In Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, 1995.

A Typed Calculus of Synchronous Processes

Simon Gay
<sjg3@doc.ic.ac.uk>

Rajagopal Nagarajan
<rn4@doc.ic.ac.uk>

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, U.K.

Abstract

We propose a typed calculus of synchronous processes based on the structure of interaction categories. Our aim has been to develop a calculus for concurrency that is canonical in the sense that the typed λ-calculus is canonical for functional computation. We show strong connections between syntax, logic and semantics, analogous to the familiar correspondence between the typed λ-calculus, intuitionistic logic and cartesian closed categories.

1 Introduction

Types are fundamental to the study of functional computation, for both theoretical and practical reasons. On the foundational side there are elegant connections between the typed λ-calculus, intuitionistic logic and cartesian closed categories, leading to the Propositions as Types paradigm [14] and the development of categorical logic [9, 17]. From a practical point of view, compile-time type reconstruction is a boon to the programmer in languages such as Standard ML and Haskell.

Turning to concurrency, the situation is much less satisfactory. There is no generally accepted foundation for typed concurrent programming, and even a canonical untyped calculus has been slow to emerge. While the difficulties of constructing a type system are much greater in the concurrent than in the sequential case, so too are the potential benefits: there is a wide range of concurrent program properties which one might hope to specify as types and verify by type-checking. Examples are deadlock-freedom, liveness and fairness.

Abramsky's recent work on interaction categories [1–4] has established a semantic foundation for typed concurrency with the capability of constructing types which specify complex properties. It extends the Propositions as Types paradigm to concurrency, with classical linear logic [13] forming the core of the type system; the use of a classical linear logic sequent as a process interface description is the key idea of the Proofs as Processes interpretation [5]. Interaction categories have *types* (*specifications*) as objects, *processes* as morphisms and *interaction* as composition. When types are viewed as specifications, the typing rule for categorical composition becomes a compositional proof rule for the specified property; a striking application of this idea is the development of a type system for deadlock-freedom [3, 12].

We use these ideas as the basis for a typed calculus of synchronous processes. The ultimate goal of the research begun in this paper is to develop a calculus for typed concurrency which may be considered canonical in the sense that the typed λ-calculus is canonical for functional computation. In addition, there should be a close connection with logic and semantics via the Curry-Howard isomorphism and a categorical logic correspondence. The calculus in its present form cannot be considered canonical, and indeed it is not as yet clear what canonicity means for concurrency. However, it represents a successful first step and the entire theoretical development is in the same spirit as that of the typed λ-calculus.

The syntax of our calculus is inspired by those of existing process calculi, especially SCCS [19], and also by linear realisability algebras [6]. The types are based on classical linear logic, with some extensions relating to time. The calculus has an operational semantics in the usual style, which yields a notion of *typed* bisimulation. Instead of a Subject Reduction theorem stating that types are unchanged by transitions, there are two results: *Dynamic Subject Reduction*, which states that transitions cause types to evolve in predictable ways,

and *Static Subject Reduction*, which states that an approximation to the type of a term (its number of ports) is preserved.

We also define a categorical semantics which interprets a typed term as a morphism in a category with suitable abstract structure. Appropriate categories are *synchronous interaction categories* [12]; one such is $\mathcal{SP}roc$ [3] (see also the Appendix). Correctness and Full Abstraction results relate the categorical semantics in $\mathcal{SP}roc$ to the operational semantics.

Finally, we indicate how the standard results concerning the construction of initial models of typed λ-calculi as syntactic categories can be transferred to this concurrent situation.

2 Syntax

A *signature* $\mathcal{S}g$ for a process calculus is specified by the following data.

- A collection of *ground types*. The collection of *types* is then defined by the grammar

$$\alpha ::= \gamma \mid \alpha \otimes \alpha \mid \alpha \parr \alpha \mid \alpha^{\perp} \mid \circ\alpha$$

  in which γ is any ground type.

- A collection of *ground actions*. The collection of *actions* is then defined by the grammar

$$\pi ::= \sigma \mid * \mid (\pi, \pi)$$

  in which σ is any ground action.

- A collection of *ground prefixes*, each of which consists of a ground action and a pair of ground types, written $\mathsf{Pre}\ \sigma : \gamma \to \gamma'$. These are subject to the restriction that if $\mathsf{Pre}\ \sigma : \gamma \to \gamma'$ and $\mathsf{Pre}\ \sigma : \gamma \to \gamma''$ are ground prefixes then $\gamma' = \gamma''$.

The *prefixes* generated by $\mathcal{S}g$ are the ground prefixes together with the expressions $\mathsf{Pre}\ \pi : \alpha \to \beta$ which can be derived from the ground prefixes by means of the rules in Figure 1. The action $*$ represents idling and corresponds to the type constructor $\circ$. It is always available as a prefix.

The next step is to define the *raw processes*, which are untyped process terms:

$$\begin{array}{lll} P & ::= & \pi : P \mid \parr_z^{x,y}(P) \mid P \otimes_z^{x,y} P \mid P \mathbin{;} P \mid \\ & & P\backslash_{x,y} \mid I_{u,v} \mid P + P \mid \mathsf{nil}(x_1, \ldots, x_n) \mid \\ & & \mathsf{fix}_u(X = E(X)). \end{array}$$

We assume a countable collection $\{x, y, \ldots\}$ of *variables* and, using the standard notions of free and bound variables (in the above grammar, x and y are always bound and other variables are free), work with raw processes up to α-equivalence.

The *proved processes* generated by $\mathcal{S}g$ are the expressions $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n$ which can be derived using the rules in Figure 2. The form of a proved process is exactly as specified by the Proofs as Processes interpretation. In such an expression P is a raw process, the α_i are types, and the x_i are variables. The order of the $x_i : \alpha_i$ is unimportant. The expression should be read "the process P has interface $x_1 : \alpha_n, \ldots, x_n : \alpha_n$" and should not be confused with the usual notation for intuitionistic sequents, in which an expression $\Gamma \vdash A$ means that A can be proved from the hypotheses Γ. Each variable corresponds to a port of the process, and each port has a type. Labelling ports in this way means that process constructions are able to refer to particular ports. A port is a place in which actions can happen; so this calculus, unlike many others, makes a clear distinction between ports and actions.

The prefixing rule takes a proved process $P \vdash x : \beta$ and a prefix action π, and forms a process $\pi : P \vdash x : \alpha$ which, as in SCCS, can do a π action and become P. Note that there is a change of type, which is governed by the prefix judgement containing π, i.e. $\mathsf{Pre}\ \pi : \alpha \to \beta$. The reason for the change in type caused by prefixing is that the calculus is intended to be interpreted in an interaction category such as $\mathcal{SP}roc$. Because a type in $\mathcal{SP}roc$ contains a safety specification, i.e. a set of permissible finite traces, if P satisfies the safety specification S then $\pi : P$ satisfies the specification $T \stackrel{\text{def}}{=} \{\pi s \mid s \in S\} \cup \{\varepsilon\}$. In general these specifications are different, and this difference is reflected in the different syntactic types of P and $\pi : P$. The precise connection between the semantic specifications S and T is expressed syntactically by the judgement $\mathsf{Pre}\ \pi : \alpha \to \beta$.

The Prefix rule has as its hypothesis a process with just one port. If a process has more ports, we can repeatedly use the following derivation which combines two ports into one and uses the prefix combination rule to prefix a pair of actions:

$$\dfrac{\dfrac{P \vdash x : \gamma, y : \delta}{\parr_z^{x,y}(P) \vdash z : \gamma \parr \delta} \qquad \dfrac{\mathsf{Pre}\ a : \alpha \to \gamma \quad \mathsf{Pre}\ b : \beta \to \delta}{\mathsf{Pre}\ (a, b) : \alpha \parr \beta \to \gamma \parr \delta}}{(a, b) : \parr_z^{x,y}(P) \vdash z : \alpha \parr \beta}$$

We can also turn a process $\parr_z^{x,y}(P) \vdash z : \gamma \parr \delta$ into $P \vdash x : \gamma, y : \delta$ by cutting it with a suitable process constructed from identity axioms. The difference between $\parr_z^{x,y}(P) \vdash z : \gamma \parr \delta$ and $P \vdash x : \gamma, y : \delta$ lies solely in how the interface is viewed—the process terms will be identified by the categorical semantics.

$$\frac{}{\mathsf{Pre}\ *\ :\ \circ\,\alpha \to \alpha} \qquad \frac{\mathsf{Pre}\ \pi : \alpha \to \beta}{\mathsf{Pre}\ \pi : \alpha^{\perp} \to \beta^{\perp}}$$

$$\frac{\mathsf{Pre}\ \pi : \alpha \to \gamma \quad \mathsf{Pre}\ \pi' : \beta \to \delta}{\mathsf{Pre}\ (\pi, \pi') : \alpha \otimes \beta \to \gamma \otimes \delta} \qquad \frac{\mathsf{Pre}\ \pi : \alpha \to \gamma \quad \mathsf{Pre}\ \pi' : \beta \to \delta}{\mathsf{Pre}\ (\pi, \pi') : \alpha \parr \beta \to \gamma \parr \delta}$$

Figure 1: Prefixes Generated by a Process Signature

Prefixing Rule

$$\frac{P \vdash x : \beta \quad \mathsf{Pre}\ \pi : \alpha \to \beta}{\pi : P \vdash x : \alpha}\ \text{Prefix}$$

Multiplicative Rules

$$\frac{P \vdash \Gamma, x : \alpha, y : \beta}{\parr^{x,y}_{z}(P) \vdash \Gamma, z : \alpha \parr \beta}\ \text{Par} \qquad \frac{P \vdash \Gamma, x : \alpha \quad Q \vdash \Delta, y : \beta}{P \otimes^{x,y}_{z} Q \vdash \Gamma, \Delta, z : \alpha \otimes \beta}\ \text{Tensor}$$

Connection Rules

$$\frac{P \vdash \Gamma, x : \alpha \quad Q \vdash \Delta, x : \alpha^{\perp}}{P \mathbin{;_x} Q \vdash \Gamma, \Delta}\ \text{Cut} \qquad \frac{P \vdash \Gamma, x : \alpha, y : \alpha^{\perp}}{P \backslash_{x,y} \vdash \Gamma}\ \text{Cycle}$$

$$\frac{}{I_{x,y} \vdash x : \alpha^{\perp}, y : \alpha}\ \text{Axiom}$$

Summation Rules

$$\frac{P \vdash \Gamma \quad Q \vdash \Gamma}{P + Q \vdash \Gamma}\ \text{Sum} \qquad \frac{}{\mathsf{nil}_{\alpha_1,\ldots,\alpha_n}(x_1, \ldots, x_n) \vdash x_1 : \alpha_n, \ldots, x_n : \alpha_n}\ \text{Nil}$$

Recursion Rule

$$\frac{\begin{array}{c} X \vdash x : \alpha \\ \vdots \\ E(X) \vdash x : \alpha \end{array}}{\mathsf{fix}_x(X = E(X)) \vdash x : \alpha}$$

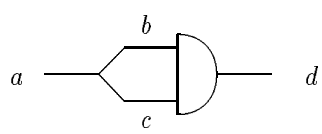
where X is sequential and guarded in E.

Figure 2: Proved Processes Generated by a Process Signature

The Tensor rule gives another way of combining two ports into one. However, this time the ports are taken from two different processes, and the resulting process has an interface formed from the interfaces of the original ones. Combining processes by Tensor is like the synchronous product in SCCS, except that there is no possibility of communication between them. The Cut rule connects processes together in such a way that communication is not only allowed, but required. In SCCS terms, it is like a combination of synchronous product and restriction; this is exactly the interpretation of Cut put forward in Proofs as Processes, and seen in the definition of composition in $\mathcal{S}Proc$. The Cycle rule is like Cut, except that ports of the same process are connected.

The Axiom rule produces a process which acts as a buffer or wire, and is useful for rearranging interfaces. The Sum rule, as in SCCS, allows the construction of the non-deterministic combination of two processes—these processes should have the same interface. The Nil rule allows nil processes to be introduced with any interface; as usual, nil is the unit for $+$. The Recursion rule is presented here in a simplified form which only allows a single variable to be used; in the full calculus [12] any number of mutually recursive definitions are allowed. In both cases, as is standard, we restrict attention to *sequential* and *guarded* recursive definitions [20].

As an illustration of the use of the calculus, consider a simple example of a synchronous fork and an and gate connected together.



It is convenient to introduce a Double Cut rule, which consists of Cut followed by Cycle. $P \,{}_{x,y}\!\cdot\, Q$ denotes a Double Cut on ports x and y between processes P and Q; it has the effect of simultaneously forming connections between the x ports and the y ports.

$$\frac{P \vdash \Gamma, x : \alpha, y : \beta \quad Q \vdash \Delta, x : \alpha^{\perp}, y : \beta^{\perp}}{P \,{}_{x,y}\!\cdot\, Q \vdash \Gamma, \Delta}$$

This construction can be generalised to obtain a Multi Cut rule, which could replace the Cut and Cycle rules in the syntax. However, the present form of the syntax can be more easily adapted to situations in which the Cycle rule is not wanted—this may be the case when types express more subtle properties such as deadlock-freedom. A similar comment justifies distinguishing syntactically between $\otimes$ and $\parr$ even though they will be identified by the categorical semantics.

The above circuit is

$$\frac{\mathsf{fork} \vdash a : \mathbb{B}, b : \mathbb{B}^{\perp}, c : \mathbb{B}^{\perp} \quad \mathsf{and} \vdash b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}^{\perp}}{\mathsf{fork} \,{}_{b,c}\!\cdot\, \mathsf{and} \vdash a : \mathbb{B}, d : \mathbb{B}^{\perp}}$$

where $\mathbb{B}$ is the boolean type $\{\mathsf{t}, \mathsf{f}\}$. Of course, the resulting process is just a wire and has the correct type. Note that the distinction between input and output is made by the types rather than by the actions as in SCCS. The process fork can be defined informally by the following recursive equation:

$$\mathsf{fork} \quad = \quad (\mathsf{t}, \mathsf{t}, \mathsf{t}) : \mathsf{fork} + (\mathsf{f}, \mathsf{f}, \mathsf{f}) : \mathsf{fork}.$$

The derivation of fork begins with a process

$$X \vdash u : \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}$$

and prefixes

$$\begin{array}{ll} (\mathsf{t}, \mathsf{t}, \mathsf{t}) : & \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp} \rightarrow \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp} \\ (\mathsf{f}, \mathsf{f}, \mathsf{f}) : & \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp} \rightarrow \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}. \end{array}$$

The prefixes can be constructed with the prefix combination rules, if the correct ground prefixes are specified. We allow the set of all traces as permissible traces (effectively ignoring safety specifications) for this example, so the prefixes do not involve a type change. We can then construct the process

$$(\mathsf{t}, \mathsf{t}, \mathsf{t}) : X + (\mathsf{f}, \mathsf{f}, \mathsf{f}) : X \vdash u : \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}$$

and use the Recursion rule to build

$$\mathsf{fix}_u(X = (\mathsf{t}, \mathsf{t}, \mathsf{t}) : X + (\mathsf{f}, \mathsf{f}, \mathsf{f}) : X) \vdash u : \mathbb{B} \parr \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}.$$

This fix expression is what we call fork. The and process can be constructed similarly.

3 Operational Semantics

The operational semantics of the typed process calculus is defined by the transition rules in Figure 3, in which $\tilde{a}$ stands for a tuple $(a_1, \ldots, a_n)$ of actions. Transitions are defined on proved terms with explicit types, because the type of a proved term is not necessarily unique—for example, if Pre $a : \alpha \rightarrow \beta$ and Pre $a : \gamma \rightarrow \beta$ then $a : \mathsf{nil}_{\beta}(x) \vdash x : \alpha$ and $a : \mathsf{nil}_{\beta}(x) \vdash x : \gamma$ are both proved terms. The main points to note are that in the Cut rule two processes communicate by performing the same action in the ports which have been connected together; in the Tensor rule actions from two processes are combined; and in the Par rule the actions of a single process are regrouped. The Cycle rule is similar to Cut, except that the matching actions both come from the

Prefix

$$\frac{\mathsf{Pre}\ \pi : \alpha \to \beta}{\pi : P \vdash x : \alpha \xrightarrow{\pi} P \vdash x : \beta}$$

Par

$$\frac{P \vdash \Gamma, x : \alpha, y : \beta \xrightarrow{(\tilde{a},a,b)} P' \vdash \Gamma', x : \alpha', y : \beta'}{\parr_z^{x,y}(P) \vdash \Gamma, z : \alpha \parr \beta \xrightarrow{(\tilde{a},(a,b))} \parr_z^{x,y}(P') \vdash \Gamma', z : \alpha' \parr \beta'}$$

Tensor

$$\frac{P \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P' \vdash \Gamma', x : \alpha' \qquad Q \vdash \Delta, y : \beta \xrightarrow{(\tilde{b},b)} Q' \vdash \Delta', y : \beta'}{P \otimes_z^{x,y} Q \vdash \Gamma, \Delta, z : \alpha \otimes \beta \xrightarrow{(\tilde{a},\tilde{b},(a,b))} P' \otimes_z^{x,y} Q' \vdash \Gamma', \Delta', z : \alpha' \otimes \beta'}$$

Cut

$$\frac{P \vdash \Gamma, x : \alpha \xrightarrow{(\tilde{a},a)} P' \vdash \Gamma', x : \beta \qquad Q \vdash \Delta, x : \alpha^{\perp} \xrightarrow{(\tilde{b},a)} Q' \vdash \Delta', x : \beta^{\perp}}{P \mid_x Q \vdash \Gamma, \Delta \xrightarrow{(\tilde{a},\tilde{b})} P' \mid_x Q' \vdash \Gamma', \Delta'}$$

Cycle

$$\frac{P \vdash \Gamma, x : \alpha, y : \alpha^{\perp} \xrightarrow{(\tilde{a},a,a)} P' \vdash \Gamma', x : \beta, y : \beta^{\perp}}{P \backslash_{x,y} \vdash \Gamma \xrightarrow{\tilde{a}} P' \backslash_{x,y} \vdash \Gamma'}$$

Axiom

$$\frac{\mathsf{Pre}\ \pi : \alpha \to \beta}{I_{x,y} \vdash x : \alpha^{\perp}, y : \alpha \xrightarrow{(\pi,\pi)} I_{x,y} \vdash x : \beta^{\perp}, y : \beta}$$

Summation

$$\frac{P \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'}{P + Q \vdash \Gamma \xrightarrow{\tilde{a}} P' \vdash \Gamma'} \qquad \frac{Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'}{P + Q \vdash \Gamma \xrightarrow{\tilde{a}} Q' \vdash \Gamma'}$$

Recursion

$$\frac{E[\mathsf{fix}_x(X = E(X))/X] \vdash x : \alpha \xrightarrow{\pi} P \vdash x : \beta}{\mathsf{fix}_x(X = E(X)) \vdash x : \alpha \xrightarrow{\pi} P \vdash x : \beta}$$

Figure 3: Operational Semantics of the Typed Process Calculus

same process: Cycle is to Par as Cut is to Tensor. The Prefix rule is the base case.

A typed programming language usually has a Subject Reduction theorem, which states that transitions or reductions in the operational semantics do not alter the types of terms. For our typed process calculus, the situation is different as the transition rule for Prefix has the potential to change types. However, since the changes in type as a process makes transitions are not arbitrary but are controlled by the Pre expressions in the signature, it is possible to prove the following *dynamic* Subject Reduction theorem, by a straightforward induction over the transition rules.

Theorem 1 (Dynamic Subject Reduction) *If* $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n \xrightarrow{\pi} Q \vdash y_1 : \beta_1, \ldots, y_m : \beta_m$ *then* $m = n$, $\pi = (a_1, \ldots, a_n)$, *and for each* i, $x_i = y_i$ *and* Pre $a_i : \alpha_i \to \beta_i$ *is derivable.*

This leads to a general observation about the rôle of Subject Reduction theorems in the theory of typed programming languages. The point of having types is that well-typed programs have some correctness property; if this property is preserved by reductions, then programs remain correct during evaluation. But because correctness follows from typability rather than satisfaction of any particular type, the usual Subject Reduction theorems are stronger than is necessary to deduce that correctness is preserved by evaluation: it is only necessary to know that a well-typed program still has *some* type after a reduction step.

In general types of process terms are changed by transitions, but there is an aspect of the type of a term which stays the same—essentially the number of ports and the connectives with which they are combined. Thus there is also a *static* Subject Reduction result which makes this formal. Its statement requires a function θ which maps types into formal constructions involving a single ground type $\maltese$, as follows (γ is a ground type of $\mathcal{S}g$).

$$\begin{aligned}
\theta(\gamma) &\stackrel{\text{def}}{=} \maltese \\
\theta(\alpha^{\perp}) &\stackrel{\text{def}}{=} \theta(\alpha)^{\perp} \\
\theta(\circ\,\alpha) &\stackrel{\text{def}}{=} \theta(\alpha) \\
\theta(\alpha \otimes \beta) &\stackrel{\text{def}}{=} \theta(\alpha) \otimes \theta(\beta) \\
\theta(\alpha \parr \beta) &\stackrel{\text{def}}{=} \theta(\alpha) \parr \theta(\beta).
\end{aligned}$$

Lemma 2 *If* Pre $\pi : \alpha \to \beta$ *is derivable, then* $\theta(\alpha) = \theta(\beta)$.

Theorem 3 (Static Subject Reduction) *If* $P \vdash x_1 : \alpha_1, \ldots, x_n : \alpha_n \xrightarrow{(a_1,\ldots,a_n)} Q \vdash x_1 : \beta_1, \ldots, x_n : \beta_n$ *then for each* i, $\theta(\beta_i) = \theta(\alpha_i)$.

Proof: By Theorem 1, Pre $a_i : \alpha_i \to \beta_i$ is derivable for each i. By Lemma 2, $\theta(\alpha_i) = \theta(\beta_i)$ for each i. $\square$

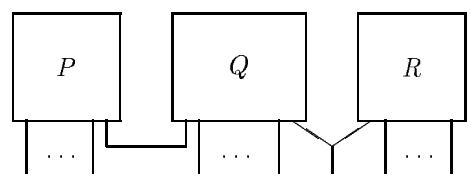
Once the operational semantics of the calculus has been defined, it is natural to use strong bisimulation as the notion of equivalence [20]. In fact we use a variation, *strong typed bisimulation*, which requires equivalent processes to have the same type. For the purposes of this paper, we simply call it strong bisimulation and keep the original notation.

Proposition 4 *Strong bisimulation is a congruence.*

The proof of the above proposition is straightforward; it is essentially the same as for SCCS [19]. It also follows from the definition of the operational semantics that substitutions do not change strong bisimulation classes.

Proposition 5 *If* $P \vdash \Gamma, x : \alpha$ *and* y *is not free in* P *then* $P \sim P[y/x]$.

The syntax of the calculus allows certain process configurations to be described in several different ways. For example, the following diagram represents the result of taking three processes, forming a connection between two of them and then using $\otimes$ to combine a port of the resulting process with a port of a third process:



In the syntax, this can be described by either of the terms $(P \mathbin{;}_x Q) \otimes_w^{u,v} R$ and $P \mathbin{;}_x (Q \otimes_w^{u,v} R)$, if the ports are suitably named. As might be expected, these processes can be shown to be bisimilar. Syntactic distinctions of this form are inevitably introduced whenever a textual notation is used to represent two- or three-dimensional structures; exactly the same problem occurs in linear logic when sequent proofs are used instead of proof nets [13]. There are many more instances of bisimulation arising for similar reasons. Another source of bisimilar processes is cut-elimination. Process configurations which would be related by cut-elimination if viewed as proof nets are bisimilar, for example $P \vdash x : \alpha$ and $P \mathbin{;}_x I_{x,y} \vdash y : \alpha$. Cut-elimination corresponds to β-reduction under the Curry-Howard isomorphism, but process calculus transitions are orthogonal to β-reductions; semantically, β-reduction is absorbed into equality. For a full list of such equivalences, see [12].

4 Categorical Semantics

The typed process calculus can be given a semantics in a suitably structured category. Let $\mathbb{C}$ be a com-

pact closed category (a $*$-autonomous category [8] in which $\otimes$ and $\parr$ coincide) with countable biproducts and a functor $!$ which interprets the exponential of linear logic (i.e. $!$ should be a comonad and each $!A$ should have a cocommutative comonoid structure [24]). Additionally, let $\mathbb{C}$ have a strict monoidal endofunctor $\circ$, and write $\mathsf{monunit} : I \to \circ I$ and $\mathsf{mon}_{A,B} : (\circ A) \otimes (\circ B) \to \circ(A \otimes B)$ for the associated isomorphisms.

An endofunctor F has the *unique fixed point property* (UFPP) [3] if for every $f : A \to FA$ and $g : FB \to B$ there is a unique $h : A \to B$ such that

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

commutes. This is a categorical formulation of the statement that certain guarded recursive equations have unique solutions; for suitable F, the equation $h = f \,;\, Fh \,;\, g$ defines h uniquely by guarded recursion. The final requirement on $\mathbb{C}$ is that the functor $X \mapsto I \oplus (\bigoplus_{n>0} \circ^n X)$ should have the UFPP. These conditions on $\mathbb{C}$ are a simplification of a general axiomatisation of synchronous interaction categories [12].

In $\mathbb{C}$ the biproducts yield a commutative monoid structure $(+, \mathsf{nil})$ on each homset [18], $\otimes$ distributes over $+$ and $(-)^\perp$ preserves $+$. There is also a partial order $\leqslant$ on each homset, defined by

$$f \leqslant g \overset{\text{def}}{\Leftrightarrow} \exists h. f + h = g.$$

This order is preserved by $\otimes$ and $(-)^\perp$.

A *structure* in $\mathbb{C}$ for a process signature $\mathcal{S}g$ is specified by the following data.

- For each ground type γ of $\mathcal{S}g$, an object $[\![\gamma]\!]$ of $\mathbb{C}$. The function $[\![\cdot]\!]$ is then extended inductively to all types by

$$\begin{aligned} [\![\alpha^\perp]\!] &\overset{\text{def}}{=} [\![\alpha]\!]^\perp \\ [\![\alpha \otimes \beta]\!] &\overset{\text{def}}{=} [\![\alpha]\!] \otimes [\![\beta]\!] \\ [\![\alpha \parr \beta]\!] &\overset{\text{def}}{=} [\![\alpha]\!] \parr [\![\beta]\!] \\ [\![\circ\, \alpha]\!] &\overset{\text{def}}{=} \circ [\![\alpha]\!]. \end{aligned}$$

  and to lists of named ports by

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] \overset{\text{def}}{=} [\![A_1]\!] \parr \ldots \parr [\![A_n]\!].$$

- For each ground prefix $\mathsf{Pre}\ \sigma : \gamma \to \gamma'$, a pair of morphisms

$$[\![\mathsf{Pre}\ \sigma : \gamma \to \gamma']\!] : \circ[\![\gamma']\!] \to [\![\gamma]\!]$$

$$[\![\mathsf{Pre}\ \sigma : \gamma \to \gamma']\!]' : [\![\gamma]\!] \to \circ[\![\gamma']\!]$$

  such that

$$[\![\mathsf{Pre}\ \sigma : \gamma \to \gamma']\!] \,;\, [\![\mathsf{Pre}\ \sigma : \gamma \to \gamma']\!]' = \mathsf{id}_{\circ[\![\gamma']\!]}$$

$$\mathsf{id}_{[\![\gamma]\!]} = \sum_{\mathsf{Pre}\ \pi : \gamma \to \gamma'} [\![\mathsf{Pre}\ \pi : \gamma \to \gamma']\!]' \,;\, [\![\mathsf{Pre}\ \pi : \gamma \to \gamma']\!].$$

The function $[\![\cdot]\!]$ is extended to all prefix judgements in a natural way, for example

$$\begin{aligned} [\![\mathsf{Pre}\ * : \circ\, \alpha \to \alpha]\!] &\overset{\text{def}}{=} \mathsf{id}_{\circ[\![\alpha]\!]} \\ [\![\mathsf{Pre}\ * : \circ\, \alpha \to \alpha]\!]' &\overset{\text{def}}{=} \mathsf{id}_{\circ[\![\alpha]\!]} \\ [\![\mathsf{Pre}\ (\pi, \pi') : \alpha \otimes \alpha' \to \beta \otimes \beta']\!] &\overset{\text{def}}{=} \\ \mathsf{mon}^{-1} \,;\, ([\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!] &\otimes [\![\mathsf{Pre}\ \pi' : \alpha' \to \beta']\!]) \\ [\![\mathsf{Pre}\ (\pi, \pi') : \alpha \otimes \alpha' \to \beta \otimes \beta']\!]' &\overset{\text{def}}{=} \\ ([\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!]' &\otimes [\![\mathsf{Pre}\ \pi' : \alpha' \to \beta']\!]') \,;\, \mathsf{mon}. \end{aligned}$$

Proposition 6 *If* $\mathsf{Pre}\ \pi : \alpha \to \beta$ *is derivable, then* $[\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!] \,;\, [\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!]' = \mathsf{id}_{\circ[\![\beta]\!]}$ *and* $[\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!]' \,;\, [\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!] \leqslant \mathsf{id}_{[\![\alpha]\!]}$.

Proof: By induction on the derivation of $\mathsf{Pre}\ \pi : \alpha \to \beta$, using the fact that $\otimes$, $\parr$ and $(-)^\perp$ preserve $\leqslant$. $\square$

For each proved process $P \vdash \Gamma$ generated by $\mathcal{S}g$, there is a morphism $[\![P \vdash \Gamma]\!] : I \to [\![\Gamma]\!]$ in $\mathbb{C}$, defined by induction on the derivation of $P \vdash \Gamma$. Note that for any A, B, C and D, $\mathsf{regroup} : (A \parr B) \otimes (C \parr D) \to A \parr ((B \otimes C) \parr D)$ and $\mathsf{unitl} : I \otimes A \to A$ are canonical morphisms in any $*$-autonomous category; Ap and Λ are *evaluation* and *currying* respectively; and iso is used to denote any canonical isomorphism.

Axiom $[\![I_{x,y} \vdash x : \alpha^\perp, y : \alpha]\!] \overset{\text{def}}{=} \Lambda(\mathsf{unitl}_{[\![\alpha]\!]})$.

Cut $[\![P \underset{x}{\cdot} Q \vdash \Gamma, \Delta]\!] \overset{\text{def}}{=}$
$\mathsf{iso} \,;\, ([\![P]\!] \otimes [\![Q]\!]) \,;\, \mathsf{regroup} \,;\, (\mathsf{id} \parr \mathsf{Ap} \parr \mathsf{id}) \,;\, \mathsf{iso}$.

Cycle $[\![P \backslash_{x,y} \vdash \Gamma]\!] \overset{\text{def}}{=} [\![P]\!] \,;\, \mathsf{iso} \,;\, (\mathsf{id} \parr \mathsf{Ap}) \,;\, \mathsf{iso}$.

Tensor $[\![P \otimes_z^{x,y} Q \vdash \Gamma, z : \alpha \otimes \beta, \Delta]\!] \overset{\text{def}}{=}$
$\mathsf{iso} \,;\, ([\![P]\!] \otimes [\![Q]\!]) \,;\, \mathsf{regroup} \,;\, \mathsf{iso}$.

Par $[\![\parr_z^{x,y}(P) \vdash \Gamma, z : \alpha \parr \beta]\!] \overset{\text{def}}{=} [\![P \vdash \Gamma, x : \alpha, y : \beta]\!]$.

Summation $[\![P + Q \vdash \Gamma]\!] \overset{\text{def}}{=} [\![P \vdash \Gamma]\!] + [\![Q \vdash \Gamma]\!]$.

Prefix $[\![\pi : P \vdash x : \alpha]\!] \overset{\text{def}}{=}$
$\mathsf{monunit} \,;\, \circ[\![P]\!] \,;\, [\![\mathsf{Pre}\ \pi : \alpha \to \beta]\!]$.

Recursion Consider $\mathsf{fix}_u(X = E(X)) \vdash u : \alpha$. $E(X) = E_0 + E_1(X) + E_2(X) + \cdots$ where $E_0 \vdash u : \alpha$ is a constant process and for $n > 0$, $E_n(X)$ is a (possibly empty) sum of terms in which X is guarded by n prefixes. For each term $\pi_1 : \ldots : \pi_n : X$ there is a morphism $f_{\pi_1 \ldots \pi_n} : \circ^n [\![\alpha]\!] \to [\![\alpha]\!]$ constructed from the prefixes. Now let $g_0 = [\![E_0]\!]$, let g_n be the sum of the morphisms f_n (nil if $E_n(X) = \mathsf{nil}$) and form $g : I \oplus \circ[\![\alpha]\!] \oplus \circ^2[\![\alpha]\!] \oplus \cdots \to [\![\alpha]\!]$ using the coproduct

property of $\oplus$. Also, there is a canonical morphism $h : I \to I \oplus \circ I \oplus \cdots$ constructed from $\mathsf{monunit}$ and $\circ$. Finally, $[\![\mathsf{fix}_u(X = E(X))]\!]$ is the morphism $I \to [\![\alpha]\!]$ defined by applying the UFPP of $X \mapsto I \oplus (\bigoplus_{n>0} \circ^n X)$ to h and g. The structure of ! is used to interpret the full version of the recursion rule [12], but lack of space prevents us from describing it here. This is the only point at which ! becomes important—it does not appear in the syntax.

5 Semantics in $\mathcal{SP}roc$

The typed process calculus is intended to have an interpretation in $\mathcal{SP}roc$. There is a structure in $\mathcal{SP}roc$ for any process signature $\mathcal{S}g$, defined as follows.

A process signature $\mathcal{S}g$ defines a labelled transition system whose states are the ground types of $\mathcal{S}g$, whose labels are the actions appearing in the ground prefixes, and with $\gamma \xrightarrow{\pi} \gamma' \iff \mathsf{Pre}\ \pi : \gamma \to \gamma'$. If this labelled transition system is considered as a directed graph with labelled edges, there are a number of connected components; for each component c, there is a set Σ_c of actions consisting of the labels which occur in c. For example, if $\mathcal{S}g$ has ground types A, B, C and ground prefixes

$$\begin{array}{l} \mathsf{Pre}\ a : A \to B \\ \mathsf{Pre}\ b : A \to B \\ \mathsf{Pre}\ c : B \to C \\ \mathsf{Pre}\ d : C \to A \end{array}$$

then the graph is

$$\begin{tikzcd} A \arrow[rr, shift left, "a"] \arrow[rr, shift right, "b"'] & & B \arrow[dl, "c"] \\ & C \arrow[ul, "d"] & \end{tikzcd}$$

In this case all the types are in the same connected component. The set of all labelled paths in this graph, starting from some type, defines a safety specification which any process of that type satisfies; this follows from the Dynamic Subject Reduction theorem. In practice, one starts with a desired safety specification, represents it as the set of paths in a graph, and uses the graph to define suitable ground prefixes.

In general, given a ground type γ, let $c(\gamma)$ be the connected component containing γ. The object $[\![\gamma]\!]$ is defined by

$$\begin{array}{lcl} \Sigma_{[\![\gamma]\!]} & \stackrel{\text{def}}{=} & \Sigma_{c(\gamma)} \\ S_{[\![\gamma]\!]} & \stackrel{\text{def}}{=} & \{s \mid \exists \gamma'.\gamma \xrightarrow{s}{}^* \gamma'\}. \end{array}$$

Given a prefix judgement $\mathsf{Pre}\ \pi : \gamma \to \gamma'$, $[\![\mathsf{Pre}\ \pi : \gamma \to \gamma']\!] : \circ[\![\gamma']\!] \to [\![\gamma]\!]$ is defined as $(*, \pi) : \mathsf{id}_{[\![\gamma']\!]}$ and $[\![\mathsf{Pre}\ \pi : \gamma \to \gamma']\!]' : \circ[\![\gamma']\!] \to [\![\gamma]\!]$ is defined as $(\pi, *) : \mathsf{id}_{[\![\gamma']\!]}$. The conditions which these morphisms must satisfy are easily checked.

For each proved process $P \vdash \Gamma$, there is a synchronisation tree $\mathsf{tree}(P \vdash \Gamma)$ defined by the operational semantics. The key correctness result states that the $\mathcal{SP}roc$ semantics produces the same tree, up to a trivial relabelling.

Proposition 7 (Correctness) *If $[\![\cdot]\!]$ is the semantics in $\mathcal{SP}roc$, then for every proved process $P \vdash \Gamma$, $[\![P \vdash \Gamma]\!] = \mathsf{tree}(P \vdash \Gamma)[(a_1, \ldots, a_n) \mapsto (*, a_1, \ldots, a_n)]$.*

Proof: By induction on the derivation of $P \vdash \Gamma$. In every case, the transition rules for $P \vdash \Gamma$ are the same as those in the $\mathcal{SP}roc$ definition of $[\![P \vdash \Gamma]\!]$. □

Corollary 8 *If $P \vdash \Gamma$ is a proved process then $\mathsf{traces}(\mathsf{tree}(P \vdash \Gamma)) \subseteq S_{[\![\Gamma]\!]}$.*

The next result links denotational equality to strong bisimulation.

Theorem 9 (Full Abstraction) *If $[\![\cdot]\!]$ is the semantics in $\mathcal{SP}roc$, then for all proved processes $P \vdash \Gamma$ and $Q \vdash \Gamma$,*

$$P \sim Q \vdash \Gamma \iff [\![P \vdash \Gamma]\!] = [\![Q \vdash \Gamma]\!].$$

Proof:

$$\begin{array}{lcl} P \sim Q \vdash \Gamma & \iff & \mathsf{tree}(P \vdash \Gamma) \sim \mathsf{tree}(Q \vdash \Gamma) \\ & \iff & [\![P \vdash \Gamma]\!] \sim [\![Q \vdash \Gamma]\!], \\ & & \text{by Proposition 7} \\ & \iff & [\![P \vdash \Gamma]\!] = [\![Q \vdash \Gamma]\!], \\ & & \text{as } = \text{ and } \sim \text{ coincide} \\ & & \text{in } \mathcal{SP}roc. \end{array}$$ □

6 Categorical Logic

A significant aspect of the theory of the typed λ-calculus is the close connection between syntax and semantics as formalised by the construction of syntactic categories and the proof of various correspondence theorems [9, 17]. Some progress has been made towards a similar connection for interaction categories. The idea is to present a *process theory* as a process signature together with a collection of *axioms*, which are expressions of the form $P = Q \vdash \Gamma$ with $P \vdash \Gamma$ and $Q \vdash \Gamma$ proved processes. There is then a collection of rules for deriving more equations, which are the *theorems* of the theory. The process calculus of this paper could be presented in this style, in which case the rules for

deriving instances of bisimulation would become rules for generating theorems. There is a notion of a model of a process theory in a suitable category (some form of interaction category), and in such a model provably equal processes have equal interpretations.

Once a process theory has been set up, it is possible to construct a category in which an object is a type of the theory, and a morphism from α to β is an equivalence class of proved processes $P \vdash x : \alpha^{\perp}, y : \beta$ under provable equality. There is a canonical model of the process theory in this category, and this model satisfies a universal property: a model of the theory in any other category factors as the canonical model followed by an interaction category functor. Conversely, given any interaction category a process theory can be extracted from it, and the operations of moving from a theory to a category and vice versa are inverse to each other. The theory has been worked out for a version of the typed process calculus in which prefixing is handled slightly differently and no commitment is made to synchrony or asynchrony [10].

7 Related Work

Several recent investigations into the foundations of concurrency have included types, with varying amounts of emphasis. Milner's action structures [21] are based on monoidal categories and may have some connection with our linear types; Honda [16, 25] has proposed a typed calculus similar to the π-calculus; Ferrari and Montanari [11], use an approach in which types are changed by transitions; and Prasad [23] has described a calculus with a type system based on Girard's LU. The key novelty of our approach is that it combines the Propositions as Types view with a categorical semantics, and aims to exploit the interaction category view of types as specifications.

8 Conclusions and Future Work

We have defined a typed process calculus based on the structure of interaction categories. In addition to an operational semantics we have defined a categorical semantics, which can be used in any category satisfying certain axioms. One such category is $\mathcal{SP}roc$, and in this case we obtain a Full Abstraction theorem relating the operational and categorical semantics.

Our calculus extends the original ideas of the Proofs as Processes interpretation, by adapting the syntax to one in which prefixing and dynamic behaviour can be defined. The formulation of a syntax for processes, with a type system based on linear logic and a categorical semantics, represents a successful transfer of the Curry-Howard isomorphism and Categorical Logic correspondence to concurrency.

The theory of interaction categories has been applied to several case studies, such as the cyclic scheduler [20] and the dining philosophers [15], in order to demonstrate the type-theoretic approach to specification and verification. This work is reported elsewhere [4, 12]. Up to now, however, we have always constructed the systems being studied by describing their components as labelled transition systems and combining them by direct application of the categorical combinators. In the present paper, we have developed a formal syntax for the construction of typed concurrent systems. This syntax provides a suitable basis for future work on automated type-checking of concurrent programs.

There are several reasons for the decision to study a synchronous calculus. One is that synchronous interaction categories have more structure and are easier to define and work with. From the syntactic point of view, an asynchronous calculus is likely to be slightly more complex, as more prefix combination rules would be needed. Nevertheless it should soon be possible to formulate an asynchronous version. Meanwhile, an extension of the synchronous calculus using the delay monads δ and Δ of $\mathcal{SP}roc$ (corresponding to Milner's construction of CCS from SCCS [20]) has been investigated [12], and this allows asynchronous processes to be constructed within the synchronous framework. The syntax of the calculus has already been through several stages of refinement, and there may be more to come: designing a syntax which is both easy to use and sufficiently powerful is a difficult task.

In our calculus, the terms (proved processes) are explicitly typed. We would like to investigate the use of implicit typing and develop a type inference/reconstruction scheme. We would also like to extend our typed calculus to incorporate *mobile* processes [22]. Finally, it is desirable to be able to adapt the typed calculus in order to define processes in categories in which types are more than just safety specifications—for example, the deadlock-free category $\mathcal{SP}roc_D$ [3].

Acknowledgements

We would like to thank Samson Abramsky for his influential ideas, which form the basis of this work. Martín Abadi, Samson Abramsky, Michel Chaudron, Michael Huth, Radha Jagadeesan and Guy McCusker made valuable comments on a draft of this paper. Simon Gay was funded by an EPSRC Studentship and EU project COORDINATION (ESPRIT BRA 9102), and Raja Nagarajan by EU project CONFER (ESPRIT BRA 6454); we gratefully acknowledge their support.

References

[1] 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.

[2] 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.

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

[4] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*, NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, 1995. To appear. Also avilable as `marktoberdorf.ps.gz` in the directory theory/papers/Abramsky via anonymous ftp to `theory.doc.ic.ac.uk`.

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

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

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

[8] M. Barr. $*$-*Autonomous Categories*, volume 752 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

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

[10] R. L. Crole, S. J. Gay, and R. Nagarajan. An internal language for interaction categories. In C. L. Hankin, I. C. Mackie, and R. Nagarajan, editors, *Theory and Formal Methods 1994: Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods.*, 1994. To appear.

[11] G. Ferrari and U. Montanari. Typed additive concurrency. Submitted for publication., 1994.

[12] S. J. Gay. *Linear Types for Communicating Processes*. PhD thesis, University of London, 1995.

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

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

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

[16] K. Honda. Types for dyadic interaction. In *CONCUR 93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

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

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

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

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

[21] R. Milner. Action structures. Technical Report 92-249, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1993.

[22] 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.

[23] S. Prasad. Towards a formulae-as-types view of communicating applicative programs (extended summary). Technical report, ECRC, 1994.

[24] R. Seely. Linear logic, $*$-autonomous categories and cofree coalgebras. In *Contemporary Mathematics*, 1987.

[25] K. Takuechi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

Appendix

The Interaction Category $\mathcal{SP}roc$

We now present a very brief introduction to $\mathcal{SP}roc$, the category of synchronous processes. Full definitions can be found elsewhere [3, 12].

An object of $\mathcal{SP}roc$ is a pair $A = (\Sigma_A, S_A)$ where Σ_A is an alphabet of actions and $S_A \subseteq^{nepref} \Sigma_A^*$ is a safety specification, i.e. a non-empty prefix-closed set of traces. A *process* of type A, written $p \models A$, is a synchronisation tree with labels from Σ_A, such that $\mathsf{traces}(p) \subseteq S_A$; strong bisimulation is the notion of equivalence. Here we will define operations on synchronisation trees by labelled transition rules; formal models of synchronisation trees can be obtained by, for example, working with Aczel's non-well-founded set representation [7].

Given A and B, the object $A \otimes B$ has

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

The duality is trivial on objects: $A^\perp = A$. Hence all the multiplicative connectives are the same: $A \parr B = A \multimap B = A \otimes B$. Now, a morphism $p : A \to B$ is a process p such that $p \models A \multimap B$. Since $\otimes$ is self-dual, $\mathcal{SP}roc$ is not only $*$-autonomous but *compact-closed*.

Composition is defined in line with the slogan "relational composition extended in time". If $p : A \to B$ and $q : B \to C$, so that $p \models A \multimap B$ and $q \models B \multimap C$, then $p\,;q : A \to C$ can be defined by labelled transitions:

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

in which matching of actions takes place in the common type B (as in relational composition), at each time step. This is the "interaction" of interaction categories.

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. If the process id with sort Σ_A is defined by

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

then id_A is obtained by pruning id so that it satisfies the safety specification $S_{A\multimap A}$.

We extend $\otimes$ and $(\cdot)^\perp$ to functors by defining their action on morphisms as follows. If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ and $p^\perp : C \to A$ are defined by

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

The tensor unit I is defined by $\Sigma_I = \{*\}$, $S_I = \{*^n \mid n < \omega\}$. We also have $\perp = I$. The correct notion of "point" in a $*$-autonomous category is a morphism from I, and indeed we can identify a process of type A with a morphism $p : I \to A$.

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

$$\frac{p \xrightarrow{((a,b),c)} q}{\Lambda(p) \xrightarrow{(a,(b,c))} \Lambda(q).}$$

The application morphism $\mathsf{Ap}_{A,B} : (A \multimap B) \otimes A \to B$ is obtained by pruning Ap (as for the identity morphisms), where Ap is defined by

$$\frac{a \in \Sigma_A \quad b \in \Sigma_B}{\mathsf{Ap} \xrightarrow{(((a,b),a),b)} \mathsf{Ap}.}$$

The definitions of the other structural morphisms, such as $\mathsf{symm} : A \otimes B \to B \otimes A$, are similar.

The coincidence of $\otimes$ and $\parr$, despite being a degeneracy which one would usually seek to avoid in models of linear logic, is very useful when connecting processes together—it validates a cycle rule which allows arbitrary process networks to be constructed.

Products in $\mathcal{SP}roc$ are defined by taking disjoint union of alphabets; the terminal object has an empty alphabet. Because of the self-duality, this gives biproducts and a zero object.

There are three delay functors which manipulate the temporal structure. The unit delay functor $\circ$ has

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

assuming that $* \notin \Sigma_A$. Given $p : A \to B$ we define $\circ p : \circ A \to \circ B$ by $\circ p \xrightarrow{(*,*)} p$. The other delay functors, δ which allows for delay before the first action and Δ which allows for delay anywhere except before the first action, have the structure of monads and correspond to Milner's delay operators [20].