Nordic Journal of Computing 2(1995), 181–220

A LOGICAL VIEW OF CONCURRENT CONSTRAINT PROGRAMMING

NAX P. MENDLER[*]
Department of Mathematics
University of Ottawa
Ottawa, Ontario, Canada
nmendler@csi.uottawa.ca

PRAKASH PANANGADEN[†]
School of Computer Science
McGill University
Montreal, Quebec, Canada
prakash@cs.mcgill.ca

P.J. SCOTT[‡]
Department of Mathematics
University of Ottawa
Ottawa, Ontario, Canada
scpsg@acadvm1.uottawa.ca

R.A.G. SEELY[§]
Department of Mathematics
McGill University
Montreal, Quebec, Canada
rags@math.mcgill.ca

Abstract. Concurrent Constraint Programming (CCP) has been the subject of growing interest as the focus of a new paradigm for concurrent computation. Like logic programming it claims close relations to logic. In fact CCP languages *are* logics in a certain sense that we make precise in this paper. In recent work it was shown that the denotational semantics of determinate concurrent constraint programming languages forms a fibred categorical structure called a hyperdoctrine, which is used as the basis of the categorical formulation of first-order logic. What this shows is that the combinators of determinate CCP can be viewed as logical connectives. In this paper we extend these ideas to the operational semantics of such languages and thus make available similar analogies for a much broader variety of languages including indeterminate CCP languages and concurrent block-structured imperative languages.

CR Classification: F3.1, F3.2, D1.3, D3.3

Key words: Concurrent constraint programming, simulation, logic, categories, hyperdoctrines

1. Introduction

Concurrent constraint programming (CCP) has emerged as an important paradigm within the realm of asynchronous computation, with close ties to

[*] Research supported in part by NSERC and Ontario-Quebec Bilateral Exchange, Ontario Ministry of Education.
[†] Research supported in part by NSERC and FCAR.
[‡] Research supported in part by NSERC and Ontario-Quebec Bilateral Exchange, Ontario Ministry of Education.
[§] Research supported in part by NSERC and FCAR.

Received November 1994. Accepted May 1995.

logic. Though logic and computation have always been related, the synergy between these two fields has been sharply increasing in the last decade. For example the proofs-as-programs paradigm [Girard 1989], long exemplified for functional languages by the simply-typed lambda calculus, has grown to include a whole family of typed lambda calculi, linear calculi and more recently typed process calculi, through Abramsky's work on "proofs-as-processes" and linear realizability algebras [Abramsky 1994, 1993]. However there is another relationship between logic and computation that one sees in concurrent constraint programming, and logic programming in general: not through the proofs-as-programs view but rather through programs as "proof search" [Miller 1994].

In this paper we present yet another connection between logic and computation that may be summarized by the slogan "Concurrency *is* Logic". In the CCP framework we show that one can think of processes as formulas and process combinators (*e.g.* hiding and parallel composition) as logical connectives. This correlation is not merely vague analogy; rather, concurrent constraint programs turn out to be instances of an abstract, fibred categorical presentation of first-order logic *via* the structure called a hyperdoctrine [Lawvere 1969]. It is one of the fundamental results in categorical logic that hyperdoctrines correspond to, indeed *are*, logics with quantifiers. We shall develop that viewpoint in detail for CCP.

In earlier work [Panangaden *et al.* 1993], we discovered this hyperdoctrinal framework for *determinate* CCP modeled by closure operators. At the time, it seemed this view was an interesting coincidence arising from the fact that closure operators carry a logic-like structure. However, in the present paper we show the phenomenon to be far more pervasive. First, we extend our earlier hyperdoctrinal model to include the operational semantics of CCP languages. This involves a detailed study of simulation, with particular emphasis on how CCP programs interact with the environment. We then show how to apply these results to the indeterminate case. In the final sections we indicate that key parts of our modelling extend to other concurrent languages that are not remotely like concurrent constraint programming.

Our slogan "Concurrency is Logic" involves three identifications, to be contrasted with the realizability (or proofs-as-processes) viewpoint (for example see Abramsky [1994]):

Concurrency is Logic	Proofs-as-Processes
• Processes are Formulas	• Types are Formulas
• Combinators are Connectives	• Type Constructors are Connectives
• Simulations are Proofs	• Processes are Proofs

Note the fundamental distinction: processes as formulas versus processes as proofs. The basic dichotomy being reflected here is between logic programming and functional programming. Logic programming arises from proof construction and concurrent constraint programming comes from this tradition, whereas computation in functional programming is identified as proof normalization.

On the categorical side the inspiration for our work comes from the profound insights of Lawvere [1969, 1970]. He realized that first-order logic could be elegantly captured categorically, with quantifiers interpreted as certain functors adjoint to substitution. This is to be contrasted with the *ad hoc* algebraization of first-order logic achieved by Tarski and his followers [Henkin *et al.* 1971]. We present Lawvere's ideas in expository form in Section 3. Given this abstract categorical view one can pin down precisely the relationship to logic as was done by Seely [1983]: there is a natural bijection between Lawvere hyperdocrines and first-order logics, in a precise sense familiar to categorical logicians [Lambek and Scott 1986]. Thus, these categories *are* logics. This work extends to a host of important theories: Martin-Löf type theories, polymorphic lambda calculi, linear logic, *etc.*

The categorical viewpoint permits treating syntax and semantics uniformly: a semantical interpretation is simply a morphism from a (syntactically-presented) hyperdoctrine to a "model"-hyperdoctrine. This view is illustrated in the case of the closure operator hyperdoctrine for determinate CCP in Example 3.2 and extended to a natural class of fibred categories sufficient for the operational semantics of indeterminate CCP.

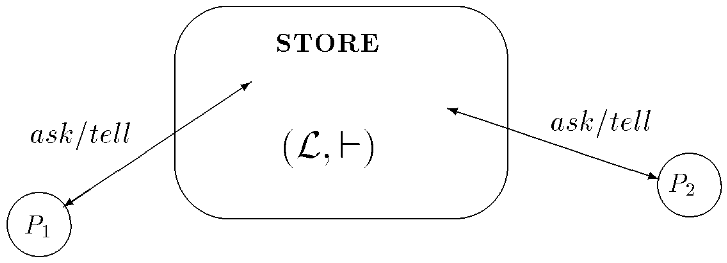
2. Concurrent Constraint Programming

In this section we give a brief summary of the denotational semantics of determinate concurrent constraint languages [Saraswat and Rinard 1990, Saraswat *et al.* 1991]. This semantics is given in terms of closure operators and leads to a very pleasant hyperdoctrine. This led to the subsequent research that forms the main thrust of the present paper. A detailed discussion of programming idioms within the paradigm of concurrent constraint programming, both determinate and indeterminate, is contained in the book by Saraswat [1993] based on his CMU dissertation.

The main point of concurrent constraint programming is, in Saraswat's words, to put "partial information in the hands of the programmer". Imagine a system consisting of concurrent processes interacting via shared data in some data base. The shared data can be thought of as a collection of assertions in some fragment of first-order logic $(\mathcal{L}, \vdash)$.

Processes communicate by adding information to the common pool of data (a "tell" operation) or by asking whether an assertion is entailed by the existing pool of data (an "ask" operation). Thus a CCP language includes:

- ∘ A data/store language $(\mathcal{L}, \vdash)$ for describing the assertions one may make (*the constraint system*).
- ∘ An ask-tell process language for describing how processes interact with the data pool.



By analogy with imperative programming we call the collection of shared assertions (or constraints) the "store". In contrast with imperative programming, however, a store might only give partial information about the values of variables. For example rather than saying that a variable has the value 1994, it might merely say that the variable lies between 1729 and 4104. The ask-tell language will be a simple process calculus, based upon the primitive notions of ask and tell. The tell operation is the mechanism for communication: it takes a constraint ϕ and adds it to the common data pool. The ask construct is the mechanism for synchronization. Given a constraint ϕ, $ask(\phi)$ succeeds or fails depending upon whether the store entails ϕ. In the former case, the process continues, in the latter case the process suspends until (if ever) more data becomes available. The "ask" construct is what gives the programmer the ability to manipulate partial information since one can query the store despite the information being only partially defined.

2.1 Constraint Systems

It is convenient to go to a more abstract level and introduce constraint systems $\langle D, \vdash \rangle$ in the style of Dana Scott's information systems [Scott 1982], the only difference being we have no notion of consistency. From this view there is a set, D, of assertions that can be made. Each assertion is a syntactically denotable object in the programming language. The set D is equipped with a (compact) entailment relation $\vdash$, by which we mean that if an assertion p is entailed by any subset of D, it must be entailed by a finite subset of D. We write $\mathcal{P}_f(D)$ for the set of *finite* subsets of D and write $\sigma \subseteq_f \tau$ to mean that σ is a finite subset of τ. This leads us to:

DEFINITION 2.1. *A* **constraint system** *is a structure* $\langle D, \vdash \rangle$*, where* D *is a non-empty (countable) set of* **assertions** *or* **(primitive) constraints** *and* $\vdash \subset \mathcal{P}_f(D) \times D$ *is a relation, called the* **entailment relation** *satisfying:*
C1 $\sigma \vdash p$ *if* $p \in \sigma$
C2 $\sigma \vdash q$ *if* $\sigma \vdash p$ *for all* $p \in \tau$ *and* $\tau \vdash q$.

We extend $\vdash$ to a relation on $\mathcal{P}_f(D) \times \mathcal{P}_f(D)$ by defining:

$$\sigma \vdash \tau \quad \text{iff} \quad \sigma \vdash p \text{ for all } p \in \tau$$

In this notation, **C2** becomes:

C2′ $\sigma \vdash q$ if $\sigma \vdash \tau$ and $\tau \vdash q$.

A constraint system is propositional in the sense that its assertions or constraints are quantifier free. Indeed, a canonical example of a constraint system is any quantifier-free fragment of a first-order language, equipped with the usual notion of entailment. However processes will contain constraints, and thus involve local variables; in this sense propositions effectively get existentially quantified. Thus in our discussion we will talk about existential quantification even though the language of assertions does not have explicit quantifiers.

The store is equipped with a query-answering system that answers entailment queries. The exact mechanism of this subsystem is a parameter of our theory and is abstracted out of the discussion. Thus the issues of how to resolve constraints efficiently are orthogonal to our concerns.

Definition 2.2. *The* **elements** *of a constraint system $\langle D, \vdash \rangle$ are those subsets σ of D such that $p \in \sigma$ whenever $\tau \subseteq_f \sigma$ and $\tau \vdash p$. The set of all such elements is denoted by $|D|$.*

A store is thought of as its entailment closure, so the denotations of stores are precisely elements of the constraint system. As is well known, $(|D|, \subseteq)$ is a complete algebraic lattice; the top element represents the inconsistent store.

2.2 Ask-Tell Languages

The following discussion in this subsection is a condensation of the discussion in Saraswat *et al.* [1991]. The syntax and operational semantics of the language are given in Table I. We assume given a constraint system $(D, \vdash)$. We use the letter σ to stand for an element of the constraint system. The basic combinators are the **ask** and **tell** written $ask(\sigma) \rightarrow P$ and $tell(\sigma)$ respectively. Intuitively, $ask(\sigma) \rightarrow P$ executes by asking the store whether σ holds, if it does then P executes, otherwise the process suspends; $tell(\sigma)$ adds σ to the constraints already in D. We shall assume that there is a process NIL with no transitions. As far as the effect on the store is concerned, NIL could be simulated by a process like $tell(\mathsf{true})$ which does have a transition but makes no difference to the store.

In the description of the operational semantics we have transitions between processes. The transitions carry labels that describe the store before and after the transition. The transition rules should be self evident; the guarded choice makes the language indeterminate. The only point that needs explanation is the notion of a *fresh* variable. When we say that a variable is fresh we mean that it is completely new, *i.e.* it is not used as a local variable by any other process and certainly does not occur as a global variable. We assume that bound variables can be changed at will (α-conversion) to

Syntax.

$$P ::= NIL \mid tell(\sigma) \mid ask(\sigma) \rightarrow P \mid P \parallel P \mid \nu x.P \mid \textstyle\sum_{i=1}^{n} ask(\sigma_i) \rightarrow P_i$$

Operational Semantics.

Tell	$tell(\sigma) \xrightarrow{(\tau, \sigma \wedge \tau)} NIL$
Ask	$(ask(\sigma) \rightarrow P) \xrightarrow{(\tau,\tau)} P$ if $\tau \vdash \sigma$
Parallel	$\dfrac{P \xrightarrow{(\sigma,\tau)} P'}{P \parallel Q \xrightarrow{(\sigma,\tau)} P' \parallel Q}$ $\quad\dfrac{P \xrightarrow{(\sigma,\tau)} P'}{Q \parallel P \xrightarrow{(\sigma,\tau)} Q \parallel P'}$
Hiding	$\nu x.P \xrightarrow{(\sigma,\sigma)} P$, where x is fresh.
Guarded Choice	$[\sum_{i=1}^{n} ask(\sigma_i) \rightarrow P_i] \xrightarrow{(\sigma,\sigma)} P_j$, where $\sigma \vdash \sigma_j$

Above, σ, τ range over assertions in some constraint system $(D, \vdash)$ while P and Q range over processes. The notation $P \xrightarrow{(\sigma,\tau)} Q$ means that P with store σ becomes Q with store τ.

TABLE I: Operational semantics for Ask-and-Tell cc languages

ensure this. Thus in particular two parallel components have no local variables in common. This is how the term, "fresh" is used in the semantics of block-structured languages and in ordinary logic or the lambda calculus.

A more careful presentation of the operational semantics of hiding follows. We assume that the store contains a list of "private" variables. These are the variables that appear as the result of hiding. Any information pertaining to these variables is available only to the process that created this private variable. In particular the environemnt cannot see any of these private variables. Existential quantification provides the precise notion of "hiding the information". Thus if $\vec{u}$ are the private variables in a store σ the globally visible part of the store is $\exists\vec{u}.\sigma$. In most of the discussion we will suppress this explicit mention of the notion of private variables and simply use existential quantification to capture the visible part of the store.

In earlier presentations of the operational semantics we used the following presentation of this rule:

$$\frac{A \xrightarrow{(\exists_x \sigma, \tau)} B}{\nu x.A \xrightarrow{(\sigma, \sigma \wedge \exists_x \tau)} \nu x.(\tau, B)}$$

The intuitive explanation is as follows. Suppose that we have $\nu x.A$ in the store σ. Now A cannot see any references to the x in the store σ hence we have to consider the transitions of A in the store $\exists_x.\sigma$ as is shown above the line. The resulting store τ may contain references to x but this x cannot be visible to the global environment and hence the global store seen is $\sigma \wedge \exists_x.\tau$, the σ is back precisely because the x referred to in it *is* visible to the environment. Finally the information in τ needs to be still available to B so we have the notion of a "private store" for B represented by the pair (τ, B).

This explanation is of couse filled with the explicit discussion of variables and scoping that the notion of "fresh" variables avoids. In the present version we would say simply that the store is τ but that x is a "private variable" and hence the visible part of the store is $\exists_x.\tau$. We need not existentially quantify the original store σ since x is a variable that has never appeared before. We also assume that alpha conversion of bound variables can be freely done as needed. We end the discussion with a tiny example. Consider the process $\nu x.tell(x = 1)$ in the initially empty store. It does the hiding step generating the fresh variable x, henceforth designated private (and used). Then it adds the formula $x = 1$ to the store. The resulting store is $x = 1$ with x designated private. The part of the store visible to other processes (and the environment) is thus $\exists_x.x = 1$ which is logically equivalent to the trivial proposition **true**. Thus in terms of visible effect this process is the same as NIL.

The denotational semantics in Table II refers to the determinate fragment. The basic idea of this semantics is to model processes as certain special functions, closure operators, acting on stores. In order to model a process *compositionally*, it suffices to record its resting points. Mathematically, this is mirrored by the fact that a closure operator is completely specified by its set of fixed points. Given this representation of closure operators, we can define some operations on sets of fixed points that are clumsy to state in terms of closure operators *qua* functions. Most notably, one can define intersection of sets of fixed points of closure operators. It is quite awkward to write down this combinator in terms of functions; roughly speaking it describes "interleaving". It turns out that this operation is exactly what one needs to model parallel composition.

The motivation for using closure operators is as follows. A closure operator is extensive (increasing), which reflects the fact that the processes add information to the store. A closure operator is also idempotent, reflecting the intuition that classical entailment is not affected by having multiple "instances" of an assumption. Finally we require monotonicity (and continuity) for the usual computability reasons. In Table II we have left out details like the definition of the environment mechanism and procedures. Nevertheless, from the denotational semantics certain connections with logic are clear. The ask construct looks very much like an implication, the parallel construct is essentially conjunction and hiding resembles existential quantification. In the next section we discuss hyperdoctrines and make these analogies precise.

Semantic Equations.

$$\begin{aligned}
[\![tell(\sigma)]\!] &= \{\tau \in |D| \mid \tau \vdash \sigma\} \\
[\![ask(\sigma) \to P]\!] &= \{\tau \in |D| \mid \tau \vdash \sigma \Rightarrow \tau \in [\![P]\!]\} \\
[\![P \parallel Q]\!] &= \{\tau \in |D| \mid \tau \in [\![P]\!] \wedge \tau \in [\![Q]\!]\} \\
[\![\nu x.P]\!] &= \{\tau \in |D| \mid \exists \sigma \in [\![P]\!]. \exists_x \tau = \exists_x \sigma\}
\end{aligned}$$

TABLE II: Denotational Semantics for Determinate Ask and Tell cc languages

3. Introduction to hyperdoctrines

Category theory developed in the 1940's, starting from a pioneering paper of Eilenberg and Mac Lane, in response to the need to relate disparate branches of mathematics. In particular, subjects as different as algebra and topology were coming together to form algebraic topology and the need for a framework that was general enough to encompass both subjects was felt.

In the present work we have an analogous goal: to relate two seemingly disparate structures in a precise way. Categorical logic is the natural framework for this. The idea is that (1) intuitionistic first-order logic *is* a particular kind of categorical structure called a hyperdoctrine and (2) concurrent constraint programming forms an instance of this structure. This makes the case that CCP *is* logic. We would like to emphasize this point since it seems to have caused some confusion in earlier presentations of this work: a hyperdoctrine *is* a logic presented abstractly. It is not merely something that captures some "aspect" of logic, such as substitution or binding. It captures *all* aspects of the construction of proofs and even gives a more refined treatment of logic in that it talks about "equality" between proofs. The significance of hyperdoctrines being logics is that once we show that CCP is a logic, with processes playing the role of formulas, then all the apparatus of logic is immediately available for reasoning about programs. A programmer can think about programs exactly as if she were manipulating logical formulas in an elementary way.

Of course there are many kinds of logics and correspondingly many different kinds of hyperdoctrines. These vary from being very similar to ordinary first-order logic to being radically different, for example, linear logic [Girard 1987]. It will turn out that determinate CCP is exactly the fragment of intuitionistic first-order logic with existential quantification and conjunction (what are called "elementary existential doctrines" in the original treatment of Lawvere) while indeterminate CCP is a little different; it corresponds to a first-order logic with weakening and the usual rules for existential quantification and conjunction but without contraction. We now proceed to the presentation of the theory of hyperdoctrines.

In dealing with logical structures, it is often useful to be able to abstract away from the details of the syntax and from such syntactical matters as how to handle variables (free *vs* bound variables, α-conversion, variable "clashes", and the like). Various algebraic approaches to first-order logic have been developed over the years, *e.g.* cylindric and polyadic algebras [Henkin *et al.* 1971], and more recently Lawvere's fibred categories (hyperdoctrines) [Lawvere 1969, Lawvere 1970, Seely 1983]. The latter led to the development of modern categorical logic and type theory [Lambek and Scott 1986].

3.1 Basic ideas

We suppose the reader to be familiar with the basics of category theory up to the notion of adjoint functors [Lambek and Scott 1986, Mac Lane 1971]. The basic references on indexed category theory and logic are Lawvere [1970], Seely [1983], where the reader is referred for further details.

EXAMPLE 3.1. **Motivating example: the pre-ordered case**

Imagine we are in the context of a many-sorted first-order theory. The standard presentation is as follows. In describing a first-order language (FOL) one has a collection of basic sorts, a collections of individual variables with syntactically distinguishable subcollections for each sort, a family of function symbols, which includes constants if there are any and which almost always includes pairing $\langle , \rangle$, and a family of relation symbols, which almost always includes equality. One now can inductively define the terms and similarly inductively define the formulas. Finally one has a notion of entailment constructed from some given axioms and rules of inference. The resulting algebraic structure is very rich and when presented purely as a collection of equations requires a thick book to describe all the algebraic structure [Henkin *et al.* 1971].

We now give a simplified categorical treatment of this same structure. In this example we are not worrying about the structure of proofs, only about the fact that a formula is entailed by a set of other formulas; this is the sense in which the treatment is simplified. The data above can be organized in the following way. We start with the sorts, written as $U, V, W, \ldots$. We construct a "base" category consisting of the sorts. In addition to the basic sorts there are all the possible finite products, *e.g.* sorts of the form $U \times V \times U$ and so on. The terms in the FOL appear as the arrows in the base category. What arrows are there? To start with we have at least all the identity arrows and all the projections, *e.g.* arrows from $U \times V$ to U and V which project out the appropriate component. We have all the arrows required to make the products be categorical products. These include, for example, arrows, called "diagonals" and written Δ from V to $V \times V$; concretely we have $\Delta(x) = \langle x, x\rangle$. Finally we have arrows for all the function symbols in the FOL. Thus, for example, if $f : U \times V \to W$ is a function symbol with the indicated arity we include f as an arrow in the base. The base

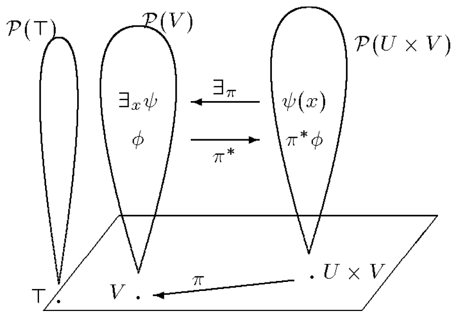


Fig. 1: Picture of a hyperdoctrine – I

is of course closed up so that all the arrows required by compositions of arrows are included. This last corresponds simply to making sure that the collection of terms in the FOL is closed under substitutions. In short the base category simply describes the usual inductive definition of the terms; the objects are the sorts and the arrows are the terms themselves.

The formulas are stratified according to the sorts of the variables that they refer to. Thus if a formula, $\phi(x, y)$, refers to two variables of sort U, V respectively, we say that ϕ "lives in the fibre over" $U \times V$. More precisely, we organize the formulas into sets indexed by the sorts of their free variables. Thus, for example, if a formula has two variables of sort U and three of sort W, it is in the set indexed by $U \times U \times W \times W \times W$. These sets of formulas are called "fibres" and the imagery is that of a set of formulas as a fibre growing out of a base object. Now the formulas in each fibre are quotiented by two-way entailment; this is well-known to logicians as the Lindenbaum-Tarski algebra and to computer scientists is reminiscent of a "term model" construction [Bell and Machover 1977]. For a theory based on classical logic, each $\mathcal{P}(U)$ is a Boolean algebra. For theories based on other logics, each $\mathcal{P}(U)$ is an appropriate poset (Heyting algebra, *etc.*). We picture the fibred structure as a family of "balloons" $\mathcal{P}(U)$ attached to the base (see Fig. 1). Note that $\mathcal{P}(\top)$ is the set of sentences (= closed formulas) of the theory, where we have written $\top$ for the terminal object of the base category.

A very important simplification has occurred in passing to this view. Essentially variables have lost their status as "names", they only keep track of the slots in the various terms or formulas. Thus we have freed ourselves from any discussion of renaming gymnastics, scoping of names, capture and other subtleties. Of course explicit variables are nice for human readability but there is no reason that the foundational account of what logic is should be burdened with the apparatus needed to talk about explicit variables.

Recall that a term t of sort U in the language of the theory is an arrow $t: U_1 \times \ldots \times U_n \longrightarrow U$, where $U_1 \times \ldots \times U_n$ describes the sorts of the free variables occurring in t. An arrow $f: V_1 \times \ldots V_m \rightarrow U_1 \times \ldots \times U_n$ is an

n-tuple of terms with m free variables. Given such a term t and a formula $\phi \in \mathcal{P}(U)$ with one free variable x of sort U, substitution of t for x defines a Boolean algebra morphism $t^*: \mathcal{P}(U) \longrightarrow \mathcal{P}(U_1 \times \ldots \times U_n) : \phi \mapsto \phi[t/x]$ *i.e.* t^* preserves the Boolean algebra structure of the fibres $\mathcal{P}(S)$. We call t^* *substitution along* t. Then $\mathcal{P}$ becomes a contravariant functor $\mathbf{B}^{op} \longrightarrow \mathbf{Bool}$ (with $\mathcal{P}(t) = t^*$).

Substitution $\pi^*: \mathcal{P}(U_1 \times \ldots \times U_n) \longrightarrow \mathcal{P}(U_1 \times \ldots \times U_n \times U_{n+1})$ takes a formula ϕ with n free variables and "adds a new dummy free variable x" to ϕ. The arrow $\pi: U_1 \times \ldots \times U_n \times U_{n+1} \longrightarrow U_1 \times \ldots \times U_n \in \mathbf{B}$ projects onto the first n components. Think of $U_1 \times \ldots \times U_n \times U_{n+1} \in \mathbf{B}$ as the list $(1, \cdots, n+1)$, denoted $\mathbf{n+1}$. Then $\pi^*: \mathcal{P}(\mathbf{n}) \longrightarrow \mathcal{P}(\mathbf{n+1})$ maps $\phi(x_1, \cdots, x_n) \mapsto \phi[\pi(x_1, \cdots, x_n, x)]$ where $\phi[\pi(x_1, \cdots, x_n, x)]$ signifies the substitution in ϕ of the $n+1$-list $\pi(x_1, \cdots, x_n, x)$ for the n-list $(x_1, \cdots, x_n)$, where x is a new variable of sort U_{n+1}.

More interesting from our point of view is what becomes of the quantifier $\exists$ (and dually $\forall$). Since a quantifier removes a free variable, x say, from a formula, clearly it induces a map $\mathcal{P}(U_1 \times \ldots \times U_n \times U_{n+1}) \longrightarrow \mathcal{P}(U_1 \times \ldots \times U_n)$. The properties of the existential (respectively universal) quantifier amount to the fact that it induces a functor left (respectively right) adjoint to the substitution functor π^* (for every possible such projection π.) That is, letting $\vdash_n$ denote entailment in $\mathcal{P}(U_1 \times \ldots \times U_n)$, we have the following bijective correspondences between entailments:

$$\frac{\exists_x \phi(\vec{x}, x) \vdash_n \psi(\vec{x})}{\phi(\vec{x}, x) \vdash_{n+1} \pi^*(\psi)} \qquad\qquad \frac{\pi^*(\phi) \vdash_{n+1} \psi(\vec{x}, x)}{\phi \vdash_n \forall_x \psi(\vec{x}, x)}$$

Writing $\exists_\pi: \mathcal{P}(U_1 \times \ldots \times U_n \times U_{n+1}) \longrightarrow \mathcal{P}(U_1 \times \ldots \times U_n)$ for the map $\phi(\vec{x}, x) \mapsto \exists_x \phi(\vec{x}, x)$ (and similarly for $\forall_\pi$), the above equivalences show that we have adjoint functors $\exists_\pi \dashv \pi^* \dashv \forall_\pi$. Finally, we shall later discuss "generalized quantifiers" $\exists_t, \forall_t$, corresponding to the adjoints to substitution along arbitrary maps t. (This analysis of the quantifiers is originally in Lawvere [1969].)

We illustrate the hyperdoctrinal structure in this case in Fig. 1 and 2; the fibres ("balloons") over a type are the formulas with free variables of that type, and the substitution and existential functors are illustrated as appropriate arrows. The map $\Delta: V \longrightarrow V \times V$ is the diagonal map which may be thought of as a (generalized) term $\langle x,\ x \rangle$ with one free variable x. $\square$

To step to a more general setting, we make some modifications in the setup above. We replace the pre-ordered (entailment-based) structure on $\mathcal{P}(X)$ (where X is just $U_1 \times \ldots \times U_n$) with more general categorical structure corresponding to the *proof-theory* of the logical theory. One should imagine the arrows of the fibres $\mathcal{P}(X)$ are (equivalence classes of) derivations of formulas with one variable of sort X. As the theories we shall consider in this paper will generally have finite conjunction (and perhaps no other

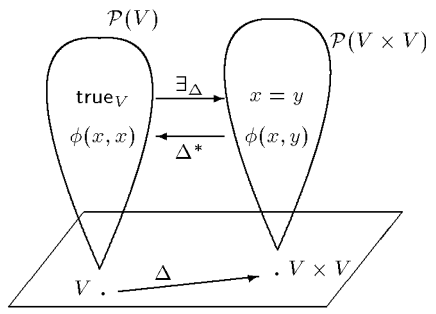


Fig. 2: Picture of a hyperdoctrine – II

logical connectives), we shall suppose the $\mathcal{P}(X)$ to be Cartesian categories. Likewise, our theories will generally only have the existential quantifier, so we will only have the functors $\exists_\pi$. This suggests the following definition. By **CCat** we mean the category of small Cartesian categories (categories with finite products and terminal objects) and Cartesian functors (functors that preserve the finite products and terminal objects on the nose.)

DEFINITION 3.1. *Suppose* $\mathbf{B}$ *is a category with finite products. A strict* $\mathbf{B}$*-based hyperdoctrine (or a hyperdoctrine over* $\mathbf{B}$*) is a contravariant functor* $\mathcal{P}: \mathbf{B}^{op} \longrightarrow \mathbf{CCat}$ *satisfying*

(i) *for any arrow* $t: X \longrightarrow Y$ *of* $\mathbf{B}$*, the functor* $\mathcal{P}(t)$ *(usually denoted* t^**)* $\mathcal{P}(Y) \longrightarrow \mathcal{P}(X)$ *has a left adjoint* $\exists_t \dashv t^*$ *(so* $\exists_t: \mathcal{P}(X) \longrightarrow \mathcal{P}(Y)$*),*

(ii) **Frobenius Reciprocity:** *for any arrow* $t: X \longrightarrow Y$ *of* $\mathbf{B}$*, and any objects* $\phi \in \mathcal{P}(X)$*,* $\psi \in \mathcal{P}(Y)$ *the canonical map* $\exists_t(t^*\psi \wedge \phi) \longrightarrow \psi \wedge \exists_t\phi$ *is an isomorphism,*

(iii) **Beck-Chevalley Condition:** *for any pullback diagram in* $\mathbf{B}$

$$\begin{tikzcd}
X \arrow[r, "t"] \arrow[d, "r"'] & Y \arrow[d, "s"] \\
X' \arrow[r, "t'"'] & Y'
\end{tikzcd}$$

and for any $\phi \in \mathcal{P}(Y)$ *the canonical morphism* $\exists_r(t^*\phi) \longrightarrow t'^*(\exists_s\phi)$ *is an isomorphism.*

The key result in Seely [1983] is that in a natural sense, every hyperdoctrine is equivalent to one derived from a logical theory. Thus a logic and a hyperdoctrine are exactly the same. The precise theorem is the following.

Th is the category of first-order $\langle \wedge, \exists, = \rangle$ theories with interpretations of theories as morphisms. (In this context we would require that interpretations of theories preserve proof-theoretic structure.) **Hyp** is the category of hyperdoctrines (over arbitrary base **B**) and morphisms of hyperdoctrines (this is defined in Seely [1983], but in fact is the obvious notion of indexed functor that preserves appropriate structure). (These are actually 2-categories, but we shall suppress this additional structure here.)

THEOREM 3.1. *The categories* **Th** *and* **Hyp** *are equivalent. In particular, a theory $\mathcal{T}$ canonically induces a hyperdoctrine $\mathcal{P}(\mathcal{T})$, and similarly a hyperdoctrine $\mathcal{P}$ canonically induces a theory $\mathcal{T}(\mathcal{P})$, in such a way that the composite correspondances produce equivalent structures:*

$$\mathcal{T} \sim \mathcal{T}(\mathcal{P}(\mathcal{T})) \qquad \mathcal{P} \sim \mathcal{P}(\mathcal{T}(\mathcal{P}))$$

The paper cited considers only the case of full first-order logic, but the method is quite general and easily extends to a whole range of logics. So a hyperdoctrine may then basically be regarded as a contravariant functor $\mathcal{P}$ from the category of sorts to a category of suitable categories: each sort X is sent to the collection $\mathcal{P}(X)$ of all formulas or "predicates" whose free variables are of that sort. $\mathcal{P}(X)$ carries the categorical structure depending on the logic, for example, a Cartesian category where products are given by conjunction. An arrow t in the category of sorts (essentially a term of the theory) is sent to the "substitution" functor t^* which substitutes the term into predicates with the appropriate free variables. These substitution functors have left adjoints given by existential quantifiers $\exists_t$. (If the logic allows universal quantification, $\forall_t$, it will be a *right* adjoint to t^*, although in this paper we concentrate on the existential quantifier). So far we have only indicated how projection substitutions π^* have these adjoints, but if the logic includes equality, this may be generalized, as we shall see below.

There are several points about this definition to make before continuing. First, by having the functor $\mathcal{P}$ map into **CCat**, we are making two requirements: first that the categories (or "fibres") $\mathcal{P}(X)$ have finite products, and second, that the functors t^* preserve this structure exactly. If the logic were to also have exponentiation, Frobenius Reciprocity would guarantee that the functors t^* would preserve that as well—as it is, in this context it guarantees that the quantifier is well-behaved with respect to quantifying formulas with dummy free variables occurring in part of the expression: $\exists_x(\psi \wedge \phi(x)) \cong \psi \wedge \exists_x \phi$ where ψ does not contain the variable x freely. The Beck-Chevalley condition continues this guarantee of proper behaviour for the quantifier by requiring that substitution preserve quantification (as we required substitution to preserve conjunction). (The full story here may be found in Seely [1983], where a precise logical characterization of the Beck-Chevalley condition is given.) The point of course is that for a hyperdoctrine derived from a logical theory, these conditions are automatically satisfied.

In addition, by introducing the "generalized" existential quantifier $\exists_t$, we are assuming that the logic has an equality predicate. To see this, let us first consider the logical interpretation of $\exists_t\phi$. If the theory includes equality, then $\exists_t\phi$ can be interpreted as $\exists_x(t(x) = y \wedge \phi(x))$, where $t\colon X \longrightarrow Y, \phi \in \mathcal{P}(X)$. On the other hand, the formula $x = y$ can be interpreted using the generalized quantifier as $\exists_\Delta \mathsf{true}_X$, where $\Delta\colon X \longrightarrow X \times X$ is the diagonal arrow $x \mapsto \langle x,\, x\rangle$ and true_X is the terminal object (corresponding to the formula "true") of $\mathcal{P}(X)$. The reader can easily check that what we have just said suggests that $\exists_\Delta \mathsf{true}_X$ is to be interpreted by the formula $\exists_z(\langle z,\, z\rangle = \langle x,\, y\rangle \wedge \mathsf{true}_X)$ which is clearly equivalent to the formula $x = y$. This is illustrated in Fig. 2. The conditions imposed on hyperdoctrines imply that this is categorically sound in general.

As pointed out in Seely [1983], for the purposes of constructing a first-order logic from a hyperdoctrine there are just three types of pullback diagrams we need to check for the Beck-Chevalley condition.

$$\begin{tikzcd}
X \arrow[r, "\langle 1_X,\, t\rangle"] \arrow[d, "t"'] & X \times Y \arrow[d, "t \times 1_Y"] \\
Y \arrow[r, "\Delta"'] & Y \times Y
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times X \arrow[r, "1_A \times t"] \arrow[d, "r \times 1_X"'] & A \times Y \arrow[d, "r \times 1_Y"] \\
B \times X \arrow[r, "1_B \times t"'] & B \times Y
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "1_X"] \arrow[d, "1_X"'] & X \arrow[d, "\Delta"] \\
X \arrow[r, "\Delta"'] & X \times X
\end{tikzcd}$$

And these three diagrams generate four isomorphisms that must be checked (the first diagram generates two by reflection about the main diagonal, and the others each generate one). In logical notation these are:

(1) $\exists_x(t(x) = y) \wedge \phi(x, t(x)) \;\cong\; \exists_x(t(x) = y) \wedge \phi(x, y)$

(2) $\exists_{x'}(\langle x',\, t(x')\rangle = \langle x,\, y\rangle) \wedge \phi(t(x')) \;\cong\; \exists_{y'}(\langle y',\, y'\rangle = \langle t(x),\, y\rangle) \wedge \phi(y')$

(3) $\exists_{x'}(x' = x) \wedge \phi(t(x')) \;\cong\; \exists_{y'}(y' = t(x)) \wedge \phi(y')$

(4) $\exists_{x'}(\langle x',\, x'\rangle = \langle x,\, x\rangle) \wedge \phi(x') \;\cong\; \phi(x)$

Depending on the type of logic one has in mind, it is possible to generalize the definition of hyperdoctrine in a number of ways. For example, we have already mentioned having universal quantification. One such generalization that is relevant for this paper is to weaken the structure imposed on the fibres, so that instead of having Cartesian products we merely require that the fibres have tensor products. (A variant of this idea, suitable for polymorphic linear logic, is outlined in Seely [1990]; the following definition is based on the notion defined there.) Because of the particular properties of the tensor product we will have in the models we consider in this paper, we will add the condition that the unit of the tensor is a terminal object, but a more general definition may easily be given if required.

Before we proceed further we will give an outline of the definition of symmetric monoidal categories.

DEFINITION 3.2. *A* **symmetric monoidal category** *is a category M with a bifunctor, usually written $.\otimes. : M \times M \longrightarrow M$ and together with an object I, called the* **unit** *for $\otimes$ and natural isomorphisms $assoc, sym, unit$ which express associativity, symmetry and unit laws for $\otimes$ and which satisfy certain coherence conditions [Mac Lane 1971].*

Examples of such categories abound in mathematics, indeed they are much more common than cartesian-closed categories. Finite dimensional vector spaces, Banach spaces, semilattices, and complete partial orders with strict continuous functions are examples of symmetric monoidal categories.

For us the main reason to be interested in them is that for indeterminate CCP it turns out that we have a tensor product rather than a product. However the tensor we end up with has many of the features of a product, for example it has projections, which means that we have a tensor product that shares many of the logical features of conjunction.

By $\mathbf{MCat}_1$ we mean the category of small symmetric monoidal categories with terminal objects satisfying the property that the terminal object is also the unit for the tensor product, and functors that preserve the tensor products and terminal objects on the nose.

DEFINITION 3.3. *Suppose* $\mathbf{B}$ *is a category with finite products. A strict* $\mathbf{B}$*-based $\otimes$-hyperdoctrine (or a $\otimes$-hyperdoctrine over* $\mathbf{B}$*) is a contravariant functor $\mathcal{P}: \mathbf{B}^{op} \longrightarrow \mathbf{MCat}_1$ satisfying*

(i) *for any arrow $t: X \longrightarrow Y$ of* $\mathbf{B}$*, the functor $\mathcal{P}(t)$ (usually denoted t^*) $\mathcal{P}(Y) \longrightarrow \mathcal{P}(X)$ has a left adjoint $\exists_t \dashv t^*$ (so $\exists_t: \mathcal{P}(X) \longrightarrow \mathcal{P}(Y)$),*

(ii) **Frobenius Reciprocity:** *for any arrow $t: X \longrightarrow Y$ of* $\mathbf{B}$*, and any objects $\phi \in \mathcal{P}(X)$, $\psi \in \mathcal{P}(Y)$ the canonical map $\exists_t(t^*\psi \otimes \phi) \longrightarrow \psi \otimes \exists_t \phi$ is an isomorphism,*

(iii) **Beck-Chevalley Condition:** *for any pullback diagram in* $\mathbf{B}$

$$\begin{tikzcd}
X \arrow[r, "t"] \arrow[d, "r"'] & Y \arrow[d, "s"] \\
X' \arrow[r, "t'"'] & Y'
\end{tikzcd}$$

and for any $\phi \in \mathcal{P}(Y)$ the canonical morphism $\exists_r(t^*\phi) \longrightarrow t'^*(\exists_s \phi)$ is an isomorphism.

The only change here is that we have substituted the tensor $\otimes$ for the product $\wedge$ in the fibres. This does affect the statement of Frobenius, but makes little other change. In particular, we have a direct analogue to Theorem 3.1: we replace **Th** with the category **ATh** of theories in the variant of (affine) logic which has dropped the structure rule of contraction, and whose logical rules for the product $\otimes$ are the same as the tensor rules in linear logic. Similarly we must replace **Hyp** with the category $\otimes$-**Hyp** of $\otimes$-hyperdoctrines.

THEOREM 3.2. *The categories* **ATh** *and* $\otimes$-**Hyp** *are equivalent.*

We could generalise this definition by weakening the requirement that t^* preserve the tensor exactly by asking that t^* be a *closed* functor, in the sense that there is a natural map (transformation) $t^*\psi \otimes t^*\phi \longrightarrow t^*(\psi \otimes \phi)$. With this, the canonical map $\exists_t(t^*\psi \otimes \phi) \longrightarrow \psi \otimes \exists_t\phi$ may still be defined, and one could even ask that it be an isomorphism—the logical significance of this generalization is still unclear, and as it is not illustrated by our examples, we shall not dwell on it further. Another generalization of the structure of hyperdoctrines involves weakening the functorial structure of $\mathcal{P}$. Again, as this is not needed for this paper, we shall not consider the resulting notion of *fibration.*

EXAMPLE 3.2. **The closure operator hyperdoctrine**
In Panangaden *et al.* [1993] it was shown that the closure operator model of determinate concurrent constraint programming induces a hyperdoctrine, in such a way that makes precise the analogies with logic suggested by the structure of the model. Moreover, this hyperdoctrine arises naturally from the underlying logic of the constraint system. First, from the discussion of Example 3.1 it is clear how to get a hyperdoctrine $\mathcal{P}\colon \mathbf{B}^{op} \longrightarrow \wedge$-**Preord** with $\wedge$-preordered fibres, where the order is given by $\vdash$, from the underlying constraint system, as given by Table I. We shall denote by **B** the category of sorts and terms for the underlying language—as in Example 3.1, by $\wedge$-**Preord** the category of $\wedge$-preorders with monotone functions, by **CAL** the category of complete algebraic lattices with monotone maps, and by $\mathbf{CAL}^{adj}$ the category of complete algebraic lattices with adjoint pairs of monotone maps as morphisms (the direction of the morphism being given by the left adjoint). We use the following constructions (functors) to obtain the closure operator hyperdoctrine from this (see Panangaden *et al.* [1993] for the details). First, given any $\wedge$-preorder P, let $\mathcal{C}(P)$ be the complete algebraic lattice of closure operators (*i.e.* monotone, idempotent, and increasing endomorphisms) on the (complete algebraic) lattice $\mathcal{F}(P)$ of filters (*i.e.* up-closed, finite $\wedge$-closed subsets ordered by inclusion) in P. Closure operators do form a complete algebraic lattice under the extensional order (or equivalently, under the order of reverse inclusion of their sets of fixed points). The idea now is to apply $\mathcal{C}$ to fibres $\mathcal{P}(U)$ of the constraint hyperdoctrine; since we are concerned to preserve the structure of the adjunctions $\exists_t \vdash t^*$, it is perhaps not surprising that for technical reasons it is necessary to define $\mathcal{C}$ on maps so as to give adjoint pairs. So given a monotone map $t\colon P \longrightarrow Q$ in $\wedge$-**Preord** we define $\mathcal{C}(t)$ to be the adjoint pair given by the

following construction; to see that it is adjoint, and for further details, see Panangaden *et al.* [1993]. t induces an adjoint pair of maps

$$\begin{tikzcd}\mathcal{F}(P) \arrow[r, shift left, "t^{\uparrow}"] & \mathcal{F}(Q) \arrow[l, shift left, "t^{-1}"]\end{tikzcd}$$

where $t^{\uparrow}$ is the map $u \mapsto$the filter generated by the direct image of u under t, and t^{-1} is given by inverse image. From such an adjoint pair, we can construct an adjoint pair

$$\begin{tikzcd}\mathcal{C}(P) \arrow[r, shift left, "t^{\sharp}"] & \mathcal{C}(Q) \arrow[l, shift left, "t^{\flat}"]\end{tikzcd}$$

by simple composition. We use the fact that any monotone endomorphism on a complete algebraic lattice generates a closure operator—indeed there is a least closure operator extending the given map. So $t^{\sharp}$ maps a closure operator $c \in \mathcal{C}(P) : \mathcal{F}(P) \longrightarrow \mathcal{F}(P)$ to the closure operator generated by $t^{-1}; c; t^{\uparrow}$ and $t^{\flat}$ maps a closure operator $d \in \mathcal{C}(Q) : \mathcal{F}(Q) \longrightarrow \mathcal{F}(Q)$ to the closure operator generated by $t^{\uparrow}; d; t^{-1}$. Composing the functor $\mathcal{C}$ with the original hyperdoctrine and forgetting the adjoint pairs then gives the closure operator hyperdoctrine $\mathcal{CP}\colon \mathbf{B}^{op} \longrightarrow \mathbf{CAL}$. The existence of the adjoint pairs guarantees that this has the required existential adjoint structure, and it is easy to see that the existential quantifier in the logic of the hyperdoctrine is given by the hiding (ν) operator. The parallel construct is the product in the fibres, and so is indeed "conjunction" in this logic.

4. The Simulation Preorder

In this section we begin the extension of the preceding theory to the indeterminate case. Of course the denotational semantics is no longer given by closure operators but, rather, by sequences akin to failures. Rather than use the denotational semantics, we will develop the connection between logic and concurrent constraint processes by working with the operational semantics. In the determinate case we get a good match with the traditional notion of hyperdoctrine but in the indeterminate case we will end up with a $\otimes$-hyperdoctrine.

As before we construct a hyperdoctrine from the processes but we need to have a notion of morphism, or at least preorder, between processes. It turns out that the "right" notion is simulation. We develop two, related, notions of simulation, one based on a morphism and one based on the derived preorder. This leads to theories that are similar with respect to the relationship with logic but quite different with respect to their treatment of process equivalence. This section concentrates on the preorder version. We omit proofs here since they are all corollaries of proofs for the simulation morphism case discussed in the next section. In this section we define the simulation preorder but it should be clear that we are really defining a notion of simulation morphism and then deriving the preorder by collapsing simulation morphisms. Of course a similar remark can be made about the notion of simulation in traditional process algebra.

The basic idea of simulation ultimately derives from the notion of bisimulation introduced by Milner and Park in the context of CCS [Milner 1989] and put into elegant and general form by several authors, especially Joyal *et al.* [1993]; see also the article by Nielsen and Winskel [1994]. However, there is an important subtlety in our theory. The labels on transitions are not uninterpreted tokens, furthermore there are nontrivial relations between labels which affects the notion of simulation. Furthermore the notion of simulation has to capture the idea that the store has a visible part corresponding to the global variables. Once the notion of simulation is in place the proofs are not difficult, however, considerable care must be exercised in coming up with this definition. Indeed the bulk of the time was spent exploring several possible notions of simulation before we arrived at the notion described below.

A program interacts with the environment and makes moves in response to the environment. A simulation relation must capture this interaction. Thus one has to imitate not just the process moves but also the environment moves. In the present context we model this by considering sequences of moves made by a process interspersed by moves made by the environment. This kind of structure was used in the denotational semantics of the indeterminate concurrent constraint programming language in Saraswat *et al.* [1991] and by various authors [Brookes 1993, Horita *et al.* 1994] in modeling imperative concurrent languages. In the present case the information content of the store can only be increased, thus the sequences take a particularly simple form.

We need to ensure that enough of the effect of environment actions are "remembered", this is conveniently done by using the store. We need, therefore, to work with pairs consisting of a process and a store rather than with a process in isolation. On the other hand we want simulation to capture the possible behaviours of processes in all possible stores, thus we need to somehow "quantify" over the possible stores. First we define what it means for a process-store pair to simulate another such pair, then we define simulation between processes. All the complexity of the following definition arises from the fact that we are looking closely at the internal moves, they are not just dismissed as so-called τ moves, *i.e.* hidden internal moves. The transition labels are interpreted as indicating updates to the store. When referring to a transition given by the operational semantics we will use pairs of stores as labels. We write an arrow with a $*$ below it to indicate a finite sequences of transitions defined by the operational semantics.

DEFINITION 4.1. *We say that* (P,σ) **simulates** (Q,τ), *written* $(P,\sigma) \preceq (Q,\tau)$, *if (i)* $\sigma \vdash \tau$ *and (ii) whenever* $Q \xrightarrow{(\tau',\tau'')} Q'$, *where* $\tau' = \tau \wedge \phi$ *and* ϕ *is a formula only involving the global variables, then for some* P', σ', σ'' *we have* $P \xrightarrow{(\sigma',\sigma'')}_{*} P'$, *with* $\sigma' = \sigma \wedge \phi$, $\sigma'' \vdash \tau''$ *and such that* $(P',\sigma'') \preceq (Q',\tau'')$.

We say, P simulates Q if (P, true) simulates (Q, true). If $P \preceq Q$ and $Q \preceq P$, we say that P and Q are **similar** *and write $P \sim Q$.*

There are several points that need to be made. The stores are equipped with a notion of private variables that were introduced by processes of the form $\nu x.P$ making hiding steps. When we write $\sigma \vdash \tau$ we mean that references to information regarding private variables has been removed by existential quantification.

The store is the basic observable, thus, the first requirement of simulation is that the simulating store entail the store being simulated. Second, the environment may upgrade the store before the processes under consideration make any moves. Thus, in the above, we allow the simulated process to make a move starting, not in the original store τ, but in a (possibly) upgraded store τ'. The discrepancy between τ and τ' represents the effect of the environment on the store. The environment cannot, however, make an arbitrary change to the store; it can merely *add* information about the *global variables* of the processes. Hence the requirement that $\tau' = \tau \wedge \phi$, where ϕ only involves the global variables. The simulating process can only be expected to simulate successfully if the environment does exactly the same thing in each case, hence the requirement on σ'. Finally after the process being simulated makes its move and the simulating process makes the moves, possibly many, needed for the simulation, the resulting process-store combinations continue to stand in the simulation relation. The fact that we allow the simulating process to take several steps means that we are working with a notion analogous to "weak" bisimulation. As a result of the last requirement the definition is given in so-called "co-inductive" form which can be interpreted in the same way as is traditionally done for bisimulation, *i.e.* by unwinding the definition as Milner's original treatment did or by using fixed points as in Park's formulation.

Before discussing the doctrinal structures that arise it is worth looking at some attempts to define simulation that do not work. One could have tried to define simulation between processes by saying that the simulating process has to mimic the behaviour of the simulated process in any store. If one does this then the co-inductive definition becomes strange. Suppose that P simulates Q and Q makes a step in store τ to become Q' in store τ' and that P mimics this in τ to become τ''. Now is P' supposed to simulate Q' in any store? If so then the effect that P had on the store is forgotten and many intuitive instances of the simulation relation fail to hold. What if we just stuck to process-store pairs? Then when we attempt to construct existential quantification we have problems with Frobenius Reciprocity. Roughly speaking the situation is this. Suppose that we had (P, σ), when we perform existential quantification we would naturally try $(\nu x.P, \exists_x.\sigma)$, but now the correspondence between the x in the process and the x in the store is broken. Thus we need to factor out the effect of stores in order for existential quantification to work properly but we need to keep the stores around in order for basic examples to work. As will be seen in the next section, what we are really doing is defining a labelled transition system and using the notion of a morphism between labelled transition systems. The states of the labelled transition system encode the information in the stores, including information added by the external environment, but the initial states allow one to speak of processes without tying them down to specific stores. The examples below will clarify the situation.

EXAMPLE 4.1.

$$P : ask(z = 0) \rightarrow [tell((x = 1) \wedge (y = 1))]$$

$$Q : ask(z = 0) \rightarrow [tell(x = 1) \parallel tell(y = 1)]$$

Here P upgrades the store in one step while Q does the same thing but in two steps. These can simulate each other. This example shows why we need to record the contribution of the stores in the definition of simulation. If we could only keep track of corresponding steps then P could not simulate the second step of Q.

EXAMPLE 4.2.

$$\begin{array}{rlcl} P: & [ask(x = 1) & \rightarrow & tell(y = 2) + tell(z = 3)] \\ & +[ask(x = 1) & \rightarrow & tell(y = 2)] \\ & +[ask(x = 1) & \rightarrow & tell(z = 3)] \end{array}$$

$$Q : ask(x = 1) \quad \rightarrow \quad [tell(y = 2) + tell(z = 3)]$$

Here the $+$ in Q and inside the first clause of P is shorthand for the guarded choice with trivial guards. In this example P and Q can make the same choices but they are made at different "times"; these two processes can simulate each other. This is the concurrent constraint programming version of the famous CCS example of two processes that are trace equivalent but not bisimilar. We have not defined bisimilarity yet but it should be more or less clear what it would be and that these processes are not bisimilar. Thus two-way simulation does not imply bisimilarity.

We are ready to discuss the hyperdoctrine structure now. The base category is, as before, the category $\mathbf{B}$, which was used in the closure operator hyperdoctrine. Each fibre consists of processes with free variables of the sorts given by the base object. Thus a process Q belongs to some fibre, each fibre defines a set of free variables and the free variables of Q are required to be contained in the set of free variables associated with its fibre. The stores may of course contain other private variables but they are hidden from the environment. The fibres are preordered by the simulation preorder. The following propositions summarize the situation. We note the following proposition which has a routine proof.

PROPOSITION 4.1. *The simulation relation is a preorder.*

This justifies our writing $P \preceq Q$ if P simulates Q.

The next few propositions establish the structure of the fibres. We begin with the trivial observation that each fibre has terminal objects, given by NIL. Note that NIL is similar to $tell(\mathsf{true})$ or $ask(\mathsf{false}) \rightarrow tell(\phi)$, for instance. The next proposition shows that we have a tensor product in the fibres and thus that we are ready to begin showing that we have a $\otimes$-hyperdoctrine.

PROPOSITION 4.2. *If $P \preceq Q$ and $P' \preceq Q'$ then $[P \parallel P'] \preceq [Q \parallel Q']$.*

This means that the parallel composition is a bifunctor and hence a suitable candidate for a tensor product. It is worth noting that NIL, in addition to being terminal, is also the unit for the tensor product as the following obvious proposition states.

PROPOSITION 4.3. *For any process P we have $P \sim NIL \parallel P$.*

In fact parallel composition is more than just a tensor, it has projections as well as the following obvious proposition shows.

PROPOSITION 4.4. *$P \parallel Q \preceq P$ and $P \parallel Q \preceq Q$.*

Given a tensor, we can now show that "ask" is a limited form of implication. This proposition suggests that the parallel composition is the "correct" tensor to use in the present theory.

PROPOSITION 4.5. *The following adjunction tableau holds:*

$$\frac{[P \parallel tell(\phi)] \preceq \;\; Q}{P \preceq \;\; ask(\phi) \rightarrow Q}$$

This is, of course, not necessary to show that we have a hyperdoctrine. The proof will be given in the next section.

The next proposition is essential since it makes the fundamental connection between the local variable construct and existential quantification *via* the notion of adjunction. In order to state it properly we must first explain a point of notation. Recall that we are viewing processes as being stratified by sets of free variables. If S is an object in the base category and Q is in the fibre over S, we could also have Q be in the fibre over some other base object S' where the free variables associated with S are all included in those associated with S'. We write $\pi^*(Q)$ for the process Q viewed as living over a fibre with one more free variable. In the syntax of concurrent constraint programming there is no difference between Q and $\pi^*(Q)$, it just means that some variable, usually which one is clear from context, does not occur free in Q.

Proposition 4.6. *The following adjunction tableau holds:*

$$\frac{\nu x.P \preceq Q}{P \preceq \pi^*(Q).}$$

This is, of course the important adjunction that tells us that the concurrent constraint programming languages are equipped with existential quantification. All these adjunctions work equally well in the determinate and in the indeterminate cases.

Finally we need the Beck conditions and the Frobenius reciprocity condition. In our context, Frobenius is the following more or less obvious proposition.

Proposition 4.7. **(Frobenius Reciprocity)**

$$\nu x.[P \parallel \pi^*(Q)] \sim (\nu x.P) \parallel Q.$$

Proof. The statement of the proposition implies that x does not occur free in Q. Thus in each case the local declaration causes a fresh variable x to be in the store and Q cannot affect or be affected by any proposition involving x. Any step of one of the above processes can be literally mimicked by the other as can immediately be seen from the operational semantics. □

Essentially the Beck condition says that substitution interacts properly with existential quantification. The following propositions capture the key points. We use the standard notation $P[t/x]$ to mean the result of replacing free occurrences of x in P by the term t.

Proposition 4.8. *For any term t in the term language of the constraint system we have*

(1) $\nu x.tell(x = t) \sim NIL$

(2) $\nu x.[P \parallel tell(x = t)] \sim P$, *where P does not contain a free occurrence of x.*

Proof. The first statement is, of course, an immediate consequence of the fact that the process $\nu x.tell(x = t)$ cannot make any observable effect on the store. The second statement follows by applying Frobenius and the first statement and recalling that NIL is the tensor unit. □

Proposition 4.9. $P[t/x] \sim \nu x.[P \parallel tell(x = t)]$ *where t is assumed not to contain any free occurrence of x.*

Proof. If x is not free in P then the LHS is just P while the RHS is similar to P by the preceding proposition. Thus we can assume that x actually occurs free in P. The following argument is the outline of a proof by structural induction. A process interacts with the store in two ways, by asking or telling. Suppose that an occurrence of x in P is of the form $tell(\phi(x))$, where we have indicated the dependence on x explicitly. On the

LHS this becomes $tell(\phi(t))$, which adds the formula $\phi(t/x)$ to the store. The latter formula is logically equivalent to $\exists_x.[(x = t) \wedge \phi(x)]$. Thus as far as this transition goes the two processes can mimic each other. If the occurrence of x is of the form $ask(\phi(x)) \rightarrow P'$ then on the LHS this becomes $ask(\phi(t/x)) \rightarrow P'[t/x]$ which can make a transition only if the store entails $\phi(t/x)$. On the RHS the process has added $x = t$ to the store, so if the store on the LHS entails $\phi(t/x)$ the store on the RHS entails $\phi(x)$ since it contains the additional fact $x = t$. Conversely if the store on the right entails $\phi(x)$, in virtue of the fact that it contains $x = t$ it will immediately entail $\phi(t/x)$. Now consider any proof that the RHS store entails $\phi(x)$. The formulas used in this proof either came from the environment, in which case they do not involve x and are present in the LHS store as well, or they came from the actions of P-derivatives in which case the version with t replacing x is present in the LHS store. Thus one can construct a proof that the LHS store entails $\phi(t/x)$ by carrying out the substitution everywhere in the proof constructed on the RHS. Thus in each of these cases the two processes can mimic each other on a step by step basis. □

The Beck conditions are now consequences of this. There are four essential cases corresponding to the three types of pullback diagrams that must exist in any category with finite products (the first type of pullback gives rise to two isosimilarity conditions) [Seely 1983]. We leave the proofs as exercises for the reader.

Proposition 4.10.

(1) $\nu x.[tell(t(x) = y)] \parallel P(x, t(x)) \sim \nu x.[tell(t(x) = y)] \parallel P(x, y)$

(2) $\nu x'.[tell(\langle x',\ t(x')\rangle = \langle x,\ y\rangle)] \parallel P(t(x'))$
$\sim \nu y'.[tell(\langle y',\ y'\rangle = \langle t(x),\ y\rangle)] \parallel P(y')$

(3) $\nu x'.[tell(x' = x)] \parallel P(t(x')) \sim \nu y'.[tell(y' = t(x))] \parallel P(y')$

(4) $\nu x'.[tell(\langle x',\ x'\rangle = \langle x,\ x\rangle)] \parallel P(x') \sim P(x)$

We now discuss some features of the simulation preorder that distinguish the indeterminate language from the determinate language.

Proposition 4.11.

(1) In both the determinate and the indeterminate concurrent constraint programming languages we have $P \parallel P \preceq P$ for any P.

(2) In the determinate language $P \preceq P \parallel P$ for any P.

(3) In the indeterminate language $P \not\preceq P \parallel P$ for some P.

Proof. We give a sketch of the proofs. The first is easy. Note that $P \preceq NIL$ and $P \preceq P$ so by functoriality of parallel composition we get $P \parallel P \preceq P \parallel NIL \sim P$. Note that we used the fact that NIL is both terminal and the unit for the tensor.

To show the second part we need a pair of lemmas whose proof is straightforward and familiar, but long, and is hence omitted [Saraswat *et al.* 1991]. Note that the first is clearly not true in languages with "tell" guards.

Lemma 4.1. (Operational monotonicity) *If a transition is enabled in any store then it is enabled in any upgraded store as well.*

The following only applies to the determinate language.

Lemma 4.2. (Confluence) *In the determinate language if a process P in a store σ can execute two transition sequences leading to P', σ' and P'', σ'', it is always possible to find a process P''' and store σ''' such that $P' \xrightarrow{(\sigma', \sigma''')}_* P'''$ and $P'' \xrightarrow{(\sigma'', \sigma''')}_* P'''$.*

Now suppose that $P \parallel P$ executes a sequence of transitions ending up with the store σ and with the process $P_1 \parallel P_2$. We would like to show how to associate a pair (P', τ) with $(P_1 \parallel P_2, \sigma)$ in such a way that the conditions for a morphism are fulfilled. Because of operational monotonicity we can assume that all environment contributions occur in a single step at the beginning. Thus we can consider purely internal transitions of $(P \parallel P, \phi)$ leading eventually to $(P_1 \parallel P_2, \sigma)$. Let us assume for the moment that the transitions can be partitioned into two separate subsequences $P \xrightarrow{(\phi, \sigma_1)}_* P_1$ and $P \xrightarrow{(\phi, \sigma_2)}_* P_2$ with $\sigma = \sigma_1 \wedge \sigma_2$. Now by confluence we can find P', τ with the required properties. The only caveat is that we might not be able to partition the transition sequence of $P \parallel P$ as claimed above. This will happen only if one of the processes adds some information to the store that the other one uses to answer an ask. But since the two processes in question are both derivatives of a single determinate process the second process can add this information itself. We can use confluence to construct a pair of execution sequences $P \xrightarrow{(\phi, \sigma_1)}_* P_1'$ and $P \xrightarrow{(\phi, \sigma_2)}_* P_2'$, where P_1' and P_2' differ from their unprimed counterparts by having executed a few more transitions. One can now apply confluence to $P \xrightarrow{(\phi, \sigma_1)}_* P_1'$ and $P \xrightarrow{(\phi, \sigma_2)}_* P_2'$ to construct the required (P', τ). Since $\tau \vdash \sigma_1$ and $\tau \vdash \sigma_2$ we have $\tau \vdash \sigma$. Since this construction was based on an arbitrary finite transition sequence of $P \parallel P$ it clearly applies to all the states of the labelled transition system associated with $P \parallel P$ and describes a simulation morphism. Thus P can simulate $P \parallel P$.

For part three we note the following simple example. The point is that in the indeterminate language we do not have the confluence property and the above result is false. For example, suppose $P = [tell(x = 1) + tell(y = 1)]$, then $P \parallel P$ could produce the store $(x = 1) \wedge (y = 1)$ starting from the empty store, which P cannot produce starting from the empty store. □

An immediate corollary of part (2) of the above proposition is that parallel composition defines a product for the fibres in the determinate language.

Corollary 4.1. *In the determinate version of the concurrent constraint programming language, parallel composition defines a product.*

PROOF. We have already noted that $P \parallel Q \preceq P$ and $P \parallel Q \preceq Q$ in proposition 4.4. Suppose that $P \preceq Q$ and $P \preceq R$. Since parallel composition is tensor we have $P \parallel P$ can simulate $Q \parallel R$. Since we are looking at determinate processes only we have P can simulate $P \parallel P$. Thus by transitivity of the simulation preorder we have P can simulate $Q \parallel R$. □

On the other hand, the indeterminate language is also equipped with a product as the following proposition shows. We recall that $P + Q$ is shorthand for $ask(\mathsf{true}) \to P + ask(\mathsf{true}) \to Q$.

PROPOSITION 4.12. *If P can simulate Q and R, then P can simulate $Q+R$.*

PROOF. The only steps that $Q + R$ can make are $(Q + R) \xrightarrow{(\phi,\sigma)} Q'$ or $(Q+R) \xrightarrow{(\phi,\tau)} R'$. By assumption each of these can be simulated by P and the resulting configurations can be simulated by the resulting P-derivative as well. □

Note that $P+Q$ cannot simulate $P \parallel Q$ so $\parallel$ is not a product in the situation described by the indeterminate language; $+$ is the product while $\parallel$ is tensor product.

To summarize, one can construct a hyperdoctrine with the same base category as before and with processes as the objects in the fibres. The arrows in the fibres are instances of the simulation preorder. Parallel composition plays the role of tensor product and "ask" and "tell" are adjoints, showing that there is a limited notion of implication between processes. In the determinate language the parallel composition plays the role of Cartesian product whereas in the indeterminate language the choice construct acts as the Cartesian product.

	$\parallel$ is tensor	$\parallel$ is product	$+$ is product
Determinate CCP	Yes	Yes	No
Indeterminate CCP	Yes	No	Yes

5. Simulation Morphisms

A simulation between processes provides detailed information about how the behaviours of the two processes correspond; in particular there may be quite different ways in which the two behaviours correspond. In the last section we collapsed this information and merely recorded that a correspondence existed. In the present section we look at the more refined notion of simulation where we actually keep track of the way in which one process simulates another. We get a category of processes where the morphisms are simulations. The basic adjunction structure is still present but other logical aspects are altered. In particular the notion of isomorphism is very different as will be seen in the next section. As a result, although Frobenius Reciprocity does hold, the Beck conditions do not quite hold in full generality.

We can formalize the notion of simulation by adapting the notion of morphism between labelled transition systems defined, for example, by Nielsen and Winskel [1994]. The idea is to define a labelled transition associated with each process. We first recapitulate the (slightly modified) definition of labelled transition system. We assume that we are working with some fixed constraint system and have, therefore, a well-defined notion of formulas that can be in the stores.

Definition 5.1. *A* **labelled transition system** *is given by a quadruple* (Q, q_0, L, T)*, where* Q *is a set of states,* q_0 *is a distinguished member of* Q*, called the initial state,* L *is a set of labels and* T *is a transition relation defined as a subset of* $Q \times L \times Q$*. We write* $l{:}\, q \to q'$ *rather than* (q, l, q') *for the members of* T*. The relation* T *satisfies (i) for every state* q *there is a transition called the identity,* $i_q{:}\, q \to q$*, (ii) if* $l{:}\, q \to q'$ *and* $l{:}\, q \to q''$ *are both in* T *then* $q' = q''$*.*

Definition 5.2. *A* **constraint-labelled transition system** *(abbreviated CL-transition system) is a labelled transition system in which the states have the form of process-store pairs and the labels are formulas of the constraint system.*

Now we define a pair of CL-transition systems associated with a process.

Definition 5.3. *Let* P *be a process. We define a CL-transition system* $\mathcal{L}(P)$ *with states consisting of process-store pairs and transitions given by the operational semantics of CCP in the following way. The initial state is* (P, true)*. Suppose that* (P_1, σ) *is any state of* $\mathcal{L}(P)$ *and* $P_1 \xrightarrow{(\sigma', \sigma'')} P_2$ *is a transition given by the operational semantics, where* σ' *is any store such that* $\sigma' = \sigma \wedge \phi$*, and where* ϕ *only refers to global variables of* P*. Then there is a transition from* (P_1, σ) *to* (P_2, σ'') *labelled by* σ' *in* $\mathcal{L}(P)$*. Similarly we define the labelled transition system* $\mathcal{L}^*(P)$ *exactly as above except that we have a transition for every finite transition sequence given by the operational semantics.*

Note that the only processes are the derivatives of P. Thus, by definition, every state is reachable from the initial state.

The transition labels in $\mathcal{L}(P)$ describe the effect of the environment. We could have used pairs of stores to label the transitions of $\mathcal{L}(P)$, as we do in the operational semantics, but the second member of any such pair is the same as the store component of the target pair in the labelled transition system, so we just use the first member of the pair of stores as the label. The ϕ in the definition above represents information added to the store by the action of the environment.

Definition 5.4. *A* **morphism** *f from a CL-transition system, $\mathcal{L}(Q)$ to a CL-transition system $\mathcal{L}(P)$, is a map f_0 from the states of $\mathcal{L}(Q)$ to the states of $\mathcal{L}(P)$ and a map f_1 from transitions of $\mathcal{L}(Q)$ to the transitions of $\mathcal{L}^*(P)$ such that*

(1) $f_0((Q, \mathsf{true})) = (P, \mathsf{true})$, i.e. *$f_0$ takes the initial state to the initial state,*

(2) if $f_1(\tau' \colon (Q', \tau) \to (Q'', \tau'')) = \sigma' \colon (P', \sigma) \to (P'', \sigma''))$ *then* $f_0((Q', \tau)) = (P, \sigma)$ *and* $f_0((Q'', \tau'')) = (P'', \sigma'')$, i.e. *we have an lts morphism in the standard sense,*

(3) if $f_1(\tau' \colon (Q, \tau) \to (Q', \tau'')) = \sigma' \colon (P, \sigma) \to (P', \sigma'')$ *then* $\sigma \vdash \tau, \sigma' \vdash \tau', \sigma'' \vdash \tau''$.

The same definition can of course be used for relating CL-transition systems of the form $\mathcal{L}^*(P)$ with each other and with those of the form $\mathcal{L}(P)$.

Now we can define simulation as a morphism of labelled transition systems.

Definition 5.5. *A* **simulation** *f from P to Q is a morphism of CL-transition systems from $\mathcal{L}(Q)$ to $\mathcal{L}^*(P)$, also called f, such that if $f_0((Q', \tau)) = (P', \sigma)$, $(Q', \tau) \xrightarrow{\tau'} (Q'', \tau'')$ is a transition of $\mathcal{L}(Q)$, $f_0((Q'', \tau'')) = (P'', \sigma'')$ and $f_1(\tau') = \sigma'$ then $\sigma \vdash \tau$, $\sigma' \vdash \tau'$, $\sigma'' \vdash \tau''$ and $\sigma' = \sigma \wedge \phi$ where $\tau' = \tau \wedge \phi$.*

We ensure by the last condition in the definition of morphism that the simulating process is subject to the same environment actions as the simulated process. The other conditions ensure that the simulating process always has a stronger store than the simulated process. The key point is that we have a function f, which precisely describes the correspondence. Recall that the definition of "stronger store" implicitly takes into account that the private variables are removed by existential quantification before we compare the stores.

Convention: We will often talk about transitions and morphisms between CL-transition systems. Rather than constantly repeating that the store τ' is of the form $\tau \wedge \phi$ as above we will write an unprimed Greek letter to indicate the starting store for a transition, a singly-primed letter to indicate the store that is obtained by an environment action and a doubly primed Greek letter for the final store. Furthermore when we are talking about constructing a simulation we will not keep repeating that the label on the simulating transition (σ' above for example) records the same environment contribution.

We record a couple of obvious but important facts.

Proposition 5.1. *The collection of processes of a concurrent constraint programming language can be organized into a category* **Sim**, *the* **simulation category**, *with processes as objects and simulations as morphisms. Composition of two morphisms $f \circ g$ is given by the functions $f_0 \circ g_0$ and $f_1 \circ g_1$. The identity is given by (id_0, id_1).*

The following proposition tells us that the simulation preorder of the last section is just the preorder collapse of the category **Sim**.

PROPOSITION 5.2. *P simulates Q if and only if there is a simulation f from P to Q.*

The explication of the logical structure of concurrent constraint programming in this section closely parallels the treatment of the last section. We must however show how the simulation maps are constructed. In order to develop the hyperdoctrine with simulation morphisms we need to know that $\|$ is a tensor. First we need to relate the CL-transition system of $P \parallel Q$ with the CL-transition systems of P and Q.

PROPOSITION 5.3. *The structure of $\mathcal{L}(P \parallel Q)$ (respectively $\mathcal{L}^*(P \parallel Q)$) is of the following form. For every state of the form (P_1, σ) of $\mathcal{L}(P)$ and (Q_1, τ) of $\mathcal{L}(Q)$ there is a state $(P \parallel Q, \sigma \wedge \tau)$ of $\mathcal{L}(P \parallel Q)$. For every transition $(P_1, \sigma) \xrightarrow{\sigma'} (P_1', \sigma'')$ in $\mathcal{L}(P)$ there are transitions of the form $(P_1 \parallel Q_1, \tau) \xrightarrow{\tau'} (P_1' \parallel Q_1, \tau'')$ in $\mathcal{L}(P \parallel Q)$ for every τ and ϕ, with ϕ only referring to global variables, such that $\tau \vdash \sigma$, $\sigma' = \sigma \wedge \phi$, $\tau' = \tau \wedge \phi$. Similarly for transition of $\mathcal{L}(Q)$.*

PROOF. It is easy to prove by induction on the length of transition sequences, given by the operational semantics, that any derivative of $P \parallel Q$ has the form $P' \parallel Q'$ where P' is a derivative of P and Q' is a derivative of Q. Given the rule for transitions of a parallel composition, the required isomorphism is then easily constructed. □

We need two preliminary lemmas which can easily be proved by structural induction. The first is a slight refinement of operational monotonicity.

LEMMA 5.1. *If a transition $P \xrightarrow{(\sigma, \sigma')} P'$ is possible according to the operational semantics then for any formula ϕ it is possible to have the transition $P \xrightarrow{(\sigma \wedge \phi, \sigma' \wedge \phi)} P'$.*

The second lemma says that a process that does not have x as a free variable cannot depend on information that refers to x.

LEMMA 5.2. *Suppose that x does not occur free in the process P. If the transition $P \xrightarrow{(\sigma, \sigma \wedge \psi)} P'$ is possible then the transition $P \xrightarrow{(\exists_x.\sigma, (\exists_x.\sigma) \wedge \psi)} P'$ is also possible and ψ will have no free occurrences of x.*

Now we can state the basic construction that leads to viewing parallel composition as a tensor product.

PROPOSITION 5.4. *If $f\colon P \longrightarrow Q$ and $f'\colon P' \longrightarrow Q'$ are simulation morphisms then there is a simulation morphism, denoted by $f \otimes f'$, from $P \parallel P'$ to $Q \parallel Q'$.*

PROOF. In view of the preceding proposition, the labelled transition systems $\mathcal{L}(Q \parallel Q')$ and $\mathcal{L}^*(P \parallel P')$ can be assumed to have the structure described in the last proposition. The simulation morphism $f \otimes f'$ is constructed as follows. The CL transition systems in question all have the property that all their states are reachable from the initial state. This means that we can construct $f \otimes f'$ by induction on the length of the shortest transition sequence from the initial state.

We set $(f \otimes f')_0((Q \parallel Q', \mathsf{true})) = (P \parallel P', \mathsf{true})$. We define $f \otimes f'$ inductively as follows. Suppose that $(f \otimes f')_0((Q_1 \parallel Q'_1, \tau)) = (P_1 \parallel P'_1, \sigma)$ and there is a transition in $\mathcal{L}(Q \parallel Q')$ of the form $(Q_1 \parallel Q'_1, \tau) \xrightarrow{\tau'} (Q_2 \parallel Q'_1, \tau'')$. This must arise from a transition of the operational semantics of the form $Q_1 \xrightarrow{(\tau', \tau'')} Q_2$. We need to find a suitable transition in $\mathcal{L}(Q)$ but we cannot just assert that $\mathcal{L}(Q)$ has a transition of the form $(Q_1, \tau) \xrightarrow{\tau'} (Q_2, \tau'')$ since we first need to argue that the pair (Q_1, τ) is a possible state of $\mathcal{L}(Q)$. In fact it is not! First recall that a store really represents an entailment-closed set of formulas, the exact representing formula is not important. Now consider the formula τ, it has three components: formulas added by the environment, formulas added by the component Q and its derivatives and formulas added by the component Q' and its derivatives. In the LTS $\mathcal{L}(Q)$ the first two components of τ can be mimicked exactly but in the third component any references to the local variables of Q' cannot be mimicked by the environment. More precisely we can assert the following by using appropriate environment actions. Let $\vec{u}$ be all the local variables of τ, the LTS $\mathcal{L}(Q)$ will contain the state $(Q_1, \exists\vec{u}.\tau)$. Since the variables $\vec{u}$ do not occur in Q_1 we can invoke Lemma 5.2 to assert that the following transition exists

$$Q_1 \xrightarrow{\exists\vec{u}.\tau', \exists\vec{u}.\tau''} Q'_1$$

according to the operational semantics and hence in the LTS $\mathcal{L}(Q)$ we must have the transition

$$(Q_1, \exists\vec{u}.\tau) \xrightarrow{\exists\vec{u}.\tau'} (Q'_1, \exists\vec{u}.\tau'').$$

Now the simulation f will specify a transition in $\mathcal{L}^*(P)$, say $(P_1, \sigma) \xrightarrow{\sigma'} (P'_1, \sigma'')$, that corresponds to this transition and, by the structure of the transition system for a parallel composition, there is a transition $(P_1 \parallel P_2, \sigma) \xrightarrow{\sigma'} (P'_1 \parallel P_2, \sigma'')$ in $\mathcal{L}^*(P \parallel P')$ associated with this latter transition. We set $(f \otimes f')((Q_1 \parallel Q_2, \tau) \xrightarrow{\tau'} (Q'_1 \parallel Q_2, \tau'')) = (P_1 \parallel P_2, \sigma) \xrightarrow{\sigma'} (P'_1 \parallel P_2, \sigma'')$. We proceed analogously for transitions of the other component. Checking that this is indeed a simulation is routine but tedious. $\square$

It is absolutely essential that the CL-transition systems record environment moves. Without this the above proposition would be false as was noted in the course of the proof. It is easy to verify that this correspondence is functorial.

PROPOSITION 5.5. *The correspondence that takes processes P and Q to $P \parallel Q$ and simulation morphisms f, f' to $f \otimes f'$ is a bifunctor and forms a tensor product.*

Remark: (This remark can be skipped unless one is worried about fine points in the definition of tensor products.) The alert reader might worry about the coherence conditions required for a tensor product, these are described, for example, in the book Mac Lane [1971]. These are, however, trivial in our context. One requires various natural isomorphisms and the commutativity of certain diagrams. These require constructing the simulations that witness the isomorphisms. Since the isomorphic processes are just, in one case, $P \parallel Q$ and $Q \parallel P$ and, in the other case, $P \parallel (Q \parallel R)$ and $(P \parallel Q) \parallel R$ the simulation morphisms are basically identity morphisms and the coherence requirement is trivialized. □

In fact parallel composition is more than a tensor product; as we saw in the last section it is in fact "almost a product". In the sense of the present section this means that there are projections but the universal property fails.

PROPOSITION 5.6. *Given any two processes P_1 and P_2 there are simulations, called ρ_1 and ρ_2, from $P_1 \parallel P_2$ to P_1 and P_2 respectively.*

PROOF. Recall that the actual maps go in the direction opposite the simulation. Thus we need to specify maps from $\mathcal{L}(P_1)$ and $\mathcal{L}(P_2)$ to $\mathcal{L}^*(P_1 \parallel P_2)$. The simulation ρ_1 is given by the scheme

$$\rho_1((Q_1, \sigma) \xrightarrow{\sigma'} (Q_1', \sigma'')) = (Q_1 \parallel Q_2, \sigma) \xrightarrow{\sigma'} (Q_1' \parallel Q_2, \sigma'')$$

and ρ_2 is given similarly. It is obvious that these are simulations. □

The reason that this does not actually give a true Cartesian product is that $P \parallel Q$ cannot always be simulated by anything that can simulate both P and Q. In the indeterminate language $P + Q$ can simulate both P and Q but it certainly cannot simulate $P \parallel Q$ in general. As before we have that $P + Q$ is a true Cartesian product in the indeterminate language. The proof of the following proposition is an easy exercise and is omitted.

PROPOSITION 5.7. *If P_1 and P_2 are processes, there are simulation morphisms η_1 and η_2 from $P_1 + P_2$ to P_1 and P_2 respectively. If R is another process equipped with simulation morphisms f_1 to P_1 and f_2 to P_2, there is a unique simulation morphism f from R to $P_1 + P_2$ such that $\eta_i \circ f_i = f, i = 1, 2$.*

We now address the fundamental adjunction that exhibits the scoping construct as a left adjoint and hence as an existential quantification. In order to do this we need to discuss the fibres and the indexed structure. The base category is the category of sorts. Each fibre is a category with processes as objects and simulations as morphisms. The free variables in the processes have to be of the sorts specified by the base object. Where does the notion of "local variable" or "private variable" enter the theory? The answer is "in the *simulations*". Recall that we had first to define CL-transition systems and then the simulation maps. In the definition of $\mathcal{L}(P)$ (or $\mathcal{L}^*(P)$) for some process P, we had to encode environment moves which were restricted to being formulas that only refer to the free variables whereas the stores could contain formulas that refer to private variables. Thus the base object specifies what the environment moves can be and hence influences the notion of simulation. It is important to clarify the meaning of π^* in this case. A process P in some fibre has an associated CL-transition system $\mathcal{L}(P)$; when one uses π^* to move P to another fibre we get quite another CL-transition system with different environment moves. From the point of view of the indexed structure the object $\pi^*(P)$ looks just like the object P but the morphisms are quite different. For example consider the process $ask(y = 1) \longrightarrow tell(z = 2)$ with store $(z > 0) \wedge (x = 1)$ in the fibre over N^2 where y and z are the free variables of sort N. Now x is a local variable, perhaps created by some now terminated subprocess. The environment can add information to the store, for example $y = 1$, thus allowing the *ask* to proceed. But the environment cannot add any information about the local variable x. Now the process $\pi^*(ask(y = 1) \longrightarrow tell(z = 2))$ lives in the fibre over N^3 with x now a global variable. Now the environment can add $x = y$ to the store, also enabling the *ask* in the process (pun intended).

PROPOSITION 5.8. *The following adjunction tableau holds*

$$\frac{\nu x.P \xrightarrow{f} Q}{P \xrightarrow{\hat{f}} \pi^*(Q).}$$

PROOF. Suppose that we are given a simulation morphism f from $\nu x.P$ to Q. We must construct a simulation morphism $\hat{f}$ from P to $\pi^*(Q)$. We spell out how $\hat{f}$ maps transitions of $\mathcal{L}(\pi^*(Q))$ to transitions of $\mathcal{L}^*(P)$. Consider the transition $(Q_1, \tau_1) \xrightarrow{\tau} (Q_2, \tau_2)$ of $\mathcal{L}(\pi^*(Q))$. The stores may have x appear as a free variable and $\tau = \tau_1 \wedge \phi$, where ϕ may have a free x. Now in $\mathcal{L}(Q)$ there is a state $(Q_1, \exists_x.\tau_1)$; it is not *a priori* obvious that such a state is reachable, but the next remarks will make clear that it is. In $\mathcal{L}(Q)$ we can have an environment move in which the environment adds $\exists_x.\tau$ to the store $\exists_x.\tau_1$. Note that it is not enough to consider an environment move in which just $\exists_x.\phi$ is added to the store. Consider, for example, a situation in which a process is asking whether $y = 3$ and the facts $z = 3$ and $y = x$ are in the store. The environment could now state that $z = x$ which makes the store strong enough to answer the query being asked. If we now move to the situation where the environment cannot mention x then the existential quantification of the formula $x = z$ just yields the trivial formula **true**. However by adding $z = y$, the environment can have the same effect as it did before without having to mention x explicitly. Incidentally this shows how to reach the state $(Q_1, \exists_x.\tau_1)$ in $\mathcal{L}(Q)$ by adjusting the environment moves that were needed to reach (Q_1, τ_1) in $\mathcal{L}(\pi^*(Q))$.

Note that any derivative of Q cannot have occurrences of x as a global variable, hence by Lemma 5.2 above, if the transition $(Q_1, \tau_1) \xrightarrow{\tau} (Q_2, \tau_2)$ occurs in $\mathcal{L}(\pi^*(Q))$ we will have the transition $(Q_1, \exists_x.\tau_1) \xrightarrow{\exists_x.\tau} (Q_2, \exists_x.\tau \wedge \psi)$; since $\tau_2 = \tau \wedge \psi$ and ψ has no free occurrences of x we have $(\exists_x.\tau) \wedge \psi = \exists_x.\tau_2$. Using the simulation f we have a transition $(P_1, \sigma_1) \xrightarrow{\sigma} (P_2, \sigma_2)$ with the required entailments between stores. In the stores σ *etc.*, the variable x may occur as a local variable but no environment move can have added a formula mentioning x. Using Lemma 5.1 we see that there is a transition $(P_1, \sigma_1 \wedge \tau_1) \xrightarrow{\sigma_1 \wedge \tau} (P_2, \sigma_2 \wedge \tau)$. This transition serves to simulate the original transition that we started with. This is the inductive step in the construction of the simulation $\hat{f}$. The base case is of course trivial.

The reverse direction of the adjunction is almost immediate. As before we have a transition $(Q_1, \tau_1) \xrightarrow{\tau} (Q_2, \tau_2)$ but now all the formulas are free of x. Thus this is a possible transition of $\mathcal{L}(\pi^*(Q))$ in which the environment has never added a formula that mentioned x even though it is free to do so. This transition must be simulated by a transition of $\mathcal{L}^*(P)$, in which also there are no environment moves that added formulas mentioning x. The stores in the transitions of $\mathcal{L}^*(P)$, however, can mention x but any such formula must be added by a transition of a derivative of P. Thus they are legitimate transitions of $\mathcal{L}^*(\nu x.P)$ with x being regarded as a local variable. Thus the same correspondence can be used above to construct the simulation required in the upper half of the adjunction tableau. □

Before proceeding to the Beck conditions and Frobenius we will discuss the ask/tell adjunction briefly.

Proposition 5.9. *The following adjunction tableau holds,*

$$\frac{[tell(\phi) \parallel P] \xrightarrow{f} Q}{P \xrightarrow{g} ask(\phi) \to Q.}$$

Proof. We sketch the construction of the simulations informally. Suppose we are given f. An initial transition of $ask(\phi) \to Q$ can only occur if there is an environment move that adds more than ϕ, say σ, where $\sigma \vdash \phi$. But then P is only required to simulate the move given σ as well. If P is started in a store σ such that $\sigma \vdash \phi$ we have that P and $P \parallel tell(\phi)$ are indistinguishable. Furthermore we know that the latter can simulate Q and that once the ask has been answered the process $ask(\phi) \to Q$ can only make the same moves as Q. Thus we can use the correspondence defined by f to define g. In the reverse direction suppose that we are given g. Now we need to simulate a move of Q. If this is in a store that entails ϕ we know that any transition of Q is possible for $ask(\phi) \to Q$ so we can use g to define f. If the store σ does not entail ϕ we can use the transition of the $tell(\phi)$ process to upgrade the store to $\sigma \wedge \phi$ for P. We know that P in $\sigma \wedge \phi$ can simulate any move by Q in $\sigma \wedge \phi$ and hence certainly in σ alone. □

The Frobenius reciprocity condition and the Beck conditions are as in the preorder situation except that we must explicitly show the simulation isomorphisms.

Proposition 5.10. *The following isomorphism holds:* $\nu x.(P \parallel \pi^*(Q)) \cong (\nu x.P) \parallel Q$.

Proof. This is almost immediate. The effect of having $\pi^*(Q)$ is to have essentially the same process as Q but there is now the possibility that the environment can make moves that add information about x. However the νx serves precisely to eliminate this possibility. Thus the two transition systems are identical. □

The Beck conditions are analogues of the Beck conditions discussed for the preorder case. However the requirement that there be an isomorphism turns out to be too strong. We need a definition first.

Definition 5.6. *Suppose L is any LTS and suppose that $\alpha = (q_0, l, q_1)$ is a transition from the initial state. We define the LTS L/α, called the quotient of L by α, to be the LTS with initial state q_1, the transitions and states being the ones reachable from q_1.*

Proposition 5.11. *The processes $P[t/x]$ and $\nu x.[P \parallel tell(x = t)]$ can simulate each other, if t has no free occurrences of x. Furthermore, one can choose simulation morphisms f, g such that $f \circ g$ gives the identity simulation on $P[t/x]$ while $g \circ f$ gives a simulation of $\nu x.[P \parallel tell(x = t)]$ by a quotient of itself.*

Proof. We need to exhibit a pair of morphisms between the CL-transition systems $\mathcal{L}(P[t/x])$ and $\mathcal{L}(\nu x.[P \parallel tell(x = t)])$. Since these processes are in the same fibre the environment moves are the same. A structural induction proof establishes the required correspondence. The only nontrivial step is the base case. If P is of the form $tell(\phi)$ then $P[t/x]$ is of the form $tell(\phi[t/x])$ which has the following transition $(tell(\phi[t/x]), \sigma) \xrightarrow{\sigma'} (NIL, \sigma' \wedge \phi[t/x])$. The other CL-transition system $\mathcal{L}^*(\nu x.[P \parallel tell(x = t)])$ has the transition $(\nu x.[tell(\phi) \parallel tell(x = t)], \sigma) \xrightarrow{\sigma'}_* (NIL, \exists_x.\sigma' \wedge \phi \wedge (x = t)$. The latter store is logically equivalent to $\sigma' \wedge \phi[t/x]$. However the latter process has the additional step $tell(x = t)$. One can consider the quotient in which it does this step first; the rest of the transitions then can easily be seen to be isomorphic to the transitions of $P[t/x]$. □

Looking at this proof, it is clear however that of the four essential cases of Beck given in Proposition 4.10, only the last suffers the problem above, and that the other three are indeed isomorphisms. The key point here is that in each of the first three cases the *tell* step is included on each side. It was shown in Seely [1983] that these cases are sufficient for an "almost reasonable" logic with equality, lacking essentially just the properties that $x = x$ is terminal (that is, isomorphic to true) and that $x = y$ is isomorphic to $\langle x, x\rangle = \langle y, y\rangle$. Many of the basic properties of the logic of equality do nevertheless remain: reflexivity, symmetry, transitivity, substitution, and the main equivalences of derivations. Indeed, an example of a structure (groupoid representations) with this weaker structure was discussed in Seely [1983].

What this shows is that while the Beck condition is very close to holding the condition does not exactly hold. However it also shows that the notion of isomorphism in the simulation category is very strong. We do not get the full correspondence with logic but we do have all the equations that follow from the adjunction and Frobenius. Of course, as far as the logic without equality is concerned, we do have all the required structure.

6. Bisimulation and Isomorphism

A key question in any study of the operational semantics of processes is the role of bisimulation and other related notions of process equivalence. Given the notion of simulation preorder and simulation morphism we immediately have two notions of equivalence. (The first is repeated from Definition 4.1.)

Definition 6.1. *Two processes P, Q are* **similar** *if $P \preceq Q$ and $Q \preceq P$. We write $P \sim Q$.*

Definition 6.2. *Two processes P, Q are* **isosimilar** *if there are simulation morphisms $f : P \longrightarrow Q$ and $g : Q \longrightarrow P$ such that $f \circ g$ and $g \circ f$ are both identity morphisms.*

We define bisimilarity as we defined the simulation preorder. We use the same convention as before of primed stores to indicate implicit environment effects and double primed stores to indicate the final stores.

Definition 6.3. *We say that two process-store pairs (P, σ) and (Q, τ) are* **bisimilar** *if whenever $P \xrightarrow{(\sigma', \sigma'')} P'$ is a possible transition defined by the operational semantics, there is a finite transition sequence $Q \xrightarrow{(\tau', \tau'')} Q'$ with the τ and σ stores and the corresponding primed analogues all being logically equivalent and (P', σ'') and (Q', τ'') being bisimilar, and* vice versa *with the roles of P and Q exchanged.*

Except for the roles of the stores this is the definition used in concurrency theory. The key facts are given by the next two propositions.

PROPOSITION 6.1. *If two processes are bisimilar they must be similar but there are similar processes that are not bisimilar.*

This needs no proof. The positive direction is clear and the negative direction has already been demonstrated by Example 4.2.

PROPOSITION 6.2. *Isosimilarity implies bisimilarity but there are bisimilar processes that are not isosimilar.*

PROOF. The positive direction is clear. Isosimilarity establishes an isomorphism of transition systems and thus one can always set up the correspondence needed in the definition of bisimilarity. For the negative direction note that for a process P, we have that P and $P+P$ are bisimilar, by the usual arguments familiar in process algebra, but they are not isosimilar. To see this note that a simulation of P by $P+P$ would require that a move of P would have to be simulated by one, or the other or both copies of P in $P+P$. Whereas a move of either copy of P in $P+P$ has to be simulated by a move P for the reverse simulation. Suppose that the left copy of P in $P+P$ simulates moves of P. Then under composition a move by the right copy of P in $P+P$ would be simulated by a move of the left copy of P, in short the composition is not the identity simulation. The same thing happens with any other choice that we make for the simulation of P by $P+P$. $\Box$

The upshot is that isosimilarity is a very strong condition. We noted that the Beck condition does not fully hold in the last section even though the processes come "close" to satisfying the Beck conditions. It is straightforward to check that they are in fact all bisimilar. This raises the interesting possibility of studying the generalization of hyperdoctrines where the processes identities are to be satisfied only up to bisimilarity rather than up to isomorphism.

7. Other Process-based Logical Structures

In this section we summarize results about other structures that can be viewed in terms of fibred categories. We look at a couple of denotational semantic models of indeterminate concurrent constraint programming, one due to Saraswat *et al.* [1991] and one due to de Boer and Palamidessi [1991]. We also briefly recount an analysis of a linear variant of concurrent constraint programming. Finally we discuss a simple concurrent imperative language with block structure [Brookes 1993]. These last two examples show that the view that local variables define an existential quantification is not dependant on the notion of monotonic update. Curiously enough, one cannot, in any obvious way that we could see, view local variables in sequential languages as an existential quantification since one needs parallel composition to express a Frobenius type property.

In the treatment of Saraswat *et al.* [1991] processes are modeled as sets of interaction sequences. Each sequence gives information about the resting points of a process, *i.e.* points at which the process cannot move unless the environment adds more information to the store, plus information about how the process and the environment interacted in order to get to the resting point. Each such sequence is encoded as a special kind of closure operator. The sets of sequences are closed under certain operations and one can prove a full abstraction theorem with respect to a natural notion of observability. The theory of de Boer and Palamidessi [1991] is based on very similar ideas about encoding interaction sequences that lead to the resting point. However they also represent additional information that allows one to handle deadlock. They have a different treatment of full abstraction resulting from different closure conditions.

In both cases we can construct fibred categories with posetal fibres. One takes the usual base category. The fibres consist of processes ordered by reverse inclusion of sets of sequences. In each case one can easily check that one has parallel composition giving a tensor and νx is a left adjoint to substitution and hence can be viewed as an existential quantification. One can verify both the Frobenius condition and the Beck conditions easily. Formally these amount to the following statements. The first statement below says that parallel composition is tensor.

$$\frac{[\![A]\!] \supseteq [\![B]\!], [\![A']\!] \supseteq [\![B']\!]}{[\![A \parallel A']\!] \supseteq [\![B \parallel B']\!]}$$

The next statement says that $+$ is product.

$$\frac{[\![A]\!] \supseteq [\![B]\!], [\![A']\!] \supseteq [\![B]\!]}{[\![A + A']\!] \supseteq [\![B]\!]}$$

Finally we have the fundamental adjunction.

$$\frac{[\![\nu x.A]\!] \supseteq [\![B]\!]}{[\![A]\!] \supseteq \pi^*([\![B]\!])}$$

It is important to note that this would be wrong without π^*.

The ask/tell adjunction breaks down in the theory of de Boer and Palamidessi; this is not to be construed as a weakness or a criticism. What is interesting is that there is “negative information” recorded in the theory and this, though essential for treating deadlock, breaks the ask/tell adjunction. The required statement is the following:

$$\frac{[\![\mathbf{tell}(\phi) \parallel A]\!] \supseteq [\![B]\!]}{[\![A]\!] \supseteq [\![\mathbf{ask}(\phi) \to B]\!]}$$

Consider $A = NIL$, $B = tell(\phi)$. Clearly the inclusion above the line is an identity but below the line we have $[\![NIL]\!] \supseteq [\![ask(\phi) \to tell(\phi)]\!]$, which is not valid since $ask(\phi) \to tell(\phi)$ has a deadlock mode that NIL does not have. Deadlock introduces negative information. If we do not introduce deadlock information then this adjunction holds as well. In the example above $ask(\phi) \to tell(\phi)$ is the same as NIL in the theory of Saraswat *et al.* [1991] because neither can make any observable difference to the store.

7.1 Linear Concurrent Constraint Programming

Recently several authors have been looking at linear concurrent constraint programming in large part because it allows one to stay close to logic while giving one the freedom to retract information [Saraswat and Lincoln 1992, de Boer, Palamidessi, and Best 1994]. One has the same setup and indeed even the same syntax for processes but now resources get *consumed* by an *ask*. One must think in terms of consumption of resources and flow of resources rather than consumption of facts. In order to model the traditional notion of logical formulas in the store one needs to introduce exponentials to model renewable resources as in Girard's original treatment of linear logic [Girard 1987].

It is straightforward to write down an operational semantics embodying the above ideas. One can then define the analogue of the simulation morphism of the last section and construct an indexed category. There are some crucial differences. First of all, what is the analogue of requiring that $\sigma \vdash \tau$ as a way of expressing the correspondence between the simulating store σ and the simulated store τ? Since resources may get consumed one cannot just use the notion of entailment. Instead we say that there is a map between stores that describes exactly which resources correspond to which resources. Furthermore we demand that the simulation maintain the existence of such a map. Finally when the labelled transition systems associated with processes are constructed, the environment moves have to acquire the status of full-blown transitions. Now the environment moves can remove resources from the store. With the definitions redeveloped as just sketched, one can establish the adjunction

$$\frac{\nu x.P \to Q}{P \to \pi^*(Q).}$$

and can also check Frobenius. It is not clear what the appropriate Beck conditions should be and which of the various theories of linear hyperdoctrines are appropriate. This is the subject of further study. The fact that the existential quantification phenomenon occurs in this "nonmonotonic" situation is very interesting since monotonicity has been such a key part of the preceding theory.

8. Conclusions

The main result of this paper is that indeterminate concurrent constraint programming equipped with a suitable notion of simulation between processes forms a coherent tensor hyperdoctrine. In simpler terms, one can view parallel composition as conjunction and local scoping of variables as existential quantification. We carried out this investigation using two closely related notions of simulation, simulation as a preorder and simulation as a morphism and found that the logical structure was the same but the induced notions of process equivalence are quite different with bisimulation sandwiched strictly between them. From the point of view of pure concurrency theory the following points are of interest. The notion of simulation is more "extensional" than is usual. Thus we do not just track moves of one process by move sequences of the simulating process. Rather we look at the observable effect of the move. This makes it possible that a process with some possible moves is simulated by a NIL process. Furthermore even when the moves do correspond the relation between the labels is nontrivial and is not the usual, purely intentional, matching of labels but is, instead, the relation of logical entailment.

A significant point is the following. One can look at programming languages that were not developed with any logical connections in mind, and nevertheless still see the correspondence with hyperdoctrines and hence with logic. There have been a number of studies of imperative languages with parallel composition and local variables, for example by Horita *et al.* [1994] and by Brookes [1993]. Once again processes are modeled by interaction sequences that describe the interaction between the process and the environment. Using the familiar pattern one can set up a fibred structure and check that the basic adjunction holds and that Frobenius holds as well. The latter amounts to **new** x *in* $[P \parallel Q] = [$**new** x *in* $P] \parallel Q$ where Q has no free occurrences of x. All these examples suggest that there is indeed a common logical core to all concurrent programming languages.

There are several points to understand and extend. Our most pressing concern is to flesh out the treatment of linear concurrent constraint programming and understand completely its logical structure and the relationship to linear hyperdoctrines. Secondly we wish to understand the programming analogue of the universal quantifier. Roughly speaking this should somehow express receptivity. This idea comes through in the work of Lincoln and Saraswat on linear concurrent constraint program. Even more importantly it is easy to see that the "scope extrusion" phenomenon of the π calculus is mimicked by the interplay between the existential quantifier and the universal quantifier in ordinary logic. This suggests that understanding universal quantifiers is important for understanding "mobility" in languages like the π calculus.

It would be very interesting to understand how higher-order process calculi fit into this picture. In particular we do not see yet what the logical significance is of the "functions as processes" correspondence discovered by Milner [1992]. Finally we are actively developing and studying a synchronous, linear language from the viewpoint of the present paper.

Acknowledgements

We have benefitted from discussions with many colleagues including Rick Blute, Djordje Cubric, Radha Jagadeesan, Alan Jeffrey, Catuscia Palamidessi, and Vijay Saraswat. We would like to thank NSERC and FCAR for financial support and the Ontario-Quebec cooperative program for a travel

grant.

References

ABRAMSKY, S. 1994. Proofs as Processes. *Theoretical Computer Science* 135, pp. 5–9.

ABRAMSKY, S., AND JAGADEESAN, R. 1993 New foundations for the geometry of interaction. *Information and Computation*, 111(1):53–119, May 1993.

BELL, J., AND MACHOVER, M. 1977. *A Course in Mathematical Logic*. North-Holland.

BROOKES, S. 1993. Full Abstraction for a Shared Variable Parallel Language. In *Proceedings of 8th Annual IEEE Symposium On Logic In Computer Science*, 98–109.

DE BOER, F.S., AND PALAMIDESSI, C. 1991. A Fully abstract model for Concurrent Constraint Programming. In *Proceedings of TAPSOFT/CAAP 1991*, Number 493 in Lecture Notes In Computer Science, 296–319.

DE BOER, F.S., PALAMIDESSI, C., AND BEST, E. 1994 Concurrent constraint programming with information removal. Unpublished manuscript.

GIRARD, J.-Y. 1987. Linear Logic. *Theoretical Computer Science 50*, 1–102.

GIRARD, J.-Y. 1989. *Proofs and Types*. Volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

HENKIN, LEON, MONK, J. DONALD, AND TARSKI, ALFRED. 1971. *Cylindric Algebras (Part I)*. North Holland Publishing Company.

HORITA, E.W., DE BAKKER, J.W., AND RUTTEN, J.J.M.M. 1994. Fully Abstract Denotational Models for Nonuniform Concurrent Languages. *Information and Computation* **115**, 1, 125–178.

JOYAL, A., NIELSEN, M., AND WINSKEL, G. 1993. Bisimulation and Open Maps. In *Proceedings of 8th Annual IEEE Symposium On Logic In Computer Science*, 418–427.

LAMBEK, J., AND SCOTT, P.J. 1986. *Introduction to Higher Order Categorical Logic*. Volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.

LAWVERE, F.W. 1969. Adjointness in foundations. *Dialectica 23*, 281–296.

LAWVERE, F.W. 1970. Equality in hyperdoctrines and the comprehension schema as an adjoint functor. In *Proc. New York Symposium on Applications of Categorical Logic*, A. Heller, Editor.

MAC LANE, S. 1971. *Categories for the Working Mathematician*. Volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York.

MILLER, D. 1994. A Multiple-Conclusion Meta-Logic. In *Proceedings of 9th Annual IEEE Symposium On Logic In Computer Science*, 272–281.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall.

MILNER, R. 1992. Functions as Processes. *Mathematical Structures in Computer Science* 2 (2), pp. 119–141.

NIELSEN, M., AND WINSKEL, G. 1994. Models for Concurrency. In *Handbook of Logic and Foundations for Theoretical Computer Science*. Oxford University Press.

PANANGADEN, P., SARASWAT, V., SCOTT, P.J., AND SEELY, R.A.G. 1993. A Hyperdoctrinal View of Concurrent Constraint Programming. In *Semantics: Foundations and Applications; Proceedings of REX Workshop*, Number 666 in Lecture Notes In Computer Science, 457–476.

PANANGADEN, P., AND SHANBHOGUE, V. 1992. The Expressive Power of Indeterminate Dataflow Primitives. *Information and Computation 98*, 1, 99–131.

SARASWAT, V.A. 1993. *Concurrent Constraint Programming*. MIT Press.

SARASWAT, V.A., JAGADEESAN, R., PANANGADEN, P., AND RINARD, M. 1991. Semantic foundations of concurrent constraint programming. In *Proceedings of the Baastad 91 workshop on Concurrency*, Number 63 in Programming Methodology Group Chalmers University of Technology and Goteborg University, 385–427.

SARASWAT, V.A., AND LINCOLN, P. 1992. Higher-order, linear concurrent constraint programming.

Saraswat, Vijay, and Rinard, Martin. 1990. Concurrent Constraint Programming. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 232–245.

Saraswat, Vijay, Rinard, Martin, and Panangaden, Prakash. 1991. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*.

Scott, D. 1982 Domains for denotational semantics. In: *Ninth International Colloquium On Automata Languages and Programming*, Number 140 in Lecture Notes In Computer Science.

Seely, R.A.G. 1983. Hyperdoctrines, natural deduction and the Beck conditions. *Zeitschr. f. math. Logik und Grundlagen d. Math. 29*, 505–542.

Seely, R.A.G.. 1990. Polymorphic linear logic and topos models. *C.R. Math. Rep. Acad. Sci. Canada XII*, 1, 21–26.