University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Linear Types for Communicating Processes

Simon John Gay

A thesis submitted for the degree of
Doctor of Philosophy of the University of London
and for the Diploma of the Imperial College.

January 1995

To my family

Abstract

The use of types is well established in sequential computation, with a range of benefits which are especially clear in the functional programming paradigm.

- Types are specifications of simple correctness properties, which can be automatically verified by a compiler.
- The Curry-Howard isomorphism provides elegant connections with intuitionistic logic.
- The categorical semantics of a language can be naturally structured by its type system.

Abramsky's theory of interaction categories provides a framework for the transfer of these benefits to concurrency. This thesis is a study of the foundations and applications of that theory.

- The fundamental examples of interaction categories are defined: the category $\mathcal{SP}roc$ of synchronous processes and the category $\mathcal{ASP}roc$ of asynchronous processes. An abstract axiomatisation of interaction categories is proposed, based on the essential features of these examples.
- The use of $\mathcal{SP}roc$ as a semantic category of processes is illustrated by defining a semantics of dataflow. This is shown to agree with the classical Kahn semantics, and is applied to the language LUSTRE.
- New categories are defined in which the types are powerful enough to guarantee deadlock-freedom of processes satisfying them, so that type-checking methods can be used to support compositional reasoning about deadlock-freedom. This idea is applied to both synchronous and asynchronous examples.
- A calculus of synchronous processes is defined, with a type system based on the structure of interaction categories; this corresponds to classical linear logic under the Curry-Howard isomorphism. The calculus has a denotational semantics in any category with suitable structure, which is specified abstractly, and an operational semantics which matches the categorical semantics when $\mathcal{SP}roc$ is taken as the semantic category.

Acknowledgements

First of all, I would like to thank my supervisor, Samson Abramsky. He has provided a great many suggestions for research topics, and without the foundation of his work on interaction categories my thesis would have been very different. Quite apart from all the technical help I have received, and the great deal I have learnt from him during my Ph.D. studies, he has made it possible for me to attend more conferences and workshops than I had any right to expect. He also encouraged me to collaborate with Rajagopal Nagarajan; this collaboration has produced a lot of material for both of our theses. I will take this opportunity to thank Raja for making the Ph.D. experience less lonely than it might otherwise have been. Among the rest of my colleagues at Imperial, two deserve special mention: Roy Crole, who has helped and encouraged me in various ways during the last three years, and Ian Mackie, with whom it has been a pleasure to share the experience of progressing from the first steps of research to the final writing-up. Radha Jagadeesan, whose time at Imperial coincided with my first year of research, was always enthusiastic, encouraging and inspiring. Mark Dawson maintains the computer system which made writing this thesis possible, and has always been helpful with systems-related problems. Paul Taylor provides a LaTeX system superior to that which we could expect without him; also, I have used his commutative diagram and proof tree macro packages while writing this thesis. Samson Abramsky, Roy Crole, Ian Mackie, Guy McCusker, Chris Hankin, Lindsay Errington and Raja Nagarajan have all made valuable comments on drafts of this thesis; I have also enjoyed discussions with Greg Meredith on various related topics.

Outside Imperial, I would like to thank the Esprit Basic Research Action 6454 (CONFER) for providing funding to attend its workshops, and the participants in those workshops for the opportunity to air my ideas in a friendly atmosphere. The same applies to the participants in the joint Imperial-Cambridge-Oxford Games Seminars. I would also like to thank K. V. S. Prasad for making it possible for me to visit Chalmers University, Carolyn Brown for inviting me to give a seminar at Sussex, and Martin Hyland for spending so much time with me and Raja when we visited Cambridge. The UK Engineering and Physical Sciences Research Council funded my Ph.D. through a studentship, and I have received additional funding from BNR Europe Ltd. (formerly STC Technology Ltd.). The final month's work on my thesis was carried out while I was employed by Esprit Basic Research Action 9102 (Coordination). I thank all of these organisations for their financial support.

Finally, I would like to thank my family for their support at every stage of my edu-

cation, and Sheila Dickinson, whose love and encouragement have helped enormously during the past few months.

Statement

This thesis presents a substantial amount of Samson Abramsky's work as essential background material. In Chapter 3, Section 3.2 describes his material, and Section 3.4 contains my novel formulation of some of his material. In Chapter 5, Sections 5.3, 5.4.1 and 5.4.2 all describe Abramsky's work.

The rest of the research reported in the thesis was carried out either solely by myself or as a collaboration between myself and Rajagopal Nagarajan. In Chapter 3, Section 3.3 describes joint work and Section 3.6 describes my own work. In Chapter 4, Sections 4.3 and 4.4 describe joint work and Sections 4.5 to 4.8 describe my own work. In Chapter 5, Sections 5.2 and 5.5 describe joint work, and Sections 5.4.3 and 5.4.4 describe my own work; all of Chapter 6 describes joint work.

Throughout the thesis, the presentation of all material is entirely my own.

Contents

List of Figures

1

Introduction

The best way to introduce the content of this thesis is by explaining its title: "Linear Types for Communicating Processes". "Communicating Processes" means concurrent systems, constructed from independent computing agents which interact with each other, collaborating towards the solution of a computational problem. Many such agents may reside on a single machine, or they may be physically distributed—the key point is the conceptual identification of separate subsystems, which is the organisational step needed as a preliminary to the understanding of parallel computers.

The next word, which begins to narrow down the focus of the thesis, is "Types". The use of types in programming languages has been very successful, particularly in the case of functional programming where there is a well-established and elegant body of type theory. Some of this theory is described in Section 1.1. However, in the field of concurrency the situation regarding types is much less satisfactory. Although several formalisms for concurrent processes exist, many are untyped; when types are included, they usually impose only very weak constraints on processes. In order to emphasise this point, some of the existing approaches to concurrent types are reviewed at the end of this chapter. Two consequences of this deficiency in the theory of concurrency are of particular interest as far as this thesis is concerned. The first is that the connections between types, syntax and semantics which are described for sequential computation in Section 1.1 exist only in a rudimentary form for concurrent computation. The second is that if types incorporated specifications of interesting program properties, then the techniques of automatic type checking which have proved so useful in sequential programming could be applied to the verification of concurrent systems. The possibility of remedying this situation has recently arisen with the appearance of Abramsky's "interaction categories" approach to semantics and types for concurrency. Indeed, this thesis is essentially an exploration of the theory and application of interaction categories.

The interaction categories framework is the culmination of several years' work by Abramsky towards a satisfactory theory of types for concurrency. Its history is traced in Section 1.3. One of its notable features is the use of linear logic as a basis for concurrent type systems; linear logic is briefly described in Section 1.2. This explains the appearance of the word "Linear" in the title of the thesis.

1.1 Typed λ-calculus and the Curry-Howard Isomorphism

There is an elegant connection between typed functional programming and intuitionistic logic, which goes variously by the names "Curry-Howard Isomorphism", "Propositions as Types Paradigm" and "Proofs as Programs Correspondence". Consider the term formation rules for the simply typed λ-calculus, which is the core of typed functional programming languages.

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha} \qquad \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta} \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

If the terms are removed, leaving just the types, the result is precisely a sequent presentation of the natural deduction rules for intuitionistic logic.

$$\frac{}{\Gamma, \alpha \vdash \alpha} \qquad \frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \qquad \frac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$$

A corollary of this observation is that intuitionistic proofs have computational content. Given a proof, the term formation rules can be used to build up a λ-expression whose type is the proposition which was proved. Conversely, a typed λ-expression viewed as a functional program has correctness properties (such as always sending arguments of the right types to functions) by virtue of the fact that it is a proof of something. Much more discussion of this area can be found in [GLT89].

Semantics can also be treated within the same framework. A deduction system such as the logic presented above can be used to construct a category in which the objects are propositions and a morphism is a proof of the proposition corresponding to its target, from the hypotheses corresponding to its source. The Propositions as Types view then yields a construction of a category whose objects are types and whose morphisms are (essentially) typed λ-terms. The type constructors present in the λ-calculus (corresponding to logical connectives) determine the properties of the associated category—for example, λ-calculus with pairing gives rise to a cartesian closed category. Any concrete category with the necessary structure can then be used to give a semantics to the λ-calculus being considered, and the semantic definitions are naturally structured according to the inductive definition of the types being used. Thus the abstract category serves as a specification for concrete mathematical models of the language. Such correspondences can be developed for a variety of different theories; a good introduction to this area is [Cro94].

The aim of this thesis is to extend as much of this theory as possible to concurrency. This means not only developing a notion of type for processes, but doing it within the framework of the Curry-Howard isomorphism. This offers the prospect of a logical view of concurrent types, whether they state simple facts analogous to the property of being

an integer or capture more complex behavioural specifications. The categorical view goes hand-in-hand with the logical one, and in the context of concurrency it should give guidance as to how semantic domains of processes ought to be structured. A Curry-Howard isomorphism for concurrency represents a unification, to some extent, with the theory of functional programming (subject to resolving differences between the logics and categorical structures involved) and this may shed light on how functional and concurrent languages can best be integrated. Finally, such a theory will include a calculus of typed processes, and it is to be hoped that the design of this calculus might expose some of the fundamental primitives of typed concurrent programming.

1.2 Linear Logic

The origin of linear logic can be traced to work of Lambek on the mathematical analysis of grammatical structures in natural language [Lam58], but it was only with its rediscovery by Girard [Gir87] that it came to the widespread attention of computer scientists. The fundamental novelty of linear logic is the absence of the structural rules of contraction and weakening. These rules allow the hypotheses of a proof to be used more than once (contraction) or not at all (weakening), so in their absence every hypothesis must be used exactly once. This considerably reduces the number of theorems which can be proved, so the structural rules are reintroduced in a restricted form by means of a modality. A formula of the form $!A$ can be weakened or contracted at will, but there are conditions regulating how such formulae can be introduced.

The restrictions on copying and discarding of hypotheses make for a natural view of formulae as resources, and this is one reason why linear logic (in its intuitionistic form) has been widely applied to computation. The basic idea is that by using linear formulae as types, more information about how a program uses its inputs can be captured than is possible with intuitionistic types. One consequence of linearity is that some of the intuitionistic connectives split into two versions: linear logic has two conjunctions ($\otimes$ and $\&$) and two disjunctions ($\parr$ and $\oplus$). The conjunctions have computational interpretations as type constructors corresponding to strict ($\otimes$) and lazy ($\&$) pairs; the lazy pair can be used as a conditional, in which case $\oplus$ is the natural type of a case-selector. Girard and Lafont [Laf88, GL87] have applied these ideas to lazy functional programming and issues of garbage collection. Abramsky [Abr93a] has defined a term calculus corresponding under the Curry-Howard Isomorphism to intuitionistic linear logic, and Mackie [Mac94] has implemented a functional programming language based on this calculus.

The fourth connective, $\parr$, does not generally appear in intuitionistic linear logic, although Hyland and de Paiva [HdP93] have studied an intuitionistic system which does

include it. It is in classical linear logic that $\parr$ finds its place, as the de Morgan dual of $\otimes$ under the linear negation $(-)^\perp$. From the beginning, this version of linear logic has been advertised as having some connection with concurrency. This is for two reasons: firstly, that negation in a classical logic can be interpreted as modulating the distinction between input and output, and secondly, that classical *linear* logic (in contrast to classical logic) is constructive and so it is reasonable to attempt to extract computational content from proofs along the lines of the Curry-Howard Isomorphism. Early work on connecting classical linear logic with computation included Abramsky's proof expressions [Abr93a], with a natural execution model involving multiple processors working on a pool of computational goals, and Lafont's interaction nets [Laf90], in which programming means defining connectives in a generalised logic and building proofs from them.

Figure 1.1 shows the rules of deduction for classical linear logic, in sequent form. Sequents are one-sided, of the form $\vdash A_1, \ldots, A_n$. It is also possible to present the deduction rules in terms of two-sided sequents of the form $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$, but it then turns out that such a sequent is equivalent to $\vdash A_1^\perp, \ldots, A_m^\perp, B_1, \ldots, B_n$. Thus there is no loss of generality in working with one-sided sequents; and in fact, there are then fewer rules, so the system becomes simpler.

This style of presentation assumes that the operation of linear negation is given for atomic formulae, with $A^{\perp\perp} = A$ for all A, and defined for compound formulae by de Morgan duality:

$$\begin{array}{rcl@{\qquad}rcl}
(A \otimes B)^\perp & \stackrel{\text{def}}{=} & A^\perp \parr B^\perp & I^\perp & \stackrel{\text{def}}{=} & \bot \\
(A \parr B)^\perp & \stackrel{\text{def}}{=} & A^\perp \otimes B^\perp & \bot^\perp & \stackrel{\text{def}}{=} & I \\
(A \mathbin{\&} B)^\perp & \stackrel{\text{def}}{=} & A^\perp \oplus B^\perp & \top^\perp & \stackrel{\text{def}}{=} & \mathbf{0} \\
(A \oplus B)^\perp & \stackrel{\text{def}}{=} & A^\perp \mathbin{\&} B^\perp & \mathbf{0}^\perp & \stackrel{\text{def}}{=} & \top \\
(\,!\,A)^\perp & \stackrel{\text{def}}{=} & ?\,(A^\perp) & (\,?\,A)^\perp & \stackrel{\text{def}}{=} & !\,(A^\perp)
\end{array}$$

The Exchange rule, which is the only one of the classical structural rules to be retained, allows the formulae in a sequent to be permuted arbitrarily. Omitting even this structural rule would lead to non-commutative linear logic, in which the connective $\otimes$ is not commutative. Several non-commutative versions of linear logic have been studied, for example in [Yet90] and [BG91], with various restrictions on the use of the Exchange rule, but for this thesis the commutative case is sufficient.

The Axiom allows formulae to be introduced in dual pairs, and the Cut rule allows a dual pair of formulae to be eliminated. The Tensor rule introduces the $\otimes$ connective. In both the Tensor and Cut rules, the lists of formulae Γ and Δ in the hypotheses are strictly preserved; if some formula C occurs in both hypotheses, then there are two occurrences of C in the conclusion. Rules which maintain contexts in this way are

Structural Rule

$$\frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}\ \text{Exchange}$$

Identity Rules

$$\frac{}{\vdash A^{\perp}, A}\ \text{Axiom} \qquad \frac{\vdash \Gamma, A \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ \text{Cut}$$

Multiplicative Rules

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ \text{Tensor} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \text{Par}$$

Additive Rules

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\ \text{With} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \text{L Plus} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \text{R Plus}$$

Unit Rules

$$\frac{}{\vdash I}\ \text{Unit} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \text{Bottom} \qquad \frac{}{\vdash \top, \Gamma}\ \text{Top}$$

Exponential Rules

$$\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ \text{Contr} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ \text{Weak}$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ \text{Der} \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\ \text{Prom}$$

Figure 1.1: Sequent rules for Classical Linear Logic.

known as *multiplicative*. The Par rule introduces the connective $\parr$, dual to $\otimes$. With only one hypothesis, the multiplicative nature of this rule is not apparent, but $\parr$ is also classed as a multiplicative connective.

The additive rules have a different effect on contexts; again, this is only apparent from the With rule which has two hypotheses. The connective $\&$ behaves as a conditional; a proof of $\vdash \Gamma, A \,\&\, B$ offers a choice between a proof of $\vdash \Gamma, A$ and a proof of $\vdash \Gamma, B$. The dual connective $\oplus$ allows a choice to be made, depending on which of the two Plus rules is used to introduce it. Such choices are resolved when the cut-elimination procedure is applied to a proof. A cut of the form

$$\dfrac{\dfrac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\,\text{With} \qquad \dfrac{\Delta, A^{\perp}}{\Delta, A^{\perp} \oplus B^{\perp}}\,\text{L Plus}}{\vdash \Gamma, \Delta}\,\text{Cut}$$

is replaced by the simpler cut

$$\dfrac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}\,\text{Cut}$$

in which the proof of $\vdash \Gamma, B$ has been discarded. When cut-elimination is regarded as computation via the Curry-Howard isomorphism, this mechanism allows conditionals to be programmed.

Each of the binary connectives has a unit: I for $\otimes$, $\perp$ for $\parr$, $\top$ for $\&$ and $\mathbf{0}$ for $\oplus$. There is a rule for introducing each unit, apart from $\mathbf{0}$. In each case, the unit rule is the nullary form of the rule for the corresponding connective.

The exponential rules allow contraction and weakening to be carried out on formulae to which the modality $?$ (dual to $!$) has been applied. The Promotion rule allows $!$ to be introduced, as long as the context consists only of formulae of the form $?X$. A proof of $\vdash \Gamma, !A$ should be thought of as offering a choice between copying $!A$, discarding it, and converting it to A. The formula $!A$ can be cut against $?A^{\perp}$, and the cut-elimination procedure chooses what to do with $!A$ on the basis of the rule which was used to introduce $?A^{\perp}$. The Contraction rule corresponds to copying, the Weakening rule to dscarding, and the Dereliction rule to removing the $!$. For example, the cut

$$\dfrac{\vdash ?\Gamma, !A \qquad \dfrac{\vdash \Delta, ?A^{\perp}, ?A^{\perp}}{\vdash \Delta, ?A^{\perp}}\,\text{Contraction}}{\vdash ?\Gamma, \Delta}\,\text{Cut}$$

reduces to

$$\dfrac{\dfrac{\vdash ?\Gamma, !A \qquad \dfrac{\vdash \Delta, ?A^{\perp}, ?A^{\perp}}{\vdash ?\Gamma, \Delta, ?A^{\perp}}\,\text{Cut}}{\vdash ?\Gamma, ?\Gamma, \Delta}\,\text{Cut}}{\vdash ?\Gamma, \Delta}\,\text{Contraction}$$

in which the final application of Contraction is actually several applications, one to each formula in $?\Gamma$.

Finally, a useful derived connective is linear implication.

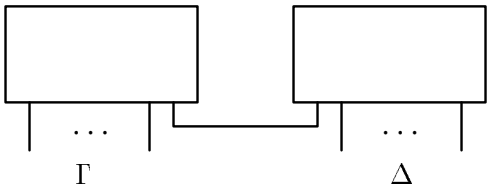
$$A \multimap B \stackrel{\text{def}}{=} A^{\perp} \parr B$$

Section 1.3 describes Abramsky's next step in the direction of a Curry-Howard isomorphism for concurrency, and shows how these sequent rules can be interpreted as constructions on processes.

1.3 Proofs as Processes

In an unpublished lecture [Abr91] Abramsky proposed an interpretation of classical linear logic proof nets as concurrent processes (specifically, π-calculus terms)—the Proofs as Processes interpretation. This interpretation is the basis for a concurrent Curry-Howard isomorphism. Both the interpretation of a classical linear logic sequent as the interface specification of a process, and the interpretation of the connectives as constructions on processes, are direct precursors of the interaction categories paradigm. Following on from Abramsky's original lecture, Bellin and Scott [BS94] have made a detailed study of the translation of proof nets into the π-calculus; a version of Abramsky's original lecture has also now been published [Abr94c].

The starting point of the Proofs as Processes interpretation is that a classical linear logic sequent $\vdash A_1, \ldots, A_n$ corresponds to a process interface specification: there are n ports in the interface, and the ports have types $A_1, \ldots, A_n$. In the case of functional programming, a sequent $A_1, \ldots, A_n \vdash B$ describes the n inputs and single output of a function; using one-sided sequents for process interface specifications means that this way of distinguishing between inputs and outputs is lost. Instead, the linear negation $(-)^{\perp}$ is used to represent the distinction between input and output. If A is the type of some output port, then $A^{\perp}$ is the type of an input port to which that output can be connected. When there are two processes with compatible input and output ports, the Cut rule can be used to form a connection between them. Pictorially, the construction looks like this.



The connection between the two processes is a private channel. This corresponds to a restricted or hidden name in process calculus. Hence the Proofs as Processes slogan:

cut is parallel composition + restriction.

The interpretation of the Axiom is a process with two ports, of types A and $A^{\perp}$. The actual process used is a bidirectional buffer, which copies information from one port to the other. This process allows either port to be used as an input—the only restriction is that when data is received at one port, it is sent out from the other.

Because a process has several ports, which do not necessarily appear in a fixed order within a sequent, a process interface actually consists of a set of named and typed ports, written

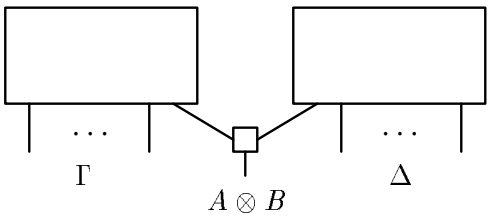
$$\vdash x_1 : A_1, \ldots, x_n : A_n.$$

Abramsky uses expressions of the form

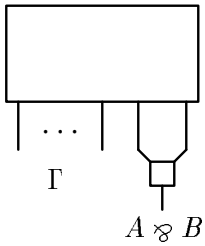
$$P \vdash x_1 : A_1, \ldots, x_n : A_n$$

in which P is the process having the specified interface. This should not be confused with the standard notation $\Gamma \vdash A$ for intuitionistic entailment. Chapter 6 introduces a typed process calculus in which typed processes have exactly this form.

The sequent rules for the multiplicatives allow ports to be grouped into bundles. The Tensor rule takes two ports, one from each of two processes, and combines them into a single port.



The Par rule is similar, but operating on two ports from the same process.



This geometrical view of the distinction between $\otimes$ and $\parr$ is a theme which will recur at various points throughout this thesis.

The additive connectives also have process interpretations: $\&$ as a conditional construction, offering a choice between two different processes, and $\oplus$ as the dual choice-making operator. The exponential rules, which in classical linear logic allow proofs to be duplicated, give a means of copying processes. This is related to the replication operator of the π-calculus, and hence to recursion: recursive definitions can be encoded in terms

of replication. It is also possible to give process interpretations of the unit rules, although Abramsky does not actually do this because the units do not have very natural interpretations as types.

1.4 Synchrony and Asynchrony

The terms *synchronous* and *asynchronous* are used with a variety of meanings by different authors. In this thesis they have specific meanings which may not coincide exactly with their meanings elsewhere. Here, *synchronous* means that there is a global clock with respect to which actions or events are synchronised. This is a strong requirement: when considering a process with several ports, something must happen in every port at every clock tick. The result is a situation similar to that of SCCS [Mil83]. On the other hand, in the asynchronous models which are considered in this thesis it is always possible for processes to delay, and even when two processes are interacting there may be periods during which they proceed independently with no synchronisation. The view of communication underlying these models is like that of CCS [Mil89], so there must be the possibility of at least enough synchronisation to allow one process to input at the same time as another outputs; in contrast to CCS, simultaneous actions are allowed even when they do not involve communication.

1.5 Outline of the Thesis

Chapter 2: Background

This chapter reviews some background material which is needed in later chapters. The first part defines labelled transition systems and the notions of strong bisimulation and observation equivalence. It also defines the syntax and operational semantics of CCS and SCCS, so that they can be used as informal notations later on. The second part collects together the definitions leading up to that of a linear category, and presents the details of the general construction of the linear logic exponentials which will appear in a particular case when $\mathcal{SP}roc$ is defined.

Chapter 3: Interaction Categories

In this chapter the two basic examples of interaction categories are defined: $\mathcal{SP}roc$, the category of synchronous processes, and $\mathcal{ASP}roc$, the category of asynchronous processes. The relationship between them is discussed, and ways of abstractly axiomatising interaction categories are considered. This chapter forms the foundation for the rest of

the thesis; later chapters make extensive use of $\mathcal{SP}roc$, $\mathcal{ASP}roc$ and variations on them.

Chapter 4: Synchronous Dataflow

This chapter contains the first application of interaction categories—an analysis of dataflow. First, the synchronous dataflow language LUSTRE is described, and then a general scheme for constructing dataflow networks in $*$-autonomous categories is presented. Using this scheme, a model of LUSTRE is defined in the category $\mathcal{SP}roc$. In general, modelling synchronous dataflow networks in $\mathcal{SP}roc$ gives an alternative to the classical Kahn semantics, and this chapter also contains a comparison of the $\mathcal{SP}roc$ semantics and the Kahn semantics, including sufficient conditions on a network for the two models to agree. Finally, this comparison is used to show that the $\mathcal{SP}roc$ semantics of LUSTRE is in agreement with the Kahn semantics which has previously been used.

Chapter 5: Verification

In this chapter, the interaction categories paradigm is used to address verification issues. Two kinds of property are discussed: safety and deadlock-freedom. While safety properties can be handled in $\mathcal{SP}roc$ and $\mathcal{ASP}roc$, more refined properties require new categories. Abramsky's idea of specification structures is presented; this is a general way of adding properties to the types in some category. The first example is his specification structure for synchronous deadlock-freedom: this yields a category based on $\mathcal{SP}roc$ in which typed processes do not deadlock and typed composition preserves this property. Next, a category of asynchronous deadlock-free processes is constructed, using a similar idea but starting from a variation of $\mathcal{ASP}roc$ (in which a notion of fairness is introduced as a way of dealing with problems of divergence). In both the synchronous and asynchronous cases, examples are given which illustrate how types can be used to assist with reasoning about deadlock-freedom.

Chapter 6: Typed Process Calculus

This chapter introduces a typed calculus of synchronous processes, based on the structure of interaction categories. The operational semantics and the way it relates to types lead to a new perspective on traditional subject reduction theorems: in the calculus, transitions do not preserve types, but change them in predictable ways. The calculus can be given a denotational semantics in any synchronous interaction category, for example $\mathcal{SP}roc$; when $\mathcal{SP}roc$ is used, a semantic soundness theorem relates the operational and denotational semantics by establishing that every typed process satisfies the safety specification of its semantic type. Furthermore, the categorical semantics

is sound with respect to the natural notion of strong bisimulation arising from the operational semantics.

Chapter 7: Conclusions and Future Work

The final chapter summarises the results of the thesis, and reflects on the relationship between the material in the thesis and other approaches to typed concurrency. It also indicates some possible directions for future work on interaction categories and their applications.

1.6 Related Work

There have been a number of recent pieces of work in the area of types for concurrency. These will now be surveyed, so that their approaches can be borne in mind as the thesis develops. At the end, their possible connections with interaction categories will be indicated.

Sorts in the π-calculus

The π-calculus was introduced by Milner, Parrow and Walker [MPW89] as a candidate canonical calculus for processes. An introduction can be found in [Mil91]. In this calculus, processes communicate by sending values to each other along named channels; a key feature is that channel names can themselves be sent as values. Milner defines the notions of *sort* and *sorting* as follows. To start with, a collection $\mathcal{S}$ of *subject sorts* is assumed. Each name is then assigned a subject sort, and sorted names are denoted $x : S$. An object sort is a finite sequence of subject sorts. A *sorting* over $\mathcal{S}$ is a partial function $ob : \mathcal{S} \rightharpoonup \mathcal{S}^*$ which describes, for any name $x : S$, the sort of name-vector which can be input or output along it. For example, if there are sorted names $x : S$, $y : T$, $z : U$ and a sorting ob such that $ob(S) = (TU)$, then $x(yz)$ and $\bar{x}[yz]$ are both valid prefixes but $x(y)$ and $\bar{x}[zy]$ are not. The sort of an agent is a sequence of subject sorts, which are the sorts of the names over which the agent is abstracted; thus a process (whose abstractions have been located at particular names) always has sort $()$. If a process has been constructed within the discipline of some sorting ob it is said to be *well-sorted* for ob. A well-sorted process uses names consistently: when reduced according to the operational semantics, it will never be the case that, for example, one subprocess tries to output two names along a name x while another subprocess is trying to input three names.

Sortings are made formal by the introduction of a collection of well-sorted process

formation rules, analogous to typed term-formation rules in type theory. By analogy with functional programming, such rules can be used as the basis of an algorithm which constructs, if possible, a sorting respected by a given process [Gay93, HV92, Tur94]. Sorts can be extended to incorporate a distinction between input and output [PS93].

Knowing that a process is well-sorted guarantees a particular kind of good behaviour, and this is one benefit which a type system is expected to provide. However, sorts do not fit very well into the view of a type as an interface specification. This is because as soon as the ports on which inputs and outputs will be received and sent have been specified, a process has sort (), so the sort does not convey any information about whether the process can communicate with others.

Types for Dyadic Interaction

Honda [Hon93] has proposed a process calculus with a type system which conveys more information than sorts. Types are defined by the grammar

$$\delta ::= \alpha \mid \downarrow \delta \mid \uparrow \delta \mid \delta\,;\delta \mid \delta \,\&\, \delta \mid \delta \oplus \delta$$

in which α is an atomic type, $\uparrow$ and $\downarrow$ denote output and input, $\delta\,;\delta$ is sequential composition, and $\&$ and $\oplus$ offer and make choices. There is also a notion of duality: each type δ has a co-type $\overline{\delta}$, defined inductively to make sequential composition self-dual, $\&$ and $\oplus$ duals, and $\uparrow$ and $\downarrow$ duals. The next step is to introduce typed names, and then construct typed actions and typed processes from them. Actions are constructed from constant symbols applied to names, and inductively with rules matching the grammar of types. Terms are constructed inductively with rules for nil, parallel composition, restriction and replication. There are two constructions which move between actions and terms. If P is a term, $.P$ is an action. If Ω is an action and a is a name, $a : \Omega$ is a term in which the action has been located at the name.

To define reduction of terms, a reduction relation is first defined which allows two actions to reduce to a term. Reduction of terms then allows two located actions in parallel to reduce (if their names match) to the term resulting from reduction of the actions. The transition system of terms with this reduction relation is then studied.

A system of type inference is given, dealing with sequents of the form $\vdash P \succ \Gamma$ in which P is a term and Γ is an assignment of types to names. Γ can be thought of as a description of the interface of the process P, except that the typing rules do not force the names mentioned in Γ to appear in P. For terms typed in this system, with the transition relation defined previously, a subject reduction theorem is proved and as a corollary it is shown that reducing well-typed terms does not lead to run-time errors. For a restricted class of terms, the *name-complete* terms (in which names and

co-names always appear together so that communication should always be possible), it is shown that typability implies deadlock-freedom.

The types studied in this work extend sorts by providing a number of type constructors which allow the formation of hierarchies of types rather than the fairly flat structure of sorts. As Honda points out, the type system remains very syntactic—there is no semantics of types, and this limits the amount of information which can be deduced from the fact that a process has a certain type.

Typed Additive Concurrency

Ferrari and Montanari [FM94] have a goal similar to that described in the introduction to this thesis, namely to discover a good notion of type for concurrency and give a semantic universe of typed process behaviours. Their approach is rather different from the sorting approach and Honda's; as will become clear, it has some similarities with the interaction categories approach. They give a semantics of typed transitions (and more generally, sequences of transitions) in terms of matrices. The interpretation of a transition $E_1 : \lambda_1 \xrightarrow{a} E_2 : \lambda_2$ between typed process terms $E_1 : \lambda_1$ and $E_2 : \lambda_2$ is a matrix of shape $\lambda_1 \times \lambda_2$. The notion of type which they propose, and which can be used to index rows and columns of a matrix, is *locality type*. A locality type is essentially a binary tree whose leaves represent "processing sites", which seem to correspond to the idea of ports. The processing sites are named by paths from the root; concretely the name of a processing site is a sequence of l and r labels. The idea is that the type of a process represents the "degree of concurrency", in terms of number of processing sites, required for its execution. A matrix of shape $\lambda_1 \times \lambda_2$ has m rows and n columns, where m and n are the numbers of leaves in the locality types λ_1 and λ_2 respectively. The entries in a matrix are elements of some idempotent semiring (a ring without additive inverses in which addition is idempotent), and part of the interpretation of transitions is an assignment of an element of the underlying semiring to each atomic action. It is perfectly possible for matrices representing transitions to be non-square, which corresponds to situations in which a process changes its type during the course of a transition.

In the CCS-like process calculus which they consider, a matrix is given for each atomic transition $a.E \xrightarrow{a} E$, and there is a matrix construction rule for each clause in the structured operational semantics. Thus there is a rule for each of the three clauses defining parallel composition, so that each possible transition of $a.E \mid \bar{a}.F$ is represented by a different matrix. Sequences of transitions are represented by matrices constructed by multiplication. Since matrices can be considered as the morphisms of a category whose objects are locality types, the general scheme is that types are objects

and process *behaviours* are morphisms.

This interpretation of transitions is defined independently of the underlying semiring; the only requirement is that atomic actions can be represented as elements of the semiring. So by choosing the semiring appropriately, different properties of actions can be considered. A matrix can be thought of as an observation of a transition, and hence it is possible to treat different notions of observation in the same framework. Several examples are given, in which different semirings are used to capture different aspects of process behaviour. Bisimulation can be defined, with the requirement of matching not actions themselves but their matrix interpretations. Again, different underlying semirings lead to different bisimulations.

The primary motivation for this work is the use of matrices to give a semantics of process behaviours; there is much less emphasis on the types and what they might say about processes.

Action Structures

In a recent series of papers [Mil93a, Mil93b, Mil93c, Mil93d] Milner has introduced and studied *action structures*. An action structure is an algebra in which process behaviours can be described and analysed. The intention is that by axiomatising the essential features of *actions* at a suitable level of generality, action structures can serve as the foundation of a wide variety of models of concurrency and interaction. There are two points of contact between the theory of action structures and the ideas of this thesis. The first is that action structures are categories; the second is that in these categories the objects, which play the rôle of types, have a monoidal structure.

An action structure is a strict monoidal category with certain additional properties. The objects of the category are called *arities*, and the monoidal operation is denoted by $\otimes$. The arity $\mathbf{1}$ is a strict unit, so that $\mathbf{1} \otimes m = m = m \otimes \mathbf{1}$ for any arity m. The morphisms of the category are called *actions*, and can be thought of as the building blocks of process behaviours. Given a pair m, n of arities in an action structure A, the collection $A_{m,n}$ of morphisms between them is equipped with a preorder, written $\searrow$ and called *reaction*. This preorder is used to describe evolution of actions: if $a \searrow b$ then the action a can become the action b by carrying out some computational activity. There is also a collection of *names*, and for each name x a functor $\mathbf{ab}_x$ which is used to capture the effect of abstracting over x.

There is a uniform construction of a process calculus over any action structure. A syntax similar to that of the π-calculus is defined—this syntax includes operations of delay, replication, prefixing, non-deterministic sum, abstraction and tensor product. The tensor product of processes is a synchronous parallel composition, from which

asynchronous parallel composition can be defined by means of the delay operator. Prefix actions are taken from the underlying action structure, and process transitions are defined in terms of the reaction preorder. This construction assigns an arity to each process; the tensor operation on arities corresponds to the tensor product of processes and the abstraction functors correspond to the operation of abstraction on processes, which is used to parameterise processes on inputs. Once the process calculus has been defined, another uniform construction yields a bisimulation congruence for it. Further papers [MMP94, Mil94] introduce the idea of a *control structure*, which provides an alternative means of describing the constructions of a process calculus.

Thus the action structures theory supports a general construction of typed process calculi. Milner gives examples of action structures from which synchronous and asynchronous versions of the π-calculus can be generated; in these examples the arities usually consist of the natural numbers with addition as the monoidal operation, but the π-calculus with sorting can be obtained by taking the arities to be sequences of sorts. The theory developed so far concentrates on the use of action structures to describe process dynamics, rather than the possible use of the type structure to analyse program specifications; this is a key difference between it and the theory developed in this thesis.

2

Background

2.1 Introduction

This chapter presents some background material which it will be useful to have available in Chapter 3. It is divided into two parts. The first reviews some preliminaries on labelled transition systems and process calculus, using the notation of CCS [Mil89] and SCCS [Mil83]. It defines bisimulation and observation equivalence, which are the fundamental equivalences used for synchronous and asynchronous processes in the rest of the thesis. The main proof technique, coinduction, is also defined. Similar definitions can be found in Milner's book [Mil89], in greater detail and with more discussion.

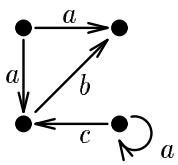
The second part makes a series of definitions leading up to that of a linear category. While these definitions are standard, it is not always easy to find them collected together and clearly presented in the literature. Also included is a description of Barr's construction [Bar91] of the linear logic exponential as a cofree cocommutative comonoid.

2.2 Process Calculus

2.2.1 Labelled Transition Systems

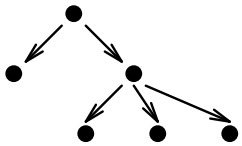
In operational terms, a process consists of a set of *states* between which it can move by performing various *actions*. This idea is formalised by the definition of labelled transition systems. A *labelled transition system* is a triple $(X, L, \longrightarrow)$ in which X is a set of *states*, L is a set of *labels* or *actions*, and $\longrightarrow \subseteq X \times L \times X$ is the *transition relation*. When $x, y \in X$ and $a \in L$, the notation $x \xrightarrow{a} y$ is used for $(x, a, y) \in \longrightarrow$. It is often convenient to refer to a labelled transition system by its set of states.

A labelled transition system can be described as a directed graph with labelled edges. The states are the vertices of the graph, and each edge represents a transition. When the set of states is finite, this can be a very easy way to define the transition relation. For example, the graph



defines a labelled transition system, with four states which have not been named.

The graph corresponding to a labelled transition system may be cyclic, but there is an important class of labelled transition systems whose graphs are trees, for example:



These are the *synchronisation trees*, which will be used to represent processes throughout the thesis.

When working with a process calculus, there is typically a labelled transition system whose states include all the processes which can be defined. Thus there is a blurring of the distinction between *processes* and *states*: each state of a process is a process in its own right, and every process is a state of the ambient labelled transition system.

2.2.2 Bisimulation

A *simulation* on a labelled transition system $(X, L, \longrightarrow)$ is a binary relation R on X such that if $(x, y) \in R$ then whenever $y \xrightarrow{a} y'$, there is $x' \in X$ such that $x \xrightarrow{a} x'$ and $(x', y') \in R$. The idea is that $(x, y) \in R$ then x can simulate the behaviour of y in the sense of being able to perform any action which y can, and this ability is inherited by the subsequent state. A *bisimulation* on a labelled transition system is a simulation whose converse is also a simulation. Explicitly, if R is a bisimulation and $(x, y) \in R$, then

- whenever $x \xrightarrow{a} x'$, there is $y' \in X$ such that $y \xrightarrow{a} y'$ and $(x', y') \in R$
- whenever $y \xrightarrow{a} y'$, there is $x' \in X$ such that $x \xrightarrow{a} x'$ and $(x', y') \in R$.

There are many bisimulations on a given labelled transition system; including, for example, the empty relation. A useful equivalence is defined by

$$\sim \stackrel{\text{def}}{=} \bigcup \{R \mid R \text{ is a bisimulation}\}.$$

This relation is itself a bisimulation; it is also an equivalence relation, and is the largest bisimulation in the sense that all others are included in it. From now on, the $\sim$ relation is referred to simply as bisimulation.

Bisimulation on a given labelled transition system can be characterised as the greatest fixed point of a certain operator on binary relations. If $(X, L, \longrightarrow)$ is a labelled transition system and $S \subseteq X \times X$, then the relation $F(R) \subseteq X \times X$ is defined by $(x, y) \in F(R)$ if and only if

- whenever $x \xrightarrow{a} x'$, there is $y' \in X$ such that $y \xrightarrow{a} y'$ and $(x', y') \in R$, and
- whenever $y \xrightarrow{a} y'$, there is $x' \in X$ such that $x \xrightarrow{a} x'$ and $(x', y') \in R$.

Bisimulation is the greatest (with respect to set inclusion) fixed point of F.

Because bisimulation is a greatest fixed point, the proof technique of *coinduction* [MT91] is naturally associated with it. It follows from the definition of F that a relation R is a bisimulation if and only if $R \subseteq F(R)$. Since $\sim$ is the largest bisimulation, to show that $x \sim y$ it is sufficient to show that $(x, y) \in R$ for some bisimulation R. In turn, to show that some relation R is a bisimulation it suffices to show that $R \subseteq F(R)$. This technique is used throughout the thesis when establishing instances of bisimulation.

2.2.3 Observation Equivalence

When studying processes it is often useful to regard certain actions as *silent* or *unobservable*. Such actions are typically used to represent internal communication steps whose details are not visible to the outside world. There is a variation of the notion of bisimulation, called weak bisimulation, which takes into account the unobservability of a particular action (which is usually called τ). It is similar to bisimulation in that one process has to match the transitions of another, but with the crucial difference that a τ transition can be matched by any number of τ transitions, including zero.

Suppose that $(X, L_\tau, \longrightarrow)$ is a labelled transition system, where $L_\tau = L \cup \{\tau\}$ and $\tau \notin L$. There is a labelled transition system $(X, L_\tau^*, \Longrightarrow)$ in which, if $t = a_1 \ldots a_n \in L_\tau^*$, $\stackrel{t}{\Longrightarrow}$ is defined by

$$x \stackrel{t}{\Longrightarrow} y \quad \stackrel{\text{def}}{\Leftrightarrow} \quad x(\xrightarrow{\tau})^* \xrightarrow{a_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{a_n} (\xrightarrow{\tau})^* y.$$

A transition $\stackrel{t}{\Longrightarrow}$ represents a sequence of $\longrightarrow$ transitions whose observable actions are as specified by t. Conversely, for any $s \in L_\tau^*$, $\hat{s} \in L^*$ is defined to be the sequence of actions obtained by removing all occurrences of τ from s.

A *weak bisimulation* on a labelled transition system $(X, L_\tau, \longrightarrow)$ is a binary relation R on X such that if $(x, y) \in R$ then

- whenever $x \xrightarrow{a} x'$, there is $y' \in X$ such that $y \stackrel{\hat{a}}{\Longrightarrow} y'$ and $(x', y') \in R$

- whenever $y \xrightarrow{a} y'$, there is $x' \in X$ such that $x \stackrel{\hat{a}}{\Longrightarrow} x'$ and $(x', y') \in R$.

The largest weak bisimulation is called *observation equivalence* and is written $\approx$. Observation equivalence is the greatest fixed point of the operator F defined by $(x, y) \in F(R)$ if and only if

- whenever $x \xrightarrow{a} x'$, there is $y' \in X$ such that $y \stackrel{\hat{a}}{\Longrightarrow} y'$ and $(x', y') \in R$, and
- whenever $y \xrightarrow{a} y'$, there is $x' \in X$ such that $x \stackrel{\hat{a}}{\Longrightarrow} x'$ and $(x', y') \in R$.

As before, the proof technique of coinduction is available.

In the CCS literature, bisimulations in which τ actions are not treated specially (i.e. the relations which were called bisimulations in the previous section) are known as *strong bisimulations*, and the largest strong bisimulation is called *strong equivalence*. This is the origin of the contrasting term "weak bisimulation". In this thesis, the equivalence $\sim$ is used when all actions are considered to be observable, and it is called bisimulation. The equivalence $\approx$ is used when τ is treated as unobservable, and it is called observation equivalence.

2.2.4 CCS and SCCS

Later in the thesis it will be convenient to use the syntax of both CCS and SCCS to describe processes which appear in some of the examples. CCS is defined over a set Act of *actions*, with $\tau \in \mathsf{Act}$. For each $a \in (\mathsf{Act} - \{\tau\})$ there is $\bar{a} \in (\mathsf{Act} - \{\tau\})$. The actions a and $\bar{a}$ are complementary in the sense that they can interact with each other during a communication.

The set $\mathcal{E}$ of *processes* is defined by the grammar

$$P ::= \mathsf{nil} \mid \alpha.P \mid P + P \mid (P \mid P) \mid P[f] \mid P \setminus A \mid \mathsf{fix}(X = P)$$

in which $\alpha \in \mathsf{Act}$, $A \subseteq (\mathsf{Act} - \{\tau\})$, $f : (\mathsf{Act} - \{\tau\}) \to (\mathsf{Act} - \{\tau\})$ and X is a variable. These expressions are given meaning by the transition rules in Figure 2.1, which define a labelled transition system $(\mathcal{E}, \mathsf{Act}, \longrightarrow)$. Considering the collection of transition rules as a system for proving that instances of the transition relation hold, there is one axiom: the rule allowing $\alpha.P$ to do the action α and become P. $P + Q$ is a process which may behave either as P or as Q, the choice being made at the time of its first transition. $P \mid Q$ consists of P and Q placed in parallel, which means that they may make independent transitions or communicate with each other. In a communication step P does the action l (which is not τ) and Q does $\bar{l}$; the result is a silent action τ. $P \setminus A$ is the process P with actions in the set A suppressed, and $P[f]$ is the process

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha, \bar{\alpha} \notin A}{P \setminus A \xrightarrow{\alpha} P' \setminus A} \qquad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \frac{E\{\mathsf{fix}(X = E)/X\} \xrightarrow{\alpha} P}{\mathsf{fix}(X = E) \xrightarrow{\alpha} P}$$

Figure 2.1: Operational Semantics of CCS

P with its actions renamed according to the function f. Finally, $\mathsf{fix}(X = E)$ is a recursively defined process whose transitions are obtained by unwinding the definition.

SCCS (Synchronous CCS) is a version of CCS in which simultaneous actions are allowed [Mil83]. Simultaneity is introduced by specifying that the set Act of actions has an abelian group structure. The group product is used to construct compound actions which consist of the simultaneous occurrence of their factors. Instead of the parallel composition $\mid$ of CCS, there is a synchronous product operation $\times$ which is defined by the single transition rule

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P \times Q \xrightarrow{\alpha\beta} P' \times Q'.}$$

If α and β are mutual inverses in this rule, the resulting action of $P \times Q$ is the group identity 1 which corresponds to τ in CCS.

The other difference between SCCS and CCS is that prefixing in SCCS is denoted by $\alpha : P$. The summation, restriction, relabelling and recursion operators are all present in SCCS.

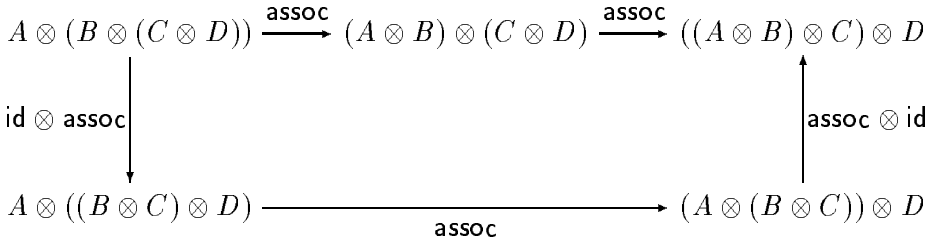
When working with CCS, a choice of equivalence is possible. If one does not wish to treat τ actions specially, then strong bisimulation is the equivalence to use. More commonly, τ is treated separately, and an equivalence is needed which treats τ as unobservable. It turns out that observation equivalence is not a congruence, i.e. is not preserved by all the process constructions, so the usual equivalence used is *observation congruence* which is defined to be the largest congruence contained in observation equivalence. Observation congruence is usually denoted by $=$ and often referred to simply as equality. As an added bonus, it is possible to give a complete axiomatisation for equality [Mil89]. When working with SCCS, strong bisimulation is always used. This thesis deals with both synchronous and asynchronous processes. Strong bisimulation is used as the equivalence in the synchronous case, and observation equivalence is used

in the asynchronous case. The general principle is to use the finest equivalence which makes all the desired equations (which generally means those specifying categorical structure) hold.

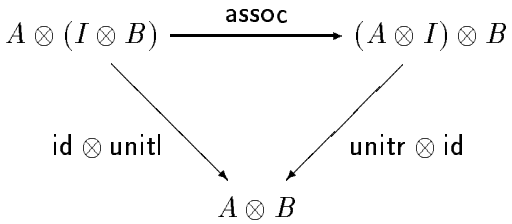
2.3 Linear Categories

As Seely [See87] points out, a categorical model of classical linear logic requires a $*$-autonomous category with finite products, and a comonad with some extra properties. A large amount of material on $*$-autonomous categories can be found in Barr's lecture notes [Bar79]; his later paper [Bar91] contains a selection of the material relevant to linear logic, together with some additional results. Martì-Oliet and Meseguer [MOM91] also give a clear definition of a $*$-autonomous category.

A *monoidal* category [Mac71] is a category $\mathbb{C}$ with a bifunctor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, an object I, and natural isomorphisms whose components are $\mathsf{unitl}_A : I \otimes A \to A$, $\mathsf{unitr}_A : A \otimes I \to A$ and $\mathsf{assoc}_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C$. The natural isomorphisms have to satisfy certain coherence conditions, namely that the diagrams



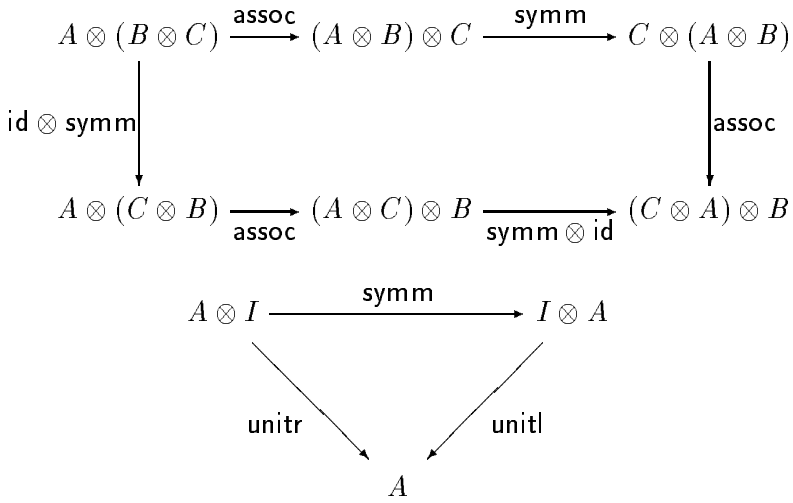
and



commute for all objects A, B, C, D of $\mathbb{C}$, and that $\mathsf{unitl}_I = \mathsf{unitr}_I : I \otimes I \to I$. The purpose of these coherence conditions is to ensure that no non-trivial automorphisms can be constructed from the basic natural isomorphisms.

A *symmetric* monoidal category is a monoidal category $\mathbb{C}$ with a natural isomorphism whose components are $\mathsf{symm}_{A,B} : A \otimes B \to B \otimes A$, such that $\mathsf{symm}^{-1}_{A,B} = \mathsf{symm}_{B,A}$ and satisfying the additional coherence conditions specified by commutativity of the

following diagrams for all objects A, B, C of $\mathbb{C}$.



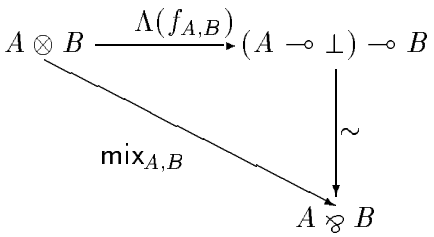
A symmetric monoidal *closed* category (also called an *autonomous* category) is a symmetric monoidal category $\mathbb{C}$ such that for each object B of $\mathbb{C}$, the functor $- \otimes B$ has a right adjoint. The adjoint functor is denoted by $B \multimap -$. The adjunction specifies a bijection between the homsets $\mathbb{C}(A \otimes B, C)$ and $\mathbb{C}(A, B \multimap C)$, natural in A and C. If $f : A \otimes B \to C$ then the corresponding morphism $A \to (B \multimap C)$ is called $\Lambda(f)$. The counit of the adjunction is a natural transformation whose components are morphisms $\mathsf{Ap}_{A,B} : (A \multimap B) \otimes A \to B$. Given $g : A \to (B \multimap C)$, the corresponding morphism $A \otimes B \to C$ is defined by $(g \otimes \mathsf{id}_B)\,;\mathsf{Ap}_{B,C}$. Λ and Ap are usually called *currying* and *application*.

A $*$-*autonomous* category is a symmetric monoidal closed category $\mathbb{C}$ with a *dualising object*, i.e. an object $\perp$ such that for every object A of $\mathbb{C}$, the morphism $\Lambda(\mathsf{symm}_{A,A\multimap\perp}\,;\mathsf{Ap}_{A,\perp}) : A \to (A \multimap \perp) \multimap \perp$ is an isomorphism. The functor $(-)^\perp : \mathbb{C}^{\mathrm{op}} \to \mathbb{C}$ is then defined by $A^\perp \stackrel{\mathrm{def}}{=} A \multimap \perp$, and is an equivalence between $\mathbb{C}$ and $\mathbb{C}^{\mathrm{op}}$. It also follows that $A \multimap B \cong (A \otimes B^\perp)^\perp$.

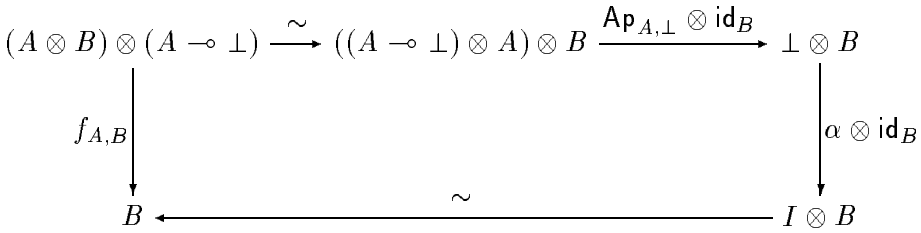
Defining a bifunctor $\parr$ by de Morgan duality, i.e. $A \parr B \stackrel{\mathrm{def}}{=} (A^\perp \otimes B^\perp)^\perp$, specifies another symmetric monoidal structure on $\mathbb{C}$ with $\parr$ as the monoidal operation, $\perp$ as unit, and the structural morphisms also defined by duality. So, for example, $\mathsf{parunitl}_A$ is defined by $\mathsf{parunitl}_A \stackrel{\mathrm{def}}{=} (\mathsf{unitl}^{-1}_{A^\perp})^\perp : \perp \parr A \to A$. Because $\perp$ is a dualising object, it also follows that $(I \multimap \perp) \cong \perp$. Furthermore, $A \multimap B \cong A^\perp \parr B$.

A $*$-autonomous category has sufficient structure to interpret the multiplicative fragment of classical linear logic. For the additives, finite products and coproducts are also required. Using linear logic notation rather than standard categorical notation, binary product is denoted by $\&$ and a terminal object by $\top$. Because of the duality, a $*$-autonomous category with finite products also has finite coproducts, defined by $A \oplus B \stackrel{\mathrm{def}}{=} (A^\perp \,\&\, B^\perp)^\perp$ and $\mathbf{0} \stackrel{\mathrm{def}}{=} \top^\perp$.

A *compact closed* category (in Barr's terminology, a *compact* category) is a $*$-autonomous category in which the functors $\otimes$ and $\parr$ are naturally isomorphic. Stating this formally requires first of all that there is an isomorphism $\alpha : \bot \cong I$. This allows a natural transformation $\mathsf{mix} : \otimes \dot{\to} \parr$ to be defined by



where $f_{A,B} : (A \otimes B) \otimes (A \multimap \bot) \to B$ is defined by



The term *linear category* has been used [See87, MOM91] for a $*$-autonomous category with finite products and coproducts, but in this thesis the term is reserved for a category which also has the structure needed to interpret the exponentials ! and ?.

A *linear* category is a $*$-autonomous category $\mathbb{C}$ with finite products and coproducts and a comonad $(!, \varepsilon, \delta)$, satisfying certain extra properties. For each A there should be a cocommutative comonoid structure

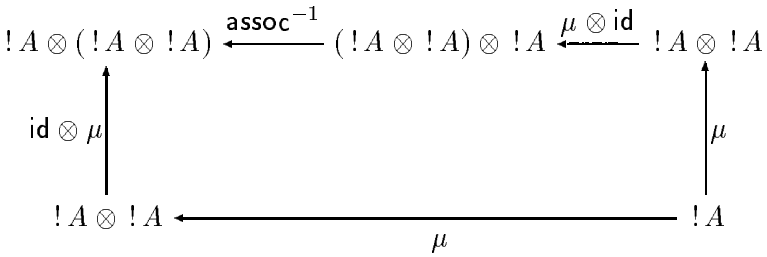
$$I \xleftarrow{\eta} !A \xrightarrow{\mu} !A \otimes !A$$

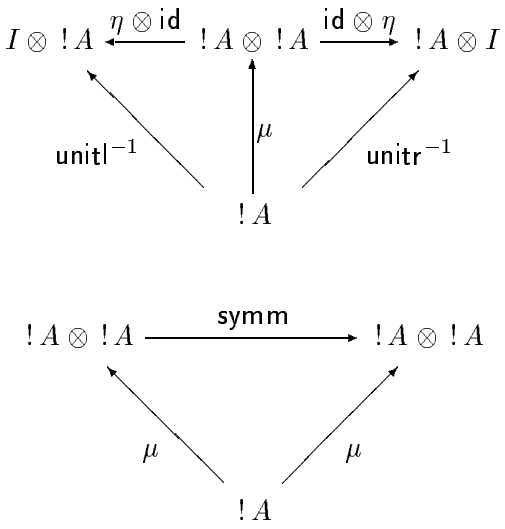
on $!A$, such that the additional isomorphisms

$$\left.\begin{array}{rcl} !(A \mathbin{\&} B) & \cong & !A \otimes !B \\ !\mathbf{1} & \cong & I \end{array}\right\} \tag{2.1}$$

hold. The functor ? is defined by de Morgan duality: $?A \stackrel{\text{def}}{=} (!(A^{\bot}))^{\bot}$.

For $(!A, \eta, \mu)$ to be a cocommutative comonoid, the diagrams

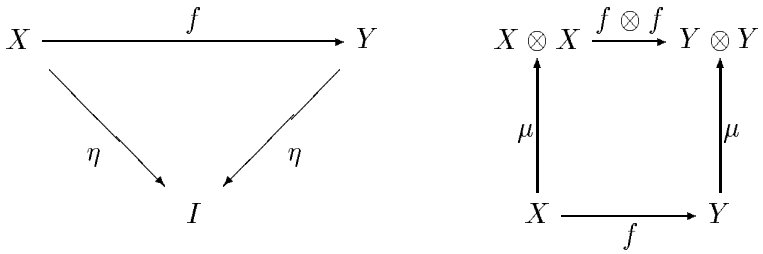




must commute.

The morphisms η and μ from the cocommutative comonoid structure interpret weakening and contraction; the morphism ε of the comonad structure interprets dereliction; and if $f : !A \to B$ then the promoted morphism $f^! : !A \to !B$ is defined by $f^! \stackrel{\text{def}}{=} \delta \,;\, !f$.

If objects X and Y of $\mathbb{C}$ have cocommutative comonoid structures, a cocommutative comonoid morphism $f : X \to Y$ is a morphism such that the diagrams



commute. $\mathcal{C}om(\mathbb{C})$ is the category of cocommutative comonoids in $\mathbb{C}$ with these morphisms. If the forgetful functor $U : \mathcal{C}om(\mathbb{C}) \to \mathbb{C}$ has a right adjoint, i.e. a cofree functor from $\mathbb{C}$ to $\mathcal{C}om(\mathbb{C})$, then the comonad $(!, \varepsilon, \delta)$ can be defined from this adjunction [Mac71]. In this case the functor $!$, being a right adjoint, preserves limits, so the product of A and B in $\mathbb{C}$ (i.e. $A \,\&\, B$) is mapped to the product of $!A$ and $!B$ in $\mathcal{C}om(\mathbb{C})$ (which, it turns out, is $!A \otimes !B$). Similarly, the terminal object $\top$ in $\mathbb{C}$ is mapped to the terminal object in $\mathcal{C}om(\mathbb{C})$, which is I. Thus the isomorphisms (2.1) hold for quite general reasons.

Specifying that $!$ can be obtained as a right adjoint is quite a strong requirement, which is by no means satisfied by all models of linear logic. But in some of the situations considered in this thesis, it is possible to use Barr's construction [Bar91] of cofree

cocommutative comonoids as symmetric algebras. This construction applies when the functors $\otimes$ and $\parr$ are isomorphic (in this situation, $\mathbb{C}$ is *compact closed* or simply *compact*) and $\mathbb{C}$ has countable *biproducts*, i.e. countable products and coproducts which are isomorphic. From now on, the binary biproduct will be written $\oplus$. The nullary case gives a zero object $\mathbf{0}$. Hence for each pair A, B of objects, there is a morphism $\mathbf{0}_{A\to B} : A \to B$ defined by $\mathbf{0}_{A\to B} \stackrel{\text{def}}{=} A \to \mathbf{0} \to B$, using the fact that $\mathbf{0}$ is initial and terminal.

The first step in the construction of ! is to define, for each $n \geqslant 0$, the *symmetric tensor power* functor $\otimes^n_s$. For each permutation $\sigma \in S_n$ there is a canonical isomorphism

$$p_\sigma : \otimes^n A \to \otimes^n A$$

constructed from symm. The object $\otimes^n_s A$ is defined by the simultaneous equaliser diagram

$$\otimes^n_s A \xrightarrow{e_A} \otimes^n A \xrightarrow{\{p_\sigma\}} \otimes^n A$$

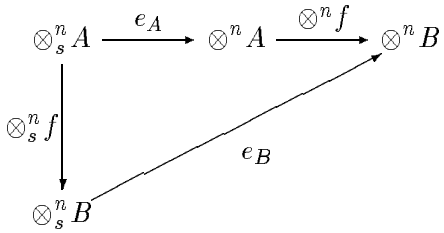
in which all the p_σ are equalised, if this equaliser exists in $\mathbb{C}$. If $f : A \to B$ then

$$\otimes^n_s f : \otimes^n_s A \to \otimes^n_s B$$

is defined as follows. For any $\sigma \in S_n$ there are canonical automorphisms p_σ and q_σ on $\otimes^n A$ and $\otimes^n B$ respectively. By naturality, $(\otimes^n f)\,;q_\sigma = p_\sigma\,;\otimes^n f$ and so

$$e_A\,;\otimes^n f : \otimes^n_s A \to \otimes^n B$$

has equal composites with all of the q_σ. Using the equaliser property, $\otimes^n_s f$ is defined to be the unique morphism $\otimes^n_s A \to \otimes^n_s B$ such that



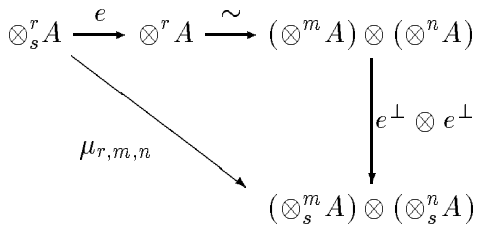
commutes. It is easy to verify, using the uniqueness of a morphism satisfying the equaliser property, that $\otimes^n_s$ preserves identities and composition. Finally, the functor ! is defined on objects by

$$!A \stackrel{\text{def}}{=} \oplus_{n\geqslant 0} \otimes^n_s A$$

and similarly on morphisms. The rest of the construction also requires that for every A and $n \geqslant 0$, $e_A\,;e^{\perp}_{A^\perp} = \mathsf{id}_{\otimes^n_s A}$.

For the cocommutative comonoid structure on $!A$, $\eta : !A \to I$ is defined by $\eta \stackrel{\text{def}}{=} \pi_0$, where the π_i are projections from a countable biproduct, using the fact that $\otimes^0_s A \cong I$.

To define $\mu : !A \to !A \otimes !A$, the first step is to use distributivity of $\otimes$ over $\oplus$ to write $!A \otimes !A \cong \oplus_{m,n}[(\otimes_s^m A) \otimes (\otimes_s^n A)]$. This means that given, for each r, m and n, a morphism $\mu_{r,m,n} : \otimes_s^r A \to (\otimes_s^m A) \otimes (\otimes_s^n A)$, μ can be defined from the product and coproduct properties of $\oplus$. If $m + n \neq r$, then $\mu_{r,m,n} \stackrel{\text{def}}{=} \mathbf{0}_{\otimes_s^r A \to (\otimes_s^m A) \otimes (\otimes_s^n A)}$. If $m + n = r$ then $\mu_{r,m,n}$ is defined by this diagram.



It is straightforward to verify that $(!A, \eta, \mu)$ is a cocommutative comonoid.

The adjunction specifying cofreeness requires a choice of natural bijection

$$\mathbb{C}(UA, B) \cong \mathcal{C}om(\mathbb{C})(A, !B)$$

for each $A \in \mathcal{C}om(\mathbb{C})$ and $B \in \mathbb{C}$. Given $h : A \to !B$, $\bar{h} : UA \to B$ is defined by $\bar{h} \stackrel{\text{def}}{=} h \,;\, \pi_1$, using the fact that $\otimes_s^1 B \cong B$. Given $g : UA \to B$, $\hat{g} : A \to !B$ is defined via a collection of morphisms $\hat{g}_n : A \to \otimes_s^n B$. To define $\hat{g}_n$, first note that the cocommutative comonoid structure of A gives a canonical morphism $\alpha : A \to \otimes^n A$ and that α has equal composites with all the $p_\sigma : \otimes^n A \to \otimes^n A$. Hence $\alpha \,;\, \otimes^n g$ has equal composites with all the $q_\sigma : \otimes^n B \to \otimes^n B$, and so $\hat{g}_n$ can be taken as the morphism defined by the equaliser property of $\otimes_s^n B$. By lengthy manipulation of diagrams arising from the definitions, it can be verified that $\hat{g}$ is a cocommutative comonoid morphism, $\bar{\hat{g}} = g$, and $\hat{\bar{h}} = h$.

When the comonad structure of $!$ is defined from this adjunction, $\varepsilon_A : !A \to A$ is given by $\varepsilon_A \stackrel{\text{def}}{=} \overline{\mathsf{id}_{!A}}$ (so concretely $\varepsilon_A = \pi_1$), and $\delta_A : !A \to !!A$ is given by $\delta_A \stackrel{\text{def}}{=} \widehat{\mathsf{id}_{!A}}$.

2.4 Discussion

The purpose of this chapter has been to gather together the prerequisites for the next chapter. The definitions relating to process calculus have been included in order to establish the framework in which subsequent work takes place. Now that the abstract structure of linear categories has been described, in Chapters 3 and 5 it will be sufficient to define concrete realisations of this structure in the particular categories under consideration. Of course, the main subject of this thesis is interaction categories rather than merely linear categories. Later chapters contain several examples of interaction categories, first defining their structure in concrete terms and then working towards an abstract definition.

3

Interaction Categories

3.1 Introduction

The philosophy of interaction categories is that composition is a dynamic process of *interaction* rather than a static composition of functions. This will be seen clearly throughout this chapter, and it means that interaction categories have a very different flavour from many of the categories familiar from traditional mathematics. Two interaction categories, both due to Abramsky, are defined in this chapter: the category $\mathcal{SP}roc$ of synchronous processes, and the category $\mathcal{ASP}roc$ of asynchronous processes. $\mathcal{SP}roc$ comes first, because historically it was the first to be discovered and also because it is a little simpler than $\mathcal{ASP}roc$. Following a similar sequence of definitions to that in Abramsky's papers [Abr93b, Abr94b], the structure of $\mathcal{SP}roc$ as a linear category is defined. After this comes the definition of the additional structure which makes $\mathcal{SP}roc$ an interaction category. This is followed by a description of an unsuccessful attempt to construct a category of asynchronous processes by applying delay operators to certain synchronous morphisms, and finally by the definition of $\mathcal{ASP}roc$, in which all processes are asynchronous. This thesis uses the "full" version of $\mathcal{ASP}roc$ rather than the simplified version which has appeared previously [Abr94a]; furthermore, the full version is presented in a slightly different way which makes the key differences from $\mathcal{SP}roc$ more apparent. Once the categories have been defined, the relationships between them and their different presentations are discussed. Although $\mathcal{ASP}roc$ does not arise from a general construction of asynchronous processes in terms of $\mathcal{SP}roc$, the two categories are not completely unrelated: there is an embedding of $\mathcal{ASP}roc$ into $\mathcal{SP}roc$. A comparison is also made between the presentation of $\mathcal{ASP}roc$ used in this thesis and Abramsky's presentation.

Although the name "interaction categories" was introduced only at the time of the discovery of $\mathcal{SP}roc$, there are other categories which, with hindsight, can also be seen as examples. These are the categories of games used by Abramsky and Jagadeesan to model linear logic [AJ94], Berry and Curien's category of concrete data structures and sequential algorithms [Cur93], and Abramsky and Jagadeesan's "geometry of interaction" categories [AJ92]. These examples are not discussed in this thesis, but the reader who is familiar with any of them may find it interesting to compare them with

$\mathcal{SP}roc$ and $\mathcal{ASP}roc$.

There is one question which has not yet been addressed: what is an interaction category, other than one of a collection of examples? At the time of writing, there is no abstract definition which both encompasses all of the examples, and specifies enough structure to support all of the desired constructions in particular categories. The final section of this chapter discusses possible axioms. It is convenient to distinguish between synchronous and asynchronous interaction categories; Chapter 6 uses one abstract definition of a synchronous interaction category to give a semantics to a typed calculus of synchronous processes. One of the aims of this thesis is to use interaction categories to analyse a variety of examples, and thus accumulate experience of which aspects of the structure are the most important and should be captured by an axiomatisation.

3.2 The Category $\mathcal{SP}roc$

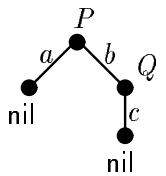
3.2.1 Processes

The first step in the definition of $\mathcal{SP}roc$ as a category of processes is a suitable definition of a process. Rather than working with strong bisimulation equivalence classes of labelled transition systems, it is convenient to use synchronisation trees as canonical representations of the equivalence classes. The most obvious way to formalise the notion of a synchronisation tree is to define the class ST_L of synchronisation trees with label-set L by the recursive equation

$$\mathsf{ST}_L \quad \cong \quad \wp(L \times \mathsf{ST}_L). \tag{3.1}$$

With this definition, a synchronisation tree consists of the set of transitions to its descendants: $P = \{(a, Q) \mid P \xrightarrow{a} Q\}$. The simplest way of solving the isomorphism (3.1) is to use Aczel's theory of non-well-founded sets [Acz88]. In this theory, ST_L is the final coalgebra of the functor $X \mapsto \wp(L \times X)$. The final coalgebra property supports non-well-founded recursive definitions of elements of ST_L, and provides a principle of coinduction. A detailed development of non-well-founded set theory, and a description of its application to processes, can be found in Aczel's book.

The processes in the synchronisation tree



are represented by the sets

$$\begin{aligned} P &\stackrel{\text{def}}{=} \{(a, \mathsf{nil}), (b, Q)\} \\ Q &\stackrel{\text{def}}{=} \{(c, \mathsf{nil})\} \\ \mathsf{nil} &\stackrel{\text{def}}{=} \varnothing. \end{aligned}$$

In this thesis, processes will usually be defined either by SCCS (or, when asynchronous processes are considered, CCS) prefixing and summation notation, or by transition rules. For example,

$$\begin{aligned} P &\stackrel{\text{def}}{=} a : \mathsf{nil} + b : Q \\ Q &\stackrel{\text{def}}{=} c : \mathsf{nil} \end{aligned}$$

or

$$P \xrightarrow{a} \mathsf{nil} \qquad P \xrightarrow{b} Q \qquad Q \xrightarrow{c} \mathsf{nil}.$$

These processes have only finite behaviours, but infinite processes can be defined by recursive equations or transition rules. For example,

$$\begin{aligned} P &\stackrel{\text{def}}{=} a : Q \\ Q &\stackrel{\text{def}}{=} b : P \end{aligned}$$

is typical of the kind of non-well-founded recursive definition which Aczel's theory supports.

3.2.2 $\mathcal{SP}roc$ as a Category

An object of $\mathcal{SP}roc$ is a pair $A = (\Sigma_A, S_A)$ in which Σ_A is an *alphabet* (*sort*) of *actions* (*labels*) and $S_A \subseteq^{nepref} \Sigma_A^*$ is a *safety specification*, i.e. a non-empty prefix-closed subset of Σ_A^*. If A is an object of $\mathcal{SP}roc$, a *process of type* A is a process P with sort Σ_A such that $\mathsf{traces}(P) \subseteq S_A$. Considering P as a labelled transition system, $\mathsf{traces}(P)$ is the set of sequences labelling finite paths from the root. The set of sequences labelling finite and infinite paths is $\mathsf{alltraces}(P)$ and the set of sequences labelling infinite paths is $\mathsf{inftraces}(P)$. The following coinductive definition is equivalent to

this description.

$$\begin{aligned} \mathsf{alltraces}(P) &\stackrel{\text{def}}{=} \{\varepsilon\} \cup \{a\sigma \mid P \xrightarrow{a} Q, \sigma \in \mathsf{alltraces}(Q)\} \\ \mathsf{traces}(P) &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{alltraces}(P) \mid \sigma \text{ is finite}\} \\ \mathsf{inftraces}(P) &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{alltraces}(P) \mid \sigma \text{ is infinite}\}. \end{aligned}$$

The fact that P is a process of type A is expressed by the notation $P : A$. This use of ":" should not cause any confusion with the synchronous prefix operation.

The most convenient way of defining the morphisms of $\mathcal{SP}roc$ is first to define a $*$-autonomous structure on objects, and then say that the morphisms from A to B are processes of the internal hom type $A \multimap B$. This style of definition is typical of interaction categories; definitions of categories of games [AJ94] follow the same pattern. Given objects A and B, the object $A \otimes B$ has

$$\begin{aligned} \Sigma_{A\otimes B} &\stackrel{\text{def}}{=} \Sigma_A \times \Sigma_B \\ S_{A\otimes B} &\stackrel{\text{def}}{=} \{\sigma \in \Sigma^*_{A\otimes B} \mid \mathsf{fst}^*(\sigma) \in S_A, \mathsf{snd}^*(\sigma) \in S_B\}. \end{aligned}$$

The duality is trivial on objects: $A^\perp \stackrel{\text{def}}{=} A$. This means that at the level of types, $\mathcal{SP}roc$ makes no distinction between input and output. Because communication in $\mathcal{SP}roc$ consists of synchronisation rather than value-passing, processes do not distinguish between input and output either.

The definition of $\otimes$ makes clear the extent to which processes in $\mathcal{SP}roc$ are synchronous. An action performed by a process of type $A \otimes B$ consists of a pair of actions, one from the alphabet of A and one from that of B. Thinking of A and B as two ports of the process, synchrony means that at every time step a process must perform an action in every one of its ports.

In a $*$-autonomous category in which $A^\perp \cong A$, it follows that $A \otimes B \cong A \parr B$ and hence the category is compact closed. Furthermore, $A \multimap B \cong A \otimes B$. In $\mathcal{SP}roc$, the objects A and $A^\perp$ are not just isomorphic but are actually defined to be *equal*, and so the multiplicative connectives coincide up to equality: $A \parr B = A \multimap B = A \otimes B$. However, preserving the notational distinction between the connectives will make it clear that in many parts of this thesis a less degenerate category than $\mathcal{SP}roc$ could be used. There will also be times when the compact closed structure of $\mathcal{SP}roc$ is crucial for certain applications. Not all interaction categories are compact closed, but in general those that are support more process constructions than those that are not. This will become clearer in Chapters 4 and 5.

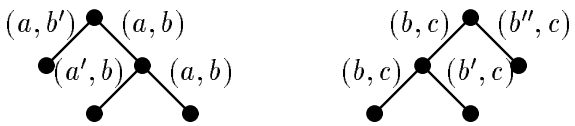
A morphism $p : A \to B$ of $\mathcal{SP}roc$ is a process p of type $A \multimap B$ (so p has to satisfy a certain safety specification). Since $A \multimap B = A \otimes B$ in $\mathcal{SP}roc$, this amounts to saying that a morphism from A to B is a process of type $A \otimes B$. The reason for giving the definition in terms of $\multimap$ is that it sets the pattern for all interaction category definitions, including cases in which there is less degeneracy.

If $p : A \to B$ and $q : B \to C$ then the composite $p\,;q : A \to C$ is defined by labelled transitions.

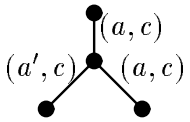
$$\frac{p \xrightarrow{(a,b)} p' \qquad q \xrightarrow{(b,c)} q'}{p\,;q \xrightarrow{(a,c)} p'\,;q'}$$

At each step, the actions in the common type B have to match. The processes being composed constrain each other's behaviour, selecting the possibilities which agree in

B. For example, if p and q are as shown:



then $p\,;q$ is this tree.



This ongoing communication is the "interaction" of interaction categories. If the processes in the definition terminated after a single step, so that each could be considered simply as a set of pairs, then the labelled transition rule would reduce to precisely the definition of relational composition. This observation leads to the *SProc* slogan: *processes are relations extended in time.*

The identity morphisms are synchronous buffers: whatever is received by $\mathsf{id}_A : A \to A$ in the left copy of A is instantaneously transmitted to the right copy (and *vice versa*—there is no real directionality). This is exactly the Proofs as Processes interpretation of an identity axiom. The following auxiliary definition helps to define the identity processes. If P is a process with sort Σ and $S \subseteq^{nepref} \Sigma^*$ then the process $P\restriction S$, also with sort Σ, is defined by the transition rule

$$\frac{P \xrightarrow{a} Q \qquad a \in S}{P\restriction S \xrightarrow{a} Q\restriction(S/a)}$$

where $S/a \stackrel{\text{def}}{=} \{\sigma \mid a\sigma \in S\}$. Note that the condition $a \in S$ in the transition rule refers to the singleton sequence a rather than the action a. In this thesis, there is no notational distinction between these uses of a.

The identity morphism $\mathsf{id}_A : A \to A$ is defined by $\mathsf{id}_A \stackrel{\text{def}}{=} \mathsf{id}\restriction S_{A \multimap A}$ where the process id with sort Σ_A is defined by the transition rule

$$\frac{a \in \Sigma_A}{\mathsf{id} \xrightarrow{(a,a)} \mathsf{id}.}$$

Proposition 3.1 *SProc* is a category.

Proof: The first step is to prove that if $f : A \to B$ and $g : B \to C$, then

$$f\,;g : A \to C.$$

This requires $\mathsf{traces}(f\,;g) \subseteq S_{A \multimap C}$. Firstly, $\mathsf{traces}(f) \subseteq S_{A \multimap B}$ implies

$$\{\mathsf{fst}^*(s) \mid s \in \mathsf{traces}(f)\} \subseteq S_A,$$

and similarly $\mathsf{traces}(g) \subseteq S_{B \multimap C}$ implies

$$\{\mathsf{snd}^*(s) \mid s \in \mathsf{traces}(g)\} \subseteq S_C\,.$$

If $s \in \mathsf{traces}(f\,;g)$ then $\mathsf{fst}^*(s) = \mathsf{fst}^*(t)$ for some $t \in \mathsf{traces}(f)$, and $\mathsf{snd}^*(s) = \mathsf{snd}^*(t')$ for some $t' \in \mathsf{traces}(g)$. Hence $s \in S_{A \multimap C}$.

The proof that composition is associative is by coinduction. We wish to prove that for any $f : A \to B$, $g : B \to C$ and $h : C \to D$, $f\,;(g\,;h) = (f\,;g)\,;h$. Define a relation R on $\mathsf{ST}_{\Sigma_{A \multimap D}}$ by

$$R \stackrel{\text{def}}{=} \{(u\,;(v\,;w),(u\,;v)\,;w) \mid u \in \mathsf{ST}_{\Sigma_{A \multimap B}}, v \in \mathsf{ST}_{\Sigma_{B \multimap C}}, w \in \mathsf{ST}_{\Sigma_{C \multimap D}}\}.$$

The aim is to show that R is a bisimulation, and hence is contained in strong bisimulation; since strong bisimulation is equality, this completes the proof.

If $u\,;(v\,;w) \xrightarrow{\alpha} x$ then from the transition rule defining composition, $\alpha = (a,d)$ and there is some b such that $u \xrightarrow{(a,b)} u'$ and $v\,;w \xrightarrow{(b,d)} y$, with $x = u';y$. Again from the definition of composition, there is some c such that $v \xrightarrow{(b,c)} v'$ and $w \xrightarrow{(c,d)} w'$, with $y = v';w'$. So overall we have $u\,;(v\,;w) \xrightarrow{(a,d)} u'\,;(v'\,;w')$. Similarly $(u\,;v)\,;w \xrightarrow{(a,d)} (u'\,;v')\,;w'$. And from the definition of R, $(u'\,;(v'\,;w'),(u'\,;v')\,;w') \in R$. A symmetrical argument shows that $u;(v;w)$ can make the same transitions as $(u;v);w$, and that the subsequent states are related by R. Hence R is a bisimulation.

The proof that $f\,;\mathsf{id}_B = f$ when $f : A \to B$ is slightly less direct. If id is the process defined as before but with sort Σ_B, a coinduction argument similar to the one above establishes that $f\,;\mathsf{id} = f$, using the relation S on $\mathsf{ST}_{\Sigma_{A \multimap B}}$ defined by

$$S \stackrel{\text{def}}{=} \{(u\,;\mathsf{id},u) \mid u \in \mathsf{ST}_{\Sigma_{A \multimap B}}\}.$$

Because the behaviour of f in B satisfies the safety specification S_B, $f\,;(\mathsf{id}\restriction S_{B \multimap B}) = f\,;\mathsf{id} = f$ and so $f\,;\mathsf{id}_B = f$. □

3.2.3 $\mathcal{SP}$roc as a $*$-Autonomous Category

The definitions of $\otimes$ and $(-)^\perp$ can now be extended to morphisms, making them into functors. If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ and $p^\perp : C^\perp \to A^\perp$ are defined by transition rules.

$$\frac{p \xrightarrow{(a,c)} p' \qquad q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((a,b),(c,d))} p' \otimes q'} \qquad\qquad \frac{p \xrightarrow{(a,c)} p'}{p^\perp \xrightarrow{(c,a)} p'^\perp}$$

The tensor unit I is defined by

$$\Sigma_I \stackrel{\text{def}}{=} \{*\} \qquad\qquad S_I \stackrel{\text{def}}{=} \{*^n \mid n < \omega\}.$$

The following notation provides a useful way of defining the structural morphisms needed to specify the rest of the $*$-autonomous structure. If P is a process with sort Σ, and $f : \Sigma \rightharpoonup \Sigma'$ is a partial function, then $P[f]$ is the process with sort Σ' defined by

$$\frac{P \xrightarrow{a} Q \qquad a \in \mathsf{dom}(f)}{P[f] \xrightarrow{f(a)} Q[f].}$$

The canonical isomorphisms $\mathsf{unitl}_A : I \otimes A \cong A$, $\mathsf{unitr}_A : A \otimes I \cong A$, $\mathsf{assoc}_{A,B,C} : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$ and $\mathsf{symm}_{A,B} : A \otimes B \to B \otimes A$ are defined by

$$\begin{array}{lcl}
\mathsf{unitl}_A & \stackrel{\text{def}}{=} & \mathsf{id}_A[(a,a) \mapsto ((*,a),a)] \\
\mathsf{unitr}_A & \stackrel{\text{def}}{=} & \mathsf{id}_A[(a,a) \mapsto ((a,*),a)] \\
\mathsf{assoc}_{A,B,C} & \stackrel{\text{def}}{=} & \mathsf{id}_{A\otimes(B\otimes C)}[((a,(b,c)),(a,(b,c))) \mapsto ((a,(b,c)),((a,b),c))] \\
\mathsf{symm}_{A,B} & \stackrel{\text{def}}{=} & \mathsf{id}_{A\otimes B}[((a,b),(a,b)) \mapsto ((a,b),(b,a))].
\end{array}$$

If $f : A \otimes B \to C$ then $\Lambda(f) : A \to (B \multimap C)$ is defined by

$$\Lambda(f) \stackrel{\text{def}}{=} f[((a,b),c) \mapsto (a,(b,c))].$$

The evaluation morphism $\mathsf{Ap}_{A,B} : (A \multimap B) \otimes A \to B$ is defined by

$$\mathsf{Ap}_{A,B} \stackrel{\text{def}}{=} \mathsf{id}_{A\multimap B}[((a,b),(a,b)) \mapsto (((a,b),a),b)].$$

All of the structural morphisms are essentially formed from identities, and the only difference between f and $\Lambda(f)$ is a reshuffling of ports. In each of the above uses of relabelling, the partial function on sorts is defined by means of a pattern-matching notation.

If P is a process of type A then $P[a \mapsto (*,a)]$ is a morphism $I \to A$ which can be identified with P. This agrees with the view of global elements (morphisms from I, in a $*$-autonomous category) as inhabitants of types.

Proposition 3.2 $\mathcal{SP}roc$ is a compact closed category.

Proof: Verifying the coherence conditions for $\otimes$ is straightforward, given the nature of the canonical isomorphisms as relabelled identities. The properties required of Λ and Ap are equally easy to check. Since $(-)^\perp$ is trivial, it is automatically an involution. This gives the $*$-autonomous structure; compact closure follows from the coincidence of $\otimes$ and $\parr$. □

3.2.4 Safety Specifications, Isomorphisms and Minimal Types

So far, the safety specifications in the objects of $\mathcal{SP}roc$ have not been very prominent. This will change in the next section, when they become essential for the definition

of products and coproducts, but now is a good time to illustrate some of their other implications.

In $\mathcal{R}el$, the category of sets and relations, two objects are isomorphic precisely when they are isomorphic as sets, i.e. when there is a bijection between them. But in $\mathcal{SP}roc$, isomorphism is not determined by such a simple relationship between sorts. For example, let objects A and B be defined by $\Sigma_A \stackrel{\text{def}}{=} \{a\}$, $S_A \stackrel{\text{def}}{=} \{a^n \mid n < \omega\}$, $\Sigma_B \stackrel{\text{def}}{=} \{a, b\}$ and $S_B \stackrel{\text{def}}{=} \{b^n \mid n < \omega\}$. Then the morphisms $f : A \to B$ and $g : B \to A$ defined by $f = (a, b) : f$ and $g = (b, a) : g$ are mutual inverses. The point is that because S_B does not allow the action a to appear, B effectively has a singleton sort and so "looks like" A. Conversely, objects with the same sort need not be isomorphic. Suppose $\Sigma_A \stackrel{\text{def}}{=} \Sigma_B \stackrel{\text{def}}{=} \{a\}$, $S_A \stackrel{\text{def}}{=} \{\varepsilon, a\}$ and $S_B \stackrel{\text{def}}{=} \{\varepsilon, a, aa\}$. Because all traces in S_A have length $\leqslant 1$, any morphism into or out of A only has traces of length $\leqslant 1$. If $f : A \to B$ and $g : B \to A$ then $g ; f$ also has no traces of length > 1, but id_B has the trace $(a, a)(a, a)$ of length 2, and so $g ; f \neq \mathsf{id}_B$.

Because satisfaction of a type A by a process P is determined by a simple subset relationship ($\mathsf{traces}(P) \subseteq S_A$), if P has type A then it also has any type with a larger safety specification. Defining $\mathsf{sort}(P)$ to be the set of actions which P can actually do, the minimal type satisfied by P is $(\mathsf{sort}(P), \mathsf{traces}(P))$. Then for any object B with $\mathsf{sort}(P) \subseteq \Sigma_B$ and $\mathsf{traces}(P) \subseteq S_B$, $P : B$. There are a few places later in the thesis where it will be very convenient to make use of this feature.

3.2.5 Products and Coproducts

Being a $*$-autonomous category, $\mathcal{SP}roc$ is a model of multiplicative linear logic. It also has the structure needed to interpret the additives, namely finite products and coproducts. The binary coproduct functor $\oplus$ is defined on objects by

$$\begin{aligned} \Sigma_{A \oplus B} &\stackrel{\text{def}}{=} \Sigma_A + \Sigma_B \\ S_{A \oplus B} &\stackrel{\text{def}}{=} \{\mathsf{inl}^*(s) \mid s \in S_A\} \\ &\cup \quad \{\mathsf{inr}^*(s) \mid s \in S_B\}. \end{aligned}$$

If $p : A \to C$ and $q : B \to D$ then $p \oplus q : A \oplus B \to C \oplus D$ is defined by

$$\begin{aligned} p \oplus q &\stackrel{\text{def}}{=} p[(a, c) \mapsto (\mathsf{inl}(a), \mathsf{inl}(c))] \\ &\cup \quad q[(b, d) \mapsto (\mathsf{inr}(b), \mathsf{inr}(d))]. \end{aligned}$$

The insertions $\mathsf{inl}_{A,B} : A \to A \oplus B$ and $\mathsf{inr}_{A,B} : B \to A \oplus B$ are defined by

$$\begin{aligned} \mathsf{inl}_{A,B} &\stackrel{\text{def}}{=} \mathsf{id}_A[(a, a) \mapsto (a, \mathsf{inl}(a))] \\ \mathsf{inr}_{A,B} &\stackrel{\text{def}}{=} \mathsf{id}_B[(b, b) \mapsto (b, \mathsf{inr}(b))] \end{aligned}$$

and, for $p : A \to C$, $q : B \to C$, $[p,q] : A \oplus B \to C$ is

$$\begin{array}{rcl} [p,q] & \stackrel{\text{def}}{=} & p[(a,c) \mapsto (\mathsf{inl}(a),c)] \\ & \cup & q[(b,c) \mapsto (\mathsf{inr}(b),c)]. \end{array}$$

In these definitions, the operation $\cup$ on processes means union of the representations as sets, i.e. non-deterministic sum of synchronisation trees.

Proposition 3.3 The above definitions make $A \oplus B$ a coproduct of A and B.

Proof: Suppose $p : A \to C$ and $q : B \to C$. It is easy to check that $\mathsf{inl} \,;\, [p,q] = p$, because the $\mathsf{inl}(a)$ actions of inl can only match the $\mathsf{inl}(a)$ actions of $[p,q]$, and these come from p. Thus the result of the composition is the same as $\mathsf{id}_A \,;\, p$, i.e. p. Similarly $\mathsf{inr} \,;\, [p,q] = q$.

Now suppose that $h : A \oplus B \to C$ with $\mathsf{inl} \,;\, h = p$ and $\mathsf{inr} \,;\, h = q$. The first action of h must be either $(\mathsf{inl}(a),c)$ or $(\mathsf{inr}(b),c)$. In the first case, $h \xrightarrow{(\mathsf{inl}(a),c)} h'$ and because we know that $\mathsf{inl} \xrightarrow{(a,\mathsf{inl}(a))} \mathsf{inl}'$, it must also be true that $p \xrightarrow{(a,c)} p'$. Furthermore, the definition of composition means that $\mathsf{inl}' \,;\, h' = p'$. Now, because $S_{A \oplus B} = S_A + S_B$ and the first action of h was $(\mathsf{inl}(a),c)$, any subsequent behaviour of h (which means any behaviour of h') must involve actions in the A component rather than the B component. So composing h' with inl' does not restrict the behaviour of h', and we have $h' = p'[(x,z) \mapsto (\mathsf{inl}(x),z)]$. The same is true of the first step, and so h contains $p[(x,z) \mapsto (\mathsf{inl}(x),z)]$. Similarly, h contains $q[(y,z) \mapsto (\mathsf{inr}(y),z)]$. Since these are the only two possibilities, according to the two possibilities for the first action of h, it follows that $h = [p,q]$. □

The proof of uniqueness of the source tupling morphism relies crucially on the safety specification of $A \oplus B$. To see this, consider objects A, B and C with $\Sigma_A = \{a\}$, $\Sigma_B = \{b\}$ and $\Sigma_C = \{c\}$. Define morphisms $p : A \to C$ and $q : B \to C$ by

$$\begin{array}{rcl} p & \stackrel{\text{def}}{=} & (a,c) : \mathsf{nil} \\ q & \stackrel{\text{def}}{=} & (b,c) : \mathsf{nil} \end{array}$$

so that

$$[p,q] \;=\; (\mathsf{inl}(a),c) : \mathsf{nil} + (\mathsf{inr}(b),c) : \mathsf{nil}.$$

If objects have no safety specifications, there is a morphism $r : A \oplus B \to C$ defined by

$$r \stackrel{\text{def}}{=} [p,q] + (\mathsf{inl}(a),c) : (\mathsf{inr}(b),c) : \mathsf{nil}.$$

Then $\mathsf{inl} \,;\, r = p$ and $\mathsf{inr} \,;\, r = q$, violating uniqueness of $[p,q]$. This is the first point at which safety specifications have been necessary—everything up to now would have

worked just as well if objects consisted only of alphabets. But without safety specifications, there would only be a weak coproduct, which means that the source tupling morphism is not uniquely defined by the requirement that $\mathsf{inl} ; [f, g] = f$ and $\mathsf{inr} ; [f, g] = g$.

Since $\oplus$ is a coproduct, its dual is a product; because all objects of $\mathcal{SP}roc$ are self-dual, this means that $A \oplus B$ is itself also a product of A and B—so, in fact, a biproduct. There is also a zero object $\mathbf{0}$ which has $\Sigma_{\mathbf{0}} \stackrel{\text{def}}{=} \varnothing$ and $S_{\mathbf{0}} \stackrel{\text{def}}{=} \{\varepsilon\}$.

Proposition 3.4 The object $\mathbf{0}$ is initial and terminal in $\mathcal{SP}roc$.

Proof: The only safe trace for $\mathbf{0}$ is the empty trace, so a morphism $A \to \mathbf{0}$ cannot make any transitions and must be nil. Similarly for a morphism $\mathbf{0} \to A$. □

The binary biproducts and zero object just defined also give $\mathcal{SP}roc$ all finite biproducts. The definitions can be extended to countably infinite biproducts in the obvious way, by taking a countable disjoint union of sorts and safety specifications.

Proposition 3.5 $\mathcal{SP}roc$ has all countable biproducts.

From now on, any mention of biproducts in $\mathcal{SP}roc$ refers to the specified biproducts defined in this section.

When a category has biproducts and a zero object, it is possible to define a commutative monoid structure on each homset [Mac71]. If $p, q : A \to B$ then $p + q : A \to B$ is defined by

$$\begin{aligned} p + q &\stackrel{\text{def}}{=} A \stackrel{\Delta_A}{\to} A \oplus A \stackrel{[p,q]}{\to} B \\ &= A \stackrel{\langle p,q \rangle}{\to} B \oplus B \stackrel{\nabla_B}{\to} B \end{aligned}$$

where $\Delta_A \stackrel{\text{def}}{=} \langle \mathsf{id}_A, \mathsf{id}_A \rangle$ is the diagonal and $\nabla_B \stackrel{\text{def}}{=} [\mathsf{id}_B, \mathsf{id}_B]$ the codiagonal. The unit is defined by $\mathbf{0}_{A \to B} \stackrel{\text{def}}{=} A \to \mathbf{0} \to B$.

In $\mathcal{SP}roc$, this construction yields the non-deterministic sum of CCS (when strong bisimulation is taken as the notion of equivalence). The proof of Proposition 3.4 shows that the unique morphisms into and out of $\mathbf{0}$ are nil processes, and so $\mathbf{0}_A$ is also nil. To unravel the definition of $+$, consider the composition $\langle p, q \rangle ; \nabla_B$. Pairing creates a union of the behaviours of p and q, but with disjointly labelled copies of B. Composing with ∇_B removes the difference between the two copies. Hence in terms of the concrete set representations of processes, $p + q = p \cup q$. A choice can be made between p and q at the first step, but then the behaviour continues as behaviour of p or behaviour of q. This is precisely the natural representation in terms of synchronisation trees of the non-deterministic sum in CCS.

Because for each A the functors $A \otimes -$ and $- \otimes A$ have right adjoints, they preserve colimits, and in particular $\oplus$. Hence $\otimes$ distributes over countable biproducts, and $A \otimes \mathbf{0} \cong \mathbf{0}$. This extends to the commutative monoid operation $+$: for any morphisms $p : A \to B$ and $q, r : C \to D$, $p \otimes (q+r) = (p \otimes q)+(p \otimes r)$ and $(p+q) \otimes r = (p \otimes q)+(p \otimes r)$. Also $p \otimes \mathsf{nil} = \mathsf{nil} = \mathsf{nil} \otimes p$.

3.2.6 $\mathcal{SP}roc$ as a Linear Category

If $\mathcal{SP}roc$ is to be considered as a model of full classical linear logic, it must also have structure to interpret the exponentials ! and ?. It is sufficient to define ! and its properties; ? is then given by de Morgan duality. Since $\mathcal{SP}roc$ is self-dual, ? will be the same as !.

It turns out that $\mathcal{SP}roc$ models the exponentials in the very strong sense described in Section 2.3. So in principle, it is only necessary to define $\otimes^n_s$ and show that it has the required equaliser property. However, it is more useful to concretely define all the structure associated with !, because there is a certain amount of work involved in decoding the general construction.

The object $\otimes^n_s A$ is defined by

$$\begin{aligned} \Sigma_{\otimes^n_s A} &\stackrel{\text{def}}{=} \mathcal{M}_n(\Sigma_A) \\ S_{\otimes^n_s A} &\stackrel{\text{def}}{=} \{\varphi^*_n(s) \mid s \in S_{\otimes^n A}\} \end{aligned}$$

where $\mathcal{M}_n(X)$ is the set of multisets of size n on X, and $\varphi_n : \Sigma_{\otimes^n A} \to \Sigma_{\otimes^n_s A}$ is defined by

$$\begin{aligned} \varphi_0(*) &\stackrel{\text{def}}{=} \{\!|\}\!| \\ \varphi_n(a_1, \ldots, a_n) &\stackrel{\text{def}}{=} \{\!|a_1, \ldots, a_n|\!\} \quad n > 0. \end{aligned}$$

The equaliser morphism $e : \otimes^n_s A \to \otimes^n A$ is defined by

$$e \stackrel{\text{def}}{=} \mathsf{id}_{\otimes^n A}[(\tilde{a}, \tilde{a}) \mapsto (\varphi_n(\tilde{a}), \tilde{a})]$$

where $\tilde{a} = (a_1, \ldots, a_n)$.

Proposition 3.6 Denoting by p_σ the canonical automorphism of $\otimes^n A$ corresponding to the permutation σ, the morphism $e : \otimes^n_s A \to \otimes^n A$ is the equaliser of all the p_σ.

Proof: All the composites $e \,;\, p_\sigma$ are equal, because e maps any permutation of $(a_1, \ldots, a_n)$ on the right to the multiset $\{\!|a_1, \ldots, a_n|\!\}$ on the left. If $q : X \to \otimes^n A$ also has equal composites with all the p_σ, then a morphism $h : X \to \otimes^n_s A$ is required such that $h \,;\, e = q$. Now, $h \,;\, e = q$ implies $h \,;\, e \,;\, e^\perp = q \,;\, e^\perp$, and since $e \,;\, e^\perp = \mathsf{id}_{\otimes^n_s A}$ this means $h = q \,;\, e^\perp$. Concretely, this defines h by $h \stackrel{\text{def}}{=} q[(x, \tilde{a}) \mapsto (x, \varphi_n(\tilde{a}))]$; the argument also proves uniqueness of h. □

With this concrete definition of $\otimes_s^n$, the general constructions of Section 2.3 can be translated into $\mathcal{SP}roc$. For the functorial action of $\otimes_s^n$, if $g : A \to B$ then

$$\otimes_s^n g \stackrel{\text{def}}{=} (\otimes^n g)[(\tilde{a}, \tilde{b}) \mapsto (\varphi_n(\tilde{a}), \varphi_n(\tilde{b}))].$$

For the cocommutative comonoid structure

$$I \xleftarrow{\mathsf{weak}_A} !A \xrightarrow{\mathsf{contr}_A} !A \otimes !A$$

it is straightforward to define $\mathsf{weak}_A \stackrel{\text{def}}{=} \mathsf{id}_I[(*,*) \mapsto (*, \{\!|\}\!|)]$. The comultiplication contr_A is defined via morphisms $\mathsf{contr}_A^{r,m,n} : \otimes_s^r A \to (\otimes_s^m A) \otimes (\otimes_s^n A)$. If $r \neq m+n$ then $\mathsf{contr}_A^{r,m,n} \stackrel{\text{def}}{=} \mathsf{nil}$, and if $r = m+n$ then $\mathsf{contr}_A^{r,m,n}$ is defined by transition rules.

$$\frac{\mathsf{id}_{\otimes_s^r A} \xrightarrow{(\alpha,\alpha)} \mathsf{id}_{\otimes_s^r A'}}{\mathsf{contr}_A^{(r,m,n)} \xrightarrow{(\alpha,(\beta,\gamma))} \mathsf{contr}_{A'}^{(r,m,n)}} \; |\beta| = m, |\gamma| = n, \beta \cup \gamma = \alpha$$

For the comonad structure, the counit $\mathsf{der}_A : !A \to A$ is defined by

$$\mathsf{der}_A \stackrel{\text{def}}{=} \mathsf{id}_A[(a,a) \mapsto (\{\!|a|\!\}, a)].$$

Rather than defining the comultiplication, it is easier to define promotion directly (effectively using Manes' streamlined presentation of monads [Man76]). If $f : !A \to B$ then $f^! : !A \to !B$ is defined via morphisms $f_n : !A \to \otimes_s^n B$, where

$$\begin{aligned} f_n \stackrel{\text{def}}{=} \; & (\otimes_s^n f)[(\{\!|\{\!|a_{11}, \ldots, a_{1m_1}|\!\}, \ldots, \{\!|a_{n1}, \ldots, a_{nm_n}|\!\}|\!\}, \{\!|b_1, \ldots, b_n|\!\}) \mapsto \\ & (\{\!|a_{11}, \ldots, a_{nm_n}|\!\}, \{\!|b_1, \ldots, b_n|\!\})]. \end{aligned}$$

It follows from the general construction that this really defines the cofree cocommutative comonoid and a comonad; this can also be verified directly from the concrete definitions.

To understand how the operations on $!A$ work, it is useful to think of them as demand-driven. The morphism $\mathsf{contr}_A : !A \to !A \otimes !A$ has two "clients", the two copies of $!A$ on the right hand side of the arrow. Each client specifies the number of copies it requires, by operating in the appropriate component of the biproduct making up $!A$. The contr_A morphism adds the two numbers together and passes this requirement to the left. The morphisms $\mathsf{der}_A : !A \to A$ and $\mathsf{weak}_A : !A \to I$ specify one copy and zero copies respectively, by only operating in the corresponding component of $!A$.

3.2.7 Time

So far, none of the constructions in $\mathcal{SP}roc$ have made use of the fact that morphisms are *processes* with dynamic behaviour. Everything that has been discussed applies

equally well to the category of sets and relations. The next step is to justify the claim that $\mathcal{SP}roc$ looks like "relations extended in time" by defining some structure which allows the temporal aspects of the category to be manipulated.

The basic construction dealing with time is the unit delay functor $\circ$. It is defined on objects by

$$\begin{aligned} \Sigma_{\circ A} &\stackrel{\text{def}}{=} \{*\} + \Sigma_A \\ S_{\circ A} &\stackrel{\text{def}}{=} \{\varepsilon\} \cup \{*\sigma \mid \sigma \in S_A\}. \end{aligned}$$

It is notationally convenient to write $*$ instead of $\mathsf{inl}(*)$, assuming that $* \notin \Sigma_A$. Given $f : A \to B$, $\circ f : \circ A \to \circ B$ is defined by the single transition $\circ f \xrightarrow{(*,*)} f$.

It is straightforward to check that $\circ$ is indeed a functor. In fact it is a strict monoidal functor.

Proposition 3.7 There are isomorphisms $\mathsf{mon}_{A,B} : (\circ A) \otimes (\circ B) \to \circ(A \otimes B)$ (natural in A and B) and $\mathsf{monunit} : I \to \circ I$.

Proof: $\mathsf{monunit} : I \cong \circ I$ is defined by

$$\mathsf{monunit} \xrightarrow{(\bullet,*)} \mathsf{id}_I$$

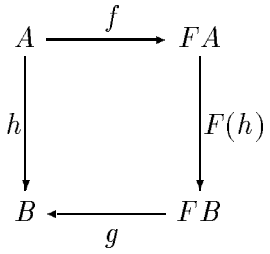
where $\Sigma_I = \{\bullet\}$. $\mathsf{mon}_{A,B} : (\circ A) \otimes (\circ B) \cong \circ(A \otimes B)$ is defined by

$$\mathsf{mon}_{A,B} \xrightarrow{((*,*),*)} \mathsf{id}_{A \otimes B}.$$

In both cases the inverse is obtained by considering the process as a morphism in the opposite direction. It is easy to check that these are isomorphisms and that mon is natural. □

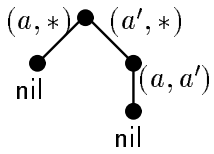
It will often be sufficient to consider $\circ$ not as a strict monoidal functor but just as a monoidal functor, which means just using $\mathsf{monunit}$ and mon, not their inverses. It is probably better to leave strictness out of an axiomatisation of $\circ$; this will be discussed later in this chapter, but one reason is that there are interaction categories in which $\circ$ is not strict.

A novel feature of $\circ$ is that it has the *unique fixed point property*, defined as follows. Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor. F has the unique fixed point property (UFPP) if for all $f : A \to FA$ and $g : FB \to B$ there is a unique $h : A \to B$ such that

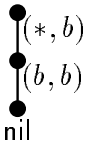


commutes.

It is worth spending a little time illustrating how the UFPP works, as it is a fundamental property of $\circ$ and one which will be important later. Suppose $f : A \to \circ A$ is the synchronisation tree



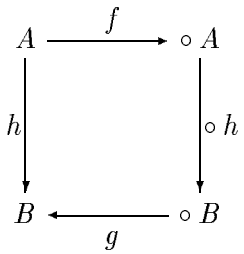
and $g : \circ B \to B$ is the synchronisation tree



so that the possible behaviours of f and g are described by the transitions

$$f \xrightarrow{(a,*)} \mathsf{nil} \qquad f_1 \xrightarrow{(a,a')} \mathsf{nil} \qquad g \xrightarrow{(*,b)} g_1 \qquad g_1 \xrightarrow{(b,b)} \mathsf{nil}.$$
$$f \xrightarrow{(a',*)} f_1$$

Now the aim is to define $h : A \to B$ so that the UFPP diagram commutes.



Using the definition of composition by transition rules, the transition $\circ h \xrightarrow{(*,*)} h$ and the desired equation $h = f\,;\circ h\,; g$, the following transitions can be deduced:

$$f\,;\circ h\,; g \xrightarrow{(a,b)} \mathsf{nil}\,; h\,; g_1 \qquad f\,;\circ h\,; g \xrightarrow{(a',b)} f_1\,; h\,; g_1.$$

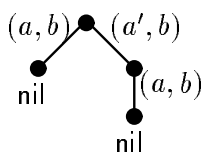
Hence

$$h \xrightarrow{(a,b)} \mathsf{nil} \qquad h \xrightarrow{(a',b)} f_1\,; h\,; g_1.$$

Using the definition of composition again, and these transitions, gives

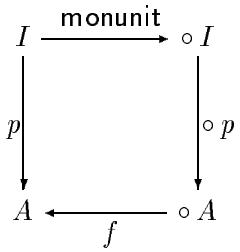
$$f_1\,; h\,; g_1 \xrightarrow{(a,b)} \mathsf{nil}\,; (f_1\,; h\,; g_1)\,; \mathsf{nil}$$

i.e. $f_1\,; h\,; g_1 \xrightarrow{(a,b)} \mathsf{nil}$. If h is the synchronisation tree



then the UFPP diagram commutes; furthermore, h was completely determined by the requirement that this diagram commute, and hence is the unique such morphism.

More interesting examples of the UFPP in use appear when f and g have infinite behaviours. In that case, the UFPP says something about the existence of processes satisfying guarded recursive definitions (guarded in the usual process calculus sense that recursive calls only occur under prefixes). Consider a simple guarded recursive definition, for example $p = a : p$ in SCCS notation, and suppose that its intended interpretation is as a definition of a process of type A where $\Sigma_A = \{a, b\}$ and S_A contains all traces over Σ_A. This means constructing a morphism $p : I \to A$ whose behaviour in A is a stream of a actions. To do this, let $f : \circ A \to A$ be the process defined by $f \stackrel{\text{def}}{=} \{((*, a), \mathsf{id}_A)\}$, and apply the UFPP to f and $\mathsf{monunit} : I \to \circ I$. Then there is a unique $p : I \to A$ such that



commutes. The behaviour of p is determined by the same feedback argument as before. At the first step $\mathsf{monunit}$ does $(\bullet, *)$, $\circ p$ does $(*, *)$ and f does $(*, a)$, the result being that p does $(\bullet, a)$. At the next step, $\circ p$ does the action performed by p at the first step, namely $(\bullet, a)$, and f can do (a, a). This gives a second step action for p of $(\bullet, a)$. Continuing in this way, the nth action of p is generated by the $(n-1)$th action of p being delayed by $\circ$ and copied by f, and in the end $p = (\bullet, a) : p$.

Proposition 3.8 $\circ$ has the UFPP.

Proof: Given $f : A \to \circ A$ and $g : \circ B \to B$, any $h : A \to B$ making the UFPP diagram commute satisfies a certain guarded recursive equation. Since guarded recursive equations over synchronisation trees have unique solutions [Mil89], h exists and is unique. □

Many other functors, constructed from $\circ$, also have the UFPP. The connection with guarded recursive definitions, hinted at above, shows which functors do have the UFPP. In general, any guarded recursive definition has some pattern of prefix actions appearing before the recursive call. For any such definition there is a functor with the same pattern of unit delays. For example, corresponding to the definition $P = a : b : P$ is the functor $\circ\circ$; corresponding to the definition $P = a : b : P + c : P$ is the functor $\circ\circ \oplus \circ$. All such functors have the UFPP. At the end of this chapter there is some discussion of how to be more formal about which functors should have the UFPP.

Abramsky [Abr94b] proves that if a functor F has the UFPP then an invariant object (i.e. X such that $FX \cong X$) must be unique if it exists. In $\mathcal{SP}roc$, I is the unique invariant object of $\circ$, but this need not happen in other interaction categories.

Apart from $\circ$ there are two other delay functors: the initial delay δ and the propagated delay Δ. These are the same as the operators used by Milner [Mil83, Mil89] to construct CCS from SCCS, and they can also be used to construct asynchronous processes in the synchronous framework of $\mathcal{SP}roc$.

The functors δ and Δ are defined on objects by

$$\begin{aligned}
\Sigma_{\delta A} &\stackrel{\text{def}}{=} \mathbf{1} + \Sigma_A \\
S_{\delta A} &\stackrel{\text{def}}{=} \{*^n\sigma \mid (n < \omega) \wedge (\sigma \in S_A)\} \\
\Sigma_{\Delta A} &\stackrel{\text{def}}{=} \mathbf{1} + \Sigma_A \\
S_{\Delta A} &\stackrel{\text{def}}{=} \{\varepsilon\} \cup \{a_1 *^{n_1} a_2 *^{n_2} a_3 \ldots \mid (n_i < \omega) \wedge (a_1 a_2 a_3 \ldots \in S_A)\}
\end{aligned}$$

and on morphisms by transition rules.

$$\frac{}{\delta f \xrightarrow{(*,*)} \delta f} \qquad \frac{f \xrightarrow{(a,b)} f'}{\delta f \xrightarrow{(a,b)} f'} \qquad \frac{f \xrightarrow{(a,b)} f'}{\Delta f \xrightarrow{(a,b)} \delta \Delta f'}$$

It is straightforward to check that δ and Δ really are functors, because a transition of $(\delta f);(\delta g)$ is either a transition of $f;g$ or is a delay step resulting from delays by both f and g. δ can be defined in terms of $\circ$, since

$$\delta A \cong A \oplus \circ A \oplus \circ^2 A \oplus \cdots \oplus I$$

but it does not seem possible to do the same for Δ. The observation that

$$\Delta(\circ A) \cong \circ \delta \Delta A$$

comes close and describes the operation of Δ, but it is not a definition since not every object is of the form $\circ A$.

The delay functors δ and Δ have the structure of monads. Consider first the monad structure of δ. The unit η_A is the same process as id_A, but with the type $A \to \delta A$ instead of $A \to A$. The multiplication $\mu_A : \delta\,\delta A \to \delta A$ converts both idle actions on the left into the single idle action on the right. Writing $\mu_A : \delta_1\,\delta_2 A \to \delta A$ allows the definition to be made by these rules.

$$\frac{}{\mu_A \xrightarrow{(*_1,*)} \mu_A} \qquad \frac{}{\mu_A \xrightarrow{(*_2,*)} \mu^1_A}$$

$$\frac{}{\mu^1_A \xrightarrow{(*_2,*)} \mu^1_A} \qquad \frac{\mathsf{id}_A \xrightarrow{(a,a)} P}{\mu^1_A \xrightarrow{(a,a)} P.}$$

A possible trace of μ_A is shown in the table.

$\delta_1\,\delta_2\,A$	$\delta\,A$
$*_1$	$*$
$*_1$	$*$
$*_2$	$*$
a	a
$\vdots$	$\vdots$

The monad structure of Δ is defined similarly, but acting on propagated delays instead of just initial delays.

Proposition 3.9 (δ, η, μ) and (Δ, η', μ') are monads.

Proof: This is a straightforward verification; the monad structure operates on sequences of $*$ actions in a way corresponding to the monoid structure of the natural numbers with addition. □

The combined delay functor $\delta\Delta$ is also a monad by virtue of the fact that there is a distributive law, i.e. a natural transformation $\mathsf{dist} : \Delta\delta \dot{\to} \delta\Delta$. Writing $*_\delta$ and $*_\Delta$ for the delay actions, $\mathsf{dist}_A : \Delta\delta\,A \to \delta\Delta\,A$ is defined by transition rules.

$$\frac{}{\mathsf{dist}_A \xrightarrow{(*_\delta, *_\delta)} \mathsf{dist}_A} \qquad \frac{}{\mathsf{dist}_A \xrightarrow{(*_\Delta, *_\delta)} \mathsf{dist}_A} \qquad \frac{\mathsf{id}_A \xrightarrow{\alpha} P}{\mathsf{dist}_A \xrightarrow{\alpha} P}$$

A typical trace is as shown.

$\Delta\delta\,A$	$\delta\Delta\,A$
$*_\delta$	$*_\delta$
$*_\Delta$	$*_\delta$
$\vdots$	$\vdots$
$*_\Delta$	$*_\delta$
$*_\delta$	$*_\delta$
$*_\Delta$	$*_\delta$
$\vdots$	$\vdots$
$*_\Delta$	$*_\delta$
$*_\delta$	$*_\delta$
$\vdots$	$\vdots$
a	a
$*_\Delta$	$*_\Delta$
$\vdots$	$\vdots$

The unit delay functor $\circ$ is not a monad. Proving this is an application of the UFPP.

Proposition 3.10 Let $\mathbb{C}$ be any category, (T, η, μ) a monad on $\mathbb{C}$, and suppose that T has the UFPP. Then $\mathbb{C}$ has a terminal object 1, and T is naturally isomorphic to the constant functor valued at 1.

Proof: For any objects A and B, applying the UFPP to the morphisms $\eta_A : A \to TA$ and $\mu_B : T^2B \to TB$ gives a unique morphism $f : A \to TB$ such that

$$\begin{array}{ccc} A & \xrightarrow{\eta_A} & TA \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle Tf} \\ TB & \xleftarrow[\mu_B]{} & T^2B \end{array}$$

commutes. But for any $g : A \to TB$, $\eta_A \,;\, Tg \,;\, \mu_B = g \,;\, \eta_{TB} \,;\, \mu_B = g$ by naturality and the monad laws, so the UFPP implies that there is a unique morphism from A to TB. Since this is true for any A and B, TB must be a terminal object as claimed. □

Corollary 3.11 $\circ$ is not a monad.

Proof: In $\mathcal{SP}roc$, there is a terminal object $\mathbf{0}$ with $\Sigma_{\mathbf{0}} = \varnothing$. But because $|\Sigma_{\circ A}| \geqslant 1$ and $* \in S_{\circ A}$ for any object A, $\circ A$ cannot be isomorphic to $\mathbf{0}$. □

3.3 Asynchrony via Kleisli Categories

The delay operators δ and Δ are functorial versions of the operators used to construct the asynchronous calculus CCS on top of the synchronous calculus SCCS. It would be very satisfying if the same construction could be carried out over $\mathcal{SP}roc$ to yield a category of asynchronous processes. Since δ, Δ and the combination $\delta\,\Delta$ are all monads, an obvious approach is to construct the Kleisli category of one of the monads.

Let $\mathbb{C}$ be a category and (T, η, μ) a monad on $\mathbb{C}$. The *Kleisli category* $\mathbb{C}_T$ is constructed as follows. The objects are objects TA of $\mathbb{C}$. A morphism f from A to B in $\mathbb{C}_T$ is a morphism $f : A \to TB$ in $\mathbb{C}$. Composition of $f : A \to TB$ and $g : B \to TC$ is defined by $f \,;\, Tg \,;\, \mu_C$. The identity on A is $\eta_A : A \to TA$.

Now consider the Kleisli category $\mathcal{SP}roc_\Delta$ of the monad Δ over $\mathcal{SP}roc$. The interpretation of a process of type A in this category should be a morphism $I \to A$, which means a morphism $I \to \Delta\, A$ in $\mathcal{SP}roc$. Such a morphism represents a process with a certain amount of asynchronous behaviour—it can delay in a pattern specified by Δ. Another possibility would be to use the monad $\delta\Delta$ to introduce processes which can delay at any point.

This idea seems simple enough, but applying it leads to several technical problems—and in fact, these problems seem to be insuperable. The asynchronous category ($\mathcal{SP}roc_\Delta$, for the sake of argument) should be an interaction category, so it certainly needs a $*$-autonomous structure. Jacobs [Jac94] describes a number of results about monads and comonads on symmetric monoidal closed categories, and discusses the possibility of lifting a monoidal structure on a category to one on a Kleisli category. Such a lifting of structure requires a *strong* monad, which means that there should be a natural transformation whose components are $\tau_{A,B} : A \otimes \Delta B \to \Delta(A \otimes B)$. In $\mathcal{SP}roc$, the intention was that $\tau_{A,B}$ should retime the behaviour in A so that the same pattern of delays occurs in both A and B. A typical trace of $\tau_{A,B}$ would look like this.

A	ΔB	$\Delta(A \otimes B)$
a_1	b_1	a_1 b_1
a_2	$*$	$*$
a_3	$*$	$*$
a_4	b_2	a_2 b_2
a_5	b_3	a_3 b_3
$\vdots$	$\vdots$	$\vdots$ $\vdots$

The first problem is that the $\tau_{A,B}$ are not components of a natural transformation. To see this, let $f : A \to C$ be defined by $f \stackrel{\text{def}}{=} (a, c) : \mathsf{nil}$ and consider a naturality square.

$$\begin{array}{ccc} A \otimes \Delta B & \xrightarrow{\tau_{A,B}} & \Delta(A \otimes B) \\ \Big\downarrow{\scriptstyle f \otimes \Delta\, \mathsf{id}_B} & & \Big\downarrow{\scriptstyle \Delta(f \otimes \mathsf{id}_B)} \\ C \otimes \Delta B & \xrightarrow[\tau_{C,B}]{} & \Delta(C \otimes B) \end{array}$$

Round the left and bottom of the square, $f \otimes \Delta\, \mathsf{id}_B$ can do $((a, b), (c, b))$ and become nil, for some $b \in \Sigma_B$, and $\tau_{C,B}$ can do $((c, b), (c, b))$. So the composite does $((a, b), (c, b))$ at the first step and then must become nil. Going round the top and right of the square, the same actions can happen at the first step, but at the second step there is a difference. $\tau_{A,B}$ can do $((a, *), *)$ for any $a \in \Sigma_A$, and of course $\Delta(f \otimes \mathsf{id}_B)$ can do $(*, *)$. So the composite has a possible action $((a, *), *)$ at the second step, and hence is not equal to the other direction round the square.

Even apart from the fact that τ is not natural, the possible definitions of a monoidal structure on $\mathcal{SP}roc_\Delta$ make liberal use of the strength, and this tends to have the effect of bringing processes into synchronisation with each other when they should really be independently timed.

Rather than going into more details, it is sufficient to say that although several attempts were made to construct an asynchronous category using these ideas, nothing

came of them; fortunately, Abramsky discovered a direct construction of an asynchronous category which could be used instead. This construction is described in the next section.

3.4 The Category $\mathcal{ASP}roc$

3.4.1 Asynchronous Processes

$\mathcal{ASP}roc$, the category of asynchronous processes, is based on equivalence classes of synchronisation trees modulo observation equivalence. The silent action for the observation equivalence is specified as part of the sort of a process. If P is a process whose silent action is τ, then (as synchronisation trees) $P \approx \tau.P$ and this means that as processes, $P = \tau.P$. Equivalently, any process P has the transition $P \xrightarrow{\tau} P$. This is a very useful way to think about observation equivalence classes, and emphasises the point that a process can always delay by doing silent actions—this is what asynchrony means in this thesis. In all discussion of $\mathcal{ASP}roc$, observation equivalence is denoted by $P = Q$. Note that this is different from the standard CCS notation in which equality means observation congruence.

In most of the discussion of $\mathcal{ASP}roc$ in this chapter, the distinction between synchronisation trees and observation equivalence classes is blurred slightly. Any mention of processes refers to equivalence classes; but when operations such as composition are defined by transition rules, these rules define synchronisation trees and it is then necessary to prove that the operation are well-defined on processes.

3.4.2 $\mathcal{ASP}roc$ as a Category

The following sequence of definitions is similar to that in [Abr94a], although that paper takes a simpler view and treats a process as a set of traces. Furthermore, the explicit appearance of a τ action in each alphabet is a novel feature of this presentation as compared to Abramsky's original (unpublished) presentation of the full version of $\mathcal{ASP}roc$.

An object of $\mathcal{ASP}roc$ is a triple $A = (\Sigma_A, \tau_A, S_A)$, in which Σ_A is a set of actions, $\tau_A \in \Sigma_A$ is the *silent action*, $S_A \subseteq^{nepref} \mathsf{ObAct}(A)^*$ is a safety specification, and $\mathsf{ObAct}(A) \stackrel{\text{def}}{=} \Sigma_A - \{\tau_A\}$ is the set of observable actions of A.

A *process* with *sort* Σ and *silent action* $\tau \in \Sigma$ is an observation equivalence class of synchronisation trees with label set Σ.

A *process* P *of type* A, written $P : A$, is a process P with sort Σ_A and silent action τ_A

such that $\mathsf{obtraces}(P) \subseteq S_A$, where (again using a coinductive definition)

$$\begin{aligned}
\mathsf{allobtraces}(P) &\stackrel{\text{def}}{=} \{\varepsilon\} \cup \{a\sigma \mid P \stackrel{a}{\Longrightarrow} Q, \sigma \in \mathsf{allobtraces}(Q)\} \\
\mathsf{obtraces}(P) &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{allobtraces}(P) \mid \sigma \text{ is finite}\} \\
\mathsf{infobtraces}(P) &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{allobtraces}(P) \mid \sigma \text{ is infinite}\}
\end{aligned}$$

As before the morphisms are defined via the object part of the $*$-autonomous structure. Given objects A and B, the object $A \otimes B$ has

$$\begin{aligned}
\Sigma_{A\otimes B} &\stackrel{\text{def}}{=} \Sigma_A \times \Sigma_B \\
\tau_{A\otimes B} &\stackrel{\text{def}}{=} (\tau_A, \tau_B) \\
S_{A\otimes B} &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{ObAct}(\Sigma_{A\otimes B})^* \mid \sigma\restriction A \in S_A, \sigma\restriction B \in S_B\}
\end{aligned}$$

where, for $\alpha \in \mathsf{ObAct}(\Sigma_{A\otimes B})$,

$$\alpha\restriction A \stackrel{\text{def}}{=} \begin{cases} \mathsf{fst}(\alpha) & \text{if } \mathsf{fst}(\alpha) \neq \tau_A \\ \varepsilon & \text{otherwise} \end{cases}$$

and for $\sigma \in \mathsf{ObAct}(\Sigma_{A\otimes B})^*$, $\sigma\restriction A$ is obtained by concatenating the individual $\alpha\restriction A$. The projection $\sigma\restriction B$ is defined similarly.

The duality is trivial on objects: $A^\perp \stackrel{\text{def}}{=} A$.

A morphism $p : A \to B$ of *ASProc* is a process p such that $p : A \multimap B$.

Composition is defined as in *SProc*. If $p : A \to B$ and $q : B \to C$, then the composite $p\,;q : A \to C$ is defined by labelled transitions.

$$\frac{p \xrightarrow{(a,b)} p' \qquad q \xrightarrow{(b,c)} q'}{p\,;q \xrightarrow{(a,c)} p'\,;q'}$$

In this rule, any of the actions a, b, c can be τ.

It is necessary to prove that composition is well-defined on observation equivalence classes.

Proposition 3.12 If $p, q : A \to B$ with $p \approx q$, and $r : B \to C$, then $p\,;r \approx q\,;r$.

Proof: Suppose $p\,;r \xrightarrow{(a,c)} p'\,;r'$. Then by the definition of composition, $p \xrightarrow{(a,b)} p'$ and $r \xrightarrow{(b,c)} r'$ for some $b \in \Sigma_B$. Because $p \approx q$, $q \stackrel{\widehat{(a,b)}}{\Longrightarrow} q'$ with $q' \approx p'$. Now, r is able to match any (τ_A, τ_B) actions of q with (τ_B, τ_C) actions of its own, and each one results in a (τ_A, τ_C) action of $q\,;r$. Thus $q\,;r \stackrel{\widehat{(a,c)}}{\Longrightarrow} q'\,;r'$, and "inductively" $q'\,;r' \approx p'\,;r'$. Similarly the other way round, and hence $p\,;r \approx q\,;r$. □

Proposition 3.13 If $p : A \to B$ and $q : B \to C$ then the following are derived transition rules for $p\,;q$.

$$\frac{p \xrightarrow{(a,\tau_B)} p'}{p\,;q \xrightarrow{(a,\tau_C)} p'\,;q} \qquad \frac{q \xrightarrow{(\tau_B,c)} q'}{p\,;q \xrightarrow{(\tau_A,c)} p'\,;q}$$

Proof: This follows from the fact that any process can make a τ transition without changing state. □

As in $\mathcal{SP}roc$, it is straightforward to prove that if $f : A \to B$ and $g : B \to C$, then $f\,;g$ satisfies the safety specification necessary to be a morphism $A \to C$.

Although $\mathcal{ASP}roc$ is a category of asynchronous processes, the identity morphisms are still *synchronous* buffers. As a candidate identity morphism, a synchronous buffer seems likely to work, given the definition of composition; of course, once it has been shown to be an identity, no other choice is possible.

The identity morphism $\mathsf{id}_A : A \to A$ is defined as in $\mathcal{SP}roc$: $\mathsf{id}_A \stackrel{\text{def}}{=} \mathsf{id}\restriction S_{A\multimap A}$ where the process id with sort Σ_A is defined by

$$\frac{a \in \Sigma_A}{\mathsf{id} \xrightarrow{(a,a)} \mathsf{id}.}$$

Just as in $\mathcal{SP}roc$, if P is a process with sort Σ and $S \subseteq^{nepref} \Sigma^*$ then the process $P\restriction S$, also with sort Σ, is defined by the transition rule

$$\frac{P \xrightarrow{a} Q \qquad a \in S}{P\restriction S \xrightarrow{a} Q\restriction(S/a).}$$

3.4.3 $\mathcal{ASP}roc$ as a $*$-Autonomous Category

If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ and $p^\perp : C^\perp \to A^\perp$ are defined by transition rules.

$$\frac{p \xrightarrow{(a,c)} p' \qquad q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((a,b),(c,d))} p' \otimes q'} \qquad \frac{p \xrightarrow{(a,c)} p'}{p^\perp \xrightarrow{(c,a)} p'^\perp}$$

As with composition there are two derived transitions for $p \otimes q$, which make clear the sense in which $\otimes$ is an asynchronous parallel composition.

Proposition 3.14 If $p : A \to C$ and $q : B \to D$ then the following are derived transition rules for $p \otimes q$.

$$\frac{p \xrightarrow{(a,c)} p'}{p \otimes q \xrightarrow{((a,\tau_B),(c,\tau_D))} p' \otimes q} \qquad \frac{q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((\tau_A,b),(\tau_C,d))} p \otimes q'}$$

The tensor unit I is defined by

$$\Sigma_I \stackrel{\text{def}}{=} \{\tau_I\} \qquad\qquad S_I \stackrel{\text{def}}{=} \{\varepsilon\}.$$

The morphisms expressing the symmetric monoidal closed structure are defined as in $\mathcal{SP}roc$, by combining identities.

Proposition 3.15 $\mathcal{ASP}roc$ is a compact closed category.

3.4.4 Products and Coproducts

The functor $\oplus$ can be defined in $\mathcal{ASP}roc$ but it only gives weak biproducts. However, by making a choice of pairing operation it is possible to carry out some of the constructions which were done for biproducts in $\mathcal{SP}roc$. On objects, $\oplus$ is defined by

$$\begin{array}{rcl} \Sigma_{A\oplus B} & \stackrel{\text{def}}{=} & \mathsf{ObAct}(A) + \mathsf{ObAct}(B) + \{l, r\} + \tau_{A\oplus B} \\ S_{A\oplus B} & \stackrel{\text{def}}{=} & \{l\mathsf{inl}^*(s) \mid s \in S_A\} \\ & \cup & \{r\mathsf{inr}^*(s) \mid s \in S_B\}. \end{array}$$

If $p : A \to C$ and $q : B \to D$ then $p \oplus q : A \oplus B \to C \oplus D$ is defined by

$$\begin{array}{rcl} p \oplus q & \stackrel{\text{def}}{=} & \{((l,l), p[(a,c) \mapsto (\mathsf{inl}(a), \mathsf{inl}(c))])\} \\ & \cup & \{((r,r), q[(b,d) \mapsto (\mathsf{inr}(b), \mathsf{inr}(d))])\} \end{array}$$

where inl and inr are considered to act on silent actions by $\mathsf{inl}(\tau_A) \stackrel{\text{def}}{=} \mathsf{inr}(\tau_B) \stackrel{\text{def}}{=} \tau_{A\oplus B}$. The coproduct insertions $\mathsf{inl}_{A,B} : A \to A \oplus B$, $\mathsf{inr}_{A,B} : B \to A \oplus B$ are defined by

$$\begin{array}{rcl} \mathsf{inl}_{A,B} & \stackrel{\text{def}}{=} & \{((\tau_A, l), \mathsf{id}_A[(a,a) \mapsto (a, \mathsf{inl}(a))])\} \\ \mathsf{inr}_{A,B} & \stackrel{\text{def}}{=} & \{((\tau_B, r), \mathsf{id}_B[(b,b) \mapsto (b, \mathsf{inr}(b))])\} \end{array}$$

and, for $p : A \to C$ and $q : B \to C$, $[p, q] : A \oplus B \to C$ is defined by

$$\begin{array}{rcl} [p,q] & \stackrel{\text{def}}{=} & \{((l, \tau_C), p[(a,c) \mapsto (\mathsf{inl}(a), c)])\} \\ & \cup & \{((r, \tau_C), q[(b,c) \mapsto (\mathsf{inr}(b), c)])\}. \end{array}$$

Proposition 3.16 $\mathsf{inl}_{A,B} \,;\, [p,q] = p$ and $\mathsf{inr}_{A,B} \,;\, [p,q] = q$.

Proof:

$$\begin{array}{rcll} \mathsf{inl}_{A,B} \,;\, [p,q] & = & (\tau_A, \tau_C).p & \\ & = & p & \text{by observation equivalence.} \end{array}$$

The other case is similar. □

To see why $A \oplus B$ is not a coproduct of A and B, consider the morphisms $p : A \to C$ and $q : B \to C$ defined by

$$p \stackrel{\text{def}}{=} (\tau_A, c).\mathsf{nil} \qquad q \stackrel{\text{def}}{=} (\tau_B, c).\mathsf{nil}$$

so that

$$[p, q] = (l, \tau_C).(\tau_{A \oplus B}, c).\mathsf{nil} \cup (r, \tau_C).(\tau_{A \oplus B}, c).\mathsf{nil}.$$

Now define $h : A \oplus B \to C$ by

$$h \stackrel{\text{def}}{=} (l, c).\mathsf{nil} \cup (r, c).\mathsf{nil}.$$

It is easy to check that $\mathsf{inl}_{A,B}\,;h = (\tau_A, c).\mathsf{nil} = p$ and $\mathsf{inr}_{A,B}\,;h = q$. But since $h \neq [p, q]$, the uniqueness part of the definition of a coproduct has been violated.

For $p, q : A \to B$, $p + q$ is defined as in $\mathcal{SP}roc$, as if $\oplus$ were a genuine biproduct. With this definition, $+$ turns out to be the internal choice $\sqcap$ of CSP; strictly, since $+$ is not associative, it is an *implementation* of $\sqcap$. In terms of CCS, the operation which has been defined is $\tau.p + \tau.q$. This is no surprise, because the genuine CCS $+$ is not well-defined on observation equivalence classes. One way to get CCS $+$ might be to construct a variation of $\mathcal{ASP}roc$ in which processes are observation *congruence* classes of synchronisation trees, an interesting possibility which has not yet been investigated.

3.4.5 $\mathcal{ASP}roc$ as a Linear Category

Defining exponentials in $\mathcal{ASP}roc$ is more difficult than in $\mathcal{SP}roc$ and in fact, at the time of writing, a satisfactory definition has not been worked out. There are two obvious approaches to try. One is to use the symmetric algebra construction as in $\mathcal{SP}roc$, but although it is possible to define the equaliser $e : \otimes^n_s A \to \otimes^n A$ the lack of genuine biproducts means that the rest of the construction cannot be carried through. The other approach is to follow the example of the construction of exponentials in categories of games [AJM94]; this construction is based on viewing $!A$ as the infinite tensor power $\otimes^\omega A$, quotiented by an equivalence relation to remove the distinction between different components. This has not yet been investigated.

3.4.6 Time

In $\mathcal{ASP}roc$, the delay monads δ and Δ are less meaningful than in $\mathcal{SP}roc$, since delay is built into all the definitions. But the unit delay functor $\circ$ is still important. On objects it is defined by

$$\Sigma_{\circ A} \stackrel{\text{def}}{=} \{*\} + \Sigma_A$$

$$\begin{aligned} \tau_{\circ A} &\stackrel{\text{def}}{=} \tau_A \\ S_{\circ A} &\stackrel{\text{def}}{=} \{\varepsilon\} \cup \{*\sigma \mid \sigma \in S_A\}. \end{aligned}$$

If $f : A \to B$ then $\circ f : \circ A \to \circ B$ is defined by the transition $\circ f \xrightarrow{(*,*)} f$.

Proposition 3.17 $\circ$ is a functor, and has the UFPP.

Proof: As in $\mathcal{SP}roc$. □

In $\mathcal{ASP}roc$ as in $\mathcal{SP}roc$, $\circ$ has a unique invariant object, but this is J defined by

$$\Sigma_J \stackrel{\text{def}}{=} \{*, \tau_J\} \qquad\qquad S_J \stackrel{\text{def}}{=} \{*^n \mid n < \omega\}$$

rather than I. Hence $\circ$ is not a *strict* monoidal functor on $\mathcal{ASP}roc$. The other part of strictness, namely that $\circ(A \otimes B) \cong (\circ A) \otimes (\circ B)$, also fails. There is an inclusion of behaviours: for any trace in $\circ(A \otimes B)$ there is a corresponding trace in $(\circ A) \otimes (\circ B)$ with the initial $*$ replaced by $(*,*)$. However, there are traces in $(\circ A) \otimes (\circ B)$ which do not begin with $(*,*)$, and these traces have no analogue in $\circ(A \otimes B)$.

3.5 An Alternative Presentation of $\mathcal{ASP}roc$

The presentation of $\mathcal{ASP}roc$ used in the preceding sections is designed to highlight the crucial difference from $\mathcal{SP}roc$, i.e. the use of observation equivalence. Abramsky's original presentation does not mention τ actions explicitly, but uses the empty set as the silent action in every type. To ensure that the action $\varnothing$ is available in each type, the actions in a type A are taken to be subsets of Σ_A. The definition of $\otimes$ is also different: $\Sigma_{A\otimes B}$ is $\Sigma_A + \Sigma_B$ instead of $\Sigma_A \times \Sigma_B$. This means that the actions available in $A \otimes B$ include $\{a\}$, $\{b\}$ and $\{a, b\}$ for $a \in \Sigma_A$ and $b \in \Sigma_B$. These actions correspond, in the presentation used here, to (a, b), (a, τ_B) and (τ_A, b) respectively.

One advantage of the original presentation is that it makes the connection with CCS clearer. The transition rules for $\otimes$ become

$$\frac{p \xrightarrow{m} p'}{p \otimes q \xrightarrow{m} p' \otimes q} \qquad \frac{q \xrightarrow{n} q'}{p \otimes q \xrightarrow{n} p \otimes q'}$$

$$\frac{p \xrightarrow{m} p' \quad q \xrightarrow{n} q'}{p \otimes q \xrightarrow{m \cup n} p' \otimes q'}$$

if the inl and inr tags are omitted. The first two rules are essentially the same as the rules for CCS parallel composition; only the third rule, allowing simultaneous actions,

does not correspond to anything in CCS. In fact, the presence of the third rule means that this is the parallel composition of ASCCS [Mil83].

The presentation with explicit τ actions was originally formulated in the hope that it might facilitate the transfer of the construction of ! from $\mathcal{SP}roc$ to $\mathcal{ASP}roc$. As it turned out, it didn't make it any easier to define ! in $\mathcal{ASP}roc$, but it did make the relationship between $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ clearer. In the original presentation of $\mathcal{ASP}roc$, the use of sets of labels as actions seemed slightly mysterious, but as soon as τ actions are explicitly introduced it is easy to see that observation equivalence is the only real difference between the two categories. Furthermore, the new presentation allows composition and $\otimes$ to be defined by exactly the same transition rules as in $\mathcal{SP}roc$.

3.6 Axioms for Interaction Categories

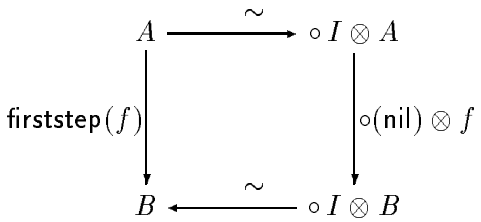
The question "What is an interaction category?" has not yet been answered, although both $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ have been claimed as examples. The theory of interaction categories should include an abstract definition. Ideally, an interaction category should at least be a linear category, which is a well-understood concept. However, taking the linear structure as the starting point means that $\mathcal{ASP}roc$ must be excluded, because no definition of ! has been established. On the other hand, this is the only piece of structure which is missing from $\mathcal{ASP}roc$, and one can certainly hope that in the future it will be filled in. In any case, the most interesting part of the axiomatisation question concerns the delay operators and in particular the unit delay functor. For this reason it is useful to keep $\mathcal{ASP}roc$ in consideration, as it is the fundamental example of an asynchronous interaction category, and interesting issues arise from the relationship between asynchrony and delays. This will become clear later in this section.

The most important property of the unit delay functor seems to be the UFPP; more generally, the UFPP is an essential property not just of $\circ$ but of other *guarded* functors. An obvious approach to axiomatisation is to define guardedness in terms of some easily specified categorical structure, and then require that every guarded functor does indeed have the UFPP.

3.6.1 Guardedness in $\mathcal{SP}roc$

The intuitive idea of guardedness is that if G is a guarded functor then for any morphisms f and g, Gf and Gg do the same thing at the first step. This idea can be captured quite neatly in $\mathcal{SP}roc$, as follows; in fact, it is convenient to work in the most general possible categorical structure.

Let $\mathbb{C}$ be a symmetric monoidal category with finite biproducts. Suppose that $\mathbb{C}$ has a strict monoidal endofunctor $\circ$ such that $\circ(\mathsf{nil}) \neq \mathsf{nil}$, where nil is the unit of the commutative monoid structure induced on a homset by the biproducts. For each A and B, define a function $\mathsf{firststep}$ on $\mathbb{C}(A, B)$ by

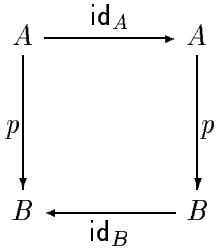


Now say that an endofunctor G on $\mathcal{C}$ is *guarded* if for any A, B and $f, g : A \to B$, $\mathsf{firststep}(G(f)) = \mathsf{firststep}(G(g))$.

In $\mathcal{SP}roc$, the operation $\mathsf{firststep}$ gives the possible actions which a process may do initially. Any nil process is an empty synchronisation tree, and so $\circ(\mathsf{nil})$ does only a single action $(*, *)$. The synchronous nature of $\otimes$ means that $\circ(\mathsf{nil}) \otimes f$ is also a tree of depth one. Finally, in the definition of $\mathsf{firststep}(f)$ the $*$ actions are removed by canonical isomorphisms to give a process of depth one whose possible actions are the possible first actions of f. The argument that $\circ$ on $\mathcal{SP}roc$ has the UFPP also shows that $\mathcal{SP}roc$ satisfies the *guarded functor axiom*:

- every guarded functor has the UFPP.

The requirement that $\circ(\mathsf{nil}) \neq \mathsf{nil}$ may seem curious, but it is necessary to avoid degeneracy. If $\circ(\mathsf{nil}) = \mathsf{nil}$, then $\mathsf{firststep}(f) = \mathsf{nil}$ for all f, and so in this case all functors would be guarded. Imposing the guarded functor axiom would then force all functors to have the UFPP, in particular the identity functor. Then for any A and B there would be a unique morphism $p : A \to B$ making the diagram



commute, i.e. all homsets would be singletons. There is an analogy here with fixed points of continuous functions on the ω-cpo $(A^\omega, \sqsubseteq)$ of traces over a set A with the prefix order. If f is continuous and $f(\varepsilon) \sqsupset \varepsilon$ then f has a unique fixed point, but demanding that every continuous function had a unique fixed point would force A to consist of just a bottom element.

There are some natural closure conditions on the class of guarded functors, which are satisfied in $\mathcal{SP}roc$ and can also be shown to hold in the abstract situation.

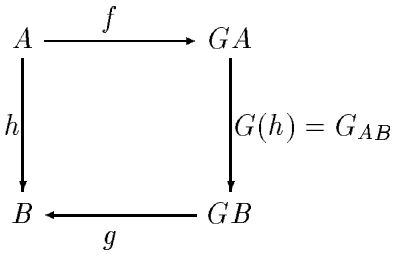
Proposition 3.18

1. Any constant functor is guarded.
2. $\circ$ is guarded.
3. If G is guarded and F is any functor then GF is guarded.
4. A countable biproduct of guarded functors is guarded.

Proof:

1. If K is the constant functor at some object A, then $Kf = \mathsf{id}_A$ for every morphism f. So for any f and g, $\mathsf{firststep}(Kf) = \mathsf{firststep}(Kg) = \mathsf{firststep}(\mathsf{id}_A)$.
2. For any f, $\mathsf{firststep}(\circ(f)) = \mathsf{firststep}(\circ(\mathsf{nil}))$.
3. If G is guarded then for any morphisms f and g, $\mathsf{firststep}(G(f)) = \mathsf{firststep}(G(g))$. Hence $\mathsf{firststep}(G(F(f))) = \mathsf{firststep}(G(F(g)))$.
4. Follows from the fact that $\mathsf{firststep}(f \oplus g) = \mathsf{firststep}(f) \oplus \mathsf{firststep}(g)$, which in turn follows from distributivity of $\otimes$ over $\oplus$. □

The guarded functor axiom is not enough to ensure that a category really consists of processes, i.e. entities with temporal behaviour. Consider the category $\mathcal{R}el$ of sets and relations, and take $\circ$ to be the constant functor at $I = \{*\}$. Then $\mathsf{firststep}(f) = f$ for all f, and so a functor G is guarded iff for all f and g, $G(f) = G(g)$. This means that for every A and B there is $G_{AB} : GA \to GB$ such that for all $f : A \to B$, $G(f) = G_{AB}$. The UFPP for G becomes: for any $f : A \to GA$ and $g : GB \to B$ there is a unique $h : A \to B$ such that



commutes. This is satisfied because the square defines h uniquely by composition. So $\mathcal{R}el$ satisfies the guarded functor axiom. It can be excluded from the class of interaction categories by imposing a simple extra condition which is satisfied by $\mathcal{SP}roc$ but not by $\mathcal{R}el$: that $\circ$ be faithful.

Chapter 6 requires the functor $!\oplus_{n>0} \circ^n$ on $\mathcal{SP}roc$ to have the UFPP. It is convenient to prove this now.

Lemma 3.19 For any object A of $\mathcal{SP}roc$, $\otimes^n_s(\circ A) \cong \circ \otimes^n_s A$.

Proof: We have

$$\begin{aligned}
\Sigma_{\otimes^n_s(\circ A)} &= \mathcal{M}_n(\Sigma_A + \{*\}) \\
S_{\otimes^n_s(\circ A)} &= \{\varphi^*_n(s) \mid s \in S_{\otimes^n(\circ A)}\} \\
\Sigma_{\circ \otimes^n_s A} &= \mathcal{M}_n(\Sigma_A) + \{*\} \\
S_{\circ \otimes^n_s A} &= \{*\varphi^*_n(s) \mid s \in S_{\otimes^n A}\}
\end{aligned}$$

and so

$$\begin{aligned}
S_{\otimes^n_s(\circ A)} &= \{\varphi^*_n(*,\ldots,*)\varphi^*_n(t) \mid t \in S_{\otimes^n A}\} \\
&= \{\{\!|*,\ldots,*|\!\}\varphi^*_n(t) \mid t \in S_{\otimes^n A}\}.
\end{aligned}$$

Thus an isomorphism can be constructed by prefixing a translation between $*$ and $\{\!|*,\ldots,*|\!\}$ onto $\mathsf{id}_{\otimes^n_s A}$. □

Proposition 3.20 The functor $!\oplus_{n>0} \circ^n$ on $\mathcal{SP}roc$ has the UFPP.

Proof: For $n > 0$,

$$\begin{aligned}
!(\circ^n A) &\stackrel{\text{def}}{=} \oplus_{m \geqslant 0} \otimes^m_s (\circ^n A) \\
&\cong \oplus_{m \geqslant 0} (\circ^n \otimes^m_s A).
\end{aligned}$$

Now,

$$\begin{aligned}
!\oplus_{n>0} \circ^n A &= !\&_{n>0} \circ^n A \\
&\cong \otimes_{n>0}(!\circ^n A) \\
&\cong \otimes_{n>0}(\oplus_{m \geqslant 0}(\circ^n \otimes^m_s A)) \\
&= (\oplus_{m \geqslant 0}(\circ \otimes^m_s A)) \otimes (\oplus_{m \geqslant 0}(\circ^2 \otimes^m_s A)) \otimes \cdots
\end{aligned}$$

which, when multiplied out, results in a coproduct of terms of the form $\circ(\cdots)$. Thus the functor $!\oplus_{n>0} \circ^n$ can be expressed as a coproduct of guarded functors, and is itself guarded. All guarded functors on $\mathcal{SP}roc$ have the UFPP, hence the result. □

3.6.2 Guardedness in $\mathcal{ASP}roc$

The definition of $\mathsf{firststep}$ in the previous section relies crucially on the fact that $\mathsf{nil} \otimes f = \mathsf{nil}$ for any morphism f. In $\mathcal{ASP}roc$ this is not true, so that definition of $\mathsf{firststep}$ cannot

be used. It is possible to define $\mathsf{firststep}$ in $\mathcal{ASP}roc$, using the concrete representation of processes as synchronisation trees, but at the time of writing there does not seem to be any suitable abstract definition of $\mathsf{firststep}$ which also works for $\mathcal{ASP}roc$. An alternative is to use a syntactic description of guarded functors, based on the closure conditions of the previous section.

Let $\mathbb{C}$ be a symmetric monoidal category with finite (possibly weak) biproducts and a monoidal endofunctor $\circ$, such that for every morphism f, $\circ(f) \neq \mathsf{nil}$. Define $\mathcal{GF}$ to be the smallest collection of endofunctors satisfying:

- $K \in \mathcal{GF}$ if K is a constant functor.
- $\circ \in \mathcal{GF}$.
- For $G, G' \in \mathcal{GF}$, $G \oplus G' \in \mathcal{GF}$.
- For $G \in \mathcal{GF}$ and any endofunctor F, $GF \in \mathcal{GF}$.

The guarded functor axiom becomes

- Every functor in $\mathcal{GF}$ has the UFPP.

Now it can be shown that $\mathcal{ASP}roc$ satisfies this axiom. For $f : A \to B$, define $\mathsf{firststep}(f) : A \to B$ by the following transition rule.

$$\frac{f \stackrel{\alpha}{\Longrightarrow} f'}{\mathsf{firststep}(f) \stackrel{\alpha}{\longrightarrow} \mathsf{nil}}\ \alpha \neq \tau_{A \multimap B}$$

So $\mathsf{firststep}(f)$ either consists of the possible first observable actions of f, or is nil if $f = \mathsf{nil}$.

Lemma 3.21 If $G \in \mathcal{GF}$ then for any $f, g : A \to B$, $\mathsf{firststep}(G(f)) = \mathsf{firststep}(G(g))$.

Proof: By induction on the definition of $\mathcal{GF}$. Suppose $f, g : A \to B$. If K is the constant functor at A, then $\mathsf{firststep}(f) = \mathsf{firststep}(g) = \mathsf{firststep}(\mathsf{id}_A)$. For the unit delay functor, $\mathsf{firststep}(\circ f) = \mathsf{firststep}(\circ g) = (*,*).\mathsf{nil}$. If G has the desired property and F is any endofunctor, then $\mathsf{firststep}(GF(f)) = \mathsf{firststep}(GF(g))$ and so GF also has the property. If G and G' are any endofunctors, then

$$\begin{aligned} \mathsf{firststep}(G(f) \oplus G'(f)) &= \mathsf{firststep}(G(g) \oplus G'(g)) \\ &= (l,l).\mathsf{nil} + (r,r).\mathsf{nil}. \end{aligned}$$

□

Lemma 3.22 If G is an endofunctor on $\mathcal{ASP}roc$ such that for any $f, g : A \to B$, $\mathsf{firststep}(G(f)) = \mathsf{firststep}(G(g))$, then G has the UFPP.

Proof: By the same argument that $\circ$ has the UFPP. □

These lemmas establish

Proposition 3.23 $\mathcal{ASP}roc$ satisfies the guarded functor axiom.

Actually, in $\mathcal{ASP}roc$ the class of guarded functors satisfies slightly stronger closure conditions: in the clause for $\oplus$, G and G' could be any endofunctors, not just elements of $\mathcal{GF}$. This can be seen from the proof of Lemma 3.21.

3.6.3 Guardedness in Other Categories

In Chapter 5 another interaction category is defined: $\mathcal{SP}roc_D$, the category of deadlock-free processes. It is based on $\mathcal{SP}roc$, but the formulation of the guarded functor axiom in terms of the structure of $\mathcal{SP}roc$ is not appropriate for it. Without going into too many details at this stage, $\mathcal{SP}roc_D$ has distinct products and coproducts, and although each homset has the $+$ operation, there are no nil morphisms. This suggests a new formulation of the guarded functor axiom, along the lines of the one which worked for $\mathcal{ASP}roc$, but without assuming biproducts in the definition of $\mathcal{GF}$. $\mathcal{GF}$ should be the smallest collection of endofunctors satisfying:

- $K \in \mathcal{GF}$ if K is a constant functor.
- $\circ \in \mathcal{GF}$.
- For $G, G' \in \mathcal{GF}$, $G \oplus G' \in \mathcal{GF}$ and $G \,\&\, G' \in \mathcal{GF}$.
- For $G \in \mathcal{GF}$ and any endofunctor F, $GF \in \mathcal{GF}$.

3.6.4 Synchronous and Asynchronous Categories

The definition of $\mathsf{firststep}$ in terms of abstract categorical properties requires the monoidal functor $\circ$ to be strict. The fact that $\circ$ is strict in $\mathcal{SP}roc$ but not in $\mathcal{ASP}roc$ is connected with the distinction between synchronous and asynchronous processes—it is the synchronous nature of $\otimes$ in $\mathcal{SP}roc$ which allows mon to be inverted. Once $\mathsf{firststep}$ has been used to define guardedness, proving that the collection of guarded functors is closed under $\oplus$ requires $\oplus$ to be a biproduct rather than a weak biproduct. More generally, if products and coproducts are distinct, $\otimes$ preserves coproducts and so the collection of guarded functors is closed under $\oplus$ (but not $\&$).

There seem to be properties which hold only for synchronous interaction categories, but none which hold only for asynchronous interaction categories. This means that rather than defining two varieties of category, the approach to an axiomatisation should be to define interaction categories and then pick out the synchronous ones. In a synchronous interaction category, other characterisations of certain structure become possible: the guarded functors can be defined via firststep, and if biproducts exist they can be used to define the + operation.

3.6.5 Other Forms of the UFPP

This chapter has discussed the UFPP in a simple form, but it is possible to state more general versions. One which will be useful in Chapter 6 is the multiple UFPP. A functor F has the *multiple unique fixed point property* if for any $n \geqslant 1$, objects $A_1, \ldots, A_n, B_1, \ldots, B_n$ and morphisms $f_i : A_i \to FA_i$, $g_i : FB_i \to B_i$, there is a unique collection of morphisms $h_i : A_i \to B_i$ such that this diagram commutes.

$$\begin{array}{ccc} A_1 \otimes \cdots \otimes A_n & \xrightarrow{f_1 \otimes \cdots \otimes f_n} & FA_1 \otimes \cdots \otimes FA_n \\ \Big\downarrow{\scriptstyle h_1 \otimes \cdots \otimes h_n} & & \Big\downarrow{\scriptstyle (Fh_1) \otimes \cdots \otimes (Fh_n)} \\ B_1 \otimes \cdots \otimes B_n & \xleftarrow[g_1 \otimes \cdots \otimes g_n]{} & FB_1 \otimes \cdots \otimes FB_n \end{array}$$

In both $\mathcal{SP}roc$ and $\mathcal{ASP}roc$, guarded functors have the multiple UFPP. This can be proved using a similar argument to that of the proof that $\circ$ has the UFPP.

3.6.6 Axioms

In the light of the observations made in the last few sections, the following definition seems reasonable. An interaction category should be a $*$-autonomous category (not necessarily compact closed; in $\mathcal{SP}roc_D$, $\otimes$ and $\parr$ are distinct). It should have finite products and coproducts, possibly only weak; these do not have to coincide. There should be a commutative operation + on each homset; in some categories this may have a unit nil. There should be a monoidal endofunctor $\circ$ such that for all morphisms f, $\circ f \neq$ nil (when nil exists), allowing a class of guarded endofunctors to be defined as in the previous section. Finally, the category should satisfy some form of the guarded functor axiom.

A synchronous interaction category should be defined as one in which the functor $\circ$ is strict, and there are products and coproducts. If there are biproducts, then the + operation specified as part of the interaction category structure should coincide with

the one definable from the biproducts. Also, the inductively-defined class of guarded functors should be the same as the class arising from the definition of guardedness in terms of firststep.

3.7 Discussion

The two basic examples of interaction categories, $\mathcal{SP}roc$ and $\mathcal{ASP}roc$, have been defined in this chapter. The definition of $\mathcal{SP}roc$ closely follows Abramsky's presentation [Abr93b, Abr94b], but $\mathcal{ASP}roc$ is presented in an alternative style which emphasises the rôle of observation equivalence as the key difference between $\mathcal{ASP}roc$ and $\mathcal{SP}roc$.

It does not seem possible to construct a category of asynchronous processes from $\mathcal{SP}roc$ by categorical means, but there is a connection between asynchrony and synchrony in the form of a functor which embeds $\mathcal{ASP}roc$ into $\mathcal{SP}roc$ and preserves the monoidal structure.

There are not yet very many different examples of interaction categories, which makes it difficult to be sure that any proposed axioms are sufficiently general. Furthermore, $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ are both rather degenerate in the sense that there is no distinction between the objects A and $A^{\perp}$. Nevertheless, some axioms for interaction categories have been suggested in this chapter. These axioms do not rely on the degeneracy of $\mathcal{SP}roc$ and $\mathcal{ASP}roc$; this should maximise the chance that they will be more widely applicable, and indeed when the less degenerate category $\mathcal{SP}roc_D$ of synchronous deadlock-free processes is defined in Chapter 5, it will turn out to satisfy them. The proposed axioms specify $*$-autonomous structure in order to capture the static, configurational aspects of interaction categories; the axioms addressing the dynamic aspects hinge on the requirement that a certain class of guarded functors should have the unique fixed point property. In general this class is defined inductively, but there is also a proposed definition of a synchronous interaction category which has slightly more structure. In a synchronous category, it becomes possible to define guardedness abstractly; the definition of the class of guarded functors then becomes a theorem describing its closure properties.

4

Synchronous Dataflow

4.1 Introduction

Dataflow is a model of computation in which a program is expressed as a network of computing agents which communicate with each other by sending data tokens along fixed links. It is natural to view the nodes of the network as independent processes, which means that dataflow is one possible way of organising concurrent computation. The theme of this chapter is the use of $\mathcal{SP}roc$ to describe the construction and operation of dataflow networks. In fact, since the existence of a global clock is a fundamental assumption of $\mathcal{SP}roc$, dataflow computation is also assumed to be synchronous. This is not an arbitrary or artificial assumption, as a number of synchronous dataflow languages (for example LUSTRE [HCRP91] and SIGNAL [GGBM91]) have been used successfully for applications such as digital signal processing and real-time control.

The main results of this chapter relate to the semantics of synchronous dataflow at a high level of abstraction, but they are also applied to the specific language LUSTRE. Section 4.2 introduces LUSTRE, and also serves as an illustration of the operation of dataflow networks. That section also reviews the standard Kahn semantics of dataflow [Kah74], in which nodes are assumed to compute continuous functions of their input streams, and its application to LUSTRE. Section 4.3 shows how the structure of a compact closed category provides all the operations necessary for the construction of dataflow networks, and in Section 4.4 this result is applied to define a semantics of LUSTRE in the particular compact closed category $\mathcal{SP}roc$.

Having described the Kahn semantics of LUSTRE and also a semantics in $\mathcal{SP}roc$, the aim is to establish that they agree with each other in the sense of predicting the same behaviour for a given network. In Section 4.5, a version of the Kahn semantics is defined for networks whose nodes compute functions which are not only continuous but synchronous; a synchronous function on streams is one which always produces additional output when additional data arrives simultaneously at all of its inputs. Such networks are given a semantics in $\mathcal{SP}roc$ in Section 4.6. This allows a comparison to be made between the two semantics at a very general level, and this is done in Section 4.7. Agreement is established, subject to some very natural restrictions on the

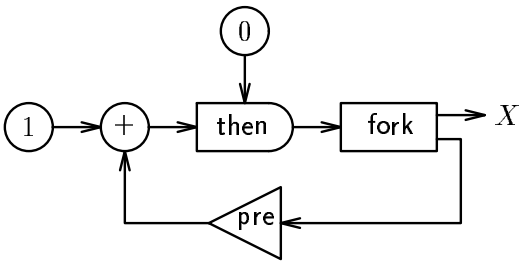


Figure 4.1: A Network in LUSTRE

formation of feedback loops. Finally, in Section 4.8 this result is applied to LUSTRE.

4.2 *The Language* LUSTRE

The network shown in Figure 4.1 can serve as an introduction both to the general nature of dataflow computation and to some of the specific features of LUSTRE. Data tokens, which in this case are natural numbers, flow in the direction of the arrows. The nodes 1 and 0 produce streams of 1s and 0s respectively; they are the sources of data which drive the whole network. The fork node is a signal-splitter: each of its two outputs carries the same sequence of tokens as its input. The + node, as expected, adds together the numbers appearing at its inputs, and outputs the result. The different shapes of the nodes in Figure 4.1 have no special significance; they are just a LUSTRE convention which allows different operators to be easily identified in a diagram.

The synchronous nature of LUSTRE has an important effect on the way data moves in the network. At every tick of the global clock each node receives tokens from all of its inputs and simultaneously produces an output token. This answers any questions about what happens if, for example, the + node receives one input before the other: the situation simply doesn't arise.

The node then (the name then is an alternative to the name -> which is the official LUSTRE syntax) works by copying one of its input tokens to the output at the first step, and thereafter copying tokens from the other input to the output. If the sequence of tokens appearing at the top input of a then node is

$$(x_0, x_1, x_2, \ldots)$$

and the sequence appearing at the left input is

$$(y_0, y_1, y_2, \ldots)$$

then the output sequence is

$$(x_0, y_1, y_2, \ldots).$$

In the example network, the top input of **then** is connected to the 0 node, so the first output of **then** is 0 and subsequent outputs are copied from the output of the + node. Thus although the 0 node continues to produce tokens, none but the first are ever used.

The final node in the example, **pre**, is a single-step delay. At the first step, it outputs the token $\perp$ which is an undefined value present in every data type. Subsequently the output at each step is the token which was input at the previous step. Introducing the $\perp$ token means that the definition of addition must be extended to cover inputs of $\perp$. The appropriate modification is to specify that for all x, $x+\perp=\perp$.

The behaviour of the example network is determined by these definitions. At the first step, the **pre** node outputs $\perp$. The + node receives this $\perp$, together with a 1 from the 1 node, and outputs $\perp$. This $\perp$ is ignored by the **then** node, which outputs the 0 it receives on its other input. The 0 token output by **then** is copied by the **fork** node, one copy being emitted as the first output from the network and the other copy being received by the **pre** node as input. At this first step, it is irrelevant which token is received by **pre**; it outputs $\perp$ regardless. This is what allows the above calculation of the network's output to get off the ground.

At the second step, **pre** outputs the token which it received at the first step, namely 0. This means that the output of + is 1, as another 1 token is emitted by the 1 node. The output of + is passed straight through **then**, and copied by **fork**. This results in an output of 1 from the network, and an input of 1 to the **pre** node. At every subsequent step the calculation is essentially the same as at the second; the output from the previous step comes out of **pre**, is added to 1, and sent to **fork**. The result is that the network outputs the sequence 01234... consisting of the natural numbers in order.

So far, LUSTRE has been described as a dataflow language, and the above analysis of the example program is based on thinking about a picture of a network. But there is also a concrete syntax for LUSTRE, in which data sequences are defined by recursive expressions constructed from the operators. A LUSTRE program takes a declarative form, as a collection of such definitions. The program corresponding to the natural numbers network is

$$X = 0 \textsf{ then } (\textsf{pre}(X) + 1)$$

or, to use the official syntax,

$$X = 0 \textsf{ -> } (\textsf{pre}(X) + 1).$$

In such programs, the **fork** node does not appear; it is just a way of graphically representing multiple occurrences of variables. For the rest of this chapter, it will be most convenient to continue with the graphical view.

One important feature of LUSTRE is missing from the example. According to the synchrony hypothesis, data tokens are always produced and consumed in time with the ticks of the global clock; but what happens if a node does not produce output at the same rate at which it consumes input? There really has to be a way of representing delays or gaps in sequences of data. The full story is that LUSTRE does not work with sequences of tokens but with data *streams*, which contain timing information as well as tokens. A *stream* consists of a sequence of data values together with a *clock* which indicates when the data values actually appear. As a first approximation, the clock can be thought of as a sequence of boolean values, one for each instant of global time, in which t means that a data value appears at the corresponding time and f means that there is a gap. Thus if an integer stream has data values 12345... and clock *tfftftf*..., the data actually appears as in the table.

Global time	0	1	2	3	4	5	6	7	...
Clock	t	f	f	t	t	f	t	f	...
Data	1			2	3		4		...

In fact, the situation is slightly more subtle than this. A clock is not a boolean *sequence* but a boolean *stream*, and so has a clock of its own. Clocks may be nested in this way to any finite depth, until ultimately the global clock (also known as the *basic clock*) appears.

The simplest LUSTRE operators are the *data operators*. These act purely on sequences of values, and do not affect clocks. They are just the sequence extensions of functions on data values. Without being specific about exactly which data types exist, the data operators include arithmetic functions on integers, boolean operations, conditionals, and other common functions.

Apart from the data operators, there are four *sequence operators*. The first two, **then** and **pre**, have already been described, and the clocks of their output streams are the same as the clocks of their input streams. In contrast, the final two operators produce outputs whose clocks differ from the clocks of their inputs. If E is a stream and B is a boolean stream, E **when** B produces a stream whose clock is B, by discarding the values of E which appear when B is false. The table shows the effect of **when** and also **current**, which removes one level of clock from a stream. The clock of **current**(X) is the clock of the clock of X; in the table, Y has the same clock as B. In terms of clocks, **current** inverts the effect of **when**, but it cannot recover the data values which **when** discards.

E	e_0	e_1	e_2	e_3	e_4	e_5	...
B	t	f	f	t	f	t	...
$X = E$ **when** B	$x_0 = e_0$			$x_1 = e_3$		$x_2 = e_5$	...
$Y =$ **current**(X)	e_0	e_0	e_0	e_3	e_3	e_5	...

The streams to which a data operator are applied must all have the same clock; similarly, the two input streams of a **then** node must have the same clock. In addition, **current** can only be applied to a stream whose clock is not the basic clock. Clocks other than the basic clock are introduced by the **when** operator, which converts a boolean sequence into a clock. Thus equality of clocks is not statically decidable, and so the definition of LUSTRE uses an approximation to clock equality based on syntactic identity of the identifiers representing clocks.

4.2.1 The Semantics of LUSTRE

There is a very elegant and simple semantics of dataflow due to Kahn [Kah74], which is based on the assumption that the output sequences of each node are continuous functions (with respect to the prefix order) of the input sequences. For simplicity, suppose that all data items come from some set A, so that continuity relates to the ω-cpo $(A^\omega, \sqsubseteq)$. It is straightforward to extend the analysis to allow different data types. A node with m inputs and n outputs is a tuple $\langle f_1, \ldots, f_n \rangle$ with each $f_i : (A^\omega)^m \to A^\omega$ a continuous function. A network is constructed by making connections between outputs and inputs of nodes. An output can only be connected to a single input, and each input to a single output; an explicit node $\mathsf{fork} = \langle \mathsf{id}, \mathsf{id} \rangle$ is used to split signals where necessary. The behaviour of a compound network, i.e. the sequences appearing at every point as a result of some given input sequences, is given by the least solution (constructed as the least fixpoint of an appropriate continuous function) of the simultaneous equations obtained from the functional relationships between the sequences. Since **fix** is itself a continuous function, the network can be encapsulated as a node in its own right.

This style of semantics can be applied to LUSTRE, but the presence of clocks means that some extra analysis is needed. Because clocks cannot always be statically determined, a clock calculation phase is an essential part of the compilation procedure [HCRP91] and also of the semantics. The purpose of this phase is to determine whether or not streams which should have the same clock actually do. If they do not, the program is deemed incorrect. One approach to the semantics of LUSTRE is to assume that a correct program is given, flatten all the streams by inserting delay tokens (representing the absence of data) whenever the clock is false, and then use the Kahn semantics of the resulting network. This is the approach taken in this chapter, and the resulting semantics of LUSTRE is compared with the *SProc* semantics. Jensen [Jen94] has recently taken another approach: his semantics is again based on Kahn's, but clock calculation is also represented by a process of taking least fixed points. The result is a semantics of both correct and incorrect programs, which offers the prospect of a semantic characterisation of clock consistency. It remains to be seen exactly how this semantics is related to the version of the Kahn semantics used in this chapter, and to

the $\mathcal{SP}roc$ semantics.

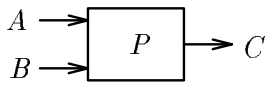
4.3 Dataflow in a Compact Closed Category

The structure of a compact closed category of processes supports the construction of dataflow networks, in a very general way. This is the basis for the definition of a semantics of LUSTRE in the particular compact closed category $\mathcal{SP}roc$.

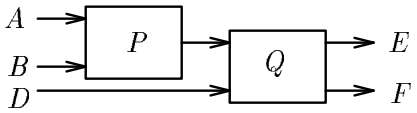
A general dataflow network is constructed from several *nodes*, each of which has a number of inputs and outputs, and which can be linked together by connecting outputs to inputs. A node works by receiving data tokens on its inputs and emitting tokens from its outputs. The following facts provide an informal inductive definition of networks. Note that they distinguish between connections which involve feedback (the formation of loops) and those which don't; this distinction is quite important when discussing semantics.

- A single node is a network.
- A network can be regarded as a node.
- Connecting two networks by plugging some outputs of one into some inputs of the other produces a network.
- Placing two networks side by side without connection produces a network.
- Adding a feedback loop from an output of a network to an input of the network produces a network.

Suppose that for each data type A, there is an object (also called A) in the category which is suitable for modelling that type. Exactly what this means depends on the actual category; when the model in $\mathcal{SP}roc$ is defined in Section 4.6, the object A will have a sort containing the values in the type A. A node



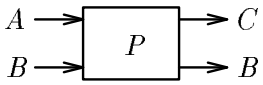
is modelled by a morphism $P : A \otimes B \to C$. Another node Q can be connected to P to form a simple network.



The node Q is modelled by a morphism $Q : C \otimes D \to E \otimes F$, and the morphism modelling the network is

$$(P \otimes \mathsf{id}_D)\,;Q : A \otimes B \otimes D \to E \otimes F.$$

This shows categorical composition in use as the operation corresponding to plugging an output into an input. Now consider the formation of a cyclic connection, or feedback loop, in a network. The general situation is that a network has already been constructed, and can be modelled by a morphism $P : A \otimes B \to C \otimes B$.



The loop is to be formed by connecting the B output to the B input, resulting in a process $\overline{P}$. In order to do this, it is essential that the category be compact closed rather than simply $*$-autonomous. This can best be illustrated by not making use of the isomorphism between $\otimes$ and $\parr$ until absolutely necessary.

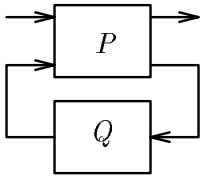
Starting with the morphism P, monoidal closure can be used to form $\Lambda(P) : A \to (B \multimap (C \otimes B))$. The following calculation corresponds to forming the cycle, and defines the morphism $\overline{P}$.

$$\begin{array}{ccccccc}
A & \xrightarrow{\Lambda(P)} & B \multimap (C \otimes B) & \xrightarrow{\sim} & B^\perp \parr (C \otimes B) & \xrightarrow[*]{\sim} & B^\perp \otimes (C \otimes B) \\
\downarrow{\scriptstyle \overline{P}} & & & & & & \downarrow{\scriptstyle \sim} \\
C & \xleftarrow{\mathsf{Ap}_{B,C}} & (B \multimap C) \otimes B & \xleftarrow{\sim} & (B^\perp \parr C) \otimes B & \xleftarrow[*]{\sim} & B^\perp \otimes (C \otimes B)
\end{array}$$

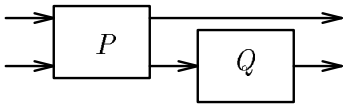
Here, the arrows marked by $*$ are the isomorphisms between $\otimes$ and $\parr$. This calculation can also be carried out by replacing one instance of the $\otimes - \parr$ isomorphism with the isomorphism, present in any compact closed category, between $\perp$ and I.

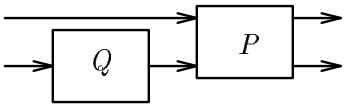
$$\begin{array}{ccccccc}
A & \xrightarrow{\Lambda(P)} & B \multimap (C \otimes B) & \xrightarrow{\sim} & B^\perp \parr (C \otimes B) & \xrightarrow{\quad\sim\quad} & B^\perp \otimes (C \otimes B) \\
\downarrow{\scriptstyle \overline{P}} & & & & & & \downarrow{\scriptstyle \sim} \\
C & \xleftarrow{\sim} & I \otimes C & \xleftarrow{\sim} & \perp \otimes I & \xleftarrow{\mathsf{Ap}_{B,\perp} \otimes \mathsf{id}_C} & ((B \multimap \perp) \otimes B) \otimes C
\end{array}$$

In general, even if the method of performing this calculation is fixed, there are still many ways of constructing the same network. For example,

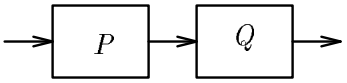


can be constructed by attaching a feedback loop to either of these networks:

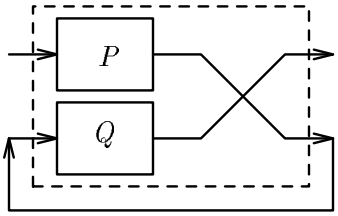




Even a straightforward connection between two nodes



can be viewed as a feedback by first placing the nodes in parallel to form a network, then making the cyclic connection.



The coherence theorem for compact closed categories [KL80] guarantees that however a network is constructed from certain components, the result is the same morphism.

Some additional justification is needed for the use of Ap to form cycles; it produces morphisms of the correct types, but this does not necessarily guarantee the correct behaviour. In *SProc*, the definition of Ap forces the two ports being connected to have the same behaviour. More general, but heuristic, arguments are that in the acyclic case, using Ap results in plain composition and is thus doing the desired job of making a connection between two ports; and that Ap should be dual to an identity morphism, which suggests that it should simply transfer information from one place to another. A rigorous argument is the proof of agreement between the *SProc* and Kahn semantics of synchronous dataflow, to appear later in this chapter.

Note that compact closure is essential for the calculation involving a cyclic connection—this approach would not work if types could not be considered to be formed just with $\otimes$ rather than with combinations of $\otimes$ and $\parr$.

4.4 Lustre *in SProc*

The next step is to use the scheme described in the previous section to define a semantics of Lustre in the particular category *SProc*. The specific inputs to the general construction are suitable objects of *SProc* to model the data types of Lustre, and morphisms which capture the behaviour of the Lustre operators. If A is a datatype (really, just a set) such as integer or boolean, there is an *SProc* object A whose sort consists of the given data values, and whose safety specification allows all traces over the

sort. LUSTRE requires an element $\perp$ in each datatype, and so for each corresponding $\mathcal{SP}roc$ object A, $\perp \in \Sigma_A$.

The combined delay monad $\delta\Delta$ is used to represent nested clocks. A stream with n levels of nested clock can be flattened by inserting $*_i$ at each point of delay arising from the ith clock. The resulting sequence is a trace in the $\mathcal{SP}roc$ type $(\delta\Delta)^n A$, if $*_i$ is the delay action belonging to the ith iteration of $\delta\Delta$. An output from a LUSTRE node, of datatype A and with n clock levels, becomes an output of type $(\delta\Delta)^n A$ in $\mathcal{SP}roc$. This type can be conveniently abbreviated to $A^{(n)}$.

Considering types as constraints on application of operators to arguments, the comments in the previous section about arguments being required to have suitable clocks suggest that in the $\mathcal{SP}roc$ model a type should incorporate a clock. An $\mathcal{SP}roc$ type could be defined with the alphabet of $(\delta\Delta)^n A$ and a safety specification which allowed only data sequences with some particular clock. However, this would mean that types could not be statically assigned to LUSTRE operators or programs. It is only possible to use types which embody some approximation to clocks—in this case, the depth of nesting. This allows the $\mathcal{SP}roc$ representation of current to have a type which forces its argument to have a non-basic clock, by requiring that it has at least one level of clock nesting.

The processes modelling the various operators can now be defined. These processes assume that their input streams not only have the same depth of clock nesting, but also genuinely have the same clock. First, the data operators. If $f : A_1 \times \cdots \times A_m \to B$ is a function, the stream extension $f^\omega : A_1^\omega \times \cdots \times A_m^\omega \to B^\omega$ is a data operator. The $\mathcal{SP}roc$ process $f : A_1 \otimes \cdots \otimes A_m \to B$ is defined by

$$\frac{a_i \in \Sigma_{A_i}}{f^\omega \xrightarrow{(a_1,\ldots,a_m,f(a_1,\ldots,a_m))} f^\omega}$$

and then the process $f^{(n)} \stackrel{\text{def}}{=} (\delta\Delta)^n f^\omega : A^{(n)} \to B^{(n)}$ can take account of n levels of clock. Thus for each data operator there is a whole series of processes, indexed by the depth of clock nesting. All inputs to $f^{(n)}$ have the same amount of clock nesting; since the language stipulates that all inputs to a data operator must actually have the same clock, this does not impose an unwanted constraint on network construction. The signal copier fork of type A is a data operator, formed from the function $\mathsf{copy} : A \to A \times A$ defined by $\mathsf{copy}(x) \stackrel{\text{def}}{=} (x, x)$.

The sequence operators also come in different versions, indexed by the clock depth of the input. Because the input and output of a pre node have the same clock, it can be modelled by the process $\mathsf{pre}^{(n)} \stackrel{\text{def}}{=} (\delta\Delta)^n \mathsf{pre}$ where $\mathsf{pre} : A \to A$ is a process which

works on inputs having the basic clock. The definition of pre is

$$\frac{a \in \Sigma_A}{\mathsf{pre} \xrightarrow{(a,\bot)} \mathsf{pre}_a} \qquad \frac{a,b \in \Sigma_A}{\mathsf{pre}_a \xrightarrow{(b,a)} \mathsf{pre}_b.}$$

At the first step, any input action is accompanied by an output of $\bot$; this is the meaning of the first transition rule. Subsequently the second rule, defining the auxiliary process pre_a, takes over. The parameter of the auxiliary process functions as a memory of the previous input; according to the second transition rule, it is updated at each step.

Similarly, a then node whose inputs and output have n clock levels is modelled by the process $\mathsf{then}^{(n)} \stackrel{\text{def}}{=} (\delta\Delta)^n\mathsf{then}$, where $\mathsf{then} : A \otimes A \to A$ is defined by

$$\frac{a,b \in \Sigma_A}{\mathsf{then} \xrightarrow{(a,b,a)} \mathsf{then}_1} \qquad \frac{a,b \in \Sigma_A}{\mathsf{then}_1 \xrightarrow{(a,b,b)} \mathsf{then}_1.}$$

At the first step, the first input is copied to the output; subsequently, the auxiliary process then_1 takes over and repeatedly copies the second input to the output.

Because when and $\mathsf{current}$ nodes change the clocks of their inputs, the corresponding processes have to be defined directly for each clock depth. The definition of the process $\mathsf{when}^{(n)} : A^{(n)} \otimes \mathbb{B}^{(n)} \to A^{(n+1)}$, where $\mathbb{B}$ is a boolean datatype, has two cases according to whether the second input is t or f.

$$\frac{a \in \Sigma_A}{\mathsf{when}^{(n)} \xrightarrow{(a,t,a)} \mathsf{when}^{(n)}} \qquad \frac{a \in \Sigma_A}{\mathsf{when}^{(n)} \xrightarrow{(a,f,*_{n+1})} \mathsf{when}^{(n)}.}$$

A $\mathsf{current}$ node whose input has n clock levels (where $n \geqslant 1$) is modelled by the process $\mathsf{current}^{(n)} : A^{(n)} \to A^{(n-1)}$. A parameter is used to record the last non-delay token received.

$$\frac{a \in \Sigma_{A^{(n-1)}}}{\mathsf{current}^{(n)} \xrightarrow{(a,a)} \mathsf{current}_a^{(n)}} \qquad \frac{}{\mathsf{current}^{(n)} \xrightarrow{(*_n,\bot)} \mathsf{current}^{(n)}}$$

$$\frac{b \in \Sigma_{A^{(n-1)}}}{\mathsf{current}_a^{(n)} \xrightarrow{(b,b)} \mathsf{current}_b^{(n)}} \qquad \frac{}{\mathsf{current}_a^{(n)} \xrightarrow{(*_n,a)} \mathsf{current}_a^{(n)}.}$$

This completes the definition of the semantics of Lustre in *SProc*. The next few sections take a more abstract view of dataflow; Lustre will reappear in Section 4.8.

4.5 The Kahn Semantics of Synchronous Dataflow

In this section, the Kahn semantics is adapted to synchronous dataflow networks, and in the next section a general semantics of synchronous dataflow in *SProc* is defined.

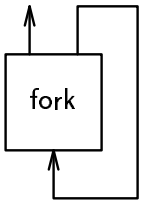


Figure 4.2: A Small Dataflow Network

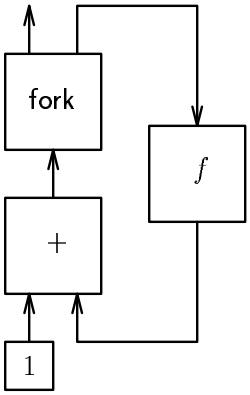


Figure 4.3: A Non-trivial Network

These two semantics are compared in Section 4.7, but before setting up the necessary machinery it is interesting to have a look at the main difference between them.

The least fixpoint approach of the Kahn semantics always gives a unique behaviour for a network, even if that behaviour consists only of empty sequences. For example, the network in Figure 4.2 outputs the empty sequence because nothing in the loop spontaneously produces data. Formally, the output x is defined by the equation $x = x$, and the least solution of this equation is $x = \varepsilon$. A network with feedback can only produce non-trivial output if it involves a node which produces output data without receiving any input. The simplest example is a node $\langle f \rangle$ with function $f : A^\omega \to A^\omega$ defined by $f(\sigma) = a\sigma$ for some fixed a. In the network of Figure 4.3, the node 1 produces the sequence $111\ldots$ and the function f is defined by $f(\sigma) = 0\sigma$. The output x is defined by $x = 111\ldots + f(x)$, and the least solution of this equation (i.e. the least fixed point of $\lambda x.111\ldots + 0x$) is $x = 1234\ldots$. This is closely related to the LUSTRE network of Figure 4.1. As already suggested, it is feedback which can cause differences between the $\mathcal{SP}roc$ semantics of dataflow and the Kahn semantics. In fact, as will become clear, the examples in Figures 4.2 and 4.3 illustrate the two cases at the heart of this issue.

Kahn's semantics does not assume a synchronous or asynchronous view of concurrency. Its treatment of dataflow nodes as continuous stream functions abstracts away from

considerations of timing. In *SProc*, on the other hand, processes are fundamentally synchronous and any asynchrony has to be explicitly represented by delays. However, the intended application of an *SProc* semantics of dataflow is to *synchronous* languages, and it is possible to take advantage of this fact to make the definition and analysis of the *SProc* model much simpler. A synchronous dataflow node has the property that tokens arrive simultaneously at all its inputs, and this causes tokens to be produced simultaneously at all its outputs. Considering each output sequence to be a continuous function of the input sequences, each such function has the property that if all its inputs increase then so does its output. A convenient way to state this is to say that the output sequence is always at least as long as the input sequences. So, a continuous function $f : (A^\omega)^n \to A^\omega$ is *synchronous* if

$$(\forall i.\mathsf{length}(x_i) \geqslant m) \Rightarrow \mathsf{length}(f(x_1, \ldots, x_n)) \geqslant m.$$

The *SProc* semantics of synchronous dataflow defined in the next section assumes that nodes are specified by synchronous functions. This assumption is only valid if it makes sense to consider Kahn semantics with synchronous functions, so this needs to be checked. There is no problem with the case of simple connections between distinct networks.

Proposition 4.1 If $f : (A^\omega)^m \to A^\omega$ and $g : (A^\omega)^n \to A^\omega$ are synchronous then so is the function $h : (A^\omega)^{m+n-1} \to A^\omega$ defined by

$$h(x_1, \ldots, x_{m+n-1}) = f(g(x_1, \ldots, x_n), x_{n+1}, \ldots, x_{m+n-1}).$$

Proof: Straightforward. □

For feedback loops, more care is needed. If $f : (A^\omega)^{n+1} \to A^\omega$ then the network in Figure 4.4, formed by adding a feedback loop to a node described by f, is described by the function $g : (A^\omega)^n \to A^\omega$ where

$$g(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \mathsf{fix}(\lambda y.f(y, x_1, \ldots, x_n)).$$

Expanding the fix expression gives

$$g(\bar{x}) = \bigsqcup(f(\varepsilon, \bar{x}), f(f(\varepsilon, \bar{x}), \bar{x}), \ldots).$$

If $f(\varepsilon, \bar{x}) = \varepsilon$, which is certainly possible even for a synchronous f, then $g(\bar{x}) = \varepsilon$ for all $\bar{x}$ and g is not synchronous. It has already become clear that agreement between the Kahn semantics and an *SProc* semantics can only be expected if there is some additional condition on the functions describing nodes which are involved in feedback loops. Roughly speaking, the condition is that some output should be produced before any input is received. Formally, a synchronous function $f : (A^\omega)^n \to A^\omega$ is a *source* if

$$(\forall i.\mathsf{length}(x_i) \geqslant m) \Rightarrow \mathsf{length}(f(x_1, \ldots, x_n)) \geqslant m + 1.$$

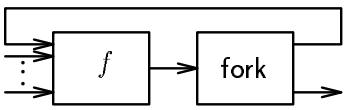


Figure 4.4: A Feedback Loop

Proposition 4.2 If $f : (A^\omega)^{n+1} \to A^\omega$ is a source then the function $g : (A^\omega)^n \to A^\omega$ describing the network formed by connecting the output of f to an input, as in Figure 4.4, and defined by $g(x_1 \ldots x_n) \stackrel{\text{def}}{=} \mathsf{fix}(\lambda y.f(y, x_1, \ldots, x_n))$, is a source.

Proof: Define a series y_r of sequences by

$$\begin{aligned} y_0 &\stackrel{\text{def}}{=} \varepsilon \\ y_{r+1} &\stackrel{\text{def}}{=} f(y_r, \bar{x}). \end{aligned}$$

Then $g(\bar{x}) = \bigsqcup_{r \geqslant 0} y_r$. Suppose $\mathsf{length}(x_i) \geqslant m$ for each i. Because f is a source, $\mathsf{length}(y_r) \geqslant r$ for each $r \leqslant m$. Hence $\mathsf{length}(g(\bar{x})) \geqslant \mathsf{length}(f(y_m, \bar{x})) \geqslant m + 1$. □

This proposition is only stated for a function f with at least two inputs. If f has just one input, so that forming the loop produces a network with no inputs, then the output is $\mathsf{fix}(f)$. In the case of continuous functions on sequences with the prefix order, f being strictly monotonic means that $\mathsf{fix}(f)$ is either ε or an infinite sequence, depending on whether $f(\varepsilon) = \varepsilon$ or $f(\varepsilon) \neq \varepsilon$.

4.6 An SProc Semantics of Synchronous Dataflow

Restricting attention to dataflow networks in which all the functions are synchronous takes a step towards *SProc*, but *SProc* is even more synchronous than that. It is based on a global clock, and every process must perform an action in every one of its ports at each time step. If processes model dataflow nodes, then at each time step every node must receive data at all of its inputs and produce data on all of its outputs. This means that the operation of the "natural numbers" network in Figure 4.3 can no longer be described by saying that the f node produces the first token which initiates an iteration towards the least fixed point. Rather, the definition of composition in *SProc* means that a feedback loop generates all possible streams which satisfy the constraints. The effect of this is that the network in Figure 4.2 generates all possible output streams: the only constraint on its behaviour is that the input to the fork node is the same as one of the outputs, and this is a defining property of fork. The problem is thus to find conditions on places at which feedback loops can be formed, which are sufficient to exclude instances of this phenomenon. In the synchronous dataflow language LUSTRE, a requirement for correct programs is that every loop must contain a pre node, which

outputs some irrelevant value at the first step, and thereafter outputs the value which was received at the previous step. When LUSTRE is modelled in $\mathcal{SP}roc$, this condition is certainly enough to ensure that feedback behaves well; however, the aim of this section is to work more abstractly and talk about properties of nodes as processes rather than particular nodes arising from concrete languages.

In general, composition in $\mathcal{SP}roc$ does not preserve determinism, where determinism means that the action performed by a process in some state determines the next state. For example, the processes $p : A \to B$ and $q : B \to C$ defined by

$$\begin{aligned} p &= (a, b) : p' + (a, b') : p'' \\ q &= (b, c) : q' + (b', c) : q'' \end{aligned}$$

are both deterministic in this sense, but

$$p \,;\, q = (a, c) : (p' \,;\, q') + (a, c) : (p'' \,;\, q'')$$

may not be. However, the processes modelling dataflow nodes are deterministic, and are combined to form processes which also model dataflow nodes and hence should also be deterministic. Clearly there must be some property of the processes modelling nodes, which is preserved by composition and which implies determinism. This property will be identified in the next section, when a comparison is made between the $\mathcal{SP}roc$ semantics and the Kahn semantics.

Now to define the $\mathcal{SP}roc$ semantics of synchronous dataflow. Suppose again that all tokens come from the same datatype A. There is an $\mathcal{SP}roc$ object $\hat{A}$ with $\Sigma_{\hat{A}} = A$ and $S_{\hat{A}} = \Sigma_{\hat{A}}^*$. A node with m inputs and n outputs becomes a morphism

$$\underbrace{\hat{A} \otimes \cdots \otimes \hat{A}}_{m} \to \underbrace{\hat{A} \otimes \cdots \otimes \hat{A}}_{n}.$$

The first step is to formalise the sense in which a process used to model a node should capture its behaviour. A process $p : \hat{A}_1 \otimes \cdots \otimes \hat{A}_m \to \hat{A}$ *computes* the continuous function $f : A_1^\omega \times \cdots \times A_m^\omega \to A^\omega$ if

$$\forall \sigma \in \mathsf{traces}(p), \pi_{m+1}^*(\sigma) \sqsubseteq f(\pi_1^*(\sigma), \ldots, \pi_m^*(\sigma))$$

$$\forall \sigma \in \mathsf{inftraces}(p), \pi_{m+1}^\omega(\sigma) = f(\pi_1^\omega(\sigma), \ldots, \pi_m^\omega(\sigma))$$

$$\forall \tau \in A_1^\omega \times \cdots \times A_m^\omega, \exists \sigma \in \mathsf{traces}(p).\langle \pi_1^\omega, \ldots, \pi_m^\omega \rangle = \tau.$$

The first of these requirements can be seen as a *safety* condition, the second as a *liveness* condition, and the third as a *totality* condition. A process $q : \hat{A}_1 \otimes \cdots \otimes \hat{A}_m \to \hat{A}_1 \otimes \cdots \otimes \hat{A}_n$ computes the family of continuous functions $\langle f_1, \ldots, f_n \rangle$ where each

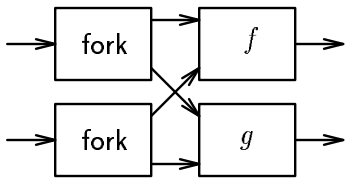


Figure 4.5: Modelling a Node

$f_i : (A^\omega)^m \to A^\omega$ if for each $1 \leq i \leq n$, the process $p ; \pi_i : \hat{A}_1 \otimes \cdots \otimes \hat{A}_m \to \hat{A}$ computes f_i, where $\pi_i : \hat{A}_1 \otimes \cdots \otimes \hat{A}_n \to \hat{A}$ is defined by

$$\frac{a_1, \ldots, a_m \in \Sigma_{\hat{A}}}{\pi_i \xrightarrow{(a_1,\ldots,a_m,a_i)} \pi_i.}$$

A process *models* a node if it computes the family of functions forming the node.

The idea behind this definition is that a process computes a function if for any finite input it never produces more output than the function would, for any infinite input it produces exactly what the function would, and it can respond to all inputs which the function can.

The process modelling a node $\langle f_1, \ldots, f_n \rangle$ is defined by first defining processes $[f_i]$ which compute the f_i, and then combining them as in Figure 4.5 (which illustrates the case of a node with two inputs and two outputs). The process $\mathsf{fork} : \hat{A} \to \hat{A} \otimes \hat{A}$ is defined by

$$\frac{a \in \Sigma_{\hat{A}}}{\mathsf{fork} \xrightarrow{(a,a,a)} \mathsf{fork}.}$$

Let $f : (A^\omega)^m \to A^\omega$ be a synchronous function. The process $[f] : \hat{A}^m \to \hat{A}$ which computes it is defined by

$$\frac{a_1, \ldots, a_m \in \Sigma_{\hat{A}}}{[f] \xrightarrow{(a_1,\ldots,a_m,\mathsf{hd}(f(a_1,\ldots,a_m)))} [\lambda(\sigma_1, \ldots, \sigma_m).\mathsf{tail}(f(a_1\sigma_1, \ldots, a_m\sigma_m))].}$$

Because f is synchronous, $f(a_1, \ldots, a_m) \neq \varepsilon$ and so $\mathsf{hd}(f(a_1, \ldots, a_m))$ is always well-defined; furthermore, $\lambda(\sigma_1, \ldots, \sigma_m).\mathsf{tail}(f(a_1\sigma_1, \ldots, a_m\sigma_m))$ is also synchronous.

It is worth mentioning a common special case. If the function f is the stream extension of a function $f' : A^m \to A$, then the definition of $[f]$ reduces to

$$\frac{a_1, \ldots, a_m \in A}{[f] \xrightarrow{(a_1,\ldots,a_m,f'(a_1,\ldots,a_m))} [f].}$$

It is now possible to prove that $[f]$ computes the function f. For notational simplicity, consider the case of a single input; the multiple-input case is a straightforward generalisation.

Lemma 4.3 If $[f]$ performs the actions $(a_1, b_1)(a_2, b_2)\ldots(a_n, b_n)$ then the resulting process is $[g]$ where

$$f(a_1 \ldots a_n \sigma) = b_1 \ldots b_n g(\sigma).$$

Proof: By induction on n. If $n = 1$ then, by the definition of $[f]$, the first action (a_1, b_1) leads to state $[\lambda\sigma.\mathsf{tail}(f(a_1\sigma))]$. Also from the definition, $b_1 = \mathsf{hd}(f(a_1))$, so for all σ, $f(a_1\sigma) = b_1\mathsf{tail}(f(a_1\sigma))$, which is the required equation.

Now assume the result for action traces of length $< n$, and all functions. As before, the first action (a_1, b_1) leads to state $[\lambda\sigma.\mathsf{tail}(f(a_1\sigma))]$. By the induction hypothesis, the next $n-1$ steps lead to a state $[g]$ such that $\mathsf{tail}(f(a_1 \ldots a_n\sigma)) = b_2 \ldots b_n g(\sigma)$. Also $b_1 = \mathsf{hd}(f(a_1 \ldots a_n\sigma))$, so $f(a_1 \ldots a_n\sigma) = b_1 \ldots b_n g(\sigma)$ as required. □

Proposition 4.4 $[f]$ computes f.

Proof: For safety, suppose $(a_1, b_1)\ldots(a_n, b_n)$ is a trace of $[f]$. By Lemma 4.3, $f(a_1 \ldots a_n\sigma) = b_1 \ldots b_n g(\sigma)$ for any σ, where $[g]$ is the state of the process after n steps. Setting $\sigma = \varepsilon$ gives $f(a_1 \ldots a_n) = b_1 \ldots b_n g(\varepsilon)$ and hence $b_1 \ldots b_n \sqsubseteq f(a_1 \ldots a_n)$.

For liveness, suppose τ is an infinite trace of $[f]$; let α and β be its first and second projections, and let (α_i) and (β_i) be chains of finite prefixes of α and β. So $\alpha = \bigsqcup_i \alpha_i$ and $\beta = \bigsqcup_i \beta_i$. By continuity $f(\alpha) = f(\bigsqcup_i(\alpha_i)) = \bigsqcup_i f(\alpha_i)$, and each $f(\alpha_i) \sqsubseteq \beta$. So $f(\alpha) \sqsubseteq \beta$.

Also, $\beta_i \sqsubseteq f(\alpha_i)$ for each i (by safety), hence $\bigsqcup_i(\beta_i) \sqsubseteq \bigsqcup_i f(\alpha_i)$, i.e. $\beta \sqsubseteq f(\alpha)$. Thus $\beta = f(\alpha)$.

Finally, totality follows from the fact that in the definition of $[f]$ the input action at the first step can be anything. □

A process is uniquely determined by the property of computing f. It is also notationally simpler to prove this for a single input.

Proposition 4.5 If p computes f then $p = [f]$.

Proof: Suppose the first step of p is (a, b). By safety, $b \sqsubseteq f(a)$ and so $b = \mathsf{hd}(f(a))$. Hence the first action of p is the same as that of $[f]$. If the next state of p is q, it suffices to show that q computes $\lambda\sigma.\mathsf{tail}(f(a\sigma))$. A bisimulation argument then completes the proof.

For safety: if σ is a trace of q, $(a, b)\sigma$ is a trace of p. So

$$\begin{aligned} b\mathsf{snd}^*(\sigma) &\sqsubseteq f(a\mathsf{fst}^*(\sigma)) \\ &= b\mathsf{tail}(f(a\mathsf{fst}^*(\sigma))) \end{aligned}$$

and so $\mathsf{snd}^*(\sigma) \sqsubseteq (\lambda\tau.\mathsf{tail}(f(a\tau)))(\mathsf{fst}^*(\sigma))$.

A similar argument establishes liveness, and totality follows from totality of p. □

Given a node N, the process $[N]$ intended to model it is defined by using fork as indicated previously.

Proposition 4.6 The process $[N]$ models the node N.

Proof: Straightforward from the definition of $[N]$, the fact that any function f is computed by the process $[f]$, and the definition of fork. □

Again, there is a uniqueness result.

Proposition 4.7 $[N]$ is the unique process modelling N.

Proof: Follows from the result for functions. □

Given a network W, the process $[W]$ is defined from the processes modelling the individual nodes by using the compact-closed structure of $\mathcal{SP}roc$ as described in the previous section.

4.7 Comparison of the Semantics

Establishing agreement between the Kahn semantics and the $\mathcal{SP}roc$ semantics means proving that when a network W is viewed as a node $\overline{W}$ by means of the Kahn semantics, the process $[W]$ models that node. This is the same as proving that $[W] = [\overline{W}]$. The restriction on forming feedback loops is that when connecting output i of a network V to some input, the function f_i in the node $\overline{V}$ encapsulating V should be a source. The next proposition gives some sufficient conditions, expressed syntactically (that is, in the graphical "syntax" of networks), that outputs are produced by sources.

Proposition 4.8 Suppose W and V are networks and output i of W is a source. Then

1. In the network obtained by connecting some outputs of V to inputs of W, output i is a source.
2. If V has only one input and one output, then in the network obtained by connecting output i of W to the input of V, the output of V is a source. More generally, connecting sources to all inputs of a function results in a source.
3. In the network obtained by connecting some output $j \neq i$ of W to an input of W, where j is a source, output i is a source.

Proof: Let the function in W producing output i be f, and suppose W has n inputs.

1. Suppose V has r inputs and functions g_j. In the new network, output i is produced by function h where

$$h(x_1, \ldots, x_{r+n}) \quad = \quad f(g_1(x_1, \ldots, x_r), \ldots, g_r(x_1, \ldots, x_r), x_{r+1}, \ldots, x_{r+n}).$$

   If $\mathsf{length}(x_j) \geqslant p$ for each j, then $\mathsf{length}(g_k(x_1, \ldots, x_r)) \geqslant p$ for each k, because g_k is synchronous. Hence $\mathsf{length}(h(x_1, \ldots, x_m)) \geqslant p + 1$ because f is a source.

2. If $f_1, \ldots, f_n$ are sources and g has n inputs, suppose that $\mathsf{length}(x_j) \geqslant p$ for each $1 \leqslant j \leqslant m$. Then for each k, $\mathsf{length}(f_k(x_1, \ldots, x_m)) \geqslant p + 1$ because f_k is a source. Hence $\mathsf{length}(g(f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m))) \geqslant p + 1$ because g is synchronous.

3. If output j is produced by function g, then (for example in the case of two inputs) the output of the new network for input x is y where $y = f(z, x)$ and $z = g(z, x)$. If $\mathsf{length}(x) \geqslant n$ then $\mathsf{length}(z) \geqslant n$ by the argument of the proof of Proposition 4.2. Hence $\mathsf{length}(y) \geqslant n + 1$ because f is a source. □

To go with the condition that an output of a network is produced by a source, there is a property of processes. If a node has a source as one of its functions and the process modelling it receives some input, then the token produced at the output which is the source should depend only on the state of the process and not on the input tokens. A process $p : A_1 \otimes \cdots \otimes A_n \to B$ is *initially independent of input i* if

$$\forall a_1, \ldots, a_n, a_i', b, b'.[(p \xrightarrow{(a_1,\ldots,a_i,\ldots,a_n,b)} q) \wedge (p \xrightarrow{(a_1,\ldots,a_i',\ldots,a_n,b')} q')] \Rightarrow b = b'.$$

A process p is *independent of input i* if all its derivatives are initially independent of input i. A process $q : A_1 \otimes \cdots \otimes A_n \to B_1 \otimes \cdots \otimes B_m$ *has output j independent of input i* if the process $q\,;\pi_j$ is independent of input i.

Proposition 4.9 If $N = \langle f_1, \ldots, f_n \rangle$ is a node in which f_i is a source, then the process $[N]$ has output i independent of all inputs.

Proof: It is enough to prove that the process $[f_i]$ has output independent of all inputs. If f_i is a source, then for any input tokens $a_1, \ldots, a_m$ we have $f_i(a_1, \ldots, a_m) = b\sigma$ for some σ and some token b. From the definition of $[f_i]$, the first output is b regardless of the values of the inputs. Hence the output is initially independent of all inputs. The next state of $[f_i]$ is $[\lambda(\sigma_1, \ldots, \sigma_m).\mathsf{tail}(f_i(a_1\sigma_1, \ldots, a_m\sigma_m))]$, and $\lambda(\sigma_1, \ldots, \sigma_m).\mathsf{tail}(f_i(a_1\sigma_1, \ldots, a_m\sigma_m))$ is also a source. Thus all derivatives have output independent of all inputs. □

It should be noted that the set $\mathcal{P}$ of all processes (of a certain type) which have output independent of all inputs, is defined as the greatest fixed point of the function which takes a set of processes and prefixes them all by initial actions in which the output does not depend on the inputs. The proof that if f is a source then $[f]$ has output independent of all inputs is a (rather informal) proof by coinduction [MT91] that $\{[f] \mid f \text{ is a source}\} \subseteq \mathcal{P}$.

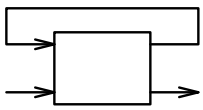
The main result comparing the $\mathcal{SP}roc$ semantics to the Kahn semantics is the following theorem.

Theorem 4.10 Let W be a network constructed according to the restriction that feedback loops are only formed from outputs which are produced by sources. Let $\overline{W}$ be the node obtained by encapsulating the Kahn semantics of W into a collection of functions. Then $[W]$ models $\overline{W}$.

Proof: The proof is by induction on the structure of W.

- If W is a single node N, there is nothing to prove, because it has already been established that $[N]$ models N.
- If W is formed from networks U and V by connecting some outputs of U to inputs of V, assume for simplicity that U and V each have a single input and a single output. It is sufficient to show that for functions f and g, the process $[f]\,;[g]$ computes gf; or, alternatively, that $[f]\,;[g] = [gf]$.

  Suppose $[f]$ receives an input a at the first step. The output is $b = \mathsf{hd}(f(a))$. With this input, $[g]$ outputs $c = \mathsf{hd}(g(b))$. The process $[f];[g]$ thus inputs a and outputs c. On the other hand, $[gf]$ inputs a and outputs $\mathsf{hd}(gf(a)) = \mathsf{hd}(g(b)) = c$. The next state of $[f]$ is $[\lambda\sigma.\mathsf{tail}(f(a\sigma))]$, that of $[g]$ is $[\lambda\tau.\mathsf{tail}(g(b\tau))]$, and that of $[gf]$ is $[\lambda\sigma.\mathsf{tail}(gf(a\sigma))]$. But the composite of $\lambda\sigma.\mathsf{tail}(f(a\sigma))$ and $\lambda\tau.\mathsf{tail}(g(b\tau))$ is $\lambda\sigma.\mathsf{tail}(g(b\mathsf{tail}(f(a\sigma)))) = \lambda\sigma.\mathsf{tail}(gf(a\sigma))$. Hence a bisimulation argument establishes the result.
- If W is formed by juxtaposition of U and V then $[W] = [U] \otimes [V]$, and the result follows easily from the fact that $[U]$ and $[V]$ model U and V.
- Suppose W is formed from V, which has two inputs and two outputs, by connecting an output to an input.



Encapsulating the Kahn semantics of V gives a node $\overline{V}$ with functions f and g (say), and by the induction hypothesis $[V]$ models $\overline{V}$. Suppose it is the f output which has been connected to the first input. Then according to the Kahn semantics, the function in $\overline{V}$ is h defined by $h(x) = g(\mathsf{fix}(s_x), x)$ where $s_x(u) = f(u, x)$. It remains to show that $[W]$ computes h.

For safety, consider a finite trace of $[W]$. It comes from a finite trace of $[V]$ in which the f output is the same as the first input. Let $\alpha = a_1 \ldots a_n$ be the projection of the trace onto the f output (so α is the trace in the loop), $\beta = b_1 \ldots b_n$ the trace on the second input, and $\gamma = c_1 \ldots c_n$ the trace on the g output. Think of $[V]$ as containing $[f]$ and $[g]$. By safety of $[f]$, $a_1 = \mathsf{hd}(f(a_1, b_1))$ and this is equal to $\mathsf{hd}(f(\varepsilon, \varepsilon))$ because f is a source. Hence $a_1 \sqsubseteq f(\varepsilon, \varepsilon) \sqsubseteq f(\varepsilon, \beta) = s_\beta(\varepsilon)$. Similarly $a_2 = \mathsf{hd}(\mathsf{tail}(f(a_1a_2, b_1b_2))) = \mathsf{hd}(\mathsf{tail}(f(a_1, b_1)))$ as f is a source. So $a_1a_2 \sqsubseteq f(a_1, b_1) \sqsubseteq f(s_\beta(\varepsilon), b_1) \sqsubseteq f(s_\beta(\varepsilon), \beta) = s_\beta^2(\varepsilon)$. Proceeding in this way gives $\alpha \sqsubseteq s_\beta^n(\varepsilon) \sqsubseteq \mathsf{fix}(s_\beta)$. By safety of $[g]$, $\gamma \sqsubseteq g(\alpha, \beta) \sqsubseteq g(\mathsf{fix}(s_\beta), \beta) = h(\beta)$.

For liveness, we consider traces and their projections as before, but now α, β and γ are infinite sequences. By liveness of $[f]$, $\alpha = f(\alpha, \beta)$, and so $\mathsf{fix}(s_\beta) \sqsubseteq \alpha$. We now argue that $\mathsf{fix}(s_\beta)$ is infinite. Because f is a source, $s_\beta(\varepsilon) = f(\varepsilon, \beta) \sqsupseteq f(\varepsilon, \varepsilon) \sqsupset \varepsilon$. So $s_\beta(\varepsilon)$ has length ≥ 1. Hence $f(s_\beta(\varepsilon), b_1)$ has length ≥ 2, and so does $s_\beta(s_\beta(\varepsilon)) = f(s_\beta(\varepsilon), \beta) \sqsupseteq f(s_\beta(\varepsilon), b_1)$. Similarly $s_\beta^n(\varepsilon)$ has length $\geq n + 1$. Hence $\mathsf{fix}(s_\beta)$ is infinite, and so it is actually the unique fixed point of s_β. Thus $\alpha = \mathsf{fix}(s_\beta)$. Liveness of $[g]$ means that $\gamma = g(\mathsf{fix}(s_\beta), \beta) = h(\beta)$.

Finally, for totality we use the fact that the output of $[f]$ is independent of all inputs. This means that at each step the next token added to the stream in the loop is determined independently of the tokens added to the inputs. Totality of $[f]$ and $[g]$ means that the new token in the loop can always be accepted as an input token, and also any token can be received at the other input. □

This theorem will be used in the next section to check that the *SProc* semantics of LUSTRE, which has already been defined directly rather than in terms of processes modelling continuous functions, agrees with the Kahn semantics. As a sideline, it is possible to identify the property of the processes used in the *SProc* semantics which causes them to remain deterministic when composed.

A process p is *initially deterministic* if

$$\forall a.[(p \xrightarrow{a} q) \wedge (p \xrightarrow{a} q')] \Rightarrow q = q'.$$

A process p is *deterministic* if all its derivatives are initially deterministic. A process $p : A \otimes \cdots \otimes A \to B$ is *initially functional* if

$$\forall a_1, \ldots, a_n, b, b'.[(p \xrightarrow{(a_1, \ldots, a_n, b)} q) \wedge (p \xrightarrow{(a_1, \ldots, a_n, b')} q')] \Rightarrow b = b'.$$

A process p is *functional* if all its derivatives are initially functional.

Lemma 4.11 If N is a node, then $[N]$ is functional and deterministic.

Proof: It follows from the definitions that $[N]$ is initially functional and initially deterministic. Also, any derivative of $[N]$ models a node and is thus initially functional and initially deterministic. Hence $[N]$ is functional and deterministic. □

Although determinism is not necessarily preserved by composition, the combination of functionality and determinism *is* preserved.

Lemma 4.12 If $p : A_1 \otimes \cdots \otimes A_n \to B_i$ and $q : B_1 \otimes \cdots \otimes B_m \to C$ are functional and deterministic, then

$$\mathsf{id} \otimes \cdots \otimes \mathsf{id}p \otimes \mathsf{id} \otimes \cdots \otimes \mathsf{id} : B_1 \otimes \cdots B_{i-1} \otimes A_1 \otimes \cdots \otimes A_n \otimes B_{i+1} \otimes \cdots \otimes B_m \to C$$

is functional and deterministic.

Proof: For notational simplicity consider the case $m = n = 1$, so that $p : A \to B$ and $q : B \to C$ and the aim is to establish functionality and determinism of $p\,;q : A \to C$. Suppose $p\,;q \to^* r$; then $r = p_1\,;q_1$ with $p \to^* p_1$ and $q \to^* q_1$. If $p_1\,;q_1 \xrightarrow{(a,c)} s$ and $p_1\,;q_1 \xrightarrow{(a,c)} s'$ then is b such that $p_1 \xrightarrow{(a,b)} p_2$, $q_1 \xrightarrow{(b,c)} q_2$, $s = p_2\,;q_2$ and similarly b' such that $p_1 \xrightarrow{(a,b')} p_2'$, $q_1 \xrightarrow{(b',c)} q_2'$, $s' = p_2'\,;q_2'$. By functionality of p, $b = b'$ and so by determinism of p, $p_2 = p_2'$. By determinism of q, $q_2 = q_2'$. Hence $s = p_2\,;q_2 = p_2'\,;q_2' = s'$ and so $p\,;q$ is deterministic.
If $p_1\,;q_1 \xrightarrow{(a,c)} s$ and $p_1\,;q_1 \xrightarrow{(a,c)} s'$ then there is b such that $p_1 \xrightarrow{(a,b)} p_2$, $q_1 \xrightarrow{(b,c)} q_2$, $s = p_2\,;q_2$ and there is b' such that $p_1 \xrightarrow{(a,b')} p_2'$, $q_1 \xrightarrow{(b',c')} q_2'$, $s' = p_2'\,;q_2'$. By functionality of p, $b = b'$ and so by functionality of q, $c = c'$. Hence $p\,;q$ is functional. □

Lemma 4.13 If $p : A_1 \otimes \cdots \otimes A_m \to C$ and $q : B_1 \otimes \cdots \otimes B_n \to D$ are functional and deterministic, then so is $p \otimes q$.

Proof: Projecting either output of $p \otimes q$ gives a process whose functionality and determinism follows directly from that of p or q. □

Lemma 4.14 If $P : A_1 \otimes \cdots \otimes A_m \otimes C \to B_1 \otimes \cdots \otimes B_n \otimes C$ is functional and deterministic, and the C output is independent of all inputs, then the process $\hat{P} : A_1 \otimes \cdots \otimes A_m \to B_1 \otimes \cdots \otimes B_n$, obtained from P by connecting the C output to the C input, is also functional and deterministic.

Proof: For simplicity consider the case $m = n = 1$, so $P : A \otimes C \to B \otimes C$. To show that $\hat{P}$ is functional, suppose that $\hat{P} \xrightarrow{(a,b)} \hat{Q}$ and $\hat{P} \xrightarrow{(a,b')} \hat{Q}'$. This means that

$P \xrightarrow{(a,c,b,c)} P'$ and $P \xrightarrow{(a,c',b',c')} Q'$ (and then $\hat{Q}$ and $\hat{Q}'$ are obtained from Q and Q' by forming feedback loops). Because the C output is independent of all inputs, $c = c'$. Hence $P \xrightarrow{(a,c,b,c)} P'$ and $P \xrightarrow{(a,c,b',c)} Q'$. Functionality of P implies that $b = b'$. So $\hat{P}$ is initially functional. The same argument applies at subsequent steps to show that all derivatives of $\hat{P}$ are initially functional, and hence $\hat{P}$ is functional.

To show that $\hat{P}$ is deterministic, suppose that $\hat{P} \xrightarrow{(a,b)} \hat{Q}$ and $\hat{P} \xrightarrow{(a,b)} \hat{Q}'$. By the same argument as before, $P \xrightarrow{(a,c,b,c)} Q$ and $P \xrightarrow{(a,c,b,c)} Q'$. By determinism of P, $Q = Q'$ and so $\hat{Q} = \hat{Q}'$. Hence $\hat{P}$ is initially deterministic; as before, the same argument as subsequent steps establishes determinism. □

4.8 *Application to* LUSTRE

There is now a semantics of LUSTRE in *SProc*, defined directly from the properties of the LUSTRE operators, and also a general semantics of synchronous dataflow in *SProc*. The general *SProc* semantics agrees with the Kahn semantics, provided that the restriction on formation of feedback loops is obeyed. The LUSTRE operators define synchronous functions on data sequences, so in order to show that the *SProc* semantics of LUSTRE agrees with the Kahn semantics it is necessary to check that the *SProc* processes computing these synchronous functions are the same as the processes defined directly from the LUSTRE operators.

The first and simplest case is that of the data operators. Suppose that $f : A^m \to A$ is a function of m arguments on some data type (it wouldn't make any difference to what follows if the arguments were not all of the same type). The function $\bar{f} : (A^\omega)^m \to A^\omega$ is defined as follows. For the case in which all the arguments of $\bar{f}$ are finite sequences of the same length n, then

$$\bar{f}(x_1x_2\ldots x_n, \ldots, z_1z_2\ldots z_n) \stackrel{\text{def}}{=} f(x_1, \ldots, z_1)f(x_2, \ldots, z_2)\ldots f(x_n, \ldots, z_n).$$

If the arguments have different lengths but are not all infinite sequences, then

$$\bar{f}(x, y, \ldots, z) \stackrel{\text{def}}{=} \bar{f}(x_1\ldots x_n, y_1\ldots y_n, \ldots, z_1\ldots z_n)$$

where n is the minimum of the lengths of the arguments. If all the arguments are infinite sequences, then $\bar{f}$ is defined by continuity. Now, $\bar{f}$ is synchronous, and it also produces the output expected of a LUSTRE data operator implementing the function f. The process $[\bar{f}]$ which computes $\bar{f}$ is defined by

$$\frac{a_1, \ldots, a_m \in \Sigma_{\hat{A}}}{[\bar{f}] \xrightarrow{(a_1,\ldots,a_m,\mathsf{hd}(\bar{f}(a_1,\ldots,a_m)))} [\lambda(\sigma_1, \ldots, \sigma_m).\mathsf{tail}(\bar{f}(a_1\sigma_1, \ldots, a_m\sigma_m))].}$$

Since $\mathsf{hd}(\bar{f}(a_1,\ldots,a_m)) = f(a_1,\ldots,a_m)$ and $\lambda(\sigma_1,\ldots,\sigma_m).\mathsf{tail}(\bar{f}(a_1\sigma_1,\ldots,a_m\sigma_m)) = \bar{f}$, this reduces to

$$\frac{a_1,\ldots,a_m \in \Sigma_{\hat{A}}}{[\bar{f}] \xrightarrow{(a_1,\ldots,a_m,f(a_1,\ldots,a_m))} [\bar{f}].}$$

If all streams were on the basic clock, so that delays never entered the picture, this would be exactly the process used in the direct *SProc* semantics of LUSTRE. But in fact, $(\delta\Delta)^n[\bar{f}]$ was used, and should be compared with the process computing a stream function which can accept delay tokens as inputs. Writing $(\delta\Delta)^n A$ for $A \cup \{*_1,\ldots,*_n\}$, define a partial function $f_{\delta\Delta} : ((\delta\Delta)^n A)^m \to (\delta\Delta)^n)A$ by

$$\begin{array}{rcll} f_{\delta\Delta}(*_i,\ldots,*_i) & \stackrel{\text{def}}{=} & *_i & 1 \leqslant i \leqslant n \\ f_{\delta\Delta}(a_1,\ldots,a_m) & \stackrel{\text{def}}{=} & f(a_1,\ldots,a_m) & a_j \neq *_i. \end{array}$$

The function $f_{\delta\Delta}$ is not defined if its inputs are delays from different clock levels, but this doesn't matter because in a valid LUSTRE program all the inputs to a data operator have to be on the same clock. A sequence function $\bar{f}_{\delta\Delta}$ can be defined as before, and there is a corresponding *SProc* process $[\bar{f}_{\delta\Delta}]$. What has to be checked is that $(\delta\Delta)^n[\bar{f}] = [\bar{f}_{\delta\Delta}]$. This is easy because, intuitively, both are just f with a trivial action on $*_i$ tokens added.

Since the **then** operator behaves as one data operator (which copies the first argument to the output) at the first step, and subsequently as another data operator (which copies the second argument to the output), essentially the same argument can be used to show that the correct process was used to model it in the *SProc* semantics.

For the **pre** operator, there is a synchronous function $\overline{\mathsf{pre}} : A^\omega \to A^\omega$ defined by

$$\overline{\mathsf{pre}}(\sigma) \stackrel{\text{def}}{=} \bot\ \sigma.$$

So the modelling process is defined by

$$\frac{a \in \Sigma_{\hat{A}}}{[\overline{\mathsf{pre}}] \xrightarrow{(a,\mathsf{hd}(\overline{\mathsf{pre}}(a)))} [\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))]}$$

which is equivalent to

$$\frac{a \in \Sigma_{\hat{A}}}{[\overline{\mathsf{pre}}] \xrightarrow{(a,\bot)} [\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))].}$$

The direct definition of the *SProc* semantics of LUSTRE used a process **pre** which could do $(a,\bot)$ at the first step and then become pre_a, using a parameter to store the token being delayed. The definition of $[\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))]$ is

$$\frac{b \in \Sigma_{\hat{A}}}{[\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))] \xrightarrow{(b,\mathsf{hd}((\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma)))b))} [\lambda\tau.\mathsf{tail}((\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma)))(b\tau))]}$$

which is equivalent to

$$\frac{b \in \Sigma_{\hat{A}}}{[\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))] \xrightarrow{(b,a)} [\lambda\tau.\mathsf{tail}(\overline{\mathsf{pre}}(b\tau))]}$$

and so $[\lambda\sigma.\mathsf{tail}(\overline{\mathsf{pre}}(a\sigma))] = \mathsf{pre}_a$ because they satisfy the same recursive definition. As was the case for the data operators, the delays can easily be inserted.

The case of **when** is similar to that of the data operators, except that the delay actions are incorporated from the beginning. There is a function

$$\overline{\mathsf{when}} : (\delta\Delta)^n A \times (\delta\Delta)^n \mathbb{B} \to (\delta\Delta)^{n+1} A$$

defined by

$$\begin{aligned} \overline{\mathsf{when}}(a, t) &\stackrel{\text{def}}{=} a \\ \overline{\mathsf{when}}(a, f) &\stackrel{\text{def}}{=} *_{n+1} \end{aligned}$$

and the synchronous function computed by a **when** node is the stream extension of $\overline{\mathsf{when}}$. It is straightforward to check that the process used to model **when** is the same as the process computing this synchronous function.

For each $a \in (\delta\Delta)^{(n-1)} A$ there is a synchronous function

$$\overline{\mathsf{current}}_a : ((\delta\Delta)^n A)^\omega \to ((\delta\Delta)^{(n-1)} A)^\omega$$

defined by

$$\begin{aligned} \overline{\mathsf{current}}_a(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon \\ \overline{\mathsf{current}}_a(*_n\sigma) &\stackrel{\text{def}}{=} a\overline{\mathsf{current}}_a(\sigma) \\ \overline{\mathsf{current}}_a(b\sigma) &\stackrel{\text{def}}{=} b\overline{\mathsf{current}}_b(\sigma) \quad (b \neq *_n). \end{aligned}$$

The operation of a **current** node is described by the function $\overline{\mathsf{current}}_\perp$. It is straightforward to check that the process computing $\overline{\mathsf{current}}_\perp$ is the same as the process which was used to model a **current** node.

This chapter has shown that to obtain agreement between the Kahn semantics and the *SProc* semantics of a network, it is necessary to restrict the formation of feedback loops to cases in which the output is a source. When dataflow nodes can potentially contain any function whatsoever, this is a non-trivial restriction. But in the case of Lustre, the only node whose output is not strict in the inputs is **pre**, and indeed the output of **pre** is a source. Any feedback loop from a port which is not the output of a **pre** node is forced by the Kahn semantics to contain the empty stream. Thus any such feedback loop can be replaced by a node which produces no output. So any Lustre network is equivalent to one in which every feedback loop involves a source, and hence there is a normal form for Lustre networks such that the *SProc* semantics of a normalised network agrees with the Kahn semantics.

4.9 Discussion

In this chapter, interaction categories have been applied to the analysis of dataflow computation, with a number of interesting results. First of all, the structure of a compact closed category supports the construction of arbitrary networks, and compact closure is essential for the formation of loops. This demonstrates that the coincidence of $\otimes$ and $\parr$ in $\mathcal{SP}roc$, which is an undesirable degeneracy if one is interested purely in models of linear logic, can be extremely useful. If processes exist which capture the behaviour of dataflow nodes, then this construction can be used to give a semantics of a dataflow language in a compact closed category. In this way, $\mathcal{SP}roc$ can be used to define the semantics of the dataflow language LUSTRE [HCRP91]. The delay operators of $\mathcal{SP}roc$ allow LUSTRE streams, which may have a complex structure of nested clocks, to be flattened into simple sequences of tokens without losing any timing information.

The classical semantics of dataflow is Kahn's model [Kah74], in which nodes are assumed to compute continuous functions of their input streams. A version of this semantics, tailored for synchronous dataflow lanaguages, has been introduced in this chapter. The new notion of a synchronous stream function is defined, in order to capture the fact that a synchronous node produces additional output whenever it receives additional input. A general semantics of synchronous networks is defined in $\mathcal{SP}roc$. This gives an interesting view of feedback loops: the relational nature of composition in $\mathcal{SP}roc$ means that the semantics generates all the possible behaviours which are compatible with the constraints imposed by the loop. This is in contrast to the view of the Kahn semantics, in which the behaviour of a network with loops is reached by a process of iteration. However, it is shown that the $\mathcal{SP}roc$ semantics and the Kahn semantics predict the same behaviour for all networks in which the formation of feedback loops is restricted to points at which output is being produced independently of the input.

This condition on the formation of cycles is related to Wadge's work on deadlock detection in dataflow networks [Wad81]. What he means by deadlock is precisely the phenomenon of a feedback loop in which no data is generated, resulting in a network which produces no output. The use of the term "deadlock" corresponds to the idea that such a network cannot produce output until it has received that very output as input. Wadge's approach to detection or prevention of deadlock is to assign a numerical link between each output of a node and each input, to represent the extent to which the output depends on the current, previous or future values of the input. For example, in a LUSTRE **pre** node the link is 1 because the first output depends only on the second input. In a node corresponding to a LUSTRE data operator all links are 0, because input is required immediately to produce output. The language LUCID [AW85] has

a node called next, whose first output token is the second input token; this node has a link of -1. Once these links have been defined, the *cycle sum test* can be used to determine whether or not a given network can deadlock. In each cycle, there is a closed path passing from an input to an output of each node, and from an output of one node to an input of the next node. Summing the numerical links along this path yields an integer, the *cycle sum*. The condition for deadlocks to be excluded is that every cycle sum is strictly positive. It is essential to add the links around the entire cycle, because for example the negative link arising from the presence of a next node cancels the positive effect of a pre node. Wadge then relates this notion to the stream functions corresponding to the nodes, by observing that a function corresponding to a link of $+1$ always outputs a stream one token longer than its input, and so on. In the synchronous networks considered in this chapter, negative links cannot occur because a node must always produce another output token when another input token arrives. This means that to avoid deadlock, it is sufficient to have at least one positive link in every cycle; in LUSTRE, the only node with a positive link is pre, hence the condition that every cycle must contain a pre node. In terms of the stream functions, every positive link corresponds to a source; because there are no negative links, the actual positive value is irrelevant. To summarise the connection with Wadge's theory, the results of this chapter establish the equivalence of the Kahn and *SProc* semantics for non-deadlocking networks; the synchronous model allows the condition for absence of deadlock to be simplified to the inclusion of a source in every cycle.

The observation that LUSTRE streams can be flattened by means of the *SProc* delay operators allows the Kahn semantics of synchronous dataflow to be applied to it, assuming that a clock consistency check has already been carried out. Defining synchronous functions for the various LUSTRE nodes and then applying the *SProc* semantics of synchronous dataflow results in an alternative definition of an *SProc* process for each LUSTRE node, but it turns out that these processes are the same as those which were defined directly when giving the first model of LUSTRE in *SProc*. The end result is that LUSTRE has a semantics in *SProc*, and this agrees with the Kahn semantics as long as every feedback loop contains a pre node; this requirement is already a condition for correctness of LUSTRE programs.

LUSTRE is not the only language which can be analysed in this way—an *SProc* semantics of SIGNAL [GGBM91] has also been defined [GN93]. Because of the additional complexities of the clock structure of SIGNAL, a detailed comparison of its *SProc* and Kahn semantics awaits further work.

5

Verification

5.1 Introduction

In Chapter 1 it was claimed that one of the benefits of the interaction categories approach to concurrency is the ability to use types to specify complex properties of processes and then use type-checking techniques to verify such properties. In this chapter, that claim will be justified. Two kinds of property are considered: safety and deadlock-freedom. The categories $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ have safety specifications built into their types, but it turns out that some additional work is needed to make use of them. This is described in Section 5.2.

For properties other than safety, there is a general method for systematically enriching the types of a given category by the addition of extra constraints: Abramsky's notion of specification structure, which he originally applied to deadlock-freedom of synchronous processes. The general construction is described in Section 5.3; the specific details relating to deadlock-freedom are presented in Section 5.4 and applied to dataflow networks. The result of this construction is a category of synchronous processes in which types guarantee freedom from deadlock, and hence the typed process combinators yield compositional proof rules for deadlock-freedom. In fact there are two different specification structures for deadlock-freedom, which turn out to define the same category.

In Section 5.5 the ideas behind the construction of the synchronous deadlock-free category are adapted and applied to asynchronous processes. A new category $\mathcal{FP}roc$ is defined, in which all processes are fair, and a specification structure for deadlock-freedom is constructed over this category. The result is another category in which compositional reasoning about deadlock-freedom is supported. The use of this category is illustrated by an analysis of the dining philosophers problem.

In both the synchronous and asynchronous cases, the deadlock-free category is $*$-autonomous but not compact closed, so cyclic process configurations cannot always be formed. Because it is important to be able to construct cycles in many examples, proof rules are established which embody sufficient conditions for cycles to be allowable.

5.2 Safety

The categories $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ have safety properties built into their types, but it is not completely straightforward to use them to express interesting safety properties of processes. If P is a process of type A, then P has a certain safety property defined by the type A. But this safety property may not be the one of interest for the verification of P, especially if A has been constructed from simpler types. For example, suppose the types B and C in $\mathcal{ASP}roc$ each have a single observable action: say $\Sigma_B \stackrel{\text{def}}{=} \{b, \tau_B\}$ and $\Sigma_C \stackrel{\text{def}}{=} \{c, \tau_C\}$. Defining a process P by $P = (b, \tau_C).(\tau_B, c).P$, it easily follows that $P : B \otimes C$. Now in fact, P satisfies a much stronger safety specification which requires that the actions b and c alternate, but $S_{B\otimes C}$ imposes no restriction on when b and c actions occur in relation to each other. It would be difficult to specify that P should do b and c alternately, because there is no way to choose S_B and S_C so that $S_{B\otimes C}$ expresses the desired safety property.

The obvious solution is to define a type T with the same alphabet as $B \otimes C$ but a more restricted safety specification, and prove that P also has type T. However, the typing $P : B \otimes C$ contains the information that P has two ports of types B and C, and this information is lost when P is given the type T. A way to express the fact that P satisfies the stronger specification represented by the type T, without forgetting what the interface looks like, is to define a "safety morphism" $s : B \otimes C \to B \otimes C$ such that P satisfies the specification in T iff $P = P\,;s$ (in this composition, P is viewed as a morphism $I \to B \otimes C$). Taking s to be the identity morphism id_T considered as a morphism $B \otimes C \to B \otimes C$, first note that $\mathsf{traces}(P\,;s) \subseteq \mathsf{traces}(P)$ because s acts as a buffer. Now

$$\begin{aligned} P = P\,;s \quad &\Rightarrow \quad \mathsf{traces}(P) = \mathsf{traces}(P\,;s) \\ &\Rightarrow \quad \mathsf{traces}(P) \subseteq \mathsf{traces}(P\,;s) \\ &\Rightarrow \quad \mathsf{traces}(P) \subseteq S_T \qquad\qquad \text{because } \mathsf{traces}(P\,;s) \subseteq S_T \end{aligned}$$

and

$$\begin{aligned} \mathsf{traces}(P) \subseteq S_T \quad &\Rightarrow \quad P \text{ has type } T \\ &\Rightarrow \quad P\,;s = P \qquad \text{because } s = \mathsf{id}_T. \end{aligned}$$

Hence $P = P\,;s \iff \mathsf{traces}(P) \subseteq S_T$. This allows safety conditions to be stated equationally, without necessarily constructing types with appropriate safety specifications.

A suitable example with which to illustrate this idea is the specification of a cyclic scheduler [Mil89].

5.2.1 The Scheduler

Suppose there are n tasks, or client processes, which can be started by signals $a_1, \ldots, a_n$ and which indicate completion by signals $b_1, \ldots, b_n$. The aim is to control the clients in such a way that

1. the clients are started in cyclic order, beginning with number 1;
2. for each i, the initiation a_i and the completion b_i must alternate.

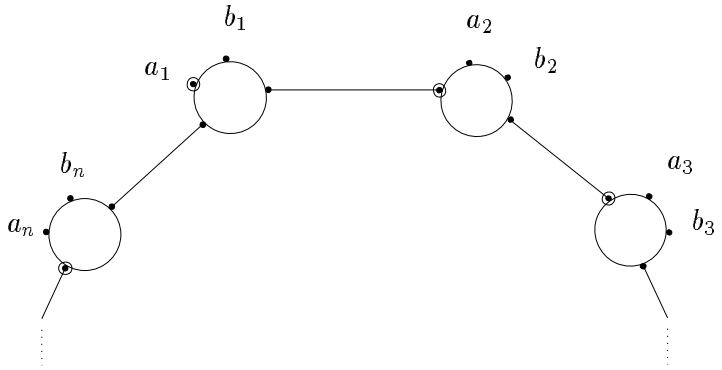
This should be done by implementing a scheduler process which connects to the n clients and ensures that they start and stop in the correct order. Following Milner, the scheduler is implemented as a ring of n cells, each controlling one of n client processes and talking to its two neighbours in the ring. This means that the implementation is independent of the actual number of clients; once the basic cell has been implemented, any number can be linked up to form a scheduler. Initially, a cell can be in one of two states, depending on whether it is going to communicate first with its client or with a neighbour. These two states are represented by the processes `cell` and `dcell` respectively. In CCS, the definitions are

$$\begin{aligned} \texttt{cell} &= a.c.(b.d.\texttt{cell} + d.b.\texttt{cell}) \\ \texttt{dcell} &= d.\texttt{cell} \end{aligned}$$

and the two states of a cell can be represented by the pictures

`cell` `dcell`

in which the ringed ports are the ones ready to fire. The arrangement of cells in the scheduler is:



In CCS, the ith cell is constructed by relabelling a,b,c,d in `cell` or `dcell` to a_i,b_i,c_i,$\overline{c}_{i-1}$. Parallel composition and restriction are then used to form the private connections

between the cells. The relabelling is necessary to ensure that the correct ports become connected; when using interaction categories, such relabelling is not needed.

According to the pattern of ringed ports in this picture, one cell is in the `cell` state and the rest are in the `dcell` state. If the cycle is completed with `dcell` processes, then this is the initial state of the complete scheduler. The way in which the scheduler works is that the a_1 action happens first, starting client 1. Then the first `cell` sends a signal to its neighbour, by means of the c action, telling it that client 2 can be started. Because the c port of cell 1 is connected to the d port of cell 2, the second cell (which started in the `dcell` state) moves to the `cell` state and is ready to start client 2. Meanwhile, cell 1 has entered a state in which it is waiting for either a stop signal from its client, or a message that client n has been started. When both of these messages have been received, in either order, client 1 can be started again.

The aim now is to construct the scheduler as a process in $\mathcal{ASP}roc$. The same type X can be used for each port, with $\Sigma_X \stackrel{\text{def}}{=} \{\bullet, \tau_X\}$ and $S_X \stackrel{\text{def}}{=} \{\bullet^n \mid n < \omega\}$. The type of `cell` and `dcell` will be $X^\perp \parr (X \otimes X^\perp) \parr X$, which it is sometimes convenient to write as $X_d^\perp \parr (X_a \otimes X_b^\perp) \parr X_c$ to show which port corresponds to which action in the CCS definition. The copies of X with $(-)^\perp$ applied correspond to the ports which are thought of as inputs, and the unadorned copies of X are output ports. This choice is arbitrary; all that matters is that two ports to be connected have types dual to each other. In $\mathcal{ASP}roc$, of course, this is no constraint, but as mentioned several times before, it is clearer to keep track of such logical distinctions.

The type $X^\perp \parr (X \otimes X^\perp) \parr X$ has fifteen observable actions (such as $(\bullet, \tau_X, \tau_X, \bullet)$) but only four of them are needed for the definition of a cell: $(\bullet, \tau_X, \tau_X, \tau_X)$, $(\tau_X, \bullet, \tau_X, \tau_X)$, $(\tau_X, \tau_X, \bullet, \tau_X)$ and $(\tau_X, \tau_X, \tau_X, \bullet)$. These actions can be abbreviated to d, a, b and c respectively in accordance with the notation used for actions in the description of the scheduler. The processes `cell` and `dcell` can be built from these actions, and because the safety specification of X is so simple, it is easy to see that they have the desired type. The next step is to connect the cells together to form the scheduler process, called `sched`. Using the categorical calculation from Section 4.3 (corresponding to the Cut rule of Chapter 6), $n-1$ cells can be connected in a line, the c port of each one being attached to the d port of the next. An application of the Cycle rule (again, a calculation from Section 4.3) then results in the cyclic process `sched` with type $(X \otimes X^\perp) \parr \cdots \parr (X \otimes X^\perp)$ in which $2n$ copies of X appear. For this example, a simplified adaptation of the typed process calculus notation which will be introduced in Chapter 6 is useful, so that the line of cells is $\texttt{cell} \cdot \texttt{dcell} \cdot \ldots \cdot \texttt{dcell}$ and the complete system is $\texttt{sched} = (\texttt{cell} \cdot \texttt{dcell} \cdot \ldots \cdot \texttt{dcell})\backslash_{c,d}$. The notation $P \cdot Q$ indicates a connection between two processes; in general it is essential to specify which ports are involved, but for this example the diagram suffices. The notation $P\backslash_{c,d}$ indicates

a connection between the c and d ports of P, forming a cycle.

In the above typing of the cell, the a and b ports appear as $X \otimes X^{\perp}$ rather than $X \parr X^{\perp}$. This is so that a cell can be connected to a client whose type is $X^{\perp} \parr X$, which is the natural type for a process with two ports. Again, in *ASProc* these distinctions are unimportant, but are included here for a clearer exposition.

5.2.2 Verifying the Scheduler

In this section, the original specification of the scheduler is interpreted as a safety specification. This means that the goal is merely to show that nothing undesirable happens, and not to prove anything about desirable properties of the infinite behaviour of the scheduler. Specifically, a safe scheduler should never start clients in the wrong order or attempt to restart a client before it has stopped, but it could stop after a finite time and fail to do any further scheduling.

This safety property defines a safety morphism

$$\texttt{schedsafety} : (X \otimes X^{\perp}) \parr \cdots \parr (X \otimes X^{\perp}) \to (X \otimes X^{\perp}) \parr \cdots \parr (X \otimes X^{\perp}).$$

As indicated by the previous general remarks, `schedsafety` is the identity morphism on $(X \otimes X^{\perp}) \parr \cdots \parr (X \otimes X^{\perp})$ restricted to the behaviours allowed by the safety specification.

Proving that the scheduler satisfies its specification means proving that

$$\texttt{sched} = \texttt{sched} \,;\, \texttt{schedsafety}.$$

The problem can be broken down by taking advantage of the fact that the scheduler is implemented as a ring of cells, and defining a safety specification for a cell such that safety of all the cells implies safety of the scheduler. Safety for a cell means that

- a and b happen alternately, starting with a
- c and d happen alternately
- a happens between d and c, and only then.

The first clause specifies alternation of start and stop signals for a client; clearly if this is true of each cell then it is true for each client when the cells are connected up, because connecting processes together makes a selection from each of their behaviours but cannot introduce any new ones. The other clauses ensure that a cell's interactions with its neighbours are correctly sequenced in relation to its own start and stop signals. The d action tells a cell that the previous cell has started its client, and the c action

is used to tell the next cell that the current client has been started. Thus the third clause ensures that in between these two actions, a cell actually does issue a start signal. Furthermore, once c has been done, a does not happen again until the signal d has been received. The last two clauses between them guarantee that the start signals are issued in the correct cyclic order by the scheduler.

This almost allows the safety specification of the scheduler to be recovered, but to ensure that client 1 is started first, the difference between a `cell` and a `dcell` must be taken into account. The specification of a `cell` is the above plus the requirement that a happens first, and the specification of a `dcell` is the above plus the requirement that d happens first.

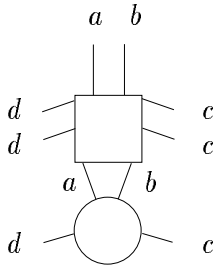
This defines two safety morphisms:

$$\texttt{cellsafety}, \texttt{dcellsafety} : X^\perp \parr (X \otimes X^\perp) \parr X \to X^\perp \parr (X \otimes X^\perp) \parr X,$$

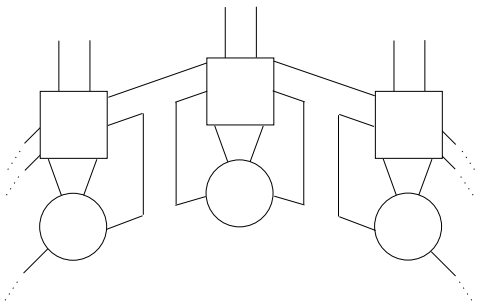
and it is clear from the definitions of the processes that $\texttt{cell} = \texttt{cell}\,;\texttt{cellsafety}$ and $\texttt{dcell} = \texttt{dcell}\,;\texttt{dcellsafety}$. The definition of the scheduler can be rewritten as

$$\begin{aligned} \texttt{sched} &= (\texttt{cell} \cdot \texttt{dcell} \cdot \ldots \cdot \texttt{dcell}) \setminus_{c,d} \\ &= ((\texttt{cell}\,;\texttt{cellsafety}) \cdot (\texttt{dcell}\,;\texttt{dcellsafety}) \cdot \\ &\quad \ldots \cdot (\texttt{dcell}\,;\texttt{dcellsafety})) \setminus_{c,d} . \end{aligned}$$

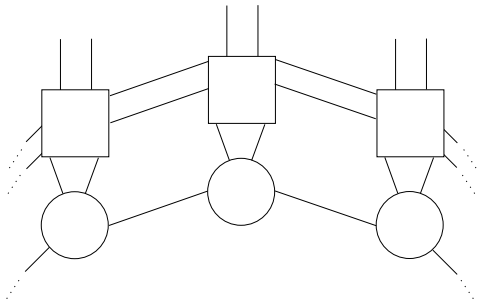
An alternative, perhaps clearer, pictorial view is that each `cell`;`cellsafety` or `dcell`; `dcellsafety` looks like this.



The circle is a `cell` or `dcell`, with a,b,c and d ports, and the square is a safety morphism with two copies of each port. The scheduler, according to the above calculation, is a ring formed from these processes.



Because `cellsafety` and `dcellsafety` are identity morphisms (considered to be in larger types), `cell` = `cell` ; `cellsafety` and `dcell` = `dcell` ; `dcellsafety`, this arrangement of processes is equivalent to the following alternative arrangement.



It is not clear whether this calculation can be justified on the basis of interaction category axioms, but it is easy to see why it holds in $\mathcal{ASP}roc$ for this example, by thinking about information flow: in the first arrangement, output from the c and d ports of a cell is made to satisfy the specification by passing through `cellsafety`, but since it has already been established that `cell` satisfies its specification, the cells can be connected directly together as in the second arrangement; it then makes no difference if the `cellsafety` processes are also connected together. Thus

$$\begin{aligned}\texttt{sched} \;=\; & ((\texttt{cell}\cdot\texttt{dcell}\cdot\ldots\cdot\texttt{dcell})\backslash_{c,d})\,; \\ & ((\texttt{cellsafety}\cdot\texttt{dcellsafety}\cdot\ldots\cdot\texttt{dcellsafety})\backslash_{c,d}).\end{aligned}$$

The equational way of stating that safety of all the cells implies safety of the scheduler is

$$\begin{aligned}& (\texttt{cellsafety}\cdot\texttt{dcellsafety}\cdot\ldots\cdot\texttt{dcellsafety})\backslash_{c,d} \\ = \;& ((\texttt{cellsafety}\cdot\texttt{dcellsafety}\cdot\ldots\cdot\texttt{dcellsafety})\backslash_{c,d})\,;\texttt{schedsafety}\end{aligned}$$

and so

$$\begin{aligned}\texttt{sched} \;=\; & ((\texttt{cell}\cdot\texttt{dcell}\cdot\ldots\cdot\texttt{dcell})\backslash_{c,d})\,; \\ & ((\texttt{cellsafety}\cdot\texttt{dcellsafety}\cdot\ldots\cdot\texttt{dcellsafety})\backslash_{c,d})\,;\texttt{schedsafety}.\end{aligned}$$

Working backwards through the previous equations gives

$$\begin{aligned}\texttt{sched} \;=\; & (((\texttt{cell}\,;\texttt{cellsafety})\cdot(\texttt{dcell}\,;\texttt{dcellsafety})\cdot \\ & \ldots\cdot(\texttt{dcell}\,;\texttt{dcellsafety}))\backslash_{c,d})\,;\texttt{schedsafety} \\ =\; & ((\texttt{cell}\cdot\texttt{dcell}\cdot\ldots\cdot\texttt{dcell})\backslash_{c,d})\,;\texttt{schedsafety} \\ =\; & \texttt{sched}\,;\texttt{schedsafety}\end{aligned}$$

which is the desired conclusion. This calculation is an example of a general scheme for structuring correctness proofs of processes which are built from subcomponents.

5.3 Specification Structures

A specification structure S over a category $\mathcal{C}$ allows the construction of a new category $\mathcal{C}_S$ in which the objects of $\mathcal{C}$ have been enriched by the addition of extra properties. In terms of types, the result is that a type carries more information and it is correspondingly more difficult for a term to inhabit it. The structure of $\mathcal{C}$ (in the case of $\mathcal{SP}roc$ or $\mathcal{ASP}roc$, the structure of an interaction category) can be lifted to $\mathcal{C}_S$, and there is a faithful functor $\mathcal{C}_S \to \mathcal{C}$ which forgets the additional properties in the types. The fact that $\mathcal{C}_S$ is a category gives compositional proof rules for programs satisfying the additional specifications, which can be used no matter how the necessary properties of the programs were established. A typical starting point might be a collection of processes and their types in $\mathcal{C}$. In some cases it will be possible, for quite general reasons, to assign a type in $\mathcal{C}_S$ to a process which has a type in $\mathcal{C}$. For other processes it will be necessary to use traditional verification methods to prove that they have certain properties and can thus be typed in $\mathcal{C}_S$. But once this has been done, any of the processes can be combined according to the available rules in $\mathcal{C}_S$, yielding processes which again have types in $\mathcal{C}_S$. Thus some (or, ideally, much) of the work of verification can be packaged up into appropriate types which incorporate properties preserved by composition.

Formally, a *specification structure* S over a category $\mathcal{C}$ is defined by the following data.

- For each object A of $\mathcal{C}$, a set $P_S A$ of *predicates* or *properties* over A.
- For each pair A, B of objects of $\mathcal{C}$, a relation $S_{A,B} \subseteq \mathcal{C}(A,B) \times P_S A \times P_S B$.

$S_{A,B}(f,\theta,\varphi)$ will be written $\theta\{f\}\varphi$. This relation is required to satisfy the following conditions.

- For each object A of $\mathcal{C}$ and each $\theta \in P_S A$, $\theta\{\mathsf{id}_A\}\theta$.
- For all objects A, B, C of $\mathcal{C}$, morphisms $f : A \to B$, $g : B \to C$, and properties $\theta \in P_S A$, $\varphi \in P_S B$, $\psi \in P_S C$, if $\theta\{f\}\varphi$ and $\varphi\{g\}\psi$ then $\theta\{f\,;g\}\psi$.

If S is a specification structure on $\mathcal{C}$, then an object of $\mathcal{C}_S$ is a pair (A,θ) with A an object of $\mathcal{C}$ and $\theta \in P_S A$, and a morphism $f : (A,\theta) \to (B,\varphi)$ is a morphism $f : A \to B$ in $\mathcal{C}$ such that $\theta\{f\}\varphi$.

Proposition 5.1 $\mathcal{C}_S$ is a category, and there is a faithful functor $\mathcal{C}_S \to \mathcal{C}$ defined by $(A,\theta) \mapsto A$ and $f \mapsto f$.

Proof: The conditions on the relation $S_{A,B}$ are precisely those necessary to ensure that the identity morphisms of $\mathcal{C}$ are also the identities in $\mathcal{C}_S$, and that composition in $\mathcal{C}_S$ can be defined. The forgetful functor from $\mathcal{C}_S$ to $\mathcal{C}$ is obviously faithful. □

In general, there isn't a corresponding free functor from $\mathcal{C}$ to $\mathcal{C}_S$, because unless the sets of properties have more structure there is no canonical way to choose a property over each object.

If $\mathcal{C}$ has a type structure specified by various functors and natural transformations, it can be lifted to $\mathcal{C}_S$ by giving each functor an action on properties and verifying that the components of the natural transformations satisfy the conditions necessary for them to be morphisms in $\mathcal{C}_S$. For example, if $\mathcal{C}$ has a tensor product $\otimes : \mathcal{C}^2 \to \mathcal{C}$ it can be lifted to C_S by defining

$$\otimes_{A,B} : P_S A \times P_S B \to P_S(A \otimes B)$$

satisfying

$$\theta\{f\}\varphi, \theta'\{g\}\varphi' \Rightarrow \theta \otimes \theta'\{f \otimes g\}\varphi \otimes \varphi'$$

and then defining $\otimes$ on $\mathcal{C}_S$ by

$$(A, \theta) \otimes (B, \varphi) \stackrel{\text{def}}{=} (A \otimes B, \theta \otimes_{A,B} \varphi).$$

If assoc is the associativity natural isomorphism in $\mathcal{C}$, it lifts to $\mathcal{C}_S$ provided that for all objects A, B, C of $\mathcal{C}$ and properties $\theta \in P_S A$, $\varphi \in P_S B$, $\psi \in P_S C$,

$$(\theta \otimes \varphi) \otimes \psi\{\mathsf{assoc}_{A,B,C}\}\theta \otimes (\varphi \otimes \psi).$$

Examples of specification structures include the following.

- The specification structure P over $\mathcal{S}et$ defined by taking the set of properties over any set X to be X itself (so that a property over X is just an element of X, when X is non-empty). If $x \in X$, $y \in Y$ and $f : X \to Y$ then $x\{f\}y \stackrel{\text{def}}{\Leftrightarrow} f(x) = y$. Then $\mathcal{S}et_P$ is the category of pointed sets and point-preserving functions. This example shows that $\mathcal{C}_S$ may have fewer objects than $\mathcal{C}$: the set $\varnothing$ has no properties and so there is no corresponding object in $\mathcal{S}et_P$. The product structure of $\mathcal{S}et$ can be lifted to $\mathcal{S}et_P$ by taking $\times_{A,B} : P_P A \times P_P B \to P_P(A \times B)$ to be the identity function on $A \times B$. For $\times$ to be a functor on $\mathcal{S}et_P$ requires, for $f : A \to C$ and $g : B \to D$,

  $$a\{f\}c, b\{g\}d \Rightarrow (a, b)\{f \times g\}(c, d)$$

  which is clearly true. The rest of the product structure on $\mathcal{S}et_P$ can be checked similarly—for example, $\pi : A \times B \to C$ satisfies $(a, b)\{\pi\}a$ for any $a \in A$, $b \in B$.

- The specification structure S over $\mathcal{S}et$ defined by taking a property over a set X to be a partial order on X. If $\leqslant$ and $\sqsubseteq$ are partial orders on X and Y respectively, then $\leqslant\{f\}\sqsubseteq$ when $\forall a, b \in X.[a \leqslant b \Rightarrow f(a) \sqsubseteq f(b)]$. Then $\mathcal{S}et_S$ is the category $\mathcal{P}os$ of partially ordered sets and monotonic functions.

- A combination of the two previous examples: the specification structure S over $\mathcal{S}et_P$ defined by taking a property over (X, x) to be a partial order on X in which x is the least element. If $\leqslant$ and $\sqsubseteq$ are partial orders on X and Y respectively, and x, y are the least elements, then $\leqslant\{f\}\sqsubseteq$ when $f(x) = y$ and $\forall a, b \in X.[a \leqslant b \Rightarrow f(a) \sqsubseteq f(b)]$. Then $(\mathcal{S}et_P)_S$ is the category of partially ordered sets with bottom, and strict monotonic functions.

As suggested by the examples, a specification structure is essentially the same as a faithful functor. If $F : \mathcal{D} \to \mathcal{C}$ is faithful, define a specification structure S over $\mathcal{C}$ by

$$P_S A \stackrel{\text{def}}{=} \{X \in \mathsf{ob}\mathcal{D} \mid FX = A\}$$

and if $f : A \to B$ in $\mathcal{C}$ and $X \in P_S A$, $Y \in P_S B$,

$$X\{f\}Y \stackrel{\text{def}}{\Leftrightarrow} \exists \hat{f} : X \to Y \text{ in } \mathcal{D} \text{ with } F(\hat{f}) = f.$$

It is easy to check that S satisfies the specification structure conditions. An object of $\mathcal{C}_S$ is a pair (A, X) with $FX = A$, i.e. essentially just an object X of $\mathcal{D}$, and a morphism $(A, X) \to (B, Y)$ is a morphism $f : A \to B$ of $\mathcal{C}$ such that $f = F(\hat{f})$ for some (necessarily unique) morphism $\hat{f} : X \to Y$ of $\mathcal{D}$. Thus $\hat{f}$ can be taken as the morphism in $\mathcal{C}_S$, and this recovers $\mathcal{D}$. Conversely, starting with a specification structure S over $\mathcal{C}$ and constructing $\mathcal{C}_S$, the forgetful functor $U : \mathcal{C}_S \to \mathcal{C}$ can be used to recover S.

5.4 Synchronous Deadlock-Freedom

In this chapter, deadlock means termination. A more refined treatment might consider *unsuccessful* termination as deadlock; the view taken here is that all termination is unsuccesful. A process may have both terminating and non-terminating behaviours, but a *deadlock-free* process is one which has no maximal finite behaviours. For example, the process $a.b.\mathsf{nil}$ can deadlock; the process P defined by $P = a.P + b.\mathsf{nil}$ can deadlock although it can also generate the infinite trace a^ω; the process Q defined by $Q = a.b.Q$ is deadlock-free.

Deadlock-freedom is not preserved by composition: two processes may individually be deadlock-free, but when forced to communicate they could deadlock each other by

being unable to agree on a sequence of actions to perform. For example, if the CCS processes P and Q are defined by $P = a.b.P$ and $Q = \bar{a}.c.Q$ then composing them means forming the process $(P \mid Q)\backslash\{a, b, c\}$. In this process, P and Q can communicate for a single step, but then deadlock occurs because P must do b next while Q can only do c.

In order to construct a category of deadlock-free processes which are guaranteed to remain deadlock-free when composed with each other, more information is needed than just the fact that a process runs forever. The rest of this section describes two different approaches to building suitable extra information into the types, by constructing a specification structure over $\mathcal{SP}roc$. Both of these approaches are due to Abramsky. It will then turn out that they are equivalent.

5.4.1 Sets of Processes

The first construction of a specification structure for deadlock-freedom takes a property over a type to be a set of processes of that type. This is clearly the most general notion of property, and has no inherent connection with deadlocks. For these properties to say anything about deadlock-freedom, the sets of processes must be carefully chosen in a way which will now be described.

First, say that a process P of type A *converges*, written $P\downarrow$, if whenever $P \stackrel{s}{\longrightarrow}{}^{*} Q$ there is $a \in \Sigma_A$ and a process R such that $Q \stackrel{a}{\longrightarrow} R$. Convergence means deadlock-freedom; the reason for the choice of terminology is an analogy with proofs of strong normalisation in Classical Linear Logic [Gir87, Abr93a].

Given processes P and Q of type A, the process $P \sqcap Q$ of type A is defined by

$$\frac{P \stackrel{a}{\longrightarrow} P' \qquad Q \stackrel{a}{\longrightarrow} Q'}{P \sqcap Q \stackrel{a}{\longrightarrow} P' \sqcap Q'.}$$

For each type A, the *orthogonality* relation on the set of processes of type A is defined by

$$P \perp Q \stackrel{\text{def}}{\Leftrightarrow} (P \sqcap Q)\downarrow .$$

If P and Q are orthogonal, they can communicate without deadlocking: any common trace can be extended by an action which is available to both processes.

Because only deadlock-free processes are of interest in this section, it is convenient to restrict attention to those types of $\mathcal{SP}roc$ whose safety specifications do not force termination. Such types are called *progressive*: A is progressive if

$$\forall s \in S_A.\exists a \in \Sigma_A.sa \in S_A.$$

The full subcategory of $\mathcal{SP}roc$ consisting of just the progressive objects is denoted by $\mathcal{SP}roc_{pr}$; this is the category over which the specification structure for deadlock-freedom will be defined. $\mathcal{SP}roc_{pr}$ inherits all the structure of $\mathcal{SP}roc$, apart from the zero object.

For each object A of $\mathcal{SP}roc_{pr}$, let $\mathsf{Proc}(A)$ be the set of convergent processes of type A. The orthogonality relation is extended to sets of processes by defining, for $U, V \subseteq \mathsf{Proc}(A)$ and $P : A$,

$$\begin{aligned} P \perp U &\stackrel{\text{def}}{\Leftrightarrow} \forall Q \in U. P \perp Q \\ U \perp V &\stackrel{\text{def}}{\Leftrightarrow} \forall P \in U. P \perp V. \end{aligned}$$

Orthogonality then generates an operation of negation on sets of processes, defined by

$$U^{\perp} \stackrel{\text{def}}{=} \{P \in \mathsf{Proc}(A) \mid P \perp U\}.$$

The next lemma is true quite generally of any operation $(-)^{\perp}$ defined in this way from a symmetric orthogonality relation.

Lemma 5.2 For all $U, V \subseteq \mathsf{Proc}(A)$,

$$\begin{aligned} U \subseteq V &\Rightarrow V^{\perp} \subseteq U^{\perp} \\ U &\subseteq U^{\perp\perp} \\ U^{\perp} &= U^{\perp\perp\perp}. \end{aligned}$$

The specification structure D for deadlock-freedom over $\mathcal{SP}roc_{pr}$ can now be defined. A property over a type A is a non-empty, $^{\perp\perp}$-invariant set of convergent processes of type A:

$$P_D A \stackrel{\text{def}}{=} \{U \subseteq \mathsf{Proc}(A) \mid (U \neq \varnothing) \wedge (U^{\perp\perp} = U)\}.$$

Also associated with the definition of $(-)^{\perp}$ is a closure operator: for any $U \subseteq \mathsf{Proc}(A)$, $U^{\perp\perp}$ is the smallest $^{\perp\perp}$-invariant set of processes containing U.

It will be essential to know that when $U \in P_D A$, $U^{\perp} \in P_D A$. It is always true that $U^{\perp}$ is $^{\perp\perp}$-invariant, so all that is needed is that $U^{\perp} \neq \varnothing$. To establish this, consider the process max_A of type A defined by

$$\frac{a \in S_A}{\mathsf{max}_A \xrightarrow{a} \mathsf{max}_{A/a}}.$$

This definition applies to any object A; if A is progressive then $\mathsf{max}_A \downarrow$. For any process P of type A, $P \sqcap \mathsf{max}_A = P$, and so if $P \downarrow$ then $P \perp \mathsf{max}_A$. Hence whenever $U \in P_D A$, $\mathsf{max}_A \in U^{\perp}$ and so $U^{\perp} \neq \varnothing$.

The relation $U\{f\}V$ is defined via a *satisfaction* relation between processes and properties, written $P \models U$. In this case, the definition is very simple:

$$P \models U \quad \stackrel{\text{def}}{\Leftrightarrow} \quad P \in U$$

but satisfaction in the next specification structure is more complex so it is useful to set the pattern of definitions now. When the linear operations have been defined on properties, the definition

$$U\{f\}V \quad \stackrel{\text{def}}{\Leftrightarrow} \quad f \models U \multimap V$$

can be used to specify the morphisms of the deadlock-free category. This category will be called $\mathcal{SP}roc_D$; its definition in terms of a satisfaction relation follows the same pattern as the original definition of $\mathcal{SP}roc$.

The $(-)^\perp$ operation on sets of processes has already been defined. The multiplicative operations are defined in terms of $\otimes$:

$$\begin{aligned} U \otimes V &\stackrel{\text{def}}{=} \{P \otimes Q \mid P \in U, Q \in V\}^{\perp\perp} \\ U \parr V &\stackrel{\text{def}}{=} (U^\perp \otimes V^\perp)^\perp \\ U \multimap V &\stackrel{\text{def}}{=} (U \otimes V^\perp)^\perp. \end{aligned}$$

In order to prove that D satisfies the specification structure axioms, a few lemmas are needed.

Lemma 5.3 If $U \subseteq V \subseteq \mathsf{Proc}(A)$ then $\mathsf{id}_A \in U \multimap V$.

Proof: We need $\mathsf{id}_A \in (U \otimes V^\perp)^\perp$. Now,

$$\begin{aligned} (U \otimes V^\perp)^\perp &= \{P \otimes Q \mid P \in U, Q \in V^\perp\}^{\perp\perp\perp} \\ &= \{P \otimes Q \mid P \in U, Q \in V^\perp\}^\perp \end{aligned}$$

so it is enough to show $\mathsf{id}_A \perp \{P \otimes Q \mid P \in U, Q \in V^\perp\}$. Let $P \in U$ and $Q \in V^\perp$. $U \subseteq V$ implies $V^\perp \subseteq U^\perp$, so $Q \in U^\perp$ and hence $P \perp Q$. For any common trace s of id_A and $P \otimes Q$, $\mathsf{fst}^*(s)$ is a trace of P and $\mathsf{snd}^*(s)$ is a trace of Q, and $\mathsf{fst}^*(s) = \mathsf{snd}^*(s)$. So there is an action a such that $\mathsf{fst}^*(s)a$ is a trace of P and $\mathsf{snd}^*(s)a$ is a trace of Q. Hence (a, a) is an action such that $s(a, a)$ is a trace of both id_A and $P \otimes Q$. This means that $(\mathsf{id}_A \sqcap (P \otimes Q))\!\downarrow$, and so $\mathsf{id}_A \perp P \otimes Q$. □

For the next two lemmas, a slight abuse of notation is useful. If $f : A \to B$ and $P : A$, there is a process $P\,;f$ of type B obtained by regarding P as a morphism $I \to A$, composing with f, and then regarding the resulting morphism $I \to B$ as a process of type B. Similarly, if $Q : B^\perp$ there is a process $f\,;Q$ of type A.

Lemma 5.4 If $f : A \to B$, $U \in P_D A$, $V \in P_D B$, $f \in U \multimap V$ and $P \in U$, then $P ; f \in V$.

Proof: We have $f \perp \{P \otimes Q \mid P \in U, Q \in V^\perp\}$ and need to prove, for any $Q \in V^\perp$, $(P ; f) \perp Q$. Let $Q \in V^\perp$ and let s be a common trace of $P ; f$ and Q. The definition of composition means that there is a trace t of f such that $\mathsf{fst}^*(t)$ is a trace of P and $\mathsf{snd}^*(t) = s$. Because $f \perp (P \otimes Q)$, there is an action (a, b) such that $t(a, b)$ is a trace of f, $\mathsf{fst}^*(t)a$ is a trace of P and $\mathsf{snd}^*(t)b$ is a trace of Q. Then sb is a common trace of $P ; f$ and Q, so $((P ; f) \sqcap Q)\!\downarrow$ as required. □

The proof of the next lemma is similar.

Lemma 5.5 If $f : A \to B$, $U \in P_D A$, $V \in P_D B$, $f \in U \multimap V$ and $Q \in V^\perp$, then $f ; Q \in U^\perp$.

Proposition 5.6 D is a specification structure over $\mathcal{SP}roc_{pr}$.

Proof: The first requirement is that if A is any object of $\mathcal{SP}roc_{pr}$ and $U \in P_D A$, $U\{\mathsf{id}_A\}U$. This follows from Lemma 5.3.

Next, suppose that A, B, C are objects of $\mathcal{SP}roc_{pr}$ and $U \in P_D A$, $V \in P_D B$ and $W \in P_D C$. If $f : A \to B$ and $g : B \to C$ with $U\{f\}V$ and $V\{g\}W$, we need $U\{f ; g\}W$. Thus the goal is to prove that

$$f ; g \quad \perp \quad \{P \otimes R \mid P \in U, R \in W^\perp\}.$$

Suppose that $P \in U$ and $R \in W$. By Lemmas 5.4 and 5.5, $P ; f \in V$ and $g ; R \in V^\perp$. Hence $(P ; f) \perp (g ; R)$, i.e. $((P ; f) \sqcap (g ; R))\!\downarrow$. It follows that $((f ; g) \sqcap (R \otimes S))\!\downarrow$, as can easily be checked. □

It is now legitimate to talk about the category $\mathcal{SP}roc_D$ of deadlock-free processes. Next, the definition of the $*$-autonomous structure on $\mathcal{SP}roc_D$ can be completed. There is only one property over I, namely the set $\{P\}$ where P is the unique convergent process of type I. Concretely, $P = * : P$. Clearly $P \sqcap P = P$, so $P \perp P$ and $\{P\}^\perp = \{P\}$. Also, of course, $P = \mathsf{max}_I$. This means that $I_D = \perp_D$. For $\otimes$ to be a functor on $\mathcal{SP}roc_D$ requires that for morphisms $f : A \to C$ and $g : B \to D$ of $\mathcal{SP}roc$ and sets $U \in P_D A$, $V \in P_D B$, $W \in P_D C$, $Z \in P_D D$ with $U\{f\}W$ and $V\{g\}Z$,

$$U \otimes V\{f \otimes g\}W \otimes Z.$$

The monoidal structure of $\mathcal{SP}roc$ lifts to $\mathcal{SP}roc_D$ provided that, for all properties U,

V, W over appropriate objects, the following hold.

$$U \otimes (V \otimes W)\{\mathsf{assoc}_{A,B,C}\}(U \otimes V) \otimes W$$
$$(U \otimes V)\{\mathsf{symm}_{A,B}\}(V \otimes U)$$
$$(I_D \otimes U)\{\mathsf{unitl}_A\}U$$
$$(U \otimes I_D)\{\mathsf{unitr}_A\}U.$$

The closed structure requires that whenever $(U \otimes V)\{f\}W$, $U\{\Lambda(f)\}(V \multimap W)$; and that

$$((U \multimap V) \otimes U)\{\mathsf{Ap}_{A,B}\}V.$$

Linear negation is functorial if whenever $U\{f\}V$, $V^\perp\{f^\perp\}U^\perp$. It is straightforward to verify all of these conditions. For example, consider the case of symm.

Proposition 5.7 If $U \in P_D A$ and $V \in P_D B$, then $(U \otimes V)\{\mathsf{symm}_{A,B}\}(V \otimes U)$.

Proof: We need $\mathsf{symm} \in (U \otimes V) \multimap (V \otimes U)$, i.e. $\mathsf{symm} \in ((U \otimes V) \otimes (V \otimes U)^\perp)^\perp$, or equivalently $\mathsf{symm} \perp \{P \otimes Q \mid P \in U \otimes V, Q \in (V \otimes U)^\perp\}$.

First, suppose that $Q \in (V \otimes U)^\perp = \{R \otimes S \mid R \in V, S \in U\}^{\perp\perp\perp}$, i.e.

$$Q \perp \{R \otimes S \mid R \in V, S \in U\}.$$

Defining $Q' \stackrel{\text{def}}{=} Q[(b,a) \mapsto (a,b)]$, it is clear that $Q' \perp \{S \otimes R \mid S \in U, R \in V\}$ and so $Q' \in (U \otimes V)^\perp$.

Now suppose that $P \in U \otimes V$ and $Q \in (V \otimes U)^\perp$, and s is a common trace of symm and $P \otimes Q$. The definition of symm means that $\mathsf{fst}^*(s) = \mathsf{snd}^*(s)[(b,a) \mapsto (a,b)]$. Also, $\mathsf{fst}^*(s)$ is a trace of P and $\mathsf{snd}^*(s)[(b,a) \mapsto (a,b)]$ is a trace of Q'. Because $P \perp Q'$, there is an action (a,b) available to both P and Q'. So Q can do (b,a), and $P \otimes Q$ can do $((a,b),(b,a))$. This action is also available to symm. Hence $\mathsf{symm} \perp P \otimes Q$, as required. □

The rest of the structure of $\mathcal{SP}roc$ will soon be defined on $\mathcal{SP}roc_D$, but first it is useful to have a supply of properties over each type. For any object A, the process max_A of type A has already been defined. Now define M_A as a synonym for $\mathsf{Proc}(A)$.

Proposition 5.8 For every object A of $\mathcal{SP}roc_{pr}$, $\{\mathsf{max}_A\}^\perp = M_A$ and $M_A^\perp = \{\mathsf{max}_A\}$.

Proof: For any $P \in \mathsf{Proc}(A)$, $P \perp \mathsf{max}_A$. Hence $M_A \perp \{\mathsf{max}_A\}$. This means that $\{\mathsf{max}_A\}^\perp \supseteq M_A$; also, $\{\mathsf{max}_A\}^\perp \subseteq M_A$. This gives $\{\mathsf{max}_A\}^\perp = M_A$.

For the second part, we already have $\{\mathsf{max}_A\} \subseteq M_A^\perp$. Now suppose that $P \neq \mathsf{max}_A$. There is a process P', a trace s and an action $a \in \Sigma_A$ such that $sa \in S_A$ and $P \stackrel{s}{\longrightarrow}{}^* P'$ but P' cannot do a. Define Q to be the same process as P, except that the node P' is

replaced by Q' where $Q' = a : \mathsf{max}_{A/sa}$. Then $P \sqcap Q$ does not converge, so $P \not\perp M_A$. □

Corollary 5.9 $\{\mathsf{max}_A\}^{\perp\perp} = \{\mathsf{max}_A\}$ and $M_A^{\perp\perp} = M_A$.

This means that for every type A, there are at least two constructions of properties over A, leading to M_A and $\{\mathsf{max}_A\}$. However, these properties are not always distinct.

Proposition 5.10 If the object A is such that $s \in S_A \Rightarrow \exists! a \in \Sigma_A . sa \in S_A$ then max_A is the only process of type A, and conversely.

In this situation, $\{\mathsf{max}_A\} = M_A$; in fact, it must also be the case that $A \cong I$.

It is easy to calculate $\otimes$ and $\parr$ of these properties.

Proposition 5.11 For any objects A and B of $\mathcal{SP}roc_{pr}$, $\{\mathsf{max}_A\} \otimes \{\mathsf{max}_B\} = \{\mathsf{max}_{A \otimes B}\}$.

Proof: This follows from the fact that $\mathsf{max}_A \otimes \mathsf{max}_B = \mathsf{max}_{A \otimes B}$. □

Corollary 5.12 $M_A \parr M_B = M_{A \parr B}$.

Proposition 5.13 For any objects A and B of $\mathcal{SP}roc_{pr}$, $M_A \otimes M_B = M_{A \otimes B}$.

Proof: Since $M_A \otimes M_B = \{P \otimes Q \mid P \in M_A, Q \in M_B\}^{\perp\perp}$, it is enough to prove that $\{P \otimes Q \mid P \in M_A, Q \in M_B\}^{\perp} = \{\mathsf{max}_{A \otimes B}\}$. Clearly

$$\mathsf{max}_{A \otimes B} \perp \{P \otimes Q \mid P \in M_A, Q \in M_B\}.$$

Suppose that $R \in \mathsf{Proc}(A \otimes B)$ and $R \neq \mathsf{max}_{A \otimes B}$. At some point in the tree of R, there is an action (a, b) which is unavailable. For simplicity, say that R cannot do (a, b). Then if $P = a : \mathsf{max}_{A/a}$ and $Q = b : \mathsf{max}_{B/b}$, $(P \otimes Q) \not\perp R$. □

Corollary 5.14 $\{\mathsf{max}_A\} \parr \{\mathsf{max}_B\} = \{\mathsf{max}_{A \parr B}\}$.

Properties of the form M_A and $\{\mathsf{max}_A\}$ can be used to show that $\mathcal{SP}roc_D$ is not compact closed. Consider the types A and B with $\Sigma_A = \{a, b\}$, $\Sigma_B = \{c, d\}$ and unrestricted safety specifications. Then

$$\begin{aligned} M_A \otimes \{\mathsf{max}_B\} &= \{P \otimes \mathsf{max}_B \mid P \in M_A\}^{\perp\perp} \\ M_A \parr \{\mathsf{max}_B\} &= (\{\mathsf{max}_A\} \otimes M_B)^{\perp} \\ &= \{\mathsf{max}_A \otimes Q \mid Q \in M_B\}^{\perp}. \end{aligned}$$

Defining processes X and Y of type $A \otimes B$ by

$$\begin{aligned} X &= (b,c) : X + (a,d) : X \\ Y &= (a,c) : Y + (b,d) : Y \end{aligned}$$

it is easy to see that $X \in \{P \otimes \mathsf{max}_B \mid P \in M_A\}^\perp$ and $Y \in \{\mathsf{max}_A \otimes Q \mid Q \in M_B\}^\perp$. But $X \not\perp Y$, which means that $X \notin \{P \otimes \mathsf{max}_B \mid P \in M_A\}^{\perp\perp}$. Hence $M_A \otimes \{\mathsf{max}_B\} \neq M_A \parr \{\mathsf{max}_B\}$.

Loss of compact closure is to be expected, as in general the arbitrary formation of cycles is likely to lead to deadlock. $\mathcal{SP}roc_D$ does, however, validate the Mix rule; this will be proved later.

It is clear that if the objects A and B are progressive, so is $A \oplus B$. Thus $\mathcal{SP}roc_{pr}$ has non-empty coproducts, and hence also products. These can be lifted to $\mathcal{SP}roc_D$.

$$\begin{aligned} U \oplus V &\stackrel{\text{def}}{=} (\{P[a \mapsto \mathsf{inl}(a)] \mid P \in U\} \cup \{Q[b \mapsto \mathsf{inr}(b)] \mid Q \in V\})^{\perp\perp} \\ U \,\&\, V &\stackrel{\text{def}}{=} (U^\perp \oplus V^\perp)^\perp. \end{aligned}$$

The requirements for these to define products and coproducts on $\mathcal{SP}roc_D$ are that for any progressive objects A, B of $\mathcal{SP}roc$ and $U \in P_D(A)$, $V \in P_D(B)$:

$$\begin{gathered} (U \,\&\, V)\{\pi_1\}U \\ (U \,\&\, V)\{\pi_2\}V \\ U\{\mathsf{inl}\}(U \oplus V) \\ V\{\mathsf{inr}\}(U \oplus V). \end{gathered}$$

The definitions extend to countable products and coproducts in the obvious way. The zero object of $\mathcal{SP}roc$ is not progressive, because it has an empty sort, and so it cannot be lifted to $\mathcal{SP}roc_D$.

Not only do $\otimes$ and $\parr$ become distinct in $\mathcal{SP}roc_D$, but so too do $\oplus$ and $\&$. To see this, consider the types A and B defined by $\Sigma_A = \{a\}$, $\Sigma_B = \{b\}$ and with unrestricted safety specifications. Then

$$\begin{aligned} M_A \oplus M_B &= (\{\mathsf{max}_A\} \cup \{\mathsf{max}_B\})^{\perp\perp} \\ &= \{\mathsf{max}_A, \mathsf{max}_B\}^{\perp\perp} \\ M_A \,\&\, M_B &= (M_A \oplus M_B)^\perp \\ &= \{\mathsf{max}_A, \mathsf{max}_B\}^\perp, \end{aligned}$$

omitting inl and inr for clarity. Now, $\{\mathsf{max}_A, \mathsf{max}_B\}^\perp = \{\mathsf{max}_A + \mathsf{max}_B\}$, but

$$\{\mathsf{max}_A + \mathsf{max}_B\}^\perp \supseteq \{\mathsf{max}_A + \mathsf{max}_B, \mathsf{max}_A, \mathsf{max}_B\}$$

and so $M_A \oplus M_B$ is strictly larger than $M_A \,\&\, M_B$.

Although $\mathcal{SP}roc_D$ does not have biproducts, the non-deterministic $+$ operation still makes sense. If $U \in P_D A$, $P, Q \in U$ and $R \in U^\perp$, then $P + Q \perp R$. This means $P + Q \in U^{\perp\perp} = U$. The nil processes, however, do not exist in $\mathcal{SP}roc_D$ as they are not convergent. So each homset of $\mathcal{SP}roc_D$ has a commutative and associative $+$ operation but this operation has no unit.

The delay functors are lifted to $\mathcal{SP}roc_D$ by means of the following definitions.

$$\begin{aligned} \circ U &\stackrel{\text{def}}{=} \{\circ P \mid P \in U\}^{\perp\perp} \\ \delta U &\stackrel{\text{def}}{=} \{\delta P \mid P \in U\}^{\perp\perp} \\ \Delta U &\stackrel{\text{def}}{=} \{\Delta P \mid P \in U\}^{\perp\perp}. \end{aligned}$$

In the case of $\circ$ the application of $(-)^{\perp\perp}$ is unnecessary, as $\{\circ P \mid P \in U\}$ is already ${}^{\perp\perp}$-invariant. The conditions required for $\circ$, δ and Δ to be functors on $\mathcal{SP}roc_D$ are

$$\begin{gathered} \circ U \{\circ f\} \circ V \\ \delta U \{\delta f\} \delta V \\ \Delta U \{\Delta f\} \Delta V \end{gathered}$$

when $f : A \to B$ in $\mathcal{SP}roc$, $U \in P_D A$, $V \in P_D B$ and $U\{f\}V$. The conditions for the monad (δ, η, μ) to lift to $\mathcal{SP}roc_D$ are

$$\begin{gathered} U \{\eta\} \delta U \\ \delta\,\delta U \{\mu\} \delta U \end{gathered}$$

and similarly for Δ. The functor $\circ$ on $\mathcal{SP}roc_D$ has the UFPP provided that whenever $f : A \to \circ A$ and $g : \circ B \to B$ satisfy $U\{f\}\circ U$ and $\circ V\{g\}V$, the morphism $h : A \to B$ defined by the UFPP in $\mathcal{SP}roc$ satisfies $U\{h\}V$.

All of these conditions are satisfied, so the entire temporal structure of $\mathcal{SP}roc$ lifts to $\mathcal{SP}roc_D$.

The lack of biproducts means that the construction of $!$ as a cofree cocommutative comonoid cannot be used in $\mathcal{SP}roc_D$. But it is sufficient to define $!$ on sets of processes and check that the structural morphisms lift from $\mathcal{SP}roc$ to $\mathcal{SP}roc_D$.

$$\begin{aligned} !U &\stackrel{\text{def}}{=} \{!P \mid P \in U\}^{\perp\perp} \\ ?U &\stackrel{\text{def}}{=} (!(U^\perp))^\perp \end{aligned}$$

The conditions required of the structural morphisms are

$$\begin{gathered} !U \{\mathsf{weak}\} I_D \\ !U \{\mathsf{contr}\} !U \otimes !U \\ !U \{\mathsf{der}\} U \end{gathered}$$

and, if $f : !A \to B$ with $!U\{f\}V$,

$$!U\{f^{\dagger}\}!V.$$

Proposition 5.15 $\mathcal{SP}roc_D$ is a $*$-autonomous category with countable (non-empty) products and coproducts, exponentials, unit delay functor and delay monads. The forgetful functor $U : \mathcal{SP}roc_D \to \mathcal{SP}roc$ preserves all of this structure.

5.4.2 Ready Specifications

The specification structure presented in the previous section results in a category which supports compositional verification of deadlock-freedom, but there is perhaps a lack of intuition about the use of sets of processes. There is an alternative specification structure for deadlock-freedom, which was in fact the first to be discovered, and which may be more easily motivated. Surprisingly, it turns out to be equivalent to the sets of processes approach.

As mentioned before, the reason why deadlock-freedom is not generally preserved by composition is that two deadlock-free processes may, when forced to communicate, reach states from which no further communication is possible even though both processes have more actions available. This observation leads to the idea that if a type is to guarantee compositional deadlock-freedom, it must specify something about which actions a process must be prepared to perform in certain states. The way in which this information is captured is via the notions of ready pair and ready specification.

A *ready pair* over an $\mathcal{SP}roc$ object $A = (\Sigma_A, S_A)$ is a pair (s, X) in which $s \in S_A$ and $\varnothing \neq X \subseteq \Sigma_A$, such that $\forall x \in X.sx \in S_A$. The set X is the *ready set* of the ready pair. The set of ready pairs over an object A is denoted by $\mathsf{RP}(A)$. If P is a process of type A, then

$$\begin{aligned} \mathsf{initials}(P) &\stackrel{\text{def}}{=} \{x \in \Sigma_A \mid \exists Q.P \xrightarrow{x} Q\} \\ \mathsf{readies}(P) &\stackrel{\text{def}}{=} \{(s, X) \mid (P \xrightarrow{s}{}^{*} Q) \wedge (X = \mathsf{initials}(Q))\}. \end{aligned}$$

For any process P, $\mathsf{readies}(P)$ is the set of pairs (s, X) representing the actions (those in X) which P is ready to engage in after performing a sequence s of actions. Note that $\mathsf{readies}(P)$ does not necessarily consist entirely of ready pairs; it can contain pairs $(s, \varnothing)$. P is *deadlock-free* if and only if there is no trace s such that $(s, \varnothing) \in \mathsf{readies}(P)$. For example,

$$\mathsf{readies}(a.b.\mathsf{nil} + a.c.\mathsf{nil}) = \{(\varepsilon, \{a\}), (a, \{b\}), (a, \{c\}), (ab, \varnothing), (ac, \varnothing)\}$$

and if $P = a.P$,

$$\mathsf{readies}(P) = \{(a^n, \{a\} \mid n < \omega)\}.$$

The idea of a ready pair, and the related notions of *failures* and *refusals*, appear in the process algebra literature [BW90, BHR84, Hoa85]. There, however, they are used to define semantic alternatives to bisimulation; the use made of ready pairs in this chapter is very different.

As before, the specification structure hinges on an orthogonality relation. The relation $\perp$ on $\mathsf{RP}(A)$ is defined by

$$(s, X) \perp (t, Y) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad ((s = t) \Rightarrow X \cap Y \neq \varnothing).$$

The idea is that if (s, X) and (t, Y) are ready pairs of two processes which are supposed to be communicating, $(s, X) \perp (t, Y)$ means that if they have been communicating so far $(s = t)$ there is some action which they are both prepared to do next $(X \cap Y \neq \varnothing)$ and thus continue the communication.

The properties needed for the specification structure are *ready specifications*. A ready specification over an object A is a non-empty set R of ready pairs over A, satisfying

- $((s, X) \in R) \wedge (x \in X) \Rightarrow \exists Y.(sx, Y) \in R$
- $(sx, Y) \in R \Rightarrow \exists X.[(s, X) \in R \wedge x \in X]$.

The set of ready specifications over A is denoted by $\mathsf{RS}(A)$. Not all objects have ready specifications: $\mathsf{RS}(A) \neq \varnothing$ if and only if A is progressive. Again, the specification structure is defined over $\mathcal{SP}roc_{pr}$.

In the standard way, the orthogonality relation is extended to ready specifications and used to define an operation of linear negation. The set $P_{D'}A$ of properties over an object A of $\mathcal{SP}roc_{pr}$ is the set of ready specifications θ over A such that $\theta^{\perp\perp} = \theta$. The relation $\theta\{f\}\varphi$ is again defined via a satisfaction relation between processes of any type A and ready specifications over A. Satisfaction is defined by

$$P \models \theta \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \mathsf{readies}(P) \subseteq \theta.$$

Because θ contains no pairs with empty ready sets, $P \models \theta$ implies that P is deadlock-free. The next step is to define a $*$-autonomous structure on the objects of $\mathcal{SP}roc_{D'}$, which enables the definition $\theta\{f\}\varphi \stackrel{\text{def}}{\Leftrightarrow} f \models \theta \multimap \varphi$ to be made.

Recall that an object of $\mathcal{SP}roc_{D'}$ is a pair (A, θ) in which A is an object of $\mathcal{SP}roc_{pr}$ and $\theta \in \mathsf{RS}(A)$ is a ready specification such that $\theta^{\perp\perp} = \theta$. Linear negation is defined on objects by $(A, \theta)^{\perp} \stackrel{\text{def}}{=} (A^{\perp}, \theta^{\perp}) = (A, \theta^{\perp})$. It is easier to define $\parr$ before $\otimes$; the definition is

$$(A, \theta) \parr (B, \varphi) \stackrel{\text{def}}{=} (A \parr B, \theta \parr \varphi)$$

where

$$\theta \parr \varphi \stackrel{\text{def}}{=} \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \theta^\perp, (\mathsf{snd}^*(s), V) \in \varphi^\perp\}^\perp.$$

This looks like a definition by de Morgan duality; the reason is that, since $R^{\perp\perp\perp} = R^\perp$ for any ready specification R, the easiest way to construct a $^{\perp\perp}$-invariant ready specification is by making it $S^\perp$ for some S. The corresponding $\otimes$ is then defined by de Morgan duality from $\parr$.

Before proving that D' is a specification structure, it is interesting to consider some example calculations with ready specifications. Let A and B be objects of $\mathcal{SP}roc_{pr}$ with $\Sigma_A \stackrel{\text{def}}{=} \{a, b\}$, $\Sigma_B \stackrel{\text{def}}{=} \{c, d\}$ and unrestricted safety specifications. Define a ready specification θ over A by $\theta \stackrel{\text{def}}{=} \{(s, \{a, b\}) \mid s \in \Sigma_A^*\}$. For the rest of this example, since the safety specifications in these types impose no constraint, the traces can be omitted from ready pairs, leaving just the ready *sets*. So θ is the set $\{\{a, b\}\}$ containing a single ready set. The definition of orthogonality gives

$$\begin{aligned} \theta^\perp &= \{\{a\}, \{b\}, \{a, b\}\} \\ \theta^{\perp\perp} &= \{\{a, b\}\} \end{aligned}$$

so that $\theta^{\perp\perp} = \theta$ and hence $\theta \in P_D A$. For an example of a ready specification which is not $^{\perp\perp}$-invariant, take $\alpha \stackrel{\text{def}}{=} \{\{a\}\}$, so that

$$\begin{aligned} \alpha^\perp &= \{\{a\}, \{a, b\}\} \\ \alpha^{\perp\perp} &= \{\{a\}, \{a, b\}\}. \end{aligned}$$

Then $\alpha^{\perp\perp} \neq \alpha$ (although $\alpha^\perp$ is, of course, $^{\perp\perp}$-invariant).

Defining a ready specification φ over B by $\varphi \stackrel{\text{def}}{=} \{\{c\}, \{d\}, \{c, d\}\}$, analogy with $\theta^\perp$ gives

$$\begin{aligned} \varphi^\perp &= \{\{c, d\}\} \\ \varphi^{\perp\perp} &= \varphi. \end{aligned}$$

It is now possible to calculate $\theta \parr \varphi$ and $\theta \otimes \varphi$, using a simplified form of the definition of $\parr$ which results from omitting traces from ready pairs.

$$\begin{aligned} \theta \parr \varphi &= \{U \times V \mid U \in \theta^\perp, V \in \varphi^\perp\}^\perp \\ &= \{\{(a,c),(a,d)\}, \{(b,c),(b,d)\}, \{(a,c),(a,d),(b,c),(b,d)\}\}^\perp \\ &= \{X \mid X \cap \{(a,c),(a,d)\} \neq \varnothing, X \cap \{(b,c),(b,d)\} \neq \varnothing\} \\ &= \{\{(a,c),(a,d),(b,c),(b,d)\}, \{(a,c),(a,d),(b,c)\}, \{(a,c),(a,d),(b,d)\}, \\ &\quad \{(a,c),(b,c),(b,d)\}, \{(a,d),(b,c),(b,d)\}, \\ &\quad \{(a,c),(b,c)\}, \{(a,c),(b,d)\}, \{(a,d),(b,c)\}, \{(a,d),(b,d)\}\}. \end{aligned}$$

$$\begin{aligned}
\theta \otimes \varphi &= (\theta^{\perp} \parr \varphi^{\perp})^{\perp} \\
&= \{U \times V \mid U \in \theta, V \in \varphi\}^{\perp\perp} \\
&= \{\{(a,c),(b,c)\},\{(a,d),(b,d)\},\{(a,c),(a,d),(b,c),(b,d)\}\}^{\perp\perp} \\
&= \{X \mid X \cap \{(a,c),(b,c)\} \neq \varnothing, X \cap \{(a,d),(b,d)\} \neq \varnothing\}^{\perp} \\
&= \{\{(a,c),(b,c),(a,d),(b,d)\},\{(a,c),(b,c),(a,d)\},\{(a,c),(b,c),(b,d)\}, \\
&\quad \{(a,c),(a,d),(b,d)\},\{(b,c),(a,d),(b,d)\}, \\
&\quad \{(a,c),(a,d)\},\{(a,c),(b,d)\},\{(b,c),(a,d)\},\{(b,c),(b,d)\}\}^{\perp} \\
&= \{\{(a,c),(b,c),(a,d),(b,d)\},\{(b,c),(a,d),(b,d)\},\{(a,c),(a,d),(b,d)\}, \\
&\quad \{(a,c),(b,c),(b,d)\},\{(a,c),(b,c),(a,d)\}, \\
&\quad \{(a,c),(b,c)\},\{(a,d),(b,d)\}\}.
\end{aligned}$$

These calculations show that $\theta \parr \varphi \neq \theta \otimes \varphi$, because the sets $\{(a,c),(b,d)\}$ and $\{(a,d),(b,c)\}$ are in $\theta \parr \varphi$ but not $\theta \otimes \varphi$. Thus $\mathcal{SP}roc_{D'}$ is not compact closed. However, the Mix rule is valid in $\mathcal{SP}roc_{D'}$, and this will be proved after establishing that D' is a specification structure and $\mathcal{SP}roc_{D'}$ is $*$-autonomous.

Lemma 5.16 If A is an object of $\mathcal{SP}roc_{pr}$ and $\alpha, \beta \in P_{D'}A$ with $\alpha \subseteq \beta$, then $\alpha\{\mathsf{id}_A\}\beta$.

Proof: We need $\mathsf{id}_A \models \alpha^{\perp} \parr \beta$. This means

$$\mathsf{readies}(\mathsf{id}_A) \subseteq \alpha^{\perp} \parr \beta$$

or equivalently

$$\mathsf{readies}(\mathsf{id}_A) \subseteq \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha, (\mathsf{snd}^*(s), V) \in \beta^{\perp}\}^{\perp}$$

i.e.

$$\mathsf{readies}(\mathsf{id}_A) \perp \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha, (\mathsf{snd}^*(s), V) \in \beta^{\perp}\}.$$

If $(s, X) \in \mathsf{readies}(\mathsf{id}_A)$, $(\mathsf{fst}^*(s), U) \in \alpha$ and $(\mathsf{snd}^*(s), V) \in \beta^{\perp}$, the requirement is $X \cap (U \times V) \neq \varnothing$.

$\alpha \subseteq \beta \Rightarrow \beta^{\perp} \subseteq \alpha^{\perp}$, so $(\mathsf{snd}^*(s), V) \in \alpha^{\perp}$. Hence $(\mathsf{fst}^*(s), U) \perp (\mathsf{snd}^*(s), V)$. Also, $(s, X) \in \mathsf{readies}(\mathsf{id}_A) \Rightarrow \mathsf{fst}^*(s) = \mathsf{snd}^*(s)$ and so $U \cap V \neq \varnothing$. If $a \in U \cap V$ then $(a, a) \in U \times V$ and it only remains to show that $(a, a) \in X$.

Because $(\mathsf{fst}^*(s), U)$ is a ready pair and $a \in U$, $\mathsf{fst}^*(s)a \in S_A$ and so $s(a,a) \in S_{A^{\perp} \parr A}$. This means that after the trace s, id_A must be able to do (a, a), and thus $(a, a) \in X$ as required. □

Proposition 5.17 D' is a specification structure over $\mathcal{SP}roc_{pr}$.

Proof: Firstly, if A is any object of $\mathcal{SP}roc_{pr}$ and $\theta \in P_{D'}A$, we need to check that $\theta\{\mathsf{id}_A\}\theta$. This is a special case of Lemma 5.16.

Secondly, suppose that A, B, C are objects of $\mathcal{SP}roc_{pr}$ and $\theta \in P_{D'}A$, $\varphi \in P_{D'}B$, $\psi \in P_{D'}C$. If $f : A \to B$ and $g : B \to C$ with $\theta\{f\}\varphi$ and $\varphi\{g\}\psi$, we need to check that $\theta\{f\,;g\}\psi$. The statement that $\theta\{f\}\varphi$ and $\varphi\{g\}\psi$ is equivalent to $f \models \theta^\perp \parr \varphi$ and $g \models \varphi^\perp \parr \psi$, so we have

$$\begin{array}{rcl}(s,X) \in \mathsf{readies}(f) & \Rightarrow & (s,X) \perp \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \theta, (\mathsf{snd}^*(s), V) \in \varphi^\perp\} \\ (t,Y) \in \mathsf{readies}(g) & \Rightarrow & (t,Y) \perp \{(t, V \times W) \mid (\mathsf{fst}^*(t), V) \in \varphi, (\mathsf{snd}^*(t), W) \in \psi^\perp\}.\end{array}$$

To deduce $\theta\{f\,;g\}\psi$ we need $f\,;g \models \theta^\perp \parr \psi$, which means checking that

$$\begin{array}{l}(u,Z) \in \mathsf{readies}(f\,;g) \Rightarrow \\ \qquad\qquad (u,Z) \perp \{(u, U \times W) \mid (\mathsf{fst}^*(u), U) \in \theta, (\mathsf{snd}^*(u), W) \in \psi^\perp\}.\end{array}$$

Suppose $(u,Z) \in \mathsf{readies}(f;g)$. The definition of composition in $\mathcal{SP}roc$ implies that there are $(s,X) \in \mathsf{readies}(f)$ and $(t,Y) \in \mathsf{readies}(g)$ with $\mathsf{fst}^*(s) = \mathsf{fst}^*(u)$, $\mathsf{snd}^*(t) = \mathsf{snd}^*(u)$, $\mathsf{snd}^*(s) = \mathsf{fst}^*(t)$ and

$$Z = \{(a,c) \in \Sigma_{A \multimap C} \mid \exists b \in \Sigma_B.((a,b) \in X \wedge (b,c) \in Y)\}.$$

For any W with $(\mathsf{snd}^*(t), W) \in \psi^\perp$, define $K \stackrel{\text{def}}{=} \{b \in \Sigma_B \mid \exists c \in W.(b,c) \in Y\}$. For any V with $(\mathsf{fst}^*(t), V) \in \varphi$, we have $Y \cap (V \times W) \neq \varnothing$ by the orthogonality condition on $\mathsf{readies}(g)$. So there is $(b,c) \in Y$ with $b \in V$ and $c \in W$. This means that $b \in K$, and so $K \cap V \neq \varnothing$. Hence $(\mathsf{fst}^*(t), K) \in \varphi^\perp$.

Finally, for any U with $(\mathsf{fst}^*(s), U) \in \theta$, the orthogonality condition on $\mathsf{readies}(f)$ means that $X \cap (U \times K) \neq \varnothing$. So there is $(a,b) \in X$ with $a \in U$ and $b \in K$. Because $b \in K$ there is $c \in W$ with $(b,c) \in Y$. Hence $(a,c) \in Z$; also $a \in U$ and $c \in W$, and so $Z \cap (U \times W) \neq \varnothing$ as required. □

This proposition yields another category of deadlock-free processes, $\mathcal{SP}roc_{D'}$. The structure of an interaction category can be defined on it, as before. Here, the definitions of the various operations on ready specifications are listed, and the conditions which they must satisfy are the same as in the previous section.

$$\begin{array}{rcl}I_{D'} & \stackrel{\text{def}}{=} & \{(*^n, \{*\}) \mid n < \omega\} \\ \bot_{D'} & \stackrel{\text{def}}{=} & I_{D'} \\ \theta \mathbin{\&} \varphi & \stackrel{\text{def}}{=} & (\{(\mathsf{inl}^*(s), \mathsf{inl}(X)) \mid (s,X) \in \theta^\perp\} \cup \{(\mathsf{inr}^*(t), \mathsf{inr}(Y)) \mid (t,Y) \in \varphi^\perp\})^\perp \\ \theta \oplus \varphi & \stackrel{\text{def}}{=} & (\theta^\perp \mathbin{\&} \varphi^\perp)^\perp \\ \circ\theta & \stackrel{\text{def}}{=} & (\{(\varepsilon, \{*\})\} \cup \{(*s, X) \mid (s,X) \in \theta^\perp\})^\perp \\ \delta\,\theta & \stackrel{\text{def}}{=} & (\{(*^n, \{*\} \cup X) \mid (\varepsilon, X) \in \theta^\perp, n \geqslant 0\} \cup \{(*^n s, X) \mid (s,X) \in \theta^\perp\})^\perp\end{array}$$

$$
\begin{array}{rcl}
\Delta\,\theta & \stackrel{\text{def}}{=} & (\{(\varepsilon, X) \mid (\varepsilon, X) \in \theta^{\perp}\} \cup \{(s, \{*\} \cup X) \mid (s{\restriction}\Sigma_A, X) \in \theta^{\perp}, s{\restriction}\Sigma_A \neq \varepsilon\})^{\perp} \\
\wp_s^n(\,\theta) & \stackrel{\text{def}}{=} & \{(\varphi_n^*(s), \varphi_n(X_1 \times \cdots \times X_n)) \mid \forall i.(\pi_i^*(s), X_i) \in \theta^{\perp}\}^{\perp} \\
\otimes_s^n(\,\theta) & \stackrel{\text{def}}{=} & \wp_s^n(\,\theta^{\perp})^{\perp} \\
!\,\theta & \stackrel{\text{def}}{=} & \&_{n \geqslant 0} \otimes_s^n \theta \\
?\,\theta & \stackrel{\text{def}}{=} & (\,!\,\theta^{\perp})^{\perp}
\end{array}
$$

The Mix rule is valid in $\mathcal{SP}roc_{D'}$. What this means is that for any objects A, B of $\mathcal{SP}roc$ and ready specifications $\alpha \in P_{D'}A$, $\beta \in P_{D'}B$, the identity morphism $\mathsf{id}_{A \otimes B}$ lifts to a morphism $(A \otimes B, \alpha \otimes \beta) \to (A \otimes B, \alpha \wp \beta)$ in $\mathcal{SP}roc_{D'}$. It is possible to check this by considering inclusions of ready specifications.

Lemma 5.18 If A, B are objects of $\mathcal{SP}roc$ and $\alpha \in P_{D'}A$, $\beta \in P_{D'}B$, then $\alpha \otimes \beta \subseteq \alpha \wp \beta$.

Proof: This is equivalent to

$$
\begin{array}{l}
\{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha, (\mathsf{snd}^*(s), V) \in \beta\}^{\perp\perp} \subseteq \\
\qquad\qquad \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha^{\perp}, (\mathsf{snd}^*(s), V) \in \beta^{\perp}\}^{\perp}
\end{array}
$$

or

$$
\begin{array}{l}
\{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha^{\perp}, (\mathsf{snd}^*(s), V) \in \beta^{\perp}\} \subseteq \\
\qquad\qquad \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \alpha, (\mathsf{snd}^*(s), V) \in \beta\}^{\perp}.
\end{array}
$$

If $(\mathsf{fst}^*(s), U) \in \alpha^{\perp}$, $(\mathsf{snd}^*(s), V) \in \beta^{\perp}$, $(\mathsf{fst}^*(s), X) \in \alpha$ and $(\mathsf{snd}^*(s), Y) \in \beta$ then there are $a \in U \cap X$ and $b \in V \cap Y$, so $(a, b) \in (U \times V) \cap (X \times Y)$. □

Lemmas 5.16 and 5.18 together imply

Proposition 5.19 $\mathcal{SP}roc_{D'}$ validates the Mix rule.

Just as for $\mathcal{SP}roc_D$, there are a few results which allow some types to be assigned automatically in $\mathcal{SP}roc_{D'}$.

Proposition 5.20 For any object A of $\mathcal{SP}roc_{pr}$, the following hold.

1. $\mathsf{RP}(A)$ is a ready specification
2. $\mathsf{RP}(A)^{\perp\perp} = \mathsf{RP}(A)$
3. $\mathsf{RP}(A)^{\perp} = \{(s, \{x \in \Sigma_A \mid sx \in S_A\}) \mid s \in S_A\}$.

Proof:

1. Because A is progressive, there is $a \in \Sigma_A$ such that $a \in S_A$. Hence $(\varepsilon, \{a\})$ is a ready pair and so $\mathsf{RP}(A) \neq \varnothing$. If $(s, X) \in \mathsf{RP}(A)$ and $a \in X$, then $sa \in S_A$. Let $Y = \{x \in \Sigma_A \mid sax \in S_A\}$. Because A is progressive, $Y \neq \varnothing$, so $(sa, Y) \in \mathsf{RP}(A)$. If $(sa, Y) \in \mathsf{RP}(A)$ then $(s, \{a\}) \in \mathsf{RP}(A)$. Thus $\mathsf{RP}(A)$ satisfies the conditions necessary to be a ready specification.

2. $\mathsf{RP}(A)^{\perp\perp}$ is a set of ready pairs, so $\mathsf{RP}(A)^{\perp\perp} \subseteq \mathsf{RP}(A)$. Also $\mathsf{RP}(A) \subseteq \mathsf{RP}(A)^{\perp\perp}$ for general reasons.

3. It is clear that $\{(s, \{x \in \Sigma_A \mid sx \in S_A\}) \mid s \in S_A\} \perp \mathsf{RP}(A)$. Conversely, suppose that $(s, X) \perp \mathsf{RP}(A)$. Because $(s, \{x\}) \in \mathsf{RP}(A)$ for every $x \in \Sigma_A$ such that $sx \in S_A$, the definition of orthogonality means that $x \in X$ for each such x. Hence $X = \{x \in \Sigma_A \mid sx \in S_A\}$ as claimed.

□

Corollary 5.21 If A is an object of $\mathcal{SP}roc_{pr}$, and is such that for each $s \in S_A$ there is a unique $x \in \Sigma_A$ such that $sx \in S_A$, then $\mathsf{RP}(A)^{\perp} = \mathsf{RP}(A)$. The converse is also true.

Lemma 5.22 For any A and B,

$$\begin{aligned} \mathsf{RP}(A) \parr \mathsf{RP}(B) &= \mathsf{RP}(A \parr B) \\ \mathsf{RP}(A)^{\perp} \parr \mathsf{RP}(B)^{\perp} &= \mathsf{RP}(A \parr B)^{\perp}. \end{aligned}$$

Proof: For the first part,

$$\begin{aligned} &\mathsf{RP}(A) \parr \mathsf{RP}(B) \\ &= \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \mathsf{RP}(A)^{\perp}, (\mathsf{snd}^*(s), V) \in \mathsf{RP}(B)^{\perp}\}^{\perp} \\ &= \{(s, U \times V) \mid U = \{a \in \Sigma_A \mid \mathsf{fst}^*(s)a \in S_A\}, V = \{b \in \Sigma_B \mid \mathsf{snd}^*(s)b \in S_B\}\}^{\perp} \\ &= \{(s, \{(a,b) \in \Sigma_{A \parr B} \mid s(a,b) \in S_{A \parr B}\}) \mid s \in S_{A \parr B}\}^{\perp} \\ &= \mathsf{RP}(A \parr B)^{\perp\perp} \\ &= \mathsf{RP}(A \parr B). \end{aligned}$$

For the second part,

$$\begin{aligned} &\mathsf{RP}(A)^{\perp} \parr \mathsf{RP}(B)^{\perp} \\ &\qquad = \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \mathsf{RP}(A), (\mathsf{snd}^*(s), V) \in \mathsf{RP}(B)\}^{\perp}. \end{aligned}$$

For each $a \in \Sigma_A$ and $b \in \Sigma_B$ such that $\mathsf{fst}^*(s)a \in S_A$ and $\mathsf{snd}^*(s)b \in S_B$, we have $(\mathsf{fst}^*(s), \{a\}) \in \mathsf{RP}(A)$ and $(\mathsf{snd}^*(s), \{b\}) \in \mathsf{RP}(B)$. So for every such a and b,

$$(s, \{(a,b)\}) \in \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \mathsf{RP}(A), (\mathsf{snd}^*(s), V) \in \mathsf{RP}(B)\}.$$

The only ready pair with trace s and orthogonal to all of these is $(s, \{(a,b) \mid s(a,b) \in S_{A \parr B}\})$. □

Corollary 5.23 For any A and B,

$$\begin{aligned} \mathsf{RP}(A) \parr \mathsf{RP}(B) &= \mathsf{RP}(A) \otimes \mathsf{RP}(B) \\ \mathsf{RP}(A)^\perp \parr \mathsf{RP}(B)^\perp &= \mathsf{RP}(A)^\perp \otimes \mathsf{RP}(B)^\perp. \end{aligned}$$

Proposition 5.24 If $P : A_1 \parr \cdots \parr A_n$ in $\mathcal{SP}roc$, the A_i are progressive and $P\downarrow$, then $P : (A_1, \mathsf{RP}(A_1)) \parr \cdots \parr (A_n, \mathsf{RP}(A_n))$ in $\mathcal{SP}roc_{D'}$.

Proof: It is immediate that if $P : A$ in $\mathcal{SP}roc$, A is progressive and P is deadlock-free, then $P : (A, \mathsf{RP}(A))$ in $\mathcal{SP}roc_{D'}$. By Lemma 5.22, the result can be obtained by applying this observation to the type $A_1 \parr \cdots \parr A_n$. □

This result means that any deadlock-free process in $\mathcal{SP}roc$ can be given a type in $\mathcal{SP}roc_{D'}$. But this does not mean that any two deadlock-free processes which are composable in $\mathcal{SP}roc$ are also composable in $\mathcal{SP}roc_{D'}$. For example, consider $\mathcal{SP}roc$ objects A, B, C with $\Sigma_A = \{a\}$, $\Sigma_B = \{b, b'\}$, $\Sigma_C = \{c\}$ and unrestricted safety specifications. Define processes $f : A \to B$ and $g : B \to C$ by $f = (a, b) : f$ and $g = (b', c) : g$. The automatically generated $\mathcal{SP}roc_{D'}$ typings are

$$\begin{aligned} f &: (A, \mathsf{RP}(A)) \parr (B, \mathsf{RP}(B)) \\ g &: (B, \mathsf{RP}(B)) \parr (C, \mathsf{RP}(C)) \end{aligned}$$

which in terms of morphisms give

$$\begin{aligned} f &: (A, \mathsf{RP}(A))^\perp \to (B, \mathsf{RP}(B)) \\ g &: (B, \mathsf{RP}(B))^\perp \to (C, \mathsf{RP}(C)). \end{aligned}$$

But of course these morphisms are not composable, because $\mathsf{RP}(B)^\perp \neq \mathsf{RP}(B)$. In fact, in this example, $\mathsf{RP}(B) = \{(s, \{b, b'\}), (s, \{b\}), (s, \{b'\}) \mid s \in S_B\}$ and $\mathsf{RP}(B)^\perp = \{(s, \{b, b'\}) \mid s \in S_B\}$. The only case in which this automatic assignment of types can be used effectively is when $\mathsf{RP}(B)^\perp = \mathsf{RP}(B)$, which happens precisely in the circumstances described by Corollary 5.21.

5.4.3 Equivalence of D and D′

As indicated earlier, the specification structures D and D' are equivalent, despite their apparently rather different approaches to specifying deadlock-freedom. Equivalence is meant in the strongest possible sense: for each object A of $\mathcal{SP}roc_{pr}$ there is a bijection between $P_D A$ and $P_{D'} A$, and this bijection preserves all the operations on properties exactly.

The first lemma shows that the two specification structures are based on the same idea of when processes can communicate. Here and subsequently there are two versions of

orthgonality and all operations on properties, for which the same notation is used; this should not cause confusion because of the convention of using $\theta, \varphi, \ldots$ for ready specifications and $U, V, \ldots$ for sets of processes.

Lemma 5.25 If $P, Q \in \mathsf{Proc}(A)$, then $P \perp Q \iff \mathsf{readies}(P) \perp \mathsf{readies}(Q)$.

Proof: Suppose $P \perp Q$, $(s,X) \in \mathsf{readies}(P)$ and $(s,Y) \in \mathsf{readies}(Q)$. So $P \stackrel{s}{\longrightarrow}{}^* P'$ and $Q \stackrel{s}{\longrightarrow}{}^* Q'$, and orthogonality of P and Q implies that there is an action a such that $P' \stackrel{a}{\longrightarrow} P''$ and $Q' \stackrel{a}{\longrightarrow} Q''$. This means that $a \in \mathsf{initials}(P') = X$ and $a \in \mathsf{initials}(Q') = Y$, so $(s,X) \perp (s,Y)$.

Conversely suppose $\mathsf{readies}(P) \perp \mathsf{readies}(Q)$, $P \stackrel{s}{\longrightarrow}{}^* P'$ and $Q \stackrel{s}{\longrightarrow}{}^* Q'$. Then $(s, \mathsf{initials}(P')) \in \mathsf{readies}(P)$ and $(s, \mathsf{initials}(Q')) \in \mathsf{readies}(Q)$, and this implies

$$\mathsf{initials}(P') \cap \mathsf{initials}(Q') \neq \varnothing.$$

Thus there is an action a such that $P' \stackrel{a}{\longrightarrow} P''$ and $Q' \stackrel{a}{\longrightarrow} Q''$, so $P \perp Q$. □

Now define, for $U \in P_D A$ and $\theta \in P_{D'} A$,

$$\begin{aligned} F(\theta) &\stackrel{\text{def}}{=} \{P \in \mathsf{Proc}(A) \mid \mathsf{readies}(P) \subseteq \theta\} \\ G(U) &\stackrel{\text{def}}{=} \bigcup \{\mathsf{readies}(P) \mid P \in U\}. \end{aligned}$$

This gives a correspondence between properties in the two specification structures. There is also a connection between the two satisfaction relations, which is very easy to establish.

Lemma 5.26

$$\begin{aligned} \mathsf{readies}(P) \subseteq \theta \quad &\iff \quad P \in F(\theta) \\ P \in U \quad &\Rightarrow \quad \mathsf{readies}(P) \subseteq G(U). \end{aligned}$$

The other half of the second implication will follow later. First, it is essential to prove that F and G are mutually inverse.

Proposition 5.27 If $\theta \in P_{D'} A$ then $GF(\theta) = \theta$.

Proof: If $(s,X) \in GF(\theta)$ there is $P \in F(\theta)$ with $(s,X) \in \mathsf{readies}(P)$. This means that $(s,X) \in \theta$, because $P \in F(\theta) \Rightarrow \mathsf{readies}(P) \subseteq \theta$. Hence $GF(\theta) \subseteq \theta$.

If $(s,X) \in \theta$ then establishing $(s,X) \in GF(\theta)$ requires $P \in F(\theta)$ such that $(s,X) \in \mathsf{readies}(P)$. This means finding P with $\mathsf{readies}(P) \subseteq \theta$ and $(s,X) \in \mathsf{readies}(P)$. Because $\theta = \theta^{\perp\perp}$, $(t, \{a \in \Sigma_A \mid ta \in S_A\}) \in \theta$ for any trace $t \in S_A$. So a suitable P is the process constructed from max_A by only allowing the actions in X after the trace s. □

Proposition 5.28 If $U \in P_D A$ then $FG(U) = U$.

Proof: If $P \in U$ then $\mathsf{readies}(P) \subseteq G(U)$, so $P \in FG(U)$. Hence $U \subseteq FG(U)$.

If $P \in FG(U)$ then $\mathsf{readies}(P) \subseteq G(U)$. So for any $(s, X) \in \mathsf{readies}(P)$ there is $Q \in U$ such that $(s, X) \in \mathsf{readies}(Q)$. If $R \in U^\perp$ and $(t, Y) \in \mathsf{readies}(R)$, this means that $(s, X) \perp (t, Y)$. Hence $P \perp R$, i.e. $P \in U^{\perp\perp} = U$. Thus $FG(U) \subseteq U$. □

Corollary 5.29 $\mathsf{readies}(P) \subseteq G(U) \Rightarrow P \in U$.

Proof: If $\mathsf{readies}(P) \subseteq G(U)$ then $P \in FG(U) = U$. □

So far, it has not been proved that $F(\theta)$ and $G(U)$ are $^{\perp\perp}$-invariant. This can be done at the same time as proving that F and G commute with $(-)^\perp$. First of all, it is clear that for any θ and U, $F(\theta^\perp) \perp F(\theta)$ and $G(U^\perp) \perp G(U)$. This proves

Lemma 5.30 $F(\theta^\perp) \subseteq F(\theta)^\perp$ and $G(U^\perp) \subseteq G(U)^\perp$.

The reverse inclusions require more work.

Lemma 5.31 If $\theta \in P_{D'} A$, there are processes $P_1, \ldots, P_n$ such that

$$\mathsf{readies}(P_1) \cup \ldots \cup \mathsf{readies}(P_n) = \theta.$$

Proof: Define a labelled transition system whose states are the ready pairs in θ, with transitions defined by

$$\frac{a \in X}{(s, X) \xrightarrow{a} (sa, Y).}$$

Because $\theta = \theta^{\perp\perp}$, $(s, \{x \in S_A \mid sa \in S_A\}) \in \theta$ for each $s \in S_A$. So for any pair $(sa, Y) \in \theta$ there is the transition $(s, \{x \in S_A \mid sa \in S_A\}) \xrightarrow{a} (sa, Y)$, which means that every state is reachable from $(\varepsilon, \{a \in \Sigma_A \mid a \in S_A\})$, except for any (ε, X) with $X \neq \{a \in \Sigma_A \mid a \in S_A\}$. Furthermore, because each transition increases the length of the trace, there are no cycles except possibly in the case of two distinct transitions between the same two states. In this latter case, the target state can be separated into two states. Overall, this means that the states (ε, X) can be taken as the processes $P_1, \ldots, P_n$. □

Proposition 5.32 $F(\theta^\perp) = F(\theta)^\perp$.

Proof: It is enough to prove $F(\theta)^\perp \subseteq F(\theta^\perp)$. If $P \in F(\theta)^\perp$ then $P \perp F(\theta)$. Let $Q_1, \ldots, Q_n$ be such that $\mathsf{readies}(Q_1) \cup \ldots \cup \mathsf{readies}(Q_n) = \theta$, so that $P \perp Q_i$ for each i implies $\mathsf{readies}(P) \perp \theta$ and hence $\mathsf{readies}(P) \subseteq \theta^\perp$. Thus $P \in F(\theta^\perp)$. □

Proposition 5.33 $G(U^\perp) = G(U)^\perp$.

Proof: To establish $G(U)^\perp \subseteq G(U^\perp)$, which is sufficient, suppose that $(s,X) \in G(U)^\perp$. For every $(t,Y) \in G(U)$, $(s,X) \perp (t,Y)$. So whenever $P \in U$ and $(t,Y) \in \mathsf{readies}(P)$, $(s,X) \perp (t,Y)$. It is possible to find $Q \in U^\perp$ such that $(s,X) \in \mathsf{readies}(Q)$, for example by starting with max_A and removing transitions from a state reached by a trace of s to obtain a ready set of X at that point. Hence $(s,X) \in G(U^\perp)$. □

Corollary 5.34 $F(\theta) = F(\theta)^{\perp\perp}$ and $G(U) = G(U)^{\perp\perp}$.

Finally, it can be proved that F and G preserve $\otimes$ and $\parr$.

Proposition 5.35 If $\theta \in P_{D'}A$ and $\varphi \in P_{D'}B$ then $F(\theta \parr \varphi) = F(\theta) \parr F(\varphi)$.

Proof: It is enough to show that $F(\theta^\perp \parr \varphi^\perp) = F(\theta^\perp) \parr F(\varphi^\perp)$. Now,

$$\begin{aligned} F(\theta^\perp) \parr F(\varphi^\perp) &= F(\theta)^\perp \parr F(\varphi)^\perp \\ &= (F(\theta) \otimes F(\varphi))^\perp \end{aligned}$$

so

$$F[\{(s, A \times B) \mid (\mathsf{fst}^*(s), A) \in \theta, (\mathsf{snd}^*(s), B) \in \varphi\}^\perp] = \{P \otimes Q \mid P \in F(\theta), Q \in F(\varphi)\}^\perp$$

is sufficient. If

$$R \in F[\{(s, A \times B) \mid (\mathsf{fst}^*(s), A) \in \theta, (\mathsf{snd}^*(s), B) \in \varphi\}^\perp]$$

then

$$\mathsf{readies}(R) \perp \{(s, A \times B) \mid (\mathsf{fst}^*(s), A) \in \theta, (\mathsf{snd}^*(s), B) \in \varphi\}.$$

For any s, A, B with $(\mathsf{fst}^*(s), A) \in \theta$ and $(\mathsf{snd}^*(s), B) \in \varphi$, $(s,X) \in \mathsf{readies}(R)$ implies that there is $(a,b) \in X$ with $a \in A$ and $b \in B$. So if $\mathsf{readies}(P) \in \theta$ and $\mathsf{readies}(Q) \in \varphi$, $R \perp (P \otimes Q)$.

For the converse, the same argument can be run backwards. □

Corollary 5.36 $F(\theta \otimes \varphi) = F(\theta) \otimes F(\varphi)$.

Proof: This follows from the fact that F preserves $\parr$ and $(-)^\perp$, and duality of $\otimes$ and $\parr$. □

Corollary 5.37 $G(U \parr V) = G(U) \parr G(V)$ and $G(U \otimes V) = G(U) \otimes G(V)$.

Proof:

$$\begin{aligned} G(U \parr V) &= G(FG(U) \parr FG(V)) \\ &= GF(G(U) \parr G(V)) \\ &= G(U) \parr G(V). \end{aligned}$$

Again, $G(U \otimes V) = G(U) \otimes G(V)$ follows easily. □

The results of this section show that the specification structures D and D' are two different ways of looking at the same category of deadlock-free processes. The final observation, allowing all of the examples to be translated between the two views, is that for any object A, $F(\mathsf{RP}(A)) = M_A$. The calculations showing that $\mathcal{SP}roc_D$ and $\mathcal{SP}roc_{D'}$ are not compact closed are based on exactly the same example but represented in two different ways. The proof that $\mathcal{SP}roc_{D'}$ validates Mix could be formulated in terms of sets of processes, but is probably simpler with ready specifications. On the other hand, the definitions of the delay functors and the exponentials are rather simpler for sets of processes. Similarly, the proof that $\oplus$ and $\&$ do not coincide can be given for D'. The next example which will be presented, an analysis of deadlock-freedom of dataflow networks, is very naturally expressed in terms of ready specifications because the key property of *receptivity* fits the idea of a maximal ready set.

It is possible that $\mathsf{readies}(P) = \mathsf{readies}(Q)$ with $P \neq Q$, and in this case the processes P and Q satisfy exactly the same ready specifications. However, it is not possible for distinct processes to be contained in exactly the same sets of processes. The resolution of this apparent conflict is the use of only $^{\perp\perp}$-invariant sets of processes as properties in the specification structure D.

Proposition 5.38 If $P \in U$ and $\mathsf{readies}(P) = \mathsf{readies}(Q)$ then $Q \in U^{\perp\perp}$.

Proof: If $R \in U^{\perp}$ then $\mathsf{readies}(R) \perp \mathsf{readies}(P)$. So $\mathsf{readies}(R) \perp \mathsf{readies}(Q)$, which means that $Q \in U^{\perp\perp}$. □

Defining two processes to be *ready-equivalent* if they have the same readies, this result says that a $^{\perp\perp}$-invariant set of processes must be the union of a collection of ready-equivalence classes. So membership of $^{\perp\perp}$-invariant sets cannot distinguish processes more finely than ready-equivalence.

5.4.4 Example: Dataflow

The view being promoted in this section is that $\mathcal{SP}roc_{D'}$ gives a type system for compositional verification of deadlock-freedom. This type system is used by starting with processes which have composable types in $\mathcal{SP}roc$, and trying to construct ready specifications for those types which allow the processes to be typed in $\mathcal{SP}roc_{D'}$. If the

processes and the ready specifications are suitable, the types in $\mathcal{SP}roc_{D'}$ should be composable, showing that the processes can be connected together without causing deadlocks. An interesting example of the use of $\mathcal{SP}roc_{D'}$ can be found by returning to synchronous dataflow networks. Clearly deadlock-freedom is a property which might be of interest when discussing dataflow. Forgetting about types for deadlock-freedom for a moment, consider the sort of argument which might be used to show that a network built in $\mathcal{SP}roc$ does not deadlock. For non-cyclic networks, it is possible to argue that because each node is always prepared to accept any input, deadlocks cannot occur if an output is always connected to an input (never to another output). Formalising this observation, say that a morphism $f : A \to B$ in $\mathcal{SP}roc$ is *receptive* if for any g such that $f \xrightarrow{s}{}^{*} g$ and any $a \in \Sigma_A$ such that $sa \in S_A$, there is $b \in \Sigma_B$ and a process h such that $g \xrightarrow{(a,b)} h$. It follows that if A is progressive and $f : A \to B$ is receptive then f is deadlock-free. It is also easy to show that receptivity is preserved by composition, and identity morphisms are receptive. Thus there is a subcategory of $\mathcal{SP}roc$ consisting of progressive types and receptive morphisms, and this subcategory is a crude approximation to an interaction category of deadlock-free processes. It is only an approximation for several reasons. Receptivity, although natural for dataflow nodes, is a very strong condition to impose on processes in general, and there are many more varieties of non-deadlocking behaviour than can be described in terms of receptivity alone. Furthermore, the receptive subcategory is not $*$-autonomous: the receptivity condition applies only to input ports, so if input and output are interchanged by $(-)^{\perp}$, the condition is not preserved. However, it does seem that as far as dataflow networks are concerned, a useful amount of work can be carried out inside the receptive subcategory. This does not extend to forming arbitrary feedback loops; as seen in Chapter 4, extra conditions on the precise form of a loop are necessary. This point will be discussed later.

This special case of compositional reasoning about deadlock-freedom is subsumed by the general specification structure approach. Working in $\mathcal{SP}roc_{D'}$, consider just two of the possible ready specifications over each object X of $\mathcal{SP}roc$: $\mathsf{RP}(X)$ and $\mathsf{RP}(X)^{\perp}$. The key result relating the receptive subcategory of $\mathcal{SP}roc$ to $\mathcal{SP}roc_{D'}$ is the following.

Proposition 5.39 Suppose X and Y are progressive objects of $\mathcal{SP}roc$, and $f : X \to Y$ is receptive. Then $f : (X, \mathsf{RP}(X)) \to (Y, \mathsf{RP}(Y))$ in $\mathcal{SP}roc_{D'}$.

Proof: We need to check that $\mathsf{RP}(X)\{f\}\mathsf{RP}(Y)$, i.e. that

$$\mathsf{readies}(f) \subseteq \mathsf{RP}(X)^{\perp} \parr \mathsf{RP}(Y)$$

or equivalently that

$$\mathsf{readies}(f) \perp \{(s, U \times V) \mid (\mathsf{fst}^*(s), U) \in \mathsf{RP}(X), (\mathsf{snd}^*(s), V) \in \mathsf{RP}(Y)^{\perp}\}.$$

Receptivity of f means that if $(s, A) \in \mathsf{readies}(f)$ then for any $x \in \Sigma_X$ such that $\mathsf{fst}^*(s)x \in S_X$, there is $y \in \Sigma_Y$ with $(x, y) \in A$. So for any U, V with $(\mathsf{fst}^*(s), U) \in \mathsf{RP}(X)$ and $(\mathsf{snd}^*(s), V) \in \mathsf{RP}(Y)^\perp$ (which means $U \subseteq \{x \in \Sigma_X \mid \mathsf{fst}^*(s)x \in S_X\}$ and $V = \{y \in \Sigma_Y \mid \mathsf{snd}^*(s)y \in S_Y\}$) there is $(x, y) \in A$ such that $(x, y) \in U \times V$. Hence $(s, A) \perp (s, U \times V)$. □

This result can also be applied to the case of two or more inputs. If $f : A \otimes B \to C$ in $\mathcal{SP}roc$ (which is the form of morphism used in Chapter 4 for a node with two inputs) then $f : (A \otimes B, \mathsf{RP}(A \otimes B)) \to (C, \mathsf{RP}(C))$ in $\mathcal{SP}roc_D$. Equivalently, since $\mathcal{SP}roc$ is compact closed, $f : (A \parr B, \mathsf{RP}(A \parr B)) \to (C, \mathsf{RP}(C))$. Because $\mathsf{RP}(A \parr B) = \mathsf{RP}(A) \parr \mathsf{RP}(B) = \mathsf{RP}(A) \otimes \mathsf{RP}(B)$, this can also be written as

$$f : (A, \mathsf{RP}(A)) \otimes (B, \mathsf{RP}(B)) \to (C, \mathsf{RP}(C)).$$

All receptive dataflow nodes can now be typed in $\mathcal{SP}roc_{D'}$ just by using ready specifications of the form $\mathsf{RP}(X)$ and $\mathsf{RP}(X)^\perp$. This means that all the non-cyclic connections between nodes which were supported in $\mathcal{SP}roc$ can also be made in $\mathcal{SP}roc_{D'}$. What about cycles? Arbitrary cyclic connections cannot be made without causing deadlocks. Given a network with an input of type X and an output of type X (and other inputs and outputs as well), what might happen if the cycle is completed? The undesirable case is that in which the output on X is always different from the input received on X at the same instant, because then no behaviour of the network is able to satisfy the constraint that the X input stream and the X output stream are the same. The analysis of dataflow in Chapter 4 introduced the condition that cycles can only be formed if the output at any step depends not on the input received at that step but only on the current state. It is now possible to formulate a slightly different condition which ensures that there is always an action in which the input and the output are the same; and, in fact, this condition can be expressed more easily in terms of ready sets than sets of processes.

Suppose that $P : \Gamma \parr X \parr X^\perp$ in $\mathcal{SP}roc$. There is a process $\overline{P} : \Gamma$ in $\mathcal{SP}roc$ which is obtained by connecting the X and $X^\perp$ ports of P together. If $P : (\Gamma, U) \parr (X, V) \parr (X^\perp, V^\perp)$ in $\mathcal{SP}roc_D$, or $P : (\Gamma, \theta) \parr (X, \varphi) \parr (X^\perp, \varphi^\perp)$ in $\mathcal{SP}roc_{D'}$, then $\overline{P}$ may not have a type in $\mathcal{SP}roc_D$ or $\mathcal{SP}roc_{D'}$. What is required is a sufficient condition for $\overline{P} : (\Gamma, U)$ in $\mathcal{SP}roc_D$. Clearly a necessary condition is that connecting the X and $X^\perp$ ports should not cause P to deadlock. Defining $\hat{P} : X \parr X^\perp$ by

$$\frac{P \xrightarrow{(\bar{a}, x, y)} Q}{\hat{P} \xrightarrow{(x,y)} \hat{Q}}$$

this condition can be expressed as

$$\hat{P} \perp \mathsf{id}_X$$

in the sets of processes framework. It guarantees that P is always able to perform actions in which the X and $X^\perp$ components are the same, but says nothing about whether P can do this while still satisfying the constraint expressed by the specification U or θ. The following condition is enough to guarantee that $\overline{P} : (\Gamma, \theta)$ in $\mathcal{SP}roc_{D'}$.

- For every $(s, A) \in \mathsf{readies}(P)$ such that $\pi^*_X(s) = \pi^*_{X^\perp}(s)$, and for every action $(\bar{a}, x, y) \in A$, there must be $z \in \Sigma_X$ such that $(\bar{a}, z, z) \in A$.

If this condition is written $\mathsf{cycle}(P)$ then the following proof rule summarises the preceeding discussion.

$$\frac{P : (\Gamma, \theta) \parr (X, \varphi) \parr (X^\perp, \varphi^\perp) \quad \mathsf{cycle}(P)}{\overline{P} : (\Gamma, \theta)}$$

In the dataflow example, it is natural to impose the condition formulated in Chapter 4 on the formation of cycles. If P is a network in which the $X^\perp$ port is an output whose possible actions do not depend on the actions performed in the X and Γ ports, then because the input X is receptive and can accept any action, it can always accept the action which the $X^\perp$ port outputs at the same instant. This means that the condition for the formation of $\overline{P}$ is satisfied.

Another approach might be to use a ready specification θ over X with the property that $\theta \parr \theta^\perp = \theta \otimes \theta^\perp$. Then the type of P could be converted into a type suitable for connection with the application morphism $\mathsf{Ap}_{X,X^\perp}$, and there would be no need to use a separate proof rule. But this approach is less general. It is difficult to construct such a θ without choosing in advance a particular action which will always be available in both ports. For example, consider the object X with $\Sigma_X = \{a, b\}$ and an unrestricted safety specification. There are four possibilities for a $^{\perp\perp}$-invariant ready specification θ over X (in the following discussion, traces are omitted and only ready *sets* are shown; also ab is written for (a, b)).

- $\theta = \mathsf{RP}(X) = \{\{a, b\}, \{a\}, \{b\}\}$
- $\theta = \mathsf{RP}(X)^\perp = \{\{a, b\}\}$
- $\theta = \theta^\perp = \{\{a, b\}, \{a\}\}$
- $\theta = \theta^\perp = \{\{a, b\}, \{b\}\}$.

In the first case,

$$\begin{aligned}
\theta \parr \theta^\perp &= \{\{aa, ab, ba, bb\}, \{aa, ba\}, \{ab, bb\}\}^\perp \\
&= \{\{aa, ab, ba, bb\}, \\
&\quad \{aa, ab, ba\}, \{aa, ab, bb\}, \{aa, ba, bb\}, \{ab, ba, bb\}, \\
&\quad \{aa, ab\}, \{aa, bb\}, \{ba, ab\}, \{ba, bb\}\}
\end{aligned}$$

$$\begin{aligned}
\theta \otimes \theta^{\perp} &= \theta^{\perp} \parr \theta^{\perp} \\
&= \{\{aa, ab, ba, bb\}, \{aa, ab\}, \{ba, bb\}\}^{\perp\perp} \\
&= \{\{aa, ab, ba, bb\}, \\
&\quad \{aa, ab, ba\}, \{aa, ab, bb\}, \{aa, ba, bb\}, \{ab, ba, bb\}, \\
&\quad \{aa, ba\}, \{aa, bb\}, \{ab, ba\}, \{ab, bb\}\}^{\perp} \\
&= \{\{aa, ab, ba, bb\}, \\
&\quad \{aa, ab, ba\}, \{aa, ab, bb\}, \{aa, ba, bb\}, \{ab, ba, bb\}, \\
&\quad \{aa, ab\}, \{ba, bb\}\}
\end{aligned}$$

and these ready specifications are different. The appearance of the ready set $\{ba, ab\}$ in $\theta \parr \theta^{\perp}$ clearly shows that a process may satisfy this ready specification but still deadlock when the X ports are connected together. The second case is similar.

In the third case,

$$\begin{aligned}
\theta \parr \theta^{\perp} &= \theta \parr \theta \\
&= \{\{aa, ab, ba, bb\}, \{aa, ba\}, \{aa, ab\}, \{aa\}\}^{\perp} \\
&= \{\{aa, ab, ba, bb\}, \\
&\quad \{aa, ab, ba\}, \{aa, ab, bb\}, \{aa, ba, bb\}, \\
&\quad \{aa, ab\}, \{aa, ba\}, \{aa, bb\}, \{aa\}\} \\
\theta \otimes \theta^{\perp} &= \theta^{\perp} \parr \theta^{\perp} \\
&= \theta \parr \theta^{\perp} \\
&= \{\{aa, ab, ba, bb\}, \\
&\quad \{aa, ab, ba\}, \{aa, ab, bb\}, \{aa, ba, bb\}, \\
&\quad \{aa, ab\}, \{aa, ba\}, \{aa, bb\}, \{aa\}\} \\
&= \theta \parr \theta.
\end{aligned}$$

This choice of θ would give a suitable type for cycle-formation, but to satisfy it a process must always be able to do the (a, a) action. The fourth case is symmetrical, requiring a process always to offer (b, b). There are of course many processes which could be cyclically connected without deadlock but which do not satisfy these types.

5.5 Asynchronous Deadlock-Freedom

The category of synchronous deadlock-free processes allows some synchronous problems to be analysed, but there are also asynchronous examples such as the scheduler for

which deadlock-freedom is of equal interest. The obvious approach to reasoning about asynchronous deadlock-freedom is to use the delay operators of $\mathcal{SP}roc_D$ to represent asynchrony, and then proceed as before. However, when this is done the only deadlock-free behaviours which the types can guarantee to exist are those in which all the processes in the system delay. Producing a meaningful analysis of deadlock-freedom for asynchronous processes requires a version of the specification structure D over $\mathcal{ASP}roc$. Since there are two formulations of the specification structure over $\mathcal{SP}roc$, there is a choice of approach in the asynchronous case. It turns out that the definitions in terms of sets of processes are easier to adapt. Before starting to do this, there are two technical problems which need to be addressed.

The first is to do with *divergence*, or *livelock*. Suppose there are morphisms $f : A \to B$ and $g : B \to C$, each of which runs forever but only does actions in B. Then even if f and g do not deadlock each other, the result of composing them is a morphism which does no observable actions at all—under observation equivalence this is the same as the nil process, which is deadlocked. This shows that when dealing with asynchronous processes, it is insufficient simply to guarantee that processes can always communicate with each other when composed. The second technical problem is that because convergence of a process will mean the ability to continue performing observable actions, there are no convergent processes of type I in $\mathcal{ASP}roc$, and hence no properties over I. This means that the asynchronous deadlock-free category will have no tensor unit; in order to retain the ability to use the $*$-autonomous structure in calculations, a different object will have to be used instead.

The first problem can be solved by making use of Hoare's solution of a similar problem [Hoa85]. He considers processes with one input and one output, which can be connected together in sequence. This is actually quite close to the categorical view in some ways: these processes have the "shape" of morphisms and can be composed, although there are no identity processes. More to the point, he is interested in conditions on processes which ensure that connecting them together does not lead to divergence. Recasting the question into the categorical framework, if $f : A \to B$ and $g : B \to C$, what is the condition that $f\,;g$ does not diverge? Hoare's solution is to specify that f should be *left-guarded* or g *right-guarded*. Left-guardedness means that every infinite trace of f should contain infinitely many observable actions in A; similarly, right-guardedness means that every infinite trace of g should contain infinitely many observable actions in C. If f is left-guarded it has no infinite behaviours which only involve actions in B, so no matter what g does there can be no divergent behaviour of $f\,;g$. Symmetrically, if g is right-guarded then $f\,;g$ does not diverge. If a process is to be a morphism in a category, it must be composable both on the left and on the right; this means that it needs to be both left-guarded and right-guarded. Requiring that a morphism

be both left- and right-guarded, i.e. that every infinite trace must contain infinitely many observable actions in both ports, amounts to a specification of *fairness*. What is needed for deadlock-freedom is a category in which all the morphisms are fair in this sense. This issue only arises in the asynchronous case, since in a synchronous category it is impossible for an infinite trace of a process to have anything other than an infinite sequence of actions in each port.

5.5.1 The Category $\mathcal{FP}roc$

The category $\mathcal{FP}roc$ (fair processes) has objects $A = (\Sigma_A, \tau_A, S_A, F_A)$. The first three components of an object are exactly as in $\mathcal{ASP}roc$. The fourth, F_A, is a subset of $\mathsf{ObAct}(A)^\omega$ such that all finite prefixes of any trace in F_A are in S_A. The interaction category operations on objects are defined as in $\mathcal{ASP}roc$, with the addition that

$$\begin{aligned} F_{A^\perp} &\stackrel{\text{def}}{=} F_A \\ F_{A\otimes B} &\stackrel{\text{def}}{=} \{s \in \mathsf{ObAct}(A \otimes B)^\omega \mid s{\restriction}A \in F_A, s{\restriction}B \in F_B\} \\ F_{A\oplus B} &\stackrel{\text{def}}{=} \{l\mathsf{inl}^\omega(s) \mid s \in F_A\} \cup \{r\mathsf{inr}^\omega(t) \mid t \in F_B\} \\ F_{\circ A} &\stackrel{\text{def}}{=} \{*s \mid s \in F_A\}. \end{aligned}$$

A process in $\mathcal{FP}roc$ is almost the same as a process in $\mathcal{ASP}roc$, except that there now has to be a way of specifying which of the infinite traces of a synchronisation tree are to be considered as actual infinite behaviours of the process. This is done by working with pairs (P, T_P) in which P is an $\mathcal{ASP}roc$ process and $\varnothing \neq T_P \subseteq \mathsf{infobtraces}(P)$. Only the infinite traces in T_P are viewed as behaviours of P, even though the tree P may have many other infinite traces. There is a condition for this specification of valid infinite traces to be compatible with transitions: if $P \stackrel{a}{\Longrightarrow} Q$ then $T_P = \{as \mid s \in T_Q\}$.

A process of type A in $\mathcal{FP}roc$ is a pair (P, T_P) as above, in which P is a process of type (Σ_A, τ_A, S_A) in $\mathcal{ASP}roc$, and $T_P \subseteq F_A$. Equivalence of processes is defined by

$$(P, T_P) = (Q, T_Q) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad (P \approx Q) \wedge (T_P = T_Q).$$

As usual, a morphism from A to B is a process of type $A \multimap B$. The identity morphism on A in $\mathcal{FP}roc$ is $(\mathsf{id}_A, F_{A\multimap A})$ where id_A is the identity on (Σ_A, τ_A, S_A) in $\mathcal{ASP}roc$. It will often be convenient to refer to $\mathcal{FP}roc$ processes by their first components, and just consider the second components as extra information when necessary; thus the process (P, T_P) may simply be written P.

For composition, if $(f, T_f) : A \to B$ and $(g, T_g) : B \to C$ then $(f, T_f)\,;(g, T_g) \stackrel{\text{def}}{=} (f\,;g, T_{f;g})$ where

$$T_{f;g} \stackrel{\text{def}}{=} \{s \in \mathsf{infobtraces}(f\,;g) \mid \exists t \in T_f, u \in T_g.[t{\restriction}A = s{\restriction}A, t{\restriction}B = u{\restriction}B, u{\restriction}C = s{\restriction}C]\}.$$

It is straightforward to check that if $T_f \subseteq F_{A \multimap B}$ and $T_g \subseteq F_{B \multimap C}$ then $T_{f;g} \subseteq F_{A \multimap C}$. The functorial action of $\otimes$ is defined by $(f, T_f) \otimes (g, T_g) \stackrel{\text{def}}{=} (f \otimes g, T_{f \otimes g})$ where, for $f : A \to C$ and $g : B \to D$,

$$T_{f \otimes g} \stackrel{\text{def}}{=} \{s \in \mathsf{infobtraces}(f \otimes g) \mid s\restriction(A, C) \in T_f, s\restriction(B, D) \in T_g, s \in F_{A \otimes B \multimap C \otimes D}\}.$$

This definition discards the infinite behaviours of $f \otimes g$ which correspond to unfair interleavings.

$\mathcal{FP}roc$ inherits the $*$-autonomous structure of $\mathcal{ASP}roc$, because all the structural morphisms, being essentially identities, are fair and the abstraction operation does not affect fairness. The exception to this is that there is no tensor unit: $\mathsf{ObAct}(I) = \varnothing$, so it is not possible to define F_I.

Proposition 5.40 $\mathcal{FP}roc$ is a compact closed category without units, which has countable (non-empty) biproducts and a unit delay endofunctor.

The specification structure for deadlock-freedom can now be defined over the progressive subcategory $\mathcal{FP}roc_{pr}$ of $\mathcal{FP}roc$, which now consists of those objects for which every safe trace can be extended to a valid infinite trace. The definitions are very similar to those for $\mathcal{SP}roc$. The essential difference is that convergence of a process means the ability to keep doing *observable* actions. Furthermore, the choice of next action should not commit the process to a branch of behaviour which can lead only to a disallowed infinite trace. If $P \in \mathsf{Proc}(A)$ then $P\!\downarrow$ means

- whenever $P \stackrel{s}{\Longrightarrow} Q$ there is $a \in \mathsf{ObAct}(A)$ and a process R such that $Q \stackrel{a}{\Longrightarrow} R$, and there is $t \in \mathsf{infobtraces}(R)$ such that $sat \in T_P$.

The definition of equivalence of $\mathcal{FP}roc$ processes P and Q, requiring $P \approx Q$ and $T_P = T_Q$, permits the possibility that although P and Q are not observation equivalent it is only the presence of branches corresponding to invalid infinite traces which causes observation equivalence to fail. However, if a process is convergent then there is no branch along which all infinite traces are invalid, so this situation does not arise. In the specification structure for deadlock-freedom over $\mathcal{FP}roc$, a property is a set of convergent processes and satisfaction is membership, just as in the synchronous case. This means that all the deadlock-free processes considered are convergent, and the equivalence behaves well for them. It is not, however, possible to require that $\mathcal{FP}roc$ should consist *only* of convergent processes, because convergence in itself is not preserved by composition. It is only when convergence is combined with satisfaction of suitable deadlock-free types that composition works.

Given P and Q of type A in $\mathcal{ASP}roc$, $P \sqcap Q$ is defined exactly as in $\mathcal{SP}roc$:

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \sqcap Q \xrightarrow{a} P' \sqcap Q'.}$$

If P and Q have type A in $\mathcal{FP}roc$ and $T_P \cap T_Q \neq \varnothing$, then $P \sqcap Q$ can be converted into an $\mathcal{FP}roc$ process of type A by defining $T_{P \sqcap Q} \stackrel{\text{def}}{=} T_P \cap T_Q$. Orthogonality is now defined by

$$P \perp Q \quad \stackrel{\text{def}}{\Leftrightarrow} \quad T_P \cap T_Q \neq \varnothing \text{ and } (P \sqcap Q)\!\downarrow .$$

It is extended to sets of processes exactly as in the synchronous case. For each object A, $P_D A$ is again the set of $^{\perp\perp}$-invariant sets of convergent processes of type A. Satisfaction is membership, and all of the operations on properties are defined exactly as before.

In the asynchronous situation, it is harder to prove that D satisfies the composition axiom. As should be expected, fairness is crucial. Before embarking on the proof, note that if $f : A \to B$ and $g : B \to C$ in $\mathcal{FP}roc$, communication between f and g when $f\,;g$ is formed can include periods in which the common action in B is τ_B. The definition of orthogonality of processes takes this into account, as it only requires an observable action to be available at some time in the future.

Lemma 5.41 Suppose $f : A \to B$, $U \in P_D A$, $V \in P_D B$ and $f \in U \multimap V$. Then for any $P \in U$, $P\,; f \in V$, and for any $Q \in V^{\perp}$, $f\,; Q \in U^{\perp}$.

Proof: We prove that $P\,; f \in V$; the other part is similar. For $Q \in V^{\perp}$ the requirement is that $((P\,; f) \sqcap Q)\!\downarrow$. Given traces s of f, t of P and u of Q with $s{\restriction}A = t$ and $s{\restriction}B = u$, there must be $b \neq \tau_B$ such that $P\,; f$ can do b after $s{\restriction}B$ and Q can do b after u. It must also be possible to extend the resulting behaviours of $P\,; f$ and Q to valid infinite traces.

Now, $f \in U \multimap V$ means $f \in (U \otimes V^{\perp})^{\perp}$, i.e. $f \in \{R \otimes S \mid R \in U, S \in V^{\perp}\}^{\perp\perp\perp}$ and hence $f \perp \{R \otimes S \mid R \in U, S \in V^{\perp}\}$. This gives $(f \sqcap (P \otimes Q))\!\downarrow$, so there is $(a, b) \neq (\tau_A, \tau_B)$ such that f can do (a, b) after s, P can do $\hat{a}$ after t and Q can do $\hat{b}$ after u; furthermore, these behaviours can be extended to valid infinite traces of f and $P \otimes Q$. If $b \neq \tau_B$ then the proof is complete. If the only possibility is $b = \tau_B$ then consider the traces $s(a, \tau_B)$ of f, $(s{\restriction}A)a$ of P and $s{\restriction}B$ of Q, and repeat the argument. We have to be sure that the case $b = \tau_B$ does not keep arising forever. But if it did, there would be an infinite trace of f in which all the actions from some point on were of the form (a, τ_B). This would contradict the fact that f satisfies the fairness specification $F_{A \multimap B}$, as this specification requires that any infinite trace of f must have infinite (and fair) projections in A and B. $\square$

Proposition 5.42 If $f : A \to B$ and $g : B \to C$, and $U \in P_D A$, $V \in P_D B$, $W \in P_D C$ with $f \in U \multimap V$ and $g \in V \multimap W$, then $f ; g \in U \multimap W$.

Proof: Given $P \in U$ and $Q \in W^\perp$, we need to show that $(f ; g) \perp (P \otimes Q)$, i.e. that $((f ; g) \sqcap (P \otimes Q))\downarrow$. By the lemma, $P ; f \in V$ and $g ; Q \in V^\perp$, so $(P ; f) \perp (g ; Q)$.

If there are traces s of f and t of g with $s{\restriction}B = t{\restriction}B$, and traces u of P and v of Q with $u = s{\restriction}A$ and $v = t{\restriction}C$, we need $(a, c) \neq (\tau_A, \tau_C)$ such that $f ; g$ and $P \otimes Q$ can do (a, c) after these traces. Convergence of $(P ; f) \sqcap (g ; Q)$ implies that there is $b \neq \tau_B$ such that $P ; f \xrightarrow{b} P' ; f'$ and $g ; Q \xrightarrow{b} g' ; Q'$, abusing notation by writing $P ; f \xrightarrow{b} P' ; f'$ for the transition made by $P ; f$ after the traces u of P and s of f. The transition of $P ; f$ means that one of the three following cases applies.

1. $P \xrightarrow{a} P'$ and $f \xrightarrow{(a,b)} f'$ with $a \neq \tau_A$.
2. $f \xrightarrow{(\tau_A, b)} f'$.
3. There is a sequence of a transitions by P and (a, τ_B) transitions by f, followed by case 1 or 2.

Similarly, the transition of $g ; Q$ means that one of three cases applies.

1. $Q \xrightarrow{c} Q'$ and $g \xrightarrow{(b,c)} g'$ with $c \neq \tau_C$.
2. $g \xrightarrow{(b,\tau)} g'$.
3. There is a sequence of c transitions by Q and (τ_B, c) transitions by g, followed by case 1 or 2.

Thus there are nine cases in all. Each one leads to the desired transition (a, c) of $f ; g$ and $P \otimes Q$, except for the case (2,2). This case gives a (τ_A, τ_C) transition and leads back to the beginning of the argument. The only undesirable possibility is that the case (2,2) keeps occurring, but this cannot actually arise because f and g are fair.

As for the need to extend the common behaviour of $f ; g$ and $P \otimes Q$ to a valid infinite trace for both processes, first note that convergence of $(P ; f) \sqcap (g ; Q)$ means that the behaviours of P, f, g and Q can all be extended to valid infinite traces. Hence the corresponding behaviours of $f ; g$ and $P \otimes Q$ can be extended, as long as the extension of the behaviour of $f ; g$ is a valid trace of $P \otimes Q$, i.e. a fair interleaving of the traces of P and Q. This is guaranteed by fairness of $f ; g$. □

The proof that identity morphisms satisfy the correct properties is the same as in the synchronous case. Hence

Proposition 5.43 D is a specification structure over $\mathcal{FP}roc_{pr}$.

The asynchronous deadlock-free category is called $\mathcal{FP}roc_D$. As in $\mathcal{SP}roc_D$ there are two obvious properties over each type A, namely M_A and $\{\mathsf{max}_A\}$, if the $\mathcal{ASP}roc$ process max_A which can always do any action is converted into an $\mathcal{FP}roc$ process by setting $T_{\mathsf{max}_A} \stackrel{\text{def}}{=} F_A$. The condition for them to be equal is that there is a unique observable action extending each trace in S_A. The first result on combining these properties is still true.

Proposition 5.44 $\{\mathsf{max}_A\} \otimes \{\mathsf{max}_B\} = \{\mathsf{max}_{A\otimes B}\}$ and $M_A \parr M_B = M_{A\parr B}$.

However, it is no longer the case that $M_A \otimes M_B = M_{A\otimes B}$. For example, take an object A with $\Sigma_A = \{a, \tau_A\}$ so that $\mathsf{max}_A = a.\mathsf{max}_A$ and $M_A = \{\mathsf{max}_A\}$. Now,

$$\begin{aligned} M_A \otimes M_A &= \{P \otimes Q \mid P, Q \in M_A\}^{\perp\perp} \\ &= \{\mathsf{max}_A \otimes \mathsf{max}_A\}^{\perp\perp} \end{aligned}$$

so $M_A \otimes M_A = M_{A\otimes A} \iff \{\mathsf{max}_A \otimes \mathsf{max}_A\}^{\perp} = \{\mathsf{max}_{A\otimes A}\}$. If $R : A \otimes A$ is defined by $R = (a, \tau_A).(\tau_A, a).R$ then $R \perp (\mathsf{max}_A \otimes \mathsf{max}_A)$ but $R \neq \mathsf{max}_{A\otimes A}$, so $\{\mathsf{max}_A \otimes \mathsf{max}_A\}^{\perp} \neq \{\mathsf{max}_{A\otimes A}\}$. This negative result slightly reduces the possibilities for using the properties M_A and $\{\mathsf{max}_A\}$ in examples, but the most important fact is that $M_A \parr M_B = M_{A\parr B}$. This gives

Proposition 5.45 If $P : A_1 \parr \cdots \parr A_n$ in $\mathcal{FP}roc$ and $P\!\downarrow$, then in $\mathcal{FP}roc_D$,

$$P : (A_1, M_{A_1}) \parr \cdots \parr (A_n, M_{A_n}).$$

As in the synchronous case, it is this result which is most useful in examples.

5.5.2 Adding a Tensor Unit

Throughout this thesis, processes have been uniformly regarded as morphisms. The units are essential for this view, as a process with just one port has to become a morphism from I. But it now turns out that $\mathcal{FP}roc_D$, a category which is intended to be used in the same way, does not have units. What can be done about this problem?

One possible approach is to abandon the view of processes as morphisms, and just work with typed processes. All the necessary operations can be defined directly on typed processes, in much the same way as they are presently defined on morphisms. This is rather unsatisfactory, as it means dropping a significant part of the theory and losing the connection with interaction categories (although retaining the ideas about types).

Another approach is to add a new object I to $\mathcal{FP}roc_D$, together with new morphisms making it into a tensor unit, to give a new category which is genuinely $*$-autonomous.

In the new category, a morphism $I \to A$ would be a process of type A (so the construction would begin not just with $\mathcal{FP}roc_D$ but with a collection of processes of each type) and composition of such morphisms would be defined in terms of the underlying Cut operation on typed processes. So this approach is essentially the same as the previous one, with the addition of a (probably rather complex) syntactic construction of a free category.

The solution which will now be described is to use a different object J, with care, as if it were a tensor unit. J is defined by

$$\begin{aligned} \Sigma_J &\stackrel{\text{def}}{=} \{\bullet, \tau_J\} \\ S_J &\stackrel{\text{def}}{=} \{\bullet^n \mid n < \omega\} \\ F_J &\stackrel{\text{def}}{=} \{\bullet^\omega\}. \end{aligned}$$

J looks rather like the tensor unit of $\mathcal{SP}roc$ and indeed, $\circ J \cong J$ in $\mathcal{FP}roc_D$. In a port of type J, nothing interesting happens; there is just a sequence of $\bullet$ actions marking time. But for each process P of type A, there is now a morphism $\hat{P} : J \to A$, defined by transition rules:

$$\frac{P \xrightarrow{a} P'}{\hat{P} \xrightarrow{(\bullet,a)} \hat{P}'} \qquad \frac{}{\hat{P} \xrightarrow{(\bullet,\tau_A)} \hat{P}}$$

and with $T_{\hat{P}} \stackrel{\text{def}}{=} \{s \in F_{J \multimap A} \mid s \restriction A \in T_P\}$.

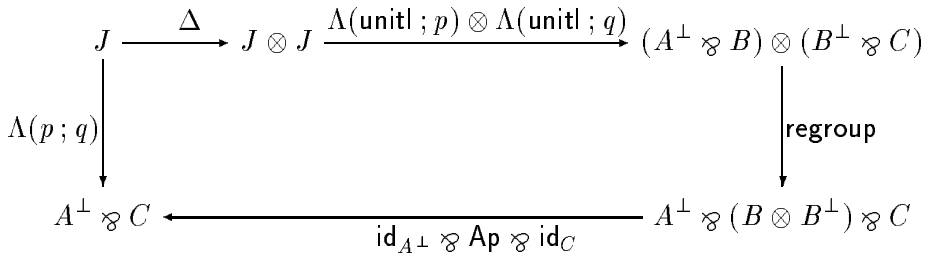
In this definition, P is considered as a synchronisation tree, and $\hat{P}$ is defined as a synchronisation tree; the final morphism is then the observation equivalence class of this tree. The second clause of the definition ensures that the $\hat{\cdot}$ operation is well-defined on observation equivalence classes.

There is an obvious condition which should be satisfied by the $\hat{\cdot}$ operation. If P is a process of type A and $Q : A \to B$, then there is a process $P \cdot Q$ of type B, and it should be the case that $\widehat{P \cdot Q} = \hat{P}\,; Q : J \to B$. It is easy to check, using the transition rules, that this is true.

If J is to be used as a tensor unit, it should be possible to eliminate it from an expression involving $\otimes$. This requires morphisms $\mathsf{unitl} : J \otimes A \to A$ and $\mathsf{unitr} : A \otimes J \to A$. It seems reasonable to require that $\Lambda(\mathsf{unitl}) = \widehat{\mathsf{id}_A}$ where id_A is considered as a process of type $A \multimap A$. This defines unitl: its transitions are $\mathsf{unitl} \xrightarrow{(\bullet,\tau_A,\tau_A)} \mathsf{unitl}$ and $\mathsf{unitl} \xrightarrow{(\bullet,a,a)} \mathsf{unitl}$ for any $a \in S_A$, with the usual restriction to safe and fair traces. Reversing the direction gives $\mathsf{unitl}^* : A \to J \otimes A$, and $\mathsf{unitl}^*\,;\mathsf{unitl} = \mathsf{id}_A$ but $\mathsf{unitl}\,;\mathsf{unitl}^* \neq \mathsf{id}_{J \otimes A}$. The definitions of unitr and unitr^* are symmetrical.

The interpretation of the Cut rule in a $*$-autonomous category uses $\mathsf{unitl}_I^{-1} : I \to I \otimes I$, and of course $\mathsf{unitl}_I^{-1} = \mathsf{unitr}_I^{-1}$. But now, $\mathsf{unitl}_J^* \neq \mathsf{unitr}_J^*$, so it is not clear which

morphism $J \to J \otimes J$ should be used. The answer is a new morphism $\Delta : J \to J \otimes J$, defined by the transitions $\Delta \xrightarrow{(\bullet,\bullet,\bullet)} \Delta$ and $\Delta \xrightarrow{(\tau_J,\tau_J,\tau_J)} \Delta$, with $T_J \stackrel{\text{def}}{=} \mathsf{infobtraces}(\Delta)$. As a check that this leads to the correct interpretation of Cut, the following diagram commutes for any $p : A \to B$ and $q : B \to C$.



5.5.3 Example: The Dining Philosophers

The problem of the dining philosophers [Hoa85] provides a good example of working with the category of asynchronous deadlock-free processes. This is a very well-known example in the concurrency literature, but it is worth reviewing the scenario here before plunging into an analysis. In a college there are five philosophers, who spend their lives seated around a table. In the middle of the table is a large bowl of spaghetti; also on the table are five forks, one between each pair of philosophers. Each philosopher spends most of his time thinking, but occasionally becomes hungry and wants to eat. In order to eat, he has to pick up the two nearest forks; when he has finished eating, he puts the forks down again. The problem consists of defining a concurrent system which models this situation; there are then various questions which can be asked about its behaviour. One is about deadlock-freedom: is it possible for the system to reach a state in which nothing further can happen, for example because the forks have been picked up in an unsuitable way? Another is about fairness: do all the philosophers get a chance to eat, or is it possible for one of them to be excluded forever? The reason for looking at the dining philosphers example in this chapter is to illustrate techniques for reasoning about deadlock-freedom, but because of the way in which the asynchronous deadlock-free category has been constructed, fairness has to be considered as well.

A philosopher can be modelled as a process with five possible actions: eating, picking up the left fork, putting down the left fork, picking up the right fork, and putting down the right fork. Calling these actions e, lu, ld, ru, rd respectively, a CCS definition of a philosopher could be $P = lu.ru.e.ld.rd.P$. There is no action corresponding to thinking: a philosopher is deemed to be thinking at all times, unless actually doing something else. In *ASProc* a philosopher has three ports: one for the eating action and one each for the left and right forks. The type of the fork ports is X, defined by $\Sigma_X \stackrel{\text{def}}{=} \{u, d, \tau_X\}$ and with S_X requiring u and d to alternate, starting with u. The type of the eating port is Y defined by $\Sigma_Y \stackrel{\text{def}}{=} \{e, \tau_Y\}$ and with S_Y allowing all traces.

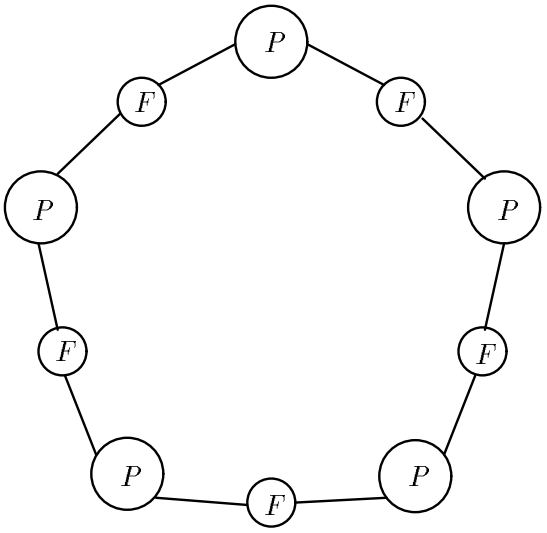


Figure 5.1: Process Configuration for the Dining Philosophers

The philosopher process can be typed as $P : X^{\perp} \parr Y \parr X$.

A fork has four actions, lu, ld, ru and rd. For the usage of these names by the fork to match their usage by the philosophers, the necessary convention is that if a fork does the action lu, it has been picked up from the right. A possible definition of a fork is $F = lu.ld.F + ru.rd.F$ and it can be typed as $F : X^{\perp} \parr X$.

Five philosophers and five forks can be connected together in the desired configuration, illustrated in Figure 5.1, by using the compact closed structure of $\mathcal{ASP}roc$, as usual. The next step is to transfer everything to $\mathcal{FP}roc$ and then to $\mathcal{FP}roc_D$.

To construct the P and F processes in $\mathcal{FP}roc$, fairness specifications must be added to the types X and Y, and the acceptable infinite behaviours of P and F must be specified. This will be done in such a way that P and F satisfy the appropriate fairness specifications. For both X and Y the fairness specification can simply be all infinite traces. This means that there is no fairness requirement on the actions within a port, but only between ports. For the types of the philosopher and the fork, $F_{X^{\perp} \parr Y \parr X}$ consists of the infinite traces whose projections into the three ports are all infinite, and similarly $F_{X^{\perp} \parr X}$.

To convert the $\mathcal{ASP}roc$ process P into an $\mathcal{FP}roc$ process, it is sufficient to take $T_P = \mathsf{infobtraces}(P)$. It is then clear that $T_P \subseteq F_{X^{\perp} \parr Y \parr X}$ because the behaviour of P simply cycles around all the available actions. Also, P is convergent because its behaviour consists of just one infinite branch. However, F has unfair infinite behaviours—for example, there is an infinite trace in which the ru and rd actions never appear. Thus T_F must be defined in such a way as to eliminate these undesirable infinite traces,

and this can easily be done by taking $T_F = F_{X^\perp \otimes X}$. Then F is convergent, because any of its finite behaviours can be extended to a fair infinite behaviour by choosing a suitable interleaving from that point on. This approach means that this section is not addressing the issue of how fairness can be achieved in the dining philosophers problem—to do that, the implementation of a fair scheduler would have to be considered. As already stated, this problem has only been introduced as an example of compositional reasoning about deadlock-freedom; fairness only appears in the minimal possible way needed for the categorical approach to be applicable.

Typing the philosopher and fork processes in $\mathcal{FP}roc_D$ requires suitable properties over the types X and Y. For Y, M_Y can be used. Because Y has only one observable action, $M_Y = M_Y^\perp$. Similarly for X, the set M_X can be used, and because the safety specification of X is such that in each state there is only one action available, $M_X = M_X^\perp$. Because $F : X^\perp \otimes X$ in $\mathcal{FP}roc$ and F is convergent, $F \models M_{X^\perp} \otimes M_X$ and so $F : (X, M_X)^\perp \otimes (X, M_X)$ in $\mathcal{FP}roc_D$. Similarly, $P : (X, M_X)^\perp \otimes (Y, M_Y) \otimes (X, M_X)$ in $\mathcal{FP}roc_D$. These typings mean that any number of philosophers and forks can be connected together in a line, and the resulting process is guaranteed to be deadlock-free. Interestingly, this applies not only to the "correct" configuration in which philosophers and forks alternate, but also to other possibilities such as a sequence of forks with no philosophers.

The interesting step of the construction consists of completing the cycle by connecting the X and $X^\perp$ ports at opposite ends of a chain in which forks and philosophers alternate. $\mathcal{FP}roc_D$ is not compact closed, so just as in the synchronous case there is a condition which a process must satisfy before two of its ports can be connected to form a cycle.

Suppose in general that $P : (\Gamma, U) \otimes (X, V) \otimes (X^\perp, V^\perp)$ in $\mathcal{FP}roc_D$. As before there is an obvious condition that forming $\overline{P}$ by connecting the X and $X^\perp$ ports should not cause a deadlock: that every trace s of P with $s{\restriction}X = s{\restriction}X^\perp$ can be extended by an action $(\bar{a}, x, x)$ of P. The action x could be τ_X, as it is permissible for the sequence of communications between the X and $X^\perp$ ports to pause, or the action tuple $\bar{a}$ could be τ_Γ, but not both. Again, to obtain $\overline{P} : (\Gamma, U)$ in $\mathcal{FP}roc_D$ it is also necessary to ensure that the specification U can still be satisfied while the communication is taking place.

The possibility of divergence does not have to be considered separately. It is conceivable that $\overline{P}$ could have a non-deadlocking infinite behaviour in which no observable actions occur in Γ, but the corresponding behaviour of P would be unfair because it would neglect the ports in Γ. Thus it is sufficient to state a condition which guarantees that forcing X and $X^\perp$ to communicate does not affect the actions available in the other ports. Just as was the case in $\mathcal{SP}roc$, this condition can be expressed in terms of ready

pairs. The definition of $\mathsf{readies}(P)$ for an $\mathcal{FP}roc$ process P of type A is

$$\begin{array}{rcl} \mathsf{initials}(P) & \stackrel{\text{def}}{=} & \{a \in \mathsf{ObAct}A \mid \exists Q.P \stackrel{a}{\Longrightarrow} Q\} \\ \mathsf{readies}(P) & \stackrel{\text{def}}{=} & \{(s, X) \mid \exists Q.[(P \stackrel{s}{\Longrightarrow} Q) \wedge (X = \mathsf{initials}(Q))]\}. \end{array}$$

The condition $\mathsf{cycle}(P)$ is now

- For every $(s, A) \in \mathsf{readies}(P)$ such that $s{\restriction}X = s{\restriction}X^{\perp}$, and every action $(\bar{a}, x, y) \in A$, there is $z \in \Sigma_X$ such that $\tau_{\Gamma \otimes X \otimes X^{\perp}} \neq (\overline{a}, z, z) \in A$.

As before, this leads to a proof rule for cycle formation.

$$\frac{P : (\Gamma, U) \otimes (X, V) \otimes (X^{\perp}, V^{\perp}) \quad \mathsf{cycle}(P)}{\overline{P} : (\Gamma, U)}$$

These ideas can now be applied to the dining philosophers problem. First of all, some traditional analysis based on reasoning about the state of the system is useful. For the moment, the e actions can be ignored as they do not have any impact on deadlocks in this system. The following cases cover all possibilities for a state.

1. If there is P_i such that both adjacent forks are down, it can pick up the left fork.
2. If there is P_i whose right fork is up and whose left fork is down, it can either put down the right fork (if it has just put down the left fork) or pick up the left fork (if its neighbour has the right fork).
3. If all forks are up and some P_i has both its forks, it can put down the left fork.
4. If all forks are up and every P_i has just one fork, they all have their left forks, and there is a deadlock.

The last case is the classic deadlocking possibility for the dining philosophers—each philosopher in turn picks up the left fork, and then they are stuck. In terms of ready sets, there is a state in which every possible next action has non-matching projections in the two X ports.

In Hoare's formulation of the dining philosophers problem [Hoa85] the philosophers are not normally seated, but have to sit down before attempting to pick up their forks. This means that the possibility of deadlock can be removed by adding a *footman*, who controls when the philosophers sit down. The footman ensures that at most four philosophers are seated at any one time, which means that there is always a philosopher with an available fork on both sides; in this way, the deadlocked situation is avoided. However, implementing this solution involves a major change to the system: there is a new process representing the footman, the philosopher processes have extra ports

on which they interact with the footman, and consequently their types need to be re-examined. It is more convenient to use an alternative approach, which will now be described.

One of the philosophers is replaced by a variant, P', which picks up the forks in the opposite order. So $P' = ru.lu.e.rd.ld.P'$ in CCS notation. Intuitively, this prevents the deadlocking case from arising, because even if the four Ps each pick up their left fork, P' is still trying to pick up its right fork (which is already in use) and so one of the Ps has a chance to pick up its right fork as well. The check that there are no deadlocks takes the form of a case analysis, as before.

1. If all the forks are up and some philosopher has both its forks, it can put one of them down, whether it is P or P'.

2. If all the forks are up and every philosopher has just one, either they all have their left fork or all the right. If they all have their left fork, then P' can put down its left fork. If they all have their right fork, then any P can put down its right fork.

3. If two adjacent forks are down, then the philosopher in between them can pick one of them up, whether it is P or P'.

4. Otherwise there is the configuration $u - \texttt{phil}_1 - d - \texttt{phil}_2 - u - \texttt{phil}_3$.

   - If $\texttt{phil}_2$ is P and has its right fork, it can put down the right fork.
   - If $\texttt{phil}_2$ is P and doesn't have its right fork, it can pick up the left fork.
   - If $\texttt{phil}_2$ is P' and has its right fork, it can pick up the left fork.
   - If $\texttt{phil}_2$ is P' and doesn't have its right fork, then $\texttt{phil}_3$ must be P and has its left fork. Then if $\texttt{phil}_3$'s right fork is down, $\texttt{phil}_3$ can pick it up. If the right fork is up and $\texttt{phil}_3$ has it, it can put down the left fork. Otherwise, $\texttt{phil}_4$ is P and has its left fork. Continuing this argument for each $\texttt{phil}_i$ with $i \geqslant 4$ leads eventually to either a possible action, or cyclically back to $i = 1$ and the deduction that $\texttt{phil}_1$ has its left fork. In the latter case, since $\texttt{phil}_1$ is P, it can pick up its right fork.

To recast this argument in terms of checking the condition on the final cyclic connection, suppose that the final connection is between the P' process and the fork on its right. Each case of the argument either produces a communication between P' and this fork, or produces a communication elsewhere in the cycle, which means that there is an action of the system in which the two ports to be connected both delay. This shows that the cycle condition is satisfied, and the proof rule can be applied.

5.6 Discussion

This chapter has shown that the interaction categories paradigm is a framework in which verification can be carried out. The first section demonstrates that the safety specifications built into the objects of $\mathcal{SP}roc$ and $\mathcal{ASP}roc$ are not there just for technical reasons, but can be used to support equational reasoning about safety properties of processes. The analysis of the cyclic scheduler shows this scheme in operation.

The rest of the chapter substantiates the claim of Chapter 1 that if behavioural properties can be encapsulated as types, then type-checking methods can be used for compositional verification of those properties. The notion of specification structure provides a systematic way of extending the types of an interaction category by the addition of extra properties, so that the structure of the category is preserved and the categorical operations can be used as compositional proof rules. As an example, a specification structure for deadlock-freedom is defined over $\mathcal{SP}roc$, to give the category $\mathcal{SP}roc_D$. Its details are complex, but several results are proved which allow many processes which have types in $\mathcal{SP}roc$ to be assigned types in $\mathcal{SP}roc_D$ with a minimum of effort. This is illustrated by using type-checking to verify deadlock-freedom of certain dataflow networks. Because $\mathcal{SP}roc_D$ is not compact closed, additional conditions must be satisfied before a cyclic connection can be made. If a process has two ports which could be connected together in $\mathcal{SP}roc$, a sufficient condition is established for that connection to be valid in $\mathcal{SP}roc_D$; this condition is also formulated as a simple proof rule.

In order to be able to discuss more interesting examples, the theory of types for deadlock-freedom is extended to include asynchronous processes. This turns out to be more difficult. Instead of $\mathcal{ASP}roc$, the starting point is a new category $\mathcal{FP}roc$ of asynchronous and fair processes; fairness has to be introduced to deal with problems of divergence. A specification structure similar to the previous one is defined, using the sets of processes approach, yielding the category $\mathcal{FP}roc_D$ of asynchronous deadlock-free processes. Before this category can be put to use, one further technical problem has to be solved: the tensor unit I of $\mathcal{ASP}roc$ does not lift to $\mathcal{FP}roc_D$, so a different object J has to be used with care as a tensor unit. Once this has been done, some results allowing typed processes to be lifted from $\mathcal{FP}roc$ to $\mathcal{FP}roc_D$ are established. Just as with $\mathcal{SP}roc_D$, there is a proof rule for cycle formation. Finally, the dining philosophers problem is analysed in order to demonstrate lifting of types from $\mathcal{FP}roc$ to $\mathcal{FP}roc_D$, combination of typed processes, and the use of the proof rule to form a cyclic connection.

6

Typed Process Calculus

6.1 Introduction

The aim of this chapter is to develop a typed calculus of synchronous processes based on the structure of interaction categories. Its syntax is inspired by existing process calculi, especially SCCS [Mil83], and also by Abramsky's linear realisability algebras (LRAs) [Abr94c, Abr91, AJ92]. The calculus is presented by means of a collection of rules for deriving typed terms; the form of a typed term is based on a one-sided classical linear logic sequent as in Proofs as Processes. The definition of the calculus is parameterised with respect to a process signature, which specifies ground types and prefix actions; the declaration of a ground type must be accompanied by a statement of which actions are available in it.

The calculus has an operational semantics, which yields a notion of typed bisimulation. Instead of the usual Subject Reduction theorems which state that types are unchanged by transitions, there are two results: Dynamic Subject Reduction, which states that transitions cause types to evolve in predictable ways, and Static Subject Reduction, which states that an approximation to the type of a term (its number of ports) is preserved.

There is also a categorical semantics, which interprets a typed term as a morphism in a synchronous interaction category. A suitable abstract definition of the required categorical structure is given, in the style of Chapter 3. One suitable category is $\mathcal{SP}roc$; it has objects capable of interpreting any collection of ground types and prefix actions, so the calculus can always be given a semantics there. The operational semantics associates a synchronisation tree with each typed process; the categorical semantics in $\mathcal{SP}roc$ also assigns a synchronisation tree to each process, and these trees are the same up to a trivial relabelling. Furthermore, denotational equality in $\mathcal{SP}roc$ is sound with respect to typed bisimulation.

There are several ways in which the theory presented in this chapter could be extended. One is the addition of delay operators, corresponding to the delay functors δ and Δ of $\mathcal{SP}roc$, to allow asynchronous processes to be constructed. This extension is outlined at the end of the chapter. Another development of the theory would be in the direction of

categorical logic, as described for the typed λ-calculus in Chapter 1. The syntax of the typed process calculus can be used as the basis for an internal language for (a restricted form of) interaction categories, and the usual correspondences between syntax and semantics can be proved. Work in this area is not yet sufficiently far advanced to be reported in detail in this thesis, but a general outline is given. Finally, it is desirable for the typed calculus to be able to define processes in categories in which types are more than safety specifications—for example, the deadlock-free category $\mathcal{SP}roc_D$ of Chapter 5. Some modification of the syntax and term formation rules would be necessary, and this is discussed at the end of the chapter.

6.2 Syntax

A *signature* $\mathcal{S}g$ for a process calculus is specified by the following data.

- A collection of *ground types*. The collection of *types* is then defined by the grammar
$$\alpha ::= \gamma \mid \alpha \otimes \alpha \mid \alpha \parr \alpha \mid \alpha^{\perp} \mid \circ \alpha$$
in which γ is any ground type.
- A collection of *ground actions*. The collection of *actions* is then defined by the grammar
$$\pi ::= \sigma \mid * \mid (\pi, \pi)$$
in which σ is any ground action.
- A collection of *ground prefixes*, each of which consists of a ground action and a pair of ground types, written $\mathsf{Prefix}\ \sigma : \gamma \to \gamma'$. These are subject to the restriction that if $\mathsf{Prefix}\ \sigma : \gamma \to \gamma'$ and $\mathsf{Prefix}\ \sigma : \gamma \to \gamma''$ are ground prefixes then $\gamma' = \gamma''$.

The *prefixes* generated by $\mathcal{S}g$ are the expressions $\mathsf{Prefix}\ \pi : \alpha \to \beta$ which can be derived from the ground prefixes by means of the rules in Figure 6.1. The following lemmas will be useful later.

Lemma 6.1 If $\mathsf{Prefix}\ \pi : \alpha \to \beta$ and $\mathsf{Prefix}\ \pi : \alpha \to \gamma$ are provable, then $\beta = \gamma$.

Proof: By induction on the derivation of $\mathsf{Prefix}\ \pi : \alpha \to \beta$. □

Lemma 6.2 If $\mathsf{Prefix}\ \pi : \alpha \to \beta$ and $\mathsf{Prefix}\ \pi : \alpha^{\perp} \to \gamma$ are derivable, then $\gamma = \beta^{\perp}$.

$$\frac{}{\mathsf{Prefix}\ * : \circ\alpha \to \alpha} \qquad \frac{\mathsf{Prefix}\ \pi : \alpha \to \beta}{\mathsf{Prefix}\ \pi : \alpha^{\perp} \to \beta^{\perp}}$$

$$\frac{\mathsf{Prefix}\ \pi : \alpha \to \gamma \quad \mathsf{Prefix}\ \pi' : \beta \to \delta}{\mathsf{Prefix}\ (\pi, \pi') : \alpha \otimes \beta \to \gamma \otimes \delta} \qquad \frac{\mathsf{Prefix}\ \pi : \alpha \to \gamma \quad \mathsf{Prefix}\ \pi' : \beta \to \delta}{\mathsf{Prefix}\ (\pi, \pi') : \alpha \parr \beta \to \gamma \parr \delta}$$

Figure 6.1: Prefixes Generated by a Process Signature

Proof: Given $\mathsf{Prefix}\ \pi : \alpha \to \beta$, $\mathsf{Prefix}\ \pi : \alpha^{\perp} \to \beta^{\perp}$ can be derived. Then by the previous lemma, $\gamma = \beta^{\perp}$. □

The next step is to define the *raw processes*, which are untyped process terms. Because there are several process constructions, a number of which bind variables in various ways, it is convenient to use a metalanguage to define the raw processes. This also makes it very straightforward to define an operation of substitution (of variables for variables) on raw processes.

The metalanguage is a simply typed λ-calculus in which, to avoid confusion between the meta level and the object level, the types are called *arities*. There are two ground arities: `var` and `proc`, so the arities are generated by the grammar

$$\alpha ::= \mathtt{var} \mid \mathtt{proc} \mid \alpha \Rightarrow \alpha.$$

The abbreviation $\alpha^n \Rightarrow \beta$ is used for $\alpha \Rightarrow \cdots \Rightarrow \alpha \Rightarrow \beta$ with n occurrences of α, and as usual the $\Rightarrow$ constructor associates to the right.

The metalanguage has certain constants, which correspond to process construction rules. The general scheme is that a process is a meta-expression of arity `proc`, and a variable (such as will appear in the process calculus) is a metavariable of arity `var`. An expression of type $\mathtt{var} \Rightarrow \mathtt{proc}$, which will be of the form $\lambda x.P$, represents a process term P with a free variable x; there may also be other free variables in P. A process construction which takes one process, and binds a variable in it, becomes a constant of arity $(\mathtt{var} \Rightarrow \mathtt{proc}) \Rightarrow \mathtt{proc}$. If this constant is F, then it can be applied to an expression of type $\mathtt{var} \Rightarrow \mathtt{proc}$, giving $F(\lambda x.P)$. A process construction G which works like F but also introduces a new free variable, becomes a constant of arity $(\mathtt{var} \Rightarrow \mathtt{proc}) \Rightarrow \mathtt{var} \Rightarrow \mathtt{proc}$. It can be applied to a process and a new variable, for example $G(\lambda x.P, y)$. The result is a process term in which x is bound and y is free.

The benefit of using a metalanguage in this way is that a single binding construction in the metalanguage, namely λ-abstraction, can be used to describe many different binding operations in the object language. This greatly simplifies the definition of

substitution.

The metaconstants needed for the process calculus are:

- nil_n of arity $\mathtt{var}^n \Rightarrow \mathtt{proc}$, for each $n \geqslant 1$
- sum of arity $\mathtt{proc}^2 \Rightarrow \mathtt{proc}$
- tensor of arity $(\mathtt{var} \Rightarrow \mathtt{proc})^2 \Rightarrow \mathtt{var} \Rightarrow \mathtt{proc}$
- par of arity $(\mathtt{var}^2 \Rightarrow \mathtt{proc}) \Rightarrow \mathtt{var} \Rightarrow \mathtt{proc}$
- cut of arity $(\mathtt{var} \Rightarrow \mathtt{proc})^2 \Rightarrow \mathtt{proc}$
- prefix_π of arity $\mathtt{proc} \Rightarrow \mathtt{proc}$, for each action π
- axiom of arity $\mathtt{var}^2 \Rightarrow \mathtt{proc}$
- mix of arity $\mathtt{proc}^2 \Rightarrow \mathtt{proc}$
- cycle of arity $(\mathtt{var}^2 \Rightarrow \mathtt{proc}) \Rightarrow \mathtt{proc}$
- $\mathsf{fix}^i(\mathcal{E})$ of arity $\mathtt{var} \Rightarrow \mathtt{proc}$, for each $i > 0$ and each collection $\mathcal{E}$ of i recursive equations (to be defined later).

This data can be used as a signature for a simply typed λ-calculus which is the metalanguage. Taking a set Var^a of variables for each arity a, the (raw) *expressions* are given by

$$e ::= x^a \mid c \mid ee \mid \lambda x^a.e$$

with $x^a \in Var^a$ for any arity a, and c any constant. Well formed expressions of arity a are written $e : a$ and are formed by the usual rules of the simply typed λ-calculus with constants. There are sets $Exp^a \stackrel{\text{def}}{=} \{e \mid e : a\}$ of well-formed expressions of arity a. As usual there are notions of free and bound variables, α-equivalence, and substitution of an expression for free occurrences of a variable in another expression. The set Raw of *raw processes* is the set of expressions of type $\mathtt{proc}$ up to α-equivalence, i.e.

$$Raw \stackrel{\text{def}}{=} Exp^{\mathtt{proc}} / \cong_\alpha .$$

There is no notational distinction between an expression $P \in Exp^{\mathtt{proc}}$ and the α-equivalence class in Raw which it represents. Also, it is convenient to introduce some syntactic sugar when writing down raw processes. For example, suppose that $P, Q \in Raw$ and that x, y, z are metavariables of arity $\mathtt{var}$. Then $\mathsf{tensor}(\lambda x.P, \lambda y.Q, z) \in Raw$, and this raw process will be written $P \otimes_z^{x,y} Q$. There are corresponding sugared versions of the syntax for the other process constructions.

The *proved processes* generated by $\mathcal{S}g$ are the expressions $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n$ which can be derived using the rules in Figure 6.2. These rules use the sugared version of the syntax. The form of a proved process, i.e. $P \vdash x_1 : \alpha_n, \ldots, x_n : \alpha_n$, is exactly as specified by the Proofs as Processes interpretation. In such an expression P is a raw process, the α_i are types, and the x_i are variables. The order of the $x_i : \alpha_i$ is unimportant. The recommended reading of the expression is "the process P has the interface $x_1 : \alpha_1, \ldots, x_n : \alpha_n$". Each variable corresponds to a port of the process, and each port has a type. Labelling ports in this way means that process constructions are able to refer to particular ports. A port is a place in which actions can happen; so this calculus, unlike many others, makes a clear distinction between ports and actions.

There is a possibility of confusion with the usual notation for intuitionistic sequents, in which an expression $\Gamma \vdash A$ means that A can be proved from the hypotheses Γ. The present notation for processes originated in the one-sided sequent presentation of classical linear logic, in which an expression $\vdash \Gamma$ means that Γ can be proved; the process term has been added in the most obvious place, namely on the left.

As in CCS, prefixes can be attached to a process to specify actions to be performed. The syntax for a process which does the action π and becomes P is $\pi : P$. This syntax is taken from the synchronous prefix construction of SCCS. The prefixing rule takes a proved process $P \vdash x : \beta$ and a prefix action π, and forms $\pi : P \vdash x : \alpha$, which involves a change of type. This change is governed by the prefix judgement containing π, i.e. $\mathsf{Prefix}\ \pi : \alpha \to \beta$. The reason for the change in type caused by prefixing is that the calculus is intended to be interpreted in an interaction category such as $\mathcal{SP}roc$. Because a type in $\mathcal{SP}roc$ contains a safety specification, knowing that P has some type β does not guarantee that $\pi : P$ also has type β. If P satisfies the safety specification S then $\pi : P$ satisfies the specification $T \stackrel{\text{def}}{=} \{\pi s \mid s \in S\} \cup \{\varepsilon\}$, and in general these specifications are different. If processes are to be constructed using prefixing, there must be several types in the signature; semantically, their safety specifications are related in the same way as S and T above, and syntactically this is indicated by the prefix judgements which form part of a process signature.

The Prefix rule has as its hypothesis a process with just one port. If a process has several ports, for example $P \vdash x : \alpha, y : \beta, z : \gamma$, the Par rule can be used to combine them in pairs, so $\otimes_u^{x,y}(P) \vdash u : \alpha \otimes \beta, z : \gamma$ and then $\otimes_v^{u,z}(\otimes_u^{x,y}(P)) \vdash v : (\alpha \otimes \beta) \otimes \gamma$ can be formed. The prefix combination rules allow prefix actions to be combined similarly. If there are prefixes $\mathsf{Prefix}\ a : \alpha \to \gamma$, $\mathsf{Prefix}\ b : \beta \to \delta$ and a process $P \vdash x : \gamma, y : \delta$ then the following derivation allows the combined action (a, b) to be prefixed onto P.

$$\dfrac{\dfrac{P \vdash x : \gamma, y : \delta}{\otimes_z^{x,y}(P) \vdash z : \gamma \otimes \delta} \qquad \dfrac{\mathsf{Prefix}\ a : \alpha \to \gamma \qquad \mathsf{Prefix}\ b : \beta \to \delta}{\mathsf{Prefix}\ (a, b) : \alpha \otimes \beta \to \gamma \otimes \delta}}{(a, b) : \otimes_z^{x,y}(P) \vdash z : \alpha \otimes \beta}$$

Connection Rules

$$\frac{}{I_{x,y} \vdash x : \alpha^{\perp}, y : \alpha}\ \text{Axiom} \qquad \frac{P \vdash \Gamma, x : \alpha \quad Q \vdash \Delta, x : \alpha^{\perp}}{P \mathbin{\dot{x}} Q \vdash \Gamma, \Delta}\ \text{Cut}$$

$$\frac{P \vdash \Gamma \quad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \text{Mix} \qquad \frac{P \vdash \Gamma, x : \alpha, y : \alpha^{\perp}}{P \backslash_{x,y} \vdash \Gamma}\ \text{Cycle}$$

Multiplicative Rules

$$\frac{P \vdash \Gamma, x : \alpha \quad Q \vdash \Delta, y : \beta}{P \otimes_z^{x,y} Q \vdash \Gamma, \Delta, z : \alpha \otimes \beta}\ \text{Tensor} \qquad \frac{P \vdash \Gamma, x : \alpha, y : \beta}{\parr_z^{x,y}(P) \vdash \Gamma, z : \alpha \parr \beta}\ \text{Par}$$

Summation Rules

$$\frac{P \vdash \Gamma \quad Q \vdash \Gamma}{P + Q \vdash \Gamma}\ \text{Sum} \qquad \frac{}{\mathsf{nil}_{\alpha_1,\ldots,\alpha_n}(x_1, \ldots, x_n) \vdash x_1 : \alpha_n, \ldots, x_n : \alpha_n}\ \text{Nil}$$

Prefixing Rule

$$\frac{P \vdash x : \beta \quad \mathsf{Prefix}\ \pi : \alpha \to \beta}{\pi : P \vdash x : \alpha}\ \text{Prefix}$$

Recursion Rule

$$\frac{\begin{array}{ccc} X_1 \vdash x_1 : \alpha_1 & \cdots & X_n \vdash x_n : \alpha_n \\ \vdots & & \\ E_1(\overline{X}) \vdash x_1 : \alpha_1 & \cdots & E_n(\overline{X}) \vdash x_n : \alpha_n \end{array} \qquad \begin{array}{ccc} X_1 \vdash x_1 : \alpha_1 & \cdots & X_n \vdash x_n : \alpha_n \\ & & \vdots \end{array}}{\mathsf{fix}_{x_i}^{i}(\overline{X} = \overline{E}(\overline{X})) \vdash x_i : \alpha_i}$$

where the derivations of the E_i make each X_j sequential and guarded.

Figure 6.2: Proved Processes Generated by a Process Signature

Semantically, $\parr_z^{x,y}(P) \vdash z : \gamma \parr \delta$ is the same process as $P \vdash x : \gamma, y : \delta$. The difference is just in how the interface is viewed.

The Tensor rule gives another way of combining two ports into one, but this time the ports are taken from two different processes. If $P \vdash \Gamma, x : \alpha$ and $Q \vdash \Delta, y : \beta$ then $P \otimes_z^{x,y} Q \vdash \Gamma, \Delta, z : \alpha \otimes \beta$ is a process which puts P and Q in parallel, and has an interface formed from the interfaces of P and Q. Combining processes by Tensor is like putting them in parallel in CCS, except that there is no possibility of communication between them. The Cut rule connects processes together in such a way that communication is not only allowed, but required. In CCS terms, it is like a combination of parallel composition and restriction; this is exactly the interpretation of Cut put forward in Proofs as Processes, and seen in the definition of composition in $\mathcal{SP}roc$. If $P \vdash \Gamma, x : \alpha$ and $Q \vdash \Delta, x : \alpha^\perp$ then the x ports of P and Q, being of dual types, could be connected together. In $P \mathbin{\dot{x}} Q \vdash \Gamma, \Delta$ this connection has been made, and the processes have to communicate.

The Axiom rule produces the process $I_{x,y} \vdash x : \alpha^\perp, y : \alpha$. This process acts as a buffer or wire, and is useful for rearranging interfaces. For example, it can be used to derive an inverse to the Par rule.

$$\dfrac{P \vdash \Gamma, z : \alpha \parr \beta \qquad \dfrac{\overline{I_{u,x} \vdash u : \alpha^\perp, x : \alpha} \qquad \overline{I_{v,y} \vdash v : \beta^\perp, y : \beta}}{I_{u,x} \otimes_z^{u,v} I_{v,y} \vdash z : \alpha^\perp \otimes \beta^\perp, x : \alpha, y : \beta}}{P \mathbin{\dot{z}} (I_{u,x} \otimes_z^{u,v} I_{v,y}) \vdash \Gamma, x : \alpha, y : \beta}$$

The Sum rule allows the construction of the non-deterministic combination of two processes which have the same interface. This operation is intended to have the same meaning as + in CCS. The Nil rule allows nil processes to be introduced with any interface; as usual, nil is the unit for +.

In order to illustrate some of the operations which have been described so far, consider the dataflow network which was used in Chapter 4 as an example of LUSTRE. This network is reproduced in Figure 6.3, annotated with channel names.

Suppose there is a process signature with a single ground type N, intended to represent the natural numbers. Forgetting about the ground prefixes for the moment, assume that proved processes have been constructed which represent the various nodes in the network. They are listed here with port names matching the annotations in Figure 6.3.

$$\begin{array}{ll} \mathsf{one} \vdash x : N & \mathsf{zero} \vdash u : N \\ \mathsf{plus} \vdash x : N^\perp, y : N^\perp, z : N & \mathsf{then} \vdash z : N^\perp, u : N^\perp, v : N \\ \mathsf{fork} \vdash v : N^\perp, w : N, t : N & \mathsf{pre} \vdash t : N^\perp, s : N \end{array}$$

The following derivation shows the Cut and Cycle rules being used to assemble the network from the individual nodes. Every step of the construction is an application of

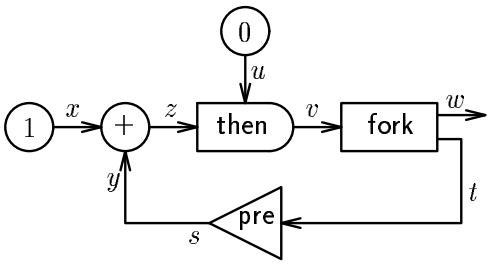


Figure 6.3: An Annotated Network

the Cut rule, except for the very last step which uses Cycle. First, the one, plus, zero and then processes are connected.

$$\dfrac{\dfrac{\mathsf{one} \vdash x : N \quad \mathsf{plus} \vdash x : N^{\perp}, y : N^{\perp}, z : N}{\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \vdash y : N^{\perp}, z : N} \quad \dfrac{\mathsf{zero} \vdash u : N \quad \mathsf{then} \vdash z : N^{\perp}, u : N^{\perp}, v : N}{\mathsf{zero} \mathbin{\dot{u}} \mathsf{then} \vdash z : N^{\perp}, v : N}}{\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then}) \vdash y : N^{\perp}, v : N}$$

Next, the fork and pre processes are added, and finally the loop is formed.

$$\dfrac{\dfrac{\dfrac{\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then}) \vdash y : N^{\perp}, v : N \quad \mathsf{fork} \vdash v : N^{\perp}, w : N, t : N}{\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then}) \mathbin{\dot{v}} \mathsf{fork} \vdash y : N^{\perp}, w : N, t : N} \quad \mathsf{pre} \vdash t : N^{\perp}, s : N}{\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then}) \mathbin{\dot{v}} \mathsf{fork} \mathbin{\dot{t}} \mathsf{pre} \vdash y : N^{\perp}, w : N, s : N}}{(\mathsf{one} \mathbin{\dot{x}} \mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then}) \mathbin{\dot{v}} \mathsf{fork} \mathbin{\dot{t}} \mathsf{pre}) \backslash_{y,s} \vdash w : N}$$

Cut is treated as an associative operator: when more than two processes are connected together, no brackets are used as long as all possible readings of the expression make sense. For example, brackets appear in the expression $\mathsf{plus} \mathbin{\dot{z}} (\mathsf{zero} \mathbin{\dot{u}} \mathsf{then})$ because $\mathsf{plus} \mathbin{\dot{z}} \mathsf{zero}$ cannot be formed (zero has no port named z). Section 6.3 explains why Cut and other operators can be treated as associative up to bisimulation.

The Recursion rule will now be described, and illustrated with definitions of some of the processes used in the example network. The full Recursion rule is rather complicated, because it describes the most general form of a set of mutually recursive process definitions. It is best explained by considering a sequence of progressively more complex cases. The simplest recursive definition in, say, SCCS constructs a process X satisfying $X = a : X$, where $=$ denotes strong bisimulation. Equality for the typed process calculus has not yet been introduced, but the notation $\mathsf{fix}(X = a : X)$ is nevertheless a useful and natural notation for this recursively defined process. If the type of X is going to be α, the type of $a : X$ should also be α, so a judgement $\mathsf{Prefix}\ a : \alpha \to \alpha$ is needed. This gives the first approximation to the recursion rule.

$$\dfrac{\mathsf{Prefix}\ \pi : \alpha \to \alpha}{\mathsf{fix}_u(X = \pi : X) \vdash u : \alpha}$$

For example, if there is a ground type N and a ground prefix $\mathsf{Prefix}\ 1 : N \to N$, then the process

$$\mathsf{fix}_x(X = 1 : X) \vdash x : N$$

can be constructed. This is the process which was called $\mathsf{one} \vdash x : N$ in the previous example.

There is no proved process corresponding to the variable X; talking about the type of X is just an informal way of discovering the correct form of the rule. The next stage is a definition with more than one prefix, for example $X = a : b : X$. If the type of X is α, the type of $b : X$ is β (different from α in general), so the required prefixes are $\mathsf{Prefix}\ a : \alpha \to \beta$ and $\mathsf{Prefix}\ b : \beta \to \alpha$. As before, X and $a : b : X$ should have the same type. Generalising to n prefixes gives the next version of the recursion rule.

$$\frac{\mathsf{Prefix}\ \pi_1 : \alpha_1 \to \alpha_2 \qquad \ldots \qquad \mathsf{Prefix}\ \pi_n : \alpha_n \to \alpha_1}{\mathsf{fix}_u(X = \pi_1 : \ldots : \pi_n : X) \vdash u : \alpha_1}$$

Multiple branches can also be accomodated. The definition $X = a : X + b : X$ is handled by this rule.

$$\frac{\mathsf{Prefix}\ \pi : \alpha \to \alpha \qquad \mathsf{Prefix}\ \pi' : \alpha \to \alpha}{\mathsf{fix}_u(X = \pi : X + \pi' : X) \vdash u : \alpha}$$

In the running example, this rule can be used to define the plus process. For each natural number n there should be a ground prefix $\mathsf{Prefix}\ n : N \to N$. It is then posible to derive $\mathsf{Prefix}\ n : N^\perp \to N^\perp$ for each n, and

$$\mathsf{Prefix}\ (a, b, c) : N^\perp \parr N^\perp \parr N \to N^\perp \parr N^\perp \parr N$$

for each a, b and c. There is a process

$$\mathsf{fix}_r(X = (0, 0, 0) : X + (0, 1, 1) : X + \cdots) \vdash r : N^\perp \parr N^\perp \parr N$$

in which the sum has a term for every case of addition; to avoid questions about infinite summation, a maximum integer can be introduced so that addition only needs to be defined in finitely many cases. This process is not quite plus, because the interface consists of a single port of type $N^\perp \parr N^\perp \parr N$ rather than three ports of types $N^\perp$, $N^\perp$ and N. This can be rectified with the technique mentioned earlier—three Axiom processes can be combined and used to split up the port. This yields a process

$$\mathsf{fix}_r(X = (0, 0, 0) : X + \cdots) \mathbin{;} ((I_{u,x} \otimes_t^{u,v} I_{v,y}) \otimes_r^{t,w} I_{w,z}) \vdash x : N^\perp, y : N^\perp, z : N$$

which is plus.

As before, it is straightforward to extend the rule to deal with longer sequences of prefix actions. The most general recursive definition in a single variable allows a

process which has already been defined to appear as a branch, as in $X = a : X + P$. This is taken care of by adding the hypothesis $P \vdash x : \alpha$ to the rule.

Finally, the full recursion rule allows several mutually recursive processes to be defined, for example

$$\begin{aligned} X &= a : X + b : Y \\ Y &= c : X + d : Y. \end{aligned}$$

For such a definition, there is again a condition on the prefixes which are involved. If the recursive equations are written $X_i = E_i(\overline{X})$ for $i = 1 \ldots n$, this condition can be expressed by saying that starting with the typed processes $X_i \vdash x_i : \alpha_i$ it must be possible to derive all the $E_i(\overline{X}) \vdash y_i : \alpha_i$. Furthermore, the variables X_i should be *sequential* and *guarded* in the E_j, in the usual process calculus sense [Mil89]. In terms of deriving the E_j in the typed calculus, this means that only prefixing and summation can be used, and every X_i must be under a prefix. Other proved processes can be incorporated into the derivations, as long as they appear as terms in a sum.

It is almost possible to define the axiom process recursively. As will be seen later, the interpretation of the process $I_{x,y} \vdash x : \alpha^\perp, y : \alpha$ in $\mathcal{SP}roc$ is essentially the same synchronous buffer as the identity morphism on the object interpreting the type α. Suppose there are two prefix actions a and b in α, which do not change the type. Then $\mathsf{Prefix}\ a : \alpha \to \alpha$ and $\mathsf{Prefix}\ b : \alpha \to \alpha$ must be derivable, and hence so are $\mathsf{Prefix}\ (a,a) : \alpha^\perp \parr \alpha \to \alpha^\perp \parr \alpha$ and $\mathsf{Prefix}\ (b,b) : \alpha^\perp \parr \alpha \to \alpha^\perp \parr \alpha$. The process

$$\mathsf{fix}_u(X = (a,a) : X + (b,b) : X) \vdash u : \alpha^\perp \parr \alpha$$

should behave as a synchronous buffer in α, as can be checked when the operational semantics of the typed process calculus is defined. However, there is a crucial difference between it and the process $I_{x,y} \vdash x : \alpha^\perp, y : \alpha$. One has two ports whereas the other has a single port of $\parr$ type. The only way to convert the port of type $\alpha^\perp \parr \alpha$ into two ports of types $\alpha^\perp$ and α is by using two axioms to derive the inverse of the par rule, as illustrated earlier, so it is not actually possible to do without axioms and use recursively defined processes instead.

6.3 Operational Semantics

The typed process calculus has an operational semantics, defined by setting up a labelled transition system on proved terms. The transition rules are listed in Figures 6.4 and 6.5. The reason for defining transitions of explicitly typed terms is that the type of a proved term is not unique—this is because the same prefix can exist in many types.

For example, if $\mathsf{Prefix}\ a : \alpha \to \beta$ and $\mathsf{Prefix}\ a : \gamma \to \beta$ then $a : \mathsf{nil}_\beta(x) \vdash x : \alpha$ and $a : \mathsf{nil}_\beta(x) \vdash x : \gamma$ are both proved terms.

Given the discussion of CCS in Chapter 2 and the definition of $\mathcal{SP}roc$ in Chapter 3, the transition rules are what should be expected. In the rules, $\tilde{a}$ stands for a tuple $(a_1, \ldots, a_n)$ of actions. The Prefix rule allows a prefixed action to be performed, and is the base case in the definition of the transition system. The Summation rules allow for non-deterministic choices, and the Recursion rule works by unwinding the definition. In the Cut rule, two processes communicate by performing the same action in the port on which they have been connected, just as in the $\mathcal{SP}roc$ definition of composition. The Tensor rule, similarly, is like the $\mathcal{SP}roc$ definition of tensor on morphisms, and the Par rule simply regroups the tuple of actions. The Mix rule is similar to the Tensor rule, except that the two tuples of actions are concatenated with no regrouping. The Cycle rule is like the Cut rule in that two matching actions disappear. There is no Nil rule, as nil processes have no transitions.

When studying typed programming languages, it is usual to prove a result, known as Subject Reduction, which states that the operational semantics does not change types: if a term M reduces to a term N then M and N have the same type. In the case of languages based on typed λ-calculi, the operational semantics deals with β-reduction; the situation for process calculi is rather different, but Subject Reduction theorems generally hold in the typed systems which have been described, for example [Hon93] and [Gay93]. Thus there should be a similar result for the calculus described in this chapter. However, it turns out that a modified form of Subject Reduction is more appropriate.

In $\mathcal{SP}roc$, types contain safety specifications, which means that if a process P of type A makes a transition $P \xrightarrow{a} Q$ there is no reason why Q should also have type A. In fact, Q has the type A/a defined in Chapter 3, and in general $S_{A/a} \neq S_A$. Since the calculus is intended to be interpretable in $\mathcal{SP}roc$, the syntax reflects this aspect of types: the Prefix judgements express the type changes caused by actions being performed. The presence of Prefix judgements makes it possible to prove a *dynamic* Subject Reduction result, which states precisely how types can change during reduction. This leads to a general observation about the rôle of Subject Reduction theorems in the theory of typed programming languages. The point of having types is that well-typed programs have some correctness property; if this property is preserved by reductions, then programs remain correct during evaluation. But because correctness follows from typability rather than satisfaction of any particular type, the usual Subject Reduction theorems are stronger than is necessary to deduce that correctness is preserved by evaluation: it is only necessary to know that a well-typed program still has *some* type after a reduction step.

Prefix

$$\frac{\mathsf{Prefix}\ \pi : \alpha \to \beta}{\pi : P \vdash x : \alpha \xrightarrow{\pi} P \vdash x : \beta}$$

Cut

$$\frac{P \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P' \vdash \Gamma', x : \beta \qquad Q \vdash \Delta, x : \alpha^{\perp} \xrightarrow{(\tilde{b},a)} Q' \vdash \Delta', x : \beta^{\perp}}{P \mathbin{\dot{x}} Q \vdash \Gamma, \Delta \xrightarrow{(\tilde{a},\tilde{b})} P' \mathbin{\dot{x}} Q' \vdash \Gamma', \Delta'}$$

Axiom

$$\frac{\mathsf{Prefix}\ \pi : \alpha \to \beta}{I_{x,y} \vdash x : \alpha^{\perp}, y : \alpha \xrightarrow{(\pi,\pi)} I_{x,y} \vdash x : \beta^{\perp}, y : \beta}$$

Mix

$$\frac{P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma' \qquad Q \vdash \Delta \xrightarrow{\tilde{b}} Q' \vdash \Delta'}{P \mid Q \vdash \Gamma, \Delta \xrightarrow{(\tilde{a},\tilde{b})} P' \mid Q' \vdash \Gamma', \Delta}$$

Cycle

$$\frac{P \vdash \Gamma, x : \alpha, y : \alpha^{\perp} \xrightarrow{(\tilde{a},a,a)} P' \vdash \Gamma', x : \beta, y : \beta^{\perp}}{P\backslash_{x,y} \vdash \Gamma \xrightarrow{\tilde{a}} P'\backslash_{x,y} \vdash \Gamma'}$$

Par

$$\frac{P \vdash \Gamma, x : \alpha, y : \beta \xrightarrow{(\tilde{a},a,b)} P' \vdash \Gamma', x : \alpha', y : \beta'}{\parr_z^{x,y}(P) \vdash \Gamma, z : \alpha \parr \beta \xrightarrow{(\tilde{a},(a,b))} \parr_z^{x,y}(P') \vdash \Gamma', z : \alpha' \parr \beta'}$$

Tensor

$$\frac{P \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P' \vdash \Gamma', x : \alpha' \qquad Q \vdash \Delta, y : \beta \xrightarrow{(\tilde{b},b)} Q' \vdash \Delta', y : \beta'}{P \otimes_z^{x,y} Q \vdash \Gamma, \Delta, z : \alpha \otimes \beta \xrightarrow{(\tilde{a},\tilde{b},(a,b))} P' \otimes_z^{x,y} Q' \vdash \Gamma', \Delta', z : \alpha' \otimes \beta'}$$

Figure 6.4: Operational Semantics of Typed Process Calculus

Summation

$$\frac{P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'}{P + Q \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'} \qquad \frac{Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'}{P + Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'}$$

Recursion

$$\frac{E_i[\mathsf{fix}^j_{x_j}(\overline{X} = \overline{E}(\overline{X}))/X_j] \vdash x_i : \alpha_i \xrightarrow{\pi} P \vdash x_i : \beta}{\mathsf{fix}^i_{x_i}(\overline{X} = \overline{E}(\overline{X})) \vdash x_i : \alpha_i \xrightarrow{\pi} P \vdash x_i : \beta}$$

Figure 6.5: Operational Semantics of Typed Process Calculus, continued

In general types of process terms are changed by transitions, but there is an aspect of the type of a term which stays the same—essentially the number of ports and the connectives with which they are combined. Thus there is also a *static* Subject Reduction result which makes this formal.

Proposition 6.3 (Dynamic Subject Reduction)
If

$$P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n \xrightarrow{\tilde{a}} Q \vdash y_1 : \beta_1, \ldots, y_m : \beta_m$$

then

- $m = n$
- $\tilde{a} = (a_1, \ldots, a_n)$
- for each $i = 1 \ldots n$, $x_i = y_i$
- for each $i = 1 \ldots n$, $\mathsf{Prefix}\ a_i : \alpha_i \to \beta_i$ is derivable.

Proof: A straightforward induction on the derivation of the transition. □

This result has an interesting consequence for the rules defining the transitions of process term formed by Cut and Cycle. If

$$P \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P' \vdash \Gamma', x : \beta$$

and

$$Q \vdash \Delta, x : \alpha^{\perp} \xrightarrow{(\tilde{b},a)} Q' \vdash \Delta', x : \gamma^{\perp}$$

then it must be the case that $\gamma = \beta$. This means that if P and Q can be connected together, and they make matching transitions to P' and Q', then P' and Q' must also have types which allow them to be connected. A similar comment applies to Cycle.

The statement of the Static Subject Reduction result requires a function θ which maps typed interfaces into formal type constructions involving a single ground type $*$, as follows.

$$\begin{array}{rcll} \theta(\gamma) & \stackrel{\text{def}}{=} & * & \text{if } \gamma \text{ is a ground type} \\ \theta(\alpha^{\perp}) & \stackrel{\text{def}}{=} & \theta(\alpha)^{\perp} & \\ \theta(\circ\,\alpha) & \stackrel{\text{def}}{=} & \theta(\alpha) & \\ \theta(\alpha \otimes \beta) & \stackrel{\text{def}}{=} & \theta(\alpha) \otimes \theta(\beta) & \\ \theta(\alpha \parr \beta) & \stackrel{\text{def}}{=} & \theta(\alpha) \parr \theta(\beta) & \\ \theta(x_1 : \alpha_1, \ldots, x_n : \alpha_n) & \stackrel{\text{def}}{=} & \theta(\alpha_1) \parr \cdots \parr \theta(\alpha_n). & \end{array}$$

Lemma 6.4 If $\mathsf{Prefix}\ \pi : \alpha \to \beta$ is derivable, then $\theta(\alpha) = \theta(\beta)$.

Proof: By induction on the derivation of $\mathsf{Prefix}\ \pi : \alpha \to \beta$. If it is a ground prefix, then both α and β are ground types, hence $\theta(\alpha) = \theta(\beta) = *$. If we have $\mathsf{Prefix}\ * : \circ\,\alpha \to \alpha$ then we immediately have $\theta(\circ\,\alpha) = \theta(\alpha)$. For $\mathsf{Prefix}\ \pi : \alpha^{\perp} \to \beta^{\perp}$ we have $\theta(\alpha^{\perp}) = \theta(\alpha)^{\perp}$ and $\theta(\beta^{\perp}) = \theta(\beta)^{\perp}$, and by the induction hypothesis $\theta(\alpha) = \theta(\beta)$. The remaining cases are similar. □

Proposition 6.5 (Static Subject Reduction) If $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n \xrightarrow{\tilde{a}} Q \vdash \Gamma$ then $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$ and for each i, $\theta(\beta_i) = \theta(\alpha_i)$.

Proof: Suppose $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n \xrightarrow{\tilde{a}} Q \vdash \Gamma$. By the Dynamic Subject Reduction result, $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$, $\tilde{a} = (a_1, \ldots, a_n)$ and for each i, $\mathsf{Prefix}\ a_i : \alpha_i \to \beta_i$ is derivable. By Lemma 6.4, $\theta(\alpha_i) = \theta(\beta_i)$ for each i. □

Once the operational semantics of the calculus has been defined, it is natural to use strong bisimulation as the notion of equivalence. It is defined as in Chapter 2 and [Mil89] except that it is parameterised on the type. Let Γ range over lists $\alpha_1, \ldots, \alpha_n$ in which the α_i are types generated by a process signature $\mathcal{S}g$. A *typed bisimulation* over $\mathcal{S}g$ is a collection of relations R_Γ on the set of proved processes $P \vdash \Gamma$, such that

- if $(P, Q) \in R_\Gamma$ then whenever $P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'$ there is $Q' \vdash \Gamma'$ such that $Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'$ and $(P', Q') \in R_{\Gamma'}$; and whenever $Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'$ there is $P' \vdash \Gamma'$ such that $P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'$ and $(P', Q') \in R_{\Gamma'}$.

The largest typed bisimulation is *strong typed bisimulation* or just *strong bisimulation*, and is denoted by $\sim_\Gamma$. Note that writing $P \vdash \Gamma$ means that $\Gamma = x_1 : \alpha_1, \ldots, x_n : \alpha_n$ but in a typed bisimulation R_Γ only the types (not the names) in Γ are relevant.

If $P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'$ and $Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Delta$ then Dynamic Subject Reduction implies that $\Delta = \Gamma'$, so the condition that $Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'$ is not a constraint. However,

if strong bisimulation were not parameterised on types, the fact that the same action can occur as a prefix in several types would mean that two processes of different types could be strongly bisimilar. While this may not necessarily be a bad thing, it would make it more difficult to establish connections between the operational semantics and the categorical semantics to be defined in the next section.

If $P \vdash \Gamma$ and $Q \vdash \Gamma$ are proved processes which are bisimilar, it will often be convenient to indicate this fact by writing $P \sim Q \vdash \Gamma$ rather than $P \sim_\Gamma Q$.

The first property of strong bisimulation which needs to be established is that it is a congruence, i.e. that all the process constructions preserve it. This is essential if strong bisimulation is to be used as an equivalence with respect to which the process constructions are well-defined. Proving that strong bisimulation is a congruence is straightforward, just as it is in CCS.

Proposition 6.6 Strong bisimulation is a congruence.

Proof: The proof has a case for each process construction.

Prefix Define a relation R by

$$R \stackrel{\text{def}}{=} \{((\pi : P_1 \vdash x : \alpha), (\pi : P_2 \vdash x : \alpha)) \mid (P_1 \sim P_2 \vdash x : \beta) \wedge (\mathsf{Prefix}\ \pi : \alpha \to \beta)\}.$$

It is easy to check that R is a bisimulation. Hence if $P_1 \sim P_2 \vdash x : \beta$ and $\mathsf{Prefix}\ \pi : \alpha \to \beta$ then $\pi : P_1 \sim \pi : P_2 \vdash x : \alpha$.

Cut Define a relation R by

$$\begin{aligned} R \stackrel{\text{def}}{=} \{&((P_1 \mathbin{\dot{x}} Q_1 \vdash \Gamma, \Delta), (P_2 \mathbin{\dot{x}} Q_2 \vdash \Gamma, \Delta)) \mid \\ &(P_1 \sim P_2 \vdash \Gamma, x : \alpha) \wedge (Q_1 \sim Q_2 \vdash \Delta, x : \alpha^\perp)\}. \end{aligned}$$

A transition of $P_1 \mathbin{\dot{x}} Q_1$ is

$$P_1 \mathbin{\dot{x}} Q_1 \vdash \Gamma, \Delta \xrightarrow{(\tilde{a},\tilde{b})} P_1' \mathbin{\dot{x}} \vdash \Gamma', \Delta'$$

where $P_1 \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P_1' \vdash \Gamma', x : \beta$ and $Q_1 \vdash \Delta, x : \alpha^\perp \xrightarrow{(\tilde{b},a)} Q_1' \vdash \Delta', x : \beta^\perp$. Because $P_1 \sim P_2$ and $Q_1 \sim Q_2$, we have

$$P_2 \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P_2' \vdash \Gamma', x : \beta$$

and

$$Q_2 \vdash \Delta, x : \alpha^\perp \xrightarrow{(\tilde{b},a)} Q_2' \vdash \Delta', x : \beta^\perp$$

with $P_2' \sim P_1'$ and $Q_2' \sim Q_1'$. So

$$P_2 \mathbin{\dot{x}} Q_2 \vdash \Gamma, \Delta \xrightarrow{(\tilde{a},\tilde{b})} P_2' \mathbin{\dot{x}} Q_2' \vdash \Gamma', \Delta'$$

and hence $(P_1' \mathbin{\dot{x}} Q_1', P_2' \mathbin{\dot{x}} Q_2') \in R$. Thus R is a bisimulation.

Tensor This case is similar to the case of Cut. The relation is

$$R \stackrel{\text{def}}{=} \{((P_1 \otimes_z^{x,y} Q_1 \vdash \Gamma, \Delta, z : \alpha \otimes \beta), (P_2 \otimes_z^{x,y} Q_2 \vdash \Gamma, \Delta, z : \alpha \otimes \beta)) \mid (P_1 \sim P_2 \vdash \Gamma, x : \alpha) \wedge (Q_1 \sim Q_2 \vdash \Delta, y : \beta)\}.$$

Par This case is similar to that of Tensor. The relation is

$$R \stackrel{\text{def}}{=} \{((\wp_z^{x,y}(P_1) \vdash \Gamma, z : \alpha \wp \beta), (\wp_z^{x,y}(P_2) \vdash \Gamma, z : \alpha \wp \beta)) \mid P_1 \sim P_2 \vdash \Gamma, x : \alpha, y : \beta\}.$$

Mix This case is again similar to that of Tensor, with the relation

$$R \stackrel{\text{def}}{=} \{((P_1 \mid Q_1 \vdash \Gamma, \Delta), (P_2 \mid Q_2 \vdash \Gamma, \Delta)) \mid (P_1 \sim P_2 \vdash \Gamma) \wedge (Q_1 \sim Q_2 \vdash \Delta)\}.$$

Cycle This case is similar to that of Par, with

$$R \stackrel{\text{def}}{=} \{((P_1 \backslash_{x,y} \vdash \Gamma), (P_2 \backslash_{x,y} \vdash \Gamma)) \mid P_1 \sim P_2 \vdash \Gamma, x : \alpha, y : \alpha^{\perp}\}.$$

Summation For this case, a standard bisimulation argument is used with the relation

$$R \stackrel{\text{def}}{=} \{((P_1 + Q_1), (P_2 + Q_2)) \mid (P_1 \sim P_2) \wedge (Q_1 \sim Q_2)\}.$$

Recursion Again, the proof involves showing that a suitable relation is a bisimulation:

$$R \stackrel{\text{def}}{=} \{((\mathsf{fix}_{x_i}^i(\overline{X} = \overline{E}(\overline{X})) \vdash x_i : \alpha_i), (\mathsf{fix}_{x_i}^i(\overline{X} = \overline{E}'(\overline{X})) \vdash x_i : \alpha_i)) \mid \forall j. E_j(\overline{X}) \sim E'_j(\overline{X}) \vdash x_j : \alpha_j\}.$$

□

The names attached to the ports of a process do not change as the process makes transitions, and so have no impact on its behaviour. They exist to facilitate the syntactic description of connections between processes. Substitution of names is a necessary operation on processes, for example to avoid clashes when several instances of the same process are used in a single system. It follows easily from the definition of the operational semantics that substitutions do not change strong bisimulation classes.

Proposition 6.7 If $P \vdash \Gamma, x : \alpha$ then $P \sim_{\Gamma,\alpha} P[y/x]$.

There is a large collection of instances of strong bisimulation which can easily be established, and which it is useful to have available when working with processes. These arise for a number of different reasons. First of all, the syntax of the calculus allows

Commutativity of Cut

$$\frac{Q \vdash \Gamma, x : \alpha \qquad R \vdash \Delta, x : \alpha^{\perp}}{Q \;{}_{\dot{x}}\; R \sim R \;{}_{\dot{x}}\; Q \vdash \Gamma, \Delta}$$

Associativity of Tensor

$$\frac{Q \vdash \Gamma, x : \alpha \qquad R \vdash y : \beta, \Delta, u : \gamma \qquad S \vdash v : \delta, \Theta}{(Q \otimes_z^{x,y} R) \otimes_w^{u,v} S \sim Q \otimes_z^{x,y} (R \otimes_w^{u,v} S) \vdash \Gamma, z : \alpha \otimes \beta, \Delta, w : \gamma \otimes \delta, \Theta}$$

Associativity of Cut

$$\frac{Q \vdash \Gamma, x : \alpha \qquad R \vdash x : \alpha^{\perp}, \Delta, y : \beta \qquad S \vdash y : \beta^{\perp}, \Theta}{(Q \;{}_{\dot{x}}\; R) \;{}_{\dot{y}}\; S \sim Q \;{}_{\dot{x}}\; (R \;{}_{\dot{y}}\; S) \vdash \Gamma, \Delta, \Theta}$$

Commutativity of Par

$$\frac{Q \vdash \Gamma, x : \alpha, y : \beta, u : \gamma, v : \delta}{\wp_z^{x,y}(\wp_w^{u,v}(Q)) \sim \wp_w^{u,v}(\wp_z^{x,y}(Q)) \vdash \Gamma, z : \alpha \wp \beta, w : \gamma \wp \delta}$$

Structural Tensor-Cut

$$\frac{Q \vdash \Gamma, x : \alpha \quad R \vdash y : \beta, \Delta, u : \gamma \quad S \vdash u : \gamma^{\perp}, \Theta}{(Q \otimes_z^{x,y} R) \;{}_{\dot{u}}\; S \sim Q \otimes_z^{x,y} (R \;{}_{\dot{u}}\; S) \vdash \Gamma, z : \alpha \otimes \beta, \Delta, \Theta}$$

Structural Par-Tensor

$$\frac{Q \vdash \Gamma, x : \alpha, y : \beta, u : \gamma \qquad R \vdash \Delta, v : \delta}{\wp_z^{x,y}(Q) \otimes_w^{u,v} R \sim \wp_z^{x,y}(Q \otimes_w^{u,v} R) \vdash \Gamma, \Delta, z : \alpha \wp \beta, w : \gamma \otimes \delta}$$

Structural Par-Cut

$$\frac{Q \vdash \Gamma, x : \alpha, y : \beta, u : \gamma \qquad R \vdash \Delta, u : \gamma^{\perp}}{\wp_z^{x,y}(Q) \;{}_{\dot{u}}\; R \sim \wp_z^{x,y}(Q \;{}_{\dot{u}}\; R) \vdash \Gamma, \Delta, z : \alpha \wp \beta}$$

Figure 6.6: Rules for Bisimulation

Summation

$$\frac{Q \vdash \Gamma \qquad R \vdash \Gamma}{Q + R \sim R + Q \vdash \Gamma}$$

$$\frac{Q \vdash \Gamma \qquad R \vdash \Gamma \qquad S \vdash \Gamma}{Q + (R + S) \sim (Q + R) + S \vdash \Gamma} \qquad \frac{Q \vdash \Gamma}{Q + Q \sim Q \vdash \Gamma}$$

Tensor Distributivity

$$\frac{Q \vdash \Gamma, x : \alpha \qquad R \vdash \Delta, y : \beta \qquad S \vdash \Delta, y : \beta}{Q \otimes_z^{x,y} (R + S) \sim (Q \otimes_z^{x,y} R) + (Q \otimes_z^{x,y} S) \vdash \Gamma, \Delta, z : \alpha \otimes \beta}$$

Par Distributivity

$$\frac{Q \vdash \Gamma, x : \alpha, y : \beta \qquad R \vdash \Gamma, x : \alpha, y : \beta}{\parr_z^{x,y}(Q + R) \sim \parr_z^{x,y}(Q) + \parr_z^{x,y}(R) \vdash \Gamma, z : \alpha \parr \beta}$$

Cut Distributivity

$$\frac{Q \vdash \Gamma, x : \alpha \qquad R \vdash \Delta, x : \alpha^\perp \qquad S \vdash \Delta, x : \alpha^\perp}{Q \dot{}_x (R + S) \sim (Q \dot{}_x R) + (Q \dot{}_x S) \vdash \Gamma, \Delta}$$

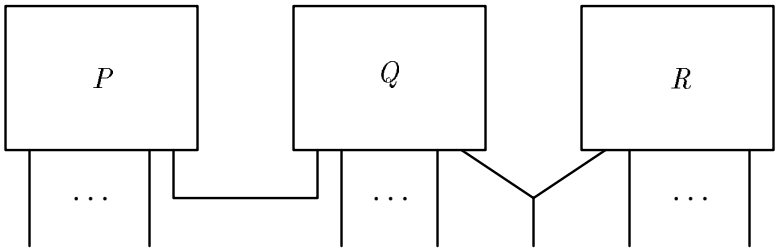
Cut Elimination Equations

$$\frac{Q \vdash \Gamma, x : \alpha \quad R \vdash \Delta, y : \beta \quad S \vdash \Theta, x : \alpha^\perp, y : \beta^\perp}{(Q \otimes_z^{x,y} R) \dot{}_z \parr_z^{x,y}(S) \sim Q \dot{}_x (S \dot{}_y R) \vdash \Gamma, \Delta, \Theta}$$

$$\frac{Q \vdash \Gamma, x : \alpha}{Q \dot{}_x I_{x,y} \sim Q[y/x] \vdash \Gamma, y : \alpha}$$

Figure 6.7: Rules for Bisimulation, continued

certain process configurations to be described in several different ways. For example, this diagram represents the result of taking three processes, forming a connection between two of them and then using $\otimes$ to combine a port of a third process.



The final configuration can be described by either of the terms $(P \mathbin{\dot{x}} Q) \otimes_w^{u,v} R$ and $P \mathbin{\dot{x}} (Q \otimes_w^{u,v} R)$, if the ports are suitably named. When a textual syntax is used to represent two- or three-dimensional structures, such artificial distinctions are inevitably introduced. Exactly the same problem occurs in linear logic when sequent proofs are used instead of proof nets. Figure 6.6 lists the bisimulation instances of this form. They are presented as inference rules, with proved processes as the premises and a bisimulation instance as the conclusion. The next group of rules relates to the properties of summation. The $+$ operation is associative, commutative and idempotent; furthermore, because the calculus is synchronous, sums are preserved by Tensor, Par and Cut. Figure 6.7 lists the corresponding proof rules. There are two cases of strong bisimulation in which the processes concerned would be related by cut-elimination steps, in the Proofs as Processes interpretation. One case is a cut between a Tensor port and a Par port, and the other is between any process and an identity axiom. Under the Curry-Howard Isomorphism, cut elimination corresponds to β-reduction in the λ-calculus, but transitions in the process calculus are orthogonal to β-reductions. Semantically, β-reduction is part of equality in an interaction category; the relational nature of composition means that arbitrary functional computations can take place in a single time step. Figure 6.7 also contains the cut elimination instances of strong bisimulation.

6.4 Categorical Semantics

The typed process calculus can be given a semantics in a suitably structured category. The structure required is that of a synchronous interaction category, in the sense of Chapter 3, with some additions. Explicitly, let $\mathbb{C}$ be a compact closed linear category with countable biproducts, such that the functor $!(\bigoplus_{n>0} \circ^n)$ has the multiple UFPP. In $\mathbb{C}$ the biproducts yield a commutative monoid structure $(+, \mathsf{nil})$ on each homset, and $+$ is preserved by $\otimes$, $(-)^\perp$ and hence $\parr$. There is also a partial order $\leqslant$ on each

homset, defined by

$$f \leqslant g \stackrel{\text{def}}{\Leftrightarrow} \exists h. f + h = g.$$

Lemma 6.8 1. For any $p : A \to B$, $\mathsf{nil}_{A \multimap B} \leqslant p$.

2. If $p \leqslant q$ then for any f, $f \otimes p \leqslant f \otimes q$ and $p \otimes f \leqslant q \otimes f$.

3. If $p \leqslant q$ then for any f, $f \parr p \leqslant f \parr q$ and $p \parr f \leqslant q \parr f$.

4. If $p \leqslant q$ then $p^\perp \leqslant q^\perp$.

Proof: Straightforward manipulation of the definition of $\leqslant$, together with the fact that $\otimes$, $\parr$ and $(-)^\perp$ all preserve $+$. □

A *structure* in $\mathbb{C}$ for a process signature $\mathcal{S}g$ is specified by the following data.

- For each ground type γ of $\mathcal{S}g$, an object $[\![\gamma]\!]$ of $\mathbb{C}$. The function $[\![\cdot]\!]$ is then extended inductively to all types by

$$\begin{aligned} [\![\alpha^\perp]\!] &\stackrel{\text{def}}{=} [\![\alpha]\!]^\perp \\ [\![\alpha \otimes \beta]\!] &\stackrel{\text{def}}{=} [\![\alpha]\!] \otimes [\![\beta]\!] \\ [\![\alpha \parr \beta]\!] &\stackrel{\text{def}}{=} [\![\alpha]\!] \parr [\![\beta]\!] \\ [\![\circ \alpha]\!] &\stackrel{\text{def}}{=} \circ [\![\alpha]\!]. \end{aligned}$$

  and to lists of named ports by

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \parr \ldots \parr [\![A_n]\!].$$

- For each ground prefix $\mathsf{Prefix}\ \sigma : \gamma \to \gamma'$, a pair of morphisms

$$[\![\mathsf{Prefix}\ \sigma : \gamma \to \gamma']\!] : \circ[\![\gamma']\!] \longleftrightarrow [\![\gamma]\!] : [\![\mathsf{Prefix}\ \sigma : \gamma \to \gamma']\!]'$$

  such that

$$[\![\mathsf{Prefix}\ \sigma : \gamma \to \gamma']\!] \,;\, [\![\mathsf{Prefix}\ \sigma : \gamma \to \gamma']\!]' = \mathsf{id}_{\circ[\![\gamma']\!]}$$

  and

$$\mathsf{id}_{[\![\gamma]\!]} = \sum_{\mathsf{Prefix}\ \pi : \gamma \to \gamma'} [\![\mathsf{Prefix}\ \pi : \gamma \to \gamma']\!]' \,;\, [\![\mathsf{Prefix}\ \pi : \gamma \to \gamma']\!].$$

The function $[\![\cdot]\!]$ is extended to all prefix judgements by

$$\begin{aligned} [\![\mathsf{Prefix}\ * : \circ\alpha \to \alpha]\!] &\stackrel{\text{def}}{=} \mathsf{id}_{\circ[\![\alpha]\!]} \\ [\![\mathsf{Prefix}\ * : \circ\alpha \to \alpha]\!]' &\stackrel{\text{def}}{=} \mathsf{id}_{\circ[\![\alpha]\!]} \\ [\![\mathsf{Prefix}\ \pi : \alpha^\perp \to \beta^\perp]\!] &\stackrel{\text{def}}{=} [\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]'^\perp \\ [\![\mathsf{Prefix}\ \pi : \alpha^\perp \to \beta^\perp]\!]' &\stackrel{\text{def}}{=} [\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]^\perp \end{aligned}$$

$$[\![\mathsf{Prefix}\ (\pi,\pi') : \alpha \otimes \alpha' \to \beta \otimes \beta']\!] \stackrel{\text{def}}{=} \mathsf{mon}^{-1}\,;([\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!] \otimes [\![\mathsf{Prefix}\ \pi' : \alpha' \to \beta']\!])$$

$$[\![\mathsf{Prefix}\ (\pi,\pi') : \alpha \otimes \alpha' \to \beta \otimes \beta']\!]' \stackrel{\text{def}}{=} ([\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]' \otimes [\![\mathsf{Prefix}\ \pi' : \alpha' \to \beta']\!]')\,;\mathsf{mon}$$

$$[\![\mathsf{Prefix}\ (\pi,\pi') : \alpha \parr \alpha' \to \beta \parr \beta']\!] \stackrel{\text{def}}{=} \mathsf{mon}^{\perp}\,;([\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!] \parr [\![\mathsf{Prefix}\ \pi' : \alpha' \to \beta']\!])$$

$$[\![\mathsf{Prefix}\ (\pi,\pi') : \alpha \parr \alpha' \to \beta \parr \beta']\!]' \stackrel{\text{def}}{=} ([\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]' \parr [\![\mathsf{Prefix}\ \pi' : \alpha' \to \beta']\!]')\,;\mathsf{mon}^{-1^{\perp}}.$$

Proposition 6.9 If $\mathsf{Prefix}\ \pi : \alpha \to \beta$ is derivable, then

$$[\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]\,;[\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]' = \mathsf{id}_{\circ[\![\beta]\!]}$$

and

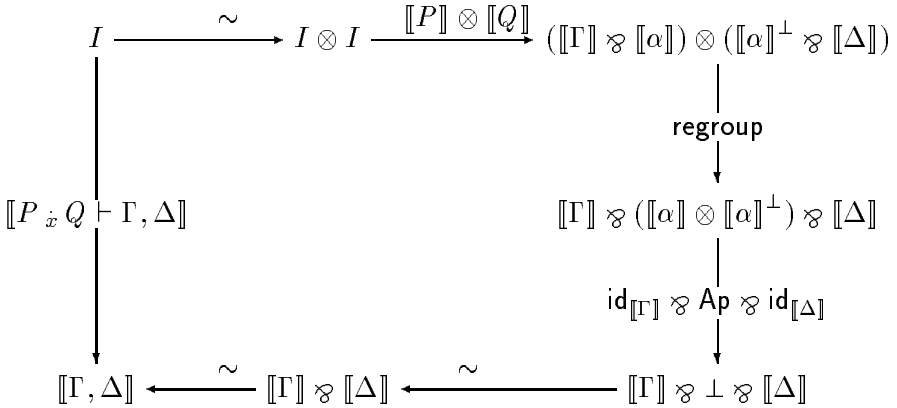
$$[\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!]'\,;[\![\mathsf{Prefix}\ \pi : \alpha \to \beta]\!] \leqslant \mathsf{id}_{[\![\alpha]\!]}.$$

Proof: By induction on the derivation of $\mathsf{Prefix}\ \pi : \alpha \to \beta$, using the fact that $\otimes$, $\parr$ and $(-)^{\perp}$ preserve $\leqslant$. □

For each proved process $P \vdash \Gamma$ generated by $\mathcal{S}g$, there is a morphism $[\![P \vdash \Gamma]\!] : I \to [\![\Gamma]\!]$ in $\mathbb{C}$, defined by induction on the derivation of $P \vdash \Gamma$ as follows.

Axiom $[\![I_{x,y} \vdash x : \alpha^{\perp}, y : \alpha]\!] \stackrel{\text{def}}{=} \Lambda(\mathsf{unitl}_{[\![\alpha]\!]}) : I \to [\![\alpha]\!]^{\perp} \parr [\![\alpha]\!]$.

Cut If $P \vdash \Gamma, x : \alpha$ and $Q \vdash \Delta, x : \alpha^{\perp}$ then $[\![P \stackrel{.}{x} Q \vdash \Gamma, \Delta]\!]$ is defined by

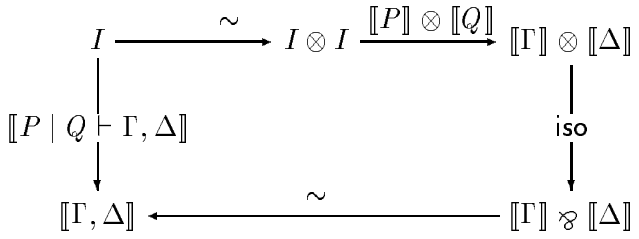


For any A, B, C and D, $\mathsf{regroup} : (A \parr B) \otimes (C \parr D) \to A \parr ((B \otimes C) \parr D)$ (which can also be written $\mathsf{regroup} : (A^{\perp} \multimap B) \otimes (C^{\perp} \multimap D) \to A^{\perp} \multimap ((B \multimap C^{\perp}) \multimap D)$) is a canonical morphism defined by

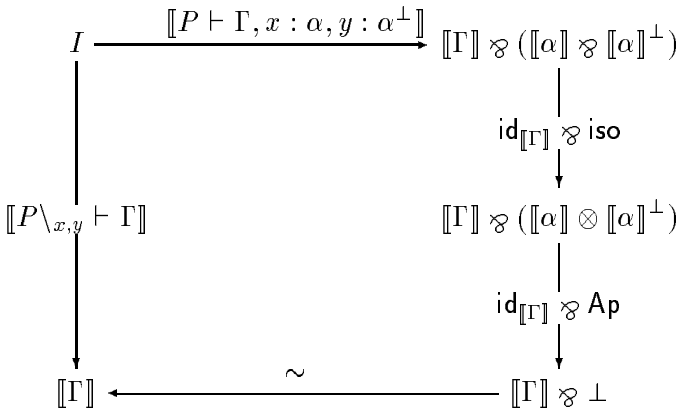
$$\mathsf{regroup} \stackrel{\text{def}}{=} \Lambda(\Lambda(\mathsf{iso}\,;(\mathsf{id} \otimes ((\mathsf{Ap} \otimes \mathsf{id})\,;\mathsf{symm}\,;\mathsf{Ap}))\,;\mathsf{Ap}))$$

with iso a canonical isomorphism.

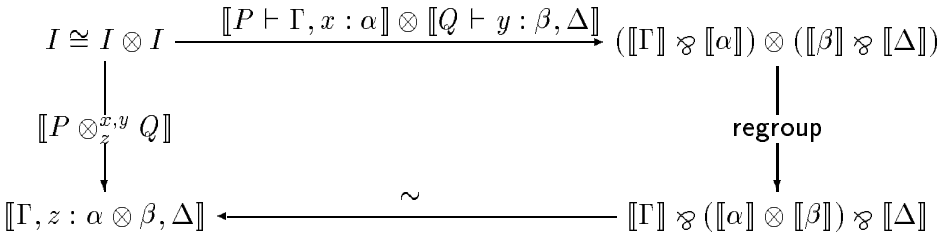
Mix $[\![P \mid Q \vdash \Gamma, \Delta]\!]$ is defined by



Cycle If $P \vdash \Gamma, x : \alpha, y : \alpha^\perp$ then $[\![P\backslash_{x,y} \vdash \Gamma]\!]$ is defined by



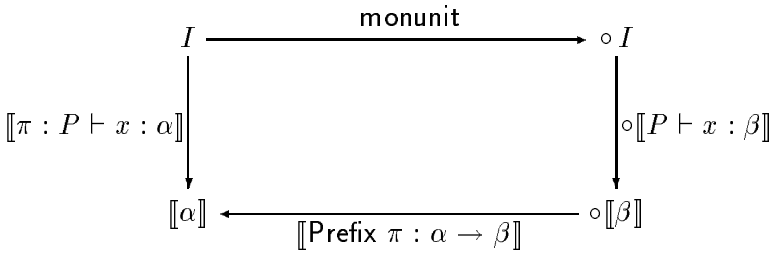
Tensor $[\![P \otimes_z^{x,y} Q \vdash \Gamma, z : \alpha \otimes \beta, \Delta]\!]$ is defined by



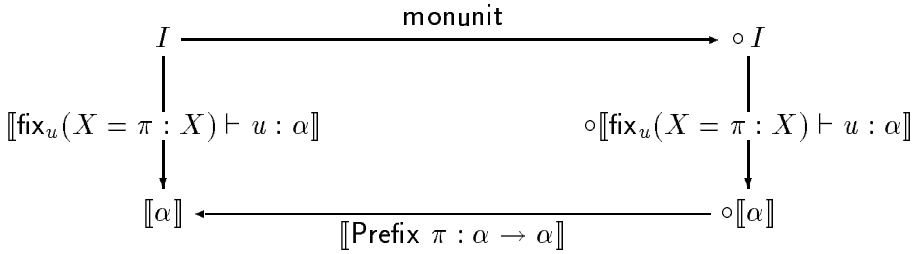
Par $[\![\parr_z^{x,y}(P) \vdash \Gamma, z : \alpha \parr \beta]\!] \stackrel{\text{def}}{=} [\![P \vdash \Gamma, x : \alpha, y : \beta]\!] : I \to [\![\Gamma]\!] \parr [\![\alpha]\!] \parr [\![\beta]\!]$.

Summation The Sum rule is interpreted by means of the operation $+$ on the homsets of $\mathbb{C}$: $[\![P + Q \vdash \Gamma]\!] \stackrel{\text{def}}{=} [\![P \vdash \Gamma]\!] + [\![Q \vdash \Gamma]\!]$.

Prefix $[\![\pi : P \vdash x : \alpha]\!]$ is defined by

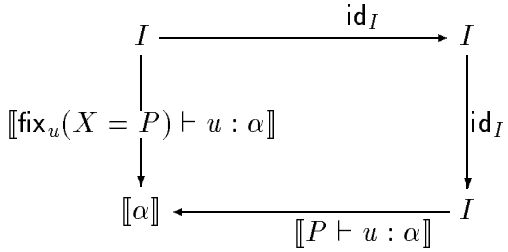


Recursion The semantics of a fix expression is defined in terms of the UFPP of a suitable functor. It is worth considering a few simpler forms of the recursion rule first, in order to see how an appropriate functor is constructed for each use of fix. In the very simplest case, $[\![\mathsf{fix}_u(X = \pi : X) \vdash u : \alpha]\!]$ is defined by this UFPP diagram.



This is exactly the same as the example given in Chapter 3 of how the UFPP can be used to construct recursively defined processes in *SProc*. For a definition with two prefixes in sequence, the UFPP of the functor $\circ\circ$ is used. If the process is $\mathsf{fix}_u(X = \pi_1 : \pi_2 : X) \vdash u : \alpha$ then there are morphisms $[\![\mathsf{Prefix}\ \pi_1 : \alpha \to \beta]\!] : \circ[\![\beta]\!] \to [\![\alpha]\!]$ and $[\![\mathsf{Prefix}\ \pi_2 : \beta \to \alpha]\!] : \circ[\![\alpha]\!] \to [\![\beta]\!]$, and the morphism $\circ\circ[\![\alpha]\!] \to [\![\alpha]\!]$ required for the UFPP diagram is $\circ[\![\mathsf{Prefix}\ \pi_2 : \beta \to \alpha]\!]\,;[\![\mathsf{Prefix}\ \pi_1 : \alpha \to \beta]\!]$.

In this way, the UFPP of $\circ^n$ can be used to interpret a recursive definition in a single variable, with a single sequence of prefix actions guarding the variable. There is also a trivial case: if $P \vdash u : \alpha$ is a proved process, then $[\![\mathsf{fix}_u(X = P) \vdash u : \alpha]\!] \stackrel{\text{def}}{=} [\![P \vdash u : \alpha]\!] : I \to [\![\alpha]\!]$. This case can be handled by the UFPP of the constant functor at I, as $[\![\mathsf{fix}_u(X = P) \vdash u : \alpha]\!]$ is the unique morphism $I \to [\![\alpha]\!]$ such that



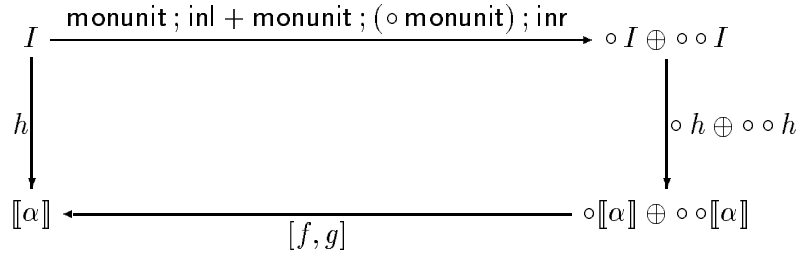
commutes. It is useful to take this view of the trivial recursive definition, as it means that every recursive definition is covered by an application of the UFPP.

Recursive definitions with several branches combined by $+$ are interpreted via the UFPP of functors constructed from iterates of $\circ$ by $\oplus$. For example, consider

$$\mathsf{fix}_u(X = a : X + b : c : X) \vdash u : \alpha.$$

As before, there are morphisms $f : \circ[\![\alpha]\!] \to [\![\alpha]\!]$ and $g : \circ\circ[\![\alpha]\!] \to [\![\alpha]\!]$ defined from the interpretations of the prefix judgements. Then $[\![\mathsf{fix}_u(X = a : X + b : c : X) \vdash u : \alpha]\!]$ is

the unique h such that this UFPP diagram commutes.



The same construction applies if one of the branches is a constant process; for example, to interpret $\mathsf{fix}_u(X = a : X + P) \vdash u : \alpha$ the functor $A \mapsto I \oplus \circ A$ is used.

The most general case, in which there are several mutually recursive definitions, makes use of the multiple UFPP and also the linear logic exponentials. Consider the example

$$\begin{aligned} X &= a : X + b : Y \\ Y &= c : X + d : Y \end{aligned}$$

in which the intended type of X is α and that of Y is β. There are morphisms $[\![\mathsf{Prefix}\ a : \alpha \to \alpha]\!] : \circ[\![\alpha]\!] \to [\![\alpha]\!]$ and $[\![\mathsf{Prefix}\ b : \alpha \to \beta]\!] : \circ[\![\beta]\!] \to [\![\alpha]\!]$ and hence

$$h \stackrel{\text{def}}{=} f + g : {!}\circ[\![\alpha]\!] \otimes {!}\circ[\![\beta]\!] \to [\![\alpha]\!]$$

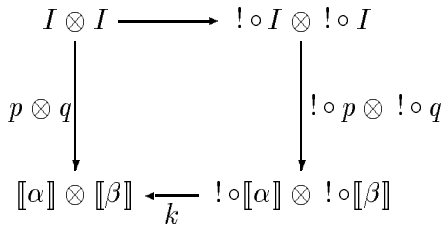
where f is defined by

$${!}\circ[\![\alpha]\!] \otimes {!}\circ[\![\beta]\!] \xrightarrow{\mathsf{id} \otimes \mathsf{weak}} {!}\circ[\![\alpha]\!] \otimes I \cong {!}\circ[\![\alpha]\!] \xrightarrow{\mathsf{der}} \circ[\![\alpha]\!] \xrightarrow{[\![\mathsf{Prefix}\ a : \alpha \to \alpha]\!]} [\![\alpha]\!]$$

and g by

$${!}\circ[\![\alpha]\!] \otimes {!}\circ[\![\beta]\!] \xrightarrow{\mathsf{weak} \otimes \mathsf{id}} I \otimes {!}\circ[\![\beta]\!] \cong {!}\circ[\![\beta]\!] \xrightarrow{\mathsf{der}} \circ[\![\beta]\!] \xrightarrow{[\![\mathsf{Prefix}\ b : \alpha \to \beta]\!]} [\![\alpha]\!].$$

Similarly there is a morphism $h' : {!}\circ[\![\alpha]\!] \otimes {!}\circ[\![\beta]\!] \to [\![\beta]\!]$ derived from the c and d prefixes. Hence there is a morphism $k : {!}\circ[\![\alpha]\!] \otimes {!}\circ[\![\beta]\!] \to [\![\alpha]\!] \otimes [\![\beta]\!]$ defined by applying contraction to ${!}\circ[\![\alpha]\!]$ and ${!}\circ[\![\beta]\!]$ and then applying $h \otimes h'$. Finally, the multiple UFPP of ${!}\circ$ gives unique $p : I \to [\![\alpha]\!]$ and $q : I \to [\![\beta]\!]$ such that this diagram commutes.



Then p and q are the interpretations of $\mathsf{fix}^1_u(X = a : X + b : Y, Y = c : X + d : Y) \vdash u : \alpha$ and $\mathsf{fix}^2_u(X = a : X + b : Y, Y = c : X + d : Y) \vdash u : \beta$. The morphism $I \to {!}\circ I$

used in the top row of the diagram is $\mathsf{iso}^{-1}\,;\mathsf{iso}^{!}\,;\,!\,\mathsf{monunit}$, where $\mathsf{iso} : !\,\mathbf{1} \to I$ is the isomorphism (2.1) of Chapter 2.

The semantics of the general recursion rule, which may define any number of mutually recursive processes which use arbitrary depths of prefixing, uses a similar construction to the above with the functor F defined by

$$FA \stackrel{\text{def}}{=} I \oplus \circ A \oplus \circ^2 A \oplus \cdots$$

instead of $\circ$. Given n typed process expressions $E_1 : \alpha_1, \ldots, E_n : \alpha_n$ in variables $X_1, \ldots, X_n$, each E_j defines a morphism $!\,F[\![\alpha_1]\!] \otimes \cdots \otimes !\,F[\![\alpha_n]\!] \to [\![\alpha_j]\!]$. These morphisms are constructed as in the simpler case above, with the addition that projection from F is used to select the correct depth of prefixing for each variable. Again using a similar construction to the one above, there is a morphism

$$k : !\,F[\![\alpha_1]\!] \otimes \cdots \otimes !\,F[\![\alpha_n]\!] \to [\![\alpha_1]\!] \otimes \cdots \otimes [\![\alpha_n]\!].$$

The multiple UFPP of $!F$ means that there are unique $p_i : I \to [\![\alpha_i]\!]$ such that

$$\begin{array}{ccc} I \otimes \cdots \otimes I & \longrightarrow & !FI \otimes \cdots \otimes !FI \\ {\scriptstyle p_1 \otimes \cdots \otimes p_n}\downarrow & & \downarrow{\scriptstyle !F(p_1) \otimes \cdots \otimes !F(p_n)} \\ [\![\alpha_1]\!] \otimes \cdots \otimes [\![\alpha_n]\!] & \underset{k}{\longleftarrow} & !F[\![\alpha_1]\!] \otimes \cdots \otimes !F[\![\alpha_n]\!] \end{array}$$

commutes. Then $[\![\mathsf{fix}^i_{x_i}(\overline{X} = \overline{E}(\overline{X})) \vdash x_i : \alpha_i]\!] \stackrel{\text{def}}{=} p_i$. The morphism $I \to !FI$ is

$$\mathsf{iso}^{-1}\,;\mathsf{iso}^{!}\,;\,!\,(\langle f_n \rangle_{n \geqslant 0})$$

where the $f_n : I \to \circ^n I$ are defined by

$$\begin{array}{rcll} f_0 & \stackrel{\text{def}}{=} & \mathsf{id}_I & \\ f_{r+1} & \stackrel{\text{def}}{=} & f_r\,;(\circ^r\,\mathsf{monunit}) & (r \geqslant 0). \end{array}$$

6.5 Semantics in $\mathcal{SP}roc$

As has been stated before, the typed process calculus is intended to have an interpretation in $\mathcal{SP}roc$. There is a structure in $\mathcal{SP}roc$ for any process signature $\mathcal{S}g$, defined as follows.

A process signature $\mathcal{S}g$ defines a labelled transition system whose states are the ground types of $\mathcal{S}g$, whose labels are the actions appearing in the ground prefixes, and with $\gamma \xrightarrow{\pi} \gamma' \iff \mathsf{Prefix}\ \pi : \gamma \to \gamma'$. If this labelled transition system is considered as a

directed graph with labelled edges, there are a number of connected components; for each component c, there is a set Σ_c of actions consisting of the labels which occur in c.

Given a ground type γ, let $c(\gamma)$ be the connected component containing γ in the labelled transition system. The object $[\![\gamma]\!]$ is defined by

$$\begin{aligned} \Sigma_{[\![\gamma]\!]} &\stackrel{\text{def}}{=} \Sigma_c \\ S_{[\![\gamma]\!]} &\stackrel{\text{def}}{=} \{s \mid \exists \gamma'.\gamma \xrightarrow{s}{}^{*} \gamma'\}. \end{aligned}$$

Given a prefix judgement $\mathsf{Prefix}\ \pi : \gamma \to \gamma'$, $[\![\mathsf{Prefix}\ \pi : \gamma \to \gamma']\!] : \circ[\![\gamma']\!] \to [\![\gamma]\!]$ is defined as $(*, \pi) : \mathsf{id}_{[\![\gamma']\!]}$ and $[\![\mathsf{Prefix}\ \pi : \gamma \to \gamma']\!]' : \circ[\![\gamma']\!] \to [\![\gamma]\!]$ is defined as $(\pi, *) : \mathsf{id}_{[\![\gamma']\!]}$. The conditions which these morphisms must satisfy are easily checked.

Proposition 6.10 If $[\![\cdot]\!]$ is the semantics in $\mathcal{SP}roc$, then for every proved process $P \vdash \Gamma$, $[\![P \vdash \Gamma]\!] = \mathsf{tree}(P \vdash \Gamma)[(a_1, \ldots, a_n) \mapsto (*, a_1, \ldots, a_n)]$.

Proof: By induction on the derivation on $P \vdash \Gamma$. In every case, the transition rules for $P \vdash \Gamma$ are the same as those in the $\mathcal{SP}roc$ definition of $[\![P \vdash \Gamma]\!]$. □

Corollary 6.11 If $P \vdash \Gamma$ is a proved process then $\mathsf{traces}(\mathsf{tree}(P \vdash \Gamma)) \subseteq S_{[\![\Gamma]\!]}$.

Corollary 6.12 If $[\![\cdot]\!]$ is the semantics in $\mathcal{SP}roc$, then for all proved processes $P \vdash \Gamma$ and $Q \vdash \Gamma$,

$$P \sim Q \vdash \Gamma \quad \Longleftrightarrow \quad [\![P]\!] = [\![Q]\!].$$

Proof:

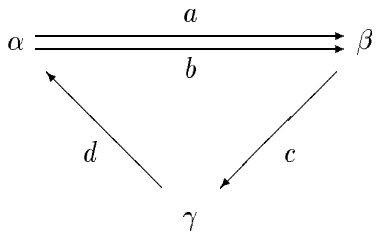
$$\begin{aligned} P \sim Q \vdash \Gamma \quad &\Longleftrightarrow \quad \mathsf{tree}(P \vdash \Gamma) \sim \mathsf{tree}(Q \vdash \Gamma) \\ &\Longleftrightarrow \quad [\![P \vdash \Gamma]\!] \sim [\![Q \vdash \Gamma]\!] && \text{by Propositon 6.10} \\ &\Longleftrightarrow \quad [\![P \vdash \Gamma]\!] = [\![Q \vdash \Gamma]\!] && \text{as } = \text{ and } \sim \text{ coincide in } \mathcal{SP}roc. \end{aligned}$$

□

Corollary 6.11 also contains useful information at a purely syntactic level. It can be deduced from the Dynamic Subject Reduction result, and thus does not depend essentially on the $\mathcal{SP}roc$ semantics. What this means is that setting up the ground prefixes amounts to defining safety specifications for the ground types, and then the rules of the calculus only permit the construction of processes which satisfy these specifications.

To illustrate this point, it is useful to begin with an intended safety specification for some type α. Suppose that the desired set of safe traces consists of all the traces generated by the regular expression $((a|b)cd)^*$, and their prefixes. Introducing the

additional types β and γ means that the safe traces correspond precisely to labelled paths from α in this graph.



The edges of the graph define four ground prefixes:

$$\begin{aligned}
&\mathsf{Prefix}\ a : \alpha \to \beta \\
&\mathsf{Prefix}\ b : \alpha \to \beta \\
&\mathsf{Prefix}\ b : \beta \to \gamma \\
&\mathsf{Prefix}\ c : \gamma \to \alpha.
\end{aligned}$$

Now, there are a great many processes of type α, for example

$$\begin{gathered}
a : \mathsf{nil}_\beta(x) \vdash x : \alpha \\
a : c : \mathsf{nil}_\gamma(x) + b : c : d : \mathsf{nil}_\alpha(x) \vdash x : \alpha \\
\mathsf{fix}_x(X = a : c : d : X) \vdash x : \alpha
\end{gathered}$$

but they all satisfy the original safety specification. This example sets the pattern for uses of the calculus in situations where non-trivial safety specifications are required.

6.6 Extensions

The typed process calculus defined in this chapter is purely synchronous, and does not incorporate the delay operations of SCCS or *SProc*. It would be useful to extend the calculus by adding δ and Δ so that asynchronous processes can be constructed. This can be done, although the theory of such an extended calculus has not been worked out as thoroughly as that of the basic calculus. This section describes, in outline, the necessary additions to the theory.

The first step is to add δ and Δ as type constructors. In order to make use of the new types, corresponding term formation rules are needed. The minimum requirement is a pair of functorial rules, such as:

Delay Functors

$$\frac{P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n}{\delta(P) \vdash x_1 : \delta\,\alpha_1, \ldots, x_n : \delta\,\alpha_n} \qquad \frac{P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n}{\Delta(P) \vdash x_1 : \Delta\,\alpha_1, \ldots, x_n : \Delta\,\alpha_n}$$

The rule for Δ allows a synchronous process to be desynchronised by inserting delays throughout its behaviour. The rule for δ is the foundation for an asynchronous prefixing construction. As in SCCS, the asynchronous prefix $a.P$ can be constructed as $a : \delta\, P$ if suitable prefix judgements can be derived. The way of building asynchronous prefixes is to start with a process $P \vdash \Delta\, \beta$ which, having type $\Delta\, \beta$, can delay at any point after the first action. The process $\delta(P) \vdash \delta\Delta\, \beta$ can also delay initially, and then $a : \delta(P) \vdash \Delta\, \alpha$ is a process which might also be written $a.P \vdash \Delta\, \alpha$. For this construction to be possible, a prefix judgement $\mathsf{Prefix}\ a : \Delta\, \alpha \to \delta\Delta\, \beta$ is needed. This can be derived by means of a new rule.

$$\frac{\mathsf{Prefix}\ \pi : \alpha \to \beta}{\mathsf{Prefix}\ \pi : \Delta\, \alpha \to \delta\Delta\, \beta}$$

It may also turn out that term formation rules corresponding to the monad structure of the delay operators are useful.

The operational semantics of delayed terms is defined by transition rules corresponding to the definitions of δ and Δ in $\mathcal{SP}roc$.

The main modification of the categorical semantics is to enable the interpretation of recursive definitions in which asynchronous prefixes are used. At present, the semantics of a recursive definition uses the UFPP of a functor in which occurrences of $\circ$ match the pattern of prefixes in the fix expression. When asynchronous prefixes are used, the pattern of delays corresponding to a sequence of prefixes is expressed by iterates of $(\circ\, \delta)$. This means that to interpret the extended calculus, the functor $!(\oplus_{n>0}(\circ\, \delta)^n)$ must have the multiple UFPP.

The Subject Reduction results still hold for the extended calculus, and when it is interpreted in $\mathcal{SP}roc$ there is the same agreement between the operational and categorical semantics.

The full development of this extension of the calculus is an area for future work.

6.7 Categorical Logic

As described in Chapter 1, a significant aspect of the Propositions as Types paradigm for the λ-calculus is the close connection between syntax and semantics as formalised by the construction of syntactic categories and the proof of various correspondence theorems [LS86, Cro94]. Some progress has been made towards a similar connection for interaction categories. The idea is to present a *process theory* as a process signature together with a collection of *axioms*, which are expressions of the form $P = Q \vdash \Gamma$ with $P \vdash \Gamma$ and $Q \vdash \Gamma$ proved processes. There is then a collection of rules for deriving more equations, which are the *theorems* of the theory. The process calculus of this

chapter could be presented in this style, in which case the rules for deriving instances of bisimulation would become rules for generating theorems. There is a notion of a model of a process theory in a suitable category (some form of interaction category), and in such a model provably equal processes have equal interpretations.

Once a process theory has been set up, it is possible to construct a category in which an object is a type of the theory, and a morphism from α to β is an equivalence class of proved processes $P \vdash x : \alpha^{\perp}, y : \beta$ under provable equality. There is a canonical model of the process theory in this category, and this model satisfies a universal property: a model of the theory in any other category factors as the canonical model followed by an interaction category functor. Conversely, given any interaction category a process theory can be extracted from it, and the operations of moving from a theory to a category and vice versa are inverse to each other. Work in this area is not sufficiently advanced to be reported fully in this thesis, but so far, the theory has been worked out for a simplified version of the typed process calculus.

This simplified calculus has no prefixing construction, but the effect of prefixing can be recovered by means of a functorial rule for $\circ$ and the provision of *process symbols*, which can be interpreted by arbitrary morphisms. Effectively, this means lifting the semantic interpretation of prefixing described in this chapter to the syntactic level. The corresponding categorical structure is that of a $*$-autonomous category with a monoidal endofunctor $\circ$ which has the UFPP; no commitment is made to synchrony or asynchrony. The development of the categorical logic of this simplified situation is described in [CGN94]; its extension to cover more general categorical structure and a more realistic process calculus is a topic for future work.

6.8 Deadlock-Freedom

Although the typed process calculus defined in this chapter can be given a semantics in any category with suitable structure, there is at present only one concrete category which has this structure, namely $\mathcal{SP}roc$. The original intention was that the category $\mathcal{SP}roc_D$ defined in Chapter 5 should also be able to interpret the calculus; this would give a syntax for deadlock-free processes. However, the present mechanism for defining and using prefixes is not subtle enough for this to be possible. In the ready specifications presentation of the deadlock-free category, suppose that X is an object of $\mathcal{SP}roc$ and $\theta \in \mathsf{RS}(X)$, so that (X, θ) is an object of $\mathcal{SP}roc_{D'}$, and that there is some syntactic type α which is interpreted by (X, θ) in a model. If the prefix actions a and b are available in α, then the Prefixing rule allows $a : P \vdash x : \alpha$ and $b : P \vdash x : \alpha$ to be formed. If these terms are to be interpreted as processes of type (X, θ), θ must contain the ready pairs $(\varepsilon, \{a\})$ and $(\varepsilon, \{b\})$. This means that $\theta^{\perp}$ does not contain $(\varepsilon, \{a\})$ or

$(\varepsilon, \{b\})$, but just $(\varepsilon, \{a, b\})$. So it should not be possible to form either $a : Q \vdash x : \alpha^{\perp}$ or $b : Q \vdash x : \alpha^{\perp}$, although $a : Q + b : R \vdash x : \alpha^{\perp}$ should be acceptable.

This example shows that if a process calculus is to be interpreted in $\mathcal{SP}roc_D$, the same prefixes cannot be made freely available in both a type and its negation. This has implications for both the prefix derivation rules and the term formation rule of Prefixing. It seems likely that the syntax and constructions of such a calculus would have to take ready specifications into account; this may not be too surprising, as the existing calculus has already been strongly influenced by the presence of safety specifications. One possibility might be a combined Prefixing and Summation rule, allowing the formation of sums of prefixed processes in which the actions offered are compatible with the ready specification of the desired type.

It it perhaps rather unfortunate that the syntax of the calculus should be affected so much by the intended semantic category. More work is needed to ascertain whether this is inevitable, or whether there can be situations in which the same syntax is suitable for a variety of semantic interpretations.

6.9 Discussion

All of the themes of the thesis have been drawn together in this chapter, making it the culmination of the present exposition of the theory and application of interaction categories. The typed process calculus extends the original ideas of the Proofs as Processes interpretation, by extending the syntax to one in which prefixing and dynamic behaviour can be defined. The formulation of a syntax for processes, with a type system based on linear logic and a categorical semantics, represents a successful transfer of the Curry-Howard isomorphism to concurrency. In this sense it is truly the high point of the thesis.

Having said that, there is still plenty of scope for further developments in the area of typed process calculi based on interaction categories. The syntax of the calculus has already been through several stages of refinement, and there may be more to come: designing a syntax which is both easily usable and sufficiently powerful is a difficult task. One obvious comment is that the present syntax may still contain too many traces of linear logic. The linear type constructors are acceptable—after all, a type system for concurrency is bound to introduce some new connectives—but the use of $\otimes$ and $\parr$ as syntactic operations on processes is a big step from the traditional notation of existing process algebras. The syntax has been designed to be easily adaptable to situations in which $\otimes$ and $\parr$ are distinct—this is the reason for using both of the multiplicative connectives even though the presence of the Mix and Cycle rules renders

them equivalent. A possible simplification would be to collapse $\otimes$ and $\parr$ in the syntax, thus specialising the calculus to the compact closed case. Mix and Cycle would then become derived rules. This should lead to a streamlined calculus with the minimal amount of syntax necessary for describing $\mathcal{SP}roc$ processes.

There are several reasons for the decision to study a synchronous calculus. One is that, as seen in Chapter 3, synchronous interaction categories have more structure and are easier to define and work with. Furthermore, there is not yet a definition of the exponentials in $\mathcal{ASP}roc$, which means that an asynchronous calculus would have no models unless the form of recursive definitions were restricted to make the use of ! unnecessary. From the syntactic point of view, an asynchronous calculus is likely to be slightly more complex, as more prefix combination rules would be needed. Nevertheless, the experience gained from the study of the synchronous calculus means that it should soon be possible to formulate an asynchronous version. Meanwhile, an extension of the synchronous calculus by delay operators has been outlined, and this should allow asynchronous processes to be constructed within the synchronous framework.

The area of categorical logic for process calculi has been briefly introduced, and should certainly be developed further. It already shows promise, and a categorical logic correspondence for a full-scale typed process calculus would round off the concurrent Curry-Howard Isomorphism in a very satisfying way. Another issue which has been raised but has yet to be resolved is that of adapting the calculus in such a way that it can be interpreted in $\mathcal{SP}roc_D$ and used for the construction of deadlock-free processes.

7

Conclusions

7.1 Summary

The starting point of interaction categories as a subject was Abramsky's discovery of $\mathcal{SP}roc$, a category which could be used to transfer to concurrency the familiar ideas of the categorical semantics of functional programming languages: types as objects, and programs as morphisms. $\mathcal{SP}roc$ and the related category $\mathcal{ASP}roc$ then provided a base from which to explore the application of existing type-theoretic ideas to concurrency, and also stimulated some new ideas concerning the rôle of types as specifications. This thesis is the story of some of that exploration.

Once the basic categories have been defined, in Chapter 3, their structure yields a type structure for processes and the categorical operations become rules for combining typed processes. This natural match between type structure and semantics is one of the benefits of the interaction categories approach. It is used to good effect in Chapter 4, where the compact closed structure of $\mathcal{SP}roc$ allows a semantics of dataflow computation to be defined at a high level of abstraction. Because the structure of interaction categories is so important for the development of the theory, an abstract axiomatisation is highly desirable. The possibilities for such an axiomatisation are discussed at the end of Chapter 3, and axioms based on the notion of guarded functor are proposed for both synchronous and asynchronous interaction categories.

The theme of Chapter 5 is the use of types to express complex behavioural properties of processes. The approach is to construct a category whose objects specify the desired properties, so that the typed process constructions corresponding to the categorical structure become compositional proof rules for those properties. This is illustrated by the construction of categories, both synchronous and asynchronous, in which the morphisms are deadlock-free processes. Compositional verification of deadlock-freedom of acyclic process configurations is supported by the categorical operations, and additional proof rules are formulated for cyclic constructions. The possibility of applying types to verification in this way is one of the main advantages of interaction categories over the other approaches to typed concurrency mentioned in Chapter 1.

In Chapter 6 all of these threads are brought together by the definition of a typed

calculus of synchronous processes. It has the linear type structure which has been used throughout the thesis, and the corresponding operations on typed processes become syntactic constructions. The calculus has a semantics in any synchronous interaction category, axiomatised along the lines suggested in Chapter 3. It also has an operational semantics in the usual process calculus style, which leads to a definition of bisimulation with respect to which the categorical semantics in $\mathcal{SP}roc$ is sound. Subject Reduction theorems describe how the type of a term can change as it makes transitions—this is a point of contact with Ferrari and Montanari's work [FM94], which starts from the assumption that transitions alter types. The syntax of the calculus is tailored to be able to deal with safety specifications, but there is also the prospect of developing modifications of the syntax allowing other properties such as deadlock-freedom to be discussed.

The introduction of the typed process calculus achieves the goal, set out in Chapter 1, of transferring the Curry-Howard isomorphism to concurrency. The fundamental design principle of the calculus is that the syntax should be derived from the natural structure of the semantic category, which means that there is a much better match between syntax and semantics than could be hoped for by adding types to an existing calculus. This in turn opens up the possibility of constructing initial models of the calculus as syntactic categories, and establishing the kind of categorical logic correspondences which already exist for various typed λ-calculi [Cro94]. As outlined in Chapter 6, this possibility has already been partially realised.

7.2 Further Research

There are many possibilities for further investigation of the topics covered by this thesis. In some cases, more work is required to remedy slight shortcomings of the present theory. At the moment there is no definition of the exponentials in $\mathcal{ASP}roc$; as mentioned in Chapter 3, it may be possible to approach this problem by adapting techniques which have been used for categories of games [AJM94]. In Chapter 5, the absence of a tensor unit in $\mathcal{FP}roc_D$ is rather unsatisfactory, and it may be that further study could improve the situation.

There are also several possibilities for continuing to pursue directions which have already been studied. The analysis of LUSTRE in Chapter 4 could be extended to cover the clock consistency calculations; this would give a more complete semantics to the language, and should have connections with Jensen's work [Jen94]. Clock calculation in the language SIGNAL, which is more complex because clocks can be specified relative to each other, should also be tackled.

Two possibilities for extension of the theory of the typed process calculus of Chapter 6 have already been mentioned—a more detailed development of the categorical logic correspondences, and the formulation of a syntax for deadlock-free processes. Still another, more ambitious possibility is the development of a typed calculus of asynchronous processes. Such a calculus would have a semantics in $\mathcal{ASP}roc$ or, better still, in a new category based on observation congruence instead of weak bisimulation—but the existence of this category is still a matter of speculation.

One of the most intriguing areas for further research is the possible connection between interaction categories and Milner's work on action structures. There are two levels at which connections might exist—the semantic level of $\mathcal{SP}roc$ or $\mathcal{ASP}roc$, and the syntactic level of the typed process calculus. An understanding of the relationship between the interaction categories and action structures theories would be very satisfying, and this avenue should certainly be explored.

Bibliography

[Abr91] S. Abramsky. Proofs as processes. Unpublished Lecture Notes, 1991.

[Abr93a] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[Abr93b] S. Abramsky. Interaction Categories (Extended Abstract). In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, pages 57–70. Springer-Verlag Workshops in Computer Science, 1993.

[Abr94a] S. Abramsky. Interaction Categories and communicating sequential processes. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 1–15. Prentice Hall International, 1994.

[Abr94b] S. Abramsky. Interaction Categories I: Synchronous processes. Paper in preparation, 1994.

[Abr94c] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135:5–9, 1994.

[Acz88] P. Aczel. *Non-well-founded sets.* CSLI Lecture Notes 14. Center for the Study of Language and Information, 1988.

[AJ92] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 211–222. IEEE Computer Society Press, 1992.

[AJ94] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543 – 574, June 1994.

[AJM94] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). In M. Hagiya and J. C. Mitchel, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[AW85] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* Academic Press, New York, 1985.

[Bar79] M. Barr. $*$-*Autonomous Categories*, volume 752 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

[Bar91] M. Barr. $*$-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(2):159–178, July 1991.

[BG91] C. Brown and D. Gurr. Relations and non-commutative linear logic. Technical Report PB-372, DAIMI, Aarhus University, November 1991. To appear in the Journal of Pure and Applied Algebra.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[BS94] G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, December 1994. To appear.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 1990.

[CGN94] R. L. Crole, S. J. Gay, and R. Nagarajan. An internal language for interaction categories. In C. L. Hankin, I. C. Mackie, and R. Nagarajan, editors, *Theory and Formal Methods 1994: Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods.*, 1994. To appear.

[Cro94] R. L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, 1993.

[FM94] G. Ferrari and U. Montanari. Typed additive concurrency. Submitted for publication., 1994.

[Gay93] S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.

[GGBM91] P. Guernic, T. Gautier, M. Borgne, and C. Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GL87] J.-Y. Girard and Y. Lafont. Linear Logic and lazy computation. In *CFLP 87: Conference on Functional and Logic programming*, volume 250 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GN93] S. J. Gay and R. Nagarajan. Modelling SIGNAL in Interaction Categories. In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*. Springer-Verlag Workshops in Computer Science, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HdP93] J. M. E. Hyland and V. C. V. de Paiva. Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64(3):273–291, 1993.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hon93] K. Honda. Types for dyadic interaction. In *CONCUR 93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[HV92] K. Honda and V. Vasconcelos. Principal typing scheme for polyadic π-calculus. Unpublished report., 1992.

[Jac94] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[Jen94] T. P. Jensen. A simple semantics of synchronous dataflow. In C. L. Hankin, I. C. Mackie, and R. Nagarajan, editors, *Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods*, 1994. To appear.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, 1974.

[KL80] G. M. Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[Laf88] Y. Lafont. The Linear Abstract Machine. *Theoretical Computer Science*, 59(1,2):157–180, 1988.

[Laf90] Y. Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, ACM Press, January 1990.

[Lam58] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced Mathematics Vol. 7. Cambridge University Press, 1986.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, Berlin, 1971.

[Mac94] I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.

[Man76] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics.* Springer-Verlag, 1976.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[Mil93a] R. Milner. Action structures. Technical Report 92-249, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1993.

[Mil93b] R. Milner. Action structures for the π-calculus. Technical Report 93-264, Department of Computer Science, University of Edinburgh, 1993.

[Mil93c] R. Milner. An action structure for the synchronous π-calculus. In *Proceedings of FCT'93*, volume 710 of *LNCS*, pages 87–105. Springer-Verlag, 1993.

[Mil93d] R. Milner. Action calculi and the π-calculus. In *Proceedings of the NATO Summer School on Logic and Computation.* Springer-Verlag, 1993.

[Mil94] R. Milner. Control structures II: Naming monoids. Draft, June 1994.

[MMP94] A. Mifsud, R. Milner, and J. Power. Control structures I. Draft, June 1994.

[MOM91] N. Martí-Oliet and J. Meseguer. From petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:69–101, 1991.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[PS93] B. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.

[See87] R. Seely. Linear logic, $*$-autonomous categories and cofree coalgebras. In *Contempory Mathematics*, 1987.

[Tur94] D. N. Turner. *Type and Polymorphism in the π-calculus*. PhD thesis, Department of Computer Science, University of Edinburgh, 1994. In preparation.

[Wad81] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[Yet90] D. N. Yetter. Quantales and (noncommutative) linear logic. *Journal of Symbolic Logic*, 55(1):41–64, March 1990.