Electronic Notes in Theoretical Computer Science 7 (1997)
URL: http://www.elsevier.nl/locate/entcs/volume7.html 21 pages

# Types as Processes, via Chu spaces

Vaughan R. Pratt [1]

*Computer Science Department*
*Stanford University*
*Stanford, USA*

**Abstract**

We match up types and processes by putting values in correspondence with events, coproduct with (noninteracting) parallel composition, and tensor product with orthocurrence. We then bring types and processes into closer correspondence by broadening and unifying the semantics of both using Chu spaces and their transformational logic. Beyond this point the connection appears to break down; we pose the question of whether the failures of the corrrespondence are intrinsic or cultural.

## 1 Introduction

Types-as-processes modernizes data-as-programs. It is the Curry-Howard propositions-as-types correspondence with propositions replaced by processes. To the extent that types and processes are both part of the working programmer's toolkit, even more than propositions, the types-as-processes correspondence is more central to the practice of programming than propositions-as-types. Moreover the connection works out very well mathematically, at least up to a point.

The similarities and differences between programs and data have been studied for several decades from many angles. The particular point of view in this paper is based on two observations:

(i) Event-state duality, and its close cousin time-information duality, work in essentially the same way as dualities of the kind encountered in many mathematical structures.

(ii) Duality is more than just a phenomenon, it can be made a universal foundation for both processes and mathematics.

Observation (i) can be traced at least as far back as Nielsen, Plotkin and Winskel [17], who establish the duality of prime event structures and prime algebraic domains. Event structures with their temporal ordering consist of

[1] This work was supported by ONR under grant number N00014-92-J-1974

events while Scott domains with their information ordering consist of states. The duality that NPW found between them is a true categorical duality in the sense that it reverses the morphisms of the respective categories.

The transitions and places of a Petri net [18] hint at this duality by being respectively universal and existential: a firing transition involves every incident edge whereas a token passing through a place involves just one path through that place. But it was only relatively recently observed [6,7] that this could be made a true duality in the morphism-reversing sense.

True duality arises in many mathematical situations, the best-known of which have important applications. The duality of locally compact abelian groups, which in the finite case reduces to the self-duality of finite abelian groups, plays a central role in the complementarity principle of quantum mechanics. The duality of Stone spaces and Boolean algebras, and of ordered Stone spaces and distributive lattices, is a cornerstone of modern model theory. And the self-duality of finite-dimensional vector spaces, along with its infinite-dimensional extensions to topological vector spaces and Hilbert spaces, is the basis for linear algebra and its many applications.

Observation (ii) constitutes the present author's present interest [23,24]. The core theorems of this observation are that all small concrete categories, as well as all categories of algebraic or relational structures and their homomorphisms, are concretely representable as categories of biextensional Chu-Mackey spaces [15,3] and their continuous maps. (A Chu-Mackey space, or just Chu space, over a set $K$ is simply a rectangular array over $K$, that is, a function from $A \times X$ to $K$ where the sets $A$ and $X$ index the rows and columns respectively. Biextensional means that all rows are distinct and likewise all columns.)

## 2 Type Algebra

### *2.1 Comparison: order and morphisms*

The principal structural elements in the connections between types, processes, and propositions are comparison, complementation, and aggregation,

In all three cases—propositions, types, and processes—objects $A, B$ can be compared via either inclusions (inequalities $A \leq B$ or $A \vdash B$ in the case of propositions) or functions (*proofs* of $A \vdash B$ in the case of propositions). The former can be seen as a special case of the latter by viewing inclusions as functions satisfying $f(x) = x$, or in the case of propositions, viewing simple inequalities as nonempty equivalence classes of proofs: $A$ entails $B$ just when there *exists* a proof of $B$ from $A$.

We may understand the process of comparing $A$ with $B$ either statically in terms of (asymmetric) distance from $A$ to $B$ or dynamically in terms of motion from $A$ to $B$. In either case there is an ongoingness or connectedness about comparison: a comparison of $A$ with $B$ and another of $B$ with $C$ invites a *composite* comparison of $A$ with $C$. This invitation is replied to statically with a suitable triangle inequality such as transitivity (a condition), or dynamically as the composite motion from $A$ to $C$ via $B$ (a construction).

When types are represented naively as sets of values, and processes equally naively as sets of traces, comparison can be reduced to inclusion. For types, inclusion represents the subtype relation. For processes it represents the safety-liveness spectrum, with safety in smaller processes and liveness in larger, as for parties.

A more sophisticated form of comparison is provided by functions, which can transform types into types or processes into processes. The respective projections of the type $A \times B$ to its constituent types $A$ and $B$ are not sensibly modeled by any inclusion. Yet important elements of inclusion are retained by these more general comparisons, permitting the familiar logic of inclusion to be extended to the considerably less familiar logic of transformation. These logics are equally a part of both type theory and process theory, and go a long way towards shaping both in essentially the same way.

For processes, when the only comparisons between processes are inclusions, the traces of the process may as well be regarded as a discrete set, one with no explicitly given structure relating its traces, since simple inclusions cannot see or use any such structure. The process $ab + ac$ (denoting the set $\{ab, ac\}$) raises the question of whether it equals $a(b + c)$. Underlying this question is another partial order, the prefix order on individual traces, which reveals $a$ as a common prefix of $ab$ and $ac$. But what does "common" mean? When you say you are looking for a dollar, do you mean that you have lost a dollar bill or that you are out to make a buck? Obviously $ab$ and $ac$ have $a$ in common as the label on their respective first events, but this leaves open whether those two events are in fact one.

One way to resolve this is to make the notion of prefix explicit by requiring that processes be prefix-closed. Thus $\{ab, ac\}$ expands to $\{\epsilon, a, ab, ac\}$. But we can also make the labeling notion explicit by working with multisets. This permits the multiset $\{\epsilon, a, a, ab, ac\}$ as an alternative result of prefix closure, now with $prefix(ab) \neq prefix(ac)$. But now inclusion is no longer well-defined: in what sense are either $\{\epsilon, a\}$ or $\{\epsilon, a, a\}$ subsets of $\{\epsilon, a, a, ab, ac\}$, other than in a counting sense?

Now we could formalize the multiset $\{a, a\}$ by taking it to be a set with two elements, say 27 and 43, and a labeling function assigning $a$ to both elements. This will work provided the corresponding elements are used for the two $a$'s in the set $\{\epsilon, a, a, ab, ac\}$ (and say 513 and 36 for $ab$ and $ac$). But in order that inclusion comparisons yield the right answer every time, we are obliged to use these arbitrary choices consistently for all traces. The objectionable part mathematically speaking is not so much the arbitrariness as the requirement of consistent usage of those arbitrary elements.

Exactly this objection applies when a gensym'd atoms like G000547 appears on the screen when printing out the result of a query in LISP. The point of printing it is that when it appears again we can make the connection.

This is where functions do a better job than inclusions. Instead of inferring the inclusion from the trace identities, express the relationship between $\{\epsilon, a, a\}$ and $\{\epsilon, a, a, ab, ac\}$ explicitly as a function from the former set to the latter. This function will map the two $a$'s of the first set injectively to the two $a$'s of the second, and is what category theorists mean by "inclusion," namely an embedding. The labels are realized by a labeling function that does need to commute with the embedding but otherwise just goes along for the ride: we no longer rely on the labels to determine which traces connect to which between two processes, the connection being made instead by an explicitly given function.

The same solution applies to the printing of the gensym'd atom G000547. If instead a line were drawn connecting the two occurrences of G000547, and the printname reduced to a mere blob, the real content would be both clearer and less cluttered up with arbitrariness.

We have thus seen two distinct roles for functions: catering for noninjections, and for keeping track of element identities in evolving multisets.

### *2.2 Complementation*

Motion is oriented, but its perceived orientation depends on that of the perceiver. Facing the other way reverses objects, motion, and composition: object $a$ becomes $\bar{a}$, motion $a \to b$ becomes $\bar{b} \to \bar{a}$, and composition $a \xrightarrow{f} b \xrightarrow{g} c$ becomes $\bar{c} \xrightarrow{\bar{g}} \bar{b} \xrightarrow{\bar{f}} \bar{a}$.

In naive type theory complementation leaves the objects (namely sets) untouched, but reverses the subtype relation. For processes, complementation interchanges safety and liveness. Both situations depended on inclusion determining a poset. Since the order dual of a poset is another poset, complementation is a function from one poset to another. Hence the logic of reverse inclusion is the same as that of inclusion, provided we stick to reasoning about the partial orders defined by inclusion and reverse inclusion.

When complementation $\bar{a}$ sends its argument $a$ into a different poset, any operation combining $a$ and its complement $\bar{a}$ would need to be heterogeneous. As long as our algebras of types and processes stick to just poset structure however and don't use such operations, this is not an issue; we just need to keep track of which poset each inequality in our reasoning refers to.

The passage from inclusions to functions works equally smoothly. The one potential sticking point is that the converse of a function is not in general another function.

Two solutions suggest themselves. The first solution is to note that the converse of a function is at least a binary relation. This suggests that if we want to continue using complementation in the concrete setting of sets and suitable morphisms between them, then the right morphisms are binary relations rather than functions.

The problem with this solution is that type theory is heavily function-oriented. Limits and colimits behave very differently in the category **Rel** of sets and binary relations from how they behave in the category **Set** of sets and functions, with the latter setting the standard for our expectations of the "normal" behavior of those operations. Particularly distressing in **Rel** is that product and coproduct are the same operation, with their object parts both corresponding to coproduct in **Set**.

Solution two meets this concern simply by observing that $\mathbf{Set}^{\mathrm{op}}$ is, if not a category of sets and functions, at least a category. This justifies complementation of functions by the same reasoning that justified complement for inclusions (that the order dual of a poset is a poset). As with posets, it is necessary to keep track of which structure a given comparison is in. In this case we need to keep track of whether any given comparison is being made with a function or an antifunction, the latter being a catchy term for the morphisms of $\mathbf{Set}^{\mathrm{op}}$.

But now we notice that we have just reinvented two of the essential ingredients of first order logic, via an ostensibly very different path from Frege and Peirce. Functions map sets to sets of course, but antifunctions in effect map Boolean algebras to Boolean algebras. More precisely, the category $\mathbf{Set}^{\mathrm{op}}$ is equivalent to the category of CABAs or complete atomic Boolean algebras (where "complete atomic" is needed only for infinite Boolean algebras).

The complement of a set $A$ is the power set $B = 2^A$, a CABA whose Boolean operations are arbitrary union and complement relative to $A$ and whose singletons $\{a\}$ for each $a \in A$ are its atoms. We recover the set $A$ from the CABA $B$ up to isomorphism (which is all that equivalence of categories requires) as the atoms of $B$.

The complement of a function $f : A \to A'$ is the complete Boolean algebra homomorphism $2^f : 2^{A'} \to 2^A$ defined by $2^f(g) = g \circ f$, where $g : A' \to 2$ and hence $g \circ f : A \to 2$. $2^f$ maps in the opposite direction to $f$, as befits a complement.

For example to complement the successor function mod 3, namely $\sigma : 3 \to 3$, first complement 3 itself. Its complement is the power set $2^3$ of 3. The complement of $\sigma$ is the function sending each subset $g : 3 \to 2$ (represented as its characteristic function) to $g \circ \sigma$. This just the predecessor function acting pointwise on the elements of each subset. The complement of the constantly zero function on 3 on the other hand does not act pointwise; instead it maps subsets of 3 containing zero to 3 itself (the whole space), and those not containing zero to the empty set.

The underlying general principle here is the contravariance of the functor $\mathrm{Hom}(-, c) : C^{\mathrm{op}} \to \mathbf{Set}$ for any object $c$ (called the *dualizing object*) in any category $C$. In the preceding example the category $C$ is **Set**, $c$ is the set 2 and $\mathrm{Hom}(-, 2) : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Set}$ is the contravariant power set functor sending each set $X$ to its powerset $\mathrm{Hom}(X, 2)$ and each function $f : X \to Y$ to the function sending each element $g$ of $\mathrm{Hom}(Y, 2)$ to the element $g \circ f$ of $\mathrm{Hom}(X, 2)$.

While this may seem a bit of a mouthful, the core of the idea is very simple. Returning to the generality of $C$ and $c$, let $a \xrightarrow{f} b$ be any morphism in $C$. The result of attaching this $f$ on the left of any morphism $g : b \rightarrow c$ and composing is to yield a morphism $a \xrightarrow{f} b \xrightarrow{g} c$. Even though $f$ points to the right, attaching it to $g$ in this way "stretches" $g$ to the left. This change in the behavior of $f$ from a right-moving morphism to a left-moving one is the essence of the contravariance. By identifying this so-called "homming into $c$" with complementation, we turn the type $a \rightarrow b$ of morphisms from $a$ to $b$ into the type $\bar{b} \rightarrow \bar{a}$ of functions from $\bar{b}$ to $\bar{a}$.

A picky point here is that complementation has turned morphisms of $C$ into functions, since we have taken $\bar{a}$ etc. to be a set, namely the set of all morphisms from $a$ to our choice $c$ of dualizing object. (We do not obtain all functions in this way however, only the CABA homomorphisms.) We would prefer that complementation yield morphisms of $C$, which would make it a homogeneous operation; in particular we could apply it a second or third time.

The necessary homogeneity can be achieved by replacing the external homfunctor $\mathrm{Hom} : C^{\mathrm{op}} \times C \rightarrow \mathbf{Set}$ as used above by a suitable *internal homfunctor* $\multimap : C^{\mathrm{op}} \times C \rightarrow C$. This makes $C$ a *closed category* in the sense that complementation, and any other construction defined in terms of the homfunctor, leaves us in $C$ instead of dumping us in (a subcategory of) **Set**.

Now it would be nice if the naive view of complementation as "facing the other direction" were realized by an involutary complementation, one satisfying double negation. In general we can rely only on the direction $a \multimap a''$, in that we can "reliably" map $a$ to $(a \multimap c) \multimap c$ but not necessarily conversely. The obvious choice of morphism for this purpose is evaluation (part of the structure of a closed category): apply to the given element of $a$ the given map in $a \multimap c$ yielding an element of $c$. In the other direction however there is no robust procedure for extracting an element of $a$ given only a map from $a \multimap c$ to $c$. Satisfying double negation must therefore be made an explicit requirement.

These two requirements together, closed categories and an involutary negation, constitute the defining characteristics of M. Barr's $*$-autonomous categories [3]. This interprets complementation classically, satisfying double negation and thus realizing the intuition of complementation as simple reversal.

A particular method of constructing such categories is the Chu-Mackey construction, described in the appendix of [3].

In the case of ordinary Chu spaces, the Chu construction applied to the category **Set**, these have the additional advantages of accommodating essentially all of transformational mathematics [24], and of extending event structures in a natural way [10,9,27].

These circumstances combine to make involutary complementation very appealing. We assume it henceforth.

### 2.3 Aggregation

Aggregation forms large collections from small. As an operation aggregation is definable at all arities, although for practical purposes it is customarily defined at arities 0 and 2, from which all other finite arities are obtained by composition. It is normally associative and often commutative.

Aggregation may be either disjunctive or conjunctive depending on how we perceive the elements being aggregated: if as alternatives then disjunctively, if as coexisting entities then conjunctively. Complementation typically interchanges these: the De Morgan dual of conjunctive aggregation can be seen as having a disjunctive quality and vice versa.

However having a disjunctive quality is not necessarily the same thing as being a specific disjunctive operation. Given two aggregations, one disjunctive and one conjunctive, it is possible that the De Morgan dual of one is different from the other. Such a situation arises for example with the De Morgan-Peirce-Schröder calculus of binary relations, abstracted to relation algebra, RA, by Jónsson and Tarski [12,13], with its logical and relative versions of conjunction and disjunction. The same structure arises with Girard's linear logic, LL, whose additive and multiplicative operations match up perfectly with the logical and relative operations respectively of relation algebra.

*Interaction with comparison.* Aggregation interacts with both comparison and complementation. The interaction with comparison depends on how the particular operation is defined. There are two basic ways of interest to us.

The first way is to define binary disjunction and conjunction as respectively left and right adjoints of the diagonal $\Delta : P \to P^2$. This is categorical jargon for

$$a \vee b \leq c \quad \text{iff} \quad a \leq c \text{ and } b \leq c$$

$$a \leq b \text{ and } a \leq c \quad \text{iff} \quad a \leq b \wedge c.$$

These are stated for posets, where they define $\vee$ and $\wedge$ respectively, as the two aggregation operations of lattices. The first can be read as saying that $a \vee b$ is the least element $c$ which is an upper bound of both $a$ and $b$; the second is its dual. These define join and meet respectively in any lattice. In relation algebra these are union and intersection of relations, which Peirce called the logical operations.

But the same definitions can be interpreted also for categories, where they define (up to isomorphism) categorical sum and product, by interpreting $\leq$ as Hom and iff as an isomorphism that is natural in $A$, $B$, and $C$ (switching to upper case for the moment). For sum the isomorphism matches up functions $h : A+B \to C$ with pairs of functions $f : A \to C, g : B \to C$, via $h(a) = f(a)$, $h(b) = g(b)$.

For product the correspondence is between pairs $f : A \to B$, $g : A \to C$ and functions $h : A \to B \times C$, via $h(a) = (f(a), g(a))$. If we interpret "element of" as "map to", then this definition of $B \times C$ identifies its elements with the pairs $(f, g)$ of elements of $B$ and $C$ respectively, which is exactly the ordinary meaning of $B \times C$ in **Set**. The nice thing is that this meaning remains the same in any other category with products.

For sum there exists a natural dual correspondence. If we interpret "predicate on" as "map to" (allowing any object $C$ to be understood as consisting of truth values) then the definition of $A + B$ identifies its predicates with the pairs $(f, g)$ of predicates on $B$ and $C$ respectively. While this is not the ordinary definition of sum in **Set**, namely as disjoint union, it is a natural way to think of sum having the additional benefit that it generalizes smoothly to any other category with sums.

The second general kind of aggregation, notated $a \otimes b$, arises for a poset $P$ when there is an implication $b \multimap c$ (also called a residual) as a binary operation $\rightarrow : P \times P \rightarrow P$, monotone in $c$ and antimonotone in $b$. For a category $C$ the counterpart of implication is an internal homfunctor $a \multimap b$ as a functor $\multimap : C^{\mathrm{op}} \times C \rightarrow C$. Such an aggregation is called *tensor product*,[2] and is defined by left adjunction to $b \rightarrow c$ in $c$ holding $b$ fixed (so $b \rightarrow$ serves as a unary operation that we are in effect going to invert). This reduces to the elementary definition

$$a \otimes b \leq c \quad \text{iff} \quad a \leq b \multimap c.$$

Although stated for posets, again it can be reinterpreted for categories as putting the morphisms from $a \otimes b$ to $c$ into a natural bijection with the morphisms from $a$ to $b \multimap c$.

When $a \otimes b$ is not commutative there are two residuals, a so-called right residual $a \multimap c$ satisfying $a \otimes b \leq c \quad \text{iff} \quad b \leq a \multimap c$ and a left residual $c \multimapinv b$ satisfying $a \otimes b \leq c \quad \text{iff} \quad a \leq c \multimapinv b$ (which is how we should have written it above, but in the symmetric case we can be sloppy). The two together determine, and are determined by, the same tensor product $a \otimes b$. A well-known example of this is provided by relation algebra, which has the (noncommutative) composition of binary relations as a tensor product in this sense, having a right and left residual respectively. For ordinary types however, and for concurrent processes with $\otimes$ taken to be orthocurrence, tensor product has in our experience always been commutative. We therefore do not pursue the noncommutative case further.

When tensor product coincides with ordinary product in a lattice, the lattice is called a *Heyting algebra*. In a category, if tensor product is an ordinary product the category is said to be *cartesian closed*. In RA this is what happens when we take implication to be neither of the two residuals, but instead ordinary Boolean implication $b \rightarrow c$, which when viewed as a function of $c$ has a left adjoint which is just logical conjunction or meet. In LL the same situation arises by taking implication to be the intuitionistic implication $A \Rightarrow B = !A \multimap B$ where $!A$ is the "underlying coalgebra" of $A$, which can be in-

[2] Usually tensor product is specified independently and one then asks whether it has a right adjoint in one or both arguments. However in practice the interdefinability of tensor product and its adjoint(s) makes it convenient to start from either end of the defining adjunction.

terpreted loosely as $A$ less whatever structure is preventing $\multimap$ from behaving like intuitionistic implication.

The whole situation with implication/residuation/internal hom and its/their left adjoint dualizes to what could naturally be called subtraction, having "tensor sum" as its right adjoint. This happens automatically in the presence of involutary complementation, treated below, as in both RA (where Peirce has called the De Morgan dual of composition *relative sum*) and LL (where Girard has called the De Morgan dual of tensor product *par*, notated $A \parr B$).

*Interactions between aggregations.* The interactions between conjunctions and disjunctions of the various kinds constitute the various distributivity laws. When either additive (meet with join, or sum with product) distributes over the other, we have either a distributive lattice or a distributive category, and that distributivity law then automatically implies the other.

Necessarily $a \otimes (b + c) = a \otimes b + a \otimes c$ because both $\otimes$ and $+$ are defined as colimits and therefore commute.

In practice $a \otimes (b \parr c) \leq (a \otimes b) \parr c$ holds but not conversely. This is weak or linear distributivity.

*Interaction with complementation.* All of the above involved only the interaction of aggregation with comparison. The interaction with classical complementation (the only kind we will consider from now on) extends this picture.

The logical or additive conjunction and disjunction are De Morgan duals of one another in both RA and LL. The same happens with the relative or multiplicative conjunction $a \otimes b$ and disjunction $a \parr b$.

## 3 Process Models

We turn now from type algebra to process algebra. Process algebra is best developed with a particular model in mind. The choice of model influences the natural choice of operations of the algebra. Therefore before treating process algebra we shall home in on a preliminary choice of process model.

Process algebras such as ACP have their origins in the algebra of regular expressions, which is made a process algebra by adjoining a suitable parallel composition operator. Regular expressions constitute an ideal algebra for sequential computation, at least up to trace semantics, and moreover one that all computer science students are taught early on. Hence this would seem a perfect starting point for process algebra.

But this passage to concurrency naively assumes that what makes the standard model of computation sequential is simply the omission of those operators that are only distinguishable from regular operators in the presence of two or more concurrent processes.

The thesis of true concurrency is that the standard model of sequential computation needs more than just additional operators to model concurrency. For the operation $a\|b$ to be more than merely $ab + ba$ when $a$ and $b$ are atomic, there should be a certain connectedness between $ab$ and $ba$ expressing their independence. Without such connectedness $ab + ba$ implies a choice of which goes first. No choice is involved for truly independent events.

The standard model can express this connectedness in terms of refinement of $a$ and $b$ into sequences of subevents $a_1a_2 \ldots a_m$ and $b_1b_2 \ldots b_n$. These then interleave as a grid. The presence of all $\binom{m+n}{m}$ such interleavings witnesses the desired connectedness.

But this is a somewhat artificial representation of this connectedness whose only redeeming feature is that it can be expressed in the standard model.

The notion is expressed better via with Mazurkiewicz traces [16], which explicitly represent independence in terms of a partial monoid equivalencing $ab$ and $ba$. Equivalently in the case of independence limited to two events at a time, higher dimensional automata (HDA's) [22] express this information geometrically in the form of a filled-in square having $ab$ and $ba$ as boundaries. HDA's go beyond Mazurkiewicz traces in being able to express nonindependence of three events any two of which are independent, e.g. three children taking turns riding two ponies.

But by far the simplest representation of $a\|b$ is as the set $\{a, b\}$ consisting of the two events $a$ and $b$.

In general, independence of a set $A$ of events is represented by its discreteness, in the sense of lack of any structure.

With discreteness as our starting point we can now proceed to assign structure to processes. A basic form of structure is temporal, expressed by constraints on the order of events. Order is naturally expressed by a partial ordering $\leq$ on $A$. We take the meaning of $a \leq b$ to be that $a$ precedes $b$ in time; equivalently, if $b$ has occurred then so has $a$.

Another form of structure is the labeling of events with actions. We would like to model a string such as $xyzzy$ as a linearly ordered set of five events each labeled with one of three actions $x, y, z$. This can be expressed in terms of an alphabet $\Sigma$ (or **Act**) of such actions together with a labeling function $\lambda : A \to \Sigma$ assigning actions to events. Such a labeled set performs the function of a multiset over $\Sigma$, a collection of elements of $\Sigma$ allowing repetitions of the same element. The set $A$ serves merely as coathangers on which to hang the labels.

A structure $(A, \leq, \lambda)$ which is both ordered and labeled forms a *partially ordered multiset* [19] or pomset. Pomsets over an alphabet $\Sigma$ generalize words over $\Sigma$ by not requiring that the order be linear.

A pomset by itself is a choiceless process in the sense that all its events must eventually happen. In the case of nonlinear pomsets one might suppose there is a discrete choice to be made in the order of events. However this assumes that order of occurrence is well-defined, which need not be the case. A run of a pomset may be the pomset itself, or any augmentation of its order which makes two incomparable elements comparable, the limiting case of which is a linearly ordered run. Augmentation indeed forces discrete choices, but augmentation is not necessary given that the pomset may be its own run.

A global notion of choice may be expressed as a set of pomsets, from which a run chooses one pomset, possibly augmented (but explicit augmentation can be avoided by using only sets of pomsets that are augment-closed). Such a set of pomsets forms a process.

A more local form of choice may be expressed at the level of events using a suitable conflict relation. Binary conflict $a\#b$ is a symmetric irreflexive binary relation specifying whether or not a given pair of events may both occur. When $a\#b$ holds, $a$ and $b$ may not both occur in the same run. In that case a choice must be made between them, assuming all events required to precede them have occurred.

A poset together with such a binary conflict relation satisfying an event structure axiom forms a *prime event structure* [17]. The event structure axiom is that if $a\#b$ and $b \leq c$ then $a\#c$. That is, if $a$ prevents $b$ and $c$ cannot happen until $b$ does, then $a$ also prevents $c$.

A pomset so equipped forms a labeled prime event structure.

A set of pomsets can be represented as a single event structure by forming the disjoint union of the pomsets as one big pomset, and then putting in conflict all pairs of events coming from different pomsets. No converse translation exists, witness for example the $V$-shaped poset with events $a, b, c$ satisfying $a \leq b$, $a \leq c$, made an event structure via $b\#c$.

Event structures, with or without labels, have become well-established in concurrency theory. In order to make a start on process algebra we settle for them for the moment.

In the next section we shall need to know which functions between event structures preserve that structure, i.e. are event structure homomorphisms. We take these to be monotone (order-preserving) functions which also preserve conflict: if $a\#b$ then $f(a)\#f(b)$.[3] In the presence of labels we add the further requirement that such homomorphisms commute with the respective labeling functions, that is, they leave the labels undisturbed.

## 4 Process Algebra

Process Algebra, PA [2], has as operations choice $A+B$, sequence $AB$, and two kinds of concurrence, $A\|B$ together with an asymmetric "left-merge" variant.

The choice, sequence, and left-merge operations have no evident counterparts in type theory. Concurrence of the noninteracting kind however is an excellent

[3] This is different from the morphisms advocated by Winskel for event structures [28], which are defined to make partially synchronous parallel composition ordinary categorical product. Our morphisms are more conventional as regards preservation of structure, and are also those that arise naturally with Chu spaces.

match to disjoint union, both being definable in the appropriate category as coproduct. Coproduct of event structures, with the morphisms as defined in the previous section, is just their disjoint union in the evident sense, with neither order nor conflict holding between events from different arguments of the coproduct. Such a compound event structure permits its components to run completely independently. In particular the coproduct of singleton event structures is just a discrete set.

An operation we have found very useful in specifying concurrent processes is orthocurrence, $A \otimes B$ [20,21]. Originally conceived as ordinary categorical product, the author's students identified it later as a tensor product. For posets and pomsets this tensor product reduces to ordinary product, both classes forming cartesian closed categories. But for metric spaces used to model real time, orthocurrence turned out not to be ordinary product. This topic is developed at much greater length in [8].

A basic example of orthocurrence is in the specification of the behavior of a system of trains and stations along a single railway line. The set of trains and the set of stations each forms a linearly ordered set (in the case of partially ordered time, as opposed say to real time). However the events of the various arrivals of trains at stations are not linearly ordered. Concurrency of this form resulting from two systems "flowing through" each other occurs routinely in nature and computation. So pervasive a concurrency phenomenon should surely be expressible in process algebra. Orthocurrence provides the appropriate operation.

Orthocurrence of posets is simpy their ordinary or categorical product, namely cartesian product of their elements. When labeled, the alphabet of the orthocurrence is the product of those of the arguments, with the evident labeling. For example if a train labeled $T$ is at a station labeled $S$ then the event of that train being at that station is labeled $(S, T)$.

For event structures, orthocurrence treats order as for posets, but treats conflict as a local phenomenon. Conflict in the orthocurrence of two event structures is defined as the least symmetric binary relation such that if $a\#b$ and $c \leq d$, or $a \leq b$ and $c\#d$, then $(a, c)\#(b, d)$. Equivalently, $(a, c)\#(b, d)$ just when either $a\#b$ and ($c \leq d$ or $d \leq c$), or ($a \leq b$ or $b \leq a$) and $c\#d$.

For example $\{a, b\} \otimes \{c, d\}$ as orthocurrence of discrete sets is itself the discrete set $\{(a, c), (a, d), (b, c), (b, d)\}$, of which any subset can occur. Putting $a$ in conflict with $b$ puts $(a, c)$ in conflict with $(b, c)$ as one would expect. However it does not put $(a, c)$ in conflict with $(b, d)$ because $a$ is "visible" only from $c$ and not from $d$, and conversely for $b$, whence there are no witnesses to the conflict of $a$ and $b$. But if $c \leq d$ then $(b, c) \leq (b, d)$, which the event structure axiom combines with $(a, c)\#(b, c)$ to obtain $(a, c)\#(b, d)$. On the other hand, $a\#b$ and $c\#d$ together need not entail $(a, c)\#(b, d)$.

This definition of orthocurrence for event structures, besides being the natural choice via the "visibility" argument, agrees exactly with the tensor product of the Chu representation of event structures.

The process algebra with operations concurrence $A \| B$ and orthocurrence $A \otimes B$ is the one we propose to match up with the coproduct and tensor product of type algebra. In this matchup, values in types correspond to events in processes (here defined as event structures). There is however no clear correspondence between the variety of structures a type might be equipped with, and an event structure, which is a very specialized type.

The rest of the paper brings types and processes together by generalizing both using Chu spaces.

## 5 Chu spaces: Rationale

A set $A$ is equipped with algebraic and/or relational structure in the form of one or more nonempty relations of any arity. Absence of structure is then represented algebraically by the empty signature—no relations—or by having only empty relations in the signature.

But absence of structure can also be represented topologically by the discrete topology: every subset of $A$ is taken to be open.

We shall however reject the usual condition on the open sets of a topological space that they be closed under arbitrary union and finite intersection. In fact we shall impose no condition at all, allowing any combination of open sets as a generalized notion of topology.

The topological representation fits naturally with the intuition of a set of events being independent when all possible states are allowed. Here the open sets correspond to states: each subset of $A$ denotes the state in which the events in that subset have occurred.

This supposes that each individual event $a$ has only two states, occurred or not occurred, represented in any given state by membership or nonmembership of $a$ in that state. We collect these two states into a set $K$ of *atomic* states. A state global to the whole process can then be described as a function $A \to K$.

With this approach of representing "open sets" of $A$ as their $K$-valued characteristic functions, a greater diversity of atomic states is easily accommodated with a larger set $K$. For example to distinguish *not-yet-started* from *ongoing* from *completed*, take $K = \{0, 1, 2\}$ respectively. This corresponds to topology with three-valued membership in its open sets.

Such three-valued topology is then competitive with higher dimensional automata. For example the interference present with three children taking turns riding two ponies, with each child permitted one ride, is expressed by allowing 26 of the $3^3 = 27$ states. The disallowed state is the one which assigns 1 to all three events (children) representing the case when every child is riding some pony.

## 6 Chu spaces: Definition

Let $K$ be a set. A *Chu space* $\mathcal{A} = (A, r, X)$ *over* $K$ consists of sets $A$ and $X$ and a function $r : A \times X \to K$.

We shall use Chu spaces as both types and processes. As a type, a Chu space is viewed as a generalized topological space having $A$ as its set of points or values, $X$ as the set of open subsets of $A$, and $r$ as the $K$-valued membership relation: $r(a, x)$ indicates the degree to which point $a$ belongs to open set $x$.

As a process, $K$ is the set of atomic states, $A$ is the set of events, $X$ is the set of (global) states, and $r(a, x)$ is the atomic state of event $a$ in state $x$.

When $r(a, x) = r(b, x)$ for all $x \in X$ we say that the points $a$ and $b$ are *isomorphic*. Dually when when $r(a, x) = r(a, y)$ for all $a \in A$ we call the states $x$ and $y$ isomorphic.

A Chu space containing no distinct isomorphic points (states) is called *separated* (*extensional*). $\mathcal{A}$ is *biextensional* when it is both separated and extensional.

For types, extensionality is a natural requirement for open sets, there being no significance attached to multiple occurrences of the same open set in a topological space. A separated or $T_0$ topological space is one all of whose points "behave" differently, and corresponds to the antisymmetry property for posets. In fact a finite topological space is exactly a preordered set (reflexive and transitive) while a $T_0$ such is exactly a poset. The $T_0$ property is thus a very mild condition to impose on a type, merely ensuring that the type contains no distinct yet isomorphic values.

For processes, distinct states are naturally expected to be distinguished by some event, whose atomic state is different in the two states. Thus extensionality is a reasonable requirement for processes. Isomorphic events $a, b$ are events that are completely synchronized, being in the same atomic state $r(a, x) = r(b, x)$ in every global state $x$. It might be reasonable to allow distinct but isomorphic events, but there is no structural significance to such multiple perfectly synchronized events, which can just as well be understood as a single event.

Hence in modeling both types and processes we shall restrict to biextensional Chu spaces.

Let $\mathcal{A} = (A, r, X)$, $\mathcal{B} = (B, s, Y)$ be two Chu spaces. A *Chu transform* $(f, \bar{f})$ from $\mathcal{A}$ to $\mathcal{B}$ consists of two functions $f : A \to B$ and $\bar{f} : Y \to X$ such that for all $a \in A$ and $y \in Y$, $s(f(a), y) = r(a, \bar{f}(y))$, the *adjointness* or *continuity* condition for Chu spaces. The evident composition is given by $(g, \bar{g})(f, \bar{f}) = (gf, \bar{f}\bar{g})$, which is a Chu transform by the reasoning $t(gf(a), z) = s(f(a), \bar{g}(z)) = r(a, \bar{f}\bar{g}(z))$.

Chu spaces over $K$ and their Chu transforms so composed form a category denoted $\mathbf{Chu}(\mathbf{Set}, K)$. The full subcategory consisting of the biextensional Chu spaces is denoted $\mathbf{chu}(\mathbf{Set}, K)$. The latter category is where we shall be working.

The following easily proved proposition is useful at this juncture.

**Proposition 6.1** *A Chu transform $(f, \bar{f})$ from an extensional space is uniquely determined by $f$. If $(f, \bar{f})$ goes to a separated space then it is uniquely determined by $\bar{f}$.*

We define a forgetful functor $U : \mathbf{Chu}(\mathbf{Set}, K) \to \mathbf{Set}$ via $U(A, r, X) = A$ and $U(f, \bar{f}) = f$. This functor is not faithful, since $f$ need not uniquely determine $\bar{f}$. However its restriction to $\mathbf{chu}(\mathbf{Set}, K)$ is faithful, making the category we shall be working in concrete.

# 7 Chu Algebra

We subsume the common algebra of types and processes under the following algebra of Chu spaces.

In the following, the Chu spaces $\mathcal{A}$ and $\mathcal{B}$ denote respectively $(A, r, X)$ and $(B, s, Y)$. We write $A + B$ for the disjoint union $A \times \{0\} \cup B \times \{1\}$ of sets $A$ and $B$, and $A \times B$ for their cartesian product.

*Dual* The dual $\mathcal{A}^\perp$ of $\mathcal{A}$ is defined as $(X, r^\smile, A)$ where $r^\smile(x, a) = r(a, x)$.

The symmetry of the adjointness implies that if $(f, \bar{f})$ is a Chu transform from $\mathcal{A}$ to $\mathcal{B}$, then $(\bar{f}, f)$ is a Chu transform from $\mathcal{B}^\perp$ to $\mathcal{A}^\perp$. This shows that $\mathbf{Chu}(\mathbf{Set}, K)$ is a self-dual category, being equivalent (in fact isomorphic) to its opposite $\mathbf{Chu}(\mathbf{Set}, K)^{\mathrm{op}}$.

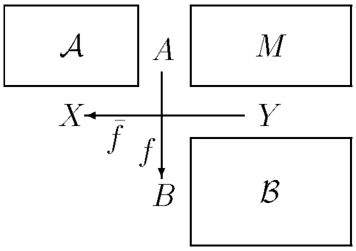
The dual of a biextensional Chu space is biextensional, whence $\mathbf{chu}(\mathbf{Set}, K)$ is also self-dual.

*Sum* The sum $\mathcal{A} + \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ is defined as $(A + B, t, X \times Y)$ where $t(a, (x, y)) = r(a, x)$ and $t(b, (x, y)) = s(b, y)$.

*Product* The product $\mathcal{A} \times \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ is defined as $(A \times B, t, X + Y)$ where $t((a, b), x) = r(a, x)$ and $t((a, b), y) = s(b, y)$.

Product is easily seen to be just the De Morgan dual of sum, satisfying $(\mathcal{A} + \mathcal{B})^\perp = \mathcal{A}^\perp \times \mathcal{B}^\perp$.

*Internal Hom* Internal hom $\mathcal{A} \multimap \mathcal{B}$ is defined as $(F, t, A \times Y)$ where $F$ is the set of all Chu transforms from $\mathcal{A}$ to $\mathcal{B}$ and $t(f, (a, y)) = s(f(a), y)$. This definition can be understood in terms of the rows of Chu spaces as representations of their points. The function $f : A \to B$ has a very natural representation as the "list" of its values at the elements of $A$. These values are taken in $B$, and hence as representations are rows indexed by $Y$. Thus $f$ can be represented in this way as an $A \times Y$ matrix $M$ whose entry $m(a, y)$ is the $y$-th element of the representation in $\mathcal{B}$ of $f(a)$. This is usefully depicted as follows.




The rows of $M$, namely the representations of the $f(a)$'s in $\mathcal{B}$, are drawn from the rows of $\mathcal{B}$ according to $f$. At the same time the columns of $M$ are drawn from the columns of $\mathcal{A}$ according to $\bar{f}$. The agreement of these two matrices, both of which must be $M$, is just the content of the adjointness or continuity condition for Chu transforms. This picture makes clear not only the workings of internal hom but also the self-duality of Chu.

Tensor product $\mathcal{A}\otimes\mathcal{B}$ as the left adjoint of internal hom turns out to be easily obtained as $\mathcal{A}\otimes\mathcal{B} = (\mathcal{A}\multimap\mathcal{B}^\perp)^\perp$. In the above diagram, transpose $\mathcal{B}$ so that $M$ becomes an $A \times B$ matrix. The points of $A \otimes B$ are then the entries (as locations, not their contents) of $M$, while its states are the possible choices of $M$, represented as the matrix $M$.

## 8 Chu spaces for types and processes

The general idea is as follows. On the one hand Chu spaces constitute a universal class of objects, by arguments given in detail elsewhere and summarized briefly in the next section. First they properly subsume the extant methods of constructing data types by both algebraic and topological means. Second, despite having an apparently fixed notion of morphism, they are as universal as arbitrary small categories. Thus as a universal source of structure with which to equip any type, they are ideally suited to type theory.

On the other hand, Chu spaces over 2 are a natural generalization of event structures, as proposed in [10] and developed in more length in Gupta's thesis [9] and by van Glabbeek and Plotkin [27]. To represent the event structure $(A, \leq, \#)$, simply take it to be the Chu space $(A, r, X)$ where $X$ consists of all order ideals of $(A, \leq)$ that do not contain both $a$ and $b$ when $a\#b$. The states of this Chu space are then exactly those configurations (sets of events) that satisfy the constraints $\leq$ and $\#$.

Although the application of Chu spaces to processes has tended to focus on $K = 2$, extending to larger $K$ would greatly enhance the process modeling capacity of the approach. At $K = 3$ it becomes possible to model mutual exclusion, and by taking $K$ to be the reals one can model real time behavior. Chu spaces are therefore well suited to the modeling of concurrency.

But it is not just the objects that match up in this way, but also some of their operations, specifically sum $A + B$ and tensor product $A \otimes B$. These are important operations for data types. Including the operation $A^\perp$ extends the algebra to the rest of multiplicative and additive linear logic, MALL, bringing in $A \times B$ as another important type operation, as well as $A \parr B$ though this operation plays a less central role in data types.

As concurrence and orthocurrence respectively, these are also important operations for processes. If we admit $A^\perp$ as an operation on processes, turning an event-oriented schedule into its corresponding state-oriented automaton, then again we obtain the whole of MALL.

## 9 Universality of Chu spaces

We now supply the promised arguments in support of the universality of Chu spaces.

Let $(A, f)$ be an algebra with carrier $A$ and one $n$-ary operation $f : A^n \to A$. Represent $(A, f)$ as the Chu space $(A, r, X)$ over $2^{n+1}$ where $r$ and $X$ are defined as follows. We take $X$ to be the set of all $n+1$-tuples $x = (x_1, \ldots, x_n, x_{n+1})$ of subsets of $A$ such that for any $n$-tuple $(a_1, \ldots, a_n)$ of elements of $A$, if $a_i \in x_i$ for all $i \leq n$ then $f(a_1, \ldots, a_n) \in x_{n+1}$. We define $r$ at $(a, x)$ to be the $n$-tuple $(a \in x_1, \ldots, a \in x_{n+1})$, for which there are $2^{n+1}$ possible values. Represent homomorphisms $h : (A, f) \to (A', f')$ between two such algebras as Chu transforms $(h, \bar{h})$ where $\bar{h}$ is the unique right adjoint to $h$, $(A, f)$ being extensional by this construction.

We have stated this for algebras, where it reads reasonably smoothly. It does however generalize to arbitrary relational structures. We have shown elsewhere [23,24] that this representation of such $n$-ary relational structures and their homomorphisms by Chu spaces over $2^n$ and their Chu transforms is a full embedding of the category of the former in that of the latter. Furthermore the embedding is concrete, meaning that the representing Chu spaces have the same carriers as the algebras they represent and transform via the same functions.

As pointed out by Lafont and Streicher [14], topological spaces can be represented as Chu spaces in the obvious way, a full embedding of **Top** in **Chu**(**Set**, 2). Combining that embedding with the above, and generalizing to structures with multiple relations and multiple sorts as per [23], demonstrates that Chu spaces can model all relational structures with and without topological structure.

However categories are considerably more general than just categories of relational structures and their homomorphisms. It is therefore natural to ask which objects of which categories $C$ are representable as Chu spaces, in the sense that $C$ can be fully embedded in Chu.

The answer is that every category $C$ whose arrows form a set $K$ (that is, every small category) embeds in **Chu**(**Set**, $K$). The representation is very simple. Represent each object $b$ of $C$ as the Chu space $(A, r, X)$ where $A$ consists of the maps $f : a \to b$ for any $a$, $X$ consists of the maps $h : b \to c$ for any $c$, and $r(f, h) = f; h$ (where $f; h$ denotes the composition $h \circ f$).

Represent each morphism $g : b \to b'$ as the pair $(\lambda f.f;g, \lambda h.g;h)$, which is a Chu transform by associativity of composition. The theorem [25] is then that this representation is a full embedding of $C$ in $\mathbf{Chu}(\mathbf{Set}, K)$. Furthermore if we take the underlying set of $b$ to consist of its elements, in the sense of maps to $b$, then this embedding is concrete (and for that matter coconcrete in the obvious dual sense). The embedding also preserves duality. On the other hand no embedding of this generality could preserve either limits or colimits.

## 10 Transformational Logic

Section 2 discussed type algebra including its logic. Algebra is justified by its logic. When the structure is based on inclusions, i.e. is posetal, the logic is the familiar one of either Boolean or intuitionistic logic, or perhaps of relation algebra. When it is based on functions, i.e. is categorical, the logic becomes the considerably less familiar categorical or transformational logic.

The categorical counterpart of inequality is ostensibly morphism. However the comparison of say $(A{\multimap}B) \otimes C$ with $A{\multimap}(B \otimes C)$ is not just a matter of giving one morphism from the former to the latter (namely the function sending $f : A \to B$ and $c \in C$ to the function $\lambda a.(f(a), c)$). This choice of morphism must vary in a natural way as $A$, $B$ and $C$ are varied (themselve by morphisms), that is, it must constitute a natural transformation.

The naturality condition must furthermore accomodate variables that may appear both covariantly and contravariantly, as in the theorem $A \otimes (A{\multimap}B){\multimap}B$. This is done by generalizing naturality to dinaturality. Dinaturality is not the strongest naturality condition, and it is sometimes desirable to strengthen dinaturality to invariance under logical relations, called logicality. We then speak of logical transformations as a subclass of dinatural transformations.

However unfamiliar, transformational logic is important as the transformational logic of types, governing how they are transformed into one another. But it is equally important as the logical basis of process transformation. Furthermore the same laws hold for process transformation as for type transformation when both are based on the same common structures as we have been proposing. This makes categorical or transformational logic all the more important.

During the past year we have been investigating the categorical logic of Chu spaces for the multiplicative fragment of linear logic, namely those operations obtainable by composition starting with $A^{\perp}$ and $A{\multimap}B$. The goal has been to identify the dinatural (or if necessary logical) transformations for this fragment, those that are suitably robust under change of variables. Recently we have shown [26] that for pairs of terms with a total of at most two occurrences of each variable, the dinatural transformations between such terms in $\mathbf{Chu}(\mathbf{Set}, 2)$ are in exact correspondence with the cut-free proofs (understood suitably abstractly) of multiplicative linear logic without MIX, a full completeness result in the sense of Abramsky and Jagadeesan [1] and Hyland and

Ong [11], but with their game semantics replaced by dinaturality semantics along the lines of Blute and Scott [4,5].

This summer with Gordon Plotkin we have been able to remove the restriction on two variables. This entailed strengthening dinaturality to logicality, necessitated by the presence of at least four dinaturals from $A \multimap A$ to itself, only one of which was accounted for by Girard's system. The others have an unnatural character that demands some such strengthening to rule them out.

A natural continuation of this work is to extend our understanding of the multiplicative fragment of linear logic to the additive fragment, and thence to the exponentials. Beyond the scope of linear logic, a more general understanding of the logical character of mathematical transformations would be highly desirable.

## 11 The Mismatches

Our types-as-processes correspondence has matched up coproduct and tensor product, along with the rest of multiplicative and additive linear logic if we include $A^\perp$. The mismatches concern the rest of type algebra and process algebra.

Here we shall look just at two other central operations of process algebra, namely choice and sequence. Both of these have natural representations as operations on Chu spaces [9].

Choice $\mathcal{A} \sqcup \mathcal{B}$ (usually $A + B$ in process algebra but we have been using $+$ for coproduct here) is defined as $(A + B, t, X + Y)$ where $t(a, x) = r(a, x)$, $t(a, y) = t(b, x) = 0$, and $t(b, y) = s(b, y)$. (This can be understood in terms of block matrices: $\mathcal{A}$ goes in the upper left, $\mathcal{B}$ in the lower right, and the rest is filled with zeros.)

Sequence $\mathcal{A}; \mathcal{B}$ is defined by first identifying in some way or other the final states of $\mathcal{A}$ and the initial states of $\mathcal{B}$. Then $\mathcal{A}; \mathcal{B} = (A + B, r, Z)$ where $Z \subseteq X \times Y$ consists of those states $(x, y)$ such that either $x$ is final in $\mathcal{A}$ or $y$ is initial in $\mathcal{B}$.

While these operations do the job for process algebra, they do not have any obvious general application in type theory. We leave as an open question whether this is due to an intrinsic difference between types and processes, or simply represents a cultural limitation, namely how we currently think about types.

## References

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

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University

Press, 1990.

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

[4] R.F. Blute and P.J. Scott. Linear Läuchli semantics. *Annals of Pure and Applied Logic*, 77:101–142, 1996.

[5] R.F. Blute and P.J. Scott. The shuffle Hopf algebra and noncommutative full completeness. *Submitted to J. Symbolic Logic*, 1996.

[6] C. Brown and D. Gurr. A categorical linear framework for Petri nets. In J. Mitchell, editor, *Logic in Computer Science*, pages 208–218. IEEE Computer Society, June 1990.

[7] C. Brown, D. Gurr, and V. de Paiva. A linear specification language for Petri nets. Technical Report DAIMI PB-363, Computer Science Department, Aarhus University, October 1991.

[8] R.T Casley, R.F. Crew, J. Meseguer, and V.R. Pratt. Temporal structures. *Math. Structures in Comp. Sci.*, 1(2):179–213, July 1991.

[9] V. Gupta. *Chu Spaces: A Model of Concurrency.* PhD thesis, Stanford University, September 1994. Tech. Report, available as ftp://boole.stanford.edu/pub/gupthes.ps.Z.

[10] V. Gupta and V.R. Pratt. Gates accept concurrent behavior. In *Proc. 34th Ann. IEEE Symp. on Foundations of Comp. Sci.*, pages 62–71, November 1993.

[11] J.M.E.
Hyland and C.-H.L. Ong. Fair games and full completeness for multiplicative linear logic without the mix-rule. Available by ftp from ftp.comlab.ox.ac.uk as fcomplete.ps.gz in /pub/Documents/techpapers/Luke.Ong, 1993.

[12] B. Jónsson and A. Tarski. Representation problems for relation algebras. *Bull. Amer. Math. Soc.*, 54:80,1192, 1948.

[13] B. Jónsson and A. Tarski. Boolean algebras with operators. Part II. *Amer. J. Math.*, 74:127–162, 1952.

[14] Y. Lafont and T. Streicher. Games semantics for linear logic. In *Proc. 6th Annual IEEE Symp. on Logic in Computer Science*, pages 43–49, Amsterdam, July 1991.

[15] G. Mackey. On infinite dimensional vector spaces. *Trans. Amer. Math. Soc.*, 57:155–207, 1945.

[16] A. Mazurkiewicz. Concurrent program schemas and their interpretation. In *Proc. Aarhus Workshop on Verification of Parallel Programs*, 1977.

[17] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[18] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress 62*, pages 386–390, Munich, 1962. North-Holland, Amsterdam.

[19] V.R. Pratt. On the composition of processes. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982.

[20] V.R. Pratt. Some constructions for order-theoretic models of concurrency. In *Proc. Conf. on Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 269–283, Brooklyn, 1985. Springer-Verlag.

[21] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[22] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.

[23] V.R. Pratt. The second calculus of binary relations. In *Proceedings of MFCS'93*, volume 711 of *Lecture Notes in Computer Science*, pages 142–155, Gdańsk, Poland, 1993. Springer-Verlag.

[24] V.R. Pratt. The Stone gamut: A coordinatization of mathematics. In *Logic in Computer Science*, pages 444–454. IEEE Computer Society, June 1995.

[25] V.R. Pratt. Broadening the denotational semantics of linear logic. In *Special Issue on Linear Logic 96*, volume 3 of *ENTCS (Electronic Notes of Theoretical Computer Science)*, Tokyo, 1996.

[26] V.R. Pratt. Towards full completeness of the linear logic of chu spaces. In *Electronic Notes in Theoretical Computer Science*, volume 6, Pittsburgh, 1997. URL: http://www.elsevier.nl/locate/entcs/volume6.html, 18 pages.

[27] R. Van Glabbeek and G. Plotkin. Configuration structures. In *Logic in Computer Science*, pages 199–209. IEEE Computer Society, June 1995.

[28] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX'88*, volume 354 of *Lecture Notes in Computer Science*, Noordwijkerhout, June 1988. Springer-Verlag.