Relations in Concurrency

Invited talk (corrected version)

Glynn Winskel, University of Cambridge Computer Laboratory, England

Abstract

The theme of this paper is profunctors, and their centrality and ubiquity in understanding concurrent computation. Profunctors (a.k.a. distributors, or bimodules) are a generalisation of relations to categories. Here they are first presented and motivated via spans of event structures, and the semantics of nondeterministic dataflow. Profunctors are shown to play a key role in relating models for concurrency and to support an interpretation as higher-order processes (where input and output may be processes). Two recent directions of research are described. One is concerned with a language and computational interpretation for profunctors. This addresses the duality between input and output in profunctors. The other is to investigate general spans of event structures (the spans can be viewed as special profunctors) to give causal semantics to higher-order processes. For this it is useful to generalise event structures to allow events which "persist."

1. Introduction

Standard relations between sets P and Q specify for a pair (p, q) in the product $P \times Q$ whether or not the pair is included in the relation. We can generalise the idea, and instead for each pair assign a *set* of ways in which the pair are related; assigning the empty set would mean not related at all. *Profunctors* arise by taking this idea one step further. Assume the sets have the structure of categories $\mathbb{P}$ and $\mathbb{Q}$ (perhaps very special categories such as partial orders). To respect the arrows, a profunctor assigns sets to pairs of objects (p, q) so that the assignment determines a covariant functor in p and a contravariant functor in q. (A concrete motivation and definition of profunctors is given in Section 3.)

Profunctors are "relations" between categories. (Profunctors are also known as distributors or bimodules [1, 15, 3].) As we'll see, profunctors recur in concurrency. They appear as appropriate "relations" for giving compositional semantics to nondeterministic dataflow. They express "relations" between categories of models for concurrency; for instance, both the inclusion of the category of synchronisation trees in the category of event structures and the operation serializing an event structure to a tree are given by profunctors. They stand for higher-order processes in a domain theory for concurrency.

We start with the model of event structures and their use in the semantics of nondeterministic dataflow. There are well-known difficulties in giving a compositional semantics to nondeterministic dataflow using standard relations between input and output. The problem is that standard relations take inadequate account of the causal relation between input and output. A compositional semantics can however be given with the nonstandard relations of profunctors. The profunctors are explained in terms of *spans* of event structures, representing the process of computation from input to output. We'll be led to the view of a nondeterministic process as a presheaf, a form of characteristic function for categories, where sets are taken as truth values. From there we'll see profunctors as "relations" between domains of processes.

The unity provided by profunctors suggests new languages and methods. I'll take advantage of this being an invited talk to present some recent, incomplete lines of work. One line is that of a process language based on profunctors. I'll outline a particular language developed with Patrick Baillot. Another line, in joint work with Lucy Saunders-Evans, concerns spans of event structures in the semantics of higher-order processes. This leads to a generalisation of event structures.

I've tried to be light and gradual in my use of category theory (occasional reference to the early parts of MacLane's book [16] may be helpful).

2. Event structures

Event structures [18, 23, 25, 26] are a model of computational processes. They represent a process as a set of event occurrences with relations to express how events causally depend on others, or exclude other events from occurring. In one of their simpler forms they consist of a set of events

on which there is a consistency relation expressing when events can occur together in a history and a partial order of causal dependency—writing $e' \leq e$ if the occurrence of e depends on the previous occurrence of e'.

In detail, an *event structure* comprises $(E, \mathrm{Con}, \leq)$, consisting of a set E, of *events* which are partially ordered by $\leq$, the *causal dependency relation*, and a *consistency relation* Con consisting of finite subsets of E, which satisfy

$$\{e' \mid e' \leq e\} \text{ is finite for all } e \in E,$$
$$\{e\} \in \mathrm{Con} \text{ for all } e \in E,$$
$$Y \subseteq X \in \mathrm{Con} \Rightarrow Y \in \mathrm{Con}, \text{ and}$$
$$X \in \mathrm{Con} \;\&\; e \leq e' \in X \Rightarrow X \cup \{e\} \in \mathrm{Con}.$$

Our understanding of the consistency predicate and the enabling relation are expressed in the notion of configuration (or state) we adopt for event structures. The events are to be thought of as event occurrences; in any history an event is to appear at most once. A configuration is a set of events which have occurred by some stage in a process. According to our understanding of the consistency predicate and causal dependency relations a configuration should be consistent and such that if an event appears in a configuration then so do all the events on which it causally depends. Here we restrict attention to finite configurations.

The *(finite) configurations*, $\mathcal{C}(E)$, of an event structure E consist of those finite subsets $x \subseteq E$ which are

Consistent: $x \in \mathrm{Con}$ and

Down-closed: $\forall e, e'.\ e' \leq e \in x \Rightarrow e' \in x$.

The configurations of an event structure are ordered by inclusion, where $x \subseteq x'$, *i.e.* x is a sub-configuration of x', means that x is a sub-history of x'. Note that an individual configuration inherits an order of causal dependency on its events from the event structure so that the history of a process is captured through a partial order of events. For an event e the set $\{e' \in E \mid e' \leq e\}$ is a configuration describing the whole causal history of the event e.

When the consistency relation is determined by the pairwise consistency of events we can replace it by a binary relation or, as is more usual, by a complementary binary conflict relation on events. It can be awkward to describe operations such as certain parallel compositions directly on the simple event structures here, because an event determines its whole causal history. One closely related and more versatile model is that of stable families, described in the addendum.

Let E and E' be event structures. A *map* of event structures $f : E \to E'$ is a partial function on events $f : E \rightharpoonup E'$ such that for all configurations x of E its direct image fx is a configuration of E' for which

$$\text{if } e_1, e_2 \in x \text{ and } f(e_1) = f(e_2) \in E', \text{ then } e_1 = e_2.$$

The map expresses how the occurrence of an event e in E induces the coincident occurrence of the event $f(e)$ whenever it is defined. The partial function f respects the instantaneous nature of events: two distinct event occurrences which are consistent with each other cannot both coincide with the occurrence of a common event in the image. Maps of event structures compose as partial functions.

We'll say the map is *total* iff the function f is total, and *rigid* iff it is total and for all configurations x of E and y of E'

$$y \subseteq f(x) \Rightarrow \exists z \in \mathcal{C}(E).\ z \subseteq x \text{ and } fz = y\ .$$

(The configuration z is necessarily unique.)

A rigid map of event structures preserves the causal dependency relation "rigidly," so that the causal dependency relation on the image fx is a copy of that on a configuration x of E; this is not so for general maps where x may be augmented with extra causal dependency over that on fx. (Recently with Lucy Saunders-Evans we have come to realize the primary nature of rigid maps in the sense that the other kinds of maps of event structures may be derived from them by a Kleisli construction—see Section 6.)

3. Nondeterministic dataflow

For dataflow networks built from only *deterministic* nodes, Kahn [14] observed that their behaviour can be captured *denotationally* as the least fixed point of a set of equations describing the components. Brock and Ackerman [4] were the first to point out that for *nondeterministic* dataflow achieving a compositional semantics was far from easy. In particular, *input-output* relations between sequences of values on input and output channels carry too little information about the behaviour of networks to support a compositional semantics.

We present two simple examples of automata A_1 and A_2, which have the same input-output relation, and yet behave differently in common context $C[-]$. The context consists of a fork process F (a process that copies every input to two outputs), through which the output of the automata A_i is fed back to the input channel, as shown in Fig. 1. Automaton A_1 has a choice between two behaviours: Either it outputs a token and stops, *or* it outputs a token, waits for a token on input and then outputs another token. Automaton A_2 has a similar nondeterministic behaviour: Either it outputs a token and stops, *or* it waits for an input token, then outputs two tokens. For both automata, the input-output relation relates empty input to the eventual output of one token, and non-empty input to one or two output tokens. But $C[A_1]$ can output two tokens, whereas $C[A_2]$ can only output a single token, choosing the first behaviour of A_2.

So there is no denotational semantics of nondeterministic dataflow in terms of traditional input-output relations. Any

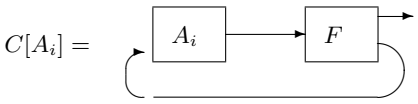
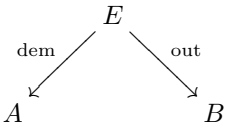


Figure 1. A context distinguishing A_1 and A_2

compositional semantics must take better account of the subtle causal dependency between input and output. There are several ways to do this, though they all fall outside the methods of *classical* domain theory, using powerdomains to adjoin nondeterminism. (Dataflow has a rich history which can't be done justice here—see [11] for a discussion and references.)

One solution is to give a semantics in terms of *spans* of event structures. Such a span comprises

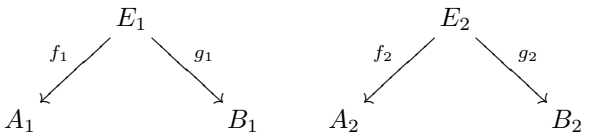


where A, B and E are event structures $\mathrm{out} : E \to B$ is a rigid map, and $\mathrm{dem} : E \to \mathcal{C}(A)$ satisfies

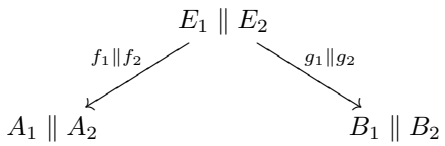
$$e \leq e' \Rightarrow \mathrm{dem}(e) \subseteq \mathrm{dem}(e') \text{ , and}$$
$$X \in \mathrm{Con} \Rightarrow \bigcup_{e \in X} \mathrm{dem}(e) \in \mathcal{C}(A) \text{ .}$$

The idea is that the occurrence of an event e in E demands the minimum input $\mathrm{dem}(e)$ and is visible as the output event $\mathrm{out}(e)$.

Such spans can be composed one after the other (essentially by a pullback construction, as both the demand and output maps extend to functions between configurations). They also have a nondeterministic sum, and compose in parallel. For example, spans

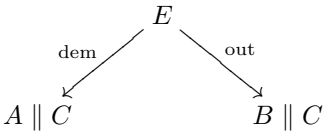


compose in parallel to give the span



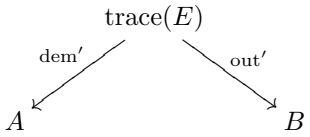
where the parallel composition $E_1 \parallel E_2$ of event structures, and maps, is given by their disjoint juxtaposition. With respect to $\parallel$ there is even a function space, discovered in joint work with Mikkel Nygaard [19]. And importantly spans also support the feedback loops of the problematic kind we've just seen.

Given a span

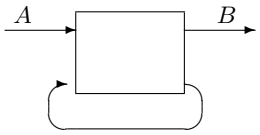


representing a process

there is a span, its *trace*,

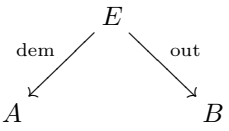


representing the process with a feedback loop:



The construction of the trace involves an intermediate construction on stable families, described in the addendum. (If the original span consists of event structures in which consistency/conflict is determined in a binary fashion, the feedback construction yields an event structure in which this is also the case.)

The central point here is that we can view spans of event structures as a form of *generalised* relation between input and output. Consider a span:



Let p be a configuration of A and q a configuration of B—so p is some particular input and q some particular output. Define the set

$$\widetilde{E}(p,q) = \{x \in \mathcal{C}(E) \mid \mathrm{dem}\, x \subseteq p \;\&\; \mathrm{out}\, x = q\} \text{ .}$$

The set $\widetilde{E}(p,q)$ consists of all the ways that input p can yield output q. Instead of simply specifying whether or not an input-output pair (p,q) obtains, as in a usual relation, $\widetilde{E}(p,q)$ gives the set of ways that (p,q) can be realized. In this sense the usual truth values, which apply in specifying

standard mathematical relations, have been replaced by a sets.

In fact $\widetilde{E}(p,q)$ is *functorial* in configurations $p \in \mathcal{C}(A)$ and $q \in \mathcal{C}(B)$. It respects the inclusion order on configurations (though, as we'll see, covariantly for input and contravariantly for output). Suppose $p \subseteq p'$ in $\mathcal{C}(A)$. Then, $\widetilde{E}(p,q) \subseteq \widetilde{E}(p',q)$—simply because any configuration x of E with demand $\mathrm{dem}\, x \subseteq p$ will make $\mathrm{dem}\, x \subseteq p'$. So, an inclusion $i : p \subseteq p'$ determines an inclusion map

$$\widetilde{E}(i,q) : \widetilde{E}(p,q) \hookrightarrow \widetilde{E}(p',q) \ .$$

Suppose $q \subseteq q'$ in $\mathcal{C}(B)$. This time the inclusion $j : q \subseteq q'$ determines a map

$$\widetilde{E}(p,j) : \widetilde{E}(p,q') \to \widetilde{E}(p,q) \ .$$

It takes x', for which $\mathrm{out}\, x' = q'$, to the unique subconfiguration $x \subseteq x'$ of E for which $\mathrm{out}\, x = q$; this exists because out is a rigid map of event structures. It is easy to see that these associations of maps with inclusions respect identities and composition, so that $\widetilde{E}(p,q)$ is functorial, covariantly in p and contravariantly in q.

In summary, we have a functor

$$\widetilde{E} : \mathcal{C}(A) \times \mathcal{C}(B)^{\mathrm{op}} \to \mathbf{Set}$$

from the product of the order $\mathcal{C}(A)$ with $\mathcal{C}(B)^{\mathrm{op}}$, the opposite order, to the category of sets. We have seen that it is a kind of generalised relation, where truth values are taken to be sets.

We can regard orders as categories, so more generally we can consider a "relation" between small categories $\mathbb{P}$ and $\mathbb{Q}$ to be a functor

$$F : \mathbb{P} \times \mathbb{Q}^{\mathrm{op}} \to \mathbf{Set} \ ,$$

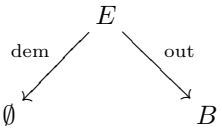
a functor from a product of categories $\mathbb{P}$ and the opposite category $\mathbb{Q}^{\mathrm{op}}$. Such a functor is called a *profunctor* from $\mathbb{P}$ to $\mathbb{Q}$ and is written as

$$F : \mathbb{P} \mathrel{-\!\!\!+\!\!\!\to} \mathbb{Q} \ .$$

The significance of profunctors was first highlighted by Bénabou and Lawvere [1, 15]. As is to be expected we can compose profunctors $F : \mathbb{P} \mathrel{-\!\!\!+\!\!\!\to} \mathbb{Q}$ and $G : \mathbb{Q} \mathrel{-\!\!\!+\!\!\!\to} \mathbb{R}$, between small categories $\mathbb{P}$, $\mathbb{Q}$ and $\mathbb{R}$, to obtain a profunctor $G \circ F : \mathbb{P} \mathrel{-\!\!\!+\!\!\!\to} \mathbb{R}$. We postpone the definition of composition to Section 4.2. With such *generalised* relations we are able to give a compositional semantics of nondeterministic dataflow [11]. The work can alternatively be carried out with spans of event structures.

4. Processes as presheaves

As motivation, consider a special span of event structures of the form



where $\emptyset$ is the unique event structure with no events and the demand function is such that $\mathrm{dem}(e) = \emptyset$, the empty input configuration, for all events e of E. No input is needed in producing output. Such a degenerate span determines a functor

$$X = \widetilde{E}(\emptyset, -) : \mathcal{C}(B)^{\mathrm{op}} \to \mathbf{Set} \ .$$

Such a functor X is called a *presheaf* over the order $\mathcal{C}(B)$. The presheaf X represents a nondeterministic process whose computation paths have the shape of configurations of B (with causal order inherited from B). Given a configuration q of B, the set $X(q)$ describes all the different computation paths of X of shape q. Because X is a contravariant functor the order $j : q \subseteq q'$ determines a function $X(j) : X(q') \to X(q)$ saying how computation paths of shape q' restrict to computation paths of shape q. The presheaf X is a form of *characteristic function*, where the truth values are sets.

We can broaden our understanding of shapes of computation paths to be objects in a small category $\mathbb{Q}$. The category $\mathbb{Q}$ is thought of as consisting of shapes of paths, where a map $j : q \to q'$ in $\mathbb{Q}$ expresses how the q extends to q'. A presheaf $X : \mathbb{Q}^{op} \to \mathbf{Set}$ specifies for q in $\mathbb{Q}$ the set $X(q)$ of computation paths of shape q. The presheaf X acts on a map $j : q \to q'$ in $\mathbb{Q}$ to give a function $X(j)$ saying how q'-paths in X restrict to q-paths in X—several paths may restrict to the same path. In this way a presheaf can model the nondeterministic branching of a process.

A presheaf X over $\mathbb{Q}$ models a nondeterministic process of which the computation paths have shapes in $\mathbb{Q}$. The category $\mathbb{Q}$ represents a *type* of the process. We can gather all the processes of type $\mathbb{Q}$ together. The category of presheaves over $\mathbb{Q}$, written $\widehat{\mathbb{Q}}$, is the functor category $[\mathbb{Q}^{op}, \mathbf{Set}]$, with objects the functors from $\mathbb{Q}^{op}$ to the category of sets, and maps the natural transformations between them.

Example 4.1 Consider the order of nonempty strings L^+ over a set of actions L, ordered by extension, so for instance $ab \leq abbc$. The presheaf category $\widehat{L^+}$ is isomorphic to the category of synchronisation trees, with arc labels in L; the maps are simulations, *i.e.* functions on nodes respecting roots, arcs and labels.

Example 4.2 This example shows how categories of event structures arise as presheaf categories. A *pomset* [22] is a labelled event structure in which all finite subsets of events are consistent. The category of finite pomsets Pom_L can be presented as a subcategory of the category of event structures where events are labelled in L, with total maps that respect labels. The category Pom_L consists of path shapes in the form partial orders of labelled events. The category of labelled event structures embeds fully and faithfully in presheaf category $\widehat{\mathrm{Pom}_L}$, which in this sense consists of generalised event structures [13].

4.1. Bisimulation

Presheaves are being thought of as nondeterministic processes on which equivalences such as bisimulation are important in abstracting away from inessential differences of behaviour. A sweeping definition of bisimulation between presheaves is derived from the notion of open map [13].

The category of presheaves is accompanied by the *Yoneda embedding*, a functor $\mathrm{y}_{\mathbb{Q}} : \mathbb{Q} \to \widehat{\mathbb{Q}}$, which fully and faithfully embeds $\mathbb{Q}$ in the category of presheaves. For every object q of $\mathbb{Q}$, the Yoneda embedding yields $\mathrm{y}_{\mathbb{Q}}(q) = \mathbb{Q}(-, q)$.

A map $h : X \to Y$, between presheaves X and Y, is *open* iff for all maps $j : q \to q'$ in $\mathbb{Q}$, any commuting square

$$\begin{array}{ccc} \mathrm{y}_{\mathbb{Q}}(q) & \xrightarrow{x} & X \\ {\scriptstyle \mathrm{y}_{\mathbb{Q}}(j)}\downarrow & & \downarrow{\scriptstyle h} \\ \mathrm{y}_{\mathbb{Q}}(q') & \xrightarrow[y]{} & Y \end{array}$$

can be split into two commuting triangles

$$\begin{array}{ccc} \mathrm{y}_{\mathbb{Q}}(q) & \xrightarrow{x} & X \\ {\scriptstyle \mathrm{y}_{\mathbb{Q}}(j)}\downarrow & \overset{z}{\nearrow} & \downarrow{\scriptstyle h} \\ \mathrm{y}_{\mathbb{Q}}(q') & \xrightarrow[y]{} & Y. \end{array}$$

That the square commutes means that the path $h \circ x$ in Y can be extended via j to a path y in Y. That the two triangles commute means that the path x can be extended via j to a path z in X which matches y.

Open maps are a generalisation of functional bisimulations, known from transition systems. Presheaves in $\widehat{\mathbb{Q}}$ are *bisimilar* iff there is a span of surjective open maps between them. Open-map bisimulation often coincides with known definitions of bisimulation. Open-map bisimulation in the presheaf category $\widehat{L^+}$ of Example 4.1 is strong bisimulation of Milner and Park. An operational understanding of open-map bisimulation on profunctors from $\mathbb{P}$ to $\mathbb{Q}$ (regarded as presheaves over $\mathbb{P}^{\mathrm{op}} \times \mathbb{Q}$) is something of an enigma (it respects the input-output duality of profunctors, *cf.* Section 5). Open maps also support a general theory of weak bisimulation [9].

4.2. Relating presheaf categories

Intuitively a presheaf over a small category $\mathbb{P}$ consists of a collection of paths with shapes in $\mathbb{P}$ glued together at sub-paths—so a presheaf resembles a computation tree but in which the branches have shapes in $\mathbb{P}$. Technically this amounts to a presheaf being expressible as a colimit of its paths.

A presheaf category has all limits and colimits given pointwise, at a particular object, by the corresponding limits and colimits of sets. In particular, a presheaf category has all sums (coproducts) of presheaves. In process terms, a sum of presheaves represents a nondeterministic sum of processes.

A category of presheaves, $\widehat{\mathbb{P}}$, is characterized as the free colimit completion of $\mathbb{P}$. The Yoneda embedding $\mathrm{y}_{\mathbb{P}} : \mathbb{P} \to \widehat{\mathbb{P}}$ satisfies the universal property that for any functor $F : \mathbb{P} \to \mathcal{C}$, where $\mathcal{C}$ is a category with all colimits, there is a colimit-preserving functor $G : \widehat{\mathbb{P}} \to \mathcal{C}$, determined to within natural isomorphism, such that $F \cong G \circ y_{\mathbb{P}}$:

$$\begin{array}{ccc} \mathbb{P} & \xrightarrow{\mathrm{y}_{\mathbb{P}}} & \widehat{\mathbb{P}} \\ & {\scriptstyle F}\searrow \quad \cong & \downarrow{\scriptstyle G} \\ & & \mathcal{C} \end{array}$$

Any presheaf category $\widehat{\mathbb{Q}}$ has all colimits. So, in particular, for any functor $F : \mathbb{P} \to \widehat{\mathbb{Q}}$, there is a colimit-preserving functor $G : \widehat{\mathbb{P}} \to \widehat{\mathbb{Q}}$, determined to within natural isomorphism, such that $F \cong G \circ y_{\mathbb{P}}$:

$$\begin{array}{ccc} \mathbb{P} & \xrightarrow{\mathrm{y}_{\mathbb{P}}} & \widehat{\mathbb{P}} \\ & {\scriptstyle F}\searrow \quad \cong & \downarrow{\scriptstyle G} \\ & & \widehat{\mathbb{Q}} \end{array}$$

By this universal property, colimit-preserving functors $G : \widehat{\mathbb{P}} \to \widehat{\mathbb{Q}}$ correspond to within natural isomorphism to functors $F : \mathbb{P} \to \widehat{\mathbb{Q}}$, and such functors are in 1-1 correspondence with profunctors $\bar{F} : \mathbb{P} \mathrel{-\!\!\!\rightarrow\!\!\!\!\!+} \mathbb{Q}$. A functor $F : \mathbb{P} \to \widehat{\mathbb{Q}}$ is a functor

$$F : \mathbb{P} \to [\mathbb{Q}^{op}, \mathbf{Set}] \,,$$

so by "uncurrying" in correspondence with a functor

$$\bar{F} : \mathbb{P} \times \mathbb{Q}^{\mathrm{op}} \to \mathbf{Set} \,,$$

viz. a profunctor $\bar{F} : \mathbb{P} \mathrel{-\!\!\!\rightarrow\!\!\!\!\!+} \mathbb{Q}$.

So profunctors arise in relating presheaf categories. We can now describe how to compose them. Given profunctors

$F : \mathbb{P} \to\!\!\!\!\!+ \mathbb{Q}$ and $G : \mathbb{Q} \to\!\!\!\!\!+ \mathbb{R}$, by "currying" we obtain functors

$$\bar{F} : \mathbb{P} \to \widehat{\mathbb{Q}} \text{ and } \bar{G} : \mathbb{Q} \to \widehat{\mathbb{R}} .$$

From the universal property we obtain a colimit-preserving functor $\bar{G}^\dagger : \widehat{\mathbb{Q}} \to \widehat{\mathbb{R}}$ so now we can form the composition of functors

$$\bar{G}^\dagger \circ F : \mathbb{P} \to \widehat{\mathbb{R}} .$$

This functor corresponds to a profunctor, which we take to be the composition of profunctors

$$G \circ F : \mathbb{P} \to\!\!\!\!\!+ \mathbb{R} .$$

Presheaves over $\mathbb{Q}$ correspond to special profunctors

$$X : \mathbf{1} \to\!\!\!\!\!+ \mathbb{Q}$$

from the category $\mathbf{1}$ comprising just a single object and its identity map. Profunctor composition specializes to the *application* of a profunctor to a presheaf. A profunctor $G : \mathbb{Q} \to\!\!\!\!\!+ \mathbb{R}$ applied to X is given by $G^\dagger(X)$, where $G^\dagger : \widehat{\mathbb{Q}} \to \widehat{R}$ is the colimit-preserving functor determined by G. Application preserves open-map bisimulation because:

Theorem 4.3 *[6] Let $H : \widehat{\mathbb{Q}} \to \widehat{\mathbb{R}}$ be a colimit-preserving functor between presheaf categories. Then H preserves surjective open maps and open-map bisimulation.*

Example 4.4 We refer to Examples 4.1 and 4.2. Recall that $\widehat{L^+}$ is the category of synchronisation trees and that $\widehat{\mathrm{Pom}_L}$ consists of generalised labelled event structures. There is an obvious functor $I : L^+ \to \mathrm{Pom}_L$ which regards a string over actions L as a pomset. By composition we obtain a functor $\mathrm{y}_{\mathrm{Pom}_L} \circ I : L^+ \to \widehat{\mathrm{Pom}_L}$, which extends by the universal property of y_{L^+} to a colimit-preserving functor $I_! : \widehat{L^+} \to \widehat{\mathrm{Pom}_L}$. The functor $I_!$ coincides with the inclusion of synchronisation trees in event structures. For general reasons, the functor $I_!$ has a right adjoint $I^* : \widehat{\mathrm{Pom}_L} \to \widehat{L^+}$ given by $I^*(E) = \widehat{\mathrm{Pom}_L}(\mathrm{y}_{\mathrm{Pom}_L} \circ I(-), E)$. The functor I^* coincides with the operation of serializing or interleaving an event structure to a tree. The functors $I_!$ and I^* relate an interleaving model and a noninterleaving model of concurrency. Both functors are colimit-preserving and so correspond to profunctors. Consequently both $I_!$ and I^* preserve bisimulation—in $\widehat{L^+}$ it is strong bisimulation, while in $\widehat{\mathrm{Pom}_L}$ open-map bisimulation turns out to be hereditary history-preserving bisimulation of Bednarczyk. This illustrates how profunctors play a fundamental role in relating categories of models for concurrency.

5. A language for profunctors

We now take a broader view. We regard small categories, $\mathbb{P}$, $\mathbb{Q}$, $\mathbb{R}$, ..., as types of processes. Processes of a particular type $\mathbb{P}$ are represented by objects in the presheaf category $\widehat{\mathbb{P}}$. Processes of different types, say $\mathbb{P}$ and $\mathbb{Q}$, are related by profunctors $F : \mathbb{P} \to\!\!\!\!\!+ \mathbb{Q}$, or equivalently by the colimit-preserving functors $F^\dagger : \widehat{\mathbb{P}} \to \widehat{\mathbb{Q}}$ they correspond to. With this broader picture we have a form of domain theory for concurrency [21] and we can ask how to construct types, and what terms denote profunctors. Our answers are guided by linear logic [10].

A tensor product of $\mathbb{P}$ and $\mathbb{Q}$ is given by the product of categories $\mathbb{P} \times \mathbb{Q}$. There is an associated function space $\mathbb{P} \multimap \mathbb{Q}$ given by $\mathbb{P}^{op} \times \mathbb{Q}$—it is easy to see that profunctors from $\mathbb{P}$ to $\mathbb{Q}$ are exactly presheaves over $\mathbb{P}^{op} \times \mathbb{Q}$. The involution of linear negation is represented by the operation which takes a profunctor $F : \mathbb{P} \to\!\!\!\!\!+ \mathbb{Q}$ to the profunctor $F^\perp : \mathbb{Q}^{op} \to\!\!\!\!\!+ \mathbb{P}^{op}$, given by

$$F^\perp(q, p) = F(p, q) ,$$

got by switching around the roles of input and output. A family of objects $\mathbb{P}_\alpha$, for $\alpha \in A$, has sums (*i.e.* coproducts) and products given in the same way on objects by their disjoint juxtaposition $\Sigma_{\alpha \in A} \mathbb{P}_\alpha$. As for the exponential ! of linear logic, there are many possible choices—see [20, 6].

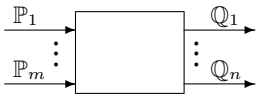
We pause to consider the meaning of linearity. For a profunctor $F : \mathbb{P} \to\!\!\!\!\!+ \mathbb{Q}$ the functor $(\bar{F})^\dagger : \widehat{\mathbb{P}} \to \widehat{\mathbb{Q}}$ is determined by its action on single computation paths of the input process. Consequently application of a profunctor satisfies a linear property: a computation path of the output results from a single computation path of the input process. Because application of a profunctor preserves colimits, it preserves nondeterministic sums, and the empty sum representing the nil process. The linear property only holds for very special operations on processes, and for example would fail to hold for most operations of parallel composition, or of prefixing a process by an action. If we are to address the concerns of concurrency we must go beyond purely linear maps, and linear logic provides the means. The facility to copy or ignore an input process is introduced by an exponential ! and nonlinear maps are obtained as linear maps from a type $!\mathbb{P}$. In fact, a lot can be achieved by just allowing the input process to be ignored, without having the facility to copy—the nonlinear maps that result are called *affine*. We allow input to be ignored through *lifting* of the input type. The operation of *lifting*, $\mathbb{P}_\perp$, introduces a new initial object $\perp$, which can be thought of as the empty path, below a small category $\mathbb{P}$.

The constructions of tensor $\mathbb{P} \times \mathbb{Q}$, sum $\Sigma_{\alpha \in A} \mathbb{P}_\alpha$, types $\mathbb{P}^{\mathrm{op}}$ and lifting $\mathbb{P}_\perp$ are used in forming the types of a language for profunctors developed with Patrick Baillot. To facilitate the duality that exists between input an output on profunctors, its typing judgements of terms t take the form

$$x_1 : \mathbb{P}_1, \cdots, x_m : \mathbb{P}_m \vdash t \dashv y_1 : \mathbb{Q}_1, \cdots, y_n : \mathbb{Q}_n ,$$

where all the variables are distinct, interpreted as a profunc-

tor from $\mathbb{P}_1 \times \cdots \times \mathbb{P}_m$ to $\mathbb{Q}_1 \times \cdots \times \mathbb{Q}_n$. We can think of the term t as a box with input and output wires for the typed variables:



The duality of input and output is caught by the rules:

$$\frac{\Gamma, x : \mathbb{P} \vdash t \dashv \Delta}{\Gamma \vdash t \dashv x : \mathbb{P}^{\mathrm{op}}, \Delta} \qquad \frac{\Gamma \vdash t \dashv x : \mathbb{P}, \Delta}{\Gamma, x : \mathbb{P}^{\mathrm{op}} \vdash t \dashv \Delta}$$

Composition of profunctors is described in the rule

$$\frac{\Gamma \vdash t \dashv \Delta \qquad \Delta \vdash u \dashv \mathrm{H}}{\Gamma \vdash \exists \Delta.\ t \times u \dashv \mathrm{H}}$$

which joins the input wires of one process to output of the other. (The "existential quantifier" can be interpreted directly as a coend and $\times$ as the product of sets.)

We can form the nondeterministic sum of processes of the same type:

$$\frac{\Gamma \vdash t_i \dashv \Delta \quad i \in I}{\Gamma \vdash \sum_{i \in I} t_i \dashv \Delta}$$

The sum denotes the coproduct of profunctors; we can regard profunctors as presheaves and form their coproduct in a pointwise fashion, using the disjoint union of sets.

The rule for lifting

$$\frac{\Gamma \vdash t \dashv \Delta, y : \mathbb{R}}{\Gamma \vdash \text{ lift } y \text{ to } y' \text{ in } t \dashv \Delta, y' : \mathbb{R}_\perp}$$

is associated with the operation extending a profunctor

$$F : \mathbb{P}_1 \times \cdots \times \mathbb{P}_m \nrightarrow \mathbb{Q}_1 \times \cdots \times \mathbb{Q}_n \times \mathbb{R}$$

to a profunctor

$$F' : \mathbb{P}_1 \times \cdots \times \mathbb{P}_m \nrightarrow \mathbb{Q}_1 \times \cdots \times \mathbb{Q}_n \times \mathbb{R}_\perp\ ,$$

which on the additional arguments acts so that

$$F'(p_1, \cdots, p_m, q_1, \cdots, q_n, \perp) = \{*\},\ \text{a singleton.}$$

The hom-set rule

$$\frac{\Gamma \vdash p' : \mathbb{P} \qquad \Delta \vdash p : \mathbb{P}}{\Gamma \vdash p \leq_{\mathbb{P}} p' \dashv \Delta}$$

introduces a term standing for the hom-set $\mathbb{P}(p, p')$. It relies on path terms, notation for paths involving free variables, and their typings; a typing judgement for a path term p

$$x_1 : \mathbb{P}_1, \cdots, x_m : \mathbb{P}_m \vdash p : \mathbb{Q}$$

denotes a functor from $P_1 \times \cdots \times \mathbb{P}_m$ to $\mathbb{Q}$. One rule for such judgements is

$$\frac{\Gamma \vdash p : \mathbb{Q}}{\Gamma^{\mathrm{op}} \vdash p : \mathbb{Q}^{\mathrm{op}}}$$

where Γ^{op} is $x_1 : \mathbb{P}_1^{\mathrm{op}}, \cdots, x_m : \mathbb{P}_m^{\mathrm{op}}$. For the type $\mathbb{P}_\perp$, there are path terms !, standing for the path $\perp$, and $!p$, where p is a path term of type $\mathbb{P}$. For a sum $\Sigma_{\alpha \in A} \mathbb{P}_\alpha$ there are path terms βp where $\beta \in A$ and p is a path term of type $\mathbb{P}_\beta$. Other instances of path term typings are $x : \mathbb{P}, y : \mathbb{Q} \vdash x \times y : \mathbb{P} \times \mathbb{Q}$ and $z : \mathbb{P} \times \mathbb{Q} \vdash z : \mathbb{P} \times \mathbb{Q}$ from which via the hom-set rule we obtain

$$x : \mathbb{P}, y : \mathbb{Q} \vdash\ z \leq_{\mathbb{P} \times \mathbb{Q}} x \times y\ \dashv z : \mathbb{P} \times \mathbb{Q}\ ,$$

which joins two inputs to a common output of tensor type. A great deal is achieved through basic manipulation of the input and output "wiring" afforded by the hom-set rules and input-output duality.

A *path* is a path term with no free variables. The notation for paths enables a further useful construction on types. For its description we take advantage of the fact that for this language types will always be orders. If p is a path of $\mathbb{P}$, then the *resumption* type, $\mathbb{P}/p$, is the order obtained from those paths strictly above p; it is the type a process of type $\mathbb{P}$ arrives at after having done the path p.

This describes the core of the language. Once extended by recursive terms and recursive types, it becomes highly expressive, and, for example, can straightforwardly encode the higher-order process language affine-HOPLA in away that is faithful to its presheaf semantics [20, 21]. A derivation in affine-HOPLA of a judgement

$$x_1 : \mathbb{P}_1, \ldots, x_n : \mathbb{P}_n \vdash M : \mathbb{Q}$$

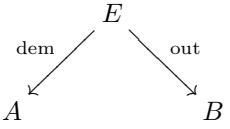
is translated into a derivation of

$$x_1 : (\widetilde{\mathbb{P}_1})_\perp, \ldots, x_n : (\widetilde{\mathbb{P}_n})_\perp \vdash \widetilde{M} \dashv y : \widetilde{\mathbb{Q}}\ .$$

The affine function space of affine-HOPLA is translated by the operation $(\mathbb{P}_\perp)^{\mathrm{op}} \times \mathbb{Q}$ on types $\mathbb{P}$ and $\mathbb{Q}$, and its affine tensor by $(\mathbb{P}_\perp \times \mathbb{Q}_\perp)/(!\times!)$. Prefixing by an action makes use of lifting and sum. Of course the language for profunctors has restrictions that moving from $\mathbb{P}_\perp$ to a full exponential $!\mathbb{P}$ should address. The language does not yet have an operational semantics, which would likely throw light on open-map bisimulation for higher-order processes.

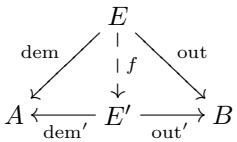
6. General spans of event structures

In Section 3 we saw that a span of event structures



determines a profunctor $\widetilde{E} : \mathcal{C}(A) \nrightarrow \mathcal{C}(B)$. Not all profunctors from the order $\mathcal{C}(A)$ to the order $\mathcal{C}(B)$ are obtained

in this way however. In particular the corresponding functor from $\mathcal{C}(A)$ to $\widehat{\mathcal{C}(B)}$ preserves pullbacks. This is why composition of the associated profunctors can be described by a simple pullback construction on spans. Maps between spans are reasonably taken to be rigid maps f as shown



where both triangles commute as maps on configurations. They induce *cartesian* natural transformations (*i.e.* where the naturality squares are pullbacks) between the corresponding functors. The spans are akin to stable functions in stable domain theory [2].

Such spans of event structures can be used to give a semantics to affine-HOPLA, as explained in Mikkel Nygaard's thesis [19]. In fact the spans were discovered in deriving an operational semantics from the presheaf semantics of affine-HOPLA. The guiding principle in designing the operational semantics was that derivations of transitions of a closed term should correspond to elements (realizers) in its presheaf denotation, at least at first order [20]. We discovered that the profunctors definable in affine-HOPLA at first-order can be represented by spans of event structures. This shed light on the affine tensor—at first-order it is the simple parallel composition $\|$ of Section 3—and explained a form of entanglement as due to consistency/conflict of events. The event-structure semantics extends to all types, though diverges from the presheaf semantics at higher types (just as the function space of stable domain theory differs from the pointwise function space of classical domain theory).

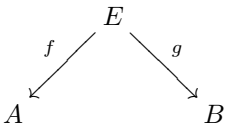
Despite event structures appearing out of the presheaf semantics of affine-HOPLA that language can be proved to not support the traditional event-structure semantics of process calculi like CCS (as given for example in [24, 27]). Nor do the spans of Section 3 appear to be sufficient. This is one motivation for very recent work with Lucy Saunders-Evans where we have been looking to more general spans of event structures.

The exploration has been helped by the gradual realization that all the maps on event structures seen here, whether they be total, partial, or even the demand maps on the left of spans of event structures, can be obtained in a uniform way from rigid maps. For example, total maps of event structures can be obtained as rigid maps from E to $T(E')$ for a monad T on the category of event structures with rigid maps. A similar Kleisli construction yields, the other kinds of maps too, though at a slight cost. Event structures must be extended to allow a "persistent" events. An event structure with persistence (E, P) is an event structure E paired with a distinguished subset of persistent events P. Configurations are defined just as before. Maps $f : (E, P) \to (E', P')$ of event structures with persistence are partial functions on events $f : E \rightharpoonup E'$ such that $fP \subseteq P'$ and for all configurations x of E its direct image fx is a configuration of E' for which now

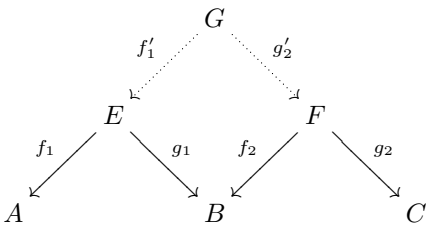
$$\text{if } e_1, e_2 \in x \text{ and } f(e_1) = f(e_2) \in (E' \setminus P'), \text{ then } e_1 = e_2.$$

The maps compose as partial functions. A map on event structures with persistence is *rigid* iff it comprises a total function which preserves the order of causal dependency. This amounts to the same definition as before when no events are persistent.

Let $\mathcal{E}$ be the category of event structures with persistence and rigid maps. It has all pullbacks and a terminal object (and so products). We can form the bicategory of spans $\mathrm{Span}_{\mathcal{E}}$. Its objects are event structures with persistence. Its maps $\mathrm{Span}_{\mathcal{E}}(A, B)$, from A to B, are spans

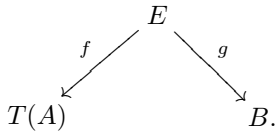


composed using pullbacks:



(Its 2-cells, maps in $\mathrm{Span}_{\mathcal{E}}(A, B)$, are the maps between the vertices of two spans making the obvious triangles commute.) $\mathrm{Span}_{\mathcal{E}}$ has a tensor and function space given by the product of $\mathcal{E}$.

Other bicategories of spans are obtained by Kleisli constructions on $\mathrm{Span}_{\mathcal{E}}$—see [5]. A monad on $\mathcal{E}$ that respects pullbacks induces both a (pseudo) monad and (pseudo) comonad on $\mathrm{Span}_{\mathcal{E}}$. In particular the demand-output spans of Section 3 arise as spans



in a Kleisli construction on spans, stemming from T, the monad on $\mathcal{E}$ associated with demand maps. But there are other largely unexplored variations in which one or both legs of the span are modified by monads. Several yield both function spaces and interesting parallel compositions of event structures.

7. Concluding remarks

Profunctors and presheaves have much wider applications beyond concurrency, in logic [17], in combinatorics via species [12] and generalised species [8] (where languages for profunctors again appear naturally and have a process reading), in algebraic topology [7], as well as in the broader world of mathematics. Here I've glossed over the details of bicategories and pseudo monads and comonads. Work with Marcelo Fiore, Nicola Gambino and Martin Hyland, is designed to make these notions more precise and workable. Name generation has been ignored here. Adding name generation to the domain theory arising from profunctors raises the issue of the existence of function spaces. The paper [28], of this volume, addresses this question for a simplification of profunctors where the role of sets as truth values is replaced by $0 < 1$.

Acknowledgements

I am grateful for discussions with Patrick Baillot, Marcelo Fiore, Thomas Hildebrandt, Martin Hyland, Mikkel Nygaard and Lucy Saunders-Evans.

Addendum: Stable families and trace

The use of stable families facilitates definitions on event structures. Here we'll use stable families to define the trace operation of Section 3.

Definition A *stable family* comprises $(E, \mathcal{F})$ where E is a set of *events* and $\mathcal{F}$ is a family of finite subsets of E, called *configurations*, satisfying:
Completeness: $Z \subseteq \mathcal{F}$ & $Z\uparrow \Rightarrow \bigcup Z \in \mathcal{F}$;
Coincidence-freeness: For all $x \in \mathcal{F}$, $e, e' \in x$ with $e \neq e'$,

$$(\exists y \in \mathcal{F}.\ y \subseteq x \ \&\ (e \in y \iff e' \notin y))\ ;$$

Stability: $\forall Z \subseteq \mathcal{F}.\ Z \neq \emptyset\ \&\ Z\uparrow \Rightarrow \bigcap Z \in \mathcal{F}$.

For $Z \subseteq \mathcal{F}$, we write $Z\uparrow$ to mean compatibility, *i.e.*

$$\exists x \in \mathcal{F} \forall z \in Z.\ z \subseteq x\ .$$

Configurations of stable families each have their own local order of causal dependency, so their own prime sub-configurations generated by their events. We can build an event structure by taking the events of the event structure to comprise the set of all prime sub-configurations of the stable family. The details follow.

Definition and Proposition Let x be a configuration of a stable family $\mathcal{F}$. For $e, e' \in x$ define

$$e' \leq_x e \text{ iff } \forall y \in \mathcal{F}.\ y \subseteq x\ \&\ e \in y \Rightarrow e' \in y.$$

When $e \in x$ define

$$\lceil e \rceil_x = \bigcap \{y \in \mathcal{F} \mid y \subseteq x\ \&\ e \in y\}\ .$$

Then $\leq_x$ is a partial order and $\lceil e \rceil_x$ is a configuration such that

$$\lceil e \rceil_x = \{e' \in x \mid e' \leq_x e\}.$$

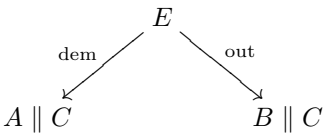
Moreover the configurations $y \subseteq x$ are exactly the down-closed subsets of $\leq_x$.

Proposition Let $(E, \mathcal{F})$ be a stable family. Then, $(P, \mathrm{Con}, \leq)$ is an event structure where:

$$\begin{aligned} &P = \{\lceil e \rceil_x \mid e \in x\ \&\ x \in \mathcal{F}\}\ , \\ &Z \in \mathrm{Con} \text{ iff } Z \subseteq P\ \&\ \bigcup Z \in \mathcal{F} \text{ and,} \\ &p \leq p' \text{ iff } p, p' \in P\ \&\ p \subseteq p'\ . \end{aligned}$$

This proposition furnishes a way to construct an event structure with events the prime configurations of a stable family. In fact we can equip the class of stable families with maps (the definitions are the same as those for event structures). The configurations of an event structure form a stable family, so in this sense event structures are included in stable families. With respect to any of the maps (rigid, total or partial), the "inclusion" functor from the category of event structures to the category of stable families has a right adjoint, which on objects is the construction we have just given, producing an event structure from a stable family. The product w.r.t. partial maps is particularly useful in giving semantics to the parallel composition of synchronising processes, as in CCS and CSP. The product is hard to define directly on the event structures of this article. It is however straightforward to define the product of stable families [24]. Right adjoints preserve limits, and so products in particular. Consequently we obtain the product of event structures by first regarding them as stable families, and then producing the event structure from the product of the stable families.

We construct the trace of a span of event structures



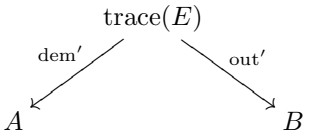
of Section 3.

Let $x \in \mathcal{C}(E)$. Say e *is secured in* x iff

$$\begin{aligned} &\exists e_1, \cdots, e_n \in x.\ e_n = e\ \ \& \\ &\qquad \forall i \leq n.\ \{e_1, \cdots, e_{i-1}\} \in \mathcal{C}(E)\ \ \& \\ &\qquad\qquad \mathrm{dem}(e_i) \cap C \subseteq \mathrm{out}\{e_1, \cdots, e_{i-1}\}\ . \end{aligned}$$

Say x is *secure* iff all its events are secured in x.

It can be shown that $(E, \mathcal{F})$, in which $\mathcal{F}$ consists of all the secure configurations of E, is a stable family. Define $\mathrm{trace}(E)$ to be the event structure $(E', \mathrm{Con}', \leq')$ with events E' consisting of the prime configurations $\lceil e \rceil_x$ of $\mathcal{F}$ for which $\mathrm{out}(e) \in B$, where consistency Con' is given by compatibility in $\mathcal{F}$ and causal dependency is given by inclusion. The original demand function dem induces a function dem' from E' to $\mathcal{C}(A)$: take $\mathrm{dem}'(e') = \bigcup_{e \in e'} \mathrm{dem}(e) \cap A$. The original output function out induces a function out' from E' to B: take $\mathrm{out}'(e') = \mathrm{out}(e)$ when e' has the form $\lceil e \rceil_x$. Together these determine a span of maps



—the trace of Section 3, and a representation of the trace on profunctors used in [11].

References

[1] Bénabou, J., Les distributeurs. Rapport n° 33. Seminaires de Mathématiques Pure, Institut de Mathématiques, Université Catholique de Louvain, 1973.

[2] Berry, G., *Modèles complètement adéquats et stable des lambda-calculus typés*. PhD thesis, L'université Paris VII, 1979.

[3] Borceux, F., *Handbook of categorical logic*, volume 1. Cambridge University Press, 1994.

[4] Brock, J. and Ackerman, W., Scenarios: A model of non-determinate computation. In Diaz, J. and Ramos, I., editors, *Formalization of Programming Concepts*, volume 107 of *LNCS*. Springer, 1981.

[5] Burroni, A., T-catégories. *C*ahiers de topologie et géométrie différentielle, XII 3, 1971.

[6] Cattani, G.L., and Winskel, G., Profunctors, open maps and bisimulation. In press, MSCS, 2005.

[7] Fahrenberg, U., Bisimulation for higher-dimensional automata. A geometric interpretation. Aalborg Univ Dept of Mathematical Sciences, Research Report R-2005-01, 2005.

[8] Fiore, M., Mathematical models of computational and combinatorial structures. Invited address, FOSSACS'05, 2005.

[9] Fiore, M., Cattani, G.L., and Winskel, G., Weak bisimulation and open maps. Proc. of LICS'99. 1999.

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

[11] Hildebrandt, T., Panangaden, P., and Winskel, G. A relational model of non-deterministic dataflow. *Mathematical Structures in Computer Science*, 2004.

[12] Joyal, A., Foncteurs analytiques et especes de structures. In *Proceedings of a Colloquium on Enumerative Combinatorics, Springer Lecture Notes in Mathematics*, vol.1234, 1985.

[13] Joyal, A., Nielsen, M., and Winskel, G., Bisimulation from open maps. *LICS '93 special issue of Information and Computation*, 127(2):164–185, 1996. Available as BRICS report, RS-94-7.

[14] Kahn, G., The semantics of a simple language for parallel programming. In *Information Processing*, volume 74, pages 471–475, 1974.

[15] Lawvere, F.W., Metric spaces, generalized logic and closed categories. *Rend. Sem. Mat. Fis. Milano*, 43:135–166, 1973.

[16] Mac Lane, S. *Categories for the Working Mathematician*. Springer, 1971.

[17] Mac Lane, S. and Moerdijk, I. , *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*, Springer, 1992.

[18] Nielsen, M., Plotkin, G.D., and Winskel, G., Petri nets, event structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1981.

[19] Nygaard, M., Domain theory for concurrency. PhD Thesis, University of Aarhus, 2003.

[20] Nygaard, M., and Winskel, G., Linearity in Process Languages. In Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02), 2002.

[21] Nygaard, M., and Winskel, G., Domain theory for concurrency. *Theoretical Computer Science* 316: 153–190, 2004.

[22] Pratt, V., Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15,1, 1986.

[23] Winskel, G., *Events in Computation*. PhD thesis, University of Edinburgh, available as a Comp. Sc. report, 1980.

[24] Winskel, G., Event structure semantics of CCS and related languages. Springer–Verlag Lecture Notes in Comp. Sc. 140 ICALP 82, 1982. An extended version is available from http://www.cl.cam.ac.uk/ gw104.

[25] Winskel, G., Event structures. Invited lectures for the Advanced Course on Petri nets, September 1986. Springer Lecture Notes in C.S., vol.255, 1987.

[26] Winskel, G., An introduction to event structures. In the lecture notes for the REX summerschool in temporal logic, May 88, in Springer Lecture Notes in C.S., vol.354, 1988.

[27] Winskel, G. and Nielsen, M., *Handbook of Logic in Computer Science*, volume IV, chapter Models for concurrency, pages 1–148. OUP, 1995.

[28] Winskel, G., Name generation and linearity. In Proceedings of 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05), this volume, 2005.