrm{U}_{|c_1\cdots c_{k-1}|+1}^{c_1\cdots c_{|u|}}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of Δ and k' wherein for $j = 1, \ldots, |u|$,

$$c_j = \begin{cases} u_j & \text{if } j \notin I^{s,u}; \\ \delta^{s,w,i}(z) & \text{otherwise} \end{cases}$$

$$= [\![\mathrm{U}_{|c_1\cdots c_{k-1}|+1}^{c_1\cdots c_{|u|}}]\!]_{\underline{A}}(y_1, \ldots, y_{|u|})$$

by the definition of $Init$ wherein for $j = 1, \ldots, |u|$

$$y_j = \begin{cases} a_{\phi(j)} & \text{if } j \notin I^{s,u}, \\ \theta^{s,w,p,z,\phi(j)}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{otherwise} \end{cases}$$

$$= y_k$$

since by definition y_i is of type c_i for $i = 1, \ldots, |u|$, and $|y_k| = 1$ since $k \notin I^{s,u}$

$$= a_{\phi(k)}$$

by the hypothesis that $k \notin I^{s,u}$.

$$= [\![\mathrm{U}_k^u]\!]_{\underline{A}}(a_{\phi(1)}, \ldots, a_{\phi(\overline{\phi}(p)-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(\overline{\phi}(p)+1)}, \ldots, a_{\phi(|u|)})$$

by hypothesis on k and the definition of $[\![.]\!]_{\underline{A}}$

$$= [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\}))$$

by hypothesis on α and by the definition of P.

Sub-case (b) $k \in I^{\underline{s},u}$. Since $k \in I^{\underline{s},u}$ and $|v| = |u_k| = 1$ it is sufficient to show that

$$F_1^{\alpha,\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

for some $r \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\varphi}(r)} = [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})).$$

We calculate as follows:

L.H.S.

$$F_1^{\alpha,\beta}(a\{p/b\}) = (([\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\}))_{j_{1,1}},\ldots,([\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\}))_{j_{1,|\delta^{\underline{s},w,p_{(z)}}|}})$$

by the definition of $F_1^{\alpha,\beta}$ with the hypothesis that $|v| = 1$

$$= (([\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\}))_1,\ldots,([\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\}))_{|\delta^{\underline{s},w,p_{(z)}}|})$$

again by the hypothesis that $|v| = 1$

$$= \begin{pmatrix} ([\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_1,\ldots, \\ ([\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_{|\delta^{\underline{s},w,p_{(z)}}|} \end{pmatrix}$$

by the definition of $\mathcal{C}$

$$= \begin{pmatrix} ([\![\mathrm{U}_{k'}^{\Delta^{\underline{s},w,p}(u,z)},\ldots,\mathrm{U}_{k'+|\delta^{\underline{s},w,p}(z)|}^{\Delta^{\underline{s},w,p}(u,z)}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_1,\ldots, \\ ([\![\mathrm{U}_{k'}^{\Delta^{\underline{s},w,p}(u,z)},\ldots,\mathrm{U}_{k'+|\delta^{\underline{s},w,p}(z)|}^{\Delta^{\underline{s},w,p}(u,z)}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_{|\delta^{\underline{s},w,p}(z)|} \end{pmatrix}$$

by hypothesis on α and the definition of $\Diamond$

$$= [\![\mathrm{U}_{k'}^{\Delta^{\underline{s},w,p}(u,z)},\ldots,\mathrm{U}_{k'+|\delta^{\underline{s},w,p}(z)|}^{\Delta^{\underline{s},w,p}(u,z)}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})$$
$$= [\![\mathrm{U}_{|c_1\cdots c_{k-1}|+1}^{c_1\cdots c_{|u|}},\ldots,\mathrm{U}_{|c_1\cdots c_{k-1}|+|\delta^{\underline{s},w,p}(z)|+1}^{c_1\cdots c_{|u|}}]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of Δ and k' wherein for $j = 1,\ldots,|u|$,

$$c_j = \begin{cases} u_j & \text{if } j \notin I^{s,u}; \\ \delta^{s,w,i}(z) & \text{otherwise} \end{cases}$$

$$= [\![\mathrm{U}_{|c_1\cdots c_{k-1}|+1}^{c_1\cdots c_{|u|}},\ldots,\mathrm{U}_{|c_1\cdots c_{k-1}|+|\delta^{\underline{s},w,p}(z)|+1}^{c_1\cdots c_{|u|}}]\!]_{\underline{A}}(y_1,\ldots,y_{|u|})$$

by the definition of $Init$ wherein for $j = 1,\ldots,|u|$

$$y_j = \begin{cases} a_{\phi(j)} & \text{if } j \notin I^{s,u}, \\ \theta^{s,w,p,z,\phi(j)}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b) & \text{otherwise} \end{cases}$$

$$= y_k$$

since by definition y_i is of type c_i for $i = 1, \ldots, |u|$, and $|y_k| = |\delta^{\underline{s},w,p}(z)|$

as $k \in I^{\underline{s},w}$ by hypothesis

$$= \theta^{\underline{s},w,p,z,\phi(k)}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

again by the hypothesis that $k \in I^{\underline{s},u}$.
It now remains to show that

$$(P(a\{p/b\}))_{\overline{\phi}(\phi(k))} = [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})).$$

We calculate as follows:

$$(P(a\{p/b\}))_{\overline{\phi}(\phi(k))} = (P(a\{p/b\}))_k$$

by Lemma 3

$$= [\![\mathrm{U}_k^u]\!]_{\underline{A}} P(a\{p/b\})$$

by the definition of $[\![.]\!]_{\underline{A}}$

$$= [\![\alpha]\!]_{\underline{A}} P(a\{p/b\})$$

by hypothesis.

Induction Hypothesis. Assume for any scheme $\alpha' \in \mu PR(\underline{\Sigma})_{u',v'}$ of α for any $u', v' \in \underline{S}^+$ of less structural complexity than α such that $w \supseteq u'$ that for each w/u'-permutation ϕ'

(A) if $i \notin I^{\underline{s},v'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha',\beta}(a\{p/b\}) = \left([\![\alpha']\!]_A(P'(a\{p/b\}))\right)_i$$

wherein

$$P'(a\{p/b\}) = (a_{\phi'(1)}, \ldots, a_{\phi'(\overline{\phi'}(p)-1)}, \widehat{[\![\beta]\!]_A}(b), a_{\phi'(\overline{\phi'}(p)+1)}, \ldots, a_{\phi'(|u'|)});$$

and

(B) if $i \in I^{\underline{s},v'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha',\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ such that

$$(P'(a\{p/b\}))_{\overline{\phi'}(r)} = ([\![\alpha']\!]_{\underline{A}}(P'(a\{p/b\})))_i.$$

Induction Step. We must show for any scheme $\alpha'' \in \mu PR(\underline{\Sigma})_{u'',v''}$ of α for any $u'', v'' \in \underline{S}^+$ such that $w \supseteq u''$ that for each w/u''-permutation ϕ''

(A) if $i \notin I^{\underline{s},v''}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha'',\beta}(a\{p/b\}) = \left([\![\alpha'']\!]_A(P''(a\{p/b\}))\right)_i$$

wherein

$$P''(a\{p/b\}) = (a_{\phi''(1)}, \ldots, a_{\phi''(\overline{\phi''}(p)-1)}, \widehat{[\![\beta]\!]_A}(b), a_{\phi''(\overline{\phi''}(p)+1)}, \ldots, a_{\phi''(|u''|)});$$

and

(B) if $i \in I^{\underline{s},v''}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_i^{\alpha'',\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ such that

$$(P''(a\{p/b\}))_{\overline{\phi''}(r)} = ([\![\alpha'']\!]_{\underline{A}}(P''(a\{p/b\})))_i.$$

We have four cases to consider:

(4) **Vectorization.** In this case $\alpha =< \alpha_1, \ldots, \alpha_m >$ for some $\alpha_i \in \mu PR(\underline{\Sigma})_{u,s_i}$ for any $s_i \in \underline{S}$ for $i = 1, \ldots, m$.
This case follows directly from m applications of the Induction Hypothesis and is left to the reader.

(5) **Composition.** In this case $\alpha = \alpha_2 \circ \alpha_1$ for some $\alpha_1 \in \mu PR(\underline{\Sigma})_{u,u'}$ and for some $\alpha_2 \in \mu PR(\underline{\Sigma})_{u',v}$ for any $u, u', v \in \underline{S}^+$.
We calculate as follows:
L.H.S.

$$F^{\alpha,\beta}(a\{p/b\}) = [\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\})$$

by the definition of $F^{\alpha,\beta}$

$$= [\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})$$

by the definition of $\mathcal{C}$

$$= [\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}} \circ [\![\Diamond^{\beta,w,p}(\alpha_1)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})$$

by hypothesis on α and the definition of $\mathcal{C}$

$$= [\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}([\![\mathcal{C}(\alpha_1,\beta]\!]_{\underline{A}}(a\{p/b\})))$$

by the definition of $\mathcal{C}$

$$= [\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}(F^{\alpha_1,\beta}(a\{p/b\})$$

by the definition of $F^{\alpha_1,\beta}$. Now by the Induction Hypothesis with $\alpha' = \alpha^1$, $y = u$, $\phi' = \phi$ and $P' = P$, we have for each $i \notin I^{\underline{s},u'}$

$$F_i^{\alpha_1,\beta}(a\{p/b\}) = [\![\alpha_1]\!]_{\underline{A}}(P(a\{p/b\}))$$

and for each $i \in I^{\underline{s},u'}$

$$F_i^{\alpha_1,\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

for some $r \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\phi}(r)} = ([\![\alpha_1]\!]_{\underline{A}}(P(a\{p/b\})))_i.$$

Therefore by Lemma 61 with $\gamma = \alpha_2$, $x = u'$, $x' = v$, $\gamma' = \alpha_1$, $y = u$, $\psi = \phi$ and $L = P$ we have for each $i \notin I^{\underline{s},v}$

$$\left([\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}F^{\alpha_1,\beta}(a\{p/b\})\right)_i = \left([\![\alpha]\!]_{\underline{A}}(P(a\{p/b\}))\right)_i$$

and for each $i \in I^{\underline{s},v}$ that

$$\left([\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}F^{\alpha_1,\beta}(a\{p/b\})\right)_i = \theta^{\underline{s},w,p,z,r'}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

for some $r' \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\phi}(r')} = ([\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})))_i$$

as required.

(6) Primitive Recursion. In this case $\alpha = *(\alpha_1,\alpha_2)$ for some $\alpha_1 \in \mu PR(\Sigma)_{u,v}$ and for some $\alpha_2 \in \mu PR(\Sigma)_{t\,u\,v,v}$.
We calculate as follows:
L.H.S.

$$F^{\alpha,\beta}(a\{p/b\}) = [\![\mathcal{C}(\alpha,\beta)]\!]_{\underline{A}}(a\{p/b\}))$$

by the definition of $F^{\alpha,\beta}$

$$[\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by the definition of $\mathcal{C}$

$$= [\![*(\Diamond^{\beta,w,p}(\alpha_1), \Diamond^{\beta,w,p}(\alpha_2))]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))$$

by hypothesis on α and the definition of $\Diamond$

$$= [\![*(\Diamond^{\beta,w,p}(\alpha_1), \Diamond^{\beta,w,p}(\alpha_2))]\!]_{\underline{A}}(y_1, \ldots, y_{|u|})$$

by the definition of $Init^{\phi,z,\underline{s},p}$ wherein for $i = 1, \ldots, |u|$

$$y_i = \begin{cases} a_{\phi(i)} & \text{if } i \notin I^{\underline{s},u}, \text{ and} \\ \theta^{\underline{s},w,p,z,\phi(i)}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } i \in I^{\underline{s},u}. \end{cases}$$

R.H.S.

$$[\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})) = [\![*(\alpha_1, \alpha_2)]\!]_{\underline{A}}(a_{\phi(1)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)})$$

by hypothesis on α and by the definition of P wherein $q = \phi(p)$. We now proceed by sub-induction on the value of $y_1 = a_{\phi(1)} = k \in \mathbb{N}$.

Sub-Basis $k = 0$.

L.H.S. $[\![*(\Diamond^{\beta,w,p}(\alpha_1), \Diamond^{\beta,w,p}(\alpha_2))]\!]_{\underline{A}}(y_1, \ldots, y_{|u|})$

$$= [\![\Diamond^{\beta,w,p}(\alpha_1)]\!]_{\underline{A}}(y_2, \ldots, y_{|u|})$$

by the definition of $[\![.]\!]_{\underline{A}}$ with the hypothesis that $y_1 = 0$

$$= [\![\Diamond^{\beta,w,p}(\alpha_1)]\!]_{\underline{A}}(Init^{\phi',z,\underline{s},p}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b))$$

by the definition of $Init$ wherein $u' = u_2 \cdots u_{|u|}$ and Φ' is the w/u'-replacement defined by

$$(\forall l \in \{1, \ldots, |u'|\}) \quad \phi'(l) = \phi(l+1).$$

$$= F^{\alpha_1,\beta}(a\{p/b\})$$

by the definition of $F^{\alpha_1,\beta}$. Notice now by the Induction Hypothesis with $\alpha' = \alpha_1$ and $v' = v$ we have for $i = 1, \ldots, |v|$ that

$$F_i^{\alpha_1,\beta}(a\{p/b\})_i = ([\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\})))_i$$

if $i \notin I^{\underline{s},v}$ and

$$F_i^{\alpha_1,\beta}(a\{p/b\})_i = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

if $i \in I^{\underline{s},v}$ for some $r \in I^{\underline{s},w}$ such that

$$P'(a\{p/b\})_{\overline{\phi}'(r)} = [\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\}))_i.$$

Therefore to complete the proof in this sub-basis case it remains to show that

$$[\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\})) = [\![*(\alpha_1, \alpha_2)]\!]_{\underline{A}}(0, a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}).$$

We calculate as follows:

$$[\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\})) = [\![\alpha_1]\!]_{\underline{A}}(a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)})$$

by definition

$$= [\![\alpha_1]\!]_{\underline{A}}(a_{\phi'(1)}, \ldots, a_{\phi'(q'-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi'(q'+1)}, \ldots, a_{\phi'(|u|)})$$

wherein $q' = \phi'(p)$ since by definition we have $\phi'(l) = \phi(l+1)$

for $l = 1, \ldots, |u'|$

$$= [\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\}))$$

by definition as required.

Sub-Induction Hypothesis. Assume that if $a_{\phi(1)} = y_1 = k$ for some fixed $k \in \mathbf{N}$ then for each $i \notin I^{\underline{s},v}$ that

$$([\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_i = ([\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})))_i$$

and for each $i \in I^{\underline{s},v}$ that

$$([\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))_i = \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r \in I^{\underline{s},w}$ such that

$$P'(a\{p/b\})_{\overline{\phi}'(r)} = [\![\alpha_1]\!]_{\underline{A}}(P'(a\{p/b\}))_i.$$

Sub-Induction Step.

L.H.S. $[\![*(\Diamond^{\beta,w,p}(\alpha_1), \Diamond^{\beta,w,p}(\alpha_2))]\!]_{\underline{A}}(k+1, y_2, \ldots, y_{|u|})$

$$= [\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}(k, y_2, \ldots, y_{|u|}, [\![\Diamond^{\beta,w,p}(\alpha)]\!]_{\underline{A}}(k, y_2, \ldots, y_{|u|}))$$

by the definition of $[\![.]\!]_{\underline{A}}$

$$= [\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}(k, y_2, \ldots, y_{|u|}, y'_1, \ldots, y'_{|v|})$$

by the Sub-induction Hypothesis wherein for $j = 1, \ldots, |v|$

$$y'_j = \begin{cases} ([\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})))_j & \text{if } j \notin I^{\underline{s},v}, \text{ and} \\ \theta^{\underline{s},w,p,z,r}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } j \in I^{\underline{s},v} \end{cases}$$

for some $r \in I^{\underline{s},w}$ such that

$$P(a\{p/b\})_{\overline{\phi}'(r)} = [\![\alpha_1]\!]_{\underline{A}}(P(a\{p/b\}))_j.$$

Recall now that by definition for $j = 2, \ldots, |u'|$ we have

$$y_j = \begin{cases} a_{\phi(j)} & \text{if } j \notin I^{\underline{s},u'}, \text{ and} \\ \theta^{\underline{s},w,p,z,\phi(j)}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } j \in I^{\underline{s},u'} \end{cases}$$

and also that

$$(P(a\{p/b\}))_{\overline{\phi}(\phi(j))} = \begin{cases} a_{\phi(j)} & \text{if } j \neq p, \text{ and} \\ \widehat{[\![\beta]\!]_{\underline{A}}}(b) & \text{if } j = p. \end{cases}$$

Therefore by Lemma 62 if we take $\delta = \alpha_2$, $w' = t\,u'\,v$, $v' = v$,

$$\chi = (k, y_2, \ldots, y_{|u'|}, y'_1, \ldots, y'_{|v|}) \in \underline{A}^{\Delta^{\underline{s},w,p}(w',z)}$$

and

$$\rho = \begin{cases} (& (k, a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}, \\ & ([\![\alpha]\!]_{\underline{A}}(a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}))_1, \\ & \vdots \\ & ([\![\alpha]\!]_{\underline{A}}(a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}))_{|v|} &) \end{cases}$$

then we have for each $l \notin I^{\underline{s},v}$

$$([\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}(k, y_2, \ldots, y_{|u'|}, y'_1, \ldots, y'_{|v|}))_l = ([\![\alpha_2]\!]_{\underline{A}}(\rho))_l$$

and for each $l \in I^{\underline{s},v}$

$$([\![\Diamond^{\beta,w,p}(\alpha_2)]\!]_{\underline{A}}(k, y_2, \ldots, y_{|u'|}, y'_1, \ldots, y'_{|v|}))_l = \theta^{\underline{s},w,p,z,r'}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $r' \in I^{\underline{s},w}$ such that

$$P(a\{p/b\})_{\overline{\phi}(r')} = [\![\alpha_2]\!]_{\underline{A}}(\rho)_l.$$

Therefore to complete the Sub-Induction step and to complete the proof of this case it remains to show that

$$[\![\alpha_2]\!]_{\underline{A}}(\rho) = [\![\alpha]\!]_{\underline{A}}(P(a\{p/b\})).$$

We calculate as follows:

R.H.S.

$[\![\alpha]\!]_{\underline{A}}(P(a\{p/b\}))$

$$= [\![\alpha]\!]_{\underline{A}}(k+1, a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)})$$

by the definition of P

$$= [\![\alpha_2]\!]_{\underline{A}}(\begin{array}{l} (k, a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}, \\ ([\![\alpha]\!]_{\underline{A}}(a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}))_1, \\ \vdots \\ ([\![\alpha]\!]_{\underline{A}}(a_{\phi(2)}, \ldots, a_{\phi(q-1)}, \widehat{[\![\beta]\!]_{\underline{A}}}(b), a_{\phi(q+1)}, \ldots, a_{\phi(|u|)}))_{|v|} \quad) \end{array}$$

by hypothesis on α and by the definition of $[\![.]\!]_{\underline{A}}$

$= [\![\alpha_2]\!]_{\underline{A}}(\rho)$

by the definition of ρ.

(7) **Minimalization.** In this case $\alpha = \mu(\alpha')$ for some $\alpha' \in \mu PR(\Sigma)_{\mathbf{n}\,u,\mathbf{b}}$.
Again this case follows directly from an application of the Induction Hypothesis and is left to the reader.

□

A.3 Deferred Proofs of Intermediate Lemmata

Proof of Lemma 60.
$[\![Eval_s^{\beta,w,p}]\!]_{\underline{A}}(t, \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|}))$

$= [\![Eval_s^{\beta,w,p}]\!]_{\underline{A}}(t,k,V_1,b,V_2)$

by the definition of θ wherein $k = \lambda_1^{\underline{s},w}(r)$, $V_1 = \Pi^{\underline{s},w_1\cdots w_{p-1}}(a_1,\ldots,a_{p-1})$ and

$V_2 = \Pi^{\underline{s},w_{p+1}\cdots w_{|w|}}(a_{p+1},\ldots,a_{|w|})$

$= [\![switch^{|I^{\underline{s},w}|,s,\lambda_1^{\underline{s},w}(p)} \circ < U_2^{t\,x},\beta_1,\ldots,\beta_{|I^{\underline{s},w}|} >]\!]_{\underline{A}}(t,k,V_1,b,V_2)$

by the definition of $Eval$ wherein $x = \delta^{\underline{s},w,p}(z)$ and for $j = 1,\ldots,|I^{\underline{s},w}|$

$$\beta_j = \begin{cases} eval_s \circ < U_1^{t\,x}, U_{j+2}^{t\,x} > & \text{if } 1 \le j < \lambda_1^{\underline{s},w}(p); \\ \beta \circ < U_1^{t\,x}, U_{j+2}^{t\,x},\ldots,U_{j+|z|+1}^{t\,x} > & \text{if } j = \lambda_1^{\underline{s},w}(p); \\ eval_s \circ < U_1^{t\,x}, U_{j+|z|+1}^{t\,x} > & \text{if } \lambda_1^{\underline{s},w}(p) < j \le |I^{\underline{s},w}|, \end{cases}$$

$= Switch_{\underline{A}}^{|I^{\underline{s},w}|,s,\lambda_1^{\underline{s},w}(p)}(k,[\![\beta_1]\!]_{\underline{A}}(t,k,V_1,b,V_2),\ldots,[\![\beta_{|I^{\underline{s},w}|}]\!]_{\underline{A}}(t,k,V_1,b,V_2))$

by the definition of $[\![.]\!]_{\underline{A}}$

$= [\![\beta_k]\!]_{\underline{A}}(t,k,V_1,b,V_2)$

by the definition of *Switch*. We now have two cases to consider:

(1) $r = p$; that is, $k = \lambda_1^{\underline{s},w}(r) = \lambda_1^{\underline{s},w}(p)$.

(2) $r \neq p$; that is, $k = \lambda_1^{\underline{s},w}(r) \neq \lambda_1^{\underline{s},w}(p)$ and

Case (1) $r = p$.

$$[\![\beta_k]\!]_{\underline{A}}(t, k, V_1, b, V_2) = [\![\beta \circ < U_1^{t\,x}, U_{k+2}^{t\,x}, \ldots, U_{k+|z|+1}^{t\,x} >]\!]_{\underline{A}}(t, k, V_1, b, V_2)$$

by hypothesis on k

$$= [\![\beta]\!]_{\underline{A}}(t, b)$$

by the definition of $[\![.]\!]_{\underline{A}}$.

Case (2) $r \neq p$. We now have a further two sub-cases to consider:

Sub-case (a) $r < p$; that is, $k < \lambda_1^{s,w}(p)$.

$$[\![\beta_k]\!]_{\underline{A}}(t, k, V_1, b, V_2) = [\![eval_s \circ < U_1^{t\,x}, U_{k+2}^{t\,x} >]\!]_{\underline{A}}(t, k, V_1, b, V_2)$$

by hypothesis on k

$$= [\![eval_s]\!]_{\underline{A}}(t, (V_1)_k)$$

by the definition of $[\![.]\!]_{\underline{A}}$ and by hypothesis on k

$$= [\![eval_s]\!]_{\underline{A}}(t, a_r)$$

by the fact that $k = \lambda_1^{s,w}(r)$ and by the definition of $V_1 = \Pi^{s,w_1 \cdots w_{p-1}}(a_1, \ldots, a_{p-1})$.

Sub-case (b) $r > p$: that is, $k > \lambda_1^{s,w}(p)$. This case is similar to Sub-case (a) and is omitted.

□

Proof of Lemma 61. By induction on the structural complexity of the scheme $\alpha \in \mathrm{PR}(\underline{\Sigma})_{x,v}$ uniformly in (y, x).

Basis Cases.

(1) Constant Functions. In this case $\gamma = c_{s'}^{y}$ for some $c \in \Sigma_{\lambda,s'}$ for some $s' \in S$ and for some $y \in \underline{S}^{+}$. Notice that in this case as $x = s' \in S$ that it is sufficient to show that

$$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\})) = [\![\gamma \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\})).$$

We calculate as follows:

L.H.S.

$$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma,\beta}(a\{p/b\})) = [\![c^{y}]\!]_{\underline{A}}(F^{\gamma,\beta}(a\{p/b\}))$$

by hypothesis on γ and by the definition of $\Diamond$

$$= [\![c^y]\!]_{\underline{A}} \circ [\![\gamma']\!]_{\underline{A}}(L(a\{p/b\}))$$

by the hypothesis on $F^{\gamma',\beta}$

$$= [\![c^y \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\}))$$

by the definition of $[\![.]\!]_{\underline{A}}$ as required.

(2) Algebraic operations. In this case $\gamma = \sigma$ for some $\sigma \in \Sigma_{y,s'}$ for some $y \in \underline{S}^+$ and for some $s' \in S$. We consider two sub-cases:
(A) $\sigma = eval_{s'}$, and
(B) $\sigma \neq eval_{s'}$.
We prove Sub-case (A) and leave Sub-case (B) to the reader.
Sub-Case (A) $\sigma = eval_{s'}$. Notice that in this sub-case that $y = \mathbf{t}\,\underline{s}$ and $x = s'$ and therefore it is sufficient to show that:

$$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)) =$$

$$[\![\gamma]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, ([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_2)$$

wherein

$$(L(a\{p/b\})_{\overline{\varphi}(r)} = ([\![\gamma']\!]_{\underline{A}})(L(a\{p/b\})))_2.$$

We calculate as follows:
L.H.S.
$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)) =$

$$[\![Eval^{w,z,p}]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b))$$

by the definition of $Eval$ and by hypothesis on γ.
R.H.S.
$[\![\gamma]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, ([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_2) =$

$$[\![eval_{s'}]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, (L(a\{p/b\}))_{\overline{\varphi}(r)})$$

by hypothesis on γ' and $(L(a\{p/b\}))_{\overline{\varphi}(r)}$ respectively. The fact that
$[\![Eval^{w,z,p}]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)) =$

$$[\![eval_{s'}]\!]_{\underline{A}}(([\![\gamma']\!]_{\underline{A}}(L(a\{p/b\})))_1, (L(a\{p/b\}))_{\overline{\varphi}(r)})$$

follows immediately by Lemma 60.

(3) Projection Functions. In this case $\gamma = \mathrm{U}^y_j$ for some $y \in \underline{S}^+$ and for some $j \in \{1,\ldots,|y|\}$.
We consider two sub-cases:
(A) $y_j = \underline{s}$, and

(B) $y_j \neq \underline{s}$.

Sub-Case (A) $y_j = \underline{s}$. In this sub-case as $x = y_j = \underline{s}$ it is sufficient to show that

$$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\})) = \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

such that

$$(L(a\{p/b\})_{\overline{\varphi}(r)} = ([\![\gamma']\!]_{\underline{A}})(L(a\{p/b\})))_1.$$

This is straightforward and is left to the reader.

Sub-Case (B) $y_j \neq \underline{s}$. In this sub-case as $x = y_j \neq \underline{s}$ it is sufficient to show that

$$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\})) = [\![\gamma \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\})).$$

Again this is straightforward and is left to the reader.

Induction Hypothesis. For each $s \in S$, for each $w/\overline{y}$-permutation ψ' for some $\overline{y}, \overline{x} \in \underline{S}^+$ such that $I^{\underline{s},w} \neq \varnothing$, for each $p \in I^{\underline{s},w}$, for each $\overline{\gamma} \in \mu\mathrm{PR}(\underline{\Sigma})_{\overline{x},\overline{x}'}$ for some $\overline{x}, \overline{x}' \in \underline{S}^+$ for each $\beta \in \mu\mathrm{PR}(\underline{\Sigma})_{t\,z,s}$ for some $z \in \underline{S}^*$ and for each $\overline{\gamma}' \in \mu\mathrm{PR}(\underline{\Sigma})_{\overline{y},\overline{x}}$ if $\overline{\gamma}$ is of less structural complexity than γ then

(A) for each $j \notin I^{\underline{s},\overline{x}}$ we have

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_j^{\overline{\gamma}',\beta}(a\{p/b\}) = \left([\![\gamma']\!]_A(L'(a\{p/b\}))\right)_j$$

wherein

$$L(a\{p/b\}) = (a_{\psi'(1)},\ldots,a_{\psi'(\overline{\psi}(p)-1)},\widehat{[\![\beta]\!]_A}(b),a_{\psi'(\overline{\psi}(p)+1)},\ldots,a_{\psi'(|\overline{y}|)});$$

and

(B) for each $j \in I^{\underline{s},\overline{x}'}$ we have

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad F_j^{\overline{\gamma}',\beta}(a\{p/b\}) = \theta^{\underline{s},w,p,z,r}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

for some $\overline{r} \in I^{\underline{s},w}$ such that

$$(L'(a\{p/b\}))_{\overline{\psi}(\overline{r})} = ([\![\overline{\gamma}']\!]_{\underline{A}}(L'(a\{p/b\})))_j$$

then

(1) if $i \notin I^{\underline{s},\overline{x}'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad \left([\![\Diamond^{\beta,w,p}(\overline{\gamma})]\!]_{\underline{A}}(F^{\overline{\gamma}',\beta}(a\{p/b\}))\right)_i = \left([\![\overline{\gamma} \circ \overline{\gamma}']\!]_A(L'(a\{p/b\}))\right)_i$$

and

(2) if $i \in I^{\underline{s},\overline{x}'}$ then

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}})$$

$$\left([\![\Diamond^{\beta,w,p}(\overline{\gamma})]\!]_{\underline{A}}(F^{\overline{\gamma}',\beta}(a\{p/b\}))\right)_i = \theta^{\underline{s},w,p,z,r'}(a_1,\ldots,a_{p-1},a_{p+1},\ldots,a_{|w|},b)$$

for some $\overline{r}' \in I^{\underline{s},w}$ such that

$$(L'(a\{p/b\}))_{\overline{\psi}'(\overline{r}')} = \left([\![\overline{\gamma} \circ \overline{\gamma}']\!]_{\underline{A}}(L'(a\{p/b\}))\right)_i.$$

As with previous proofs the vectorization and minimalization cases follow easily from the Induction Hypothesis and are left to the reader. Therefore we only consider composition and primitive recursion.

(5) Composition. In this case $\gamma = \gamma_2 \circ \gamma_1$ for some $\gamma_1 \in \mathrm{PR}(\underline{\Sigma})_{x,\tilde{x}}$ for some $\tilde{x} \in \underline{S}^+$ and for some $\gamma_2 \in \mathrm{PR}(\underline{\Sigma})_{\tilde{x},y}$. We calculate as follows:

L.H.S.

$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))$

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}([\![\Diamond^{\beta,w,p}(\gamma_1)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\})))$$

by hypothesis on γ and by the definition of $\Diamond$

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}([\![\Diamond^{\beta,w,p}(\gamma_1)]\!]_{\underline{A}}([\![\Diamond^{\beta,w,p}(\gamma')]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\}))))$$

by the definition of $\Diamond$

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}([\![\Diamond^{\beta,w,p}(\gamma_1 \circ \gamma')]\!]_{\underline{A}}(Init^{\phi,z,\underline{s},p}(a\{p/b\})))$$

by the definition of $\Diamond$

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(F^{\gamma_1 \circ \gamma',\beta}(a\{p/b\})).$$

by the definition of $F^{\gamma_1 \circ \gamma',\beta}$. Notice now by the Induction Hypothesis with $\phi' = \phi$, $\overline{\gamma} = \gamma_1 \circ \gamma'$, $\overline{x} = y$, $\overline{x}' = \tilde{x}$, and $L' = L$

$$[\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(F^{\gamma_1 \circ \gamma',\beta}(a\{p/b\})) = (d_1,\ldots,d_{|y|})$$

wherein for $i = 1,\ldots,|y|$

$$d_i = \begin{cases} [\![\gamma_2 \circ \gamma_1 \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\}))_i & \text{if } i \notin I^{\underline{s},y}, \text{ and} \\ \theta^{\underline{s},w,p,z,r'}(a_1,\ldots,a_{|w|},b) & \text{if } i \in I^{\underline{s},y} \end{cases}$$

such that

$$L(a\{p/b\})_{\overline{\varphi}(r')} = ([\![\gamma_2 \circ \gamma_1 \circ \gamma']\!]_{\underline{A}}(L(a\{p/b\}))_i$$

as required.

(6) Primitive Recursion. In this case $\gamma = *(\alpha_1, \alpha_2)$ for some $\alpha_1 \in \mathrm{PR}(\underline{\Sigma})_{x'',x'}$ and for some $\alpha_2 \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\,x''\,x',x'}$ for some $x'' \in \underline{S}^+$. We proceed by case analysis on the value of

$$F_1^{\gamma,\beta}(a\{p/b\}) = k \in \mathbf{N}.$$

(Notice that $x = \mathbf{t}\,x''$.)

Case (A) $k = 0$. We calculate as follows:

L.H.S.

$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))$

$$= [\![\Diamond^{\beta,w,p}(*(\gamma_1,\gamma_2))]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))$$

by hypothesis on γ

$$= [\![\Diamond^{\beta,w,p}(\gamma_1)]\!]_{\underline{A}}(F_2^{\gamma',\beta}(a\{p/b\}),\ldots,F_{|x|}^{\gamma',\beta}(a\{p/b\}))$$

by the definition of $[\![.]\!]_{\underline{A}}$, the definition of $\Diamond$ and the hypothesis that

$$F_1^{\gamma',\beta}(a\{p/b\}) = 0$$

$$= [\![\Diamond^{\beta,w,p}(\gamma_1)]\!]_{\underline{A}}(F^{\delta,\beta}(a\{p/b\}))$$

wherein $\delta =< \mathrm{U}_2^x,\ldots,\mathrm{U}_{|x|}^{x'} > \circ\,\gamma'$. The proof is now easily completed in this case by an application of the Induction Hypothesis with $\overline{\gamma} = \gamma_1$ and $\overline{\gamma}' = \delta$. The details are left to the reader.

Case (A) $k = k' + 1$ **for some** $k' \in \mathbf{N}$. We calculate as follows:

L.H.S.

$[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))$

$$= [\![\Diamond^{\beta,w,p}(*(\gamma_1,\gamma_2))]\!]_{\underline{A}}(F^{\gamma',\beta}(a\{p/b\}))$$

by hypothesis on γ

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(\ k', F_2^{\gamma',\beta}(a\{p/b\}),\ldots,F_{|x|}^{\gamma',\beta}(a\{p/b\}),\ldots \\ [\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(k', F_2^{\gamma',\beta}(a\{p/b\}),\ldots,F_{|x|}^{\gamma',\beta}(a\{p/b\}))\ )$$

by the definition of $[\![.]\!]_{\underline{A}}$, the definition of $\Diamond$ and the hypothesis that

$$F_1^{\gamma',\beta}(a\{p/b\}) = k'$$

$$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(F^{\delta',\beta}(a\{p/b\}),[\![\Diamond^{\beta,w,p}(\gamma)]\!]_{\underline{A}}(F^{\delta',\beta}(a\{p/b\}))$$

wherein $\delta' =< \alpha_{pred} \circ \mathrm{U}_1^x, \mathrm{U}_2^x,\ldots,\mathrm{U}_{|x|}^{x'} > \circ\,\gamma'$ and α_{pred} is the primitive recursive

scheme representing the predecessor function

$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(F^{\delta',\beta}(a\{p/b\}), F^{\gamma\circ\delta',\beta}(a\{p/b\})$

by definition

$= [\![\Diamond^{\beta,w,p}(\gamma_2)]\!]_{\underline{A}}(F^{\delta'',\beta}(a\{p/b\}))$

wherein $\delta'' =< \mathrm{U}_1^x \circ \delta', \ldots, \mathrm{U}_{|x|}^x \circ \delta', \mathrm{U}_1^{x'} \circ \gamma \circ \delta', \ldots . \mathrm{U}_{|x'|}^{x'} \circ \gamma \circ \delta' >$. As with Sub-case (A) the proof in this sub-case is now easily completed by an application of the Induction Hypothesis with $\overline{\gamma} = \gamma_1$ and $\overline{\gamma}' = \delta''$. The details are again left to the reader.

□

Proof of Lemma 62. By induction on the structural complexity of the scheme $\delta \in \mu\mathrm{PR}(\underline{\Sigma})_{w',v'}$ Uniformly in (w', v'). We prove the basis case wherein δ is a projection function and the induction case wherein δ is defined by primitive recursion and leave the other cases that are either similar or straightforward to the reader.

Basis Cases.

(3) Projection Functions. In this case $\delta = \mathrm{U}_j^x$ for some $x \in \underline{S}^+$ and for some $j \in \{1, \ldots, |x|\}$. We have two sub-cases to consider:

(A) $x_j = \underline{s}$, and

(B) $x_j \neq \underline{s}$.

We prove Sub-case (A) and leave Sub-case (B) to the reader.

Sub-Case (A) $x_j = \underline{s}$.

L.H.S.

$$[\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi) = [\![< \mathrm{U}_{j'}^{\Delta^{\underline{s},w,p}(x,z)}, \ldots, \mathrm{U}_{j'+|\delta^{\underline{s},w,p}(z)|-1}^{\Delta^{\underline{s},w,p}(x,z)} >]\!]_{\underline{A}}(\chi)$$

by the definition of $\Diamond$

$$= [\![< \mathrm{U}_{j'}^{c_1\cdots c_{|x|}}, \ldots, \mathrm{U}_{j'+|\delta^{\underline{s},w,p}(z)|-1}^{c_1\cdots c_{|x|}} >]\!]_{\underline{A}}(\chi)$$

by the definition of Δ wherein for $i = 1, \ldots, |x|$,

$$c_i = \begin{cases} x_j & \text{if } i \notin I^{\underline{s},x}; \\ \delta^{\underline{s},w,p}(z) & \text{otherwise} \end{cases}$$

and $j' = |c_1 \cdots c_{j-1}| + 1$

$$= \chi_j$$

as $\chi \in \underline{A}^{\Delta^{\underline{s},w,p}(x,z)}$

$$= \theta^{\underline{s},w,p,z,r_j}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

by the definition of χ with the hypothesis that $j \in I^{\underline{s},x}$.

Induction Hypothesis. Let A be any standard S-sorted Σ-algebra, let $s \in S$, let ϕ be a w/w''-sort permutation over $\underline{S}$ for some w such that $I^{\underline{s},w} \neq \varnothing$ and let $p \in I^{\underline{s},w}$. Also, let $\delta' \in \mathrm{PR}(\underline{\Sigma})_{v''}$ for some $w'', v'' \in \underline{S}^+$ be any scheme of less structural complexity than δ, let $\beta \in \mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, z, s}$ for some $z \in \underline{S}^+$ and let $P : \underline{A}^{w\{p/z\}} \to \underline{A}^{w''}$ be defined by

$$(\forall a\{p/b\} \in \underline{A}^{w\{p/z\}}) \quad P(a\{p/b\}) = (a_{\psi(1)}, \ldots, a_{\psi(\overline{\psi}(p)-1)}, \widehat{[\![\beta]\!]_A}(b), a_{\psi(\overline{\psi}(p)+1)}, \ldots, a_{\psi(|y|)}).$$

For each $a\{p/b\} \in \underline{A}^{w\{p/z\}}$ and for each $a'' = (a''_1, \ldots, a''_{|w''|}) \in \underline{A}^{w''}$ if

$$\chi' = (\chi'_1, \ldots, \chi'_{|w''|}) \in \underline{A}^{\Delta^{\underline{s},w,p}(w'',z)}$$

is defined by

$$\chi'_i = \begin{cases} a''_i & \text{if } i \notin I^{\underline{s},w''}, \text{ and} \\ \theta^{\underline{s},w,p,z,r_i}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{otherwise} \end{cases}$$

for some $r_i \in I^{\underline{s},w}$ for $i = 1, \ldots, |w''|$ and

$$\rho' = (\rho'_1, \ldots, \rho'_{|w''|}) \in \underline{A}^{|w''|}$$

is defined by

$$\rho'_j = \begin{cases} a''_j & \text{if } j \notin I^{\underline{s},w''}, \text{ and} \\ x_j & \text{otherwise} \end{cases}$$

for some $x_j \in \underline{A}^{\underline{s}}$ such that $x_j = (P(a\{p/b\}))_{\overline{\phi}(r_j)}$ for $j = 1, \ldots, |w''|$ then

(1) for each $l \notin I^{\underline{s},v''}$

$$\left([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi')\right)_l = \left([\![\delta]\!]_{\underline{A}}(\rho')\right)_l$$

and

(2) for each $l \in I^{\underline{s},v''}$

$$\left([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi')\right)_l = \theta^{\underline{s},w,p,z,q_l}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $q_l \in I^{\underline{s},w}$ for $i = 1, \ldots, |w''|$ such that $(P(a\{p/b\}))_{\overline{\phi}(q_l)} = (P(a\{p/b\}))_l$ for $j = 1, \ldots, |w''|$.

Induction Step.

(7) Primitive Recursion. In this case $\delta = *(\delta_1, \delta_2)$ for some $\delta_1 \in \mu\mathrm{PR}(\underline{\Sigma})_{u',v'}$ for some $u' \in \underline{S}^+$ and for some $\delta_2 \in \mu\mathrm{PR}(\underline{\Sigma})_{\mathbf{t}\, u'\, v', v'}$. (Notice that $w' = \mathbf{t}\, u'$.) We proceed by sub-induction on the value of $\chi_1 = \rho_1 = k \in \mathbf{N}$.

Sub-basis $k = 0$.

L.H.S.

$$[\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi) = [\![\Diamond^{\beta,w,p}(\delta_1)]\!]_{\underline{A}}(\chi_2, \ldots, \chi_{w'|})$$

by the definition of $\Diamond$, the definition of $[\![.]\!]_{\underline{A}}$ and the hypothesis on χ_1

$$= (d_1, \ldots, d_{|v'|})$$

by the Induction Hypothesis with $\delta' = \delta$, $w'' = u'$, $v'' = v'$, $\chi' = (\chi_2, \ldots, \chi_{|w'|})$ and $\rho' = (\rho_2, \ldots, \rho_{|w'|})$ wherein for $j = 1, \ldots, |v'|$

$$d_j = \begin{cases} [\![\delta_1]\!]_{\underline{A}}(\rho_2, \ldots, \rho_{|w'|}) & \text{if } j \in I^{\underline{s},v'}, \\ \theta^{\underline{s},w,p,z,q'}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } j \in I^{\underline{s},v'}, \end{cases}$$

for some $q' \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\phi}(q')} = ([\![\delta_1]\!]_{\underline{A}}(\rho_2, \ldots, \rho_{|w'|}))_j.$$

Therefore in this sub-basis it remains to show that

$$[\![\delta_1]\!]_{\underline{A}}(\rho_2, \ldots, \rho_{|w'|}) = [\![\delta]\!]_{\underline{A}}(0, \rho_2, \ldots, \rho_{|w'|}).$$

This is obvious.

Sub-Induction Hypothesis. Assume that for any fixed value $k \in \mathbf{N}$ that if $\chi_1 = \rho_1 = k$ then for each $l \notin I^{\underline{s},v'}$

$$([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi))_l = ([\![\delta]\!]_{\underline{A}}(\rho))_l$$

and for each $l \in I^{\underline{s},v'}$

$$([\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi))_l = \theta^{\underline{s},w,p,z,q''}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b)$$

for some $q'' \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\phi}(q')} = ([\![\delta]\!]_{\underline{A}}(\rho)_l.$$

Sub-Induction.

L.H.S.

$$[\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(\chi) = [\![\Diamond^{\beta,w,p}(\delta_2)]\!]_{\underline{A}}(k, \chi_2, \ldots, \chi_{|w'|}, [\![\Diamond^{\beta,w,p}(\delta)]\!]_{\underline{A}}(k, \chi_2, \ldots, \chi_{|w'|}))$$

by the definition of $\Diamond$, the definition of $[\![.]\!]_{\underline{A}}$ and the hypothesis on χ_1

$$= [\![\Diamond^{\beta,w,p}(\delta_2)]\!]_{\underline{A}}(k, \chi_2, \ldots, \chi_{|w'|}, d_1, \ldots, d_{|v'|})$$

by the Sub-Induction Hypothesis wherein $j = 1, \ldots, |v'|$

$$d_j = \begin{cases} [\![\delta]\!]_{\underline{A}}(\rho) & \text{if } j \notin I^{\underline{s},v'}, \\ \theta^{\underline{s},w,p,z,q''}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } j \in I^{\underline{s},v'}, \end{cases}$$

for some $q'' \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\varphi}(q'')} = (\llbracket \delta \rrbracket_{\underline{A}}(\rho))_j$$

$$= (d'_1, \ldots, d'_{|v'|})$$

by the Induction Hypothesis with $\delta' = \delta_2$, $w'' = \mathbf{t}u'v'$, $v'' = v'$, $\chi' = (k, \chi_2, \ldots, \chi_{|w'|}, d_1, \ldots, d_{|v'|})$ and $\rho' = (k, \rho_2, \ldots, \rho_{|w'|}, (\llbracket \delta \rrbracket_{\underline{A}}(\rho))_1, \ldots, \llbracket \delta \rrbracket_{\underline{A}}(\rho))_{|v'|})$ wherein for $j = 1, \ldots, |v'|$

$$d'_j = \begin{cases} \llbracket \delta_2 \rrbracket_{\underline{A}}(k, \rho_2, \ldots, \rho_{|w'|}, (\llbracket \delta \rrbracket_{\underline{A}}(\rho))_1, \ldots, \llbracket \delta \rrbracket_{\underline{A}}(\rho))_{|v'|}) & \text{if } j \in I^{\underline{s},v'}, \\ \theta^{\underline{s},w,p,z,q'''}(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{|w|}, b) & \text{if } j \in I^{\underline{s},v'}, \end{cases}$$

for some $q''' \in I^{\underline{s},w}$ such that

$$(P(a\{p/b\}))_{\overline{\varphi}(q''')} = (\llbracket \delta_2 \rrbracket_{\underline{A}}(k, \rho_2, \ldots, \rho_{|w'|}, (\llbracket \delta \rrbracket_{\underline{A}}(\rho))_1, \ldots, \llbracket \delta \rrbracket_{\underline{A}}(\rho))_{|v'|}))_j.$$

Therefore to complete the proof it remains to show that

$$\llbracket \delta_2 \rrbracket_{\underline{A}}(k, \rho_2, \ldots, \rho_{|w'|}, (\llbracket \delta \rrbracket_{\underline{A}}(\rho))_1, \ldots, \llbracket \delta \rrbracket_{\underline{A}}(\rho))_{|v'|}) = \llbracket \delta \rrbracket_{\underline{A}}(\rho).$$

Again, this is obvious.

□

Appendix B

Proof of Lemma 27 and Lemma 28

We now prove formally Lemma 27 and Lemma 28 that we used to prove the soundness and adequacy of PREQ in Chapter 5. As the proofs of both lemmata require several further intermediate results concerning the correctness the various sub-compilers of $\mathbb{C}^{\mathrm{PR}}$ and $\mathbb{C}^{\mathrm{PREQ}}$ respectively, we also state and prove these additional lemmata as appropriate. The reader should refer back to Chapter 5 for the definitions of the compilers that these results concern.

B.1 Proof of Lemma 27

To prove Lemma 27 we require the following five intermediate results.

B.1.1 Intermediate Lemmata

Let $\mathbb{C}^{\mathrm{T}}$ be defined as in Definition 57 on Page 155.

Lemma 63. *If*

$$\Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)$$

and $\mathbb{X} = \{x_1, \ldots, x_n\} \subseteq X$ *wherein* $x_i \in X_{s_i}$ *for* $i = 1, \ldots n \geq 1$ *are distinct variables then for each* $\tau \in T(\Sigma, \mathbb{X})_s$ *for any* $s \in S$

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad V_{\nu^{\mathbb{X}}(a)}(\tau) = [\![\mathbb{C}^T_{\Phi, \mathbb{X}, s_1 \cdots s_n, s}(\tau)]\!]_A(a).$$

Proof. Uniformly in $s \in S$ by simultaneous induction on the structural complexity of the term $\tau \in T(\Sigma, \mathbb{X})_s$.
Basis Cases.

(1) **Constants.** In this case $\tau = c_s$ for some $c \in \Sigma_{\lambda, s}$ for some $s \in S$.
L.H.S.

$$V_{\nu^{\mathbb{X}}(a)}(\tau) = c^A$$

by the definition of V

$$= [\![c_s^{s_1 \cdots s_n}]\!]_A(a)$$

by the definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau)]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{T}}$.

(2) **Variables.** In this case $\tau = x_i$ for some $x_i \in \mathbb{X}_s$ for some $s \in S$.
L.H.S.

$$V_{\nu^{\mathbb{X}}(a)}(\tau) = \nu^{\mathbb{X}}(a)(x_i)$$

by the definition of V

$$= a_i$$

by the definition of ν

$$= [\![\mathrm{U}_i^{s_1 \cdots s_n}]\!]_A(a)$$

by the definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau)]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{T}}$.

Induction Hypothesis. Assume for all $\tau \in T(\Sigma,\mathbb{X})_s$ for some $s \in S$ that for each term $\tau' \in T(\Sigma,\mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that

$$(\forall a \in A^{s_1 \cdots s_n}) \quad V_{\nu^{\mathbb{X}}(a)}(\tau') = [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s}(\tau)]\!]_A(a).$$

Induction.

(3) **Algebraic operations.** In this case $\tau = \sigma(\tau_1,\ldots,\tau_k)$ for some $\sigma \in \Sigma_{w,s}$ for any $w \in S^+$ and for any $s \in S$, and for some $\tau_i \in T(\Sigma,\mathbb{X})_{s_i'}$ for $i = 1,\ldots,k = |w|$ such that $s_1' \cdots s_k' = w$.
R.H.S.

$$[\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau)]\!]_A(a) = [\![\sigma \circ < \mathbb{C}^{\mathrm{T}}(\tau_1),\ldots,\mathbb{C}^{\mathrm{T}}(\tau_k) >]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{T}}$ with the hypothesis that $\tau \in T(\Sigma,\mathbb{X})$

and hence that $\sigma \in \Sigma$

$$= \sigma^A([\![\mathbb{C}^\Gamma(\tau_1)]\!]_A(a), \ldots, [\![\mathbb{C}^\Gamma(\tau_k)]\!]_A(a)$$

by the definition of $[\![.]\!]_A$

$$= \sigma^A(V_{\nu^{\mathbb{X}}(a)}(\tau_1), \ldots, V_{\nu^{\mathbb{X}}(a)}(\tau_k))$$

by k applications of the induction hypothesis

$$= V_{\nu^{\mathbb{X}}(a)}(\tau)$$

by definition.

□

Lemma 64. *If*

$$\Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)$$

such that for each

$$i \in \mathbb{P} = \{\iota^{-1}(j) \mid j \in InTermsOf(\Phi, \varsigma, \mathbb{F}\}$$

wherein $\mathbb{F} = \{f_{d,e} \mid d, e \in \mathbb{N}\}$ *and* $\iota(k) = \varsigma$ *we have*

$$[\![\mathbb{C}^{PR}_{u^{\iota(i)}, v^{\iota(i)}}(< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >)]\!]_A = [\![< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >]\!]_A$$

and $\mathbb{X} = \{x_1, \ldots, x_n\} \subseteq X$ *wherein* $x_i \in X_{s_i}$ *for* $i = 1, \ldots, n \geq 1$ *are distinct variable symbols then for each* $\tau \in T(\Sigma'', \mathbb{X})_s$ *for some* $s \in S$ *wherein*

$$\Sigma'' = \Sigma \cup (\mathbb{H} = \bigcup_{i \in \mathbb{P}} \{f_{i,1}, \ldots, f_{i,|\eta^R(i)|}\})$$

$$(\forall a = (a_1, \ldots, a_n) \in A^{s_1 \cdots s_n}) \quad V_{\nu^{\mathbb{X}}(a)}(\tau) = [\![\mathbb{C}^T_{\Phi, \mathbb{X}, s_1 \cdots s_n, s}(\tau)]\!]_A(a).$$

Proof. We proceed by induction on the structural complexity of the term τ uniformly in s.
Basis. We have two cases to consider:

(1) $\tau = c_s$; and

(2) $\tau = x_i \in X_s$.

Notice that in both these cases that $\tau \in T(\Sigma, \mathbb{X})_s$ and hence by Lemma 63 we have

$$V_{\nu^{\mathbb{X}}(a)}(\tau) = [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau)]\!]_A$$

as required.

Induction Hypothesis. Assume for any $\tau \in T(\Sigma'', \mathbb{X})_s$ for some $s \in S$ that for each scheme $\tau' \in T(\Sigma'', \mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that

$$(\forall a \in A^{s_1\cdots s_n}) \quad V_{\nu^{\mathbb{X}}(a)}(\tau') = [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau')]\!]_A(a).$$

Induction.

(3) Algebraic operations. In this case $\tau = \sigma(\tau_1, \ldots, \tau_k)$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$, and for some $\tau_i \in T(\Sigma', \mathbb{X})_{s'_i}$ for $i = 1, \ldots, k = |w|$ such that $s'_1 \cdots s'_k = w$. We have two sub-cases to consider:

(a) $\sigma \in \Sigma$; and

(b) $\sigma = f_{j,j'} \in \mathbb{H}$.

Sub-Case (a) $\sigma \in \Sigma$. This sub-sub-case follows by essentially the same argument as Case (3) of Lemma 63 and is omitted.

Sub-Case (b) $\sigma = f_{j,j'} \in \mathbb{H}$.

R.H.S.

$$\begin{aligned}
[\![\mathbb{C}^{\mathrm{T}}(\tau)]\!]_A(a) = [\![\mathrm{U}_{j'}^{\eta^R(j)} \circ \mathbb{C}^{\star\mathrm{PR}}(\phi_{\iota(j)}) \circ \\
< \mathbb{C}^{\mathrm{T}}(\tau_1), \\
\vdots, \\
\mathbb{C}^{\mathrm{T}}(\tau_k) >]\!]_A(a)
\end{aligned}$$

by the definition of $\mathbb{C}^{\mathrm{T}}$ with the hypothesis $\sigma = f_{j,j'}$

$$\begin{aligned}
= \Big([\![\mathbb{C}^{\star\mathrm{PR}}(\phi_{\iota(j)})]\!]_A(\\
[\![\mathbb{C}^{\mathrm{T}}(\tau_1)]\!]_A(a), \\
\vdots, \\
[\![\mathbb{C}^{\mathrm{T}}(\tau_k)]\!]_A(a)))\Big)_{j'}
\end{aligned}$$

by the definition of $[\![\cdot]\!]_A$

$$\begin{aligned}
= \Big([\![\mathbb{C}^{\star\mathrm{PR}}(\phi_{\iota(j)})]\!]_A(\\
V_{\nu^{\mathbb{X}}(a)}(\tau_1), \\
\vdots, \\
V_{\nu^{\mathbb{X}}(a)}(\tau_k))\Big)_{j'}
\end{aligned}$$

by k applications of the Induction Hypothesis

$$= \Big([\![\mathbb{C}^{\mathrm{PR}}_{u^{\iota(j)},v^{\iota(j)}}(< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(j) >)]\!]_A(
V_{\nu^{\mathbb{x}_{(a)}}}(\tau_1),
\vdots,
V_{\nu^{\mathbb{x}_{(a)}}}(\tau_k)) \Big)_{j'}$$

by the definition of $\mathbb{C}^{\mathrm{PR}}$

$$= \Big([\![< \phi_1, \ldots, \phi_l; \iota, \eta, \iota(j) >]\!]_A(
V_{\nu^{\mathbb{x}_{(a)}}}(\tau_1),
\vdots,
V_{\nu^{\mathbb{x}_{(a)}}}(\tau_k)) \Big)_{j'}$$

by hypothesis as $j \in \mathbb{P}$ by the definition of $\mathbb{H}$

$$= f^A_{j,j'}(V_{\nu^{\mathbb{x}_{(a)}}}(\tau_1), \ldots, V_{\nu^{\mathbb{x}_{(a)}}}(\tau_k))$$

by definition

$$= V_{\nu^{\mathbb{x}_{(a)}}}(\tau)$$

by definition.

□

Let $\mathbb{C}^{\mathbb{T}}$ be defined as in Definition 58 on Page 157.

Lemma 65. *Let $\Phi \in PREQ(\Sigma, X)$. If $\phi \in RPREQ(\Sigma', X)_{\mathbb{t}\,u,v}$ for some $u = (s_1 \cdots s_{|u|})$ and some $v = (s'_1 \cdots s'_{|v|}) \in S^+$ wherein Σ' is the common signature of Φ as defined as in Definition 50 and $\mathbb{X} = \{x_1, \ldots, x_{|u|}, t, Y_1, \ldots, Y_{|v|}\} \subseteq X$ wherein $x_i \in X_{s_i}$ for $i = 1, \ldots, |u|$, $t \in X_{\mathbb{n}}$, and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, |v|$ are distinct distinguished variable symbols then for each $\tau \in T(\Sigma, \mathbb{X})_s$ for some $s \in S$*

$$(\forall n \in T)(\forall a = (a_1, \ldots, a_n) \in A^u) \quad V_{\nu^{\mathbb{x}_{(a,n,[\![\phi]\!]_A(n,a))}}}(\tau) = [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a)).$$

Proof. By induction on the structural complexity of the term τ uniformly in $s \in S$.
Basis.

(1) *Constants.* In this case $\tau = c_s$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$.

L.H.S.

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = c^A$$

by the definition of V

$$= [\![c_s^{t\,u\,v}]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $[\![\cdot]\!]_A$

$$= [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$.

(2) *Variables.* In this case $\tau = x$ for some $x \in \mathbb{X}$ wherein x is of type s for some $s \in S$. We now have three sub-cases to consider:

(a) $x = t$.

(b) $x = x_i$ for some $i \in \{1, \ldots, |u|\}$.

(c) $x = Y_j$ for some $j \in \{1, \ldots, |v|\}$.

Sub-Case (a) $x = t$.

L.H.S.

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = \nu^{\mathbb{X}}(a, n, [\![\phi]\!]_A(n, a))(t)$$

by the definition of V and by hypothesis on τ

$$= n$$

by the definition of ν

$$= [\![\mathrm{U}_1^{\mathbf{t}\,u\,v}]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $[\![\cdot]\!]_A$

$$= [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,\mathbf{t}}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a)$$

by the definition of $\mathbb{C}^{\mathbb{T}}$.

Sub-Case (b) $x = x_i$ for some $i \in \{1, \ldots, |u|\}$.

L.H.S.

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = \nu^{\mathbb{X}}(a, n, [\![\phi]\!]_A(n, a))(x_i)$$

by the definition of V and by hypothesis on τ

$$= a_i$$

by the definition of ν

$$= [\![\mathrm{U}_{i+1}^{t\,u\,v}]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,u_i}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$.

Sub-Case (c) $x = Y_j$ for some $j \in \{1, \ldots, |v|\}$.

L.H.S.

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = \nu^{\mathbb{X}}(a, n, [\![\phi]\!]_A(n, a))(Y_j)$$

by the definition of V and by hypothesis on τ

$$= ([\![\phi]\!]_A(n, a))_j$$

by the definition of ν

$$= [\![\mathrm{U}_{|n|+2}^{t\,u\,v}]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,v_j}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$.

Induction Hypothesis. Assume for any scheme $\tau \in T(\Sigma, \mathbb{X})_s$ for some $s \in S$ that for each scheme $\tau' \in T(\Sigma, \mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau') = [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s'}(\tau')]\!]_A(n, a, [\![\phi]\!]_A(n, a)).$$

Induction.

(3) *Algebraic Operations.* In this case $\tau = \sigma(\tau_1, \ldots, \tau_k)$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$, and for some $\tau_i \in T(\Sigma, \mathbb{X})_{s'_i}$ for $i = 1, \ldots, k = |w|$ such that $s'_1 \cdots s'_k = w$.

R.H.S.

$[\![\mathbb{C}^{\mathbb{T}}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$

$$= [\![\sigma \circ < \mathbb{C}^{\mathbb{T}}(\tau_1), \ldots, \mathbb{C}^{\mathbb{T}}(\tau_k) >]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$ and the hypothesis that $\tau \in T(\Sigma, \mathbb{X})$ and hence that

$$\sigma \in \Sigma$$

$$= \sigma^A([\![< \mathbb{C}^{\mathbb{T}}(\tau_1)]\!]_A(n, a, [\![\phi]\!]_A(n, a)) \ldots, [\![< \mathbb{C}^{\mathbb{T}}(\tau_k)]\!]_A(n, a, [\![\phi]\!]_A(n, a)))$$

by the definition of $[\![.]\!]_A$

$$= \sigma^A(V_{\nu^{\mathfrak{A}}(a,n,[\![\phi]\!]_A(n,a))}(\tau_1), \ldots, V_{\nu^{\mathfrak{A}}(a,n,[\![\phi]\!]_A(n,a))}(\tau_k))$$

by k applications of the Induction Hypothesis

$$= V_{\nu^{\mathfrak{A}}(a,n,[\![\phi]\!]_A(n,a))}(\tau)$$

by the definition of V.

This concludes the Induction Step and concludes the proof.

□

Lemma 66. *Let $\phi \in RPREQ(\Sigma', \mathbb{X})_{\mathbf{t}\,u,v}$ for some $u = (s_1 \cdots s_{|u|}), v = (s'_1 \cdots s'_{|v|}) \in S^+$ wherein Σ' is the common signature of Φ as defined as in Definition 50 and $\mathbb{X} = \{x_1, \ldots, x_{|u|}, t, Y_1, \ldots, Y_{|v|}\} \subseteq X$ wherein $x_i \in X_s$, for $i = 1, \ldots |u|$, $t \in X_{\mathbf{n}}$, and $Y_j \in X_{s'_j}$ for $j = 1, \ldots, |v|$ are distinct distinguished variable symbols. If*

$$\Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)^{l,m,\iota,\eta,\varsigma}$$

is defined such that for each

$$i \in \mathbb{P} = \{\iota^{-1}(j) \mid j \in InTermsOf(\Phi, \varsigma, \mathbb{F}\} \subset \{1, \ldots, m\}$$

wherein $\mathbb{F} = \{f_{d,e} \mid d, e \in \mathbb{N}\}$ and $\iota(k) = \varsigma$ we have

$$[\![\mathbb{C}^{PR}_{u^{\iota(i)},v^{\iota(i)}}(< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >)]\!]_A = [\![< \phi_1, \ldots, \phi_l; \iota; \eta; \iota(i) >]\!]_A$$

then for each $\tau \in T(\Sigma'', \mathbb{X})$, for some $s \in S$ wherein

$$\Sigma'' = \Sigma' \cup (\mathbb{H} = \bigcup_{i \in \mathbb{P}} \{f_{i,1}, \ldots, f_{i,|\eta^R(i)|}\})$$

$$(\forall n \in T)\,(\forall a = (a_1, \ldots, a_n) \in A^u) \quad V_{\nu^{\mathfrak{A}}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a)).$$

Proof. By induction on the structural complexity of the term $\tau \in T(\Sigma'', \mathbb{X})$, uniformly in s.
Basis. We have two cases to consider:

(1) $\tau = c$ for some $c \in \Sigma_{\lambda,s}$; and

(2) $\tau = x$ for some $x \in \mathbb{X}$; that is, $x \in X_s$ for some $s \in \{s_1, \ldots, s_{|u|}, \mathbf{t}, s'_1, \ldots, s'_{|v|}\}$.

Notice that in both these cases that $\tau \in T(\Sigma, \mathbb{X})_s$ and therefore by Lemma 65 we have

$$V_{\mathbb{X}(a,n,[\![\phi]\!]_A(n,a))}(\tau) = [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

as required.

Induction Hypothesis. Assume for any term $\tau \in T(\Sigma'', \mathbb{X})_s$ for some $s \in S$ that for each term $\tau' \in T(\Sigma'', \mathbb{X})_{s'}$ for some $s' \in S$ of less structural complexity than τ that

$$V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau') = [\![\mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X},u,v,s'}(\tau')]\!]_A(n, a, [\![\phi]\!]_A(n, a)).$$

Induction.

(3) *Algebraic Operations.* In this case $\tau = \sigma(\tau_1, \ldots, \tau_k)$ for some $\sigma \in \Sigma''_{w,s}$ for some $w \in S^+$ and for some $s \in S$, and for some $\tau_i \in T(\Sigma'', \mathbb{X})_{s''_i}$ for $i = 1, \ldots, k = |w|$ such that $s''_1 \cdots s''_k = w$. We have two sub-cases to consider:

(a) $\sigma \in \Sigma$; and

(b) $\sigma = f_{p,p'} \in \mathbb{H}$.

Sub-case (a) $\sigma \in \Sigma$.

R.H.S.

$[\![\mathbb{C}^{\mathbb{T}}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$

$$= [\![\sigma \circ < \mathbb{C}^{\mathbb{T}}(\tau_1), \ldots, \mathbb{C}^{\mathbb{T}}(\tau_k) >]\!]_A(n, a, [\![\phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$ and the hypothesis that $\sigma \in \Sigma$

$$= \sigma^A([\![\mathbb{C}^{\mathbb{T}}(\tau_1)]\!]_A(n, a, [\![\phi]\!]_A(n, a)) \ldots, [\![\mathbb{C}^{\mathbb{T}}(\tau_k)]\!]_A(n, a, [\![\phi]\!]_A(n, a)))$$

by the definition of $[\![\cdot]\!]_A$

$$= \sigma^A(V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau_1), \ldots, V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau_k))$$

by k applications of the Induction Hypothesis

$$= V_{\nu^{\mathbb{X}}(a,n,[\![\phi]\!]_A(n,a))}(\tau)$$

by the definition of V.

Sub-case (b) $\sigma = f_{p,p'} \in \mathbb{H}$..

R.H.S.

$[\![\mathbb{C}^{\mathbb{T}}(\tau)]\!]_A(n, a, [\![\phi]\!]_A(n, a))$

$$= [\![\mathrm{U}^{\eta^R(p)}_{p'} \circ \mathbb{C}^{*\mathrm{PR}}_{\Phi}(\phi_{\iota(p)}) \circ < \mathbb{C}^{\mathbb{T}}(\tau_1), \ldots, \mathbb{C}^{\mathbb{T}}(\tau_k) >]\!]_A(n, a, [\![\Phi]\!]_A(n, a))$$

by the definition of $\mathbb{C}^{\mathbb{T}}$ and the hypothesis that $\sigma \in \mathbb{P}$

$$= ([\![\mathbb{C}^{*\mathrm{PR}}_{\Phi}(\phi_{\iota(p)})]\!]_A([\![\mathbb{C}^{\mathbb{T}}(\tau_1)]\!]_A(n, a, [\![\phi]\!]_A(n, a)) \ldots, [\![\mathbb{C}^{\mathbb{T}}(\tau_k)]\!]_A(n, a, [\![\phi]\!]_A(n, a))))_{p'}$$

by the definition of $[\![.]\!]_A$

$= ([\![\mathbb{C}^{*\mathrm{PR}}_{\Phi}(\phi_{\iota(p)})]\!]_A(V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_1),\ldots,V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_k)))_{p'}$

by k applications of the Induction Hypothesis

$= ([\![\mathbb{C}^{\mathrm{PR}}_{\Phi,u^{\iota(p)},v^{\iota(p)}}(<\phi_1,\ldots,\phi_l;\iota;\eta;\iota(p)>)]\!]_A(V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_1),\ldots,V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_k)))_{p'}$

by the definition of $\mathbb{C}^{\mathrm{PR}}$

$= ([\![<\phi_1,\ldots,\phi_l;\iota;\eta;\iota(p)>]\!]_A(V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_1),\ldots,V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_k)))_{p'}$

by hypothesis as $p \in \mathbb{P}$ by the definition of $\mathbb{H}$

$= f^A_{p,p'}(V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_1),\ldots,V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau_k))$

by definition

$= V_{\nu^{\mathfrak{A}}(a,n,[\phi]_A(n,a))}(\tau)$

by definition of V.

□

Let $\mathbb{C}^{*\mathrm{PR}}$ be defined as in Definition 59 on Page 158.

Lemma 67. *Let* $\Phi =< \phi_1,\ldots,\phi_l;\iota;\eta;\varsigma > \in PREQ(\Sigma,X)$. *If* $\phi \in RPREQ(\Sigma,X)_{u,v}$ *for some* $u,v \in S^+$ *then*

$$(\forall a = (a_1,\ldots,a_n) \in A^u) \quad [\![\phi]\!]_A(a) = [\![\mathbb{C}^{*PR}_{\Phi,u,v}(\phi)]\!]_A(a).$$

Proof. Uniformly in (u,v) by case analysis on the structural complexity of the scheme $\phi \in \mathrm{RPREQ}(\Sigma,X)$. We have four cases to consider:

(1) General Specifications. In this case

$$\phi \stackrel{def}{=} f(x_1,\ldots,x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1,\ldots,n \geq 1$ and for some $\tau \in T(\Sigma,\mathbb{X})_s$ for any $s \in S$ wherein $\mathbb{X} = \{x_1,\ldots,x_n\}$.

L.H.S.

$$[\![\phi]\!]_A(a) = V_{\nu^{\mathfrak{A}}(a)}(\tau)$$

by the definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau)]\!]_A(a)$$

by Lemma 63

$$= [\![\mathbb{C}^{*\mathrm{PR}}_{\Phi,s_1\cdots s_n,s}(\phi)]\!]_A(a)$$

by the definition of $[\![.]\!]_A$.

(2) Vector-Valued General Specifications. In this case

$$\phi \stackrel{def}{=} f(x_1,\ldots,x_n) = < \tau_1,\ldots,\tau_k >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1,\ldots,n \geq 1$ and for some $\tau_j \in T(\Sigma,\{x_1,\ldots,x_n\})_{s'_j}$ for any $s'_j \in S$ for $j = 1,\ldots,k > 1$. This case is similar to Case (1) in that it follows by k applications of Lemma 63 and is omitted.

(3) Primitive Recursive Specifications. In this case

$$\phi \stackrel{def}{=} f(0,x_1,\ldots,x_n) = \tau_1;$$
$$f(t+1,x_1,\ldots,x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1,\ldots,n \geq 1$ and for some $\tau_1 \in T(\Sigma,\mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma,\mathbb{X})_s$ for any $s \in S$ wherein $\mathbb{X} = \{x_1,\ldots,x_n\}$, $\mathbb{X}' = \mathbb{X} \cup \{t,Y\}$ and $t \in X_{\mathbf{n}}$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1,\ldots,n$.
Notice that as $u = \mathbf{t}\, s_1\cdots s_n$ and therefore $a = (a_1,a_2,\ldots,a_{1+|s_1\cdots s_n|})$. Consequently, we proceed by induction on the value of $a_1 \in A_{\mathbf{t}}$ using a' to represent $a_2,\ldots,a_{1+|s_1\cdots s_n|}$.
Basis. $a_1 = 0$.
L.H.S.

$$[\![\phi]\!]_A(0,a') = [\![\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s\mathbf{0}n,s}(\tau_1)]\!]_{\underline{A}}$$

by the definition of $[\![.]\!]_A$

$$= V_{\nu^{\mathbb{X}}(a')}(\tau_1)$$

by Lemma 63

$$= [\![*(\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau_1),\mathbb{C}^{\mathrm{T}}_{\Phi,\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(0,a')$$

by the definition of $\mathbb{C}^{*\mathrm{PR}}$

$$= [\![\mathbb{C}^{*\mathrm{PR}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\phi)]\!]_A(0,a')$$

by the definition of $[\![.]\!]_A$.
Induction Hypothesis. Assume for some fixed value $n \in A_{\mathbf{t}}$ that

$$[\![\phi]\!]_A(n,a') = [\![\mathbb{C}^{*\mathrm{PR}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\phi)]\!]_A(n,a').$$

Induction Step. We must show that

$$[\![\phi]\!]_A(n+1,a') = [\![\mathbb{C}^{\star\mathrm{PR}}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\phi)]\!]_A(n+1,a').$$

R.H.S.

$$[\![\mathbb{C}^{\star\mathrm{PR}}_{u,v}(\phi)]\!]_A(n+1,a') = [\![*(\mathbb{C}^{\Gamma}_{\Phi,\mathbb{X},s_1\cdots s_n,s}(\tau_1), \mathbb{C}^{\mathbb{T}}_{\Phi,\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(n+1,a')$$

by the definition of $\mathbb{C}^{\star\mathrm{PR}}$

$$= [\![\mathbb{C}^{\mathbb{T}}(\tau_2)]\!]_A(n,a',[\![*(\mathbb{C}^{\Gamma}(\tau_1),\mathbb{C}^{\mathbb{T}}(\tau_2))]\!]_A(n,a'))$$

by the definition of $[\![\cdot]\!]_A$

$$= [\![\mathbb{C}^{\mathbb{T}}(\tau_2)]\!]_A(n,a',[\![\mathbb{C}^{\star\mathrm{PR}}_{\Phi,u,s}(\phi)]\!]_A(n,a'))$$

by the definition of $\mathbb{C}^{\star\mathrm{PR}}$

$$= [\![\mathbb{C}^{\mathbb{T}}(\tau_2)]\!]_A(n,a',[\![\phi]\!]_A(n,a'))$$

by the Induction Hypothesis

$$= V_{\nu^{\mathbb{X}'}(a',n,[\![\phi]\!]_A(n,a'))}(\tau_2)$$

by Lemma 65

$$= [\![\phi]\!]_A(n+1,a)$$

by definition.

(4) Vector-Valued Primitive Recursive Specifications. In this case

$$\phi \stackrel{def}{=} f(0,x_1,\ldots,x_n) =< \tau_{1,1},\ldots,\tau_{1,m} >;$$
$$f(t+1,x_1,\ldots,x_n) =< \tau_{2,1},\ldots,\tau_{2,m} >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1,\ldots,n \geq 1$, for some $\tau_{1,j} \in T(\Sigma,\{x_1,\ldots,x_n\})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma,\{x_1,\ldots,x_n,t,Y_1,\ldots,Y_m\})_{s'_j}$ for any $s'_j \in S$ for $j = 1,\ldots,m > 1$ wherein $t \in X_n$ and $Y_1 \in X_{s'_1},\ldots,Y_m \in X_{s'_m}$ are distinguished variable symbols distinct from x_i for $i = 1,\ldots,n$.
This case is a simple generalization of Case(3) that follows by essentially the same argument and is omitted.

□

B.1.2 Proof of Lemma 27

We are now in a position to prove Lemma 27. For convenience we first re-state the lemma.

Lemma 27. *If*

$$\Phi =< \phi_1, \ldots, \phi_l; \iota; \eta; \varsigma > \in PREQ(\Sigma, X)_{u,v}$$

for some $u, v \in S^+$ *then*

$$(\forall a \in A^u) \quad [\![\Phi]\!]_A(a) = [\![\mathbb{C}^{PR}(\Phi)]\!]_A(a).$$

Proof. By induction uniformly in (u, v) on the size

$$q = |\text{InTermsOf}(\Phi, \varsigma, \mathbb{F})| \in \mathbb{N}$$

wherein $\mathbb{F} = \{f_{p,q} \mid p, q \in \mathbb{N}\}$.
Basis $q = 0$.. Notice that since by hypothesis $q = 0$ we have $i \notin \text{DefOver}(\phi_\varsigma, \mathbb{F})$ for $i = 1, \ldots, m$ and hence $\phi_\varsigma \in \text{RPREQ}(\Sigma, X)_{u,v}$. We calculate as follows:

$$[\![\Phi]\!]_A(a) = [\![\phi_\varsigma]\!]_A(a)$$

by definition

$$= [\![\mathbb{C}^{\star\text{PR}}_{\Phi,u,v}(\phi_\varsigma)]\!]_A(a)$$

by Lemma 67

$$= [\![\mathbb{C}^{\text{PR}}_{\Phi,u,v}(\Phi)]\!]_A(a)$$

by definition.
Induction Hypothesis. Assume for each $\Phi' \in \text{PREQ}(\Sigma, X)_{u',v'}$ for some $u', v' \in S^+$ such that

$$|\text{InTermsOf}(\Phi', \varsigma', \mathbb{F})| \leq k$$

for some fixed $k \in \mathbb{N}$ that

$$(\forall a' \in A^{u'}) \quad [\![\Phi']\!]_A(a') = [\![\mathbb{C}^{\text{PR}}(\Phi')]\!]_A(a').$$

Induction Step. We must show that for each $\Phi'' \in \text{PREQ}(\Sigma, X)_{u'',v''}$ for some $u'', v'' \in S^+$ such that

$$|\text{InTermsOf}(\Phi'', \varsigma'', \mathbb{F})| = k + 1$$

that

$$(\forall a'' \in A^{u''}) \quad [\![\Phi'']\!]_A(a'') = [\![\mathbb{C}^{\text{PR}}(\Phi'')]\!]_A(a'').$$

We calculate by reducing each side of the equality to a common term.
L.H.S.

$$[\![\Phi'']\!]_A(a'') = [\![\phi''_{\varsigma''}]\!]_A(a'')$$

by definition.

R.H.S.

$$[\![\mathbb{C}^{\mathrm{PR}}(\Phi'')]\!]_A(a'') = [\![\mathbb{C}^{\star\mathrm{PR}}_{\Phi,\iota'',\nu''}(\phi''_{\varsigma''})]\!]_A(a'')$$

by definition.

We now proceed by case analysis on the structural complexity of $\phi''_{\varsigma''} \in \mathrm{RPREQ}(\Sigma', X)$ wherein by definition $\Sigma' = \Sigma \cup \mathbb{H}$ wherein

$$\mathbb{H} = \bigcup_{i \in \mathbb{P}} \{f_{i,1}, \ldots, f_{i,|\eta^R(i)|}\}$$

wherein

$$\mathbb{P} = \{\iota^{-1}(j) \mid j \in \mathrm{InTermsOf}(\Phi, \varsigma'', \mathbb{F})\}.$$

We have four cases to consider:

(1) General Specifications. In this case

$$\phi''_{\varsigma''} \stackrel{def}{=} f(x_1, \ldots, x_n) = \tau$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau \in T(\Sigma', \mathbb{X})_s$ for any $s \in S$ wherein $\mathbb{X} = \{x_1, \ldots, x_n\}$.

L.H.S.

$$[\![\phi''_{\varsigma}]\!]_A(a'') = V_{\nu_{\mathbb{X}}(a'')}(\tau)$$

by definition.

Notice now that as by hypothesis

$$q = |\mathrm{InTermsOf}(\Phi'', \varsigma'', \mathbb{F}')| = k + 1$$

it must be the case that $\mathrm{DefOver}(\phi''_{\varsigma''}, \mathbb{F}') \supseteq \{p\}$ for some $p \in \{1, \ldots, m\}$ therefore by Lemma 23 for each $j \in \mathrm{InTermsOf}(\Phi'', \varsigma'', \mathbb{F})$ we have

$$|\mathrm{InTermsOf}(\Phi'', j, \mathbb{F})| < q \leq k$$

Therefore by the induction hypothesis for each $i \in \mathbb{P} = \{\iota''^{-1}(j) \mid j \in \mathrm{InTermsOf}(\Phi'', \varsigma'', \mathbb{F})\}$ we have

$$[\![\mathbb{C}^{\mathrm{PR}}(< \phi'', \ldots, \phi''_{l''}; \iota''; \eta''; \iota''(i) >)]\!]_A = [\![< \phi'', \ldots, \phi''_{l''}; \iota''; \eta''; \iota''(i) >]\!]_A$$

and therefore by Lemma 64

$$V_{\nu_{\mathbb{X}}}(a'')(\tau) = [\![\mathbb{C}^{\mathrm{T}}_{\Phi'', \mathbb{X}'', s_1 \cdots s_n, s}(\tau)]\!]_A(a'')$$

and

$$[\![\mathbb{C}^{\mathrm{T}}_{\Phi'', \mathbb{X}'', s_1 \cdots s_n, s}(\tau)]\!]_A(a'') = [\![\mathbb{C}^{\star\mathrm{PR}}(\phi''_{\varsigma''})]\!]_A(a'')$$

by definition of $\mathbb{C}^{\star\mathrm{PR}}$ as required.

(2) Vector-Valued General Specifications. In this case

$$\phi''_{\varsigma''} \stackrel{def}{=} f(x_1, \ldots, x_n) = < \tau_1, \ldots, \tau_m >$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 1$ and for some $\tau_j \in T(\Sigma', \mathbb{X})_{s'_j}$ for any $s'_j \in S$ for $j = 1, \ldots, m > 1$ wherein $\mathbb{X} = \{x_1, \ldots, x_n\}$.
This case is a simple generalization of Case (1) in that it can be reduced to showing that for each $\tau_i \in T(\Sigma', \mathbb{X})_{s_i}$ for some $s_i \in S$ for $i = 1, \ldots, k$ that

$$(\forall a \in A^{s_1 \cdots s_n}) \quad (V_{\nu^{\mathbb{X}}(a)}(\tau_1), \ldots, V_{\nu^{\mathbb{X}}(a)}(\tau_k)) = [\![< \mathbb{C}^{\mathrm{T}}(\tau_1), \ldots, \mathbb{C}^{\mathrm{T}}(\tau_k) >]\!]_A(a)$$

and is omitted.

(3) Primitive Recursive Specifications. In this case

$$\phi''_{\varsigma''} \stackrel{def}{=} f(0, x_1, \ldots, x_n) = \tau_1;$$
$$f(t+1, x_1, \ldots, x_n) = \tau_2$$

for some distinct $x_i \in X_{s_i}$ for $i = 1, \ldots, n \geq 0$ and for some $\tau_1 \in T(\Sigma', \mathbb{X})_s$ and for some $\tau_2 \in T(\Sigma', \mathbb{X}')_s$ for any $s \in S$ wherein $\mathbb{X} = \{x_1, \ldots, x_n\}$ and $\mathbb{X}' = \{x_1, \ldots, x_n, t, Y\}$ wherein $t \in X_{\mathbf{n}}$ and $Y \in X_s$ are distinguished variable symbols distinct from x_i for $i = 1, \ldots, n$.
We proceed by sub-induction on the value $r = a''_1 \in A^{\mathbf{n}}$ using b to represent $a''_2, \ldots, a''_n$.
Sub-Basis. $r = 0$.
L.H.S.

$$[\![\phi''_{\varsigma''}]\!]_A(0, b) = V_{\nu^{\mathbb{X}}(b)}(\tau_1)$$

by definition of $[\![.]\!]_A$

$$= [\![\mathbb{C}^{\mathrm{T}}_{\Phi'', \mathbb{X}, s_1 \cdots s_n, s}(\tau_1)]\!]_A(b)$$

using essentially the same argument as in Case (1)

$$= [\![*(\mathbb{C}^{\mathrm{T}}_{\Phi'', \mathbb{X}, s_1 \cdots s_n, s}(\tau_1), \mathbb{C}^{\mathrm{T}}_{\Phi'', \mathbb{X}', s_1 \cdots s_n, s, s}(\tau_2)]\!]_A(0, b)$$

by definition of PR

$$= [\![\mathbb{C}^{*\mathrm{PR}}(\phi''_{\varsigma''})]\!]_A(0, b)$$

by definition of $\mathbb{C}^{*\mathrm{PR}}$.
Sub-Induction Hypothesis. Assume for some fixed value $r \in \mathrm{N}$ that

$$[\![\phi''_{\varsigma''}]\!]_A(r, b) = [\![\mathbb{C}^{*\mathrm{PR}}_{\Phi''}(\phi''_{\varsigma''})]\!]_A(r, b).$$

Sub-Induction. We must show that

$$[\![\phi''_{\varsigma''}]\!]_A(r+1, b) = [\![\mathbb{C}^{*\mathrm{PR}}(\phi''_{\varsigma''})]\!]_A(r+1, b).$$

L.H.S.

$$[\![\phi''_{\varsigma''}]\!]_A(r+1,b) = V_{\nu^{\mathbb{X}'}(b,r,[\![\phi''_{\varsigma''}]\!]_A(r,b))}(\tau_2)$$

by the definition of $[\![.]\!]_A$.
Notice again that as by hypothesis

$$q = |\text{InTermsOf}(\Phi'', \varsigma'', \mathbb{F}')| = k+1$$

it must be the case that $\text{DefOver}(\phi''_{\varsigma''}, \mathbb{F}') \supseteq \{p\}$ for some $p \in \{1,\ldots,m\}$ therefore by Lemma 23 for each $j \in \text{InTermsOf}(\Phi'', \varsigma'', \mathbb{F})$we have

$$|\text{InTermsOf}(\Phi'', j, \mathbb{F})| < q \leq k$$

Therefore by the induction hypothesis for each $i \in \mathbb{P} = \{\iota''^{-1}(j) \mid j \in \text{InTermsOf}(\Phi'', \varsigma'', \mathbb{F})\}$ we have

$$[\![\mathbb{C}^{\text{PR}}(< \phi'', \ldots, \phi''_{l''}; \iota''; \eta''; \iota''(i) >)]\!]_A = [\![< \phi'', \ldots, \phi''_{l''}; \iota''; \eta''; \iota''(i) >]\!]_A$$

and therefore by Lemma 66

$$\begin{aligned} V_{\nu^{\mathbb{X}'}(b,r,[\![\phi''_{\varsigma''}]\!]_A(r,b))}(\tau_2) &= [\![\mathbb{C}^{\mathbb{T}}_{\Phi'',\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(r,b,[\![\phi''_{\varsigma''}]\!]_A(r,b)) \\ &= [\![\mathbb{C}^{\mathbb{T}}_{\Phi'',\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(r,b,[\![\mathbb{C}^{*\text{PR}}(\phi''_{\varsigma''})]\!]_A(r,b)) \end{aligned}$$

by the Sub-induction Hypothesis

$$= [\![\mathbb{C}^{\mathbb{T}}_{\Phi'',\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(r,b,[\![*(\mathbb{C}^{\mathbb{T}}(\tau_1),\mathbb{C}^{\mathbb{T}}(\tau_2)]\!]_A(r,b))$$

by the definition of $\mathbb{C}^{*\text{PR}}$ and by hypothesis on $\phi''_{\varsigma''}$

$$= [\![*(\mathbb{C}^{\mathbb{T}}_{\Phi'',\mathbb{X},s_1\cdots s_n,s}(\tau_1),\mathbb{C}^{\mathbb{T}}_{\Phi'',\mathbb{X}',s_1\cdots s_n,s,s}(\tau_2)]\!]_A(r+1,b)$$

by the definition of PR

$$= [\![\mathbb{C}^{*\text{PR}}(\phi''_{\varsigma''})]\!]_A(r+1,b)$$

by the definition of $\mathbb{C}^{*\text{PR}}$ and by hypothesis on $\phi''_{\varsigma''}$ as required.

(4) Vector-Valued Primitive Recursive Specifications. *In this case*

$$\begin{aligned} \phi''_{\varsigma''} \stackrel{def}{=} f(0,x_1,\ldots,x_n) &=< \tau_{1,1},\ldots,\tau_{1,m} >; \\ f(t+1,x_1,\ldots,x_n) &=< \tau_{2,1},\ldots,\tau_{2,m} > \end{aligned}$$

for some distinct $x_i \in X_{s_i}$ for $i = 1,\ldots,n \geq 0$, for some $\tau_{1,j} \in T(\Sigma',\mathbb{X})_{s'_j}$ and for some $\tau_{2,j} \in T(\Sigma',\mathbb{X}')_{s'_j}$ for any $s'_j \in S$ for $j = 1,\ldots,d > 1$ wherein $\mathbb{X} = \{x_1,\ldots,x_n\}$ and $\mathbb{X}' = \mathbb{X} \cup \{t, Y_1,\ldots,Y_m\}$ wherein $t \in X_n$ and $Y_j \in X_{s'_j}$ for $j = 1,\ldots,d$ are distinguished variable symbols distinct from x_i for $i = 1,\ldots,n$.

This case is a simple generalization of Case (3) in that it can be reduced to an induction wherein the basis case requires that we show

$$(\forall b \in A^{u''_2 \cdots u''_{|u''|}}) \quad (V_{\nu \mathfrak{x}(b)}(\tau_{1,1}), \ldots, V_{\nu \mathfrak{x}(b)}(\tau_{1,d})) = (\llbracket \mathbb{C}^{\mathrm{T}}(\tau_{1,1}) \rrbracket_A, \ldots, \llbracket \mathbb{C}^{\mathrm{T}}(\tau_{1,d}) \rrbracket_A$$

and an induction case that requires that we show for each $r \in \mathbb{N}$

$$(V_{\nu \mathfrak{x}'(b,r,[\phi''_{\zeta''}]_A(r,b))}(\tau_{2,1}), \ldots, V_{\nu \mathfrak{x}'(b,r,[\phi''_{\zeta''}]_A(r,b))}(\tau_{2,d})) =$$

$$(\llbracket \mathbb{C}^{\mathrm{T}}(\tau_{2,1}) \rrbracket_A(r, b\llbracket \phi''_{\zeta''} \rrbracket_A(r,b)), \ldots, \llbracket \mathbb{C}^{\mathrm{T}}(\tau_{1,d}) \rrbracket_A(r, b\llbracket \phi''_{\zeta''} \rrbracket_A(r,b)))$$

and is therefore omitted.

□

B.2 Proof of Lemma 28

To prove the Lemma 28 we require the following nine intermediate results.

B.2.1 Intermediate Lemmata

Let $\mathbb{C}^{\star \mathrm{PREQ}}$ be defined as in Definition 61 on Page 162.

Lemma 68. *Let* $\mathbb{X} = \{x_1, \ldots, x_{|u|}\}$ *for some* $u \in S^+$ *wherein* $x_i \in u_i$ *for* $i = 1, \ldots, |u|$. *If* $\alpha \in PR_D(\Sigma)_{u,v}$ *for some* $v \in S^+$ *and* $\phi = \mathbb{C}^{\star PREQ}_{\mathbb{X},u,v,n}(\alpha) \in T(\Sigma, \mathbb{X})_{v_n}$ *for some* $n \in \{1, \ldots, |v|\}$ *then*

$$(\forall a = (a_1, \ldots, a_{|u|}) \in A^u) \quad \llbracket \alpha \rrbracket_A(a) = V_{\nu \mathfrak{x}(a)}(\phi).$$

Proof. By induction on the structural complexity of the scheme α uniformly in (u, v).

(1) **Constant Functions.** In this case $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $w \in S^+$ and for some $s \in S$ and hence $n = 1$.
R.H.S.

$$V_{\nu \mathfrak{x}(a)}(\phi) = V_{\nu \mathfrak{x}(a)}(c^w)$$

by hypothesis on α and by the definition of $\mathbb{C}^{\star \mathrm{PREQ}}$

$$= c^A$$

by the definition of V

$$= \llbracket \alpha \rrbracket_A(a)$$

by the definition of $\llbracket \cdot \rrbracket_A$.

(2) **Algebraic Operations.** In this case $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$. Again notice that $n = 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a)}(\phi) = V_{\nu^{\mathbb{X}}(a)}(\sigma(x_1, \ldots, x_{|u|}))$$

by hypothesis on α and by the definition of $\mathbb{C}^{*\mathrm{PREQ}}$

$$= \sigma^A(\nu^{\mathbb{X}}(a)(x_1), \ldots, \nu^{\mathbb{X}u}(a)(x_{|u|}))$$

by the definition of V

$$= \sigma^A(a_1, \ldots, a_{|u|})$$

by the definition of ν

$$= [\![\alpha]\!]_A(a)$$

by the definition of $[\![.]\!]_A$.

(3) **Projection Functions.** In this case $\alpha = \mathrm{U}_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$. As with Cases (1) and (2) notice that in this case $n = 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a)}(\phi) = V_{\nu^{\mathbb{X}}(a)}(x_i)$$

by hypothesis on α and by the definition of $\mathbb{C}^{*\mathrm{PREQ}}$

$$= \nu^{\mathbb{X}u}(a)(x_i)$$

by the definition of V

$$= a_i$$

by the definition of ν

$$= [\![\alpha]\!]_A(a)$$

by the definition of $[\![.]\!]_A$.

Induction Hypothesis. Assume for any scheme $\alpha \in \mathrm{PR}_D(\Sigma)_{u,v}$ for some $u, v \in S^+$ that for any scheme $\alpha' \in \mathrm{PR}_D(\Sigma)_{u',v'}$ for some $u', v' \in S^+$ of less structural complexity than α that for some $\mathbb{X}' = \{x'_1, \ldots, x'_{|u'|}\}$ wherein $x'_j \in X_{u'_j}$ for $j = 1, \ldots, |u'|$ that if $\phi' = \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X}',u',v',k}$ for some $k \in \{1, \ldots, |v'|\}$ then

$$(\forall a' \in A^{u'}) \quad [\![\alpha']\!]_A(a') = V_{\nu \mathbb{X}'(a')}(\phi').$$

Induction.

(4) **Vectorization.** In this case $\alpha = < \alpha_1, \ldots, \alpha_m >$ wherein $\alpha_i \in \mathrm{PR}_D(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1, \ldots, m \geq 1$. This case follows easily by m applications of the Induction Hypothesis and is omitted.

(5) **Composition.** In this case $\alpha = \alpha_2 \circ \alpha_1$ wherein $\alpha_1 \in PR(\Sigma)_{u,w}$ and $\alpha_2 \in PR(\Sigma)_{w,v}$ for some $u, v, w \in S^+$. Again this case follows easily by the Induction Hypothesis and the definition of $\mathbb{C}^{*\mathrm{PREQ}}$ and is omitted.

□

Lemma 69. *Let $\mathbb{X} = \{x_1, \ldots, x_{|u|}, t, Y_1, \ldots, Y_{|v|}\}$ for some $u, v \in S^+$ such that $x_i \in X_{u_i}$ for $i = 1, \ldots, |u|$ and $Y_j \in X_{v_j}$ for $j = 1, \ldots, |v|$ are distinguished variable symbols and $t \in X_{\mathbf{t}}$ is a distinguished variables symbol, and let $\gamma \in PR(\Sigma)_{\mathbf{t}\,w,v}$ for some $w \in S^+$. If $\alpha \in PR_D(\Sigma)_{\mathbf{t}\,u\,v,v'}$ for some $v' \in S^+$ and $\phi = \mathbb{C}^{*PREQ}_{\mathbb{X},\mathbf{t}\,u\,v,v',n}(\alpha) \in T(\Sigma, \mathbb{X})_{v'_n}$ for some $n \in \{1, \ldots, |v'|\}$ then*

$$(\forall k \in T)\,(\forall a = (a_1, \ldots, a_{|u|}) \in A^u)\,(\forall a' = (a'_1, \ldots, a'_{|w|}) \in A^w)$$

$$([\![\alpha]\!]_A(k, a, [\![\gamma]\!]_A(k, a')))_n = V_{\nu \mathbb{X}(a, k, [\![\gamma]\!]_A(k, a'))}(\phi).$$

Proof. By induction on the structural complexity of the scheme α uniformly in $(\mathbf{t}\ u\ v, v')$.

(1) **Constant Functions.** In this case $\alpha = c^{\mathbf{t}\,u\,v}$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$ and hence $n = 1$.

R.H.S.

$$V_{\nu \mathbb{X}(a, k, [\![\gamma]\!]_A(k, a'))}(\phi) = V_{\nu \mathbb{X}(a, k, [\![\gamma]\!]_A(k, a'))}(c^{\mathbf{t}\,u\,v})$$

by the definition of $\mathbb{C}^{*\mathrm{PREQ}}$ and by the hypothesis on α

$$= c^A$$

by the definition of V

$$= [\![c^{\mathbf{t}\,u\,v}]\!]_A(k, a, [\![\gamma]\!]_A(k, a'))$$

by the definition of $[\![.]\!]_A$.

(2) Algebraic Operations. In this case $\alpha = \sigma$ for some $\sigma \in \Sigma_{t\,u\,v,s}$ for some $s \in S$. Again notice that $n = 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\phi) = V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\sigma(t, x_1, \ldots, x_{|u|}, Y_1, \ldots, Y_{|v|}))$$

by the definition of $\mathbb{C}^{\star\text{PREQ}}$ and by the hypothesis on α

$$= \sigma^A(\nu^{\mathbb{X}}(t), \nu^{\mathbb{X}}(x_1), \ldots, \nu^{\mathbb{X}}(x_{|u|}), \nu^{\mathbb{X}}(Y_1), \ldots, \nu^{\mathbb{X}}(Y_{|v|}))$$

by the definition of V

$$= \sigma^A(k, a_1, \ldots, a_{|u|}, \left([\![\gamma]\!]_A(k,a')\right)_1, \ldots, \left([\![\gamma]\!]_A(k,a')\right)_{|v|})$$

by the definition of ν

$$= [\![\alpha]\!]_A(k, a, [\![\gamma]\!]_A(k,a'))$$

by the definition of $[\![.]\!]_A$.

(3) Projection Functions. In this case $\alpha = \mathrm{U}_i^{t\,u\,v}$ for some i with $1 \leq i \leq |\mathbf{t}\ u\ v|$. Again as with Case (1) and Case (2) in this case $n = 1$. We have three sub-cases to consider:

(a) $i = 1$,
(b) $1 < i \leq |u| + 1$, and
(c) $i > |u| + 1$.

Sub-Case (a) $i = 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\phi) = V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(t)$$

by the definition of $\mathbb{C}^{\star\text{PREQ}}$ and by the hypothesis on α

$$= \nu'(a, k, [\![\gamma]\!]_A(k,a'))(t)$$

by the definition of V

$$= k$$

by the definition of ν

$$= [\![\mathrm{U}_1^{t\,u\,v}]\!]_A(k, a, [\![\gamma]\!]_A(k,a'))$$

by the definition of $[\![.]\!]_A$.

Sub-Case (b) $1 < i \leq |u| + 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\phi) = V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(x_j)$$

by the definition of $\mathbb{C}^{\star\text{PREQ}}$ and by the hypothesis on α wherein $j = i - 1$

$$= \nu'(a, k, [\![\gamma]\!]_A(k, a'))(x_j)$$

by the definition of V

$$= a_j$$

by the definition of ν

$$= [\![\mathrm{U}_{j+1}^{\mathrm{t\,u\,v}}]\!]_A(k, a, [\![\gamma]\!]_A(k, a'))$$

by the definition of $[\![\cdot]\!]_A$

$$= [\![\mathrm{U}_{i}^{\mathrm{t\,u\,v}}]\!]_A(k, a, [\![\gamma]\!]_A(k, a'))$$

by the definition of j.

Sub-Case (c) $i > |u| + 1$.

R.H.S.

$$V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\phi) = V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(Y_j)$$

by the definition of $\mathbb{C}^{\star\text{PREQ}}$ and by the hypothesis on α wherein $j = i - (|u| + 1)$

$$= \nu'(a, k, [\![\gamma]\!]_A(k, a'))(Y_j)$$

by the definition of V

$$= \left([\![\gamma]\!]_A(k, a')\right)_j$$

by the definition of ν

$$= [\![\mathrm{U}_{j+|u|+1}^{\mathrm{t\,u\,v}}]\!]_A(k, a, [\![\gamma]\!]_A(k, a'))$$

by the definition of $[\![\cdot]\!]_A$

$$= [\![\mathrm{U}_{i}^{\mathrm{t\,u\,v}}]\!]_A(k, a, [\![\gamma]\!]_A(k, a'))$$

by the definition of j.

Induction Hypothesis. Assume for any scheme $\alpha \in \mathrm{PR}_D(\Sigma)_{t_u v,v'}$ for some $v' \in S^+$ that for any scheme $\alpha' \in \mathrm{PR}_E(\Sigma)_{t_u v,v''}$ for some $v'' \in S^+$ of less structural complexity than α that if $\phi' = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},t_u v,v'',n}(\alpha) \in T(\Sigma,\mathbb{X})_{v''_n}$ for some $n \in \{1,\ldots,|v''|\}$ then

$$(\forall k \in T)\,(\forall a = (a_1,\ldots,a_{|u|}) \in A^u)\,(\forall a' = (a'_1,\ldots,a'_{|w|}) \in A^w)$$

$$[\![\alpha]\!]_A(k,a,[\![\gamma]\!]_A(k,a')) = V_{\nu^{\mathbb{X}}(a,k,[\![\gamma]\!]_A(k,a'))}(\phi').$$

Induction.

(4) **Vectorisation.** In this case $\alpha =< \alpha_1,\ldots,\alpha_m >$ wherein $\alpha_i \in PR(\Sigma)_{t_u v,s_i}$ for some $s_i \in S$ for $i = 1,\ldots,m \geq 1$. This case follows easily by m applications of the Induction Hypothesis and is omitted.

(5) **Composition.** In this case $\alpha = \alpha_2 \circ \alpha_1$ where $\alpha_1 \in PR(\Sigma)_{t_u v,w'}$ and $\alpha_2 \in PR(\Sigma)_{w',v''}$ for some $w' \in S^+$. Again this case follows easily by the Induction Hypothesis and the definition of $\mathbb{C}^{\star\mathrm{PREQ}}$ and is omitted.

□

Let $\mathbb{C}^{\diamond\mathrm{PREQ}}$ be defined as in Definition 62 on Page 163.

Lemma 70. *Let $n \in \mathbb{N}$ let $\alpha \in PR(\Sigma)_{u,v}$ for some $u,v \in S^+$ and let $\phi = \mathbb{C}^{\diamond PREQ}_{n,u,v}(\alpha) \in RPREQ(\Sigma',X)$ wherein Σ' is defined as in Definition 50. If either $NPPRSS'(\alpha) = 0$ or for each $i = 1,\ldots,NPPRSS(\alpha)$ and for each $j = 1,\ldots,|CoDom(PPRSS'(\alpha,i))|$ the function symbol $f_{r^a(i)+n,j} \in \Sigma'$ is defined over A by $f^A_{r^a(i)+n,j} = \left([\![PPRSS'(\alpha,i)]\!]_A\right)_j$ then*

$$(\forall a \in A^u) \quad [\![\alpha]\!]_A(a) = [\![\phi]\!]_A(a).$$

Proof. By case analysis on the structural complexity of the scheme α uniformly in (u,v). We have six cases to consider:

Basis Cases.

(1) **Constant Functions.** In this case $\alpha = c^w$ for some $c \in \Sigma_{\lambda,s}$ for some $s \in S$ and for some $w \in S^+$. We proceed as follows:

R.H.S.

$$[\![\phi]\!]_A(a) = [\![f(x_1,\ldots,x_{|w|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha))]\!]_A(a)$$

by the definition of $\mathbb{C}^{\diamond\mathrm{PREQ}}$ wherein $\mathbb{X} = \{x_1,\ldots,x_{|w|}\}$

$$= [\![f(x_1,\ldots,x_{|w|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\alpha)]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{PR}_E}$

$$= V_{\nu^{\mathbb{X}}(a)}(\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},w,s,1}(\alpha))$$

by the definition of $[\![.]\!]_A$

$$= c^A$$

by Lemma 68 and by the hypothesis on α

$$= [\![\alpha]\!]_A(a)$$

by hypothesis on α and by the definition of $[\![.]\!]_A$.

(2) **Algebraic Operations.** In this case $\alpha = \sigma$ for some $\sigma \in \Sigma_{w,s}$ for some $w \in S^+$ and for some $s \in S$. This case is similar to Case (1) and is omitted.

(3) **Projection Functions.** In this case $\alpha = \mathrm{U}_i^w$ for some $w \in S^+$ and for some i with $1 \leq i \leq |w|$. Again this case is similar to Case (1) and is omitted.

Induction.

(4) **Vectorisation.** In this case $\alpha =< \alpha_1, \ldots, \alpha_m >$ wherein $\alpha_i \in PR(\Sigma)_{u,s_i}$ for some $u \in S^+$ and for some $s_i \in S$ for $i = 1, \ldots, m \geq 1$. As with Cases (2) and (3) this case follows by a similar argument to Case (1) and is omitted.

(5) **Composition.** In this case $\alpha = \alpha_2 \circ \alpha_1$ where $\alpha_1 \in PR(\Sigma)_{u,w}$ and $\alpha_2 \in PR(\Sigma)_{w,v}$ for some $u, v, w \in S^+$. Again, as with Cases (2), (3) and (4) this case follows by a similar argument to Case (1) and is omitted.

(6) **Simultaneous Primitive Recursion.** In this case $\alpha = *(\alpha_1, \alpha_2)$ where $\alpha_1 \in PR(\Sigma)_{u,v}$ and $\alpha_2 \in PR(\Sigma)_{\mathbf{n}\,u\,v,v}$ for some $u, v \in S^+$. We proceed by induction on the value of $a_1 \in \mathbf{N}$. For simplicity we will assume that $|v| = 1$ and leave the simple generalization of the case where $|v| > 1$ to the reader.
Basis. $a_1 = 0$ (using a' to represent $a_2, \ldots, a_{|u|}$).
R.H.S.

$$[\![\phi]\!]_A(0, a') = [\![f(0, x_1, \ldots, x_{|u|}) = \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_1))$$
$$f(t+1, x_1, \ldots, x_{|u|}) = \mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X}',\mathbf{t}\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2))]\!]_A(0, a')$$

by hypothesis on α and by the definition of $\mathbb{C}^{\diamond\mathrm{PREQ}}$

$$= V_{\nu \mathbf{x}_{(a')}}(\mathbb{C}^{*\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_1)))$$

by the definition of $[\![.]\!]_A$

$$= ([\![\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_1)]\!]_A(a'))_1$$

by Lemma 68

$$= ([\![(\alpha_1)]\!]_A(a'))_1$$

by Lemma 22 as we have $\mathrm{NPRSS}(\mathbb{C}^{\mathrm{PR}_E}(\alpha_2)) = 0$

and hence by Lemma 21 $\mathbb{C}^{\mathrm{PR}_E}(\alpha_2) \in \mathrm{PR}_D(\Sigma)$

$$= [\![\alpha]\!]_A(0, a')$$

by the definition of $[\![.]\!]_A$.

Induction Hypothesis. Assume for some fixed value $k \in \mathbf{N}$ that

$$[\![\alpha]\!]_A(k, a') = [\![\phi]\!]_A(k, a').$$

Induction. We must show that

$$[\![\alpha]\!]_A(k+1, a') = [\![\phi]\!]_A(k+1, a').$$

R.H.S.

$$[\![\phi]\!]_A(k+1, a') = [\![f(0, x_1, \ldots, x_{|u|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X},u,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_1))$$
$$f(t+1, x_1, \ldots, x_{|u|}) = \mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2))]\!]_A(k+1, a')$$

by hypothesis on α and by the definition of $\mathbb{C}^{\diamond\mathrm{PREQ}}$

$$= V_{\nu^{\mathbb{X}'}(a', k, [\![\phi]\!]_A(k, a'))}(\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2)))$$

by the definition of $[\![.]\!]_A$

$$= V_{\nu^{\mathbb{X}'}(a', k, [\![\alpha]\!]_A(k, a'))}(\mathbb{C}^{\star\mathrm{PREQ}}_{\mathbb{X}',t\,u\,v,v,1}(\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2)))$$

by the Induction Hypothesis

$$= ([\![\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2)]\!]_A(a', k, [\![\alpha]\!]_A(k, a')))_1$$

by Lemma 69 with $\gamma = \alpha$ as by Lemma 21 $\mathbb{C}^{\mathrm{PR}_E}_n(\alpha_2) \in \mathrm{PR}_D(\Sigma)$

$$= ([\![\alpha_2]\!]_A(a', k, [\![\alpha]\!]_A(k, a')))_1$$

by Lemma 22 as we have $\mathrm{NPRSS}(\mathbb{C}^{\mathrm{PR}_E}(\alpha_2)) = 0$

and hence by Lemma 21 $\mathbb{C}^{\mathrm{PR}_E}(\alpha_2) \in \mathrm{PR}_D(\Sigma)$

$$= [\![\alpha]\!]_A(k+1, a')$$

by the definition of $[\![.]\!]_A$ as required.

This concludes Case (6) and concludes the proof.

□

Let $\mathbb{C}^{\bullet \mathrm{PREQ}}$ be defined as in Definition 63 on Page 164.

Lemma 71. *If $\alpha \in PR(\Sigma)_{u,v}$ for some $u, v \in S^+$ such that $NPPRSS(\alpha) = 0$ then for each $e \in \mathbb{N}^+$*

$$\Phi = \mathbb{C}^{\bullet PREQ}_{e,u,v}(\alpha) \in PREQ(\Sigma, X)_{u,v}.$$

Proof. Notice that by the well-definedness of $\mathbb{C}^{\bullet \mathrm{PREQ}}$ we immediately have $\Phi \in \mathrm{PREQ}_1(\Sigma, X)_{u,v}$ and therefore it is sufficient to show that Φ is totally-defined.

As by hypothesis $\mathrm{NPPRSS}(\alpha) = 0$ by Lemma 13 we have $\mathrm{NPPRSS}'(\alpha) = 0$ and hence

$$\bigcup_{j=1}^{j=l=1} \mathrm{InTermsOf}(\Phi, j, \Sigma' - \Sigma) = \varnothing$$

and therefore the fact that Φ is totally-defined is immediate by definition.

□

Let $\mathbb{C}^{\nabla \mathrm{PREQ}}$ be defined as in Definition 64 on Page 164.

Lemma 72. *Let*

$$\Phi = < \phi_1, \ldots, \phi_l > \in PREQ(\Sigma, X)^{l,m,\iota,\eta,1}_{u,v}$$

for some $u, v \in S^+$ be standard. If for some $e \in \{2, \ldots, l\}$ we have

$$InTermsOf(\Phi, e, \mathbb{F}) \bigcap \{e+1, \ldots, l\} = \varnothing$$

wherein

$$\mathbb{F} = \{f_{q,p} \mid q, p \in \mathbb{N}\}$$

(we denote this property by $P^e(\Phi)$) then if we define $\Phi^{\overline{e}}$ by

$$\Phi^{\overline{e}} = < \phi_1, \ldots, \phi_e; \iota'; \eta'; 1 >$$

wherein

$$\iota' = \iota[\{1, \ldots, e\}]$$

and

$$\eta' = \eta[\{1, \ldots, e\}]$$

then $\Phi^{\overline{e}}$ satisfies the following:

(a)

We have

$$\Phi^{\overline{e}} \in PREQ(\Sigma, X)^{e,e,\iota',\eta',1}_{u,v},$$

(b)

$\Phi^{\overline{e}}$ is totally-defined and standard, and

(c)

For any Σ-algebra A

$$[\![\Phi^{\overline{e}}]\!]_A = [\![\Phi]\!]_A.$$

Proof. Omitted.

□

Lemma 73. *Let*

$$\Phi =< \phi_1, \ldots, \phi_l > \in PREQ(\Sigma, X)^{l,m,\iota,\eta,1}_{u,v}$$

for some $m > l$ and for some $u, v \in S^+$. If for some $j \in \{2, \ldots, m\}$ we have

(a)

$$\iota = \{j \mapsto 1, j+1 \mapsto 2, \ldots, m \mapsto m-(j-1)\},$$

and

(b)

$$\eta = \{j \mapsto (u^1, v^1), j+1 \mapsto (u^2, v^2), \ldots, j+l-1 \mapsto (u^l, v^l)\}$$

wherein (u^i, v^i) is the type of ϕ_i for $i = 1, \ldots, l$

(we denote this property by $Q^j(\Phi)$) then if we define $\Phi^{j\downarrow}$ by

$$\Phi^{j\downarrow} =< \phi'_1, \ldots, \phi'_l; \iota'; \eta'; 1 >$$

wherein for $k = 1, \ldots, l$

$$\phi'_k = \phi_k[f_{q,p}/f_{(q-j)+1,p}]^{q=m}_{q=j},$$

$$\iota' = \{1 \mapsto 1, 2 \mapsto 2, \ldots, m-(j-1) \mapsto m-(j-1)\}$$

and

$$\eta' = \{1 \mapsto (u^1, v^1), 2 \mapsto (u^2, v^2), \ldots, m-(j-1) \mapsto (u^l, v^l)\}$$

then the following hold:

(1) We have

$$\Phi^{j\downarrow} \in PREQ(\Sigma, X)_{u,v},$$

(2) $\Phi^{j\downarrow}$ is totally-defined and standard, and

(3) For any Σ-algebra A

$$[\![\Phi]\!]_A = [\![\Phi^{j\downarrow}]\!]_A.$$

Proof. Omitted.

□

Lemma 74. *If*

$$\Phi =< \phi_1, \ldots, \phi_l > \in PREQ(\Sigma, X)_{u,v}$$

for some $l \geq 2$ *and for some* $u, v \in S^+$ *such that for each* $k = 2, \ldots, l$

$$(\bigcup_{i=k}^{i=l} InTermsOf(\Phi, i, \mathbb{F})) \bigcap \{k\} = \varnothing$$

where

$$\mathbb{F} = \{f_{q,p} \mid q, p \in \mathbb{N}\}$$

(we denote this property by $Q(\Phi)$*) then for* $d = 2, \ldots, l$ *If we define* $\Phi^{\vec{d}}$ *by*

$$\Phi^{\vec{d}} =< \phi_d, \ldots, \phi_l; \iota'; \eta'; 1 >$$

wherein

$$\iota' = \{d \rightarrowtail 1, d+1 \rightarrowtail 2, \ldots, l \rightarrowtail l-(d-1)\},$$

$$\eta' = \{d \rightarrowtail (u^d, v^d), d+1 \rightarrowtail (u^{d+1}, v^{d+1}), \ldots, l \rightarrowtail (u^{l-(d-1)}, v^{l-(d-1)})\},$$

wherein for $j = d, \ldots, l-(d-1)$ (u^j, v^j) *is the type of* ϕ_j *then the following hold:*

(1)

$$\Phi^{\vec{d}} \in PREQ(\Sigma, X)_{u^d, v^d};$$

(2)

$$[\![< \phi_1, \ldots, \phi_l; d >]\!]_A = [\![\Phi^{\vec{d}}]\!]_A.$$

Proof. Omitted. □

Lemma 75. *If*

$$\Phi =< \phi_1, \ldots, \phi_l >= \mathbb{C}^{\nabla PREQ}_{u,v}(\alpha)$$

for some $\alpha \in PR(\Sigma)_{u,v}$ *for some* $u, v \in S^+$ *then for each* $d \in \{2, \ldots, l\}$

$$[\![< \phi_1, \ldots, \phi_l; d >]\!]_A = [\![\mathbb{C}^{\nabla PREQ}_1(PPRSS(\alpha, d-1))]\!]_A.$$

Proof. First, notice that by Lemma 26 on Page 165 for $i = 2, \ldots, l$ we have

$$\phi_i = \mathbb{C}^{\bullet \mathrm{PREQ}}_i(\mathrm{PPRSS}(\alpha, i-1)).$$

Also, notice that as $Q(\Phi)$ holds (see Lemma 74) we have

(1) $$[\![< \phi_1, \ldots, \phi_l; d >]\!]_A = [\![\Phi^{\vec{d}}]\!]_A.$$

In addition, notice that by definition $Q^d(\Phi^{\overline{d}})$ holds (see Lemma 73) and therefore

$$[\![\Phi^{\overline{d}}]\!]_A = [\![(\Phi^{\overline{d}})^{d\downarrow}]\!]_A \tag{2}$$

wherein

$$\begin{aligned}(\Phi^{\overline{d}})^{d\downarrow} &=< \phi'_d, \phi'_{d+1}, \ldots, \phi'_l > \\ &=< \phi_d[f_{q,p}/f_{(q-d)+1,p}]_{q=d}^{q=l}, \phi_{d+1}[f_{q,p}/f_{(q-d)+1,p}]_{q=d}^{q=l}, \ldots, \phi_l[f_{q,p}/f_{(q-d)+1,p}]_{q=d}^{q=l} >\end{aligned}$$

by definition. Furthermore, by Lemma 26 we have

$$\mathbb{C}_1^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d-1)) =< \phi''_1, \ldots, \phi''_{l''} >$$

wherein $l'' = \mathrm{NPPRSS}(\mathrm{PPRSS}(\alpha), d-1)+1, \phi''_1 = \mathbb{C}_1^{\bullet \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d-1))$ and for $k = 2, \ldots, l''$

$$\begin{aligned}\phi''_k &= \mathbb{C}_k^{\bullet \mathrm{PREQ}}(\mathrm{PPRSS}(\mathrm{PPRSS}(\alpha, d-1), k-1) \\ &= \mathbb{C}_k^{\bullet \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d+k-2)\end{aligned}$$

by Lemma 16; that is, in $[\![\mathbb{C}^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d-1))]\!]_A$ we have

$$f_{i,q}^A = ([\![\mathrm{PPRSS}(\alpha, d+k-2)]\!]_A)_q$$

for $k = 1, \ldots, l''$ and for $q = 1, \ldots, |\eta''(k)|$.

Finally, notice that, $P^{l''}((\Phi^{\overline{d}})^{d\downarrow})$ holds (see Lemma 72) and therefore we have

$$[\![(\Phi^{\overline{d}})^{d\downarrow}]\!]_A = [\![((\Phi^{\overline{d}})^{d\downarrow})^{\overline{l''}}]\!]_A, \tag{3}$$

but

$$((\Phi^{\overline{d}})^{d\downarrow})^{\overline{l''}}$$

is

$$\mathbb{C}_1^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d-1))$$

and hence by Equations (1), (2) and (3) we have

$$[\![< \phi_1, \ldots, \phi_l; d >]\!]_A = [\![\mathbb{C}^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, d-1))]\!]_A$$

as required.

□

B.2.2 Proof of Lemma 28

We are now in a position to prove Lemma 28. For convenience we first re-state the lemma.

Lemma 28. *For each $\alpha \in PR(\Sigma)_{u,v}$ for some $u, v \in S^+$ if $\Phi = \mathbb{C}^{PREQ}(\alpha)$ then*

$$(\forall a \in A^u) \quad [\![\alpha]\!]_A = [\![\Phi]\!]_A.$$

Proof. By induction on the number $n = \mathrm{NPPRSS}(\alpha)$.

Basis Cases. We consider two basis cases:

(1) $n = 0.$; and

(2) $n = 1.$.

Basis Case (1).
R.H.S.

$$[\![\mathbb{C}^{\mathrm{PREQ}}(\alpha)]\!]_A(a) = [\![\mathbb{C}_1^{\nabla\mathrm{PREQ}}(\alpha)]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{PREQ}}$

$$= [\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha) >]\!]_A(a)$$

by Lemma 26

$$= [\![\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha)]\!]_A(a)$$

by the definition of $[\![.]\!]_A$

$$= [\![\alpha]\!]_A(a)$$

by Lemma 26 as by hypothesis $\mathrm{NPRSS}(\alpha) = 0$ and hence by Lemma 13 $\mathrm{NPPRSS}'(\alpha) = 0$.
Basis Case (2).

$$[\![\mathbb{C}^{\mathrm{PREQ}}(\alpha)]\!]_A(a) = [\![\mathbb{C}_1^{\nabla\mathrm{PREQ}}(\alpha)]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{PREQ}}$

$$= [\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1)) >]\!]_A(a)$$

by Lemma 26

$$= [\![\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha)]\!]_A(a)$$

by the definition of $[\![.]\!]_A$.
Notice now that by Lemma 26 $\mathbb{C}^{\nabla\mathrm{PREQ}}(\alpha)$ is standard and therefore

$$\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha) \in \mathrm{RPREQ}(\Sigma', X)$$

wherein

$$\Sigma' = \Sigma \cup \{f_{2,1}\} \cup \{f_{2,2}\} \cup \cdots \cup \{f_{2,|\eta^R(1)|}\}$$

and by the definition of $[\![.]\!]_A$ for $j = 1, \ldots, |\eta^R(2)|$

$$f_{2,1}^A = \left([\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\alpha); 2 >]\!]_A\right)_j.$$

Also notice that by Lemma 17 we have $r^{\alpha}(1) = 1$ and hence

$$f^{A}_{2,j} = f^{A}_{r^{\alpha}(1)+1,j}$$

and by Lemma 13 we have $\mathrm{NPPRSS}'(\alpha) = 1$. Therefore, if we can show that

$$[\![< \mathbb{C}_1^{\diamond \mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond \mathrm{PREQ}}(\alpha); 2 >]\!]_A = [\![\mathrm{PPRSS}'(\alpha, 1)]\!]_A$$

then by Lemma 70 we have

$$[\![\mathbb{C}_1^{\diamond \mathrm{PREQ}}(\alpha)]\!]_A = [\![\alpha]\!]_A$$

as required. We calculate as follows:

$$[\![< \mathbb{C}_1^{\diamond \mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond \mathrm{PREQ}}(\alpha); 2 >]\!]_A = [\![\mathbb{C}_1^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1))]\!]_A$$

by Lemma 75

$$= [\![\mathbb{C}^{\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1))]\!]_A$$

by the definition of $\mathbb{C}^{\mathrm{PREQ}}$

$$= [\![\mathrm{PPRSS}(\alpha, 1)]\!]_A$$

by Case (1) as by hypothesis $\mathrm{NPPRSS}(\alpha) = 1$ and hence by

Lemma 15 we have $\mathrm{NPPRSS}(\mathrm{PPRSS}(\alpha, 1)) = 0$

$$= [\![\mathrm{PPRSS}(\alpha, r^{\alpha}(1))]\!]_A$$

by Lemma 17

$$= [\![\mathrm{PPRSS}'(\alpha, 1)]\!]_A$$

by Definition 43.

Induction Hypothesis. Assume for some scheme $\alpha' \in \mathrm{PR}(\Sigma)_{u',v'}$ for any $u', v' \in S^+$ such that $\mathrm{NPPRSS}(\alpha') \leq k$ for some fixed $k \in \mathbf{N}$ that

$$(\forall a' \in A^{u'}) \quad [\![\alpha']\!]_A = [\![\mathbb{C}^{\mathrm{PREQ}}(\alpha')]\!]_A.$$

Induction. We must show that for some scheme $\alpha'' \in \mathrm{PR}(\Sigma)_{u'',v''}$ for any $u'', v'' \in S^+$ such that $\mathrm{NPPRSS}(\alpha'') = k + 1$ that

$$(\forall a'' \in A^{u''}) \quad [\![\alpha'']\!]_A = [\![\mathbb{C}^{\mathrm{PREQ}}(\alpha'')]\!]_A.$$

We proceed as follows:

R.H.S.

$$[\![\mathbb{C}^{\mathrm{PREQ}}(\alpha'')]\!]_A(a) = [\![\mathbb{C}_1^{\nabla \mathrm{PREQ}}(\alpha'')]\!]_A(a)$$

by the definition of $\mathbb{C}^{\mathrm{PREQ}}$

$$= [\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha''), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha'', 1)), \ldots, \mathbb{C}_{k+2}^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha'', k+1)) >]\!]_A(a)$$

by Lemma 26

$$= [\![\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha'')]\!]_A(a)$$

by the definition of $[\![.]\!]_A$.

Notice again that by Lemma 26

$$\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha'') \in \mathrm{RPREQ}(\Sigma'', X)$$

wherein

$$\begin{aligned}\Sigma'' = \Sigma &\bigcup \{f_{2,1}\} \bigcup \{f_{2,2}\} \bigcup \cdots \bigcup \{f_{2,|\eta^R(2)|}\} \\ &\bigcup \{f_{3,1}\} \bigcup \{f_{3,2}\} \bigcup \cdots \bigcup \{f_{3,|\eta^R(3)|}\} \\ &\vdots \\ &\bigcup \{f_{k+2,1}\} \bigcup \{f_{k+2,2}\} \bigcup \cdots \bigcup \{f_{k+2,|\eta^R(k+2)|}\}\end{aligned}$$

and by the definition of $[\![.]\!]_A$ for $i = 2, \ldots, k+2$ and for $j = 1, \ldots, |\eta^R(i)|$ we have

$$f_{i,j}^A = \left([\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1)), \ldots, \mathbb{C}_{k+2}^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, k+1)); i >]\!]_A\right)_j.$$

In addition notice that

$$\{r^{\alpha''}(1) + 1, \ldots, r^{\alpha''}(\mathrm{NPPRSS}'(\alpha'')) + 1\} \subseteq \{2, \ldots, \mathrm{NPPRSS}(\alpha'') + 1 = k + 2\}$$

and hence

$$\sum_{i=1}^{i=k+2} \sum_{j=1}^{j=\eta^R(i)} \{f_{i,j}\} = \sum_{i=1}^{i=\mathrm{NPPRSS}'(\alpha'')} \sum_{j=1}^{j=|\eta^R(r^{\alpha''}(i)+1)|} \{f_{r^{\alpha''}(i)+1,j}\}$$

therefore by the definition of $[\![.]\!]_A$ for $i = 1, \ldots, \mathrm{NPPRSS}'(\alpha'')$ and for $j = 1, \ldots, |\eta^R(r^{\alpha''}(i) + 1)|$ we have

$$f_{r^{\alpha''}(i)+1,j}^A = \left([\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1)), \ldots, \mathbb{C}_{k+2}^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, k+1)); r^{\alpha''}(i)+1 >]\!]_A\right)_j$$

Consequently, if we can show that for $i = 1, \ldots, \mathrm{NPPRSS}'(\alpha'')$

$$[\![< \mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1)), \ldots, \mathbb{C}_{k+2}^{\diamond\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, k+1)); r^{\alpha''}(i) >]\!]_A = [\![\mathrm{PPRSS}'(\alpha'', i)]\!]_A$$

then by Lemma 70 we have

$$[\![\mathbb{C}_1^{\diamond\mathrm{PREQ}}(\alpha'')]\!]_A(a'') = [\![\alpha'']\!]_A(a'')$$

as required. We calculate as follows.

$$[\![< \mathbb{C}_1^{\diamond \mathrm{PREQ}}(\alpha), \mathbb{C}_2^{\diamond \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, 1)), \ldots, \mathbb{C}_{k+2}^{\diamond \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha, k+1)); r^{\alpha''}(i) >]\!]_A$$

$$= [\![\mathbb{C}_1^{\nabla \mathrm{PREQ}}(\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i)))]\!]_A$$

by Lemma 75

$$= [\![\mathbb{C}^{\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i)))]\!]_A$$

by the definition of $\mathbb{C}^{\mathrm{PREQ}}$. Notice now that as by definition $r^{\alpha''}(i) \in \{1, \ldots, \mathrm{NPPRSS}(\alpha'')\}$ by Lemma 15 we have

$$\mathrm{NPPRSS}(\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i))) < \mathrm{NPPRSS}(\alpha'') = k + 1;$$

that is,

$$\mathrm{NPPRSS}(\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i))) \leq k.$$

Therefore,

$$[\![\mathbb{C}^{\mathrm{PREQ}}(\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i)))]\!]_A = [\![\mathrm{PPRSS}(\alpha'', r^{\alpha''}(i))]\!]_A$$

by the Induction Hypothesis

$$= [\![\mathrm{PPRSS}'(\alpha'', i)]\!]_A$$

by Definition 43 as required.

□

Appendix C

An Abridged Version of the RS-Flip-Flop Verification

We now present the theorem and an abridged version of the proof that was generated by our implementation of AV (see Chapter 8) as part of the automatic verification of the RS-Flip-Flop. In particular, notice the existential quantification on the 'initial' boolean values b_1 and b_2.

We have used the symbol '$\bigwedge\!\!\!\bigvee$' to indicate where parts of the proof have been omitted – the full proof is approximately one-hundred-and-fifty pages long.

Theorem. *Let E be the given system of equations. If $\underline{A}$ is some $\underline{S}$-sorted $\underline{\Sigma}$-algebra such that $\underline{A} \cong I(\underline{\Sigma}, E)$ then*

$$(\forall n_1 \in \underline{A}^{\mathrm{nat}})\,(\exists b_1 \in \underline{A}^{\mathrm{bool}})\,(\exists b_2 \in \underline{A}^{\mathrm{bool}})\,(\forall B_1 \in \underline{A}^{\underline{\mathrm{bool}}})\,(\forall B_2 \in \underline{A}^{\underline{\mathrm{bool}}})$$

$$\mathit{fflopspec}^{\underline{A}}(n_1, B_2, B_1) = \mathit{fflopimpl}^{\underline{A}}(n_1, B_2, B_1, b_2, b_1).$$

Proof. By induction on n_1.

Basis. $n_1 = 0^{\underline{A}}$.

We calculate by case analysis on the values of b_1 and b_2. There is one Sub-case to consider: $b_1 = \mathit{true}^{\underline{A}}$ and $b_2 = \mathit{false}^{\underline{A}}$.

Calculating for Sub-case 1.1. $b_1 = \mathit{true}^{\underline{A}}$ and $b_2 = \mathit{false}^{\underline{A}}$.

L.H.S.

$\mathit{fflopspec}^{\underline{A}}(0^{\underline{A}}, B_2, B_1)$

$= \mathit{false}^{\underline{A}}$

by applying $\mathit{fflopspec}^{\underline{A}}(0^{\underline{A}}, s11^{\underline{A}}, s21^{\underline{A}}) = \mathit{false}^{\underline{A}}$ with: $s11^{\underline{A}}$ as B_2 and $s21^{\underline{A}}$ as B_1.

R.H.S.

$\mathit{fflopimpl}^{\underline{A}}(0^{\underline{A}}, B_2, B_1, \mathit{false}^{\underline{A}}, \mathit{true}^{\underline{A}})$

$= f176c1^{\underline{A}}(f184c1^{\underline{A}}(0^{\underline{A}}, B_2, B_1, \mathit{false}^{\underline{A}}, \mathit{true}^{\underline{A}}), f184c2^{\underline{A}}(0^{\underline{A}}, B_2, B_1, \mathit{false}^{\underline{A}}, \ldots$

$\ldots \mathit{true}^{\underline{A}}), f184c3^{\underline{A}}(0^{\underline{A}}, B_2, B_1, \mathit{false}^{\underline{A}}, \mathit{true}^{\underline{A}}), f184c4^{\underline{A}}(0^{\underline{A}}, B_2, B_1, \mathit{false}^{\underline{A}}, \ldots$

$\ldots true^{\underline{A}}), f184c5^{\underline{A}}(0^{\underline{A}}, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), f184c6^{\underline{A}}(0^{\underline{A}}, B_2, B_1, false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}}), f184c7^{\underline{A}}(0^{\underline{A}}, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))$

by applying $fflopimp1^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}) = \ldots$
$\ldots f176c1^{\underline{A}}(f184c1^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c2^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}$

$\ldots f184c3^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c4^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), \ldots$

$\ldots f184c5^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c6^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), \ldots$

$\ldots f184c7^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}))$ with: $nx12^{\underline{A}}$ as $0^{\underline{A}}$, $sb1x12^{\underline{A}}$ as B_2, $sb2x12^{\underline{A}}$ as B_1, $b1x12^{\underline{A}}$ as $false^{\underline{A}}$ and $b2x12^{\underline{A}}$ as $true^{\underline{A}}$.
$\wedge$
$\vee$
$= false^{\underline{A}}$

by applying $f2c1^{\underline{A}}(sb1x52^{\underline{A}}, sb2x52^{\underline{A}}, sb3x52^{\underline{A}}, sb4x52^{\underline{A}}, b1x52^{\underline{A}}, b2x52^{\underline{A}}) = b1x52^{\underline{A}}$ with: $sb1x52^{\underline{A}}$ as B_2, $sb2x52^{\underline{A}}$ as B_1, $sb3x52^{\underline{A}}$ as B_2, $sb4x52^{\underline{A}}$ as B_1, $b1x52^{\underline{A}}$ as $false^{\underline{A}}$ and $b2x52^{\underline{A}}$ as $true^{\underline{A}}$.

Induction Hypothesis. Assume for all n_1 that:

$$fflopspec^{\underline{A}}(n_1, B_2, B_1) = fflopimp1^{\underline{A}}(n_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}).$$

Induction Step. We must now show that:

$$fflopspec^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1) = fflopimp1^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, true^{\underline{A}}).$$

Calculating for Sub-case 2.1. $b_1 = true^{\underline{A}}$ and $b_2 = false^{\underline{A}}$.
L.H.S.
$fflopspec^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1)$
$= dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), false^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), true^{\underline{A}})), true^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), true^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots fflopspec^{\underline{A}}(n_1, B_2, B_1))))$

by applying $fflopspec^{\underline{A}}(succ^{\underline{A}}(t_1), s11^{\underline{A}}, s21^{\underline{A}}) = dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(t_1, s11^{\underline{A}}), \ldots$
$\ldots false^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(t_1, s21^{\underline{A}}), true^{\underline{A}})), true^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(t_1, s11^{\underline{A}}), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(t_1, s21^{\underline{A}}), \ldots$

$\ldots false^{\underline{A}})), false^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(t_1, s11^{\underline{A}}), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(t_1, s21^{\underline{A}}), true^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots fflopspec^{\underline{A}}(t_1, s11^{\underline{A}}, s21^{\underline{A}}))))$ with: t_1 as n_1, $s11^{\underline{A}}$ as B_2 and $s21^{\underline{A}}$ as B_1.

$\wedge$
$\vee$

$= dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), false^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), true^{\underline{A}})), true^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), true^{\underline{A}}), eq^{\underline{A}}(eval^{\underline{A}}(n_1, B_1), true^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_7, y_7) = y_7$ with: x_7 as $succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))$ and y_7 as $mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}})))$.

R.H.S.

$fflopimp1^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})$
$= f176c1^{\underline{A}}(f184c1^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), f184c2^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}), f184c3^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), \ldots$
$\ldots f184c4^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), f184c5^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, \ldots$
$\ldots false^{\underline{A}}, true^{\underline{A}}), f184c6^{\underline{A}}(succ^{\underline{A}}(n_1), B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), f184c7^{\underline{A}}(succ^{\underline{A}}(n_1), \ldots$
$\ldots B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))$

by applying $fflopimp1^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}) = \ldots$
$\ldots f176c1^{\underline{A}}(f184c1^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c2^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}$

$\ldots f184c3^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c4^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), \ldots$

$\ldots f184c5^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), f184c6^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}), \ldots$

$\ldots f184c7^{\underline{A}}(nx12^{\underline{A}}, sb1x12^{\underline{A}}, sb2x12^{\underline{A}}, b1x12^{\underline{A}}, b2x12^{\underline{A}}))$ with: $nx12^{\underline{A}}$ as $succ^{\underline{A}}(n_1)$, $sb1x12^{\underline{A}}$ as B_2, $sb2x12^{\underline{A}}$ as B_1, $b1x12^{\underline{A}}$ as $false^{\underline{A}}$ and $b2x12^{\underline{A}}$ as $true^{\underline{A}}$.

$\wedge$
$\vee$

$= not^{\underline{A}}(or^{\underline{A}}(eval^{\underline{A}}(n_1, B_2), not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, \ldots$
$\ldots B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), eval^{\underline{A}}(n_1, B_1)))))$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_7, y_7) = y_7$ with: x_7 as $succ^{\underline{A}}(n_1)$ and y_7 as n_1.

We now proceed by case analysis on the values of $eval^{\underline{A}}(n_1, B_2)$ and $eval^{\underline{A}}(n_1, B_1)$. There are four Sub-sub-cases to consider:

Sub-sub-case 2.1.1. $eval^{\underline{A}}(n_1, B_2) = true^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = true^{\underline{A}}$.

Sub-sub-case 2.1.2. $eval^{\underline{A}}(n_1, B_2) = false^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = true^{\underline{A}}$.

Sub-sub-case 2.1.3. $eval^{\underline{A}}(n_1, B_2) = true^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = false^{\underline{A}}$.

Sub-sub-case 2.1.4. $eval^{\underline{A}}(n_1, B_2) = false^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = false^{\underline{A}}$.

Calculating for Sub-sub-case 2.1.1. $eval^{\underline{A}}(n_1, B_2) = true^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = true^{\underline{A}}$.

L.H.S

$dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}), eq^{\underline{A}}(true^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), true^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}), \ldots$
$\ldots eq^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}), eq^{\underline{A}}(true^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$

$\bigwedge$
$\bigvee$

$= dc^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, false^{\underline{A}}))$

by applying $dc^{\underline{A}}(true^{\underline{A}}, x_8, y_8) = x_8$ with: x_8 as $false^{\underline{A}}$ and y_8 as $f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$.

$= dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, false^{\underline{A}})$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_8, y_8) = y_8$ with: x_8 as $true^{\underline{A}}$ and y_8 as $dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, false^{\underline{A}})$.

$= false^{\underline{A}}$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_8, y_8) = y_8$ with: x_8 as $false^{\underline{A}}$ and y_8 as $false^{\underline{A}}$.

R.H.S

$not^{\underline{A}}(or^{\underline{A}}(true^{\underline{A}}, \ldots$
$\ldots not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}), true^{\underline{A}}))))$
$= not^{\underline{A}}(true^{\underline{A}})$

by applying $or^{\underline{A}}(true^{\underline{A}}, x_4) = true^{\underline{A}}$ with: x_4 as $not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), true^{\underline{A}}))$.

$= false^{\underline{A}}$

by applying $not^{\underline{A}}(true^{\underline{A}}) = false^{\underline{A}}$.

Calculating for Sub-sub-case 2.1.2. $eval^{\underline{A}}(n_1, B_2) = false^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = true^{\underline{A}}$.

L.H.S

$dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}), eq^{\underline{A}}(true^{\underline{A}}, \ldots$

$\ldots true^{\underline{A}})), true^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}), \ldots$
$\ldots eq^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}), eq^{\underline{A}}(true^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$
$\wedge$
$\vee$

$= dc^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}}, \ldots$
$\ldots f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $and^{\underline{A}}(false^{\underline{A}}, x_4) = false^{\underline{A}}$ with: x_4 as $false^{\underline{A}}$.

$= dc^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $and^{\underline{A}}(false^{\underline{A}}, x_4) = false^{\underline{A}}$ with: x_4 as $true^{\underline{A}}$.

$dc^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$
$= true^{\underline{A}}$

by applying $dc^{\underline{A}}(true^{\underline{A}}, x_8, y_8) = x_8$ with: x_8 as $true^{\underline{A}}$ and y_8 as
$dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, \ldots$
$\ldots false^{\underline{A}}, true^{\underline{A}})))$.

R.H.S

$not^{\underline{A}}(or^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}), true^{\underline{A}}))))$
$= not^{\underline{A}}(or^{\underline{A}}(false^{\underline{A}}, not^{\underline{A}}(true^{\underline{A}})))$

by applying $or^{\underline{A}}(x_4, true^{\underline{A}}) = true^{\underline{A}}$ with: x_4 as $f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$.

$= not^{\underline{A}}(not^{\underline{A}}(true^{\underline{A}}))$

by applying $or^{\underline{A}}(false^{\underline{A}}, x_4) = x_4$ with: x_4 as $not^{\underline{A}}(true^{\underline{A}})$.

$= not^{\underline{A}}(false^{\underline{A}})$

by applying $not^{\underline{A}}(true^{\underline{A}}) = false^{\underline{A}}$.

$= true^{\underline{A}}$

by applying $not^{\underline{A}}(false^{\underline{A}}) = true^{\underline{A}}$.

Calculating for Sub-sub-case 2.1.3. $eval^{\underline{A}}(n_1, B_2) = true^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = false^{\underline{A}}$.
L.H.S
$dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}), eq^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), true^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}), \ldots$
$\ldots eq^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}), eq^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$
$\bigwedge$
$\bigvee$
$= dc^{\underline{A}}(and^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}), true^{\underline{A}}, dc^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}), false^{\underline{A}}, \ldots$
$\ldots f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $and^{\underline{A}}(true^{\underline{A}}, true^{\underline{A}}) = true^{\underline{A}}$.

$= dc^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}), false^{\underline{A}}, \ldots$
$\ldots f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $and^{\underline{A}}(false^{\underline{A}}, x_4) = false^{\underline{A}}$ with: x_4 as $false^{\underline{A}}$.

$= dc^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}, dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}))))$

by applying $and^{\underline{A}}(x_4, false^{\underline{A}}) = false^{\underline{A}}$ with: x_4 as $true^{\underline{A}}$.

$dc^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}, dc^{\underline{A}}(true^{\underline{A}}, false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$
$= dc^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}, false^{\underline{A}})$

by applying $dc^{\underline{A}}(true^{\underline{A}}, x_8, y_8) = x_8$ with: x_8 as $false^{\underline{A}}$ and y_8 as
$dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}}))$.

$= false^{\underline{A}}$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_8, y_8) = y_8$ with: x_8 as $true^{\underline{A}}$ and y_8 as $false^{\underline{A}}$.

R.H.S

$not^{\underline{A}}(or^{\underline{A}}(true^{\underline{A}}, \ldots$
$\ldots not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}}))))$
$= not^{\underline{A}}(true^{\underline{A}})$

by applying $or^{\underline{A}}(true^{\underline{A}}, x_4) = true^{\underline{A}}$ with: x_4 as $not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}}))$.

$= false^{\underline{A}}$

by applying $not^{\underline{A}}(true^{\underline{A}}) = false^{\underline{A}}$.

Calculating for Sub-sub-case 2.1.4. $eval^{\underline{A}}(n_1, B_2) = false^{\underline{A}}$ and $eval^{\underline{A}}(n_1, B_1) = false^{\underline{A}}$.

L.H.S

$dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}), eq^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), true^{\underline{A}}, dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}), \ldots$
$\ldots eq^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}})), false^{\underline{A}}, \ldots$
$\ldots dc^{\underline{A}}(and^{\underline{A}}(eq^{\underline{A}}(false^{\underline{A}}, true^{\underline{A}}), eq^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})), false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}))))$
$\wedge$
$\vee$

$= dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, \ldots$
$\ldots false^{\underline{A}}, true^{\underline{A}}))$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_8, y_8) = y_8$ with: x_8 as $false^{\underline{A}}$ and y_8 as $dc^{\underline{A}}(false^{\underline{A}}, false^{\underline{A}}, f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}}))$.

$= f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$

by applying $dc^{\underline{A}}(false^{\underline{A}}, x_8, y_8) = y_8$ with: x_8 as $false^{\underline{A}}$ and y_8 as $f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$.

R.H.S

$not^{\underline{A}}(or^{\underline{A}}(false^{\underline{A}}, \ldots$
$\ldots not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, \ldots$
$\ldots B_1, false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}}))))$
$= not^{\underline{A}}(not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, \ldots$
$\ldots false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}})))$

by applying $or^{\underline{A}}(false^{\underline{A}}, x_4) = x_4$ with: x_4 as $not^{\underline{A}}(or^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}}), false^{\underline{A}}))$.

$= not^{\underline{A}}(not^{\underline{A}}(f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, \ldots$
$\ldots true^{\underline{A}})))$

by applying $or^{\underline{A}}(x_4, false^{\underline{A}}) = x_4$ with: x_4 as $f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$.

$= f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$

by applying $not^{\underline{A}}(not^{\underline{A}}(x_4)) = x_4$ with: x_4 as $f66c1^{\underline{A}}(mult2^{\underline{A}}(n_1, \ldots$
$\ldots succ^{\underline{A}}(succ^{\underline{A}}(0^{\underline{A}}))), B_2, B_1, B_2, B_1, false^{\underline{A}}, true^{\underline{A}})$.

□

Bibliography

Abelson and Sussman [1985]
H Abelson and G Sussman. *The Structure and Analysis of Computer Programs.* MIT Press, 1985.

Ackerman [1979]
W B Ackerman. Data flow languages. In R E Merwin, editor, *AFIPS National Computer Conference*, volume 48, pages 1087–1095, 1979.

Adams [1969]
D Adams. *A computation model with data flow sequencing.* PhD thesis, Stanford University, December 1969.

Adams [1970]
D Adams. A model for parallel computations. In L C Hobbs *et al*, editor, *Parallel Processor Systems, Technologies, and Applications*, pages 311–333. Spartan, 1970.

Amamiya *et al.* [1984]
M Amamiya, R Hasegawa, and S Ono. VALID: A High Level Functional Language for Dataflow Machines. *Rev ECL*, 32(5):793–802, 1984.

Amblard and Charles [1989]
P Amblard and H Charles. Music Synthesis Description with the Data Flow Language LUSTRE. *Microprocessing and Microprogramming*, 27:551–556, 1989.

Arvind and Gostelow [1978]
Arvind and K P Gostelow. Some Relationships between Asychronous Interpreters of a Dataflow Language. In E J Neuhold, editor, *Formal Description of Programming Concepts*, pages 95–119. North-Holland, 1978.

Arvind *et al.* [1979]
Arvind, K P Gostelow, and W Plouffe. An asynchronous programming language and computing machine. Department of Information and Computer Science Technical Report 114A, University of California, 1979.

Backus [1978]
J Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

Backus [1981]
J Backus. Is computer science based on the wrong fundamental concept of programme? In J W de Bakker and J C van Vilet, editors, *Algorithmic languages*, pages 133-165. Elsevier North-Holland, Amsterdam, 1981.

Bartha [1992a]
M Bartha. An Algebraic Model of Synchronous Systems. *Information and Computation*, 97:97–131, 1992.

Bartha [1992b]
M Bartha. Foundations of a theory of synchronous systems. *Theoretical Computer Science*, 100:325–346, 1992.

Becker and Chambers [1984]
R A Becker and J M Chambers. *S: An Interactive Environment for Data Analysis and Graphics*. Wadsworth Inc, Belmont CA, 1984.

Bellia and Levi [1986]
M Bellia and G Levi. The Relation Between Logic and Functional Languages: A Survey. *Journal of Logic Programming*, 6(3):217–236, October 1986.

Bellia *et al.* [1982]
M Bellia, E Dameri, and M Martelli. Applicative Communicating Processes In First Order Logic. In G Goos and J Hartmanis, editors, *International Symposium on Programming*, volume 137, pages 1–14. Springer-Verlag, 1982.

Bellia *et al.* [1984]
M Bellia, E Dameri, P Degano, G Levi, and M Martelli. A Formal Model For Lazy Implementations of a Prolog-Compatible Functional Language. In J A Campbell, editor, *Implementations of PROLOG*. Artificial Intelligence, pages 309–340. Ellis Harwood, 1984.

Bergstra and Klop [1983]
J Bergstra and J W Klop. Process Algebra for the Operational Semantics of Static Data Flow Networks. Mathematical Centre Technical Report IW 222/83, University of Amsterdam, 1983.

Bergstra and Tucker [1987]
J A Bergstra and J V Tucker. Algebraic Specifications of Computable and Semicomputable Data Types. *Theoretical Computer Science*, 50:137–181, 1987.

Bergstra and Tucker [1992]
J A Bergstra and J V Tucker. Equational Specifications, complete term rewriting systems, and computable and semicomputable algebras. University College of Swansea, Computer Science Division, Technical Report CSR 20–92, University College of Swansea, 1992.

Berry and Cosserat [1984]
G Berry and L Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In S D Brookes A W Roscoe and G Winkel, editors, *Seminar on Concurrency*, number 197 in LNCS, pages 389–448. Springer-Verlag, 1984.

Berry and Gonthier [1988]
G Berry and G Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *The Science of Computer Programming*, 842, 1988.

Berry and Gonthier [1991]
G Berry and G Gonthier. Incremental Development of an HDLC entity in Esterel. *Computer Networks*, 22:35–49, 1991.

Berry *et al.* [1988]
G Berry, P Couronne, and G Gonthier. *Synchronous Programming of Reactive Systems: an Introduction to ESTEREL*, pages 35–56. Elsvier Science Publishers B V (North Holland), 1988.

Berry [1991]
G Berry. A hardware implementation of pure Esterel. In *International Workshop on Formal Methods in VLSI*, Lecture Notes In Computer Science, 1991.

Beveniste and Berry [1991]
A Beveniste and G Berry. The Synchronous approach to reactive and real-time systems. *Proceedings IEEE*, 79:1270–1282, 1991.

Blom *et al.* [1993]
J Blom, A V Holden, M J Poole, J V Tucker, and H Zhang. Caress II: a general purpose tool for parallel deterministic systems, with applications to simulating cellular systems. *Journal of Physiology (London)*, 467:145, 1993. Leeds Meeting 14-15 Jan.

Blom [1992]
J Blom. Tools for the Specification and Analysis of Coupled Map Lattices. Final Year Dissertation, Department of Computer Science, University College of Swansea, 1992.

Boudol *et al.* [1990]
G Boudol, V Roy, R de Simone, and D Vergamini. Process calculi, from theory to practice: verification tools. In *Automatic verification methods for finite state systems*, number 407 in Lecture Notes In Computer Science, pages 1–10. Springer-Verlag, 1990.

Boussinot and de Simone [1991]
F Boussinot and R de Simone. The Esterel language. In *Another look at real time languages*, number 79 in Proceeding of the IEEE, pages 1293–1304, 1991.

Brock and Ackerman [1981]
J D Brock and W B Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, number 107 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

Brock [1987]
J D Brock. Operational Semantics of a Data Flow Language. MIT Lab for Computer Science Technical Memo 120, MIT, 1987.

Broy and Dendorfer [1992]
M Broy and C Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2(1):1–21, January 1992.

Broy and Lengauer [1991]
M Broy and C Lengauer. On Denotational verses Predicative Semantics. *Journal of Computer and System Sciences*, 42(1):1–29, 1991.

Broy *et al.* [1993]
M Broy, F Dederichs, C Dendorfer, M Fuchs, F Gritzer, and W Weber. The Design of Distributed Systems: An Introduction to FOCUS. Technical Report TUM-19292-2, Technische Universität Müuchen, January 1993.

Broy [1983]
M Broy. Applicative real time programming. In *Information Processing 83, IFIP World Congress, Paris*, pages 259–264. North Holland, 1983.

Broy [1986]
M Broy. A theory for nondeterminism, parallelism, communication and concurrency. *Theoretical Computer Science*, pages 1–61, 1986.

Broy [1987a]
M Broy. Predicative specification for functional programs describing communicating networks. *Information Processing Letters*, 25:93–101, 1987.

Broy [1987b]
M Broy. Semantics of finite or infinite networks of communicating agents. *Distributed Computing*, 2:13–31, 1987.

Broy [1988a]
M Broy. An example for the design of a distributed system in a formal setting - the lift problem. Technical Report MIP 8802, Universität Passau, February 1988.

Broy [1988b]
M Broy. Non-deterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

Broy [1989]
M Broy. Towards a Design Methodology for Distributed System. In M Broy, editor, *Constructive Methods In Computer Science*, volume 55 of *NATO ASI Series F: Computer and System Sciences*, pages 311–364. Springer-Verlag, 1989.

Broy [1990]
M Broy. Functional Specification of Time Sensitive Communicating Systems. In G Rozenburg, J W de Bakker, and W P de Roever, editors, *Stepwise Refinement of Distributed Systems*, volume 420 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

Broy [1992a]
M Broy. Compositional refinement of interactive systems. Working Material for the International Summer School on Program Design Calculi, 1992.

Broy [1992b]
M Broy. (Inter-)action refinement. Working material for the International Summer School on Program Design Calculi, 1992.

Burge [1975]
W H Burge. Stream Processing Functions. *IBM Journal of Research and Development*, pages 12–25, 1975.

Burstall *et al.* [1971]
R M Burstall, J S Collins, and R J Popplestone. *Programming in POP-2*. Edinburgh University Press, 1971.

Burstall *et al.* [1980]
R M Burstall, D B MacQueen, and D T Sanella. HOPE: an Experimental Applicative Language. In *Lisp Conference, Stanford*, 1980.

Cadiou [1972]
J M Cadiou. *Recursive definitions of partial functions and their computations*. PhD thesis, Stanford University, 1972.

Cai and Paige [1989]
J Cai and R Paige. Program Derivation by Fixed Point Computation. *Science of Computer Programming*, 11:197–261, 1989.

Caspi *et al.* [1987]
P Caspi, D Pilaud, N Halbwachs, and J A Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems. 14^{th} *ACM Symposium on Principles of Programming Languages, Munich*, pages 178–188, January 1987.

Church [1954]
A Church. Review of Berkeley 1954. *The Journal of Symbolic Logic*, 20:286–287, 1954.

Church [1957a]
A Church. Applications of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the summer institute for symbolic logic, Cornell University, 1957.

Church [1957b]
A Church. Binary recursive arithmetic. *Journal de mathématiques*, 9(36):39–55, 1957.

Clark and Gregory [1981]
K L Clark and S A Gregory. A Relational Language for Parallel Programming. In *ACM Conference on Functional Programming and Computer Architecture*, pages 171–178, 1981.

Clark and Gregory [1983]
K L Clark and S Gregory. PARLOG: A Parallel Logic Programming Language. Research Report 83/5, Imperial College, May 1983.

Clark and Gregory [1985]
K Clark and S Gregory. Notes on the Implementation of PARLOG. *Journal of Logic Programming*, 2(1):17–42, 1985.

Clement and Incerpi [1989]
D Clement and J Incerpi. Programming the behaviour of graphical objects using Esterel. In *Proceedings TAPSOFT*, number 352 in Lecture Notes In Computer Science. Springer-Verlag, 1989.

Codd [1968]
E F Codd. *Cellular Automata.* Academic Press, New York, 1968.

Cohn and Gordon [1990]
A Cohn and M Gordon. A Mechanized Proof of Coorectness of a Simple Counter. In K McEvoy and J V Tucker, editors, *Theoretical Foundations for VLSI Design*, volume 10 of *Tract in Theoretical Computer Science*, pages 65–96. Cambridge University Press, 1990.

Cullyer [1985]
W J Cullyer. Viper Microprocessor: Formal Specification. Technical report, Royal Signal and Radar Establishment, Malvern, 1985.

Curry [1941]
H B Curry. A formalization of recursive arithmetic. *American Journal of Mathematics*, 63:263–282, 1941.

Cutland [1980]
N J Cutland. *Computability : an Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

Dannenberg [1984]
R B Dannenberg. Artic: A Functional Language for Real-Time Control. In *Symposium on Lisp and Functional Programming*, 1984.

Davis *et al.* [1976]
M Davis, Y Matijasevic, and J Robinson. Hilbert's Tenth Problem. Diophantine Equations: Positive Aspects of A Negative Solution. In *Proceeding Symposia in Pure Mathematics*, volume 28, 1976.

de Roever [1978]
W P de Roever. On backtracking and greatest fixed points. In E J Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 621-639. North-Holland, 1978.

Dedekind [1888]
R Dedekind. *Was sind und was sollen die Zahlen.* Reprinted by Dover Publications Incorporated, New York,1963, 1888.

Dederichs [1992]
F Dederichs. Transformation verteiler Systeme: Von applikativen zu prozeduralen Darstellungen. Technical Report SFB 342/17/92, Technische Universität München, 1992.

DeGroot and Lindstrom [1986]
D DeGroot and G Lindstrom. *Logic Programming: Functions, Equations, and Relations.* Prentice Hall, 1986.

Dennis [1974]
J B Dennis. First Version of a Data Flow Procedure Language. In B Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1974.

Derrick *et al.* [1989]
J Derrick, G Lajos, and J V Tucker. Specification and verification of synchronous concurrent algorithms using the Nuprl proof development system. Centre for theoretical computer science report, University of Leeds, 1989.

Dershowitz and Jouannaud [1990]
N Dershowitz and J Jouannaud. Rewrite Systems. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 241–320. North-Holland, 1990.

Dezan *et al.* [1992]
C Dezan, H Le Verge, P Quinton, and Y Saouter. The Alpha du Centaur Experiment. In P Quinton and Y Robert, editors, *Algorithms and Parallel VLSI Architectures II*, pages 325–334. Elsevier Science Publishers, 1992.

Dyber and Sander [1988]
P Dyber and H Sander. A functional programming approach to the specification and verification of concurrent systems. Technical report, Chalmers University of Technology and University of Göteborg, Department of Computer Sciences, 1988.

Dyber [1985]
P Dyber. Program verification in a logical theory of constructions. In J Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 334–349. Springer-Verlag, 1985.

Ehrig and Mahr [1985]
H Ehrig and B Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science 6. Springer-Verlag, Berlin, 1985.

Eker and Tucker [1989]
S M Eker and J V Tucker. Specification and verification of synchronous concurrent algorithms: a case study of the Pixel Planes architecture. In R A Earnshaw P M Dew and T R Heywood, editors, *Parallel processing for computer vision and display*, pages 16–49. Addison Wesley, 1989.

Eker *et al.* [1990]
S M Eker, V Stavridou, and J V Tucker. Verification of synchronous concurrent algorithms using OBJ3. A case study of the Pixel Planes architecture. In *Proceedings of Workshop on Designing Correct Circuits*, Oxford, 1990. Springer-Verlag.

Farah [1977]
M Farah. *Correct Compilation of a Useful Subset of Lucid.* PhD thesis, University of Waterloo, Ontario, Canada, 1977.

Farmer *et al.* [1989]
D Farmer, T Toffoli, and S Wolfram, editors. *A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems*, volume 36 of *Physica D.* North-Holland, 1989.

Faustini [1982]
A Faustini. *The Equivalence of a denotational and an operational semantics for pure dataflow.* PhD thesis, University of Warwick, Computer Science Department, Coventry, United Kingdom, 1982.

Fernández and Jouannaud [1993]
M Fernández and Jean-Pierre Jouannaud. Modularity Properties of Term Rewriting Systems Revisited. Technical report, Laboratoire de Recherche en Informatique, Bât, 490, CNSR et Université de Paris Sud, 91405 Orsay, France, 1993.

France [1992]
R B France. Semantically Extended Data Flow Diagrams - A Formal Specification. *IEEE Transactions on Software Engineering*, 18(4):329–346, 1992.

Friedman and Wise [1976]
D P Friedman and D S Wise. Cons should not evaluate its arguments. In *Proceedings of the 3rd Colloquium on Automata, Languages and Programming*, pages 257–284. Edinburgh Press, 1976.

Goguen and Meseguer [1982]
J A Goguen and J Meseguer. Completeness of many-sorted equational logic. Technical Report 1, SRI, May 1982.

Goguen and Winkler [1988]
J A Goguen and T Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

Goguen *et al.* [1978]
J A Goguen, J W Thatcher, E G Wagner, and J B Wright. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R T Yeh, editor, *Current trends in programming methodology*, pages 80–149. Prentice Hall, London, 1978.

Goguen *et al.* [1992]
J A Goguen, T Winkler, J Meseguer, K Futatsugi, and Jouannaud. *Applications of Algebraic Specification Using OBJ.* Cambridge University Press, 1992.

Goguen [1987]
J A Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. In *2nd Banff Workshop on Hardware Verification*, Banff Canada, June 1987.

Goguen [1988]
J A Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.

Goguen [1990]
J A Goguen. Proving and Rewriting. In *2^{nd} International Conference on Algebraic and Logic Programming*, 1990.

Goodstein [1941]
R L Goodstein. Function theory in an axiom-free equation calculus. *Proceedings of the London Mathematical Society*, 48:401–434, 1941.

Goodstein [1957]
R L Goodstein. *Recursive number theory. A development of recursive arithmetic in a logic-free equation calculus.* North-Holland, 1957.

Goodstein [1961]
R L Goodstein. Studies in Logic and the Foundations of Mathematics. In L E J Rouwer, E W Beth, and A Heyting, editors, *Recursive Analysis.* North-Holland, 1961.

Gorden [1981]
M Gorden. A Very Simple Model of Sequential Behaviour of nMOS. In J Gray, editor, *Proceedings of International VLSI Conference.* Academic Ppess, 1981.

Gordon *et al.* [1979]
M Gordon, R Milner, and C Wadsworth. Edinburgh LCF. In *Semantics of Concurrent Computation*, number 70 in LNCS. Springer-Verlag, 1979.

Gorlatch [1992]
S Gorlatch. Parallel Program Development for a recursive numerical algorithm: a case study. Technical Report SFB 342/7/92, Technische Universität München, March 1992.

Grzegorczyk [1953]
A Grzegorczyk. Some classes of recursive functions. *Rozprawy Math*, 4, 1953.

Guatier *et al.* [1987]
T Guatier, P Le Guernia, and L Besnard. Signal: A Declarative Language For Synchronous Programming Of Real-Time Systems. Technical report, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE, 1987.

Guernic *et al.* [1991]
P Le Guernic, T Gautier, M LeBorgne, and C LeMaire. Programming real time applications with Signal. *Proceedings IEEE*, pages 1321–1336, 1991.

Gurd *et al.* [1981]
J R Gurd, J R W Glauert, and C C Kirkham. Generation of dataflow graphical object code for the Lapse programming language. In W Hdndler, editor, *CONPAR 81*, number 111 in Lecture Notes In Computer Science, pages 155–168. Springer-Verlag, June 1981.

Gutowitz [1990]
Howard Gutowitz, editor. *Cellular Automata: Theory and Experiment*, volume 45 of *Physica D*. North-Holland, 1990.

Halbwachs *et al.* [1991]
N Halbwachs, P Caspi, P Raymond, and D Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings IEEE*, 79:1305–1320, September 1991.

Halbwachs *et al.* [1992]
H Halbwachs, F Lagnier, and C Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992.

Hanna and Daeche [1993]
K Hanna and N Daeche. Strongly-Typed Theory of Structures and Behaviours. In G Milne and L Pierre, editors, *Correct Hardware Design and Verification Methods*, Lecture Notes In Computer Science, pages 39–54. Springer-Verlag, 1993.

Hansson [1993]
C H Hansson. Case studies in the specification and correctness of neural networks. Final year dissertation, University College Swansea, 1993.

Harel and Pnueli [1985]
D Harel and A Pnueli. On the Development of Reactive Systems. Weizmann Institute of Science, Rehovot, Israel, 1985.

Harman and Tucker [1988a]
N A Harman and J V Tucker. In *Formal Specification and the Design of Verifiable Computers*, Proceedings of the 1988 UK IT Conference, pages 500–503, University College of Swansea, 1988.

Harman and Tucker [1988b]
N A Harman and J V Tucker. Clocks, Retimings, and the Formal Specification of a UART. In G J Milne, editor, *The Fusion of Hardware Design and Verification*. North-Holland, 1988.

Harman and Tucker [1988c]
N A Harman and J V Tucker. Formal specifications and the design of verifiable computers. In *Proceedings of 1988 UK IT Conference*, pages 500–503. Institute of Electrical Engineers (IEE), 1988. Held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI).

Harman and Tucker [1990]
N A Harman and J V Tucker. The Formal Specification of a Digital Correlator I: Abstract User Specification. In K McEvoy and J V Tucker, editors, *Theoretical Foundations for VLSI Design*, volume 10 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

Harman and Tucker [1992]
N A Harman and J V Tucker. Consistent Refinements of Specifications for Digital Systems. In P Prinetto and P Camurati, editors, *Correct Hardware Disign Methodologies*, pages 273–295. North-Holland, 1992.

Harman and Tucker [1993]
N A Harman and J V Tucker. Algebraic Models and the Correctness of Microprocessors. Computer Science Division Technical Report CSR 2-93, University College of Swansea, 1993.

Harman [1989]
N A Harman. *Formal Specifications for Digital Systems*. PhD thesis, School of Computer Studies, University of Leeds, 1989.

Harrison [1987]
D Harrison. RUTH: A Functional Language For Real-Time Programming. In G Goos and J Hartmanis, editors, *PARLE Parallel Architectures and Languages Europe*, number 259 in LNCS, pages 297–314. Springer-Verlag, 1987.

Hearn and Meinke [1993]
B M Hearn and K Meinke. ATLAS: A Typed Language for Algebraic Specification. Technical Report 816, University of Wales University College of Swansea, 1993.

Hearn [1994]
B M Hearn. *The Design and Implementation of Typed Languages for Algebraic Specification*. PhD thesis, University College of Swansea Computer Science Department, 1994.

Heering [1986]
J Heering. Partial Evaluation and Co-Completeness of Algebraic Specifications. *Theoretical Computer Science*, 43:149–167, 1986.

Henderson and Morris [1976]
P Henderson and J H Morris. A lazy evaluator. In 3^{rd} *Conference on the Principles of Programming Languages*, pages 95–103. ACM, 1976.

Herath *et al.* [1986]
J Herath, N Saito, K Toda, Y Yamaguchi, and T Yuba. Data-flow computing base language with n-value logic. In *Fall Joint Comp Conf*, 1986.

Hilbert and Bernays [1934]
D Hilbert and P Bernays. *Grundlagen der Methematik*, volume 1. Springer (Berlin), 1934.

Hillis [1985]
W D Hillis. *The Connection Machine.* ACM Distinguished Dissertation Series. MIT Press, 1985.

Hoare [1985]
C A R Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

Hobley *et al.* [1988]
K M Hobley, B C Thompson, and J V Tucker. Specification and verification of synchronous concurrent algorithms: a case study of a convolution algorithm. In G Milne, editor, *The fusion of hardware design and verification*, pages 347–374. North-Holland, 1988. Proceedings of IFIP Working Group 10.2 Working Conference.

Hobley [1990]
K M Hobley. *The Specification and Verification of Synchronous Concurrent Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1990.

Holcombe and Ipate [1994]
M Holcombe and F Ipate. X-machines with stacks and recursive enumerable functions. Technical report, Department of Computer Science, University of Sheffield, England, 1994.

Holcombe [1988]
M Holcombe. X-Machines as a Basis for System Specification. *Software Engineering*, 3(2):69–76, 1988.

Holden *et al.* [1991a]
A V Holden, J V Tucker, and B C Thompson. Can excitable media be considered as computational systems? *Physica D*, 49:240–246. 1991.

Holden *et al.* [1991b]
A V Holden, J V Tucker, and B C Thompson. The computational structure of neural systems. In A V Holden and V I Kryukov, editors, *Neurocomputers and Attention I: Neurobiology, Synchronisation and Chaos*, pages 223–240. Manchester University Press, 1991.

Holden *et al.* [1992a]
A V Holden, J V Tucker, H Zhang, and M J Poole. Coupled Map Lattices as Computational Systems. *Chaos*, 2:367–376, 1992.

Holden *et al.* [1992b]
AV Holden, B C Thompson, J V Tucker, D Withington, and H Zhang. A Theoretical Framework for Synchronization, Coherence and Chaos in Real and Simulated Neural Networks.

In J Taylor, editor, *Workshop on Complex Dynamics in Neural Networks*, pages 223–240. Springer-Verlag, 1992.

Holden *et al.* [1993]
A V Holden, M J Poole, J V Tucker, and H Zhang. Coupling CMLs and the Synchronization of a Multilayer Neural Computing System. *Chaos, Solitons and Fractals*, 1993. To appear.

Hopcroft and Ullman [1979]
J E Hopcroft and J D Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

Johnson and Zhu [1991]
S D Johnson and Z Zhu. An Algebraic Approach to Hardware Specification and Derivation. In L Claeson, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design.* Elsevier, 1991.

Jones and Sinclair [1989]
S B Jones and A F Sinclair. Functional programming and Operating Systems. *The Computer Journal*, 32, 1989.

Jonsson [1988]
B Jonsson. A fully abstract trace model for dataflow networks. Technical Report 88016, Swedish Institute of Computer Science, 1988.

Kahn and MacQueen [1977]
G Kahn and D B MacQueen. Coroutines and networks of parallel processes. IFIP Congress, pages 993–998. Elsevier North-Holland, Amsterdam, 1977.

Kahn [1974]
G Kahn. The semantics of a simple language for parallel processing. *Proceedings IFIP Congress*, pages 471–475, 1974.

Kahn [1984]
G Kahn. A Primitive for the Control of Logic Programs. In *Proceeding of the Symposium on Logic Programing*, pages 242–251, Atlantic City, 1984. IEEE Computer Society.

Kamp and Hasler [1990]
Y Kamp and M Hasler. *Recursive Neural Networks for Associative Memory.* Wiley, 1990.

Kearney and Staples [1991]
P Kearney and J Staples. An Extensional Fixed-Point Semantics For Non-Deterministic Data Flow. *Theoretical Computer Science*, 91(2):129–179, 1991.

Kiliminster [1993]
C Kiliminster. Lecture: History of Computation Colloquium, Oxford University, 1993.

Klint [1993]
P Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

Kloos *et al.* [1986]
C D Kloos, W Dosch, and B Möller. On the Algebraic Specification of a Language for Describing Communicating Agents. In *ÖGI/ÖCG Conference, Passau*, pages 53–73, February 1986.

Kloos [1987a]
C D Kloos. Semantics of Digital Circuits. In G Goos and J Hartmanis, editors, *Semantics of Digital Circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

Kloos [1987b]
C D Kloos. STREAM: A Scheme Language for Formally Describing Digital Circuits. In G Goos and J Hartmanis, editors, *PARLE Parallel Architecture and Languages Europe*, volume 2 of *Lecture Notes in Computer Science*, pages 333–350. Springer-Verlag, 1987.

Klop [1992]
J W Klop. Term Rewriting Systems. In D Gabbay S Abramsky and T S E Maibaum, editors, *Handbook of logic in computer science*. Oxford University Press, 1992.

Knuth and Bendix [1970]
D Knuth and P Bendix. Simple Word Problems in Universal Algebra. In J Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

Kohonen [1972]
T Kohonen. Correlation matrix memories. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 174–180. MIT Press, 1972.

Kohonen [1978]
T Kohonen. *Associative Memory: A System Theoretical Approach*. Springer Verlag, 1978.

Kok [1987a]
J N Kok. A Fully Abstract Semantics for Data Flow Nets. In G Goos and J Hartmanis, editors, *PARLE Parallel Languages and Architectures Europe*, volume 2, pages 351–368. Springer-Verlag, 1987.

Kok [1987b]
J N Kok. A Fully Abstract Semantics for Data Flow Nets. In G Goos and J Hartmanis, editors, *PARLE Parallel Languages and Architectures Europe*, volume 2, pages 351–368. Springer-Verlag, 1987.

Kosiniski [1973]
P R Kosiniski. A data flow programming language for operating systems. *Proceedings ACM, Sigplan-Sigops Interface Meeting, Sigplan Notices*, 8(9):89–94, September 1973.

Kowalski [1974]
R A Kowalski. *Predicate Logic as a Programming Language*, pages 569–574. North-Holland, 1974.

Kozmidiadi and Marchenkov [1969]
V A Kozmidiadi and S S Marchenkov. Multiheaded automata. *Problems of Cybernetics, Moscow*, 21:129–160, 1969. In Russian.

Kung [1982]
H T Kung. Why systolic architectures. *Computer*, pages 37–46, January 1982.

Kurihara and Ohuchi [1992a]
M Kurihara and A Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theoretical Computer Science*, 103:273–282, 1992.

Kurihara and Ohuchi [1992b]
M Kurihara and A Ohuchi. Non-Copying term rewriting and modularity of termination. Technical report, Hokkaido University, 1992.

Landin [1965a]
P J Landin. The Correspondence Between ALGOL 60 and Church's Lambda Calculus: Part 2. *Communications of the ACM*, 8:158–165, 1965.

Landin [1965b]
P J Landin. The Correspondence Between ALGOL 60 and Church's Lambda Notation: Part 1. *Communications of the ACM*, 8:89–101, 1965.

Landin [1966]
P J Landin. The next 700 programming languages. *Communications of the Association for Computing Machinery*, 9:157–166, 1966.

Lee and Tan [1992]
P T Lee and K P Tan. Modeling of Visualized Data-Flow Diagrams Using Petri Net Model. *Software Engineering*, 7(1):4–12, 1992.

Leiserson and Saxe [1983]
C E Leiserson and J B Saxe. Optimizing Synchronous Systems. *VLSI Computing Systems*, 1:41–67, 1983.

Levi and Palamidessi [1988]
G Levi and C Palamidessi. Contributions to the Semantics of Logic Perpetual Processes. *Acta Informatica*, 25:691–711, 1988.

Levi and Pegna [1983]
G Levi and A Pegna. Top-Down Mathematical Semantics and Symbolic Execution. *RAIRO Inform. Théor*, 17:55–70, 1983.

Li and Martin [1986]
P-Y P Li and A J Martin. The Sync Model: A parallel Execution Method for Logic Programming. In *Symposium on Logic Programming*, pages 223–234, Salt Lake City, 1986. IEEE Computer Society.

Lindstrom and Panangaden [1984]
G Lindstrom and P Panangaden. Stream-Based Execution of Logic Programe. In *Proceedings of the Symposium on Logic Programming*, pages 168–176, Atlantic City, 1984. IEEE Computer Society.

Lloyd [1984]
J W Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

MacQueen and Sanella [1985]
D B MacQueen and D T Sanella. Completeness of Proof Systems for Equational Specifications. *IEEE Transactions On Software Engineering*, 11(5):454–461, May 1985.

Manna *et al.* [1973]
Z Manna, S Ness, and J Vuillemin. Inductive methods for proving properties of programs. *Communications of the Association for Computing Machinery*, 16(8):491–502, August 1973.

Marchenkov [1969]
S S Marchenkov. Elimination of Recursion Schemas In the Gregorczyk E^2 Class. *Matematicheskie Zametki*, 5(5):561–568, May 1969.

Marshall [1991]
D Marshall. CADISP: Cellular Automata Design Implementation and Specification. Final Year Dissertation, Department of Computer Science, University College of Swansea, 1991.

Martin and Tucker [1988]
A R Martin and J V Tucker. The concurrent assignment representation of synchronous systems. *Parallel Computing*, 9:227–256, 1988.

Martin [1989]
A R Martin. *The Specification and Simulation of Synchronous Concurrent Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1989.

McConnell and Tucker [1993]
B McConnell and J V Tucker. Infinite Synchronous Concurrent Algorithms: The Algebraic Specification and Verification of a Hardware Stack. In F L Bauer, W Brauer, and H Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 321–375. Springer-Verlag, 1993.

McConnell [1993]
B McConnell. *Infinite Synchronous Concurrent Algorithms.* PhD thesis, Computer Science Department, University College of Swansea, 1993.

McCulloch and Pitts [1943]
W S McCulloch and W Pitts. A logical calculus of the ideas immanent in nervous activity. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 18–27. MIT Press, 1943.

McEvoy and Tucker [1990]
K McEvoy and J V Tucker. Theoretical foundations of hardware design. In K McEvoy and J V Tucker, editors, *Theoretical foundations of VLSI design*. Cambridge University Press, 1990.

McGraw *et al.* [1985]
J McGraw, S Skedzielewski, S Allan, R Oldhoeft, J Glauert, C C Kirkham, B Noyce, and R Thomas. SISAL: Streams and Iteration in a Single Assignement Language. Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory, California, March 1985.

McIlroy [1968]
M C McIlroy. 'Coroutines'. Internal report, Bell Telephone Laboratories, Murray Hill, New Jersey, 1968.

Mead and Conway [1980]
C Mead and J Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

Meinke and Steggles [1992]
K Meinke and L J Steggles. Specification and Verification in Higher Order Algebra: A Case Study of Convolution. Technical Report CSR 16-92, University of Wales University College of Swansea, 1992.

Meinke and Tucker [1988]
K Meinke and J V Tucker. Specification and representation of synchronous concurrent algorithms. In F H Vogt, editor, *Concurrency '88*, number 335 in Lecture Notes in Computer Science, pages 163–180. Springer-Verlag, 1988.

Meinke and Tucker [1992]
K Meinke and J V Tucker. Universal Algebra. In S Abramsky, D M Gabbay, and T S E Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 189–411. Oxford University Press, 1992.

Meinke [1988]
K Meinke. *A Graph Theoretic Model of Synchronous Concurrent Algorithms*. PhD thesis, School of Computer Studies, University of Leeds, 1988.

Meinke [1990]
K Meinke. Universal Algebra in higher types. Technical Report CSR 12-90, University College of Wales University College of Swansea, 1990.

Meinke [1991a]
K Meinke. A Recursive Second Order Initial Algebra Specification of Primitive Recursion. Department of Computer Science Report CSR 8-91, University College of Swansea, 1991.

Meinke [1991b]
K Meinke. Equational Specification of Abstract Types and Combinators. In G Jäger, editor, *Computer Science Logic '91*, Lecture Notes in Computer Science, 626. Springer Verlag, Berlin, 1991.

Meinke [1992a]
K Meinke. Algebraic semantics of rewriting terms and types. In M Rusinowitch and J-L Remy, editors, *Third International Conference on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science, 656. Springer Verlag, Berlin, 1992.

Meinke [1992b]
K Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.

Melham [1988]
T F Melham. Abstraction Mechanisms for Hardware Verification. In G Birtwhistle and P Subrahmanyan, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. ????, 1988.

Mendelson [1987]
E Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole, third edition, 1987.

Meseguer and Goguen [1985]
J Meseguer and J A Goguen. Initiality, induction and computability. In M Nivat, editor, *Algebraic Methods in Semantics*, chapter 14, pages 459–541. Cambridge University Press, 1985.

Middeldorp and Toyama [1991]
A Middeldorp and Y Toyama. Completeness of combinations of constructor systems. In *Proceedings 4^{th} Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1991.

Milne [1989]
G K Milne. Timing Constraints: Formalizing their Description and Verification. In *Proceedings of Computer Hardware Description Languages and their Applications.* North-Holland, 1989.

Milner [1973]
R Milner. Model of LCF. Technical report, Computer Science Department, Stanford University, 1973.

Milner [1983]
R Milner. A Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

Milner [1984]
R Milner. A Proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, Austin, Texas, 1984.

Milner [1989]
R Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

Minsky and Papert [1969]
M Minsky and S Papert. *Perceptrons.* MIT Press, 1969.

Misunas [1975]
D P Misunas. Deadlock Avoidance in Data Flow Architecture. In *Proc. Symp. Automat. Computation and Contr.*, pages 337–343, April 1975.

Murakami and Sethi [1990]
G Murakami and R Sethi. Terminal call processing in Esterel. Technical Report 150, AT and T Bell Laboratories, 1990.

Naish [1985]
L Naish. All Solutions Predicates in Prolog. In *Proceedings of the Symposium on Logic Programming*, pages 73–77, Boston, 1985. IEEE Computer Society.

Nueckel [1988]
H Nueckel. Eine Zeigerimplementierung von Graphreduktion für eine Datenflußprache. Diploma thesis, Universität Passau, 1988.

Parker [1990]
D S Parker. *Stream Data Ananlysis in Prolog*, chapter 8, pages 249–312. The MIT Press, 1990.

Péter [1950]
R Péter. *Recursive Functions.* Unknown, 1950.

Plotkin [1981]
G D Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Århus University, Århus, Denmark, September 1981.

Plotkin [1983]
G D Plotkin. Domains. Department of Computer Science, University of Edinburgh, 1983.

Pnueli [1986]
A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J W de Bakker, W P de Roever, and G Rozenberg, editors, *Current Trends in Concurrency*, number 224 in Lecture Notes in computer Science, pages 510–584. Springer-Verlag, 1986.

Poole [1994]
M J Poole. *Synchronous Concurrect Algorithms and Dynamical Systems.* PhD thesis, Computer Science Department, University College of Swansea, October 1994.

Rabinovich [1993]
A Rabinovich. On the Schematological Equivalence of Dataflow Networks. In *Computer Science and Logic*, 1993.

Ratel *et al.* [1991]
C Ratel, N Halbwachs, and P Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. In *Conference Software for Critical Systems*, ACM-SIGSOFT, 1991.

Röddling [1964]
D Röddling. Über die Eliminierbarkeit von Definitionsschemata in der Theorie der rekursiven Funktionen. *Z. Math. Logik und Grundlag. Math.*, 10(4):315–330, 1964.

Rose [1961]
H E Rose. On the consistency and undecidability of recursive arithmetic. *Zeitschrift für mathematische logik und grundlagen der mathematik*, 7:124–135, 1961.

Rose [1962]
H E Rose. Tenary recursive arithmetic. *Mathematica Scandinavica*, 10:210–216, 1962.

Rumelhart and McClelland [1986a]
D E Rumelhart and J L McClelland, editors. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models.* MIT Press, 1986.

Rumelhart and McClelland [1986b]
D E Rumelhart and J L McClelland, editors. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 1: Foundations.* MIT Press, 1986.

Rumelhart *et al.* [1986]
D E Rumelhart, G E Hinton, and R J Williams. Learning internal representation by error propagation. In D E Rumelhart and J L McClelland, editors, *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 1: Foundations*, pages 318–362. MIT Press, 1986.

Sargeant [1982]
J Sargeant. Implementation of Structured Lucid on a Dataflow Computer. Master's thesis, University of Manchester, 1982.

Scott [1982]
D A Scott. *Domains for Denotational Semantics*, volume 140 of *Lecture Notes In Computer Science*. Springer-Verlag, 1982.

Shapiro [1983]
E Y Shapiro. A Subset of Concurrent Prolog and it Interpretor. Technical Report TR-003, ICOT, 1983.

Sharp [1991]
J A Sharp, editor. *Data Flow Computing.* Ablex Publishing Corporation, 1991.

Sheeran [1983]
M Sheeran. *μFP, an Algebraic VLSI Design Language.* D. phil., St. Cross College, November 1983.

Sheeran [1986]
M Sheeran. RUBY - a Language of Relations and Higher Order Functions. Technical report, Glasgow University, 1986.

Silbermann and Jayaraman [1992]
F S K Silbermann and B Jayaraman. A domain-theoretic approach to functional and logic programming. *Journal of Functional Programming*, 2(3):273–321, July 1992.

Simmons [1988]
H Simmons. The Realm of Primitive Recursion. *Archive Math. Logic*, 27:117–188, 1988.

Skolem [1923]
T Skolem. The foundations of elementary arithmetic by means of a recursive mode of thought, without the use of apparent variables ranging over infinite domains. In Van Hejnort, editor, *From Frege to Gödel.* North-Holland, 1923.

Sleep [1994]
R Sleep. *Term Graph Rewriting.* John Wiley and Sons Limited, 1994.

Staples and Nguyen [1985]
J Staples and V L Nguyen. A Fixed-point Semantics For Nondeterministic Data Flow. *Journal For The Association For Computing Machinery*, 32(2):411–444, 1985.

Stavridou [1993]
V Stavridou. *Formal Verification of Digital Systems.* Cambridge University Press, 1993.

Stefanescu [1987a]
G Stefanescu. On Flowchart Theories: Part I. The Deterministic Case. *Journal of Computer and System Sciences*, 35:163–191, 1987.

Stefanescu [1987b]
G Stefanescu. On Flowchart Theories: Part II. The Nondeterministic Case. *Theoretical Computer Science*, 52:307–340, 1987.

Stephens and Thompson [1992]
R Stephens and B C Thompson. Cartesian Stream Transformer Composition. University

College of Swansea, Department of Computer Science Report CSR 21-92, University College of Swansea, 1992. To appear in *Fundamenta Informaticae.*

Stephens [1991]
R Stephens. Languages and Tools for the Analysis of Simultaneous Primitive Recursive Functions with Applications to the Formal Verification of Synchronous Concurrent Algorithms. Final year dissertation, University College of Swansea, 1991.

Stoltenberg-Hansen *et al.* [1994]
V Stoltenberg-Hansen, E Griffor, and I Lindeström. *Mathematical Theory of Domains.* Cambridge Tracts in Theoretical Computer Science, 1994.

Stoy [1977]
J E Stoy. *Denotational Semantics: The Scott-Strachey Approach To Programming.* MIT Press, 1977.

Subrahmanyam and You [1984]
P A Subrahmanyam and J H You. FUNLOG = Functions + Logic: A Computational Model Integrating Functional and Logic Programming. In *International Symposium on Logic Programming*, pages 144–153. IEEE Computer Soc. Press, 1984.

Summerfield [1994]
M N Summerfield. ASTRAL: An Algebraic Stream Transformer Language. Technical report, Computer Science Department, University of Wales, Swansea, 1994.

Sun [1988]
Sun Microsystems Incorporated. *YACC*, 1988.

Sussman and Steele [1980]
G J Sussman and G L Steele. CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions. *Artificial Intelligence*, 14:1–39, 1980.

Thompson and Tucker [1985]
B C Thompson and J V Tucker. Theoretical Considerations in Algorithm Design. In R.A.Earnshaw, editor, *NATO ASI Fundamental Algorithms for Computer Graphics.* Springer-Verlag, 1985.

Thompson and Tucker [1991]
B C Thompson and J V Tucker. Equational specification of synchronous concurrent algorithms and architectures. Computer Science Division, Technical Report CSR 9.91, University College of Swansea, 1991.

Thompson and Tucker [1994]
B C Thompson and J V Tucker. Equational specifications of synchonous concurrent algorithms and architectures (second edition). Technical Report CSR 9.91, Computer Science Department, University of Wales, Swansea, 1994.

Thompson *et al.* [1992]
B C Thompson, J V Tucker, and W B Yates. Artificial Neural Networks as Synchronous Concurrent Algorithms: Algebraic Specification of a FeedForward Backpropogation Network. Technical report, University College of Swansea Computer Science Division Research Report, 1992.

Thompson [1987]
B C Thompson. *A Mathematical Theory of Synchronous Concurrent Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1987.

Tofts [1993]
C M N Tofts. The Relationship Between Synchronous Concurrent Algorithms and SCCS. Technical report, Department of Computer Science, University College of Swansea, 1993.

Toyama *et al.* [1989]
Y Toyama, J W Klop, and H P Barendregt. Termination for the direct sum of left-linear term rewriting systems. In *Proceedings 3rd Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer Verlag, 1989.

Toyama [1987a]
Y Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141–143, April 1987.

Toyama [1987b]
Y Toyama. On the Church-Rosser property for the direct sum of term rewriting sustems. *Journal of the Association for Computing Machinery*, 34(1):128–143, April 1987.

Tucker and Zucker [1988]
J V Tucker and J I Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics.* North Holland, 1988.

Tucker and Zucker [1992]
J V Tucker and J I Zucker. Theory of Computation over Stream Algebras and its Applications. In I M Havel and V Koubek, editors, *Mathematical Foundations of Computer Science 1992: 17^{th} International Symposium, Prague*, LNCS, pages 62–80. Springer-Verlag, 1992.

Tucker and Zucker [1994]
J V Tucker and J I Zucker. Computable Functions on Stream Algebras. In H Schwichtenburg, editor, *International Summer School on Proof and Computation, Marktoberdorf, 1993*, NATO Advanced Study Institute, pages 341–382. Springer-Verlag, 1994.

Turner [1985]
D A Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Language and Computer Architetcures*, Lecture Notes in Computer Science, 201. Springer-Verlag, 1985.

van Emden and de Lucena Filho [1982]
M H van Emden and G T de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K L Clark and S A Tarnlund, editors, *Logic Programming*, pages 189–198. Academic Press, 1982.

van Emden and Kowalski [1976]
M H van Emden and R A Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, 1976.

Verge *et al.* [1991]
H Le Verge, C Mauras, and P Quinton. The ALPHA Language and its Use for the Design of Systolic Arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

von Neumann [1966]
J von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966.

Vuillemin [1973]
J Vuillemin. *Proof techniques for recursive programs.* PhD thesis, Stanford University, 1973.

Wadge and Ashcroft [1985]
W W Wadge and E A Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

Wadge [1981]
W W Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

Wadge [1984]
W Wadge. Viscid, a vi-like Screen Editor written in pLucid. Technical Report DCS-40-IR, University of Victoria, Computer Science Department, 1984.

Weijland [1990]
W P Weijland. Verification of a Systolic Algorithm in Process Algebra. In K McEvoy and J V Tucker, editors, *Theoretical Foundations of VLSI Design*, volume 10 of *Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.

Welch [1983]
P H Welch. Parallel Assignment Revisited. *Software - Practice and Experience*, 13:1175–1180, 1983.

Weng [1975]
K S Weng. Stream orientated computation in recursive data flow schemas. Project MAC Technical Memo 69, MIT, 1975.

Widrow and Hoff [1960]
B Widrow and M E Hoff. Adaptive switching circuits. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 126–134. MIT Press, 1960.