Monads with merging

Exequiel Rivas, Mauro Jaskelioff

Monads with merging

Exequiel Rivas
Inria, Paris, France
exequiel.rivas-gadda@inria.fr

Mauro Jaskelioff
CIFASIS, CONICET, Rosario, Argentina
Universidad Nacional de Rosario, Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

Monoids are one of the simplest theories in which we can compose elements of a set. Similarly, monads have been used extensively to treat composition of effectful code and its denotational semantics. During the last forty years the theory of monoids has been extended with diverse merge-like operators. In this article, we replicate several of these extensions at the level of monads. Building on a well-known relation between monads and monoids, we introduce monads with additional structure that account for merging. We show how monads with merging generalise and relate to models for well-known algebraic theories for concurrency such as classic process algebras and the more recent concurrent monoids. With these results, we aim to facilitate the generalisation and comparison of different approaches to concurrency.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Functional constructs; Theory of computation → Pre- and post-conditions

Keywords and phrases Monads, Concurrency, Process Algebra, Bimonoids

Acknowledgements We are grateful to Germán Delbianco who introduced us to concurrent monoids.

1 Introduction

One of the basic questions a computational model has to answer is how to represent the sequencing of computation. Monoids are one of the simplest theories in which we can compose elements of a set, so they are a fundamental building block in program semantics. If we generalise from sets to functors, monoids turn into monads. This generalisation has been extremely profitable: Monads have been used extensively to model composition of effectful code, both in denotational semantics [24] and in the structuring of programs [30].

When tackling the problem of concurrency, sequencing is not enough: we also need a way to model programs which execute concurrently. Algebraic models of processes such as ACP, CCS or π-calculus introduce merge operators, where the details vary in each model. Some concrete examples of monoids with a merge operator are the models of process algebra PA_δ, which is a simplified version of ACP [4] without communication (Section 3), and concurrent monoids (Section 4), which is a recent approach developed by Hoare et al [15].

In this article we aim to extend the generalisation from monoids to monads, to account for a merge operator, hence obtaining monads with merging. Extensions of monoids at the level of monads proved useful for organising additional structure on a monad, and often suggested correspondence with structures used in the functional programming practice. We obtain a process monad (Section 3) which lifts process algebras to the level of functors, and a concurrent monad (Section 4), which lifts concurrent monoids to functors. We show how one may use a certain lax slice category to model Plotkin triples and recover the rules from separation logic that hold for concurrent monoids (Section 5). We expect the introduction of process monads and concurrent monads to lay the groundwork for future work leading to a more unified categorical approach to concurrency (Section 6).

2 Preliminaries

We will work mainly with cartesian products. The product of objects A and B will be denoted by $A \times B$, its projections by $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, and its pairing $\langle f, g\rangle : C \to A \times B$ for $f : C \to A$ and $g : C \to B$. The coproduct of objects will be denoted by $A + B$, the terminal object by 1 and its unique morphisms by $! : A \to 1$.

We recall the definition of a monad in two variants: first in its classical definition, and then as an extension system (or Kleisli triple).

▶ **Definition 2.1.** *A* monad *on $\mathcal{C}$ is triple (T, μ, η) where $T : \mathcal{C} \to \mathcal{C}$ is a functor, and $\eta : \mathrm{Id} \to T$ and $\mu : T \cdot T \to T$ are natural transformations such that the following identities hold*

$$\mu_X \circ T\mu_X = \mu_X \circ \mu_{TX}, \qquad \mu_X \circ T\eta_X = \mathrm{id}_{TX}, \qquad \mu_X \circ \eta_{TX} = \mathrm{id}_{TX}.$$

There is an alternative definition of a monad as an extension system. In some cases, this definition is more convenient to work with as there is no iteration of the functor T [22]:

▶ **Definition 2.2.** *An* extension system *on $\mathcal{C}$ is a triple $(T, \text{-}^{\#}, \eta)$ where $T : \mathbf{Ob}(\mathcal{C}) \to \mathbf{Ob}(C)$, $\eta_A : A \to TA$ for $A \in \mathbf{Ob}(\mathcal{C})$, and $f^{\#} : TA \to TB$ for $f : A \to TB$ in $\mathcal{C}$ such that the following equations hold:*

$$\eta_A{}^{\#} = \mathrm{id}_{TA}, \qquad f^{\#} \circ \eta_A = f, \qquad g^{\#} \circ f^{\#} = (g^{\#} \circ f)^{\#}.$$

As both presentations are equivalent [22] (take $\mu = \mathrm{id}_{TA}{}^{\#}$, $f^{\#} = \mu \circ Tf$), we will use one definition or the other depending on convenience. Given a monad $(T, \text{-}^{\#}, \eta)$ on $\mathcal{C}$, its *Kleisli category*, $\mathcal{K}\ell(T)$, has as objects those of $\mathcal{C}$, and morphisms are those of type $f : X \to TY$ in $\mathcal{C}$ (which we denote as $f : X \to_T Y$ when regarded as morphisms in $\mathcal{K}\ell(T)$). Composition for this category is defined as $f \bullet g = f^{\#} \circ g$ and identity is defined as $\mathrm{id}_X = \eta_X$.

▶ **Example 2.3.** Let $(M, \cdot, \varepsilon)$ be a monoid. Then, we have a monad (T, μ, η) on the category **Set** of sets and functions, where $TX = M \times X$, the multiplication is $\mu_X(m_1, (m_2, x)) = (m_1 \cdot m_2, x)$, and the unit is $\eta_X(x) = (\varepsilon, x)$.

We will work with functors and monads which are *strong* with respect to the cartesian product. Strength is a form of compatibility between functors and the cartesian structure.

▶ **Definition 2.4** (Strong Functor)**.** *A functor $F : \mathcal{C} \to \mathcal{C}$ is* strong *if it comes equipped with a natural transformation $\sigma_{X,Y} : FX \times Y \longrightarrow F(X \times Y)$ such that the following equations hold:*

$$\pi_1 = F(\pi_1) \circ \sigma_{X,1}, \qquad \sigma \circ (\sigma \times \mathrm{id}) \circ \alpha = F(\alpha) \circ \sigma$$

where $\alpha = \langle\langle \pi_1, \pi_1 \circ \pi_2\rangle, \pi_2 \circ \pi_2\rangle$ expresses the associativity of the cartesian product.

▶ **Definition 2.5** (Strong Monad)**.** *A monad (T, μ, η) on $\mathcal{C}$ is* strong *if the underlying functor T is strong and the strength is compatible with μ and η:*

$$\eta_{A\times B} = \sigma_{A,B} \circ (\eta_A \times \mathrm{id}), \qquad \sigma_{A,B} \circ (\mu_A \times \mathrm{id}) = \mu_{A\times B} \circ T\sigma_{A,B} \circ \sigma_{TA,B}$$

There is a similar definition of strength $\bar{\sigma}_{X,Y} : X \times FY \longrightarrow F(X \times Y)$ acting on the right, but as the cartesian product is symmetric, we can obtain it from the left strength as $\bar{\sigma} = F\gamma \circ \sigma \circ \gamma$ where $\gamma = \langle \pi_2, \pi_1\rangle$ swaps the elements in a cartesian product.

There is another way in which a functor can be made compatible with cartesian structure.

▶ **Definition 2.6** (Monoidal Functor). *A* monoidal functor *is a functor $F : \mathcal{C} \to \mathcal{C}$ equipped with a monoidal structure (m, e) where $m : FX \times FY \longrightarrow F(X \times Y)$ is a natural transformation and $e : 1 \longrightarrow F1$ is a morphism such that standard coherence diagrams commute. In addition, if the monoidal structure is compatible with γ, then the monoidal functor is* symmetric.

▶ **Definition 2.7** (Monoidal Monad). *A monad T is monoidal when there is a monoidal structure (m, e) on its underlying functor T such that $e = \eta_1$ and the monoidal and monadic structure are compatible:*

$$\eta_{A\times B} = m_{A,B} \circ (\eta_A \times \eta_B), \quad m_{A,B} \circ (\mu_A \times \mu_B) = \mu_{A\times B} \circ Tm_{A,B} \circ m_{TA\times TB}.$$

Given a strong monad T, then T as a functor can be equipped with two canonical monoidal structures:

$$\begin{array}{lll} \varphi : & TA \times TB \longrightarrow T(A \times B) & \qquad \psi : \quad TA \times TB \longrightarrow T(A \times B) \\ \varphi = & \mu \circ T\bar{\sigma} \circ \sigma & \qquad \psi = \quad \mu \circ T\sigma \circ \bar{\sigma} \end{array}$$

and $e = \eta_1 : 1 \longrightarrow T1$ in both cases. A monad is said to be *commutative* when these two monoidal structures coincide.

In this article we will work mainly in the setting of $\mathcal{C} = \mathbf{Set}$, the category of sets and functions, and all the monads presented will be taken to be on it. The terminal object $1 = \{\star\}$ is a singleton set. A particular consequence of this is that every functor F and monad T will be strong, and there is only one possible strength σ ($\bar{\sigma}$) for each. For example, the powerset monad $\mathcal{P}$ (and its finite variant $\mathcal{P}_\mathrm{f}$) has strength $\sigma(X, y) = X \times \{y\}$. In general, the formula for strength for a **Set** functor might be expressed as $\sigma(v, y) = F(\lambda x : X.(x, y))(v)$. When T is commutative, we have that there is only one possible monoidal structure on it. In turn, if a monad is monoidal then it is commutative [21].

We close this section by stating the following lemma, which will guide our discussion on extending monads with additional operators.

▶ **Lemma 2.8** (Wolff [31]). *Given a monad T on* **Set**, *$T1$ can be given two monoid structures $a \cdot^{\varphi} b = T(!)(\varphi(a, b))$ and $a \cdot^{\psi} b = T(!)(\psi(a, b))$.*

This lemma provides a basic sanity check: if monads generalise monoids then we should be able to obtain a monoid from a monad. We provide a similar lemma for every generalisation of an algebraic structure that we introduce.

3 Monads and the axiom system PA_δ

The Algebra of Communicating Processes (ACP) was initially developed by Bergstra and Klop [4] when they were investigating the solutions of unguarded recursive equations. They proposed a complete algebra of processes that can account for concurrency with and without communication. Focusing on merging, we will restrict ourselves to PA_δ, a subset of the theory without communication. We will first introduce a subset of PA_δ, called BPA_δ, which provides only non-deterministic processes, and then show how to extend it to cover PA_δ.

3.1 Basic Process Algebra

Let A be a set of atomic actions over which the processes are constructed. The atoms are combined into processes by using the operators

$$\cdot : P \times P \longrightarrow P, \qquad + : P \times P \longrightarrow P, \qquad \delta : P.$$

These operations satisfy the following laws:

$$x + y = y + x \tag{A1}$$
$$(x + y) + z = x + (y + z) \tag{A2}$$
$$x + x = x \tag{A3}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \tag{A4}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{A5}$$
$$x + \delta = x \tag{A6}$$
$$\delta \cdot x = \delta \tag{A7}$$

This system is known as BPA_δ. Notice that the operator $\cdot$ is not necessarily commutative, and that it only distributes on one side with $+$, and therefore the operations $x \cdot (y + z)$ and $x \cdot y + x \cdot z$ are different a priori. We will assume that there is a successfully terminating process ε which works as the unit of the $\cdot$ operator. This system is sometimes referred to as $\mathrm{BPA}_{\delta\varepsilon}$ [3, 2], but for simplicity we will continue calling it BPA_δ.

If we want a parallel to Wolff's lemma for BPA_δ models, we need to have an additional monoid like structure on a monad T that gives us new operations on $T1$. We will relate the monoid structure given by $\cdot$ with the monad structure, and for the $+$ operation we will relate a monoid structure on the functor T with respect to the cartesian product of functors. This means that we will structure BPA_δ at the level of monads as a monad with some form of non-determinism. The operation $+$ behaves as a non-deterministic operator, and δ as a neutral element for the non-deterministic operation. We call this kind of monad a *basic process monad*.

▶ **Definition 3.1** (Basic process monad). *A monad (T, μ, η) has basic process structure if it is equipped with natural transformations*

$$\varsigma_X : TX \times TX \longrightarrow TX, \qquad \delta_X : 1 \longrightarrow TX,$$

such that the following equalities hold.

$$\varsigma_X = \varsigma_X \circ \gamma \tag{A1'}$$
$$\varsigma_X \circ (\varsigma_X \times \mathrm{id}) = \varsigma_X \circ (\mathrm{id} \times \varsigma_X) \circ \alpha \tag{A2'}$$
$$\varsigma_X \circ \Delta = \mathrm{id} \tag{A3'}$$
$$\mu_X \circ \varsigma_{TX} = \varsigma_X \circ (\mu_X \times \mu_X) \tag{A4'}$$
$$\varsigma_X \circ \langle \mathrm{id}, \delta \circ ! \rangle = \mathrm{id} \tag{A6'}$$
$$\mu_X \circ \delta_{TX} = \delta_X \tag{A7'}$$

where $\Delta = \langle \mathrm{id}, \mathrm{id} \rangle$ duplicates objects. These primed equations for basic process monads correspond to the non-primed equations for BPA_δ. There is no equation A5' because the equivalent to A5 already holds from the monad structure.

▶ Remark 3.2. When commutativity (A1) and idempotency (A3) axioms are dropped from BPA_δ, we have a near-rig structure. If we remove the equations corresponding to commutativity (A1') and idempotency (A3') from basic process monads, we obtain a form of non-deterministic monad which was previously considered in relation to the `MonadPlus` type-class in Haskell [27, 29].

In the monadic presentation we do not take a set of atomic actions, as we do not need it to express the axioms categorically. The axioms (A4') and (A7') require that σ and δ

are algebraic operations [26], and therefore they could be equivalently presented as *generic operations*, in this case simply constants

$$\varsigma' : 1 \longrightarrow T(1+1), \qquad \delta' : 1 \longrightarrow T0$$

where 0 represents the inital object of **Set**, the empty set. This presentation might be useful for finding possible instances of non-determinism structures on a monad, in a similar spirit to that of Dahlqvist et al. [8].

▶ **Example 3.3.** The canonical monads that represent non-determinism are powerset $\mathcal{P}$ and its finitary variant $\mathcal{P}_\mathrm{f}$. Both are basic process monads, but they also satisfy the additional axioms

$$\mu_X \circ T\varsigma_X = \varsigma_X \circ (\mu_X \times \mu_X) \circ \langle T\pi_1, T\pi_2 \rangle, \qquad \mu_X \circ T\delta_X \circ T! = \delta_X \circ !$$

that are the equivalent to right distribution of $+$ and δ over the $\cdot$ operator.

This example might be generalised as follows. Given a monad T with a distributive law on the (finite) powerset $\lambda : T \circ \mathcal{P} \longrightarrow \mathcal{P} \circ T$, it is well-known that $\mathcal{P} \circ T$ is a monad [23]. Moreover, T has a basic process monad structure inherited from $\mathcal{P}$. This is true in particular of theories with linear equations [23]. Somewhat similar in spirit is the following example.

▶ **Example 3.4.** The continuation monad with answer type a semilattice with unit R, $\mathrm{Cont}_R X = R^{R^X}$, has a basic process monad structure. The zero operation is given by the constant function returning the unit of the semilattice, while the addition is obtained by pointwise application of the semilattice operation. In particular, for $R = 2$ there are two natural basic process monad structures given by conjunction and disjunction.

We can define a category of basic process monads, where the morphisms are monad morphisms which additionally preserve the additive structure, and perform an analysis similar to what Uustalu [29] did for other kinds of non-determinism. For example, Eilenberg-Moore algebras for a basic process monad have the structure of a semilattice with unit by using ς and δ. Similarly, we obtain the following result.

▶ **Lemma 3.5.** *The initial object in the category of basic process monads is $\mathcal{P}_f$.*

Proof. Let $(T, \mu, \eta, \varsigma, \delta)$ be a basic process monad. The unique morphism $¡_T : \mathcal{P}_f \to T$ is given by assigning the set $\{x_1, \ldots, x_n\}$ to $\varsigma(\eta(x_1), \varsigma(\eta(x_2), \ldots \varsigma(\eta(x_{n-1}), \eta(x_n))))$. This morphism is uniquely defined by the fact that it has to map $\{x\}$ to $\eta(x)$. ◀

Using the formula for the coproduct of monads with a free monad [19], we can construct a basic process monad on a functor.

▶ **Example 3.6.** Let $F : \mathbf{Set} \to \mathbf{Set}$ be a functor such that the free monad on F exists. The coproduct between the free monad on F and $\mathcal{P}_f$ is a basic process monad, and its carrier functor has the following form:

$$F^+ X = \mu Z.\mathcal{P}_f(X + FZ)$$

The formula is similar to that of free (list) non-deterministic monads, but replacing the list constructor with the finite powerset. The coproduct between $\mathcal{P}_f$ and a free monad always exists (Corollary VIII.14 in [1]).

We verify that our definition is on the right track by showing that a version of Wolff's lemma hold, i.e. that we recover a BPA_δ model from applying a basic process monad to the singleton 1.

▶ **Lemma 3.7.** *Let (T, μ, η) be a basic process monad. Then $T1$ is a model of* BPA_δ.

One might wonder if there are two basic process algebras on $T1$ for a basic process monad, as there are two monoid structures on a monad applied to 1. In this case, the lack of symmetry of BPA_δ with respect to $+$ and δ rule out the alternative structure.

In the spirit of Example 2.3, where we obtain a monad from a monoid, we might wonder if we can obtain a basic process monad from a BPA_δ model. In this case, setting $T_B X = B \times X$ would not work, as there are no natural morphisms $T_B X \times T_B X \to T_B X$ and $1 \to T_B X$.

3.2 Process Algebra

Having generalised from basic process algebras to basic process monads, we focus on the next step, which is to define a free merge operation that works by taking two processes and expressing all the possible interleavings between them. The axiom system PA_δ consists of the axioms for BPA_δ together with operations

$$\| : P \times P \longrightarrow P, \qquad \lfloor\!\lfloor : P \times P \longrightarrow P$$

called *merge* and *left merge* respectively, subject to the following axioms:

$$x \,\|\, y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x \tag{M1}$$

$$a \mathbin{\lfloor\!\lfloor} x = a \cdot x \tag{M2}$$

$$(a \cdot x) \mathbin{\lfloor\!\lfloor} y = a \cdot (x \,\|\, y) \tag{M3}$$

$$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z \tag{M4}$$

The variable a used in axioms M2 and M3 is restricted to atomic actions. These equations specify the operations $\|$ and $\lfloor\!\lfloor$ in the initial model of BPA_δ.

We might try to translate this model to monads. Our point of departure will be a basic process monad, and then we will add two operations with signature

$$m : TX \times TY \longrightarrow T(X \times Y), \qquad m^\ell : TX \times TY \longrightarrow T(X \times Y).$$

These will correspond to operators $\|$ and $\lfloor\!\lfloor$ respectively. We quickly get stuck if we attempt to write the PA_δ axioms. The main problem is that we do not have a way to refer to the atomic actions as in axioms M2 and M3. The solution we propose is to work with monads T which come together with a natural transformation $a : A \longrightarrow T$ that represent the inclusion of the atomic actions abstracted by a functor A. This functor, for example, could represent a signature of algebraic operations supported by the monad T. In this way, we can write the axioms M2 and M3 using the natural transformation a.

▶ **Definition 3.8** (Process monad)**.** *A basic process monad (T, μ, η) is said to be a* process monad *over the functor A if it comes equipped with an inclusion of atomic actions*

$$a : A \longrightarrow T$$

together with natural transformations

$$m : TX \times TY \longrightarrow T(X \times Y), \qquad m^\ell : TX \times TY \longrightarrow T(X \times Y).$$

such that the following equations hold.

$$m_{X,Y} = \varsigma_{X,Y} \circ (m^\ell_{X,Y} \times (T\gamma \circ m^\ell_{Y,X} \circ \gamma)) \circ \Delta \tag{M1'}$$

$$m^\ell_{X,Y} \circ (a_X \times \mathrm{id}) = \varphi \circ (a_X \times \mathrm{id}) \tag{M2'}$$

$$m^\ell_{X,Y} \circ ((\mu_X \circ a_{TX}) \times \mathrm{id}) = \mu_{X\times Y} \circ a_{T(X\times Y)} \circ A m^\ell_{X,Y} \circ \sigma_{TX,TY} \tag{M3'}$$

$$m^\ell_{X,Y} \circ (\varsigma_X \times \mathrm{id}) = \varsigma_{X\times Y} \circ (m^\ell_{X,Y} \times m^\ell_{X,Y}) \circ \mathrm{dist} \tag{M4'}$$

where $\text{dist} = \langle\langle \pi_1 \circ \pi_1, \pi_2\rangle, \langle \pi_2 \circ \pi_1, \pi_2\rangle\rangle$ *distributes the cartesian product on itself.*

In this case, the transformations m and m^ℓ are not algebraic operations, as their signatures do not match the pattern $(TX)^a \to (TX)^b$ natural in X.

As we are adding structure to our monad, examples become scarce. The following sophisticated example, given by the resumption monad transformer on a basic process monad, is quite general and covers all the cases known to us.

▶ **Example 3.9.** Let (T, μ, η) be a basic process monad. Then, the resumption monad transformer on T

$$R_T X = \mu Z.T(X + Z)$$

has a process monad structure on the atomic actions given by T. It was shown by Cenciarelli and Moggi [7] that R_T has monad structure, which can be seen as the coproduct in the category of monads of T and the free monad on the identity functor. The additive structure ς and δ is defined from the structure T, while the merging operators m and m^ℓ are defined by mutual induction (the implementation is analogous to the merge operation in [27]).

This example is related to the construction of permuting parsers introduced by Swierstra and Dijkstra [28], which was presented more abstractly in the setting of near-semirings [27]. A similar structure has been studied by Goncharov and Schröder when considering models of processes using a coinductive construction [11].

▶ **Example 3.10.** As a particular case, the resumption monad transformer can be applied to a basic process monad constructed on a functor F as in Example 3.6. In that case, using the diagonal lemma, we obtain a process monad from a functor $F : \mathbf{Set} \to \mathbf{Set}$ as follows:

$$R_F X = \mu Z.\mathcal{P}_f(X + Z + FZ)$$

Just as for basic process monads, Wolff's lemma can be adapted to the case of process monads, which give rise to a model of PA_δ when applied to 1.

▶ **Lemma 3.11.** *Let* (T, μ, η) *be a process monad on the functor* A*. Then* $T1$ *is a model of* PA_δ *on the set of actions* $A1$*.*

4 Monads and concurrent monoids

We now move to an alternative approach to concurrency that was introduced by Hoare et al [15]. In this approach, a simple model for concurrency is given based on classical algebraic structures. Interestingly, from this model one can recover the laws of separation logic. We will be interested in two of their algebraic structures: ordered bimonoids and concurrent monoids. Ordered bimonoids capture the notion of a set equipped with two operations that behave like sequencing and merging, and equipped with a notion of order which measures the degree of sequentiality imposed on its elements. Concurrent monoids are a particular case of ordered bimonoids in which the units of the two monoid structures coincide and a natural rule of interchange holds.

The first denotational accounts of semantics of concurrency used sets of computation traces, and concurrency was handled using the merge operation we saw in the previous section. This semantics reduces the notion of concurrency to that of interleaving, as it is expressed in terms of non-determinism using the $+$ operation. To avoid this kind of reduction, the natural choice is to look for an algebraic structure that has only concurrency and sequential composition.

It was well established that sequencing and concurrency were monoidal structures, with the latter being commutative, but the relation between them was missing. A quick thought might indicate that an interchange equation $(a * b); (c * d) = (a; c) * (b; d)$ might hold. However, under the presence of this kind of law, one could apply the Eckmann-Hilton argument and show that both structures coincide:

▶ **Theorem 4.1** (Eckmann-Hilton argument). *Let X be a set with two binary operations ; and $*$ such that ; has a unit $e_;$, $*$ has a unit e_* and the interchange law $(a*b);(c*d) = (a;c)*(b;d)$ holds. Then, both operations ; and $*$ coincide, and they are both commutative and associative.*

Proof. We first show that both units coincide:

$$e_; = e_;; e_; = (e_* * e_;); (e_; * e_*) = (e_*; e_;) * (e_;; e_*) = e_* * e_* = e_*$$

Since units coincide we simply call it e. We now show that both operations coincide:

$$a; b = (e * a); (b * e) = (e; b) * (a; e) = b * a = (b; e) * (e; a) = (b * e); (e * a) = b; a$$

From the same argument we see that the operation must be commutative. The proof of associativity is analogous. ◀

This argument then shows that the interchange law cannot be used to model concurrency: otherwise not only sequencing and concurrency would coincide, but also sequencing would become commutative, which is unacceptable. However, some sort of relation between both operations must be imposed, and the interchange equation is not completely misguided. The solution proposed to handle independence, or *true concurrency*, was to consider an order on processes.The idea of concurrent monoids is to use this order and *weaken* the interchange equation using this order.

As we are using an order combined with algebraic structures, we need a notion of compatibility of operations with an order. Let $(A, \sqsubseteq)$ be a partial order, then we say that an operation $\oplus : A \times A \to A$ is *compatible* with the order if $a \sqsubseteq b$ and $c \sqsubseteq d$ implies that $a \oplus c \sqsubseteq b \oplus d$. We define a first approximation to the notion of a concurrent monoid, which has two monoidal structures, and an order compatible with them, but no special relation between them.

▶ **Definition 4.2** (Ordered bimonoid). *An* ordered bimonoid *is a partially ordered set $(A, \sqsubseteq)$ together with monoid structures $(A, *, \text{nothing})$ and $(A, ;, \text{skip})$ such that $*$ and ; are compatible with $\sqsubseteq$ and $*$ is commutative.*

This notion of ordered bimonoid was used in the setting of languages with a merge operation [5], although they required nothing = skip.

▶ **Definition 4.3** (Concurrent monoid). *A* concurrent monoid *is an ordered bimonoid such that both units coincide, i.e.* nothing = skip*, and the following* interchange law

$$(a * b); (c * d) \sqsubseteq (a; c) * (b; d) \qquad \textit{holds.}$$

As we said, in this structure, unlike process algebras, there is no reduction of the operation $*$ to an interleaving of the ; operator. The order is bridging both structures without reducing one to the other. Hoare et al. [15] develop their theory around two main models for ordered bimonoids, which we introduce now.

▶ **Example 4.4** (Trace model)**.** Fix a set A. Then Traces_A is the set of finite sequences over A, i.e. the free monoid on A. The finite sets of traces, which is described as the powerset $\mathcal{P}_{\mathrm{f}}(\mathrm{Traces}_A)$, is the carrier set of the model. The interleaving $P * Q$ is the set of interleaving of traces in P and Q, and the sequencing $P; Q$ is the set of concatenation of traces in P and Q. The set $\{\varepsilon\}$ consisting of only the empty word is the unit for both ; and $*$. The order is given by subset inclusion.

▶ **Example 4.5** (Resource model)**.** Let $(\Sigma, \bullet, u)$ be a partial commutative monoid. The powerset $\mathcal{P}(\Sigma)$ has an ordered commutative monoid structure $(*, \mathrm{emp})$ defined by $p * q = \{\sigma_0 \bullet \sigma_1 \mid (\sigma_0, \sigma_1) \in \mathrm{dom}(\bullet) \wedge \sigma_0 \in p \wedge \sigma_1 \in q\}$ and $\mathrm{emp} = \{u\}$. The set of monotone maps $\mathcal{P}(\Sigma) \longrightarrow \mathcal{P}(\Sigma)$ is the carrier set of the model. The operations and their units are defined by the following formulas

$$
\begin{aligned}
(F * G)\ Y &= \bigcup \{FY_1 * GY_2 \mid Y_1 * Y_2 \subseteq Y\} \\
\mathrm{nothing}\ Y &= Y \cap \mathrm{emp} \\
(F; G)\ Y &= F(G(Y)) \\
\mathrm{skip}\ Y &= Y
\end{aligned}
$$

The order on the model is given by the reverse pointwise ordering, i.e. $F \sqsubseteq G$ is $\forall X \subseteq \mathcal{P}(\Sigma), FX \supseteq GX$.

Clearly in the traces model both units coincide, and the interchange law holds, and thus it is a concurrent monoid. In the resource model we see that ; and $*$ have distinct units.

We are interested in lifting the ordered bimonoid and concurrent monoid structures at the level of functors, in a similar fashion to what we did in Section 3 with process algebras. As before, the sequence operator ; will be modelled after the monadic multiplication and its unit skip after the monadic unit. Three ingredients are missing: the merging operation $*$, the merging unit nothing, and the order $\sqsubseteq$. The merging operation will be modelled as the free merge operation we introduced for process algebras, i.e. a natural family $m_{A,B} : TA \times TB \longrightarrow T(A \times B)$. As we require that $*$ is a commutative monoid, in this case we will require that T has a symmetric monoidal functor structure, including a unit morphism $e : 1 \longrightarrow T1$ that will represent skip.

The order structure was not present in the process algebra monad (at least not explicitly, just as the order is not explicitly presented in the algebraic presentation of semilattices). Intuitively, what we need now is an order on computations that measures the degree of sequentiality on them. This is an order structure $\sqsubseteq_I$ on TI for each set I, subject to some compatibility with the rest of the structure that models computations. We start by specifying the compatibility condition for the functor action of T, for which we follow Hughes and Jacobs [18].

▶ **Definition 4.6** (Ordered functor)**.** *An* order for a functor F *is an assignment of a partial order* $\sqsubseteq_I$ *on* FI *for each set* I*, such that for any morphism* $f : I \longrightarrow J$*, the morphism* $Ff : FI \longrightarrow FJ$ *is a monotone map with respect to* $\sqsubseteq_I$ *and* $\sqsubseteq_J$*.*

Notice that an order $\sqsubseteq_I$ on TI is the same as an order on the set $\mathbf{Set}(1, TI) = \mathcal{K}\ell(T)(1, I)$ which we denote by the same $\sqsubseteq_I$.

Next, we specify compatibility of the monad structure. In order to do this, we use a family of order structures which is an instance of what Katsumata and Sato defined as an ordered monad [20]. Following their discussion, we give a definition of order for a monad as follows.

▶ **Definition 4.7** (Ordered monad). *An* order for a monad (T, μ, η) *is an assignment of an order* $\sqsubseteq_I$ *on* TI *for each set* I*, such that the Kleisli category of* T *is enriched in the category of orders with the order corresponding to* $\mathcal{K}\ell(T)(A, B)$ *defined by* $f \sqsubseteq_{A,B} g$ *iff* $\forall a : 1 \to A, f \circ a \sqsubseteq_B g \circ a$.

▶ Remark 4.8. There is no need to require that T is an ordered functor: this automatically holds as the functor action T might be expressed as $T(f)(v) = (\eta \circ f) \bullet \widetilde{v}$ with $\widetilde{v} : 1 \to TA$ representing the element $v \in TA$.

▶ Remark 4.9. Goncharov and Schröder [12] require an additional condition on the monadic strength for compatibility with the order. In our setting this is not necessary as the strength might be represented in terms of functor application (see Section 2) and, as shown in the previous remark, the underlying functor is already compatible with the order.

The notion of order defined above only relates to the monadic structure. Comparing with ordered bimonoids, it corresponds to the condition of ; being compatible with $\sqsubseteq$. The compatibility between $*$ and $\sqsubseteq$ is still missing. We postulate it as follows.

▶ **Definition 4.10** (Ordered monoidal functor). *Let* T *be a functor with monoidal structure* (m, e) *and an assignment of an order* $\sqsubseteq_I$ *on* TI *to each set* I*. We say that the order is compatible with* (m, e) *if* $v \sqsubseteq_A v'$ *and* $w \sqsubseteq_B w'$ *imply that* $m \circ \langle v, w\rangle \sqsubseteq_{A\times B} m \circ \langle v', w'\rangle$ *for any* $v, v' : 1 \longrightarrow TA$ *and* $w, w' : 1 \longrightarrow TB$.

With all these ingredients, we define monad variants of ordered bimonoids and concurrent monoids as follows.

▶ **Definition 4.11** (Ordered monoidal monad). *A monad* $(T, \text{-}^{\#}, \eta)$ *is an* ordered monoidal monad *if it is endowed with a symmetric monoidal functor structure*

$$m_{X,Y} : TX \times TY \longrightarrow T(X \times Y), \qquad e : 1 \longrightarrow T1$$

and an order relation $\sqsubseteq$ *on* T *compatible with* (m, e).

▶ Remark 4.12. We write $f \star g$ for $m \circ (f \times g)$.

We use the name *ordered monoidal monad* instead of the more "congruent" *ordered bimonad* to avoid confusion with *bimonads* [6].

▶ **Definition 4.13** (Concurrent monad). *An ordered monoidal monad* T *is a* concurrent monad *if* $e = \eta_1 : 1 \longrightarrow T1$ *and the interchange law*

$$(h \star i) \bullet (f \star g) \sqsubseteq (h \bullet f) \star (i \bullet g) \qquad \textit{holds.}$$

▶ Remark 4.14. The requirement $e = \eta_1$ defines η on all its components, as the following reasoning by Yoneda shows that there is an equivalence between natural transformations $\mathrm{Id} \longrightarrow T$ and elements in $T1$:

$$\mathrm{Nat}(\mathrm{Id}, T) \cong \mathrm{Nat}(\mathbf{Set}(1, -), T) \cong T1 \cong \mathbf{Set}(1, T1)$$

This implies that most of the time the reasoning on η can only be done on e, and the result follows from naturality.

We can test our definition by showing that commutative monads (as commutative monoids) are concurrent monads (as concurrent monoids).

▶ **Example 4.15.** Let (T, μ, η) be a commutative monad. Then T has a unique monoidal monad structure (m, η_1), and we can define the order structure by the diagonal (equality) order. The interchange law reduces to the monoidal monad conditions.

As we did with basic process monads and process monads, we want to show that these structures at the monad level are generalising those at the set level.

▶ **Lemma 4.16.** *Let (T, μ, η) be an ordered monoidal monad (concurrent monad). Then $T1$ is an ordered bimonoid (concurrent monoid).*

Proof. As we remarked before, $T1 \cong \mathbf{Set}(1, T1)$, so we give directly the structure on the latter form. The order $\sqsubseteq$ is defined directly from the order enrichment on $\mathbf{Set}(1, T1) = \mathcal{K}\ell(T)(1, 1)$. For elements $p, q : 1 \to T1$, we define

$$p; q = q \bullet p, \qquad p * q = T! \circ (p \star q) \circ \Delta.$$

The units are skip $= \eta_1$ and nothing $= e$. ◀

As in the case of monoids, there are two ordered bimonoid structures on $T1$, which result from the symmetry of ordered bimonoid axioms.

In this structure, as in monoids, we can also go in the other direction: given an ordered bimonoid, we can lift it to an ordered monoidal monad.

▶ **Lemma 4.17.** *Let $(A, \sqsubseteq, *, \text{nothing}, ;, \text{skip})$ be an ordered bimonoid. Then we can turn it into an ordered monoidal monad with support functor $T_A X = A \times X$, operations and order. Moreover, if* nothing $=$ skip *(i.e. it is a concurrent monoid), then $e = \eta_1$ (i.e. it is a concurrent monad).*

The trace semantics example might be generalised to a concurrent monad by parametrising the set on which we take the traces.

▶ **Example 4.18.** The monad $\mathrm{Tr}_L(X) = \mathcal{P}_\mathrm{f}(\mathrm{Traces}_{L \times X})$ is concurrent. The order structure is defined by subset inclusion as before.

In this last example we have a notion of associative, commutative and idempotent addition on each $\mathrm{Tr}(X)$ given by set union $x \oplus y = x \cup y$. We can recover the subset inclusion ordering by noticing that $x \subseteq y$ iff $x \oplus y = y$, or equivalently [2], by $x \sqsubseteq y$ iff there exists z such that $x \oplus z = y$. This observation might be used to define orders on a monad. The existence of this addition might indicate a connection to process monads introduced in Section 3. More precisely, given a process monad T on A, the question is whether T is a concurrent monad with the order induced by ς and taking m of the concurrent monad to be m of the process monad. If this was true, then the set $T1$ must have a concurrent monoid structure with $* = \|$. The main problem then is that a concurrent monoid requires that ; is compatible with the order, which seems to conflict with this order. For instance, take a, b, and c atoms, and $x = a$ and $y = a + b$. Obviously, we have that $x \sqsubseteq y$ according to the order. However, if we want ; to be compatible with $\sqsubseteq$, we need to show that $c; x \sqsubseteq c; y$. This reduces to show that $c; a + z = c; (a + b)$ for some z, but this is not true in general. Hence the question of the relation between process monads and concurrent monads is still open.

5 Reasoning with Plotkin triples and the lax slice

In the preliminaries we introduced the Kleisli category of a monad. In the situation of an ordered monad T, the corresponding Kleisli category has a richer structure: it has an

ordered category structure. That means, given two objects A and B in $\mathcal{K}\ell(T)$, the hom-set $\mathcal{K}\ell(T)(A,B) = \mathbf{Set}(A,TB)$ is not only a set, but an ordered set. It complies with the following structure:

▶ **Definition 5.1** (Ordered category). *An* ordered category *is a category $\mathcal{C}$ in which each hom-set $\mathcal{C}(X,Y)$ has an order structure such that compositions*

$$\circ : \mathcal{C}(Y,Z) \times \mathcal{C}(X,Y) \longrightarrow C(X,Z) \qquad \textit{are order preserving.}$$

Then, by definition, we have that $\mathcal{K}\ell(T)$ is an ordered category. As ordered categories are a special case of 2-categories in which the local categories are thin, many of the constructions of 2-categories might be applied in $\mathcal{K}\ell(T)$. A particular case is the lax slice construction, which, given an object X on an ordered category, produces a new category as follows.

▶ **Definition 5.2** (Lax slice category). *Let $\mathcal{C}$ be an ordered category, and X an object of it. We define the* lax slice category *$\mathcal{C} /\!\!/ X$ which has as objects morphisms $\phi : A \to X$ in $\mathcal{C}$ and morphisms from $\phi : A \to X$ to $\psi : B \to X$ are the morphisms $c : A \to B$ in $\mathcal{C}$ such that $\psi \circ c \sqsubseteq \phi$ in the order $\mathcal{C}(A,X)$. The composition of morphisms is defined by the composition in $\mathcal{C}$, and the same with identities.*

This definition is almost the same as the usual slice construction, with the only difference that we have replaced equality by $\sqsubseteq$, hence obtaining more morphisms between objects.

We apply this construction to the Kleisli category $\mathcal{K}\ell(T)$ of a concurrent monad T. We take the slice construction using the terminal object 1, i.e. $\mathcal{K}\ell(T) /\!\!/ 1$. This structure can be related to the Plotkin triples studied by Hoare et al. [15] in the setting of concurrent monoids.

▶ **Definition 5.3** (Plotkin triples). *Let M be a concurrent monoid and ϕ, c and ψ elements of it. Then, we say that a* Plotkin triple *$\phi \to_c \psi$ holds if $\phi \sqsupseteq c;\psi$.*

The category $\mathcal{K}\ell(T) /\!\!/ 1$ might be seen as a typed version of Plotkin triples, where a triple $\phi \to_c \psi$ corresponds to a morphism $c : A \to TB$ from $\phi : A \to_T 1$ to $\psi : B \to_T 1$ in $\mathcal{K}\ell(T) /\!\!/ 1$. Using the lax slice category, we have a natural separation between predicates and programs, the first are represented by objects, while the latter by morphisms. Hoare et al. [15] have shown that there is a number of rules from separation logic that hold in a concurrent monoid if they are presented as Plotkin triples. We briefly state this result in the following theorem.

▶ **Theorem 5.4** (Hoare et al. [15]). *Given a concurrent monoid M, the following rules of separation logic hold:*
Consequence. $\phi \to_c \psi \wedge \phi \sqsubseteq \phi' \wedge \psi' \sqsubseteq \psi \quad \Rightarrow \quad \phi' \to_c \psi'$.
Skip. $\phi \to_{\mathrm{skip}} \phi$.
Sequencing. $\phi \to_c \psi \wedge \psi \to_{c'} \chi \quad \Rightarrow \quad \phi \to_{c;c'} \chi$.
Concurrency. $\phi \to_c \psi \wedge \phi' \to_{c'} \psi' \quad \Rightarrow \quad \phi * \phi' \to_{c*c'} \psi * \psi'$.
We replicate this theorem at the level of the category $\mathcal{K}\ell(T) /\!\!/ 1$. We introduce the notation $(\phi : A) \rightsquigarrow_c (\psi : B)$ to denote the fact that c is a morphism in $\mathcal{K}\ell(T) /\!\!/ 1$ from $\phi : A \to_T 1$ to $\psi : B \to_T 1$. We also have to take into account the natural replacements for the concurrent monoid operations at the level of programs: skip becomes η, sequencing $c;c'$ becomes $c' \bullet c$, and merging $c * c'$ becomes $c \star c'$. At the level of predicates, we need a replacement for $*$ in the concurrency rule: given predicates $\phi : A \to_T 1$ and $\phi' : C \to_T 1$, we need to form a new predicate on 1 that combines these two. We will encode it as the following morphism

$$A \times C \xrightarrow{\phi \times \phi'} T1 \times T1 \xrightarrow{m} T(1 \times 1) \xrightarrow{T!} T1$$

In conclusion, we have that there is an operator $*$ between predicates defined by $\phi * \phi' = T! \circ m \circ (\phi \times \phi') = T! \circ (\phi \star \phi')$. Doing all these replacements, we can postulate a monadic version of Theorem 5.4.

▶ **Theorem 5.5.** *Let T be a concurrent monad, the following rules of separation logic in $\mathcal{K}\ell(T) /\!\!/ 1$ hold:*
Consequence. $(\phi : A) \rightsquigarrow_c (\psi : B) \wedge \phi \sqsubseteq \phi' \wedge \psi' \sqsubseteq \psi \quad \Rightarrow \quad (\phi' : A) \rightsquigarrow_c (\psi' : B)$.
Skip. $(\phi : A) \rightsquigarrow_\eta (\phi : A)$.
Sequencing. $(\phi : A) \rightsquigarrow_c (\psi : B) \wedge (\psi : B) \rightsquigarrow_{c'} (\chi : C) \quad \Rightarrow \quad (\phi : A) \rightsquigarrow_{c' \bullet c} (\chi : C)$.
Concurrency. $(\phi : A) \rightsquigarrow_c (\psi : B) \wedge (\phi' : C) \rightsquigarrow_{c'} (\psi' : D) \quad \Rightarrow$
$(\phi * \phi' : A \times C) \rightsquigarrow_{c \star c'} (\psi * \psi' : B \times D)$.

Proof. Most cases are straightforward, so we only give the most interesting one: concurrency. We need to prove that given $\psi \bullet c \sqsubseteq \phi$ and $\psi' \bullet c' \sqsubseteq \phi'$, we have that

$$(\psi * \psi') \bullet (c \star c') \sqsubseteq \phi * \phi'.$$

Expanding the definitions and using the exchange law, we obtain the proof

$$\begin{aligned}(\psi * \psi') \bullet (c \star c') &= (T! \circ m \circ (\psi \times \psi')) \bullet (c \star c') \\ &= T! \circ ((\psi \star \psi') \bullet (c \star c')) \\ &\sqsubseteq_{A \times C} T! \circ ((\psi \bullet c) \star (\psi' \bullet c')) \\ &\sqsubseteq_{A \times C} T! \circ (\phi \star \phi') \\ &= \phi * \phi'\end{aligned}$$

◀

The reasoning presented by Plotkin triples by Hoare et al. follows a similar structure to postulating the existence of morphisms in the category $\mathcal{K}\ell(T) /\!\!/ 1$. In particular, if we focus only on $\mathcal{K}\ell(T)(1,1)$, we obtain the same structure as Plotkin triples in the concurrent monoid $T1$.

6 Future and related work

In this article we have described two notions of monad with a merging structure. The two of them replicate at the level of functors algebraic structures used to represent concurrency at the level of sets. Building on the usual relation between monads and monoids, we have introduced the notion of process monad and concurrent monad. The first reduces concurrency to non-determinism, as the usual interleaving semantics does. The latter, instead, uses a notion of order which intuitively measures simultaneity of events, and we could understand it as true concurrency. We have also shown that part of the reasoning tools for concurrent monoids can be understood in the monadic presentation as well. Many aspects remain as further work, we discuss them briefly in the next paragraphs, providing the related work as we describe it.

Concurrent Kleene Algebras

Hoare et al. have worked with further structure than concurrent monoids [16, 17]. Indeed they define a notion of Concurrent Kleene Algebra (CKA), which is a concurrent monoid which also has a notion of addition $+$ and Kleene's star iteration. It would be interesting to see this whole structure replicated at the level of monads. Goncharov studied in [10, 13] the notion of Kleene monad which captures a monadic version of Kleene algebras.

Reasoning

The order structure on a monad provides the basis for reasoning which we would like to explore further in the presence of a merging operator. Hasuo [14] studies a notion of generic weakest precondition structure for an ordered monad. Goncharov and Schröder [12] provide a complete calculus for Hoare logic when the monad has a complete semi-lattice order structure and discuss the connection to Hasuo's approach. One of the main differences is that Goncharov and Schröder assume that there is a submonad relating to predicates, which seems to work in the same spirit as the set A of assertions in a weak CKA [25].

Categorical Structures

It is well-known that if T is a monoidal monad, then $\mathcal{K}\ell(T)$ has monoidal structure. Another well-known result is that when $\mathcal{C}$ is a monoidal category, then $\mathcal{C}/M$ is a monoidal category if M has a monoid structure in $\mathcal{C}$. These two results seem to be the basis for the triples presented in Section 5. It would be interesting to generalise the result to strong monads on other categories than **Set** and other monoidal structures than cartesian product. Another categorical structure that seems to be related is that of duoidal categories, which was used by Garner and López Franco to study commutativity [9].

Functional Programming

A further point we would like to explore is the relation of these models with the practical approach of functional programming to concurrency. For example, Haskell's package `async` provides a concurrent primitive `concurrently :: IO a -> IO b -> IO (a,b)`; we would like to investigate if it is possible to understand it in terms of concurrent monads.

References

1 Jirí Adámek, Stefan Milius, Nathan Bowler, and Paul Blain Levy. Coproducts of monads on set. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 45–54, 2012.

2 J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009.

3 J. C. M. Baeten and C. Verhoef. Handbook of logic in computer science (vol. 4). chapter Concrete Process Algebra, pages 149–268. Oxford University Press, Inc., New York, NY, USA, 1995.

4 J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109 – 137, 1984.

5 Stephen L. Bloom and Zoltán Ésik. Free shuffle algebras in language varieties. *Theoretical Computer Science*, 163(1):55 – 98, 1996.

6 Alain Bruguières and Alexis Virelizier. Hopf monads. *Advances in Mathematics*, 215(2):679 – 733, 2007.

7 Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. *Category Theory and Computer Science*, 1993.

8 Fredrik Dahlqvist and Renato Neves. Compositional semantics for new paradigms: probabilistic, hybrid and beyond. *CoRR*, abs/1804.04145, 2018. `arXiv:1804.04145`.

9 Richard Garner and Ignacio López Franco. Commutativity. *Journal of Pure and Applied Algebra*, 220(5):1707 – 1751, 2016.

10 Sergey Goncharov. *Kleene monads*. PhD thesis, Universität Bremen; www.uni-bremen.de, 2010.

11 Sergey Goncharov and Lutz Schröder. A coinductive calculus for asynchronous side-effecting processes. *Inf. Comput.*, 231:204–232, October 2013.

12 Sergey Goncharov and Lutz Schröder. A relatively complete generic Hoare logic for order-enriched effects. In *Proc. 28th Annual Symposium on Logic in Computer Science (LICS 2013)*, pages 273–282. IEEE, 2013.

13 Sergey Goncharov, Lutz Schröder, and Till Mossakowski. Kleene monads: Handling iteration in a framework of generic effects. In Alexander Kurz and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science (CALCO 2009)*, volume 5728 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

14 Ichiro Hasuo. Generic weakest precondition semantics from monads enriched with order. *Theoretical Computer Science*, 604(C):2–29, November 2015.

15 C. A. R. Hoare, Akbar Hussain, Bernhard Möller, Peter W. O'Hearn, Rasmus Lerchedahl Petersen, and Georg Struth. On locality and the exchange law for concurrent processes. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, pages 250–264, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

16 C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene algebra. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, pages 399–414, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

17 C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene algebra and its foundations. *The Journal of Logic and Algebraic Programming*, 80(6):266 – 296, 2011. Relations and Kleene Algebras in Computer Science.

18 Jesse Hughes and Bart Jacobs. Simulations in coalgebra. *Theoretical Computer Science*, 327(1):71 – 108, 2004. Selected Papers of CMCS '03.

19 Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70 – 99, 2006. Clifford Lectures and the Mathematical Foundations of Programming Semantics.

20 Shin-ya Katsumata and Tetsuya Sato. Preorders on monads and coalgebraic simulations. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures*, pages 145–160, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

21 Anders Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23:113–120, 1970.

22 E. G. Manes. *Algebraic theories*. Applied Mathematical Sciences. Springer Verlag, 1976.

23 Ernest Manes and Philip Mulry. Monad compositions I: general constructions and recursive distributive laws. 18:172–208, 04 2007.

24 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55 – 92, 1991.

25 Peter W. O'Hearn, Rasmus L. Petersen, Jules Villard, and Akbar Hussain. On the relation between concurrent separation logic and concurrent Kleene algebra. *Journal of Logical and Algebraic Methods in Programming*, 84(3):285 – 302, 2015. 13th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2012).

26 Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, Feb 2003.

27 Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. A unified view of monadic and applicative non-determinism. *Science of Computer Programming*, 152(C):70–98, January 2018.

28 Doaitse S. Swierstra and Atze Dijkstra. Parse your options. In *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*, pages 234–249, Berlin, Heidelberg, 2013. Springer-Verlag.

29 Tarmo Uustalu. A divertimento on monadplus and nondeterminism. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):1086 – 1094, 2016. Articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday.

30 Philip Wadler. Comprehending monads. In *1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78. ACM, 1990.

31 Harvey Wolff. Monads and monoids on symmetric monoidal closed categories. *Archiv der Mathematik*, 24(1):113–120, Dec 1973.