Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School. NATO ASI Series F, Springer-Verlag, 1995.

Interaction Categories and the Foundations of Typed Concurrent Programming

Samson Abramsky, Simon Gay and Rajagopal Nagarajan

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London SW7 2BZ, UK.
email: {sa,sjg3,rn4}@doc.ic.ac.uk

Tutorial notes on Samson Abramsky's Interaction Categories, based on [3, 6] and incorporating additional material by Simon Gay and Raja Nagarajan in Sections 3.4, 5.2, 5.3, 5.4 and 6.3.

Abstract. We propose Interaction Categories as a new paradigm for the semantics of functional and concurrent computation. Interaction categories have *specifications* as objects, *processes* as morphisms, and *interaction* as composition. We introduce two key examples of interaction categories for concurrent computation and indicate how a general axiomatisation can be developed. The upshot of our approach is that traditional process calculus is reconstituted in functorial form, and integrated with type theory and functional programming.

A second contribution of the paper is to propose a way of combining the Propositions-as-Types and Verification paradigms based on the notion of Specification Structure. We describe specification structures for liveness properties and deadlock-freedom, and illustrate our ideas using a standard example from concurrency theory.

Contents

1 Introduction

We begin with a brief review of the achievements and limitations of existing paradigms for the semantics of computation. We then propose a new paradigm, *Interaction Categories*, with the aim of developing a unified theory encompassing both sequential and concurrent computation.

1.1 Semantic Paradigms

Denotational Semantics The most influential and longest established of current paradigms for the semantics of computation is denotational semantics. It is this paradigm which best approximates by far to the ideal of a mathematical theory of computation in the sense of McCarthy [45] or Scott [54].

The criticism we wish to lodge is one of scope. Despite its pretensions to universality, denotational semantics has an inherent bias towards a particular computational paradigm, that of *functional computation*. By this we mean, not only functional programming languages, but that whole sphere of computation in which the behaviour of the program is adequately abstracted as the computation of a function. This view of programs as functions is built into the basic mathematical framework on which denotational semantics is founded: a category of "sets" (domains) to interpret types, and certain functions between these sets to interpret programs.

Within the sphere of functional computation, denotational semantics has worked extremely well, serving not merely to describe, but to lead the way in language design and programming methods: think e.g. of types and type-checking, higher-order functions, recursive types, polymorphism, continuations, monads. These semantic insights have gone hand-in-hand with the canonical formal calculus for functional computation provided by the λ-calculus.

However, it is by now widely recognized that functional computation is just one, rather limited part of the computational universe, into which e.g. distributed systems, real-time systems and reactive systems don't really fit. The success of denotational semantics outside the sphere of functional computation has been much more limited.

Process Calculi A different family of semantic paradigms has been developed for reactive systems. The most notable of these is the process calculus paradigm, pioneered by Milner and Hoare, and exemplified by CCS [49] and CSP [32]. The great achievement of this paradigm over the past 15 years has been to develop an algebraic theory of concurrency, as a basis for structural methods of description of concurrent systems. The major limitation is that no canonical theory or calculus for concurrency has emerged; there is a veritable Babel of formalisms, combinators, equivalences. This may suggest that the current methodologies for concurrency are insufficiently constrained, or perhaps that some key ideas are still missing. Some secondary, but also significant limitations of the paradigm:

- It is type-free; a good notion of type for concurrent processes would be very desirable, but has proved elusive.
- There has been an over-emphasis on what processes *are*—as in the extensive literature on process equivalences—rather than on what structure they must possess collectively. One may compare this to the emphasis in the early days of set-theoretical foundations on which sets numbers *were*, as opposed to the modern emphasis on universal properties e.g. for Natural Numbers objects.
- Partly because of the lack of a good notion of a type, there has been a fairly systematic confusion between specifications and processes. One obvious place where this has caused problems has been in the discussion of *fairness*.

- The early design decisions of the pioneers have been copied so often that they have become almost invisible; yet there are quite basic points which may be questioned. For example, there is the rôle of *names* in process calculi; these are typically used as proper names (constants), which tends to give these calculi a highly syntactic, intensional slant from the outset.

The Great Bifurcation These two leading semantic paradigms have developed separately.[1] After 15 years this separate development must be regarded as a major open problem: how can we combine our understanding of the functional and concurrent process paradigms, with their associated mathematical underpinnings, in a single unified theory? Such a unification is required to obtain sound foundations for languages combining concurrent processes and communication on the one hand with types, higher-order constructs and polymorphism on the other. It is also needed as a basis for useful type systems for concurrency, which would allow interface constraints for concurrent modules (or objects) to be expressed. Such type systems are of prime importance. After all, it is only worthwhile to make some subsystem into a "black box" (a module or object) if we can describe its interface to its environment in some fashion significantly simpler than a detailed description of the internals of the black box.

Calculi vs. Semantic Universes At this point it will be useful to contrast the methodology implicit in presenting theories of concurrency as formal "process calculi", with that in which one presents a "semantic universe" in the form of a categorical model. The formal calculus approach starts from a set of combinators generating a syntax; then one may define a structured operational semantics or a model, various notions of equivalence, etc.

The weakness of this methodology is in the very first step; why this set of combinators rather than any other? If in fact a consensus had been reached that some calculus was canonical for concurrency in the same way and for the same kind of reasons that λ-calculus enjoys this status for functional computation, then this would not have been a problem. History has turned out otherwise, and should caution us to beware of availing ourselves too readily of the seductive freedoms of BNF.

In the categorical semantic approach, we define

- "objects" (types) A, B, C
- "morphisms" (programs) $f : A \to B$
- composition

$$\frac{f : A \to B \qquad g : B \to C}{f\,;g : A \to C}$$

Composition is the fundamental primitive of category theory, in the same sense that membership is the fundamental primitive of set theory. Once we have said what this bare framework of typed arrows closed under composition is, an enormous amount of further structure is then determined uniquely up to isomorphism. Thus the various type constructions of interest will typically be characterized by universal properties, which means: axiomatized in terms of their behaviour under composition in such a way that, if they exist at all, it can only be in (essentially) one way. For example, once we have specified a category $\mathbb{C}$ there is *only one way* it can be a cartesian closed category, and hence canonically model λ-calculus; the basic structure of arrows under composition determines all the possibilities for manipulation of higher-order functions by abstraction and application. To paraphrase Picasso: *we do not seek, we find.* Thus, in contrast to the formal calculus

[1]It should be noted that, while there is a clear contrast between Milner's operationally-based approach and the denotational paradigm, the comparison is more subtle in the case of CSP. Hoare's general approach is explicitly model-oriented [33], and CSP is based on a specific denotational model, namely the Failures model of Brookes, Hoare and Roscoe [15]. However, it seems clear that CSP has not in fact provided a basis for the kind of *rapprochement* we are after.

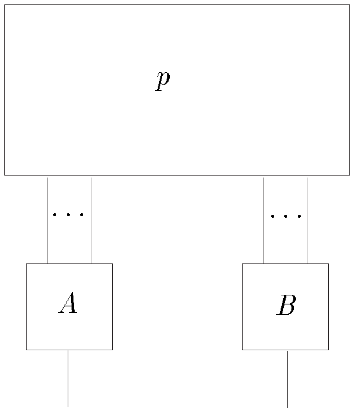


Figure 1: A process viewed as a morphism.

approach, there is a major shift from stipulation to observation of structure. Part and parcel of this is that the categorical framework imposes much more severe constraints on what counts as an acceptable definition: functoriality, naturality, universality, etc.

1.2 The Interaction Category Paradigm

We propose *Interaction Categories* as a new paradigm for the semantics of computation. In place of sets, functions, and function composition, an Interaction Category is a semantic universe where

- Types are *process specifications* A, B, C
- Morphisms are *processes* $p : A \to B$
- Composition is *interaction*

$$\frac{p : A \to B \qquad q : B \to C}{p \,;\, q : A \to C}$$

(Roughly speaking, interaction should be understood as "parallel composition + hiding/restriction" in process calculus terms).

At first sight, the reader will probably find it rather hard to accept a view of processes as arrows. Part of this is no doubt just the psychological association of the arrow notation with functions. A more substantial objection is that process interaction is symmetric and bidirectional in character. However, the reader surely had no difficulty in adjusting to the idea of a category of sets and relations; but relations have as much symmetry and no more intrinsic directionality than do communicating processes. Relations acquire a direction as arrows by convention; this meshes conveniently with the use of positional notation (ordered n-tuples) to access the components of the instances of a relation.

We take a similar view of a process *qua* morphism

$$p : A \to B.$$

We think of p as a concurrent module, with its interface to the environment represented as a bunch of wires, as in Figure 1. The types A, B *partition* the interface of p; moreover, each one may be highly structured, grouping various wires together. By accessing this interface positionally, we can get combinators that let us do process algebra *name-free*. Returning to our analogy with relations, we can contrast the name-free positional approach to the traditional use of names in process algebra as analogous to positional access via n-tuples $\langle a_1, \ldots, a_n \rangle$ vs. the labelled records representation

$$[l_1 \Rightarrow a_1 \,;\, \cdots ;\, l_n \Rightarrow a_n]$$

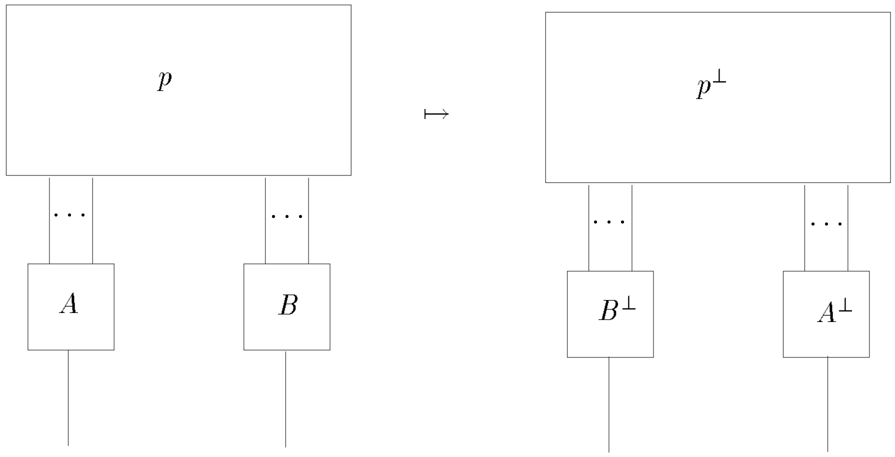


Figure 2: The duality on processes.

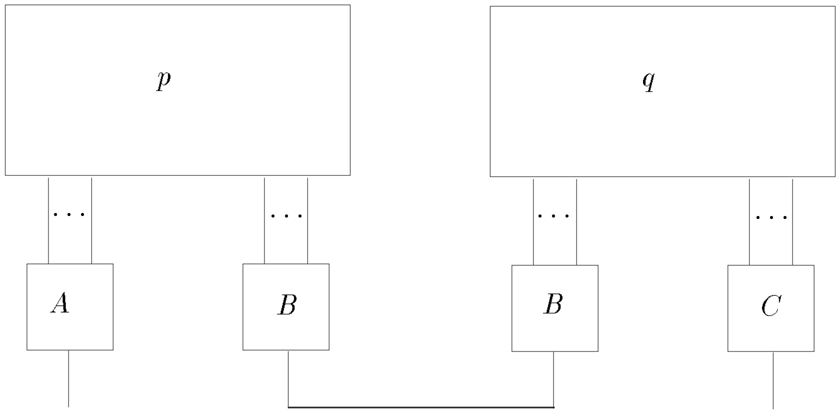


Figure 3: Composition of processes.

Again, we can draw an analogy with categorical combinators for cartesian-closed categories, which yield variable-free translations of the λ-calculus.

Because the partitioning of the interface is merely conventional, we can get a duality

$$\frac{A \xrightarrow{p} B}{A^{\perp} \xleftarrow{p^{\perp}} B^{\perp}}$$

such that $p^{\perp\perp} = p$. The action of this duality is illustrated in Figure 2. Often, it will be the case that $A^{\perp} = A$, i.e. the duality is trivial on objects; compare relational converse.

To connect processes together, we use categorical composition, as in Figure 3. The picture fits in with the previous comment about interaction (composition) as parallel composition + hiding: the "wire" between the B ports of p and q is the private connection formed by the hiding operation. Following Milner [49] and Hoare [32] all process calculi and algebras separate out the two operations of parallel composition and hiding or restriction. In [46], Milner remarks that perhaps it would be more natural to take the combined operation as the fundamental primitive; his main reason for not doing so is that this combined operation is not associative. By contrast our typed framework suffices to guarantee that our composition *is* associative.

We can use categorical type structure to control and repartition the interface. For grouping wires together, we can introduce a tensor product $A \otimes B$ with a corresponding

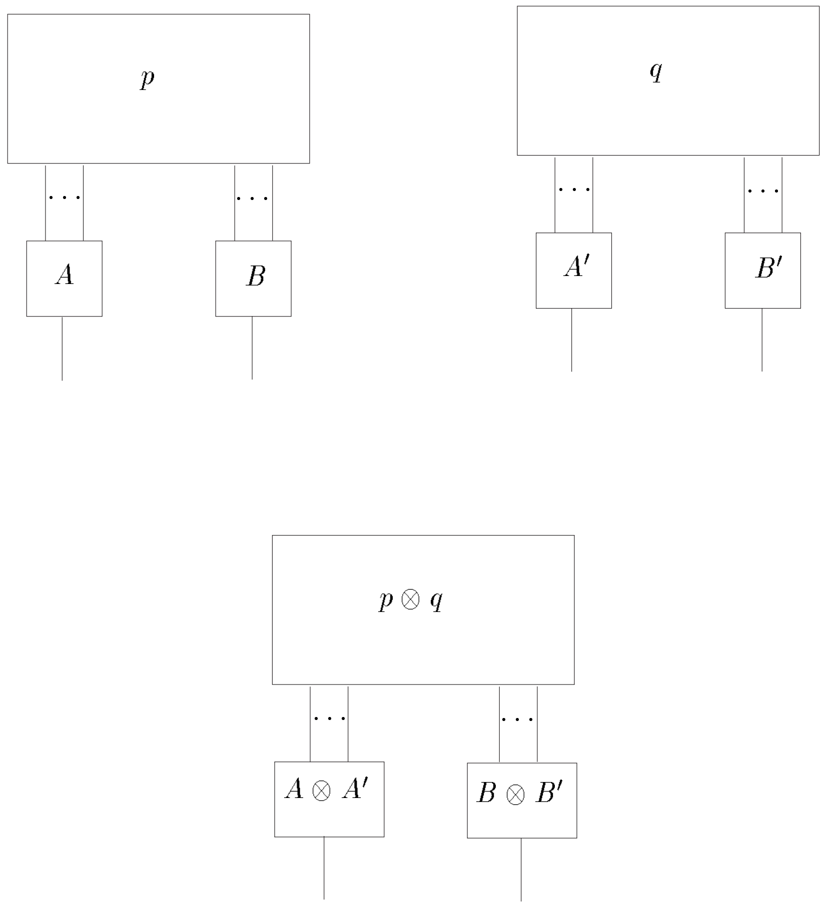


Figure 4: Combining processes by tensor product.

action on processes

$$\frac{p : A \to B \qquad q : A' \to B'}{p \otimes q : A \otimes A' \to B \otimes B'}$$

as illustrated in Figure 4. $p \otimes q$ is a disjoint parallel composition of p and q—there is no interaction between them. This tensor product should be *functorial*; while housekeeping on the grouping of wires is done positionally by canonical isomorphisms

$$\begin{array}{lllll} \mathsf{assoc}_{A,B,C} : & (A \otimes B) \otimes C & \cong & A \otimes (B \otimes C) \\ \mathsf{symm}_{A,B} : & A \otimes B & \cong & B \otimes A \\ \mathsf{unit}_A : & A \otimes I & \cong & A \end{array}$$

satisfying some standard coherence equations. All of this says that we should have the structure of a *symmetric monoidal category* [41].

Repartitioning of the interface is catered for by *currying*:

$$\frac{p : A \otimes B \to C}{\Lambda(p) : A \to (B \multimap C)}$$

which is illustrated in Figure 5, together with application

$$\mathsf{Ap}_{A,B} : (A \multimap B) \otimes A \to B.$$

This tensor structure gives us

Interpretation of the multiplicatives of Linear Logic [26]; "Linear λ-calculus" [1].	Static operations of process calculus [46] (in name-free form).

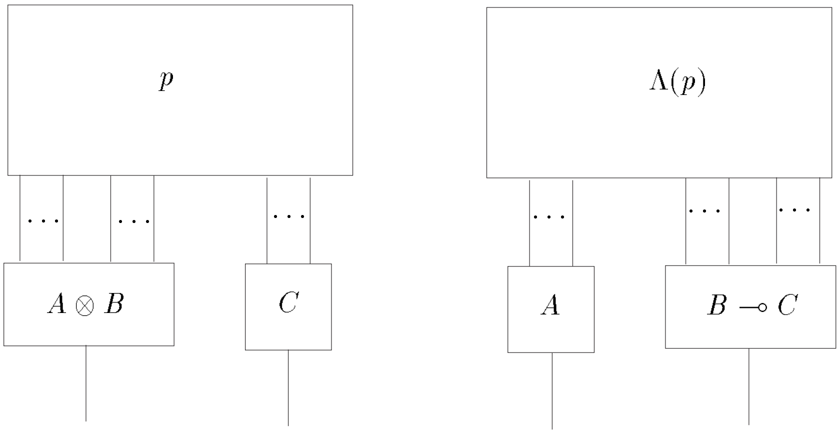


Figure 5: Currying processes.

If $A = A^\perp$ holds (or more generally if $A \otimes B = (A^\perp \otimes B^\perp)^\perp$, i.e. the category is *compact closed*) then we can use this type structure to build arbitrary process interconnection networks, including cycles. This will be explained in more detail in Section 2.4.

Continuing with our programme, we can define products and coproducts by universal properties, yielding

Additives of Linear Logic: conditionals, pairing and projections, injections and case expressions.	(disjointly) guarded sums of process algebra.

If $A = A^\perp$ holds (or more generally if products and coproducts coincide, so we have *biproducts*) this gives rise to an interpretation of non-determinism: if $p, q : A \to B$ then we can define their non-deterministic combination by

$$\begin{aligned} p + q &= A \stackrel{\Delta}{\to} A \oplus A \stackrel{[p,q]}{\to} B \\ &= A \stackrel{\langle p,q \rangle}{\to} B \oplus B \stackrel{\nabla}{\to} B \end{aligned}$$

(This is the standard derivation of a semi-additive structure from biproducts; see [41, 21]). The pairing of f and g will be a "disjointly guarded", and hence deterministic, combination of p and q; the type confusion between products and coproducts allows us to compose this with the codiagonal, which removes the tags rendering p and q disjoint, so that $p + q$ does indeed represent the non-deterministic mingling of p and q.

So we get a common core structure, supporting key elements of *both* functional programming *and* communication and concurrency. We can go further, to introduce the exponentials of Linear Logic, yielding the ability to define the full typed λ-calculus on the functional side, and replication on the process calculus side; polymorphism and recursive types; and constructs to articulate the temporal structure of processes, in the form of certain monads. In this fashion, we can obtain all the ingredients for combining the functional and communicating processes paradigms at a fundamental level, with substantial *sharing of structure*—the same formal structure being used to account for elements of both paradigms.

1.3 Concrete Examples of Interaction Categories

The ideas sketched in the previous sub-section should be understood as an informal outline of an axiomatic development of Interaction Categories. Two concrete examples of Interaction Categories for concurrent computation have been developed in some detail:

- **SProc**: an Interaction Category for synchronous processes. Some basic details of **SProc** are described in [2]; a much fuller account is given in [6]. Sections 2 and 3 of these notes present **SProc** at a level of detail somewhere in between. **SProc** has been applied to modelling a number of real-time synchronous languages including Esterel, Lustre and Signal [23, 7]; a typed calculus of synchronous processes based on the structure of Interaction Categories—with connections similar to those existing in the familiar correspondence between typed λ-calculus, intuitionistic logic, and cartesian closed categories—has also been developed [22, 24].

- **ASProc**: an Interaction Category for asynchronous processes. This will be described in detail in a forthcoming paper [4]. A simplified introduction appears in Section 5 of these notes; a similar description can be found in [3].

2 SProc: Linear Types

This section and the next describe the Interaction Category **SProc**, which is built from the following ingredients:

Objects	Concurrent System Specifications
Morphisms	Synchronisation trees
Composition	Synchronous product + restriction
Identities	Synchronous buffers ("wires" or "relays")

We will show **SProc** to have a very rich structure. Firstly, it provides a model for full Classical Linear Logic [26], and hence also, quite automatically, for typed λ-calculi [27]. It also supports a hierarchy of *delay monads* which express the temporal structure of the category. They allow asynchrony to be built on top of synchrony, as in Milner's original work on SCCS [48], but in a richer mathematical framework. The delay monads are also shown to satisfy distributive laws with respect to the exponentials, relating delay—extension in time—to replication—extension in space.

Much of the familiar process calculus material can be recovered from the structure of **SProc**. *Relabelling* and *restriction* can be described in terms of a subcategory of "embeddings"(for details see [6]); *non-determinism* arises from the semi-additive structure of **SProc**; *simulations* appear as 2-cells.

Our development of **SProc** clearly exhibits a view of processes as "relations extended in time". We also describe how the compact closed structure of **SProc** supports a "Multi-Cut" rule which allows general process networks to be described within a typed framework.

Before beginning the detailed development, a few preliminary comments are in order.

- The "specifications" considered in this section are quite rudimentary—just a sort and a set of traces (safety property). In Section 6, we will show how a tower of refinements (faithful functors)

  $$\mathbf{SProc} \hookleftarrow \mathbf{SProc}_1 \hookleftarrow \cdots \hookleftarrow \mathbf{SProc}_k$$

  can be set up, by incorporating progressively more refined specifications, covering behavioural properties such as deadlock-freedom and fairness. All these categories will share a common core structure; in particular, each of the above functors will preserve all the linear type structure.

- We will build on the established process calculus work, in particular on Milner's synchronous calculus SCCS. There are two reasons for choosing a synchronous product for the notion of interaction in **SProc**:

  - Our paradigm indicates taking buffers as the identity morphisms. This works well in both the synchronous case, where buffers impose no delay—they behave like hardware wires—and in the "buffered" or "receptive" process case [36], in which processes are insensitive to delay. Even in the asynchronous case, as we shall see in Section 5, *synchronous* buffers are the appropriate processes to take as identity morphisms.

- Synchronous calculi such as SCCS and MEIJE are known to be very expressive [19]; most other concurrent formalisms, including asynchronous calculi such as CCS and CSP, and real-time languages such as ESTEREL [14], can be interpreted as derived calculi in SCCS and MEIJE. So we lose no generality in taking synchronous interaction as the basic notion.

2.1 Preliminaries

We begin by reviewing some basic notions [49]. Let $\mathcal{L}$ be a set of labels. An *$\mathcal{L}$-labelled transition system* is a structure $(Q, \longrightarrow, q_0)$ where $q_0 \in Q$ is the initial state and $\longrightarrow \subseteq Q \times \mathcal{L} \times Q$ is the transition relation. We write $p \xrightarrow{a} q$ for $(p, a, q) \in \longrightarrow$; by abuse of notation, we refer to the whole transition system as Q. Given another such transition system $(Q', \longrightarrow', q'_0)$ a *strong simulation* from Q to Q' is a relation $R \subseteq Q \times Q'$ such that:

- $q_1 \xrightarrow{a} q_2, q_1 R q'_1 \Rightarrow \exists q'_2.[q'_1 \xrightarrow{a}{}' q'_2 \wedge q_2 R q'_2]$
- $q_0 R q'_0$.

R is a *strong bisimulation* if in addition its converse R^o is a strong simulation from Q' to Q.

We will take labelled transition systems modulo strong bisimulation as our notion of "process", hence of morphism in **SProc**. Since this is such a staple of concurrency theory, it allows immediate comparison with much of the literature. Also, strong bisimulation is the *finest* equivalence usually considered on transition systems. By showing that it suffices to yield all the equations we will need in order to establish the structure of **SProc**, we will automatically obtain the same result for any coarser equivalence. There is one important caveat to this remark. In this paper, we work within the interleaving paradigm, and ignore "true concurrency" issues [58].

In fact, instead of working explicitly with labelled transition systems quotiented by strong bisimulation, we will take advantage of Peter Aczel's work on non-well-founded sets [11], and work with *synchronisation trees* as canonical representations of strong bisimulation equivalence classes.

We define the synchronisation trees over $\mathcal{L}$ as the largest solution of the set equation

$$\mathsf{ST}_{\mathcal{L}} = \wp(\mathcal{L} \times \mathsf{ST}_{\mathcal{L}}).$$

More precisely , we take $\mathsf{ST}_{\mathcal{L}}$ as the final coalgebra of the functor

$$X \mapsto \wp(\mathcal{L} \times X).$$

The existence of this final coalgebra is guaranteed by Aczel's theory. Moreover, the final coalgebra property supports a method of definition by (non-well-founded, but "guarded") recursion, and a principle of coinduction. See [11] for further details.

Note that a synchronisation tree p yields a labelled transition system

$$(TC(p), \longrightarrow, p)$$

where

$$TC(p) = \{p\} \cup \bigcup\{TC(q) \mid \exists a.(a, q) \in p\}$$

$$p \xrightarrow{a} q \overset{\Delta}{\iff} (a, q) \in p.$$

Henceforth, we shall use the term "process" interchangeably with "synchronisation tree".

We will also make a conceptual point about synchronisation trees, in preparation for our subsequent development. The essential way in which computational processes generalise traditional mathematical objects such as functions and relations is that they have extension in time—they are *reactive systems* in Pnueli's apt terminology [43]. Moreover, they evolve over time; their behaviour in future interactions will depend on those which have already taken place. In short, they have state.

Thus we are led to generalise the notion of function

$$X \Longrightarrow Y$$

which maps an input to an output in a one-step interaction with its "environment", to a recursive specification of "functions extended in time":

$$F = X \Longrightarrow (Y \times F).$$

Such a function $f \in F$ maps an input x to an output y at the first step, and "evolves" into a continuation $f_1 \in F$:

$$f(x) = (y, f_1).$$

f_1 will depend on the particular, "observable" interaction $x \mapsto y$ which has already taken place. Thus F is the space of deterministic transducers with input alphabet X and output alphabet Y, described in an extensional fashion.

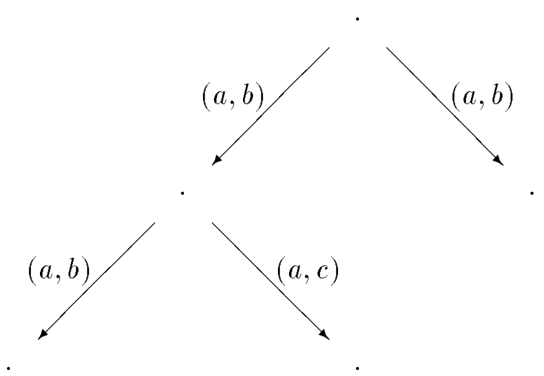
Similarly, we can generalise relations

$$\wp(X \times Y)$$

to "relations extended in time"

$$R = \wp((X \times Y) \times R),$$

and this will give a space of non-deterministic transducers. Note that these will be non-deterministic in two senses: firstly, outputs are not uniquely determined by inputs, and secondly continuations need not be uniquely determined by observable interactions, e.g.



Now note that, taking $\mathcal{L} = X \times Y$, R is precisely $\mathsf{ST}_{\mathcal{L}}$. Thus, if we introduce a cartesian product structure on actions, we can regard synchronisation trees as "relations extended in time". This simple idea will guide all our work in this section. It leads to a new way of seeing many familiar things, and brings some quite new structures to light.

Although our underlying computational paradigm of synchronous computation is similar to that of SCCS, we will not use that calculus itself. In particular, we will not assume an Abelian monoid structure on the "actions" (or labels, as we prefer to call them).

In fact, our treatment of labels represents an important conceptual break with the process calculus tradition. Instead of regarding labels as "proper names", which gives process calculus a strongly intensional slant from the outset, we will use our typed framework to take a more structural view. Our interpretations of various type constructions will involve simple set-theoretic constructions on the sets of labels—the "sorts"—associated with each type. We will use cartesian product of "sorts"—i.e. pairing of labels—to express the simultaneous performance of several actions distributed in space. Coproduct will be used to tag actions to allow choices to be made in a controlled fashion; while multisets of actions will be used to support replication of processes. In terms of the linear types, product, coproduct and multisets support the multiplicatives, additives and exponentials respectively. The labels in our framework play a similar rôle to *tokens* in information systems [55]; and we will develop a very expressive set of categorical combinators for process algebra which *never mention labels explicitly* at all; *cf.* the use of categorical combinators to give variable-free translations of functional programs [17].

It will turn out that all the processes and process operators we will need to construct will be definable from the following four combinators plus guarded recursion:

Prefixing:	$a{:}p = \{(a, p)\}$
Sum:	$\sum_{i \in I} p_i = \bigcup_{i \in I} p_i$
Synchronous product:	$p \times q = \{((a, b), p' \times q') \mid (a, p') \in p, (b, q') \in q\}$
Restricted relabelling:	$p[f] = \{(b, q[f]) \mid (a, q) \in p, fa \succcurlyeq b\}$

where $f : \mathcal{L} \rightharpoonup \mathcal{L}$ is a partial function on the ambient label set, and $fa \succcurlyeq b$ means "fa is defined and equal to b"(*cf.* [21]). In terms of labelled transitions, these combinators are specified as follows:

$$\frac{}{a : p \xrightarrow{a} p} \qquad \frac{p_i \xrightarrow{a} q}{\sum_{i \in I} p_i \xrightarrow{a} q}$$

$$\frac{p \xrightarrow{a} p' \qquad q \xrightarrow{b} q'}{p \times q \xrightarrow{(a,b)} p' \times q'} \qquad \frac{p \xrightarrow{a} q \qquad fa \succcurlyeq b}{p[f] \xrightarrow{b} q[f]}$$

Finally, some notation. Given a set X, we write X^* for the free monoid on X. We shall generally treat the canonical injection $X \rightarrowtail X^*$ as an inclusion. If $f : X \to Y$, we write $f^* : X^* \to Y^*$ for the unique homomorphism extending f. If $L \subseteq X^*$, then $L/a = \{s \in X^* \mid as \in L\}$.

2.2 The Category

We now define the category **SProc**.

Definition 2.2.1 *An object of* **SProc** *is a pair* $A = (\Sigma_A, S_A)$ *where* Σ_A *is an alphabet (sort) of actions (labels) and* $S_A \subseteq^{\text{nepref}} \Sigma_A^*$ *is a safety specification, i.e. a non-empty prefix-closed subset of* Σ_A^*.

If A is an object of **SProc**, 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 $\mathsf{alltraces}(P)$. The following coinductive definition is equivalent to this description.

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

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. In order to define the morphisms of **SProc**, we first define a $*$-autonomous structure on objects. This style of definition is typical of interaction categories; definitions of categories of games [8] follow the same pattern.

Definition 2.2.2 *Given A and B, the object $A \otimes B$ has*

$$\begin{array}{rcl} \Sigma_{A\otimes B} & = & \Sigma_A \times \Sigma_B \\ S_{A\otimes B} & = & \{\sigma \in \Sigma^*_{A\otimes B} \mid \mathsf{fst}^*(\sigma) \in S_A \wedge \mathsf{snd}^*(\sigma) \in S_B\}. \end{array}$$

The duality is trivial on objects: $A^\perp = A$.

A *-autonomous category in which $\otimes$ is self-dual, i.e. such that $(A \otimes B)^\perp \cong A^\perp \otimes B^\perp$, is called a *compact closed* category. Hence in a compact closed category, $A \mathbin{⅋} B \cong A \otimes B$. In the special case when $A^\perp \cong A$ the linear implication $\multimap$, defined by $A \multimap B = A^\perp \mathbin{⅋} B$, also coincides with $\otimes$:

$$A \multimap B \cong A \otimes B.$$

In **SProc**, 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: $A \mathbin{⅋} B = A \multimap B = A \otimes B$.

Not all interaction categories are compact closed, but as we shall see later, those that are support more process constructions than those that are not.

The definition of $\otimes$ makes clear the extent to which processes in **SProc** 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 at every one of its ports.

We can now define morphisms as processes of linear function-space type.

Definition 2.2.3 *A morphism $p : A \to B$ of* **SProc** *is a process p of type $A \multimap B$ (so p has to satisfy a certain safety specification).*

Since in **SProc** we have that $A \multimap B = A \otimes B$, 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 on terms of $\multimap$ is that it sets the pattern for all Interaction Category definitions, including cases in which there is less degeneracy.

Next, we turn to composition in **SProc**. Given $p : A \to B$, $q : B \to C$, we define $p; q : A \to C$ by:

$$p; q \quad \equiv \quad (p \times q)[((a, b), (b, c)) \mapsto (a, c)]. \tag{1}$$

Here and henceforth we specify a partial function on labels using pattern-matching notation. The intention is that the function is defined just on those arguments matching the pattern. In (1) the function is defined on those arguments

$$((a, b), (b', c)) \in (\Sigma_A \times \Sigma_B) \times (\Sigma_B \times \Sigma_C)$$

such that $b = b'$.

Composition can be specified in terms of 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. 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. Hence 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).

Definition 2.2.4 *If p is a process with sort Σ and $S \subseteq^{\text{nepref}} \Sigma^*$, then*

$$p\restriction S \quad = \quad \{(a, p'\restriction(S/a)) \mid (a,p') \in p, a \in S\}$$

where $S/a = \{\varepsilon\} \cup \{\sigma \mid a\sigma \in S\}$.

Definition 2.2.5 *The identity morphism $\mathsf{id}_A : A \to A$ is defined by $\mathsf{id}_A = \mathsf{id}\restriction S_{A \multimap A}$ where $\mathsf{id} = \{((a,a), \mathsf{id}) \mid a \in \Sigma_A\}$.*

The definitions made so far allow us to state

Proposition 2.2.6 **SProc** *is a category.*

Proof. Suppose $p : A \to B$, $q : B \to C$, $r : C \to D$. We wish to show that $(p;q);r = p;(q;r)$. Let $\mathcal{L} = \Sigma_A \times \Sigma_D$. Recall from 2.1 that $\mathsf{ST}_{\mathcal{L}}$ is the final T-coalgebra, where $TX = \wp(\mathcal{L} \times X)$. For convenience, we take $\mathsf{ST}_{\mathcal{L}} = T(\mathsf{ST}_{\mathcal{L}})$, with the structure map of the final coalgebra being the identity. Now consider the T-coalgebra

$$\alpha : \mathsf{ST}_3 \to T(\mathsf{ST}_3)$$

where

$$\begin{aligned} \mathsf{ST}_3 &= \mathsf{ST}_{\Sigma_A \times \Sigma_B} \times \mathsf{ST}_{\Sigma_B \times \Sigma_C} \times \mathsf{ST}_{\Sigma_C \times \Sigma_D} \\ \alpha(p,q,r) &= \{((a,d),(p',q',r')) \mid \exists b,c.\, p \xrightarrow{(a,b)} p', q \xrightarrow{(b,c)} q', r \xrightarrow{(c,d)} r'\}. \end{aligned}$$

We define maps $\beta_1, \beta_2 : \mathsf{ST}_3 \to \mathsf{ST}_{\mathcal{L}}$ by

$$\beta_1(p,q,r) = (p;q);r \quad \beta_2(p,q,r) = p;(q;r).$$

We claim that β_1 and β_2 are both T-coalgebra homomorphisms, i.e. that

$$(p;q);r = \{((a,d),(p';q');r') \mid \exists b,c.\, p \xrightarrow{(a,b)} p', q \xrightarrow{(b,c)} q', r \xrightarrow{(c,d)} r'\}$$

$$p;(q;r) = \{((a,d),p';(q';r') \mid \exists b,c.\, p \xrightarrow{(a,b)} p', q \xrightarrow{(b,c)} q', r \xrightarrow{(c,d)} r'\}.$$

These are immediate consequences of the definitions. But then, by the unicity part of the final coalgebra property, we must have $\beta_1 = \beta_2$, i.e. $(p;q);r = p;(q;r)$ for all p,q,r.

Similarly, to show $p;\mathsf{id}_B = p$ for $p : A \to B$, let $\mathcal{L} = \Sigma_A \times \Sigma_B$ and consider the T-coalgebra $\alpha : \mathsf{ST}_{\mathcal{L}} \to \mathsf{ST}_{\mathcal{L}}$ defined by

$$\alpha(p) = \{((a,b), p';\mathsf{id}_{B/b}) \mid p \xrightarrow{(a,b)} p'\}.$$

Define $\beta : \mathsf{ST}_{\mathcal{L}} \to \mathsf{ST}_{\mathcal{L}}$ by $\beta(p) = p;\mathsf{id}_B$. We claim that β is a T-coalgebra homomorphism, i.e. that we have the equation

$$p;\mathsf{id}_B = \{((a,b), p';\mathsf{id}_{B/b}) \mid p \xrightarrow{(a,b)} p'\}.$$

Again, this is immediate from the definitions. Applying the unicity part of the final coalgebra property, we conclude that $\beta = \mathsf{id}_{\mathsf{ST}_{\mathcal{L}}}$, i.e. that $p;\mathsf{id}_B = p$. ∎

2.3 SProc as a *-Autonomous Category

We can now complete the definitions of $\otimes$ and $^\perp$ as functors and describe the *-autonomous structure of **SProc**.

Definition 2.3.1 *If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ is defined by*

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

and $p^\perp : C \to A$ by

$$\frac{p \xrightarrow{(a,c)} p'}{p^\perp \xrightarrow{(c,a)} p'^\perp.}$$

Definition 2.3.2 *The tensor unit I is defined by*

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

Definition 2.3.3 *The canonical isomorphisms* $\mathsf{unitl} : I \otimes A \cong A$, $\mathsf{unitr} : A \otimes I \cong A$, $\mathsf{assoc} : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$ *and* $\mathsf{symm} : A \otimes B \to B \otimes A$ *are defined by*

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

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

$$\Lambda(p) = p[((a,b),c) \mapsto (a,(b,c))].$$

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

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

Note that all of the "logical" morphisms giving the symmetric monoidal closed structure are essentially formed from identities, and the only difference between p and $\Lambda(p)$ is a reshuffling of ports.

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.

We now have

Proposition 2.3.4 **SProc** *is a *-autonomous 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. ∎

As already noted, we also have

Proposition 2.3.5 **SProc** *is a compact closed category.*

2.4 Compact Closure and Multi-Cut

As we have already seen the linear type structure of **SProc** is quite degenerate, in that $A^\perp = A$, so that $\otimes = \invamp$ (**SProc** is compact-closed), $\& = \oplus$ and $! = ?$. In Section 6 we will see how to enrich the specifications of **SProc** to stronger behavioural properties. This will have the effect of "sharpening up" the Linear type structure so that the degeneracies disappear.

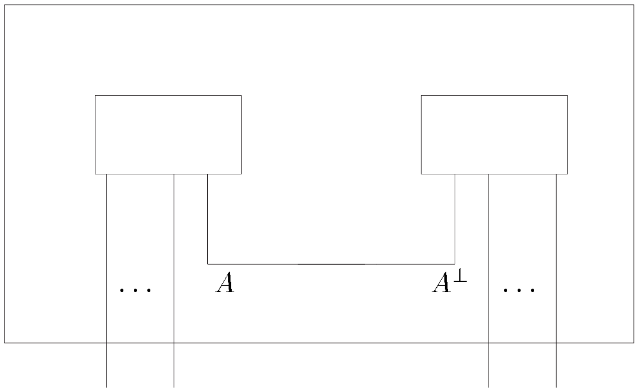
Our point here is that the looser type discipline of **SProc** can actually be *useful* in that it permits the flexible construction of a large class of processes within a typed framework.

In particular, compact closure validates a very useful typing rule which we call the *multi-cut*. (This is actually Gentzen's MIX rule [25] but we avoid the use of this term since Girard has used it for quite a different rule in the context of Linear Logic.)

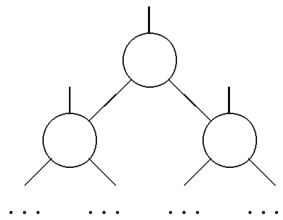
The usual Cut Rule

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

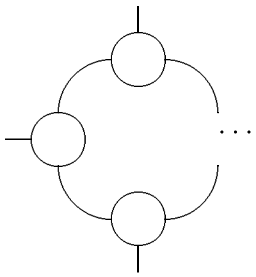
allows us to plug two modules together by an interface consisting of a single "port" [5]:



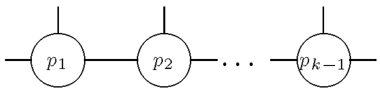
This allows us to connect processes in a tree structure



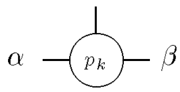
but not to construct cyclic interconnection networks



such as the Scheduler described in [49] and analysed in Section 3.4 of these notes. The problem with building a cycle is at the last step where we have already connected



To connect



we must plug both α and β simultaneously into the existing network. This could be done if we had the following "binary" version of the cut rule

$$\frac{\vdash \Gamma, A_1, A_2 \qquad \vdash \Delta, A_1^{\perp}, A_2^{\perp}}{\vdash \Gamma, \Delta}$$

or more generally the "multi-cut":

$$\frac{\vdash \Gamma, \Delta \qquad \vdash \Gamma', \Delta^{\perp}}{\vdash \Gamma, \Gamma'}$$

This rule is not admissible in Linear Logic and cannot in general be interpreted in Linear Categories. However it can always be canonically interpreted in a compact closed category (and hence in particular in **SProc**) as the following construction shows.

Let $\Gamma = A_1, \ldots A_m, \Gamma' = B_1, \ldots B_n, \Delta = C_1, \ldots C_k$. We write

$$\tilde{A} = A_1 \otimes \cdots \otimes A_m, \quad \tilde{B} = B_1 \otimes \cdots \otimes B_n, \quad \tilde{C} = C_1 \otimes \cdots \otimes C_k$$

$$\tilde{C}^{\perp} = (C_1 \otimes \cdots \otimes C_k)^{\perp} \cong C_1^{\perp} \otimes \cdots \otimes C_k^{\perp}$$

Suppose that the proofs of $\vdash \Gamma, \Delta$ and $\vdash \Gamma', \Delta'$ are interpreted by morphisms

$$f : I \longrightarrow \tilde{A} \otimes \tilde{C}, \quad g : I \longrightarrow \tilde{B} \otimes \tilde{C}^{\perp}$$

respectively. Then we can construct the required morphism $I \longrightarrow \tilde{A} \otimes \tilde{B}$ as follows:

$$\begin{tikzcd}
I \arrow[d, "\wr"', "\text{(unit)}"] \\
I \otimes I \arrow[d, "f \otimes g"] \\
(\tilde{A} \otimes \tilde{C}) \otimes (\tilde{B} \otimes \tilde{C}^{\perp}) \arrow[d, "\wr"', "\text{(canonical isos)}"] \\
\tilde{A} \otimes ((C_1 \otimes C_1^{\perp}) \otimes \cdots \otimes (C_k \otimes C_k^{\perp})) \otimes \tilde{B} \arrow[d, "\text{(evaluation)}"] \\
\tilde{A} \otimes I \otimes \cdots \otimes I \otimes \tilde{B} \arrow[d, "\wr"', "\text{(unit)}"] \\
\tilde{A} \otimes \tilde{B}
\end{tikzcd}$$

(Note that in a compact closed category $I = \perp$ so $A^{\perp} = A \multimap I$.)

In the case where $k = 1$ this construction is the internalization of composition in the category (using the autonomous structure) so it properly generalizes the standard interpretation of Cut.

2.5 Products and Coproducts

Being a *-autonomous category, **SProc** is a model of multiplicative linear logic. Interpreting the additives requires all binary products ($\&$) and coproducts ($\oplus$), and for the additive units, a terminal object ($\mathbf{1}$) and an initial object ($\mathbf{0}$). **SProc** does indeed have this structure. First we define binary coproducts.

Definition 2.5.1 *The functor $\oplus$ is defined on objects by*

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

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

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

We also define $\mathsf{inl} : A \to A \oplus B$, $\mathsf{inr} : B \to A \oplus B$ by

$$\begin{aligned}
\mathsf{inl} &= \mathsf{id}_A[(a, a) \mapsto (a, \mathtt{inl}(a))] \\
\mathsf{inr} &= \mathsf{id}_B[(b, b) \mapsto (\mathtt{inr}(b), b)]
\end{aligned}$$

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

$$\begin{aligned}
[p, q] &= p[(a, c) \mapsto (\mathtt{inl}(a), c)] \\
&\cup \ q[(b, c) \mapsto (\mathtt{inr}(b), c)].
\end{aligned}$$

Proposition 2.5.2 *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 $\mathtt{inl}(a)$ actions of inl can only match the $\mathtt{inl}(a)$ actions of $[p, q]$, and these come from p. Thus the result of the composition is the same as $\mathsf{id}_A\,; p$, ie 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 $(\mathtt{inl}(a), c)$ or $(\mathtt{inr}(b), c)$. In the first case, (a, c) must be a possible first action of p, because $\mathsf{inl}\,;h = p$. Similarly in the second case, h starts with a relabelled first action of q. Now, because of the safety specification of $A \oplus B$, which allows only an entire safe trace of A or one of B (with appropriate relabelling), a behaviour of h must continue in the same component of $A \oplus B$ in which it starts. And given that h is operating in (say) the A component, the fact that $\mathsf{inl}\,; h = p$ means that it must be behaving essentially as p. Hence $h = [p, q]$ as required. ∎

Note that the proof of uniqueness of the source tupling morphism relies crucially on the safety specification of $A \oplus B$. 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, we would only have a *weak* coproduct.

Since $\oplus$ is a coproduct, its dual is a product; because all objects of **SProc** are self-dual, this means that $A \oplus B$ is itself also a product of A and B—so, in fact, a biproduct. We will denote it by $\oplus$ rather than $\&$. There is also a zero object.

Definition 2.5.3 *The zero object $\mathbf{0}$ has $\Sigma_{\mathbf{0}} = \varnothing$ and $S_{\mathbf{0}} = \{\varepsilon\}$.*

Proposition 2.5.4 *The object $\mathbf{0}$ is initial and terminal in* **SProc**.

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 $\mathbf{0} \to A$. ∎

Proposition 2.5.5 **SProc** *has all biproducts.*

When referring to biproducts in **SProc**, we will always mean the specified biproducts defined in this section.

Since we have biproducts, we can use a standard construction [41, 21] to obtain a commutative monoid structure on each homset. This turns out to be the non-deterministic sum of CCS (modulo *strong* bisimulation).

Definition 2.5.6 *If $p, q : A \to B$ we define $p + q : A \to B$ by*

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

where $\Delta_A = \langle \mathsf{id}_A, \mathsf{id}_A \rangle$ is the diagonal and $\nabla_A = [\mathsf{id}_A, \mathsf{id}_A]$ the codiagonal. The unit is defined by $\mathbf{0}_A = A \to \mathbf{0} \to A$.

We have already seen 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 $\langle p, q\rangle \,;\, \nabla$. Pairing creates a union of the behaviours of p and q, but with disjointly labelled copies of B. Composing with ∇ removes the difference between the two copies. Hence in terms of the concrete set representations of processes, we have $p + q = p \cup q$. 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. Thus we obtain the natural representation in terms of synchronisation trees of the non-deterministic sum of CCS.

2.6 SProc as a Linear Category

To consider **SProc** as a model of full classical linear logic, we also need structure to interpret the exponentials $!$ and $?$. As mentioned earlier, the exponentials are necessary to define full typed λ-calculus on the functional side, and replication on the process calculus side. It is sufficient to define $!$ and its properties; $?$ is then given by De Morgan duality. Since **SProc** is self-dual, $?$ will be the same as $!$.

Clearly we want $!$ to be a functor. It should also have the following properties, in order to give interpretations of the exponential rules of classical linear logic.

- To interpret promotion, for each morphism $f : !A \to B$ there should be a morphism $f^\dagger : !A \to !B$.
- To interpret contraction there should be morphisms $\mathsf{contr}_A : !A \to !A \otimes !A$.
- To interpret weakening there should be morphisms $\mathsf{weak}_A : !A \to I$.
- To interpret dereliction there should be morphisms $\mathsf{der}_A : !A \to A$.
- There should also be isomorphisms $!\mathbf{1} \cong I$ and $!(A \& B) \cong !A \otimes !B$, where $\mathbf{1}$ is the terminal object.

These morphisms must satisfy various conditions, which can be organised into the statements that $!$ is a comonad, and there is a commutative comonoid structure on $!A$, together with the two isomorphisms. To spell this out, saying that $!$ is a comonad means that there are natural transformations $\varepsilon : ! \dot{\to} \mathsf{Id}$ and $\delta : ! \dot{\to} !!$ such that the diagrams

$$\begin{tikzcd} !!!A & !!A \arrow[l, "\delta_{!A}"'] \\ !!A \arrow[u, "!\delta_A"] & !A \arrow[l, "\delta_A"] \arrow[u, "\delta_A"'] \end{tikzcd} \qquad \begin{tikzcd} !A & !!A \arrow[l, "\varepsilon_{!A}"'] \arrow[r, "!\varepsilon_A"] & !A \\ & !A \arrow[ul, "\mathsf{id}"] \arrow[u, "\delta_A"] \arrow[ur, "\mathsf{id}"'] & \end{tikzcd}$$

commute. Saying that $!A$ has a commutative comonoid structure means that there are morphisms

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

such that the diagrams

$$\begin{tikzcd}
!A \otimes (!A \otimes !A) \arrow[rr, "\mathsf{assoc}"] & & (!A \otimes !A) \otimes !A \\
!A \otimes !A \arrow[u, "\mathsf{id} \otimes \mathsf{contr}_A"] & !A \arrow[l, "\mathsf{contr}_A"'] \arrow[r, "\mathsf{contr}_A"] & !A \otimes !A \arrow[u, "\mathsf{contr}_A \otimes \mathsf{id}"']
\end{tikzcd}$$

$$\begin{tikzcd}
I \otimes !A & !A \otimes !A \arrow[l, "\mathsf{weak}_A \otimes \mathsf{id}"'] \arrow[r, "\mathsf{id} \otimes \mathsf{weak}_A"] & !A \otimes I \\
& !A \arrow[ul, "\mathsf{unitl}^{-1}"] \arrow[u, "\mathsf{contr}_A"] \arrow[ur, "\mathsf{unitr}^{-1}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
!A \otimes !A \arrow[rr, "\mathsf{symm}"] & & !A \otimes !A \\
& !A \arrow[ul, "\mathsf{contr}_A"] \arrow[ur, "\mathsf{contr}_A"'] &
\end{tikzcd}$$

commute. This commutative comonoid structure directly defines weak_A and contr_A. The rest of the desired properties of ! come from the comonad structure. The morphism der_A is ε_A. Given $f : !A \to B$, $f^\dagger : !A \to !B$ is defined as $\delta_A ; !f$.

Taking this approach, the isomorphisms $!\mathbf{1} \cong I$ and $!(A \& B) \cong !A \otimes !B$ need to be established separately. An alternative is to impose the requirement that, if $\mathcal{C}$ is the category being considered as a model of Linear Logic, the forgetful functor from the category of commutative comonoids in $\mathcal{C}$ should have a right adjoint. This gives a construction of *cofree* commutative comonoids in $\mathcal{C}$; the comonad ! can be defined from the adjunction, and the extra isomorphisms hold automatically. This requirement is quite strong but is, in fact, satisfied by **SProc**. There is a general construction of cofree commutative comonoids using the definition

$$!A = \bigoplus_{n \in \omega} \otimes_s^n A$$

where $\otimes_s^n$ is the *symmetric tensor power*. This construction is carried out for **SProc** in [6], but here we will just concretely define $\otimes_s^n$ and the rest of the required structure, and directly check the relevant conditions. ! can be obtained from $\otimes_s^n$ as above.

Definition 2.6.1 *On objects, $\otimes_s^n$ is defined by*

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

where $\mathcal{M}_n(X)$ is the set of multisets of size n on X, and $\phi_n : \Sigma_A^n \to \mathcal{M}_n(\Sigma_A)$ is defined by $\phi_n(a_1, \ldots, a_n) = \{\!|a_1, \ldots, a_n|\!\}$, where $\phi_0(\cdot) = \{\!|\,|\!\}$. On morphisms, if $f : A \to B$ then

$$\otimes_s^n f = (\otimes^n f)[\phi_n].$$

The dereliction and weakening morphisms are defined by

$$\begin{aligned}\mathsf{der}_A &= \mathsf{id}_A[(a,a) \mapsto (\{\!|a|\!\}, a)] \\ \mathsf{weak}_A &= \mathsf{id}_I[(*,*) \mapsto (\{\!|\,|\!\}, *)].\end{aligned}$$

To define contraction, first define, for each m,n,r such that $m+n=r$, a morphism

$$\mathsf{contr}_A^{(r,m,n)} : \otimes_s^r A \to \otimes_s^m A \otimes \otimes_s^n A$$

by transition rules:

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

Then define $\mathsf{contr}_A^{(r)} : \otimes_s^r A \to \oplus_{m,n<\omega}(\otimes_s^m A \otimes \otimes_s^n A) \cong (\oplus_{m\in\omega} \otimes_s^m A) \otimes (\oplus_{n\in\omega} \otimes_s^n A) \cong\ !A \otimes\ !A$ *by*

$$\mathsf{contr}_A^{(r)} = \langle p_{r,m,n} \rangle_{m,n<\omega}$$

where

$$\begin{aligned} p_{r,m,n} &= \mathsf{contr}_A^{(r,m,n)} && \textit{if } m+n=r \\ &= \mathbf{nil} && \textit{otherwise.}\end{aligned}$$

Finally,

$$\mathsf{contr}_A = [\mathsf{contr}_A^{(r)}]_{r<\omega} : !A \to\ !A \otimes\ !A.$$

If $f : !A \to B$ then $f^\dagger : !A \to\ !B$ is defined by

$$f^\dagger = \langle f_n \rangle_{n<\omega}$$

where $f_n : !A \to \otimes_s^n B$ is

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

The rest of the comonad structure is then defined by $\delta_A = \mathsf{id}_{!A}^\dagger$.

Proposition 2.6.2 *For each A,*

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

is a commutative comonoid.

PROOF. To check that $\mathsf{contr}_A\,;\mathsf{symm} = \mathsf{contr}_A$, observe that if contr_A can split a multiset α into (β,γ), which is then transformed to (γ,β) by symm, then since (γ,β) is just another possible split of α it can also be generated by contr_A.

To check that $\mathsf{contr}_A\,;(\mathsf{weak}_A \otimes \mathsf{id}) = \mathsf{unitl}^{-1}$, observe that weak_A selects behaviours in $!A$ such that all the actions are $\{\!|\,|\!\}$. Thus $\mathsf{weak}_A \otimes \mathsf{id}$ selects behaviours in $!A \otimes\ !A$ which correspond to contr_A always splitting a multiset α into $(\{\!|\,|\!\}, \alpha)$. Finally, unitl^{-1} converts α to $(*,\alpha)$. The argument for the other unit law is symmetrical.

For associativity, it is easy to see that both $\mathsf{contr}_A;(\mathsf{id}\otimes\mathsf{contr}_A)$ and $\mathsf{contr}_A;(\mathsf{contr}_A\otimes\mathsf{id})$ generate all possible splits of a multiset into three partitions, with just a difference in bracketing, which can be removed by assoc. ∎

Proposition 2.6.3 $(\,!, \mathsf{der}, \delta)$ *is a comonad.*

PROOF. To see that $\delta_A\,;\,!\,\mathsf{der}_A = \mathsf{id}_{!A}$, note that

$$\begin{array}{rcl}\delta_A & = & \mathsf{id}_{!!A}[(\{\!|\{\!|a_{11},\ldots,a_{1n}|\!\},\ldots,\{\!|a_{m1},\ldots,a_{mn}|\!\}|\!\}, \\ & & \{\!|\{\!|a_{11},\ldots,a_{1n}|\!\},\ldots,\{\!|a_{m1},\ldots,a_{mn}|\!\}|\!\}) \\ & \mapsto & (\{\!|a_{11},\ldots,a_{mn}|\!\},\{\!|\{\!|a_{11},\ldots,a_{1n}|\!\},\ldots,\{\!|a_{m1},\ldots,a_{mn}|\!\}|\!\})]\end{array}$$

and

$$\begin{array}{rcl}!\,\mathsf{der}_A & = & \mathsf{id}_{!A}[(\{\!|a_1,\ldots,a_n|\!\},\{\!|a_1,\ldots,a_n|\!\}) \\ & \mapsto & (\{\!|\{\!|a_1|\!\},\ldots,\{\!|a_n|\!\}|\!\},\{\!|a_1,\ldots,a_n|\!\})].\end{array}$$

Thus in the composition $\delta_A\,;\,!\,\mathsf{der}_A$, the behaviours of δ_A are selected whose right actions are multisets of singleton multisets; so following the composition through $!\,!A$ just adds and removes a level of multiset, and the net result is the identity on $!A$.

Similarly, for $\delta_A\,;\mathsf{der}_{!A}$, the fact that

$$\begin{array}{rcl}\mathsf{der}_{!A} & = & \mathsf{id}_{!A}[(\{\!|a_1,\ldots,a_n|\!\},\{\!|a_1,\ldots,a_n|\!\}) \\ & \mapsto & (\{\!|\{\!|a_1,\ldots,a_n|\!\}|\!\},\{\!|a_1,\ldots,a_n|\!\})]\end{array}$$

means that the composition picks out the behaviours of δ_A whose right actions are singleton multisets.

Associativity of δ follows from the fact that if a multiset of multisets of multisets of actions is collapsed to a multiset of actions, it does not matter whether the collapsing is done inside-out or outside-in. ∎

3 SProc: Temporal Structure

Let us take stock of what has been achieved so far. We have interpreted Classical Linear Logic in **SProc**. General Linear Logic theory [26] ensures that we can obtain a categorical model for typed λ-calculus [27] from **SProc**. Concretely we get a translation from λ-terms into process terms built from our categorical combinators which in turn are defined from the four basic combinators. In contrast to the recent work on π-calculus [50], this translation and its correctness fall out of general theory, and we have not needed to introduce any apparatus of names and name-binding constructs.

The linear types also play an essential rôle in articulating the process-algebraic aspect of **SProc**. The canonical isomorphisms assoc, symm and unit witnessing the symmetric monoidal structure of **SProc** replace the Abelian monoid structure on actions in SCCS; they "functorialize" it. The closed structure of **SProc** is essential to allow "redistribution" of an interface between source and target. This ensures that when we set up an interaction:

$$A \xrightarrow{p} B \xrightarrow{q} C$$

we have complete control over what part of the action structure is put into B, the locus of interaction. The duality supports the symmetric, adirectional nature of concurrent interaction. The additives provide both disjointly labelled sums, and non-deterministic choice. The exponentials support replication which is essential for expressive power (*cf.* [50]).

However, the linear types by no means suffice to exploit the rich structure of **SProc**. In particular, they do not reflect the temporal structure of processes. To see this consider the following proposition.

Proposition 3.0.4 *The category* **Rel** *of sets and relations is bireflective in* **SProc**. *The reflection* $R : \mathbf{SProc} \longrightarrow \mathbf{Rel}$ *preserves all the linear category structure.* R *is defined by:*

$$\begin{array}{rcl}R(A) & = & \{a \mid a \in S_A\} \\ R(p) & = & \{(a,b) \mid \exists q.p \xrightarrow{(a,b)} q\}.\end{array}$$

This means that the linear connectives never take us out of the processes with trivial—one-step only—extension in time, i.e. relations. The moral of this observation is that while Interaction Categories should certainly model the linear types, we should also look for additional structure to reflect the fact that processes can have non-trivial extension in time. This structure should be orthogonal to the "spatial" or distributed structure expressed by the linear types.

This point can be related to the distinction made by Milner in the context of process calculi between "static" and "dynamic" operations [49]. The multiplicatives correspond to static operations (product, restriction); the additive to choice operators giving "one-step dynamics"; and the exponentials combine features of both. But the crucial operation of *prefixing* which alone yields extension in time is nowhere catered for by the linear types. This can be related to the results in [8] showing that the history-free strategies suffice to interpret the linear types.

Our approach to capturing this additional temporal structure in **SProc** is closely related to Milner's work on building asynchrony on top of synchrony in SCCS [48], but we will make full use of the Interaction Category framework to recast this work in a more conceptual fashion. This will lead us to some remarkable formal properties of these computational notions.

We will consider a hierarchy of three delay operators:

- Unit delay: $\bigcirc A$
- Unbounded delay before the first action: δA
- Propagation of unbounded delays *after* the first action: ΔA

The key role of δ and Δ as process combinators is clear from the SCCS and MEIJE literature [48, 13]. However we will recast these constructions in our framework as *monads*.

We will pay considerable attention to unit delay which is of fundamental significance in elucidating the structure of **SProc**. In SCCS terms it corresponds to prefixing with the idle action 1 [48]. Our notation is inspired by the "next time" operator of Temporal Logic [43].

It may seem surprising that we should consider just unit delay rather than general prefixing. However note that we can combine unit delay with the additives to express disjointly guarded sums

$$\sum_{i \in I} a_i : p_i \quad (a_i = a_j \Rightarrow i = j)$$

as:

$$\bigoplus_{i \in I} \bigcirc p_i.$$

We have already seen how biproducts give rise to general non-deterministic sums. Thus our categorical combinators suffice to construct general synchronization trees. Our framework allows us to do process algebra without ever naming an action.

3.1 Delay

3.1.1 Unit Delay

We define:

$$\Sigma_{\bigcirc A} = \mathbf{1} + \Sigma_A.$$

We will abuse notation by taking $\mathbf{1} = \{*\}$, assuming that $* \notin \Sigma_A$ and taking

$$\begin{aligned} \Sigma_{\bigcirc A} &= \{*\} \cup \Sigma_A \\ S_{\bigcirc A} &= \{\epsilon\} \cup \{*s \mid s \in S_A\}. \end{aligned}$$

Next we extend this to a functor on **SProc**. Given $p : A \longrightarrow B$ we define $\bigcirc p : \bigcirc A \longrightarrow \bigcirc B$ by

$$\bigcirc p \equiv (*, *) : p.$$

This functor is in fact a symmetric monoidal morphism [35]. The monoidal action is defined by:

$$m_{A,B} : \bigcirc A \otimes \bigcirc B \cong \bigcirc(A \otimes B)$$

$$u : I \cong \bigcirc I$$

$$m_{A,B} \equiv ((*, *), *) : \mathsf{id}_{A \otimes B} \quad {m_{A,B}}^{-1} = {m_{A,B}}^{\perp}$$

$$u \equiv (*, (*, *)) : \mathsf{id}_I \quad u^{-1} = u^{\perp}.$$

Proposition 3.1.1 *$\bigcirc$ is a symmetric monoidal endomorphism on* **SProc**.

The unit delay monad has further striking properties. Note firstly that the tensor unit is the unique fixpoint of $\bigcirc$. More precisely, there is an isomorphism:

$$\alpha : I \cong \bigcirc I$$

where $\alpha \equiv (*, *_1) : \mathsf{id}_I$ and $\alpha^{-1} = \alpha^{\perp}$. In fact we will show that this is the *free algebra* for the unit delay functor in the sense of Freyd [20].

Proposition 3.1.2 *α is the final $\bigcirc$-coalgebra and $\alpha^{\perp}$ the initial $\bigcirc$-algebra in* **SProc**, *i.e. (I, α) is the free $\bigcirc$-algebra in the sense of Freyd. Moreover, (I, α) is the unique $\bigcirc$-invariant object up to canonical isomorphism.*

This proposition can be obtained by an application of the following general notions. Let $T : \mathbb{C} \longrightarrow \mathbb{C}$ be an endofunctor. T has the *unique fixpoint property* if for all $f : A \longrightarrow TA, h : TB \longrightarrow B$ there is a unique $g : A \longrightarrow B$ such that

$$g = f \,; Tg \,; h.$$

Lemma 3.1.3 *If T is an endofunctor with the unique fixpoint property, then any T-invariant object is the free T-algebra; hence any two such are canonically isomorphic (so T has a unique fixpoint property on objects).*

PROOF. If $\alpha : TA \longrightarrow A$ is a T-invariant object then given $h : TB \longrightarrow B$ define $\mathsf{It}(h) : A \longrightarrow B$ as the unique fixpoint of $\mathsf{It}(h) = \alpha^{-1} \,; T(\mathsf{It}(h)) \,; h$.

By definition this will be the unique T-algebra homomorphism from (A, α) to (B, h). An exactly similar argument establishes that (A, α^{-1}) is the final T-coalgebra. ∎

Examples of endofunctors with the unique fixpoint property do arise in nature. For example if $\mathbb{C}$ is enriched over complete metric spaces and non-expansive maps then any locally contractive functor—i.e. for any A, B there is a λ with $0 \leq \lambda < 1$ such that $d_{TA,TB}(Tf, Tg) \leq \lambda \cdot d_{A,B}(f, g)$ for all $f, g : A \to B$—has the unique fixpoint property by an application of the Banach fixpoint theorem.

Functors on **SProc** whose action on morphisms is *guarded* [49] will have the unique fixpoint property. In particular, $\bigcirc$ is the basic example of a guarded functor.

Proposition 3.1.4 *$\bigcirc$ has the unique fixpoint property.*

PROOF. Let $f : A \to \bigcirc A$ and $h : \bigcirc B \to B$ be given. We attempt to define $g : A \to B$ from the fact that the diagram

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & \bigcirc A \arrow[d, "\bigcirc g"] \\ B & \bigcirc B \arrow[l, "h"] \end{tikzcd}$$

commutes. That is, we require the equation $g = f\,;\bigcirc g\,;h$ to hold. Since this is a guarded recursion in g, the equation has a unique solution in synchronisation trees [49]. ∎

To obtain Proposition 3.1.2, Lemma 3.1.3 can be applied to the isomorphism $\alpha : I \cong \bigcirc I$.

It is worth considering the morphism $\mathsf{It}(p)$ explicitly:

$$\begin{tikzcd} \bigcirc I \arrow[d, "\bigcirc\mathsf{It}(p)"'] & I \arrow[l, "\alpha"'] \arrow[d, "\mathsf{It}(p)"] \\ \bigcirc A \arrow[r, "p"'] & A \end{tikzcd}$$

$$\mathsf{It}(p) = \alpha; \bigcirc\mathsf{It}(p); p.$$

$\mathsf{It}(p)$ has a clear computational reading as *feedback* as suggested by the following "stream of consciousness" tables:

$$\begin{array}{ccc} p: & * & a_1 \\ & a_1 & a_2 \\ & a_2 & a_3 \\ & \vdots & \vdots \end{array} \qquad \begin{array}{ccc} \mathsf{It}(p): & * & a_1 \\ & * & a_2 \\ & * & a_3 \\ & \vdots & \vdots \end{array}$$

3.1.2 Unbounded Delay

We now consider the unbounded delay monad δ. δA is defined by

$$\begin{array}{rcl} \Sigma_{\delta A} & = & \mathbf{1} + \Sigma_A \\ S_{\delta A} & = & \{*^k \mid k \in \omega\} \cup \{*^k s \mid k \in \omega, s \in S_A\}. \end{array}$$

We will specify the monadic structure of δ using the streamlined presentation of [42]. Firstly, the unit

$$\eta_A : A \to \delta A$$

is defined—as a protomorphism—by

$$\eta_A \equiv \mathsf{id}_A$$

where we exploit the abuse of notation by which $S_A \subseteq S_{\delta A}$. Thus the unit forces the case of zero delay. The multiplication will "add" two nested delays, so that the monadic structure of δ will arise from the monoid $(\mathbb{N}, +, 0)$ seen as the underlying model of discrete future time.

The "Kleisli extension" [42]

$$\frac{A \xrightarrow{p} \delta B}{\delta A \xrightarrow{p^\dagger} \delta B}$$

is defined by the guarded recursion

$$p^{\dagger} \equiv p + \bigcirc(p^{\dagger}).$$

The picture is

$$\begin{array}{llllll} p: & a_1 & * & \qquad p^{\dagger}: & * & * \\ & \vdots & \vdots & & \vdots & \vdots \\ & a_k & * & & * & * \\ & a_{k+1} & b_1 & & a_1 & * \\ & \vdots & \vdots & & \vdots & \vdots \\ & & & & a_k & * \\ & & & & a_{k+1} & b_1 \\ & & & & \vdots & \vdots \end{array}$$

We can then define, for $p : A \to B$,

$$\delta p \equiv (p; \eta_B)^{\dagger},$$

and $\mu_A : \delta\delta A \to \delta A$ by

$$\mu_A \equiv \mathsf{id}^{\dagger}_{\delta A}.$$

Proposition 3.1.5 *(δ, η, μ) is a monad.*

We now show that δ has a universal characterization in terms of $\bigcirc$; this is of course the functorialization of the recursive definition of δ from unit delay in [49].

For each $A \in \mathsf{Ob}\,\mathbf{SProc}$ we define a functor $T_A : \mathbf{SProc} \to \mathbf{SProc}$ by

$$T_A B = A \oplus \bigcirc B.$$

Define $\beta : \bigcirc\delta A \to \delta A$ by

$$\beta \equiv (*_1, *) : \mathsf{id}_{\delta A},$$

and $\alpha : T_A(\delta A) \to \delta A$ by $\alpha \equiv [\eta_A, \beta]$.

Proposition 3.1.6 *$(\delta A, \alpha)$ is the free T_A-algebra.*

Proof. Firstly α is an isomorphism, with $\alpha^{-1} = \alpha^{\perp} = \langle \eta_A^{\perp}, \beta^{\perp} \rangle$. Next, we note that T_A is guarded, and hence has the unique fixpoint property, and apply Lemma 3.1.3.

3.1.3 Delay Propagation

We now turn to the delay propagation monad Δ. ΔA is defined by

$$\begin{aligned} \Sigma_{\Delta A} &= \mathbf{1} + \Sigma_A \\ S_{\Delta A} &= \{\epsilon\} \cup \{as \in \Sigma^*_{\Delta A} \mid a \in \Sigma_A, (as){\restriction}\Sigma_A \in S_A\}. \end{aligned}$$

The unit $\theta_A : A \to \Delta A$ is defined by $\theta_A \equiv \mathsf{id}_A$ just as for η_A. The Kleisli extension

$$\frac{A \xrightarrow{p} \Delta B}{\Delta A \xrightarrow{p^{\ddagger}} \Delta B}$$

is defined by

$$p^{\ddagger} \equiv \Sigma_{(a,q)\in p}\, a : (q^{\ddagger})^{\dagger}.$$

Note that we need an equation for each instance $p^{\ddagger}$; our basic combinators do not—apparently—suffice to define $(-)^{\ddagger}$ as an operator. This corresponds to the fact that Δ must be taken as primitive in SCCS [48]).

Now we define $\Delta p \equiv (p; \theta_B)^{\ddagger}$ for $p : A \to B$, and $\nu_A : \Delta\Delta A \to \Delta A$ by $\nu_A \equiv (\mathsf{id}_{\Delta A})^{\ddagger}$.

Proposition 3.1.7 *(Δ, θ, ν) is a monad.*

3.1.4 Distributive Laws

We now relate the delay monads to the exponential comonad. One motivation is conceptual: to relate delay—extension in time—to replication—extension in space. Also, having good interactions between the two structures is essential in order to keep the formalism manageable. Martin Hyland has recently proposed[2] the following structure as a way of combining computational notions: a monad (T, η, μ) and a comonad (Q, ε, δ) on a category $\mathbb{C}$, together with a *distributive law* [42]

$$\lambda : QT \longrightarrow TQ.$$

One can then define a "bi-Kleisli" category $K_{QT}(\mathbb{C})$ with morphisms $QA \rightarrow TB$ and composition of $f : QA \rightarrow TB$ and $g : QB \rightarrow TC$ given by

$$QA \xrightarrow{\delta_A} Q^2A \xrightarrow{Qf} QTB \xrightarrow{\lambda_B} TQB \xrightarrow{T_g} T^2C \xrightarrow{\mu_C} TC.$$

He also observed that the categories $\mathcal{GI}(\mathbb{C})$ constructed in [9] could be described in this way with the distributive law given by the least fixpoint combinator.

In the light of these ideas it is natural to look for distributive laws

$$\lambda : \,!\,\delta \longrightarrow \delta\,!$$

$$\Lambda : \,!\,\Delta \longrightarrow \Delta\,!$$

between the delay monads and the exponential comonad. We define

$$\begin{array}{rcl} \lambda_A & \equiv & \Sigma_{k\in\omega}(\delta \mathsf{id}_{\otimes_s^k A})[(*,*) \mapsto (\{\!\!|\underbrace{*,\ldots,*}_{k}|\!\!\},*)] \\ \Lambda_A & \equiv & \Sigma_{k\in\omega}(\Delta \mathsf{id}_{\otimes_s^k A})[(*,*) \mapsto (\{\!\!|\underbrace{*,\ldots,*}_{k}|\!\!\},*)] \end{array}$$

Proposition 3.1.8 *λ and Λ are distributive laws.*

3.2 Semi-additivity and Non-determinism

We say that a process $p \in \mathsf{ST}_{\mathcal{L}}$ is *deterministic* if, for all $s \in \mathcal{L}^*$, $p \xrightarrow{s} q$ and $p \xrightarrow{s} r$ implies that $q = r$. The processes arising from the constructions in **SProc** already considered need not be deterministic. For example, even if p and q are deterministic, $p; q$ need not be. In fact, non-determinism is explicitly accounted for by the categorical structure we have already described. It is standard that any category $\mathbb{C}$ with biproducts is *semi-additive* [41]; i.e. each hom-set $\mathbb{C}(A, B)$ has an Abelian monoid structure, and composition is bilinear. Explicitly, given $f, g : A \rightarrow B$, the addition is defined by

$$\begin{array}{rcl} f + g & = & A \xrightarrow{\Delta_A} A \oplus A \xrightarrow{[f,g]} B \\ & = & A \xrightarrow{\langle f,g\rangle} B \oplus B \xrightarrow{\nabla} B \end{array}$$

where $\Delta_A = \langle \mathsf{id}_A, \mathsf{id}_A\rangle$ is the diagonal and $\nabla_A = [\mathsf{id}_A, \mathsf{id}_A]$ is the codiagonal, with unit given by

$$0_A \;=\; A \longrightarrow \mathbf{1} = \mathbf{0} \longrightarrow A.$$

Moreover, any functor on $\mathbb{C}$ which preserves products or coproducts is automatically semi-additive.

[2] In a seminar.

As we saw in Section 2.5, **SProc** has all small biproducts. Applying the general definition, we see that $p+q$ is exactly the standard non-deterministic plus operation, with unit 0_A the standard "NIL" process. Thus we get the equations

$$
\begin{aligned}
p + 0 &= p \\
p + q &= q + p \\
(p + q) + r &= p + (q + r) \\
p; (\sum_{i \in I} q_i) &= \sum_{i \in I} (p \,;\, q_i) \\
(\sum_{i \in I} p_i); q &= \sum_{i \in I} (p_i \,;\, q).
\end{aligned}
$$

Moreover, since $A \otimes -$ and $- \otimes A$ are left adjoints, they preserve coproducts, so we get the distributivities

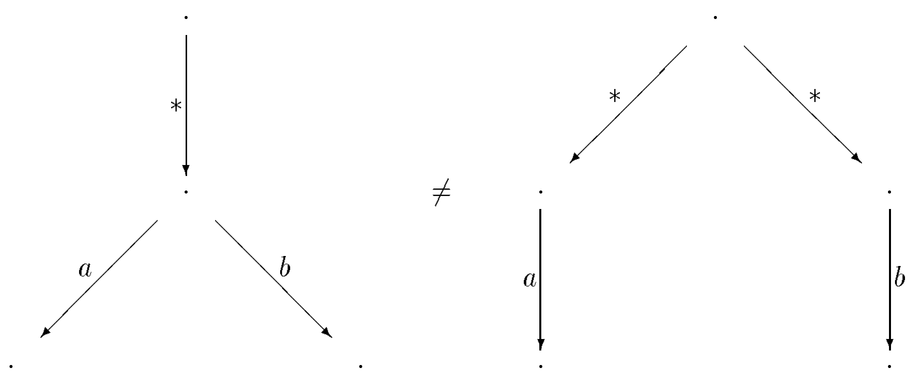
$$
\begin{aligned}
p \otimes \sum_{i \in I} q_i &= \sum_{i \in I} (p \otimes q_i) \\
\sum_{i \in I} p_i \otimes q &= \sum_{i \in I} (p_i \otimes q).
\end{aligned}
$$

Since the exponentials are constructed as cofree cocommutative comonoids, we get

$$
!(\sum_{i \in I} p_i) = \sum_{i \in I} !p_i.
$$

Thus we have recovered many familiar process algebra equations, and gained some new ones, from these very general considerations.

Turning to the delay operators, we observe firstly that unit delay is *not* semi-additive: we do not have $\bigcirc(p + q) = \bigcirc p + \bigcirc q$. This of course is the famous "non-equation" of branching-time semantics [46]:



For similar reasons, δ is not semi-additive: in $\delta p + \delta q$ the choice between p and q is made before the first delay step, while in $\delta(p + q)$ it is deferred until after the delay. However, Δ, which only introduces delays after the first action, preserves biproducts and hence is semi-additive.

3.3 Simulations as 2-cells

We have been working with processes as synchronization trees, i.e. canonical representations of equivalence classes of processes modulo strong bisimulation. But a plethora of other equivalences and pre-orders have been considered in concurrency theory [57] and a number have proved useful in verification and refinement. One approach to incorporating such notions in our framework would be to work with a notion of "process" based on some coarser equivalence; there would then be a different version of **SProc** for each such equivalence. We propose an alternative approach: if we fix some notion of "refinement" between processes, formalized as a preorder on synchronization trees[3], then we can incorporate this into **SProc** by taking the instances of the pre-order relation as 2-cells so that **SProc** becomes a 2-category [37]. Of course any specific notion of refinement must be checked to be sufficiently coherent with the structure of **SProc** that the 2-category axioms are satisfied; indeed these precisely express the key formal properties of a "good" notion of refinement.

We will illustrate this approach here with respect to *strong simulation* as a notion of refinement. We refer to Section 2.1 for the definition of strong simulation and for how a synchronization tree p can be regarded as a transition system $(TC(p), \longrightarrow, p)$.

We define $p \prec q$ ("p is strongly simulated by q") if there exists a strong simulation from p to q. It is standard from process calculus theory that this relation is a pre-order [49]. Note that the equivalence it induces:

$$p \asymp q \overset{\Delta}{\Longleftrightarrow} p \prec q \wedge q \prec p$$

is much coarser than bisimulation.

Each homset $\mathbf{SProc}(A, B)$ is a pre-order and hence a category under $\prec$. We must show that composition of 1-cells is functorial with respect to this structure. Given a diagram

$$\begin{tikzcd} A \arrow[r, shift left=1.5ex, "p"] \arrow[r, shift right=1.5ex, "p'"'] \arrow[r, phantom, "\curlywedge"] & B \arrow[r, shift left=1.5ex, "q"] \arrow[r, shift right=1.5ex, "q'"'] \arrow[r, phantom, "\curlywedge"] & C \end{tikzcd}$$

we must show that $p ; q \prec p' ; q'$. Again this is just a standard process calculus observation that simulation is a precongruence with respect to product and restriction.

We can similarly show that simulation is compatible with the Linear Category structure and the delay monads. Indeed this follows immediately from the fact that it is a precongruence with respect to our four basic process combinators plus guarded recursion (which is essentially standard process calculus theory [49]) since all of this structure was defined in terms of these. Thus we obtain:

Proposition 3.3.1 **SProc** *is a 2-category; the linear category and delay monad structure lifts to* **2-Cat***.*

We can apply the same arguments as above to any pre-order on transition systems which is compatible with strong bisimulation and a precongruence with respect to the four basic combinators and guarded recursion. For a large selection of such pre-orders including failures, acceptances, readies, and "barbed" versions of these, see [10].

3.4 An Analysis of a Cyclic Scheduler

As an illustration, we now use the ideas and techniques which have been presented so far to analyse a standard concurrency example—the cyclic scheduler of [49]. We begin by reviewing the specification and implementation of the scheduler, and then show how it can be constructed, with types, in **SProc**.

In later sections, we show how the scheduler can be typed in **ASProc** and then go on to consider safety properties, and show how the categorical framework supports their verification. Finally, as an example of the use of more refined types than that are available in either **SProc** or **ASProc**, we type the components of the scheduler in a category of deadlock-free processes, and use a type-checking argument to establish deadlock-freedom of the whole system.

The processes in our examples are defined by using an informal notation to construct synchronisation trees, which are then combined by means of the categorical operations.

[3] In fact one would in practice define the pre-order on transition systems. However following the idea that strong bisimulation is the finest reasonable equivalence, the pre-order will be well-defined on bisimulation equivalence classes and hence on synchronization trees.

Ideally we would like to use a more systematic notation—a typed process calculus, which is interpreted in the category being considered. Such a calculus (for synchronous processes) has been developed, and is presented in [22, 24].

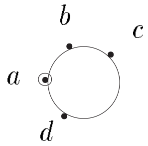
3.4.1 The Scheduler

We consider the construction of a cyclic scheduler as in [49]. The aim of the scheduler is to schedule n tasks or client processes co-ordinated by initiation and completion signals. The scheduler starts tasks off in a prescribed manner. The constraints are that each task is finished before it is started again and that the tasks are scheduled cyclically. This does not preclude, for example, starting a task before the previous one has finished. The specification as in [49] is:

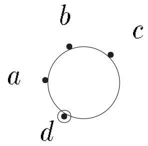
1. The clients are initiated in cyclic order using the actions $a_1, \ldots, a_n$, starting with a_1
2. For each i, the initiation a_i and the completion b_i must alternate.

The safety property we are concerned with primarily in this paper is the same as the correctness of the scheduler, i.e. the implementation of the scheduler meets the specification.

Following Milner, we implement the scheduler as a ring of n cells, each controlling one of n client processes (using a and b) and talking to its two neighbours in the ring (using c and d). Initially, a cell can be in one of two states, depending on whether it is going to communicate first with its client (this is the case for `cell`) or with a neighbour (`dcell`). The ringed ports are the ones that are ready to fire.



`cell`

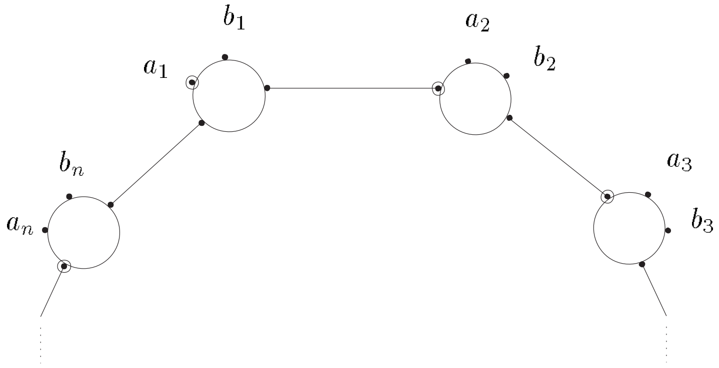


`dcell`

In CCS, the definitions are

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

The scheduler as a ring of n cells has the following arrangement.



`sched`

3.4.2 Types in SProc

The scheduler is inherently an asynchronous system: each cell has to be able to wait for signals from its neighbours and its client, and when a cell communicates it is only through one port at a time. Thus to construct and combine the cells in the synchronous category **SProc** we need to introduce asynchrony via the delay operators δ and Δ.

Each port of a cell only needs to have one action—the differently-named actions in the CCS definition refer to different ports rather than separate actions in the same port. The type X is defined by

$$\begin{array}{rcl} \Sigma_X & = & \{\bullet\} \\ S_X & = & \{\bullet^n \mid n < \omega\} \end{array}$$

which means, of course, that $X \cong I$, but we will continue to refer to it as X because we don't need to exploit any special properties of I. The type ΔX has an extra delay action $*$ which may appear anywhere except before the first $\bullet$; the type $\delta\Delta X$ has another delay action $*'$ which may appear before the first $\bullet$. We will not distinguish between $*$ and $*'$ (so, strictly speaking, we are using a type Y such that $Y \cong \delta\Delta X$).

A cell has four ports, two of which (a and c) are thought of as outputs and two (b and d) as inputs. The distinction between input and output is represented in the CCS definition by the appearance of a and c in complementary form: $\bar{a}$ and $\bar{c}$. In interaction categories this distinction is made at the level of the types of ports rather than at the level of actions. If a port of type X is considered to be an input, then the corresponding output has type $X^\perp$. In **SProc** there is no distinction between X and $X^\perp$, but in order to keep in mind the logical difference between input and output ports we will preserve the $^\perp$ notation. Thus a cell has two ports of type $\delta\Delta X$ and two of type $\delta\Delta X^\perp$. To obtain the overall type of the cell, the types of the ports are combined using $⅋$. In **SProc** this could just as well be $\otimes$, but again it is useful to stick to notation which reflects the logical distinctions between the connectives. The reason for using $⅋$ rather than $\otimes$ is the philosophy that $⅋$ relates to connected concurrency and $\otimes$ to disjoint concurrency, which suggests that $⅋$ is the appropriate connective for combining ports of a single process. This would give the typing

$$\texttt{cell} : \delta\Delta X ⅋ \delta\Delta X^\perp ⅋ \delta\Delta X ⅋ \delta\Delta X^\perp$$

or, for convenience, if we label the occurrences of X with the names of the corresponding ports in the original CCS definition,

$$\texttt{cell} : \delta\Delta X_d ⅋ \delta\Delta X_a^\perp ⅋ \delta\Delta X_b ⅋ \delta\Delta X_c^\perp.$$

However, this is not quite the type we want. The a and b ports are intended to be connected to a client process, which should also have a port of type $\delta\Delta X$ and one of type $\delta\Delta X^\perp$. Following the principle that the ports of a process should be combined by $⅋$, the type of a client should be $\delta\Delta X ⅋ \delta\Delta X^\perp$ but this is not of the right form to be connected to the $\delta\Delta X_a^\perp ⅋ \delta\Delta X_b$ appearing in the cell. In **SProc** this does not matter, because $\otimes$ and $⅋$ are the same, but we want to get the types absolutely right so that this example illustrates methods which are suitable for less degenerate categories. The solution is to use $\otimes$ to combine the $\delta\Delta X_a^\perp$ and $\delta\Delta X_b$ ports of the cell into a single port, resulting in the typing

$$\texttt{cell} : \delta\Delta X_d ⅋ (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) ⅋ \delta\Delta X_c^\perp.$$

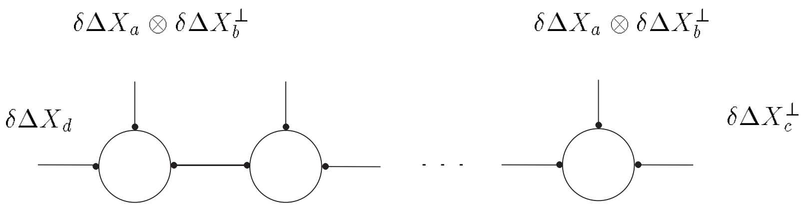
The type $\delta\Delta X_d ⅋ (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) ⅋ \delta\Delta X_c^\perp$ has a total of 16 actions available, made up of the various choices of $\bullet$ and $*$ in each port. But only four will be used in the definition of the cell—those corresponding to the original a, b, c and d. These actions are $(\bullet, *, *, *)$, $(*, \bullet, *, *)$, $(*, *, \bullet, *)$ and $(*, *, *, \bullet)$. For brevity, we will continue to use the names a, b etc.

To construct a process which behaves according to a CCS (asynchronous) definition from our basic ingredient of synchronous prefixing, we need to use the delay operators δ and Δ at appropriate points. For example, suppose the process P has type ΔA, which means that it can delay after its first action. Then δP has type $\delta\Delta A$ and can also delay initially. Attaching a synchronous prefix b, say, gives $b : P$ with type ΔA. This construction gives an asynchronous prefix in the sense that there may be delay after the prefixing action has been carried out, although the action still has to happen first with no initial delay. Finally, $\delta(b : P)$ has type $\delta\Delta A$ and is completely asynchronous. Using this idea, we can construct the scheduler cell in **SProc**. One additional fact is needed—there is an inclusion of $\delta(A \parr B)$ into $(\delta A) \parr (\delta B)$, and similarly for Δ.

Combining the cells into the scheduler can be separated into two steps: first combining n cells into a chain, and then connecting the two ends of the chain to form a cycle. The first step can be carried out in any *-autonomous category. Regarding a cell as a morphism $\texttt{cell} : I \to \delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp$, connecting the c port of one cell to the d port of another is carried out by the following categorical calculation.

$$\begin{tikzcd}
I \arrow[d, "\wr"', "\text{(unit)}"] \\
I \otimes I \arrow[d, "\texttt{cell} \otimes \texttt{cell}"] \\
\delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp \otimes \delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp \arrow[d, "\text{(canonical nat. trans.)}"] \\
\delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr (\delta\Delta X_c^\perp \otimes \delta\Delta X_d) \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp \arrow[d, "\text{(evaluation)}"] \\
\delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \perp \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp \arrow[d, "\wr"', "\text{(unit)}"] \\
\delta\Delta X_d \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \parr \delta\Delta X_c^\perp
\end{tikzcd}$$

After $n-1$ constructions of this form, the result is n cells arranged in a line:



Connecting the free ends together makes use of the coincidence of $\otimes$ and $\parr$ in **SProc** and the corresponding calculation is shown below.

$$
\begin{array}{c}
I \\
\Big\downarrow \; \texttt{ncell} \\
\delta\Delta X_d \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \delta\Delta X_c^\perp \\
\Big\downarrow \wr \; \text{(permutation)} \\
(\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \delta\Delta X_c^\perp \mathbin{⅋} \delta\Delta X_d \\
\Big\downarrow \wr \; \text{(iso in } \mathbf{SProc}\text{)} \\
(\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \delta\Delta X_c^\perp \otimes \delta\Delta X_d \\
\Big\downarrow \wr \; \text{(evaluation)} \\
(\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \perp \\
\Big\downarrow \wr \; \text{(unit)} \\
(\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b)
\end{array}
$$

The resulting process is the scheduler, with the typing

$$\mathsf{sched} : (\delta\Delta X_a^\perp \otimes \delta\Delta X_b) \mathbin{⅋} \cdots \mathbin{⅋} (\delta\Delta X_a^\perp \otimes \delta\Delta X_b).$$

The point is that we can connect client processes to the scheduler just on the basis of their type. Any process with the type $\delta\Delta X^\perp \mathbin{⅋} \delta\Delta X$ or, as $⅋ = \otimes$ in **SProc**, $\delta\Delta X^\perp \otimes \delta\Delta X$ can be hooked up to the scheduler. Since the types used in this exampe are rather weak (we have even ignored safety, to keep matters simple) this does not say very much—just the fact that client processes should have the right alphabet and interface structure. However, when we discuss **ASProc** we will introduce safety into our types and in the case of other categories which have more refined types than **SProc** or **ASProc**, we shall see that the types impose considerable constraints on processes that can be connected and hence contain a substantial amount of information.

4 SCCS in SProc

In this section, we show how Milner's synchronous calculus SCCS can be faithfully interpreted in **SProc**. The most interesting aspect of this interpretation is that it is couched in terms of the categorical structure of **SProc**. Any category with the same structure would admit a sound interpretation of SCCS (and hence of most process calculi), as well as of λ-calculus by virtue of its linear category structure. Thus we can regard this interpretation as providing useful data towards an axiomatisation of Interaction Categories.

Let (Act, m, e) be the action monoid of SCCS. We will use the type $A = (\mathsf{Act}, \mathsf{Act}^*)$ as a "universal type" over which all the terms of the SCCS calculus will be interpreted. This is somewhat analogous to the use of a reflexive object $A^A \lhd A$ to interpret the untyped λ-calculus in a cartesian closed category [39]. Note that there are isomorphisms

$$
\begin{array}{rcl}
A & \cong & \bigoplus_{a \in \mathsf{Act}} \bigcirc A_a \\
A_a & \cong & A.
\end{array}
\qquad (2)
$$

Closed terms P of SCCS will be interpreted by morphisms $[\![P]\!] : I \longrightarrow A$.

4.1 Summation

This is interpreted directly by the addition in **SProc** (*cf.* Section 2.5):

$$[\![\sum_{i \in I} P_i]\!] = \sum_{i \in I} [\![P_i]\!].$$

4.2 Prefixing

$$[\![a{:}P]\!] = I \cong \bigcirc I \xrightarrow{\bigcirc[\![P]\!]} \bigcirc A \xrightarrow{\mathsf{in}_a} \bigoplus_{a \in \mathsf{Act}} \bigcirc A_a \cong A$$

where we use the isomorphism (2).

4.3 Product and Restriction

Firstly, we define a morphism $\bar{m} : A \otimes A \longrightarrow A$ to represent the multiplication on the action monoid. Concretely, $\bar{m}$ is defined by $\bar{m} = \{(((a,b),c),\bar{m}) \mid m(a,b) = c\}$. It can also be defined more abstractly, in terms of categorical structure, as follows.

We begin by defining a functor T on **SProc** by

$$TX = \bigoplus_{a \in \mathsf{Act}} \bigcirc X.$$

Given $p : X \otimes Y \longrightarrow Z$, we wish to define

$$T_2(p) : TX \otimes TY \longrightarrow TZ.$$

We use the sequence of natural isomorphisms:

$$\begin{array}{c}
TX \otimes TY \longrightarrow TZ \\
\hline
\bigoplus_{a \in \mathsf{Act}} \bigcirc X \otimes \bigoplus_{b \in \mathsf{Act}} \bigcirc Y \longrightarrow \bigoplus_{c \in \mathsf{Act}} \bigcirc Z \\
\hline
\bigoplus_{a,b \in \mathsf{Act}} \bigcirc X \otimes \bigcirc Y \longrightarrow \bigoplus_{c \in \mathsf{Act}} \bigcirc Z \\
\hline
\bigoplus_{a,b \in \mathsf{Act}} \bigcirc (X \otimes Y) \longrightarrow \bigoplus_{c \in \mathsf{Act}} \bigcirc Z
\end{array}$$

Thus we can define

$$T_2(p) = (T_2(p)_{a,b,c})_{a,b,c \in \mathsf{Act}}$$

where $T_2(p)_{a,b,c} : \bigcirc(X \otimes Y) \longrightarrow \bigcirc Z$. We set

$$T_2(p)_{a,b,c} = \begin{cases} \bigcirc p & m(a,b) = c \\ 0 & \text{otherwise.} \end{cases}$$

(This definition forms part of the extension of T to a *multifunctor* on **SProc** viewed as a multicategory [38].) Since both T and T_2 are guarded, there is a unique invariant $\alpha : A \cong TA$, and a unique fixpoint $\bar{m} : A \otimes A \longrightarrow A$ such that

$$\begin{tikzcd}
A \otimes A \arrow[r, "\bar{m}"] \arrow[d, "\alpha \otimes \alpha"'] & A \arrow[d, "\alpha"] \\
TA \otimes TA \arrow[r, "T_2(\bar{m})"'] & TA
\end{tikzcd}$$

Now we can define
$$[\![P \times Q]\!] = I \cong I \otimes I \xrightarrow{[\![P]\!] \times [\![Q]\!]} A \otimes A \xrightarrow{\bar{m}} A.$$
Note that any "generalized static operation" [40] could be treated in the same way. In particular, restriction is interpreted by
$$[\![P \lceil X]\!] = I \xrightarrow{[\![P]\!]} A \xrightarrow{\rho_X} A$$
where ρ_X is defined by a guarded recursion in similar but simpler fashion to $\bar{m}$. The details are left as a worthwhile exercise for the reader.

4.4 Delays

Firstly, we define morphisms
$$\xi : \delta A \longrightarrow A, \quad \zeta : \Delta A \longrightarrow A$$
by
$$\begin{array}{rcl} \xi & \equiv & \mathsf{id}_{\delta A}[(*,*) \mapsto (*,e)] \\ \zeta & \equiv & \mathsf{id}_{\Delta A}[(*,*) \mapsto (*,e)]. \end{array}$$

Proposition 4.4.1 *(A, ξ) is an algebra for the monad (δ, η, μ). (A, ζ) is an algebra for the monad (Δ, θ, ν).*

Now we can define
$$\begin{array}{rcl} [\![\delta P]\!] & = & I \xrightarrow{t^{\perp}_{\delta I}} \delta I \xrightarrow{\delta [\![P]\!]} \delta A \xrightarrow{\xi} A \\ [\![\Delta P]\!] & = & I \xrightarrow{t^{\perp}_{\Delta I}} \Delta I \xrightarrow{\Delta [\![P]\!]} \Delta A \xrightarrow{\zeta} A. \end{array}$$
Here t_A is the morphism $A \rightarrow I$ defined by $t_A = \{((a,*), t_A) \mid a \in \Sigma_A\}$.

4.5 Guarded Recursion

By standard techniques [16], we can interpret any SCCS term $P(X)$ containing a process variable X as a morphism
$$[\![P]\!] : \,!A \longrightarrow A.$$
An interpretation of the recursive term $\mathsf{rec}\, X.\, P$ is provided by a morphism $p : I \longrightarrow A$ such that

$$\begin{tikzcd} (\,!A \multimap A) \otimes \,!A \arrow[r, "\mathsf{Ap}"] & A \\ I \otimes I \arrow[u, "\Lambda([\![P]\!]) \otimes (\mathsf{der}_I\,;\,!p)"] \arrow[r, "\mathsf{unit}"'] & I \arrow[u, "p"'] \end{tikzcd}$$

If X is guarded in P, then a unique such morphism exists in **SProc**. Multiple recursions can be handled similarly.

4.6 Correctness of the interpretation

Theorem 4.6.1 *For any guarded recursive term P in SCCS, let $\mathsf{ST}(P)$ be the corresponding synchronization tree as defined in [11]. Then*
$$\mathsf{ST}(P) = [\![P]\!][(*,a) \mapsto a].$$

5 Beyond SProc

Now we extend our treatment to asynchronous concurrency by introducing the interaction category **ASProc**. As we shall see, asynchrony introduces some complications which means that our structure is no longer as tight as it was for **SProc**. We also pursue the issue of finding suitable axioms to express the notion of an Interaction Category.

5.1 ASProc—An Interaction Category for Asynchronous Processes

We briefly give the basic definitions of **ASProc**, an Interaction Category for asynchronous processes. To simplify the discussion we shall model processes by sets of traces, as in [31] rather than by labelled transition systems modulo weak bisimulation, as in the "official" definition.

As in [31], a basic notion is that of a *trace* of a process. However, we differ from [31] in allowing traces to record simultaneous occurrences of events. Thus a trace on a given alphabet Σ is a finite sequence of non-empty subsets of Σ. We write $\mathtt{Tr}(\Sigma)$ for the set of traces on Σ. If $s \in \mathtt{Tr}(\Sigma)$ and $X \subseteq \Sigma$, we define $s{\restriction}X$ inductively:

$$\begin{array}{lcl} \varepsilon{\restriction}X & = & \varepsilon \\ (ms){\restriction}X & = & \begin{cases} (m \cap X)(s{\restriction}X), & (m \cap X \neq \varnothing) \\ s{\restriction}X & (m \cap X = \varnothing). \end{cases} \end{array}$$

An *object* of **ASProc** is a pair $A = (\Sigma_A, S_A)$ where Σ_A is a *sort* or *alphabet*, and $S_A \subseteq \mathtt{Tr}(\Sigma_A)$ is a non-empty prefix-closed subset of $\mathtt{Tr}(\Sigma_A)$, a *safety property* [12].

We define $A^\perp = A$, and $A \otimes B$ by

$$\begin{array}{lcl} \Sigma_{A\otimes B} & = & \Sigma_A + \Sigma_B \\ S_{A\otimes B} & = & \{s \in \mathtt{Tr}(\Sigma_{A\otimes B}) \mid s{\restriction}\Sigma_A \in S_A, s{\restriction}\Sigma_B \in S_B\}. \end{array}$$

Note that $A^\perp = A$ implies that $A \otimes B = A \multimap B = A \parr B$.

A process of sort Σ is a non-empty prefix-closed subset of $\mathtt{Tr}(\Sigma)$. We write $p \models A$ if p is a process of sort Σ_A such that $p \subseteq S_A$.

We write $p : A \to B$ iff $p \models A \multimap B$. A morphism of **ASProc** is a tuple (A, p, B) such that $p : A \to B$ (compare with a careful definition of the category of sets and relations, which also requires the domain and codomain of a morphism to be explicitly recorded).

Given $p : A \to B$, $q : B \to C$, we define $p\,;q : A \to C$ by:

$$p\,;q = \{s{\restriction}\Sigma_A, \Sigma_C \mid s \in \mathtt{Tr}(\Sigma_A + \Sigma_B + \Sigma_C), s{\restriction}\Sigma_A, \Sigma_B \in p, s{\restriction}\Sigma_B, \Sigma_C \in q\}.$$

Compare this to the definition of $[p\|q]$ (parallel composition + hiding) in [31].

Identities are defined by:

$$\mathsf{id}_A = \{s \in \mathtt{Tr}(\Sigma_A + \Sigma_A) \mid \exists t \in S_A. \forall i.\, s_i = \mathtt{inl}(t_i) \cup \mathtt{inr}(t_i)\}.$$

That is, at each stage, the identity simultaneously offers any possible actions of A at both ends of its interface—it behaves like a wire:

$$a \; \overline{\qquad\quad \mathsf{id}_A \qquad\quad} \; a$$

Allowing simultaneous actions plays a crucial rôle at this point.

We can specify the composition in terms of labelled transitions by:

$$\frac{p \xrightarrow{m} p'}{p\,;q \xrightarrow{m} p'\,;q}\; m \cap \Sigma_B = \varnothing \qquad\qquad \frac{q \xrightarrow{n} q'}{p\,;q \xrightarrow{n} p\,;q'}\; n \cap \Sigma_B = \varnothing$$

$$\frac{p \xrightarrow{m} p' \qquad q \xrightarrow{n} q'}{p\,;q \xrightarrow{(m\cup n)\setminus\Sigma_B} p'\,;q'}\; m \cap \Sigma_B = n \cap \Sigma_B$$

Proposition 5.1.1 **ASProc** *is a category.*

We can extend tensor to a functor by:

$$\frac{p : A \to B \qquad q : A' \to B'}{p \otimes q : A \otimes A' \to B \otimes B'}$$

$$p \otimes q = \{s \in \mathtt{Tr}(\Sigma_{A\otimes A' \multimap B\otimes B'}) \mid s{\restriction}\Sigma_A, \Sigma_B \in p, s{\restriction}\Sigma_{A'}, \Sigma_{B'} \in q\}$$

In terms of transitions:

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

The tensor unit I is the trivial type:

$$I = (\varnothing, \{\varepsilon\}).$$

All the further structure of a $*$-autonomous category can be defined on **ASProc**.

Proposition 5.1.2 **ASProc** *is a compact-closed category.*

To illustrate these constructions, we show how to represent the standard CSP operations of composition and hiding. Given a type $A = (\Sigma_A, \Sigma_A^*)$, processes $p, q : I \longrightarrow A$, and a synchronization alphabet $L \subseteq \Sigma_A$, we define $p \parallel_L q$ by

$$p \parallel_L q = I \cong I \otimes I \xrightarrow{p \otimes q} A \otimes A \xrightarrow{m} A$$

where m is defined recursively by

$$\begin{array}{rcl} m & = & \square_{a \in L}\{\mathtt{inl}(\mathtt{inl}(a)), \mathtt{inl}(\mathtt{inr}(a)), \mathtt{inr}(a)\} \to m \\ & \sqcap & \square_{a \notin L}\{\mathtt{inl}(\mathtt{inl}(a)), \mathtt{inr}(a)\} \to m \\ & \sqcap & \square_{a \notin L}\{\mathtt{inl}(\mathtt{inr}(a)), \mathtt{inr}(a)\} \to m. \end{array}$$

The idea is that m is the CSP "synchronization algebra", which is composed with the independent parallel composition of p and q to force synchronization exactly on actions in L.

Similarly, hiding can be defined by

$$p/L = I \xrightarrow{p} A \xrightarrow{\rho_L} A$$

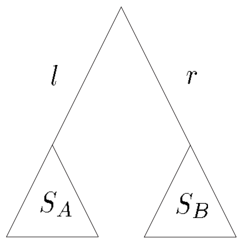
where

$$\rho_L = \mathop{\square}_{a \notin L} \{\mathtt{inl}(a), \mathtt{inr}(a)\} \to \rho_L \sqcap \mathop{\square}_{a \in L} \{\mathtt{inl}(a)\} \to \rho_L.$$

We now turn to the interpretation of the additives in **ASProc**. In fact, **ASProc** only has *weak* products (which are necessarily weak biproducts by the self-duality). These are described as follows. We define

$$\begin{array}{rcl} \Sigma_{A \oplus B} & = & \Sigma_A + \Sigma_B + \{l, r\} \\ S_{A \oplus B} & = & \{\varepsilon\} \cup \{\{l\}s \mid s \in S_A\} \cup \{\{r\}t \mid t \in S_B\} \end{array}$$

so we have



For the weak coproduct diagram

$$\begin{tikzcd} A \arrow[r, "\mathsf{inl}"] \arrow[dr, "p"'] & A \oplus B \arrow[d, dotted, "{[p,q]}"] & B \arrow[l, "\mathsf{inr}"'] \arrow[dl, "q"] \\ & C & \end{tikzcd}$$

we proceed as follows. Firstly, if $p \subseteq \mathtt{Tr}(\Sigma)$ and $f : \Sigma \to \Sigma'$ we define *relabelling* of p by f:

$$p[f] \quad = \quad \{\bar{f}(s) \mid s \in p\}$$

where $\bar{f} : \mathtt{Tr}(\Sigma) \to \mathtt{Tr}(\Sigma')$ is the canonical extension of f.

Now we define

$$\mathsf{inl} = l.\mathsf{id}_A[i] \quad \mathsf{inr} = r.\mathsf{id}_B[j]$$

where

$$i : \Sigma_A + \Sigma_A \to \Sigma_A + \Sigma_{A \oplus B}, \quad j : \Sigma_B + \Sigma_B \to \Sigma_B + \Sigma_{A \oplus B}$$

are the canonical injections, and

$$[p, q] \quad = \quad l.p[m] \cup r.q[n]$$

where

$$m : \Sigma_C + \Sigma_A \to \Sigma_C + \Sigma_{A \oplus B}, \quad n : \Sigma_C + \Sigma_B \to \Sigma_C + \Sigma_{A \oplus B}$$

are the canonical injections, and e.g.

$$l.p \quad = \quad \{\varepsilon\} \cup \{\{l\}s \mid s \in p\}.$$

Proposition 5.1.3 *These definitions yield a weak biproduct, i.e. we have*

$$\mathsf{inl} ; [p, q] = p, \quad \mathsf{inr} ; [p, q] = q.$$

These definitions easily extend to weak biproducts of arbitrary set-indexed families of types $\bigoplus_{i \in I} A_i$.

Note that in process algebra terms, these constructions yield *disjointly guarded sums*: if $p_i : I \to A_i$ then (using the weak product structure now)

$$\langle p_i \rangle_{i \in I} : I \to \bigoplus_{i \in I} A_i$$

is given by

$$\langle p_i \rangle_{i \in I} = \mathop{[]}_{i \in I} a_i \to p_i \qquad (i \neq j \Rightarrow a_i \neq a_j)$$

in CSP terms. Note that our combinators are name-free; the names are controlled indirectly via the types.

Furthermore, if we apply the general definition of non-determinism as semi-additivity, we obtain

$$p + q = p \cup q$$

which is the interpretation of non-determinism in the trace model as given e.g. in [31]. (In the more refined version of **ASProc** in which processes are synchronisation trees modulo weak bisimulation, then $p + q$ is a non-associative version of the CSP "internal choice" $p \sqcap q$ [32].)

5.2 Typing the Scheduler in ASProc

Constructing the scheduler in the asynchronous category **ASProc** is a little simpler, because there is no need to explicitly manipulate delays with the δ and Δ operators as the types have delays built-in. However, to compensate for this we introduce a slight complication which will be important later on. In the previous section we gave the type $\delta\Delta X \otimes \delta\Delta X^{\perp}$ to the port on which a cell communicates with its client. But now, we will use a single type Y for this port with the two actions a and b. This allows us to define a safety specification for Y which is not just constructed from the safety specifications of two simpler types, and which places a real constraint on the interaction between the cell and a client.

The other two ports, as before, have types $X^{\perp}$ and X, where

$$\begin{array}{rcl} \Sigma_X & = & \{\bullet\} \\ S_X & = & \Sigma_X^*. \end{array}$$

The new type Y is defined by

$$\begin{array}{rcl} \Sigma_Y & = & \{\mathtt{inl}(\bullet), \mathtt{inr}(\bullet)\} \\ S_Y & = & a.b.S_Y + ab.S_Y \end{array}$$

where the actions a,b and ab used in the definition of S_Y are abbreviations for $\{\mathtt{inl}(\bullet)\}$, $\{\mathtt{inr}(\bullet)\}$ and $\{\mathtt{inl}(\bullet), \mathtt{inr}(\bullet)\}$.

We can now construct the cell by interpreting the CCS definitions in **ASProc**; the names a and b refer to actions in Y, and the names c and d refer to the actions consisting of the label $\bullet$ in one copy or the other of X. The result is the typing

$$\mathtt{cell} : X_d \parr Y \parr X_c^{\perp}$$

which is established by checking (easily) that the safety specification of Y is satisfied. Following the same reasoning as in **SProc**, we can construct the scheduler with the following typing

$$\mathtt{sched} : Y \parr Y \parr \cdots \parr Y.$$

Therefore we can connect clients to the scheduler as long as they have the type Y, which, because of the non-trivial saefty specification on Y will weed out some processes that could be typed in the example given in Section 3.4. Note that, even in Section 3.4, we could have introduced safety as there is no difference between **SProc** and **ASProc** in this respect.

A sample client process that can be connected is

$$\mathtt{client} = a.b.\mathtt{client}$$

and another is

$$\mathtt{client} = a.b.\mathbf{nil}$$

It is easy to see that these have the right type.

5.3 Safety

Up to now we have not made a great deal of use of the safety specifications featured in the types of **SProc** and **ASProc**. Our typing of the scheduler in **SProc** was based on types whose safety specifications imposed no restriction at all on process behaviour; in **ASProc**, we defined a non-trivial safety specification for one type, but the reason for doing so is, as we shall see in the next section, more related to deadlock-freedom than safety properties. We now want to prove that our implementation of the scheduler satisfies the safety specification associated with the original statement of the problem, namely that the clients are scheduled cyclically. One way to say this is that we have a type for the process `sched`, but now want to give it another type which incorporates the desired safety specification.

In general, what we can do is start with a process p of type A, define a type S with the same alphabet as A but a more restricted safety specification, and prove that p also has type S. However, if $A = B \otimes B$, giving p the type S loses information about the structure of p's interface: it is no longer possible to see that p has two ports of type B. This was illustrated by the types we defined for the scheduler cell in **ASProc**; in order to constrain the pattern of communication between a cell and a client, we had to use a single type for the appropriate port, instead of the combination of two ports by $\otimes$.

We would like to be able to express the fact that p satisfies the stronger specification represented by the type S, without forgetting what the interface of p looks like. The way we do this is by defining a "safety morphism" $s : A \to A$ such that $p : I \to A$ satisfies the specification in S iff $p = p\,;s$. Taking s to be the identity morphism id_S considered as a morphism $A \to A$ (because $S_S \subseteq S_A$), it is easy to check that $p = p;s \iff \mathsf{traces}(p) \subseteq S_S$.

To apply this idea to the scheduler, we will work in **ASProc** (although the argument works equally well in **SProc**), but go back to a situation in which we just have a single type X. We also omit the subscripts on types indicating ports. Thus the type of a cell is

$$\texttt{cell} : X \parr (X^\perp \otimes X) \parr X^\perp.$$

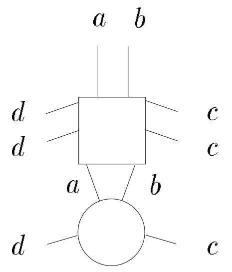
The specification in the original problem lets us define a safety morphism $\texttt{schedsafety} : (X^\perp \otimes X)^n \to (X^\perp \otimes X)^n$ (if n clients are to be scheduled, and the notation A^n means $A \parr \cdots \parr A$). Our goal is therefore to prove that $\texttt{sched} = \texttt{sched}\,;\texttt{schedsafety}$. Since the scheduler is implemented as a combination of n cells, we do this by introducing a safety specification for a cell, such that if each cell is safe then the scheduler is safe. 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.

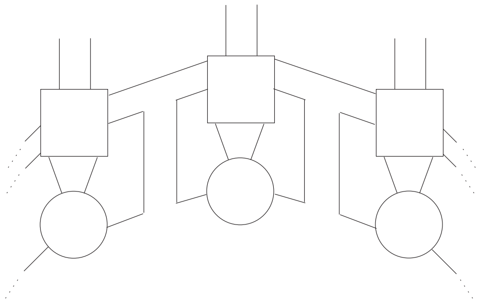
This defines $\texttt{cellsafety} : X \parr (X^\perp \otimes X) \parr X^\perp \to X \parr (X^\perp \otimes X) \parr X^\perp$, and it is clear from the definition of `cell` that $\texttt{cell} = \texttt{cell}\,;\texttt{cellsafety}$. Thus we have

$$\begin{aligned}\texttt{sched} &= (((\texttt{cell} \mid \texttt{cell}) \mid \texttt{cell}) \ldots \mid \texttt{cell}) && (3)\\ &= ((((\texttt{cell}\,;\texttt{cellsafety}) \mid (\texttt{cell}\,;\texttt{cellsafety})) \mid && (4)\\ &\quad (\texttt{cell}\,;\texttt{cellsafety})) \ldots \mid (\texttt{cell}\,;\texttt{cellsafety}))\end{aligned}$$

which describes the scheduler, starting with processes of the form

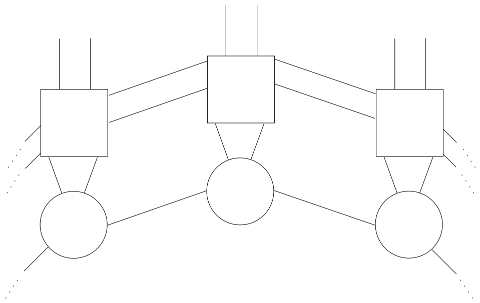


(where the circles are `cell`s and the squares are `cellsafety`s) combined in the following way



The notation `cell` | `cell` is an informal one, denoting the combination of cells in the way which has already been described. The diagrams specify the desired connections more precisely.

Because `cellsafety` is an identity morphism (considered to be in a larger type) and `cell` = `cell` ; `cellsafety`, this arrangement of processes is equivalent to the following alternative arrangement.



To check this equivalence, think 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 we know that `cell` satisfies its specification anyway, 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 we obtain

$$\begin{aligned}\texttt{sched} &= (((\texttt{cell} \mid \texttt{cell}) \mid \texttt{cell}) \ldots \mid \texttt{cell})\,; \\ &\quad (((\texttt{cellsafety} \mid \texttt{cellsafety}) \mid \texttt{cellsafety}) \\ &\quad \ldots \mid \texttt{cellsafety}).\end{aligned} \tag{5}$$

Knowing that if each cell is safe then the scheduler is safe means that the combination of the `cellsafety` processes satisfies the scheduler specification, and so we get

$$\begin{aligned}\texttt{sched} &= (((\texttt{cell} \mid \texttt{cell}) \mid \texttt{cell}) \ldots \mid \texttt{cell})\,; \\ &\quad (((\texttt{cellsafety} \mid \texttt{cellsafety}) \mid \texttt{cellsafety}) \\ &\quad \ldots \mid \texttt{cellsafety})\,; \texttt{schedsafety}.\end{aligned} \tag{6}$$

Working backwards through the previous set of equations (3)–(5):

$$\begin{aligned}\texttt{sched} &= ((((\texttt{cell}\,; \texttt{cellsafety}) \mid (\texttt{cell}\,; \texttt{cellsafety})) \mid && (7)\\ &\quad (\texttt{cell}\,; \texttt{cellsafety})) \\ &\quad \ldots \mid (\texttt{cell}\,; \texttt{cellsafety}))\,; \texttt{schedsafety} \\ &= (((\texttt{cell} \mid \texttt{cell}) \mid \texttt{cell}) && (8)\\ &\quad \ldots \mid \texttt{cell})\,; \texttt{schedsafety} \\ &= \texttt{sched}\,; \texttt{schedsafety}\end{aligned}$$

which is the desired conclusion. It should be clear that this calculation is an example of a general scheme for structuring correctness proofs of processes which are built from subcomponents.

5.4 Axioms for Interaction Categories

The question "What is an Interaction Category" has not yet been answered, although both **SProc** and **ASProc** have been claimed as examples. Ideally we would like to be able to work with an abstract definition. It seems to be easier to find axioms which describe **SProc**, with its stronger structure, than **ASProc**. This suggests that it might be useful to consider *synchronous* and *asynchronous* Interaction Categories as slightly different species. The difficulty in giving an axiomatisation arises from the temporal structure of the categories—as for the rest, we know that we want a Linear category with (possibly weak) biproducts and a commutative monoid structure on each homset.

The temporal structure of **SProc** and **ASProc** springs from the unit delay functor, and the fact that it has the UFPP. More generally, any guarded functor (for example, $\bigcirc^2$ or $\bigcirc \oplus \bigcirc$) has the UFPP. 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. There are then some natural closure conditions on the class of guarded functors, and we would also like to be able to show that they are satisfied. These conditions are:

- $\bigcirc$ is guarded.
- If G and G' are guarded, so is $G \oplus G'$.
- If G is guarded and F is any functor, GF is guarded.

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. Thus the question becomes how to state this categorically. What follows is one possibility.

Definition 5.4.1 *Let $\mathcal{C}$ be a symmetric monoidal category with countable biproducts. Suppose that $\mathcal{C}$ has a monoidal endofunctor $\bigcirc$ such that $\bigcirc(\mathbf{nil}) \neq \mathbf{nil}$, where* **nil** *is the unit of the semi-additive structure induced on a homset by the biproducts. For each A, B, define a function* $\mathsf{firststep}$ *on $\mathcal{C}(A, B)$ by*

$$\mathsf{firststep}(f) = A \cong \bigcirc I \otimes A \xrightarrow{\bigcirc(\mathbf{nil}_{A \to B}) \otimes f} \bigcirc I \otimes B \cong B.$$

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

In **SProc**, the operation $\mathsf{firststep}$ gives the possible actions which a process may do initially. Any **nil** process is an empty synchronisation tree, and so $\bigcirc(\mathbf{nil})$ does only a single action $(*, *)$. The synchronous nature of $\otimes$ means that $\bigcirc(\mathbf{nil}) \otimes f$ 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 $\bigcirc$ on **SProc** has the UFPP also allows us to prove that **SProc** satisfies the Guarded Functor Axiom:

- every guarded functor has the Unique Fixed Point Property.

The definition of $\mathsf{firststep}$ relies on the fact that $\mathbf{nil} \otimes f = \mathbf{nil}$ for any morphism f, which in turn follows from the definition of the semi-additive structure in terms of the biproducts. In an asynchronous category such as **ASProc**, only weak biproducts exist and even if a semi-additive structure is postulated separately, $\mathbf{nil} \otimes f \neq \mathbf{nil}$. The requirement that $\bigcirc(\mathbf{nil}) \neq \mathbf{nil}$ may seem curious, but it is necessary to avoid degeneracy. If $\bigcirc(\mathbf{nil}) = \mathbf{nil}$, then $\mathsf{firststep}(f) = \mathbf{nil}$ 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

$$\begin{tikzcd} A \arrow[r, "\mathsf{id}_A"] \arrow[d, "p"'] & A \arrow[d, "p"] \\ B & B \arrow[l, "\mathsf{id}_B"] \end{tikzcd}$$

commute, ie all homsets would be singletons. There is an analogy here with fixed points of continuous functions on the cpo $(A^\omega, \sqsubseteq)$ of traces over a set A with the prefix order. If f is continuous and $f(\varepsilon) \sqsubset \varepsilon$ 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.

Returning to the abstract setting, we can prove that the class of guarded functors satisfies the desired closure conditions.

Lemma 5.4.2 $\mathsf{firststep}(p \oplus q) = \mathsf{firststep}(p) \oplus \mathsf{firststep}(q)$

PROOF. This follows from distributivity of $\otimes$ over $\oplus$. ∎

Proposition 5.4.3 *1. $\bigcirc$ is guarded.*

2. If G is guarded and F is any functor then GF is guarded.

3. A countable biproduct of guarded functors is guarded.

PROOF.

1. For any f, $\mathsf{firststep}(\bigcirc(f)) = \mathsf{firststep}(\bigcirc(\mathbf{nil}))$.
2. If G is guarded then for any morphisms f and g, $\mathsf{firststep}(G(f)) = \mathsf{firststep}(G(g))$. Hence $\mathsf{firststep}(G(F(f))) = \mathsf{firststep}(G(F(g)))$.
3. Follows from the Lemma. ∎

Thus the (informal) definition of a synchronous Interaction Category is a Linear category with biproducts, satisfying the Guarded Functor Axiom.

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 of relations, **Rel**, and take $\bigcirc$ 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, g, $G(f) = G(g)$. This means that for each A, 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 $p : A \to B$ such that

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "p"'] & GA \arrow[d, "G(p) = G_{AB}"] \\ B & GB \arrow[l, "g"] \end{tikzcd}$$

commutes. This is satisfied because the square defines p uniquely by composition.

So **Rel** appears to be a synchronous Interaction Category. If we want to exclude it, there is a simple condition which is satisfied by **SProc** but not by **Rel**: that $\bigcirc$ be faithful.

For asynchronous categories, finding suitable axioms is harder. It does not seem to be possible (at least at the moment) to define $\mathsf{firststep}$ in a way that works for **ASProc**. An alternative is to use a syntactic description of the desired guarded functors, based on the closure conditions listed previously. In **ASProc**, $F \oplus G$ is guarded even if F and G are not, so the closure conditions are stronger.

Definition 5.4.4 *Let $\mathcal{C}$ be a monoidal category with a monoidal endofunctor $\bigcirc$. Let $\mathcal{GF}$ be the smallest collection of endofunctors satisfying:*

- $\bigcirc \in \mathcal{GF}$.
- *For any endofunctors F and F', $F \oplus F' \in \mathcal{GF}$.*
- *For $G \in \mathcal{GF}$ and any endofunctor F, $GF \in \mathcal{GF}$.*

We can then specify the asynchronous Guarded Functor Axiom:

- Every functor in $\mathcal{GF}$ has the Unique Fixed Point Property.

We can prove that **ASProc** satisfies this axiom by the usual argument, using the $\mathsf{firststep}$ operation which we know how to define in this particular category even if not in general.

6 Types and Specifications

We now turn to the question of how to introduce richer notions of specification into **SProc**. The approach we take in fact addresses a much broader problem, namely how to combine the Propositions-as-Types and Verification paradigms. To the best of our knowledge, this question has not been addressed in the literature before.

We begin by providing some background for our ideas. Two very different views of types and program correctness coexist rather uneasily in the foundations of functional programming. On the one hand, there is the "Propositions-as-Types correspondence" (or "Curry-Howard Isomorphism") originating in Combinatory logic and Proof Theory [18, 34]:

Formulas	Types
Proofs	(Functional) programs
Normalisation	Evaluation (i.e. reduction to normal form)

This correspondence is conceptually very tidy, and guarantees strong properties (e.g. termination) of programs simply by virtue of their type—in logical terms, by virtue of the fact that a well-typed program is necessarily a proof of something.

The drawback of strict adherence to this methodology is that it is very restrictive: even if all feasible *functions* can be defined in such formalisms as System F [27] or Martin-Löf's type theory [44], convenience of expression, and indeed some natural and efficient *algorithms*, may be lost. This drawback is all the more likely to apply with respect to concurrent programming, where specifications may involve much more subtle behavioural properties than in the functional case. Nevertheless, the attractiveness of this approach is witnessed by its appearance in a number of recent textbooks [52, 56].

The prevailing approach in current functional languages, by contrast, uses a much weaker notion of type, essentially that pioneered by Milner in the development of ML [51]. Although the formal content of the portion of type theory which forms the basis of ML-style type-checking is identical to that in the Propositions-as-Types correspondence, the behavioural properties guaranteed by the fact that a program is well-typed in ML are much weaker; essentially only a safety property, that run-time errors cannot occur [51]. This is traded off against the fact that ML-style type reconstruction can be efficiently implemented, and has indeed been outstandingly successful in practice; it seems to achieve a local optimum in striking a balance between allowing the programmer flexibility, and providing a useful conceptual discipline which filters out many errors at an early stage. On the other hand, stronger correctness properties must be guaranteed by a separate phase of explicit verification, using some logic such as LCF [28, 53] or one of its many descendants.

More generally, there is a spectrum of "type disciplines"

$$\ldots \text{ML} \ldots \text{System F} \ldots \text{Intuitionistic Type Theory} \ldots$$

Thus, a System F typing

$$f : \mathbf{list\ nat} \to \mathbf{list\ nat}$$

guarantees that f terminates, but not, say, that it sorts its input; while a type can be written in Martin-Löf's type theory to express this stronger property.

Ideally, one would like to *combine* these approaches, so that in the process of program construction, one module may be guaranteed to satisfy some correctness property simply by virtue of the type discipline under which it has been constructed, while another has been verified "by hand"; and the two modules can then be plugged together across some typed interface, using a compositional type inference rule, without regard to how these properties were established.

We now proceed to outline the basis for such a combined approach. So far as we know, the approach is completely new.

We start with some base category $\mathbb{C}$. A *specification structure* $\mathcal{S}$ for $\mathbb{C}$ is given by:

- A set $P_{\mathcal{S}}A$ of "predicates" on A for each object A of $\mathbb{C}$.
- For each pair of objects A, B a relation

$$\mathcal{S}_{A,B} \subseteq \mathbb{C}(A, B) \times P_{\mathcal{S}}A \times P_{\mathcal{S}}B.$$

We write $\mathcal{S}_{A,B}(f, \varphi, \psi)$ as $\varphi\{f\}\psi$. This is required to satisfy, for all suitable φ, ψ, θ:

$$\varphi\{\mathsf{id}_A\}\varphi \tag{9}$$

$$\varphi\{f\}\psi, \psi\{g\}\theta \Rightarrow \varphi\{f\,;g\}\theta. \tag{10}$$

Given a specification structure $\mathcal{S}$ on $\mathbb{C}$ we can define a new category $\mathbb{C}_{\mathcal{S}}$ as follows:

- The objects of $\mathbb{C}_{\mathcal{S}}$ are pairs (A, φ), where $\varphi \in P_{\mathcal{S}}A$.
- The morphisms $f : (A, \varphi) \longrightarrow (B, \psi)$ are $\mathbb{C}$-morphisms $f : A \longrightarrow B$ such that $\varphi\{f\}\psi$.

The axioms (9), (10) guarantee that $\mathbb{C}_{\mathcal{S}}$ is a category. Moreover, there is a faithful functor $U_{\mathcal{S}} : \mathbb{C}_{\mathcal{S}} \longrightarrow \mathbb{C}$ defined by

$$U_{\mathcal{S}}(A, \varphi) = A, \quad U_{\mathcal{S}}(f) = f.$$

Now suppose that $\mathbb{C}$ carries a type structure specified by some functors and natural transformations. This structure can be lifted to $\mathbb{C}_{\mathcal{S}}$ by endowing each functor with an action on predicates. For example, suppose that $\otimes : \mathbb{C}^2 \to \mathbb{C}$ is a tensor product. Then an action for $\otimes$ is given by a family of functions

$$\otimes_{A,B} : P_{\mathcal{S}}A \times P_{\mathcal{S}}B \longrightarrow P_{\mathcal{S}}(A \otimes B)$$

satisfying

$$\varphi\{f\}\psi, \varphi'\{g\}\psi' \Rightarrow \varphi \otimes \varphi'\{f \otimes g\}\psi \otimes \psi'.$$

Given such an action, the tensor lifts to $\mathbb{C}_{\mathcal{S}}$ by defining

$$(A, \varphi) \otimes (B, \psi) = (A \otimes B, \varphi \otimes_{A,B} \psi).$$

Moreover, $U_{\mathcal{S}}$ preserves $\otimes$.

Similarly, if assoc is the natural isomorphism witnessing the associativity of $\otimes$, then this will lift to $\mathbb{C}_{\mathcal{S}}$ provided that, for all A, B, C, $\varphi \in P_{\mathcal{S}}A$, $\psi \in P_{\mathcal{S}}B$, $\theta \in P_{\mathcal{S}}C$

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

For reference purposes, we explicitly list all the actions which must be defined and axioms to be satisfied in order to lift the structures exhibited for **SProc** in Section 2.

Linear category structure

Actions:

$$()^{\perp}_{A} : P_{\mathcal{S}}A \longrightarrow P_{\mathcal{S}}A^{\perp}$$

$$\otimes_{A,B} : P_{\mathcal{S}}A \times P_{\mathcal{S}}B \longrightarrow P_{\mathcal{S}}(A \otimes B)$$

$$I_{\mathcal{S}} :\longrightarrow P_{\mathcal{S}}I$$

$$\oplus_{A,B} : P_{\mathcal{S}}A \times P_{\mathcal{S}}B \longrightarrow P_{\mathcal{S}}(A \oplus B)$$

$$\mathbf{0}_{\mathcal{S}} :\longrightarrow P_{\mathcal{S}}\mathbf{0}$$

$$\otimes^{n}_{A} : (P_{\mathcal{S}}A)^{n} \longrightarrow P_{\mathcal{S}}(\otimes^{n}_{s}A)$$

(The action for the exponentials is formulated for the strong form of model as described in Section 2.6. The standard weaker notion of model can also easily be catered for.)

Axioms:

$$\phi^{\perp\perp} = \phi$$

We then define

$$\begin{aligned} \phi \parr \psi &\equiv (\phi^{\perp} \otimes \psi^{\perp})^{\perp} \\ \phi \multimap \psi &\equiv \phi^{\perp} \parr \psi. \end{aligned}$$

(We shall follow the practice of omitting type subscripts wherever possible).

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

$$(\varphi \otimes \psi) \otimes \theta)\{\mathsf{assoc}\}(\varphi \otimes \psi) \otimes \theta$$

$$\varphi \otimes \psi\{\mathsf{symm}\}\psi \otimes \varphi$$

$$\varphi \otimes I_{\mathcal{S}}\{\mathsf{unit}\}\varphi$$

$$\varphi\{f\}\psi \Rightarrow \psi^{\perp}\{f^{\perp}\}\phi^{\perp}$$

$$(\varphi \multimap \psi) \otimes \varphi\{\mathsf{Ap}\}\psi$$

$$\phi \otimes \psi\{f\}\theta \Rightarrow \varphi\{\Lambda(f)\}\psi \multimap \theta$$

$$\mathbf{0}_{\mathcal{S}}\{\mathbf{0}_{A}\}\varphi$$

$$\varphi\{\mathsf{inl}\}\varphi \oplus \psi$$

$$\psi\{\mathsf{inr}\}\varphi \oplus \psi$$

$$\varphi\{f\}\theta, \psi\{g\}\theta \Rightarrow \varphi \oplus \psi\{[f,g]\}\theta$$

$$\varphi\{f\}\psi_1, \ldots, \varphi\{f\}\psi_n \Rightarrow \varphi\{\hat{f}\}\otimes^{n}_{A}(\psi_1, \ldots, \psi_n)$$

for f commuting with all p_{σ}, $\sigma \in S_n$.

Temporal Constructions

For illustration we consider unit delay. For each A, a function

$$\bigcirc_A : P_{\mathcal{S}}A \longrightarrow P_{\mathcal{S}}(\bigcirc A)$$

satisfying

$$\phi\{p\}\psi \Rightarrow \bigcirc\phi\{\bigcirc p\}\bigcirc\psi.$$

In applying these general ideas, it will often be the case that a specification structure $\mathcal{S}$ will have considerable local structure in the fibres $P_{\mathcal{S}}A$. These will often be lattices supporting an interpretation of various logical connectives and modal or temporal operators. Standard verification or model-checking techniques can then be applied to establishing the validity of specific "Hoare triples" $\varphi\{f\}\psi$.

In the particular case where $\mathbb{C} = \mathbf{SProc}$, it will be convenient to define, for each object A, a *satisfaction relation*

$$\models_A \subseteq \mathsf{ST}_{\Sigma_A} \times P_{\mathcal{S}}A$$

and then to define

$$\varphi\{p\}\psi \stackrel{\Delta}{\iff} p \models_{A\multimap B} \varphi \multimap \psi.$$

Returning to our objective of combining types and verification, we have the following picture. By progressively adding logical properties of various kinds to our arsenal of specifications, yielding specification structure $P_1, \ldots, P_k$ as above, we can correspondingly construct a tower of "refinements"

$$\mathbb{C} \stackrel{U_1}{\hookleftarrow} \mathbb{C}_1 \cdots \stackrel{U_k}{\hookleftarrow} \mathbb{C}_k \tag{11}$$

where $\mathbb{C}_i = \mathbb{C}_{P_1,\ldots,P_i}, 1 \leq i \leq k$. Objects will be structures

$$(A, \varphi_1, \ldots, \varphi_k)$$

where $\varphi_i \in P_i(A)$; morphisms

$$p : (A, \varphi_a, \ldots, \varphi_k) \longrightarrow (B, \psi_1, \ldots, \psi_k)$$

will be $\mathbb{C}$-morphisms $p : A \rightarrow B$ satisfying $\varphi_i\{p\}\psi_i, 1 \leq i \leq k$. Thus the underlying computational objects, the processes (i.e. the proto-morphisms of the category) stay the same, but the specifications grow more subtle, and it is correspondingly harder to be a morphism in $\mathbb{C}_j$ than in $\mathbb{C}_i$ for $i < j$. This will have the effect of sharpening up the type structure as we move rightwards in (11); e.g. even if $\mathbb{C}$ is compact-closed, $\mathbb{C}_k$ may well not be. However, constructions of a logical flavour, e.g. those arising from the linear types, are guaranteed to be well-typed across the spectrum, even with the most demanding notion of specification. Thus we can combine steps of program construction arising from these logical constructions, on the one hand; and those from more ad hoc constructions, which will only a priori satisfy some relatively crude type to the left of the spectrum in (11), and must be explicitly verified using the logics of the remaining P_i. When we come to combine two such modules, e.g. by composition (interaction)

$$p : (A, \varphi) \longrightarrow (B, \psi), \quad q : (B, \psi) \longrightarrow (C, \theta)$$

it will make no difference by what combination of type checking and verification steps we came to know that these premises were satisfied; from the mere fact that they are, we can form the composition

$$p; q : (A, \varphi) \longrightarrow (C, \theta)$$

without further ado. Thus *composition* in $\mathbb{C}_k$ does indeed yield a *compositional* verification rule for *parallel composition + restriction*.

It remains to demonstrate that useful behavioural properties can be handled in this framework. We will develop two concrete examples for **SProc** in the remainder of this section.

6.1 Fair Computations and Liveness Properties

We begin by considering linear-time liveness properties [12]. A first attempt at defining such properties over a type A would be:

$$PA = (\wp(\Sigma_A^\omega), \subseteq)$$

$$p \models_A L \overset{\Delta}{\Longleftrightarrow} \mathsf{inftraces}(p) \subseteq L$$

where

$$\mathsf{inftraces}(p) = \{s \in \Sigma_A^\omega \mid \exists (p_n)_{n\in\omega}.[p = p_0 \wedge \forall i \in \omega . p_i \xrightarrow{s_i} p_{i+1}]\}.$$

Note that $\wp(\Sigma_A^\omega)$ provides a model for linear-time temporal logic, which has been extensively applied to concurrency [43]. This example will allow us to demonstrate an important feature of the framework: the clear distinction it supports between *processes* (computational entities) and *specifications* (logical entities). Although it can be tempting to blur this distinction, we firmly believe that doing so leads to confusion, and ultimately to insuperable technical problems. A perfect example of this is afforded by that classic bugbear of concurrency theory, *fairness*. In our opinion, the problems which have invariably arisen in attempts to give semantics for fairness stem from trying to force infinitary notions which belong in the logical realm of specifications into the computational realm of processes, where they do not fit.

To be specific, we focus on the theory of fairness for SCCS developed by Milner and Hennessy [47, 29, 30]. Milner introduced an "expectation" operator ε, which is similar to δ but carries the idea that delays must be finite, albeit unbounded. In fact, εp is so similar to δp that it has the same transitions; the difference must be found in a refined notion of bisimulation, involving rather intensional, syntactic considerations on the structure of computations. Hennessy's semantics has the typical feature of denotational treatments of fairness, that computability (or even continuity) considerations are left far behind.

Our proposal is to take εp and δp as *the same protomorphisms*; and to distinguish between them by the types they bear.

With this motivation, we proceed to define a specification structure P_L. Given $A \in$ $\mathsf{Ob}\,\mathbf{SProc}$, we define

$$P_L A = \{L \mid L \subseteq \wp(\Sigma_A^\omega)\}.$$

Thus liveness properties are taken to be arbitrary predicates on infinite traces. For a more restricted (and standard) notion of liveness property see [12].

Satisfaction is defined by

$$p \models_A L \iff \forall s \in \mathsf{inftraces}(p).[s \in L].$$

The remaining definitions are as follows:

$$L_A^\perp = \Sigma_A^\omega \setminus L$$

$$L \otimes L' = \{s \in \Sigma_{A\otimes B}^\omega \mid \mathsf{fst}^\omega(s) \in L, \mathsf{snd}^w(s) \in L'\}$$

$$I_P = \{*^\omega\}$$

$$\mathbf{0}_P = \varnothing$$

$$L \oplus_{A,B} L' = \{\mathtt{inl}(s) \mid s \in L\} \cup \{\mathtt{inr}(t) \mid t \in L'\}$$

$$\otimes_s^n(L_1, \ldots, L_n) = \phi_n^\omega(L_1 \otimes \ldots \otimes L_n)$$

where $\phi_n(a_1, \ldots, a_n) = \{\!|a_1, \ldots, a_n|\!\}$.

Finally, we consider the delay monads. For δ, the operation on liveness properties is

$$\delta_A : PA \rightarrow P\delta A$$

$$\delta_A L = \{s \in \Sigma_{\delta A}^\omega \mid s{\restriction}\Sigma_A \in L\} \cup \{*^\omega\}$$

while for ε we define

$$\varepsilon_A : PA \rightarrow P(\varepsilon A)$$

$$\varepsilon_A L = \{s \in \Sigma_{\varepsilon A}^{\omega} \mid s{\upharpoonright}\Sigma_A \in L\}.$$

The action of ε on (proto)morphisms is identical to that of δ. In particular, for $p : A \rightarrow B$, δp is identical as a protomorphism to εp. The difference between them is solely in the types they bear, $\delta A \rightarrow \delta B$ vs. $\varepsilon A \rightarrow \varepsilon B$; this in turn reduces to the presence of $*^{\omega}$ in δL, and its absence in εL.

Proposition 6.1.1 *P_L is a specification structure satisfying (SS1)-(SS6); hence $\mathbb{C}_{P_L}$ is a model of Linear Logic and the delay functors. $\mathbb{C}_{P_L}$ is not compact closed; moreover $I \neq \perp$.*

6.2 Deadlock-free Processes

The linear-time properties considered so far are inadequate to ensure that processes are deadlock-free. (Linear-time temporal logic as standardly applied in concurrency overcomes this limitation by allowing formulae to speak about states, including "program points". We have preferred to maintain a more extensional view, in which the properties only refer to observable behaviour.) In order to control deadlocks, we must admit some branching-time properties. A general treatment of such properties within our framework is not attempted here. Instead, we will focus on a restricted class of properties which suffice to guarantee deadlock-freedom. *Once we know that processes are deadlock-free*, linear-time safety and liveness properties of the form already considered seem to be perfectly adequate for specification purposes. The view we shall take is that finite termination = deadlock. That is, if $p \xrightarrow{s} \mathbf{0}$ for some s, then p can deadlock. Thus we do not allow directly for the possibility of successful finite termination. This can still be catered for, by representing a termination state $p\checkmark$ by infinite delay

$$p\checkmark \equiv (\checkmark, \checkmark){:}p\checkmark$$

and using the liveness properties of Section 6.1 to specify that termination after s is successful by

$$s(\checkmark, \checkmark)^{\omega} \in L.$$

This is perhaps a little crude, but simplifies the discussion.

Unfortunately, deadlock-free processes do not form a sub-category: they are easily seen not to be closed under composition. This suggests looking for a specification structure P_D just strong enough to guarantee deadlock-freedom.

The 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 $p \xrightarrow{s} q$ there is $a \in \Sigma_A$ and a process r such that $q \xrightarrow{a} r$. Convergence means deadlock-freedom; the reason for the choice of terminology is an analogy with proofs of strong normalisation in Classical Linear Logic [26, 1].

Given processes p and q of type A, the process $p \sqcap q$ of type A is defined by

$$\frac{p \xrightarrow{a} p' \qquad q \xrightarrow{a} q'}{p \sqcap q \xrightarrow{a} p' \sqcap q'.}$$

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

$$p \perp q \quad \stackrel{\Delta}{\Longleftrightarrow} \quad (p \sqcap q){\downarrow}.$$

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 **SProc** 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 **SProc** consisting of just the progressive objects is denoted by $\mathbf{SProc}_{pr}$; this is the category over which the specification structure for deadlock-freedom will be defined. $\mathbf{SProc}_{pr}$ inherits all the structure of **SProc**, apart from the zero object.

For each object A of $\mathbf{SProc}_{pr}$, 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 $p : A$,

$$\begin{array}{rcl} p \perp U & \stackrel{\Delta}{\Longleftrightarrow} & \forall q \in U.\, p \perp q \\ U \perp V & \stackrel{\Delta}{\Longleftrightarrow} & \forall p \in U.\, p \perp V. \end{array}$$

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

$$U^{\perp} \;=\; \{p \in \mathsf{Proc}(A) \mid p \perp U\}.$$

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

Lemma 6.2.1 *For all $U, V \subseteq \mathsf{Proc}(A)$,*

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

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

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

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.

$$p \models U \quad \stackrel{\Delta}{\Longleftrightarrow} \quad p \in U$$

When the linear operations have been defined on properties, the definition

$$U\{f\}V \quad \stackrel{\Delta}{\Longleftrightarrow} \quad f \models U \multimap V$$

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

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

$$\begin{aligned} U \otimes V &= \{p \otimes q \mid p \in U, q \in V\}^{\perp\perp} \\ U \mathbin{⅋} V &= (U^{\perp} \otimes V^{\perp})^{\perp} \\ U \multimap V &= (U \otimes V^{\perp})^{\perp}. \end{aligned}$$

In order to prove that D satisfies the specification structure axioms, a few lemmas are needed.

Lemma 6.2.2 *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 6.2.3 *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 6.2.4 *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 6.2.5 *D is a specification structure over $\mathbf{SProc}_{pr}$.*

Proof. The first requirement is that if A is any object of $\mathbf{SProc}_{pr}$ and $U \in P_D A$, $U\{\mathsf{id}_A\}U$. This follows from Lemma 6.2.2.

Next, suppose that A, B, C are objects of $\mathbf{SProc}_{pr}$ 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 6.2.3 and 6.2.4, $p\,;f \in V$ and $g\,;r \in V^{\perp}$. Hence $(p\,;f) \perp (g\,;r)$, i.e. $((p\,;f) \sqcap (g\,;r))\downarrow$. It follows that $((f\,;g) \sqcap (p \otimes r))\downarrow$, as can easily be checked. ∎

It is now legitimate to talk about the category $\mathbf{SProc}_D$ of deadlock-free processes. Next, the definition of the $*$-autonomous structure on $\mathbf{SProc}_D$ 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 $\{p\}^\perp = \{p\}$. This means that $I_D = \perp_D$. For $\otimes$ to be a functor on $\mathbf{SProc}_D$ requires that for morphisms $f : A \to C$ and $g : B \to D$ of $\mathbf{SProc}$ 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 $\mathbf{SProc}$ lifts to $\mathbf{SProc}_D$ provided that, for all properties U, V, W over appropriate objects, the following hold.

$$\begin{array}{c} U \otimes (V \otimes W)\{\mathsf{assoc}_{A,B,C}\}(U \otimes V) \otimes W \\ (U \otimes V)\{\mathsf{symm}_{A,B}\}(V \otimes U) \\ (I_D \otimes U)\{\mathsf{unitl}_A\}U \\ (U \otimes I_D)\{\mathsf{unitr}_A\}U. \end{array}$$

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 6.2.6 *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 $q' = q[(b,a) \mapsto (a,b)]$, 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 t is a common trace of symm and $p \otimes q$. The definition of symm means that $\mathsf{fst}^*(t) = \mathsf{snd}^*(t)[(b,a) \mapsto (a,b)]$. Also, $\mathsf{fst}^*(t)$ is a trace of p and $\mathsf{snd}^*(t)[(b,a) \mapsto (a,b)]$ 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 $\mathbf{SProc}$ will soon be defined on $\mathbf{SProc}_D$, 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 6.2.7 *For every object A of $\mathbf{SProc}_{pr}$, $\{\mathsf{max}_A\}^\perp = M_A$ 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 $p \xrightarrow{s}{}^* p'$ 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 6.2.8 $\{\mathsf{max}_A\}^{\perp\perp} = \{\mathsf{max}_A\}$ *and* $M_A^{\perp\perp} = M_A$.

This means that for every type A, there are two properties over A: M_A and $\{\mathsf{max}_A\}$. However, they are not always distinct.

Proposition 6.2.9 *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 $\mathbin{\bindnasrepma}$ of these properties.

Proposition 6.2.10 *For any objects A and B of* $\mathbf{SProc}_{pr}$*,*

$$\{\mathsf{max}_A\} \otimes \{\mathsf{max}_B\} = \{\mathsf{max}_{A\otimes B}\}.$$

PROOF. This follows from the fact that $\mathsf{max}_A \otimes \mathsf{max}_B = \mathsf{max}_{A\otimes B}$. ∎

Corollary 6.2.11 $M_A \mathbin{\bindnasrepma} M_B = M_{A \mathbin{\bindnasrepma} B}$.

Proposition 6.2.12 *For any objects A and B of* $\mathbf{SProc}_{pr}$*, $M_A \otimes M_B = M_{A\otimes B}$.*

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 6.2.13 $\{\mathsf{max}_A\} \mathbin{\bindnasrepma} \{\mathsf{max}_B\} = \{\mathsf{max}_{A \mathbin{\bindnasrepma} B}\}$.

Properties of the form M_A and $\{\mathsf{max}_A\}$ can be used to show that $\mathbf{SProc}_D$ is not compact closed. Consider the types A and B with $\Sigma_A = \{a, b\}$, $\Sigma_B = \{c, d\}$ and unrestricted safety specifications. Then

$$\begin{aligned} M_A \otimes \{\mathsf{max}_B\} &= \{p \otimes \mathsf{max}_B \mid p \in M_A\}^{\perp\perp} \\ M_A \mathbin{\bindnasrepma} \{\mathsf{max}_B\} &= (\{\mathsf{max}_A\} \otimes M_B)^{\perp} \\ &= \{\mathsf{max}_A \otimes q \mid q \in M_B\}^{\perp}. \end{aligned}$$

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 $M_A \otimes \{\mathsf{max}_A\} \neq M_A \mathbin{\bindnasrepma} \{\mathsf{max}_B\}$.

Loss of compact closure is to be expected, as in general the arbitrary formation of cycles is likely to lead to deadlock. $\mathbf{SProc}_D$ 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 $\mathbf{SProc}_{pr}$ has non-empty coproducts, and hence also products. These can be lifted to $\mathbf{SProc}_D$.

$$\begin{aligned} U \oplus V &= (\{p[a \mapsto \mathtt{inl}(a)] \mid p \in U\} \cup \{q[b \mapsto \mathtt{inr}(b)] \mid q \in V\})^{\perp\perp} \\ U \& V &= (U^{\perp} \oplus V^{\perp})^{\perp}. \end{aligned}$$

The requirements for these to define products and coproducts on $\mathbf{SProc}_D$ are that for any progressive objects A, B of **SProc** and $U \in P_D(A)$, $V \in P_D(B)$:

$$(U \& V)\{\pi_1\}U$$
$$(U \& V)\{\pi_2\}V$$
$$U\{\mathsf{inl}\}(U \oplus V)$$
$$V\{\mathsf{inr}\}(U \oplus V).$$

The definitions extend to countable products and coproducts in the obvious way. The zero object of **SProc** is not progressive, because it has an empty sort, and so it cannot be lifted to $\mathbf{SProc}_D$.

Not only do $\otimes$ and $\invamp$ become distinct in $\mathbf{SProc}_D$, 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

$$\begin{array}{rcl} M_A \oplus M_B & = & (\{\mathsf{max}_A\} \cup \{\mathsf{max}_B\})^{\perp\perp} \\ & = & \{\mathsf{max}_A, \mathsf{max}_B\}^{\perp\perp} \\ M_A \& M_B & = & (M_A \oplus M_B)^{\perp} \\ & = & \{\mathsf{max}_A, \mathsf{max}_B\}^{\perp}, \end{array}$$

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 $\mathbf{SProc}_D$ 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 $\mathbf{SProc}_D$ as they are not convergent. So each homset of $\mathbf{SProc}_D$ has a commutative and associative $+$ operation but this operation has no unit.

The delay functors are lifted to $\mathbf{SProc}_D$ by means of the following definitions.

$$\begin{array}{rcl} \bigcirc U & = & \{\bigcirc p \mid p \in U\}^{\perp\perp} \\ \delta\, U & = & \{\delta\, p \mid p \in U\}^{\perp\perp} \\ \Delta\, U & = & \{\Delta\, p \mid p \in U\}^{\perp\perp}. \end{array}$$

In the case of $\bigcirc$ the application of $(-)^{\perp\perp}$ is unnecessary, as $\{\bigcirc p \mid p \in U\}$ is already $^{\perp\perp}$-invariant. The conditions required for $\bigcirc$, δ and Δ to be functors on $\mathbf{SProc}_D$ are

$$\bigcirc U\{\bigcirc f\}\bigcirc V$$
$$\delta\, U\{\delta\, f\}\delta\, V$$
$$\Delta\, U\{\Delta\, f\}\Delta\, V$$

when $f : A \to B$ in **SProc**, $U \in P_D A$, $V \in P_D B$ and $U\{f\}V$. The conditions for the monad (δ, η, μ) to lift to $\mathbf{SProc}_D$ are

$$U\{\eta\}\delta\, U$$
$$\delta\, \delta\, U\{\mu\}\delta\, U$$

and similarly for Δ. The functor $\bigcirc$ on $\mathbf{SProc}_D$ has the UFPP provided that whenever $f : A \to \bigcirc A$ and $g : \bigcirc B \to B$ satisfy $U\{f\}\bigcirc U$ and $\bigcirc V\{g\}V$, the morphism $h : A \to B$ defined by the UFPP in **SProc** satisfies $U\{h\}V$.

All of these conditions are satisfied, so the entire temporal structure of **SProc** lifts to $\mathbf{SProc}_D$.

The lack of biproducts means that the construction of ! as a cofree cocommutative comonoid cannot be used in $\mathbf{SProc}_D$. But it is sufficient to define ! on sets of processes and check that the structural morphisms lift from **SProc** to $\mathbf{SProc}_D$.

$$\begin{array}{rcl} !\,U & = & \{\,!\,p \mid p \in U\}^{\perp\perp} \\ ?\,U & = & (\,!(U^{\perp}))^{\perp} \end{array}$$

The conditions required of the structural morphisms are

$$\begin{array}{c} !\,U\{\mathsf{weak}\}I_D \\ !\,U\{\mathsf{contr}\}\,!\,U \otimes\,!\,U \\ !\,U\{\mathsf{der}\}U \end{array}$$

and, if $f : \,!\,A \to B$ with $!\,U\{f\}V$,

$$!\,U\{f^{!}\}\,!\,V.$$

Proposition 6.2.14 $\mathbf{SProc}_D$ *is a* $*$*-autonomous category with countable (non-empty) products and coproducts, exponentials, unit delay functor and delay monads. The forgetful functor* $U : \mathbf{SProc}_D \to \mathbf{SProc}$ *preserves all of this structure.*

6.3 Deadlock-Freedom of the Scheduler

As a final example, we would like to give a proof that the scheduler does not deadlock, based on the use of types. In the previous section a category $\mathbf{SProc}_D$ of deadlock-free processes was constructed from a specification structure D over **SProc**. The obvious thing to try is to construct the scheduler with types in $\mathbf{SProc}_D$, and deduce that it does not deadlock. Unfortunately, things are not that simple. When constructing the scheduler in **SProc**, we introduced (via the δ and Δ operators) an action $*$ to stand for delay. Since we wanted the scheduler to be asynchronous, the cells were always able to perform delaying actions at any point. Therefore, the cell can be typed, and cells can be connected together and guaranteed to be deadlock-free, but the only thing we can deduce about the resulting process is that it can always delay. Thus this analysis fails to take account of the fact that we are not interested in delays, and want to know about deadlock-freedom in the sense of significant actions being possible.

The obvious solution to this difficulty is to construct a category of asynchronous deadlock-free processes, by means of a specification structure D over **ASProc**. The remainder of this section describes the construction.

From now on we will work with the full version of **ASProc** [22], rather than the simplified version presented earlier in this paper in which a process is a set of traces. The main reason for this is that the theory of asynchronous deadlock-freedom has already been developed for the full version of **ASProc**; it can be easily transferred to the simplified case.

The types in the full version of **ASProc** are the same as before, but to obtain the processes we now start with synchronisation trees as in **SProc**. Actions are still sets of labels, but now the empty set is a valid action. The key step is to work with equivalence classes of synchronisation trees modulo weak bisimulation (observation equivalence) [49]; the empty set plays the rôle of the silent action τ of CCS. The structure of the category is defined as before.

In fact, a further slight variation of **ASProc** is needed. The reason for this will be explained later, but for now just suppose that the definitions of convergence and orthogonality of processes are transferred directly to **ASProc**. The specification structure D is then defined as before, and the following general results are still available.

Proposition 6.3.1 *If a type* A *has only one action,* $M_A = (M_A)^{\perp}$.

Proposition 6.3.2 *If $p : A_1 \parr A_2 \parr \cdots \parr A_n$ in* **ASProc** *and $p{\downarrow}$, then $p : (A_1, M_{A_1}) \parr (A_2, M_{A_2}) \parr \cdots \parr (A_n, M_{A_n})$ in* $\mathbf{ASProc}_D$.

Therefore, given that the `cell` and `dcell` have the type $X_d \parr Y \parr X_c^{\perp}$ in **ASProc**, we can give them the following types in $\mathbf{ASProc}_D$:

$$\texttt{cell}, \texttt{dcell} : \quad (X, M_X)_d \parr (Y, M_Y) \parr (X^{\perp}, M_X)_c.$$

Using the fact that $M_X = M_X{}^{\perp}$, we can rewrite the types in the form

$$\begin{aligned} \texttt{cell}, \texttt{dcell} : &\quad (X, M_X)_d \parr (Y, M_Y) \parr (X^{\perp}, M_X^{\perp})_c \\ : &\quad (X, M_X)_d \parr (Y, M_Y) \parr (X, M_X)_c^{\perp} \end{aligned}$$

which have the same general shape as the types in Section 5.2.

Therefore we can plug copies of `cell` and `dcell` together in a linear string and the resulting process is guaranteed to be deadlock-free.

We need to formulate and check additional conditions to complete the cycle as we cannot form cycles in general in $\mathbf{ASProc}_D$. This is done by reasoning about actions occurring at the ports that need to be plugged together at the end to form the cycle.

Suppose in general that $P : (\Gamma, U) \parr (X, V) \parr (X^{\perp}, V^{\perp})$ in $\mathbf{ASProc}_D$. 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 α of P. The projection $\alpha{\restriction}X, X^{\perp}$ could be $\varnothing$, as it is permissible for the sequence of communications between the X and $X^{\perp}$ ports to pause, or the projection $\alpha{\restriction}\Gamma$ could be $\varnothing$, but not both. To obtain $\overline{P} : (\Gamma, U)$ in $\mathbf{ASProc}_D$ it is also necessary to ensure that the specification U can still be satisfied while the communication is taking place. All we need is a condition which guarantees that forcing X and $X^{\perp}$ to communicate does not affect the actions available in the other ports. The following definitions are useful to formulate the condition:

$$\begin{aligned} \mathsf{initials}(P) &= \{a \in \mathsf{Act}^{+}(A) \mid \exists Q. P \xrightarrow{a} Q\} \\ \mathsf{readies}(P) &= \{(s, X) \mid \exists Q.[(P \xrightarrow{s} Q) \wedge (X = \mathsf{initials}(Q))]\} \end{aligned}$$

where $\mathsf{Act}^{+}(A)$ is the set of non-empty subsets of Σ_A. We now define a condition $\mathsf{cycle}(P)$:

- For every $(s, A) \in \mathsf{readies}(P)$ such that $s{\restriction}X = s{\restriction}X^{\perp}$, and every action $\alpha \in A$, there is $z \in \Sigma_X$ such that $\varnothing \neq \alpha{\restriction}\Gamma \cup \{\texttt{inl}(z)\} \cup \{\texttt{inr}(z)\} \in A$.

This leads to a proof rule for cycle formation.

$$\frac{P : (\Gamma, U) \parr (X, V) \parr (X^{\perp}, V^{\perp}) \quad \mathsf{cycle}(P)}{\overline{P} : (\Gamma, U)}$$

Completing the cycle for the scheduler amounts to connecting the c port of a `dcell` to a d port of another `dcell`. From the definition of `dcell`, it is clear that these ports can both delay or an output from the c port of one `dcell` will be accepted by the d port of the other. Therefore the conditions of the cycle rule are satisfied and after we complete the cycle we have

$$\texttt{sched} : (Y, M_Y) \parr (Y, M_Y) \cdots \parr (Y, M_Y)$$

and it easy to check that

$$\texttt{client} : (Y, M_Y)^{\perp}$$

Again, clients can be connected to the scheduler just on the basis of their type. Any client of type $(Y, M_Y)^{\perp}$ can be connected to the scheduler and the resulting system would be deadlock-free. It is easy to check (even easier to see) which of the two clients mentioned in Section 5.2 can be given the type $(Y, M_Y)^{\perp}$.

The essential point is that the way we set up our types gives us results about deadlock-freedom for most of our problem and we need to use some specific reasoning for the remaining part. This reflects the fact that verification of programs cannot be done completely automatically, but type checking should reduce the effort to a great extent.

As hinted before, there are a couple of problems with constructing the category $\mathbf{ASProc}_D$ in this straightforward way. The first is that because the tensor unit I has no actions, it cannot be given a type in $\mathbf{ASProc}_D$. The second is to do with divergence: if $p : A \to B$ and $q : B \to C$ in $\mathbf{ASProc}_D$ then they can be composed without deadlocking each other, but there is still the possibility that they both perform actions only in B, and never in A or C. The result of the composition is then a morphism $A \to C$ which never does anything in A or C, in other words a deadlock. Thus $\mathbf{ASProc}_D$ is not a category of deadlock-free processes after all.

To avoid problems of divergence, we need a further constraint on processes. Hoare [32] discusses a similar issue in the context of CSP. He considers processes with one input and one output, designed to be connected in a pipeline—in fact, this comes rather close to our categorical view. Exactly the same divergence problem (described as *livelock*) arises in this situation, and two conditions on processes are formulated in order to avoid it. If p and q are to be connected in line, to form the process which we would denote by $p\,;q$, a sufficient condition for $p\,;q$ to be non-divergent is that p be *left-guarded* and q be *right-guarded*. Left-guardedness means that p cannot perform an infinite sequence of actions in its right port without doing some actions in its left port in between; right-guardedness is defined symmetrically. It is easy to see that these conditions ensure that $p\,;q$ does not diverge.

Adapting this idea to our categorical situation, we would like every morphism to be both left- and right-guarded, so that all composites are non-divergent. We can rephrase the guardedness conditions as a *fairness* condition: any infinite behaviour of p must do infinitely many actions in both the left and the right ports. However, we can't just say that every morphism must be fair between its left and right ports, because we still want to work in a $*$-autonomous category. For any morphism $f : A \otimes B \to C$ we have $\Lambda(f) : A \to (B \multimap C)$. Now f has to be fair between $A \otimes B$ and C, while $\Lambda(f)$ has to be fair between A and $B \multimap C$, but f and $\Lambda(f)$ are just different views of some underlying process p. Since it is not possible to deduce the fairness condition on f from that on $\Lambda(f)$, or vice versa, there must be some fairness condition on p which implies both of the others. What we do is introduce a new category $\mathbf{FProc}$ (fair processes) in which each object specifies a set of infinite traces which are considered to be fair. These sets are combined by $\otimes$ in such a way that fairness of a behaviour in $A \otimes B$ means fairness in A and B.

Formally, the category $\mathbf{FProc}$ has objects $A = (\Sigma_A, S_A, F_A)$. The first two components of an object are exactly as in $\mathbf{ASProc}$. The third, F_A, is a subset of $(\mathsf{Act}^+(A))^\omega$ such that all finite prefixes of any trace in F_A are in S_A. The interaction category operations are defined as in $\mathbf{ASProc}$, with the addition that

$$\begin{array}{rcl} F_{A^\perp} & = & F_A \\ F_{A\otimes B} & = & \{s \in (\mathsf{Act}^+(A \otimes B))^\omega \mid (s{\restriction}A \in F_A) \wedge (s{\restriction}B \in F_B)\} \\ F_{A\oplus B} & = & \{l.\mathsf{inl}^\omega(s) \mid s \in F_A\} \cup \{r.\mathsf{inr}^\omega(t) \mid t \in F_B\} \\ F_{\bigcirc A} & = & \{*s \mid s \in F_A\}. \end{array}$$

A process of type A in $\mathbf{FProc}$ is a pair (p, T_p) in which p is a process of type A in $\mathbf{ASProc}$ and T_p is a subset of the infinite traces obtainable from the synchronisation tree p. The idea is that although the tree p may have many infinite traces, only those in T_p are considered to be actual infinite behaviours of the process (p, T_p). The final condition for (p, T_p) to have type A is that $T_p \subseteq F_A$. Morphisms are then defined as usual. The $*$-autonomous structure works in $\mathbf{FProc}$, because all the structural morphisms are fair and the abstraction operation does not affect fairness. The exception to this is that there is no tensor unit: a morphism $f : I \to A$ could not be fair because there are no infinite traces at all over I.

There is one more subtlety relating to the functorial action of $\otimes$. If $f : A \to C$ and $g : B \to D$ are fair, it is still possible for $f \otimes g : A \otimes B \to C \otimes D$ to be unfair, because the definition of $\otimes$ on morphisms in **ASProc** allows all interleavings of actions from f and g, including the unfair interleavings. So, for example, there are behaviours of $f \otimes g$ in which g does no actions at all. This is the point at which the sets $T_{f \otimes g}$ comes into play: it is defined to contain just the fair interleavings of traces from T_f and T_g. This amounts to assuming the existence of a fair scheduler; we do not, however, say anything about how it works, thus treating fairness at the level of specification rather than implementation. This is why **FProc** cannot be constructed from a specification structure over **ASProc**: if S is a specification structure over $\mathbb{C}$, the morphisms of $\mathbb{C}_S$ are a selection of the morphisms of $\mathbb{C}$, but do not have any additional structure.

A specification structure D for deadlock-freedom can be defined over this category, and finally $\mathbf{FProc}_D$ is the desired category of deadlock-free processes.

In the case of the scheduler, the cells are fair to start with, and we can assume that $\otimes$ works fairly. The categorical calculations justifying the construction of the scheduler from the cells require I to exist, but we could get round this difficulty by defining the constructions we need directly on typed processes rather than morphisms. The analysis of the scheduler given in this section contains the essential features of a more complete analysis in the category $\mathbf{FProc}_D$. The full definitions of **FProc** and $\mathbf{FProc}_D$ can be found in [22].

Acknowledgements

This research was partially funded by the EU project CONFER (ESPRIT Basic Research Action 6454). It was also supported in part by the U.K. SERC (now EPSRC) under the grant "Foundational Structures for Computer Science", and in the form of a Ph.D. Studentship for Simon Gay. We would like to thank a number of people for helpful comments at various stages of this work: Charles Matthews, Claudio Hermida, Pierre-Louis Curien, Pino Rosolini, Michael Huth, Sebastian Hunt and François Lamarche. Our thanks also to Guy McCusker and Julian Rathke for their careful proof-reading. Radha Jagadeesan's joint work with the first author on Geometry of Interaction and Game Semantics helped pave the way to Interaction Categories. Continuing discussions with Tony Hoare have been illuminating.

References

[1] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] 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.

[3] 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.

[4] S. Abramsky. Interaction Categories II: Asynchronous processes. Paper in preparation, 1994.

[5] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135:5–9, 1994.

[6] S. Abramsky. Interaction Categories I: Synchronous processes. Paper in preparation, 1995.

[7] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction Categories: Illustrative examples, 1993. Abstract of talk given at the CONFER Workshop, University of Edinburgh, UK.

[8] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543 – 574, June 1994.

[9] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 111(1):53–119, 1994.

[10] S. Abramsky and S. J. Vickers. Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, 3:161–227, 1993.

[11] P. Aczel. *Non-well-founded sets*. CSLI Lecture Notes 14. Center for the Study of Language and Information, 1988.

[12] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[13] D. Austry and G. Boudol. Algèbres de processus et synchronisations. *Theoretical Computer Science*, 30:91–131, 1984.

[14] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. Technical Report 647, INRIA, 1986.

[15] 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.

[16] R. L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[17] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, 1993.

[18] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[19] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[20] P. J. Freyd. Algebraically complete categories. In A. Carboni et al., editors, *Proc. 1990 Como Category Theory Conference*, pages 95–104, Berlin, 1991. Springer-Verlag. Lecture Notes in Mathematics Vol. 1488.

[21] P. J. Freyd and A. Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library*. North-Holland, 1990.

[22] S. J. Gay. *Linear Types for Communicating Processes*. PhD thesis, University of London, 1995. Available as `theory/papers/Gay/thesis.ps.gz` via anonymous ftp to `theory.doc.ic.ac.uk`.

[23] 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.

[24] S. J. Gay and R. Nagarajan. A typed calculus of synchronous processes. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1995. To appear.

[25] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[26] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[27] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[28] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[29] M. Hennessy. Axiomatising finite delay operators. Technical report, Department of Computer Science, University of Edinburgh, 1982.

[30] M. Hennessy. Modelling finite delay operators. Technical report, Department of Computer Science, University of Edinburgh, 1983.

[31] C. A. R. Hoare. A model for communicating sequential processes. In R. McKeag and A. Macnaghten, editors, *On the Construction of Programs*. Cambridge University Press, 1980.

[32] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[33] C. A. R. Hoare. Algebra and models. In M. Broy, editor, *Program Design Calculi*, pages 161–195. Springer-Verlag, 1993.

[34] W. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Forma lism*, pages 479–490. Academic Press, 1980.

[35] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[36] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[37] G. M. Kelly and R. H. Street. Review of the elements of 2-categories. In *Lecture Notes in Mathematics*, volume 420, pages 75–103. Springer-Verlag, 1974.

[38] J. Lambek. Multicategories revisited. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 217–240, 1989.

[39] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics Vol. 7. Cambridge University Press, 1986.

[40] K. Larsen and B. Thomsen. A modal process logic. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society, 1988.

[41] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.

[42] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer-Verlag, 1976.

[43] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[44] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[45] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–69. North Holland, 1963.

[46] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[47] R. Milner. A finite delay operator in synchronous ccs. Technical report, Department of Computer Science, University of Edinburgh, 1982.

[48] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[49] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[50] 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.

[51] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[52] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.

[53] L. C. Paulson. *Logic and Computation : Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[54] D. S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, November 1970.

[55] D. Scott. Domains for denotational semantics. In M. Nielson and E. M. Schmidt, editors, *Automata, Languages and Programming: Proceedings 1982*. Springer-Verlag, Berlin, 1982. Lecture Notes in Computer Science **140**.

[56] S. Thompson. *Type Theory and Functional Programming*. Addison Wesley, 1991.

[57] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Vrije Universiteit te Amsterdam, 1990.

[58] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1995. To appear.