Concurrent Monads for Shared State

Exequiel Rivas
Tallinn University of Technology
Tallinn, Estonia
exequiel.rivas@taltech.ee

Tarmo Uustalu
Reykjavik University
Reykjavik, Iceland
Tallinn University of Technology
Tallinn, Estonia
tarmo@ru.is

ABSTRACT

In the context of programming with effects, sequential composition takes a special role as the primary control structure for combining computations. In this article, we advocate the idea that parallel composition should also be treated as a control structure, on the same footing as sequential composition. We promote the concept of concurrent monad, which axiomatizes both sequential and parallel composition, and illustrate the approach by describing two concurrent monads for interleaving shared state concurrency: one of resumptions, the other of multisets of traces.

CCS CONCEPTS

• **Theory of computation** → **Categorical semantics**; *Control primitives*.

KEYWORDS

effectful computation, concurrency, parallel composition, concurrent monoids, concurrent monads, duoidal categories

ACM Reference Format:
Exequiel Rivas and Tarmo Uustalu. 2024. Concurrent Monads for Shared State. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper is about *concurrency* in effectful computation and programming. We ask this question: Is concurrency, together with the associating primitives, such as, first of all, *parallel composition* of two effectful functions, an *effect*? More precisely, is it an effect that could be "encapsulated" in a *monad* or an *algebraic theory* similarly to exceptions, interactive input-output or state manipulation, within the standard *Moggi and Plotkin–Power approach* [22, 25] to effectfulness? If it were, it could be treated exactly the same way as these example effects. It is not unnatural to dream that this be the case.

Our answer and message in this paper is: No. Parallel composition is fundamentally a *higher-level control structure* than (low-level) "prefixing" an effect request—such as an exception raise, or an input or output operation, or a read or write operation—to a computation. Parallel composition is analogous and comparable to *sequential composition* and also other familiar control structures from high-level programming, like *iteration.* This has big consequences.

Consider this. We accept that sequential composition is something fundamentally different from an effect, namely, a high-level control structure for putting two effectful computations together into one bigger such. And monads provide an *axiomatization* of this control structure for context of effectful computation. In fact, it is their only mission in the Moggi approach, they do nothing else. In particular, monads do not axiomatize parallel composition (and not iteration either, for example). Hence, if we want to talk about both sequential and parallel composition at the same time, we should need a different mathematical structure. This structure could be monads with additional structure, to also axiomatize parallel composition alongside sequential composition, which is already there in a monad, as well as the interaction of these two compositions.

This view was first proposed and worked out by Rivas and Jaskelioff [29]. They were the first to define a specialization of monads, called *concurrent monads*, that add to monads the axiomatization of parallel composition and the interaction of sequential and parallel compositions. Paquet and Saville [24] recently considerably elaborated the theory.

In a nutshell, when a monad T on category $\mathbb{C}$ provides an operation that takes maps $X \to TY$ and $Y \to TZ$, embodying effectful functions, into a map $X \to TZ$, their sequential composition, a concurrent monad goes further and also provides a parallel composition operation that takes maps $X \to TY$ and $U \to TV$ into a map $X \otimes U \to T(Y \otimes V)$ (where $\otimes$ could, for example, be the product $\times$). A price that we pay in this paper is that $\mathbb{C}$ and T have to be an *ordered* monoidal category and functor (i.e., **Poset**-enriched, meaning that there is an order on the set of maps $X \to Y$ for any two objects X, Y resp. that the action of T on maps is monotone).[1]

There is a clear analogy and inspiration here from *Kleene algebra*. Kleene algebras axiomatize sequential composition, nondeterminism and finite iteration. *Concurrent Kleene algebras* by Hoare et al. [16] add parallel composition very much in the same spirit as our move from monads to concurrent monads; the crucial substructure of a concurrent Kleene algebra is a *concurrent monoid.* In fact, in this paper, we first generalize from concurrent monoids to what we call *concurrent categories* (by "typing" concurrent monoids) and then define concurrent monads so that the Kleisli construction yields a concurrent category. To use Kleene algebra as an inspiration for specializations of monads to capture high-level control structures

[1] In the original work of Rivas and Jaskelioff [29], the base category $\mathbb{C}$ was the ordinary category **Set**, but the Kleisli category was made ordered by requiring that the ordinary functor T factored through U : **Poset** $\to$ **Set**. Instead of **Poset**-enriched, one can also go *bicategorical.* This is finer, and it is what Paquet and Saville [24] have done, but also a lot more involved.

Conference'17, July 2017, Washington, DC, USA

https://doi.org/10.1145/nnnnnnn.nnnnnnn

is not a new idea. The major line of work of Goncharov [11, 13] on *Kleene monads* (which axiomatize Kleene-like finite iteration) and Elgot monads (which axiomatize possibly infinite Elgot iteration, controlled by decisions) is rooted in the Kleene algebra work in exactly this fashion.

We ground our message about concurrent monads in a few examples about *shared-state concurrency*. We consider two kinds. We first show discuss a very naive type of shared-state concurrency where two effectful functions put in parallel work independently and upon completion their final states are merged. Then we proceed to standard preemptive *interleaving* shared-state concurrency. We consider two semantics, *resumptions* based and *collections-of-traces* (multisets of traces) based.

We have implemented our development, in particular all examples, in Haskell. In our code, we have pretended that we can work in the ordered monoidal category **Poset** in Haskell, which is of course an approximation. The code is available from https://cs.ioc.ee/~tarmo/concurmonads/.

The paper has the following structure. First we recall concurrent monoids and look at some examples thereof. Then we generalize concurrent monoids to concurrent categories by typing them. We then define concurrent monads so that the Kleisli category of a concurrent monad is a concurrent category. By reorganizing the data of a concurrent monad we show that it is precisely a concurrent monoid object in the suitable structure that is available on the category of endofunctors. We then proceed to examples of concurrent monads for shared state. We first consider naive shared-state concurrency and then proceed to resumptions-based and collections-of-traces based semantics of preemptive interleaving shared-stated concurrency.

The paper uses some category theory that is not so much advanced but subtle. We have tried hard to make this paper and its message accessible to functional programmers and Kleene algebraists. Therefore we have suppressed some material (such as full lists of coherence conditions in some definitions), but we have unpacked the most critical definitions to the level of the most basic concepts to be able to pinpoint the fine details that make the setup work as we need. We have omitted all proofs, but point to the conceptual reasons that make the proofs work.

To better align with the common practice in effectful programming (imperative or functional, like in Haskell, with the do-notation for programming in Kleisli categories) and also Kleene algebra—and commutative diagrams of course—, we write the vertical composition (of maps and natural transformations) in the diagrammatic order, denoting it by $(\,;\,)$. The horizontal composition (of functors and natural transformations) is written in the usual applicative order and denoted by $(\cdot)$.

2 CONCURRENT MONOIDS

Concurrent monoids are the key part of the structure of concurrent Kleene algebras by Hoare et al. [16], which, in addition to sequential and parallel composition, also axiomatize nondeterministic choice and finite iteration. But they go back to Gischer [10].

In a nutshell, a concurrent monoid is an ordered set (poset) with two ordered monoid (pomonoid) structures on it, one for sequential, the other for parallel composition, agreeing in a certain way.[2]

Precisely, a *concurrent monoid* is an ordered set $(M, \leq)$ with ordered monoid structures $(\mathsf{id}, (;))$ and $(\mathsf{jd}, \|)$ satisfying *inequational interchange*:

$$\begin{array}{rcll} \mathsf{id} &\leq& \mathsf{jd} & \text{(ich-1)}\\ \mathsf{jd}\,;\mathsf{jd} &\leq& \mathsf{jd} & \text{(ich-2)}\\ \mathsf{id} &\leq& \mathsf{id} \parallel \mathsf{id} & \text{(ich-3)}\\ (k \parallel \ell)\,;(m \parallel n) &\leq& (k\,;m) \parallel (\ell\,;n) & \text{(ich-4)} \end{array}$$

An important special and common case is when inequation (ich-1) holds as an equality, i.e., $\mathsf{id} = \mathsf{jd}$. In this case, the concurrent monoid is said to be *normal*.

The structure $(\mathsf{id}, (;))$ being an ordered monoid means that id is an element of M and ; is a function $M \times M \to M$ such that ; is monotone, unital wrt. id and associative:

$$\text{if } k \leq k' \text{ and } \ell \leq \ell', \text{ then } k;\ell \leq k';\ell'$$
$$\mathsf{id}; k = k \quad k; \mathsf{id} = k \quad (k;\ell); m = k; (\ell; m)$$

Notice that it does not ask for id to be the top or bottom of the order, or even comparable with all other elements. Analogous conditions must hold about $(\mathsf{jd}, \|)$.

Intuitively, we should think of the elements of M as commands or untyped effectful functions, (;) as sequential composition, $\|$ as parallel composition and id and jd as the units of these compositions (perhaps both "doing nothing"). The relationship $k \leq k'$ should be understood to mean that k accomplishes less than k' in some way.

The inequational axioms of a concurrent monoid are not entirely independent. From inequation (ich-4) (known as *middle four interchange*), inequation (ich-1) is derivable:

$$\mathsf{id} = \mathsf{id}\,;\mathsf{id} = (\mathsf{jd} \parallel \mathsf{id})\,;(\mathsf{id} \parallel \mathsf{jd}) \leq (\mathsf{jd}\,;\mathsf{id}) \parallel (\mathsf{id}\,;\mathsf{jd}) = \mathsf{jd} \parallel \mathsf{jd} = \mathsf{jd}$$

By an obvious modification of this calculation, when (ich-4) holds as an equality for a concurrent monoid, then so does (ich-1).

(ich-1) entails the converses of (ich-2) and (ich-3):

$$\mathsf{jd} = \mathsf{id};\mathsf{jd} \leq \mathsf{jd};\mathsf{jd} \qquad \mathsf{id} \parallel \mathsf{id} \leq \mathsf{jd} \parallel \mathsf{id} = \mathsf{id}$$

Therefore (ich-2) and (ich-3) entail their own equational variants

$$\mathsf{jd}\,;\mathsf{jd} = \mathsf{jd} \qquad \mathsf{id} = \mathsf{id} \parallel \mathsf{id}$$

(But note that we only get equality here because we have assumed antisymmetry. The development in the next section shows that the equality here is even more "incidental".)

When (ich-1) holds as an equality for a concurrent monoid, then (ich-4) has as consequences

$$k\,;\ell = (k \parallel \mathsf{jd})\,;(\mathsf{jd} \parallel \ell) \leq (k\,;\mathsf{id}) \parallel (\mathsf{id}\,;\ell) = k \parallel \ell$$

and likewise

$$\ell\,;k = (\mathsf{jd} \parallel \ell)\,;(k \parallel \mathsf{jd}) \leq (\mathsf{id}\,;k) \parallel (\ell\,;\mathsf{id}) = k \parallel \ell$$

By the above observations, when (ich-4) holds as an equality for a concurrent monoid, then so do all others. Moreover, then $k\,;\ell = k \parallel \ell = \ell\,;k$. Thereby the two ordered monoid structures

[2] By 'ordered', we mean partially ordered. Preorders would work just as well, but here we ignore this opportunity for generality. The ordered monoid for parallel composition is typically required to be commutative, but in this work we find it generally unnecessarily restrictive to ask this.

have collapsed into one commutative ordered monoid. This fact with its proof is known as the Eckmann-Hilton argument.

If the order of a concurrent monoid is discrete (i.e., $\leq$ is $=$), then any inequation trivially becomes an equation. So concurrent monoids whose order is discrete are the same as commutative (discretely ordered) monoids.

We see that concurrent monoids are only interesting when their order is not discrete and (ich-4) and maybe also (ich-1) do not hold as equalities.

Concurrent monoids, especially normal concurrent monoids, are not uncommon in nature. Here are some examples.

Any idempotent semiring $(S, 0, +, 1, \times)$ is a concurrent monoid—for $\leq$ defined by $m \leq n$ iff $m + n = n$. In this example, $(;) = +$ is commutative as $+$ is so in any semiring.

The idempotent Boolean semiring $(\mathbb{B}, \mathrm{ff}, \vee, \mathrm{tt}, \wedge)$ is an example of a concurrent monoid like this. The order $\leq$ is $\supset$, which is generated by $\mathrm{ff} \supset \mathrm{tt}$. This concurrent monoid is non-normal and has both $(;) = \vee$ and $\| = \wedge$ commutative and idempotent.

With multivalued logics, one can build further examples of concurrent monoids. A three-valued logic using one (viz., the secondary one) of the two disjunction-conjunction pairs of the Łukasiewicz logic gives an example where $(;) = \vee$ and $\| = \wedge$ are both non-idempotent but still commutative. This concurrent monoid is $(\{\mathrm{ff}, \mathrm{u}, \mathrm{tt}\}, \supset, \mathrm{ff}, \vee, \mathrm{tt}, \wedge)$ where the order $\supset$ is generated by $\mathrm{ff} \supset m$, $m \supset \mathrm{tt}$ and the connectives $\vee$ and $\wedge$ are defined by the truth-tables

$\vee$	ff	u	tt
ff	ff	u	tt
u	u	tt	tt
tt	tt	tt	tt

$\wedge$	ff	u	tt
ff	ff	ff	ff
u	ff	ff	u
tt	ff	u	tt

Non-commutativity is not achievable with three truth-values. But a four-valued variation (where the disjunction and conjunction mix features of both disjunction-conjunction pairs of the Łukasiewicz logic) gives us an example where $(;) = \vee$ and $\| = \wedge$ are both non-idempotent and also non-commutative. We use $(\{\mathrm{ff}, \mathrm{u_L}, \mathrm{u_R}, \mathrm{tt}\}, \supset, \mathrm{ff}, \vee, \mathrm{tt}, \wedge)$ where $\supset$ is generated by $\mathrm{ff} \supset m$, $m \supset \mathrm{tt}$, $\mathrm{u_L} \supset \mathrm{u_R}$ and $\vee$ and $\wedge$ are defined by the truth-tables

$\vee$	ff	$\mathrm{u_L}$	$\mathrm{u_R}$	tt
ff	ff	$\mathrm{u_L}$	$\mathrm{u_R}$	tt
$\mathrm{u_L}$	$\mathrm{u_L}$	$\mathrm{u_L}$	$\mathrm{u_R}$	tt
$\mathrm{u_R}$	$\mathrm{u_R}$	tt	tt	tt
tt	tt	tt	tt	tt

$\wedge$	ff	$\mathrm{u_L}$	$\mathrm{u_R}$	tt
ff	ff	ff	ff	ff
$\mathrm{u_L}$	ff	ff	$\mathrm{u_L}$	$\mathrm{u_L}$
$\mathrm{u_R}$	ff	ff	$\mathrm{u_R}$	$\mathrm{u_R}$
tt	ff	$\mathrm{u_L}$	$\mathrm{u_R}$	tt

To conclude this section with a computationally motivated example, one of the main motivating examples for concurrent Kleene algebra, we switch from mathematics to Haskell. We thereby also start to introduce the code accompanying this paper. In Haskell, we can use the following type-classes to define ordered sets and concurrent monoids:

```
class Ordered m where
  (=<) :: m → m → Bool
class Ordered m ⇒ ConcurMonoid m where
  idS :: m
  (>.>) :: m → m → m
  idP :: m
  (<|>) :: m → m → m
```

where idS and idP corrrespond to id and jd respectively, and >.> and <|> to (;) and ‖. For programming, we require orders to be decidable and represent them accordingly as Boolean-valued functions. Of course we cannot insist in Haskell that the data of a structure satisfy the required axioms.

Our programming example of a concurrent monoid is given by the set $\mathcal{M}_\mathrm{f}(A^*)$ of finite multisets (bags) of lists over a fixed set A, where A can be an alphabet of actions and lists over A can serve as traces. In Haskell, finite multisets will be lists with a nonstandard equality relation that ignores the order of the elements. The order is multiset inclusion.

```
newtype Bag a = B { unB :: [a] } deriving Show
instance Eq a ⇒ Eq (Bag a) where
  B [] == B [] = True
  B [] == _    = False
  B (x:xs) == B ys = case remove x ys of
                       Nothing → False
                       Just ys' → B xs == B ys'
instance Eq a ⇒ Ordered (Bag a) where
  B [] =< _ = True
  B (x:xs) =< B ys = case remove x ys of
                       Nothing → False
                       Just ys' → B xs =< B ys
```

The possible interleavings of two traces are given by their shuffle coded like this:

```
(<||>) :: [a] → [a] → [[a]]
[] <||> ys = [ys]
xs <||> [] = [xs]
(x:xs) <||> (y:ys) = map (x:) (xs <||> (y:ys))
                     ++ map (y:) ((x:xs) <||> ys)
```

The concurrent monoid instance results as follows:

```
type BoT a = Bag [a]
instance Eq a ⇒ ConcurMonoid (BoT a) where
  idS = B [[]]
  (B xss) >.> (B yss) = B [xs ++ ys | xs <- xss, ys <- yss]
  idP = B [[]]
  B xss <|> B yss = B (concat [xs <||> ys | xs <- xss, ys <- yss])
```

In this example, the units id and jd coincide, so the concurrent monoid is normal. We need to use $\mathcal{M}_\mathrm{f}(A^*)$ (finite multisets) rather than $(A^*)^*$ (lists) in order to obtain associativity of $\|$. To be a legitimate order on bags, $\leq$ has to ignore the order of elements in the lists representing bags. (Also, (ich-4) does not actually hold for the finer order on lists given by order-preserving inclusion.) But of course $\mathcal{P}_\mathrm{f}(A^*)$ (the finite powerset of A) instead of $\mathcal{M}_\mathrm{f}(A^*)$ works too.

3 CONCURRENT MONADS

We will now proceed to concurrent monads à la Rivas and Jaskelioff [29]. We show to generalize concurrent monoids to concurrent categories. And then we specialize this generalization for computation with effects, arriving at concurrent monads.[3]

[3] For the reader that knows strong monads well, this story will appear very similar both by the spirit and the mathematical means employed to that connecting Freyd (a.k.a. effectful) categories and strong monads [27, 28]. Paquet and Saville [23, 24] have

3.1 Concurrent categories

The generalization of concurrent monoids to concurrent categories goes by "typing" them. The idea is that effectful functions, differently from commands, do not just produce effects, but also take argument data to return data.

An ordered monoid $(M, \leq, \mathsf{id}, ;)$ is a one-object ordered category with elements of M as maps and ; as composition. We now switch from one object to many objects. But we must also take care of the $(\mathsf{jd}, \|)$ ingredients of a concurrent monoid. For that we use that a monoid can also be viewed also a discrete (necessarily strict) monoidal category. We will generalize this perspective, making sure to introduce enough generality to cover our intended applications.

In a nutshell, a concurrent category will be an *ordered* category with an ordered *monoidal-like* structure. But to formulate that structure, we need to assume a base, which will be an ordered monoidal category (of types and pure functions)[4], so the concurrent category (of generalized functions) will be parameterized in one such. Further, we require the concurrent category to come with an identity-on-objects ordered *monoidal-like* functor from its base. (We will explain what 'monoidal-like' means in each case.)

Like we said, we need to proceed from a *base*, which has to be an ordered (=**Poset**-enriched) monoidal category $\mathbb{C} = (\mathbb{C}, \leq, \mathsf{id}, (;), \mathsf{I}, \otimes)$ in the standard senses of these terms. Let us unpack this definition. It says that we have (i) an ordered category. Explicitly, that amounts to a set $|\mathbb{C}|$ of objects (types) and, for all $X, Y \in |\mathbb{C}|$, an ordered set $(\mathbb{C}(X,Y), \leq)$ of maps (pure functions) with

- maps $\mathsf{id}_X \in \mathbb{C}(X,X)$ and an operation on maps $(;) : \mathbb{C}(X,Y) \times \mathbb{C}(Y,Z) \to \mathbb{C}(X,Z)$, monotone and unital, associative.

We also have (ii) an ordered monoidal structure. This consists of two ordered functors $\mathsf{I} : \mathbf{1} \to \mathbb{C}$, $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ and three natural families of isomorphisms $\lambda_X \in \mathbb{C}(\mathsf{I} \otimes X, X)$, $\rho_X \in (X, X \otimes \mathsf{I})$ and $\alpha_{X,Y,Z} \in \mathbb{C}((X \otimes Y) \otimes Z, X \otimes (Y \otimes Z))$ satisfying certain coherence equations. Explicitly, this ordered monoidal structure adds to what we already unpacked

- an object $\mathsf{I} \in \mathbb{C}$ and an operation on objects $\otimes : |\mathbb{C}| \times |\mathbb{C}| \to |\mathbb{C}|$, unital, associative up to isomorphism (via maps λ_X, ρ_Y, $\alpha_{X,Y,Z}$ that need to be there and must be required to be isomorphisms),
- a map $\mathsf{I} \in \mathbb{C}(\mathsf{I}, \mathsf{I})$ and operation on maps $\otimes : \mathbb{C}(X,Y) \times \mathbb{C}(U,V) \to \mathbb{C}(X \otimes U, Y \otimes V)$, monotone and unital-associative up to isomorphism (capturing that λ_X, ρ_X, $\alpha_{X,Y,Z}$ are natural)

satisfying *equational* interchange:

$$\begin{array}{c} \mathsf{id}_{\mathsf{I}} = \mathsf{I} \\ \mathsf{I} \,;\, \mathsf{I} = \mathsf{I} \\ \mathsf{id}_{X \otimes Y} = \mathsf{id}_X \otimes \mathsf{id}_Y \\ (f \otimes g) \,;\, (h \otimes i) = (f \,;\, h) \otimes (g \,;\, i) \end{array}$$

for $f \in \mathbb{C}(X,Y)$, $g \in \mathbb{C}(U,V)$, $h \in \mathbb{C}(Y,Z)$, $i \in \mathbb{C}(V,W)$ (capturing that I, $\otimes$ are functorial).

provided bicategorical versions of both of these stories in their accounts of strong pseudomonads and concurrent pseudomonads (only normal such).

[4] Here will start deviating from Rivas and Jaskelioff [29] for who the base was **Set** as an ordinary (i.e., unordered) monoidal category. But the aim and the rest of the means we use are exactly the same.

Notice that the interchange axioms are exactly those of a concurrent monoid, but typed, and that here they are stipulated as equations. Notice also that, differently from the situation with concurrent monoids, although the analogue of (ich-4) is stipulated as an equation, there is no collapse here of the two "typed monoid" structures into one commutative such. This is made possibly by the two structures working in two different dimensions. While id is a family of maps, I is just one map and the analogue $\mathsf{id}_{\mathsf{I}} = \mathsf{I}$ of (ich-1) says nothing about id_X for $X \neq \mathsf{I}$. Etc. Any ordinary (unordered) monoidal category is trivially an ordered category with all homsets $\mathbb{C}(X,Y)$ are discrete (i.e, $\leq$ is $=$).

A useful ordered (which in this case means self-enriched) category is $\mathbb{C} = $ **Poset**. It has as $|\mathbb{C}|$ the class of all ordered sets, as $\mathbb{C}(X,Y)$ the set of all monotone functions $X \to Y$. The order $\leq$ on monotone functions $X \to Y$ is obtained from the order on the codomain Y, pointwise, i.e., $f \leq g$ for monotone $f, g : X \to Y$ iff $fx \leq_Y gx$. As an ordered monoidal structure $(\mathsf{I}, \otimes)$, this ordered category has the finite-product structure $(1, \times)$, which is given by finite products on the underlying sets, with $(x,y) \leq_{X \times Y} (x', y')$ iff $x \leq_X x'$ and $y \leq_Y y'$.

The category **Set** carries a trivial ordered structure with every homset ordered discretely. As such, it is ordered isomorphic to the full sub- ordered category of **Poset** given by discretely ordered sets (full because any function between discretely ordered sets is trivially monotone).

We are now prepared to proceed to concurrent categories.

We define a *concurrent category* is an ordered *monoidal-like* category $\mathbb{K} = (\mathbb{K}, \leq^{\mathbb{K}}, \mathsf{id}^{\mathbb{K}}, (;^{\mathbb{K}}), \mathsf{I}^{\mathbb{K}}, \otimes^{\mathbb{K}})$ together with an identity-on-objects ordered *monoidal-like* functor J from the presupposed base $(\mathbb{C}, \leq, \mathsf{id}, (;), \mathsf{I}, \otimes)$. We spell out the data and also make it precise what we mean by 'monoidal-like' in each case.

$\mathbb{K}$ as (i) an ordered *category* must have the *same objects* as $\mathbb{C}$, i.e., $|\mathbb{K}| = |\mathbb{C}|$. In addition it has, for any $X, Y \in |\mathbb{K}| = |\mathbb{C}|$, an ordered set $(\mathbb{K}(X,Y), \leq^{\mathbb{K}})$ of maps and

- maps $\mathsf{id}^{\mathbb{K}} \in \mathbb{K}(X,X)$ and an operation on maps $;^{\mathbb{K}} : \mathbb{K}(X,Y) \times \mathbb{K}(Y,Z) \to \mathbb{K}(X,Z)$, monotone and unital-associative

As (ii) an ordered *monoidal-like* structure, $\mathbb{K}$ must have the object $\mathsf{I}^{\mathbb{K}} \in \mathbb{K}$ and the operation on objects $\otimes^{\mathbb{K}} : |\mathbb{K}| \times |\mathbb{K}| \to |\mathbb{K}|$ the *same* as in $\mathbb{C}$, i.e., $\mathsf{I}^{\mathbb{K}} = \mathsf{I}$ and $\otimes^{\mathbb{K}} = \otimes$. The latter operation is automatically unital and associative (in $\mathbb{K}$!) up to isomorphism via $J\lambda$, $J\rho$, $J\alpha$. In addition, it has

- a map $\mathsf{I}^{\mathbb{K}} \in \mathbb{K}(\mathsf{I}, \mathsf{I})$, for which we write jd for brevity and emphasis, and an operation on maps $\otimes^{\mathbb{K}} : \mathbb{K}(X,Y) \times \mathbb{K}(U,V) \to \mathbb{K}(X \otimes U, Y \otimes V)$, for which we write $\|$, monotone and unital, associative up to isomorphism using $J\lambda$, $J\rho$, $J\alpha$ (this expresses naturality of $J\lambda_X$, $J\rho_X$, $J\alpha_{X,Y,Z}$)

satisfying *inequational* interchange:

$$\begin{array}{cr} \mathsf{id}^{\mathbb{K}}_{\mathsf{I}} \leq^{\mathbb{K}} \mathsf{jd} & \text{(ich-1)} \\ \mathsf{jd} \,;^{\mathbb{K}} \mathsf{jd} \leq^{\mathbb{K}} \mathsf{jd} & \text{(ich-2)} \\ \mathsf{id}^{\mathbb{K}}_{X \otimes Y} \leq^{\mathbb{K}} \mathsf{id}^{\mathbb{K}}_X \parallel \mathsf{id}^{\mathbb{K}}_Y & \text{(ich-3)} \\ (k \parallel \ell) \,;^{\mathbb{K}} (m \parallel n) \leq^{\mathbb{K}} (k \,;^{\mathbb{K}} m) \parallel (\ell \,;^{\mathbb{K}} n) & \text{(ich-4)} \end{array}$$

for $k \in \mathbb{K}(X,Y)$, $\ell \in \mathbb{K}(U,V)$, $m \in \mathbb{K}(Y,Z)$, $n \in \mathbb{K}(V,W)$ (stipulating that $\mathsf{I}^{\mathbb{K}}$, $\otimes^{\mathbb{K}}$ are functorial laxly, i.e., preserve $\mathsf{id}^{\mathbb{K}}$ and $;^{\mathbb{K}}$ only inequationally). The structure is monoidal-like in that $\mathsf{I}^{\mathbb{K}}$ and $\otimes^{\mathbb{K}}$ *lax functors* rather than ordered functors.

J has to be (i) an *identity-an-objects* ordered *functor*: it must have $JX = X$ for all $X \in |\mathbb{C}|$ and be monotone and preserve id and (;) properly (that is, equationally), i.e., if $f \leq g$, then $Jf \leq^{\mathbb{K}} Jg$ for $f, g \in \mathbb{C}(X, Y)$; $J\mathsf{id} = \mathsf{id}^{\mathbb{K}}$, $J(f \mathbin{;} g) = Jf \mathbin{;^{\mathbb{K}}} Jg$.

J must also be (ii) ordered *monoidal-like*. Here monoidal-likeness means that J must be strict monoidal on objects in the sense of having as its monoidality constraints $\mathsf{id}^{\mathbb{K}}_{\mathsf{I}}$ and $\mathsf{id}^{\mathbb{K}}_{-\otimes-}$, which is enabled by $J\mathsf{I} = \mathsf{I}$ and $J(X \otimes Y) = X \otimes Y = JX \otimes JY$. But on maps J can be *oplax monoidal* in the sense of preserving I and $\otimes$ only inequationally like this: $J\mathsf{I} \leq^{\mathbb{K}} \mathsf{jd}$ and $J(f \otimes g) \leq^{\mathbb{K}} Jf \parallel Jg$.[5]

We say that a concurrent category is *normal* if (ich-1) holds as an equation, i.e., $\mathsf{id}^{\mathbb{K}}_{\mathsf{I}} = \mathsf{jd}$.

A concurrent category is necessarily normal when J is an honest strict monoidal functor, i.e., if $J\mathsf{I} = \mathsf{jd}$ and $J(f \otimes g) = Jf \parallel Jg$, because then $\mathsf{id}_{\mathsf{I}} = \mathsf{I} = J\mathsf{I} = \mathsf{jd}$. In this case, we moreover have that (ich-4) holds as an equality for maps in the image of J because (ich-4) is an equation in $\mathbb{C}$:

$$
\begin{array}{rcl}
(Jf \parallel Jg) \mathbin{;^{\mathbb{K}}} (Jh \parallel Ji) & = & J(f \otimes g) \mathbin{;^{\mathbb{K}}} J(h \parallel i) \\
& = & J((f \otimes g) \mathbin{;} (h \parallel i)) \\
& = & J((f \mathbin{;} h) \otimes J(g \otimes i)) \\
& = & J(f \mathbin{;} h) \parallel J(g \otimes i) \\
& = & (Jf \mathbin{;^{\mathbb{K}}} Jh) \parallel (Jg \mathbin{;^{\mathbb{K}}} Jh)
\end{array}
$$

Concurrent categories, so defined with inequational interchange, are now the real typed generalization of concurrent monoids. The stronger-structured base is somewhat auxiliary. Apart from just fixing a minimal pure core, the base provides, for example, the concurrent category with its unitors and associator, which must sensibly be pure.[6]

Like we already discussed concerning the base, there is by far not as much redundancy and collapse in the axioms in a concurrent category than in a concurrent monoid, but some remains. (ich-1) remains a consequence (ich-4) and thus redundant as an axiom. (ich-1) itself entails that the converse of (ich-2), so the the latter is in fact valid as an equation

$$\mathsf{jd} \mathbin{;^{\mathbb{K}}} \mathsf{jd} = \mathsf{jd}$$

(ich-1) also entails the converse of (ich-3), but only for $X = Y = \mathsf{Id}$, so we only get a special case of (ich-3) as a valid equation

$$\mathsf{id}^{\mathbb{K}}_{I} = \mathsf{id}^{\mathbb{K}}_{I} \parallel \mathsf{id}^{\mathbb{K}}_{I}$$

From (ich-4), it almost immediately follows that we have

$$(k \parallel \mathsf{id}^{\mathbb{K}}_{U}) \mathbin{;^{\mathbb{K}}} (\mathsf{id}^{\mathbb{K}}_{Y} \parallel \ell) \leq^{\mathbb{K}} k \parallel \ell \quad (\mathsf{id}^{\mathbb{K}}_{X} \parallel \ell) \mathbin{;^{\mathbb{K}}} (k \parallel \mathsf{id}^{\mathbb{K}}_{V}) \leq^{\mathbb{K}} k \parallel \ell$$

for $k \in \mathbb{K}(X, Y)$, $\ell \in \mathbb{K}(U, V)$.

We postpone examples of concurrent categories until we have defined concurrent monads and looked at their Kleisli construction. But the intuition is this: the Kleisli construction of a concurrent monad will give a concurrent category and this is a place where have effectful functions that can be composed both sequentially and in parallel exactly as promised in the introduction.

3.2 Concurrent monads

We now want a specialization of the notion of ordered monad such that is Kleisli category is a concurrent category.

The game is this. We proceed from an ordered monoidal category $\mathbb{C}$. We want that a concurrent monad T on $\mathbb{C}$ will have as its Kleisli ordered category $\mathbf{Kl}(T)$ and the associating left adjoint ordered functor $J : \mathbb{C} \to \mathbf{Kl}(T)$ a concurrent category with $\mathbb{C}$ as the base. Of course a concurrent monad should be an ordered monad with additional structure and the concurrent category should be the Kleisli category of this ordered monad with additional structure.

We want this because then the Kleisli ordered category will have $|\mathbf{Kl}(T)| = |\mathbb{C}|$, $\mathbf{Kl}(T)(X, Y) = \mathbb{C}(X, TY)$, $k \leq^{\mathbb{K}} \ell$ in $\mathbf{Kl}(T)(X, Y)$ iff $k \leq \ell$. It will come with the familiar Kleisli data

- a family of maps $\mathsf{id}^{\mathbb{K}} \in \mathbb{C}(X, TX)$ and an operation on maps $(;^{\mathbb{K}}) : \mathbb{C}(X, TY) \times \mathbb{C}(Y, TZ) \to \mathbb{C}(X, TZ)$, monotone, unital and associative

as we are used to for an ordered monad and giving us sequential composition of effectful functions. But it will also come with further data

- a map $\mathsf{jd} \in \mathbb{C}(\mathsf{I}, T\mathsf{I})$ and an operation on maps $\parallel : \mathbb{C}(X, TY) \times \mathbb{C}(U, TV) \to \mathbb{C}(X \otimes U, T(Y \otimes V))$, monotone, unital and associative

giving us also parallel composition and together with $\mathsf{id}^{\mathbb{K}}$, $;^{\mathbb{K}}$ satisfying the desirable inequational interchange.

We will in a minute present the definition of concurrent monads that meets these criteria, but first some intuition. The intuition for why the definition will do the right thing is similar to the case of ordered monads. The datum μ (multiplication) and the derived datum $(-)^*$ (Kleisli extension) of an ordered monad are what they are very much because of the (natural in the free, i.e., the unmentioned, variables) bijections between homsets

$$\frac{\mathbb{C}(X, TY) \times \mathbb{C}(TY, TZ) \to \mathbb{C}(X, TZ) \text{ nat. in } X \text{ in } \mathbf{Poset}}{\mathbb{C}(Y, TZ) \to \mathbb{C}(TY, TZ) \text{ in } \mathbf{Poset}}$$

and

$$\frac{\mathbb{C}(Y, TZ) \to \mathbb{C}(TY, TZ) \text{ nat. in } Y \text{ in } \mathbf{Poset}}{T(TZ) \to TZ \text{ in } \mathbb{C}}$$

that are instances of the Yoneda lemma. Notice here that maps in **Poset** are monotone functions between ordered sets.

A concurrent monad will have the additional data of a lax monoidal functor.[7] The datum ψ below is motivated by the following similar natural bijection of homsets:

$$\frac{\mathbb{C}(X, TY) \times \mathbb{C}(U, TV) \to \mathbb{C}(X \otimes U, T(Y \otimes V)) \text{ nat. in } X, U \text{ in } \mathbf{Poset}}{TY \otimes TV \to T(Y \otimes V) \text{ in } \mathbb{C}}$$

[5] There seems to be no established terminology, not to speak of good such, for this kind of monoidal-likeness. First of all, J is an honest ordered functor, not oplax. It is also not oplax monoidal in that we would have non-identity monoidality constraints $J\mathsf{I} \to \mathsf{I}$ and $J(X \otimes Y) \to JX \otimes JY$. Oplax monoidality here is in a different dimension, namely order, not maps, similarly to the laxity of $\mathsf{I}^{\mathbb{K}}$ and $\otimes^{\mathbb{K}}$, but for those the name 'lax functors' is correct standard terminology. Something similar happens in [7].

[6] Instead of two ordered categories $\mathbb{C}$ and $\mathbb{K}$ and an identity-on-objects ordered functor J, one can also work with just one ordered category $\mathbb{K}$ with a wide sub- ordered category, but this does not model the case when J is not faithful. In terms of the Kleisli construction of the next subsection, that happens when the unit η of the concurrent monad is not mono.

[7] In Haskell, bistrong lax monoidal functors are known as 'applicative functors' and 'monads' (which are actually bistrong monads) are treated as a specialization of 'applicative functors'. This is justified because every bistrong monad gives two lax monoidal functors with same underlying bistrong functor and, if these two happen to be the same, then the monad is lax monoidal (commutative in the sense of Kock [19]). A lax monoidal functor need not be bistrong, but a lax monoidal monad is always a bistrong monad. But nothing like this is the case for general (= non-bistrong) monads, so Haskell's terminology is dangerous.

We are ready to go. We define a *concurrent monad* on a base ordered monoidal category $(\mathbb{C}, \leq, \mathsf{I}, \otimes)$ to be an *ordered monad* (T, η, μ), that is, an ordered functor $T : \mathbb{C} \to \mathbb{C}$ with a monad structure (η, μ), together with a *lax monoidal functor* structure (ψ^0, ψ) on T that agree in a certain way. Explicitly, T must be an ordered functor coming equipped with

- families of maps $\eta_X : X \to TX$, $\mu_X : T(TX) \to TX$ natural in X,
- a map $\psi^0 : \mathsf{I} \to T\mathsf{I}$ and a family of maps $\psi_{X,Y} : TX \otimes TY \to T(X \otimes Y)$ natural in X, Y

satisfying equations

$$\begin{tikzcd} TX \arrow[d, "\eta_{TX}"'] \arrow[dr, equal] & TX \arrow[r, "T\eta_X"] \arrow[dr, equal] & T(TX) \arrow[d, "\mu_X"] \\ T(TX) \arrow[r, "\mu_X"'] & TX & TX \end{tikzcd} \qquad \begin{tikzcd} T(T(TX)) \arrow[r, "T\mu_X"] \arrow[d, "\mu_{TX}"'] & T(TX) \arrow[d, "\mu_X"] \\ T(TX) \arrow[r, "\mu_X"'] & TX \end{tikzcd}$$

$$\begin{tikzcd} \mathsf{I} \otimes TX \arrow[rr, "\lambda_{TX}"] \arrow[d, "\psi^0 \times TX"'] & & TX \arrow[d, equal] \\ T\mathsf{I} \otimes TX \arrow[r, "\psi_{\mathsf{I},X}"'] & T(\mathsf{I} \otimes X) \arrow[r, "T\lambda_X"'] & TX \end{tikzcd} \qquad \begin{tikzcd} TX \arrow[r, "\rho_{TX}"] \arrow[d, equal] & TX \otimes \mathsf{I} \arrow[r, "TX \otimes \psi^0"] & TX \otimes T\mathsf{I} \arrow[d, "\psi_{X,I}"] \\ TX \arrow[rr, "T\rho_X"'] & & T(X \otimes \mathsf{I}) \end{tikzcd}$$

$$\begin{tikzcd} (TX \otimes TY) \otimes TZ \arrow[r, "\alpha_{TX,TY,TZ}"] \arrow[d, "\psi_{X,Y} \otimes TZ"'] & TX \otimes (TY \otimes TZ) \arrow[r, "TX \otimes \psi_{Y,Z}"] & TX \otimes T(Y \otimes Z) \arrow[d, "\psi_{X,Y \otimes Z}"] \\ T(X \otimes Y) \otimes TZ \arrow[r, "\psi_{X \otimes Y,Z}"'] & T((X \otimes Y) \otimes TZ) \arrow[r, "T\alpha_{X,Y,Z}"'] & T(X \otimes (Y \otimes Z)) \end{tikzcd}$$

(these are the usual monad and lax monoidal functor data and equations) and also the interchange inequations

$$\begin{tikzcd} \mathsf{I} \arrow[r, equal] \arrow[d, equal] \arrow[dr, phantom, "\geq"] & \mathsf{I} \arrow[d, "\eta_I"] \\ \mathsf{I} \arrow[r, "\psi^0"'] & T\mathsf{I} \end{tikzcd} \qquad \begin{tikzcd} \mathsf{I} \arrow[r, "\psi^0"] \arrow[d, equal] & T\mathsf{I} \arrow[r, "T\psi^0"] \arrow[d, phantom, "\geq"] & T(T\mathsf{I}) \arrow[d, "\mu_I"] \\ \mathsf{I} \arrow[rr, "\psi^0"'] & & T\mathsf{I} \end{tikzcd} \qquad \begin{tikzcd} X \otimes Y \arrow[r, equal] \arrow[d, "\eta_X \otimes \eta_Y"'] \arrow[dr, phantom, "\geq"] & X \otimes Y \arrow[d, "\eta_{X \otimes Y}"] \\ TX \otimes TY \arrow[r, "\psi_{X,Y}"'] & T(X \otimes Y) \end{tikzcd}$$

$$\begin{tikzcd} T(TX) \otimes T(TY) \arrow[r, "\psi_{TX,TY}"] \arrow[d, "\mu_X \otimes \mu_Y"'] & T(TX \otimes TY) \arrow[r, "T\psi_{X,Y}"] \arrow[d, phantom, "\geq"] & T(T(X \otimes Y)) \arrow[d, "\mu_{X \otimes Y}"] \\ TX \otimes TY \arrow[rr, "\psi_{X,Y}"'] & & T(X \otimes Y) \end{tikzcd}$$

A concurrent monad is *normal* if (ich-1) holds as an equality, i.e., $\eta_{\mathsf{I}} = \psi^0$.

A concurrent monad that validates all interchange inequations as equalities is the same as an ordered lax monoidal monad (= an ordered commutative monad).

Now, as we promised, a concurrent monad $(T, \eta, \mu, \psi^0, \psi)$ on $\mathbb{C}$ induces a concurrent category $(\mathbb{K}, J)$ with base $\mathbb{C}$ via its Kleisli construction. The construction is as an extension of the familiar Kleisli construction for the ordered monad (T, η, μ).

The definition of the ordered monoidal-like category $\mathbb{K} = (\mathbf{Kl}(T), \leq^{\mathrm{K}}, \mathsf{I}^{\mathrm{K}}, \otimes^{\mathrm{K}})$ is as follows:

- $|\mathbf{Kl}(T)| = |\mathbb{C}|$,
  $\mathbf{Kl}(T)(X, Y) = \mathbb{C}(X, TY)$,
- $k \leq^{\mathrm{K}} \ell$ in $\mathbf{Kl}(T)(X, Y)$ iff $k \leq \ell$ in $\mathbb{C}(X, TY)$,
- $\mathrm{id}^{\mathrm{K}}_X = \eta_X$,
  $k \,;^{\mathrm{K}} \ell = k \,; T\ell \,; \mu_Z$ for $k \in \mathbf{Kl}(T)(X, Y)$, $\ell \in \mathbf{Kl}(T)(Y, Z)$,
- $\mathrm{jd} = \psi^0$,
  $k \parallel \ell = (k \otimes \ell) \,; \psi_{Y,V}$ for $k \in \mathbf{Kl}(T)(X, Y)$, $\ell \in \mathbf{Kl}(T)(U, V)$.

The ordered monoidal-like functor J is defined as follows:

- $JX = X$
  $Jf = f; \eta_Y$ for $f \in \mathbb{C}(X, Y)$.

The requirements that the definition of concurrent categories poses on $(\mathbb{K}, J)$ are easily verified.

The ordered functor J of this construction has a right ordered adjoint $K : \mathbf{Kl}(T) \to \mathbb{C}$ (of the Kleisli ordered adjunction). It is defined as follows:

- $KX = TX$,
  $Kk = Tk; \mu_Y$ for $k \in \mathbf{Kl}(T)(X, Y)$

K is monoidal-like in that it comes with *lax* monoidality constraints $\psi^0 : \mathsf{I} \to K\mathsf{I}$, $\psi_{X,Y} : KX \otimes KY \to K(X \otimes Y)$ and then modulo the constraints it preserves jd and $\parallel$ *oplaxly* in that $\psi^0; K\mathrm{jd} \leq \mathrm{id}_I; \psi^0$ and $\psi_{X,Y}; K(k \parallel \ell) \leq (Kk \otimes K\ell); \psi_{U,V}$ for $k \in \mathbf{Kl}(T)(X, Y)$ and $\ell \in \mathbf{Kl}(T)(U, V)$.

Conversely to the Kleisli construction, if the ordered functor component J of a concurrent category $(\mathbb{K}, J)$ with base $\mathbb{C}$ has a right adjoint K, then the ordered functor $T = K \cdot J$ carries the structure of a concurrent monad on $\mathbb{C}$ with $\mathbb{K}$ its Kleisli ordered category.

We go to concurrent monads on the ordered finite-product category $\mathbf{Poset} = (\mathbf{Poset}, \leq, 1, \times)$ in Haskell, together with the first example thereof. Unfortunately, Haskell's types and functions are more like the ordinary category **Set** rather than the ordered category **Poset**—we can equip types with an order, but only in ad hoc ways. Hence our formalization will necessarily be quite imperfect.

We use three type-classes to define concurrent monads: the **OrderedFunctor**, **Mnd** and **Lmf** type-classes.

An ordered functor on **Poset** is by definition a functor sending ordered sets to ordered sets, monotone functions to monotone functions, preserving their pointwise order. In Haskell, we subclass the type-class **OrderedFunctor** for ordered functors from **Functor**. This gives us an operation fmap sending arbitrary functions (not only monotone ones) to functions. We will add to it an operation ford that will send arbitrary decidable relations (not only decidable orders) to decidable relations.

```
class Functor f ⇒ OrderedFunctor f where
  ford :: (x → x → Bool) → (f x → f x → Bool)
```

The type-class **Mnd** is our type-class for monads, equivalent (but only because every functor in Haskell is bistrong) to the more standard **Monad** type-class found in the **Prelude**. Its directly encodes the categorical definition of monad that we have referred to above:

```
class Functor t ⇒ Mnd t where
  η :: x → t x
  μ :: t (t x) → t x
```

Finally, the third type-class is for lax monoidal functors:

```
class Functor t ⇒ Lmf t where
  ψ0 :: t ()
  ψ  :: t x → t y → t (x, y)
```

The type-class for concurrent monads just subclasses from all three:

```
class (OrderedFunctor t, Mnd t, Lmf t) ⇒ ConcurMnd t
```

Remember again that we cannot enforce in Haskell any axioms (equations or inequations) about the operations of a concurrent monad. Neither can use or enforce properties like that an argument to or the result from fmap is monotone.

The operations of the Kleisli concurrent category are defined through those of the concurrent monad.

```
(>>>) :: Mnd t ⇒ (x → t y) → (y → t z) → x → t z
k >>> l = μ . fmap l . k
jd :: Lmf t ⇒ () → t ()
jd () = ψ⁰
(|||) :: Lmf t ⇒ (x → t z) → (y → t w) → (x, y) → t (z, w)
(k ||| l) (x, y) = k x `ψ` l y
```

We now look at a few examples of concurrent monads.

Writing. Any fixed concurrent monoid $M = (M, \leq_M, \mathrm{id}, ;, \mathrm{jd}, \|)$ (which is an object of **Poset** with additional structure) induces a concurrent monad on **Poset**, extending the ordered *writer* monad on **Poset** for the ordered monoid $(M, \leq_M, \mathrm{id}, ;)$ to a concurrent monad.

The underlying ordinary functor is given on objects by

$$TX = M \times X$$

where the order on $M \times X$ is induced from the orders on M and X pointwisely. The action of T on maps is $Tf = M \times f$, i.e., $Tf(m, x) = (m, fx)$. This is well-defined since, if f is monotone, then Tf is also monotone. T qualifies as an ordered functor because, for any monotone functions $f, g : X \to Y$ such that $fx \leq gx$ for all x, we also have $Tf(m, x) \leq Tg(m, x)$ for all x. In Haskell, we define

```
type Writer m = (,) m
instance Functor (Writer m) where
  fmap f (m, x) = (m, f x)
instance ConcurMonoid m ⇒ OrderedFunctor (Writer m) where
  ford (<=.) (m, m') (x, y) = m =< m' && x <=. y
```

The concurrent monad operations of T apply the concurrent monoid operations of M to the first components of pairs as in this Haskell code:

```
instance ConcurMonoid m ⇒ Mnd (Writer m) where
  η x = (idS, x)
  μ (m₁, (m₂, x)) = (m₁ >.> m₂, x)
instance ConcurMonoid m ⇒ Lmf (Writer m) where
  ψ⁰ = (idP, ())
  (m₁, x) `ψ` (m₂, y) = (m₁ <|> m₂, (x, y))
instance ConcurMonoid m ⇒ ConcurMnd (Writer m)
```

The inequations of T hold thanks to the inequations of M. If inequations of a particular M hold as equalities, then M is a commutative monoid and the corresponding T is a commutative monad.

The programming motivation is the same as the usual one for the writer monad on **Set**. We should think of M as a set of updates. An element of $(m, x) \in TX$ is a computation that performs update m and returns x. The concurrent monoid data specify how updates combine together when performed sequentially or in parallel.

Reading. We can similarly extend the familiar *reader* ordered monad on **Poset** to a concurrent monad. Given an ordered set $(S, \leq_S)$ (where the order may very well be discrete), this concurrent monad has the underlying ordinary functor given on objects by

$$TX = S \Rightarrow X$$

and the order on TX, given pointwisely, only depends on the order on X. Notice that the set $S \Rightarrow X$ consists of monotone functions only (but if $\leq_S$ is discrete, all functions are monotone). Nothing else in the concurrent monad structure depends on the order of S. Here is the Haskell code for the concurrent monad structure, where, in order to have the pointwise order on $S \Rightarrow X$ decidable, we assume that S can be enumerated:

```
class Listable a where
  list :: [a]

type Reader s = (→) s
instance Functor (Reader s) where
  fmap f g = \ s → f (g s)
instance Listable s ⇒ OrderedFunctor (Reader s) where
  ford (=<) f g = and [ f s =< g s | s <- list ]
instance Mnd (Reader s) where
  η x = \ _ → x
  μ c = \ s → c s s
instance Lmf (Reader s) where
  ψ⁰ = \ _ → ()
  f `ψ` g = \ s → (f s, g s)
instance Listable s ⇒ ConcurMnd (Reader s)
```

This concurrent monad is normal, moreover it is just a commutative ordered monad.

State. As a final example for now, we consider an extension to a concurrent monad of the *state* ordered monad on **Poset**. This needs that the state set S carries the structure of a lower semilattice. In Haskell, we define this structure as follows.

```
class Semilattice s where
  ⊤ :: s
  ∧ :: s → s → s
instance (Eq s, Semilattice s) ⇒ Ordered s where
  s =< s' = s == s ∧ s'
```

The object mapping of the underlying ordinary functor of the concurrent monad is

$$TX = S \Rightarrow S \times X$$

where the order on TX is pointwise. Recall again that the set $S \Rightarrow S \times X$ consists of monotone functions only (but here $\leq_S$ can only be discrete if S is a singleton, else it cannot be a semilattice). Here is the Haskell code of the concurrent monad structure:

```
newtype State s x = S (s → (s, x))
instance Functor (State s) where
  fmap f (S g) = S (\ s → let (s', x) = g s in (s', f x))
instance (Eq s, Listable s, Semilattice s) ⇒
                                  OrderedFunctor (State s) where
  ford (<=.) (S f) (S g) = and [helper s | s <- list ]
    where helper s = let (s, x ) = f s in
                     let (s', x') = g s in
                     s =< s' && x <=. x'
```

```
instance Mnd (State s) where
  η x = S (\ s → (s, x))
  μ (S g) = S (\ s → let (s', S f) = g s in f s')
instance Semilattice s ⇒ Lmf (State s) where
  ψ⁰ = S (\ _ → (⊤, ()))
  S f `ψ` S g = S (\ s → let (s, x) = f s in
                          let (s', y) = g s in
                          (s ∧ s', (x, y)))
instance (Eq s, Listable s, SemiLattice s) ⇒ ConcurMnd (State s)
```

The most interesting operation is ψ, where the final states are reconciled using the meet operation of the semilattice. This concurrent monad is not normal—we have $\eta_1 \neq \psi^0$.

This concurrent monad is implements a very simplistic type of shared state concurrency. There is no interleaving. In parallel composition, both effectful functions are passed a copy of the initial state. They work independently, manipulating their version of the state. When both have terminated, they yield their final states, which are merged. Intuitively, the middle four interchange inequation holds because the more often two parallel effectful computations synchronize, the smaller according to the respective orders the final state and the return values get.

We will get to interesting interleaving shared state concurrency in section 4.

3.3 Duoidal categories, concurrent monoid objects

This section is for the categorically minded reader.

Monoids are sets with structure. Monads on a category $\mathbb{C}$ are endofunctors on $\mathbb{C}$ with structure, but they are also *monoid objects*, that is, monoids not in **Set**, but in the category $[\mathbb{C}, \mathbb{C}]$ of endofunctors on $\mathbb{C}$. But this is possible only because **Set** and $[\mathbb{C}, \mathbb{C}]$ are not just categories, but categories with sufficient structure making it possible to define what is means to be a monoid object in it. The important structure here is monoidality. We implicitly rely on **Set** carrying the $(1, \times)$ (=finite-product) monoidal structure and $[\mathbb{C}, \mathbb{C}]$ carrying the $(\mathsf{Id}, \cdot)$ (=functor composition) monoidal structure.

Concurrent monoids are ordered sets with structure. It would therefore be very nice if concurrent monads on an ordered category $\mathbb{C}$, similarly to monads, were concurrent monoids in the category $[\mathbb{C}, \mathbb{C}]$ wrt. some sufficient structure readily available there. We should at least need that $[\mathbb{C}, \mathbb{C}]$ is ordered, the monoidal structure of functor composition is relevant, and then we should likely need more. This calls for identifying the structure that is needed on a general ordered category $\mathbb{D}$ (other than **Poset**) in order for it to support a concept of concurrent monoid object. We will now develop this.

It turns out that $\mathbb{D}$ has to be ordered *duoidal* [2, 8]. The duoidal structure will allow us to define what it means for an object to carry two monoid structures, possibly not of the same nature, while the ordered structure (ordered homsets) is necessary to state the interchange inequations.

An *ordered duoidal category* is an ordered category $(\mathbb{D}, \leq)$ with two ordered monoidal structures $(I, \odot)$ and $(J, \circledast)$ (so $\odot$ and $\circledast$ in particular are ordered functors) and with maps

$$
\begin{aligned}
\iota &: J \to I \\
\nabla &: J \to J \odot J \\
\Delta &: I \circledast I \to I \\
\xi &: (X \odot Y) \circledast (Z \odot W) \to (X \circledast Z) \odot (Y \circledast W) \text{ nat. in } X, Y, Z, W
\end{aligned}
$$

satisfying certain coherence equations which require that I and $\odot$ (as functors $\mathbf{1} \to \mathbb{D}$ and $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$) are lax monoidal with respect to $(J, \circledast)$. In particular, we have that (I, ι, Δ) is a monoid wrt. $(J, \circledast)$ and (J, ι, ∇) is a comonoid wrt. $(I, \odot)$.

Notice that ι, ∇, Δ, ξ, differently from the unitors and associators of $(I, \odot)$ and $(J, \circledast)$ are just maps/natural transformations here, not isomorphisms/natural isomorphisms.

A *concurrent monoid object* in an ordered duoidal category $\mathbb{D} = (\mathbb{D}, \leq, I, \odot, J, \circledast)$ is an object M with two monoid structures (o, a), wrt. the $(I, \odot)$ monoidal structure, and (e, m), wrt. the $(J, \circledast)$ monoidal structure,[8] satisfying *inequational interchange*:

$$
\begin{aligned}
\iota; o &\leq e \\
\nabla; (e \odot e); a &\leq e \\
\Delta; o &\leq (o \circledast o); m \\
\xi; (m \odot m); a &\leq (a \circledast a); m
\end{aligned}
$$

Fully explicitly and as commutative diagrams, the unitality and associativity equations for the two monoid structures are

$$
\begin{tikzcd}
I \odot M \arrow[r, "\lambda^{\odot}"] \arrow[d, "o \odot M"'] & M \arrow[dr, equal] & & M \arrow[r, "\rho^{\odot}"] \arrow[drr, equal, bend right] & M \odot I \arrow[r, "M \odot o"] & M \odot M \arrow[d, "a"] \\
M \odot M \arrow[rr, "a"] & & M & & & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
(M \odot M) \odot M \arrow[r, "\alpha^{\odot}"] \arrow[d, "a \odot M"'] & M \odot (M \odot M) \arrow[r, "M \odot a"] & M \odot M \arrow[d, "a"] \\
M \odot M \arrow[rr, "a"] & & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
J \circledast M \arrow[r, "\lambda^{\circledast}"] \arrow[d, "e \circledast M"'] & M \arrow[dr, equal] & & M \arrow[r, "\rho^{\circledast}"] \arrow[drr, equal, bend right] & M \circledast J \arrow[r, "M \circledast e"] & M \circledast M \arrow[d, "m"] \\
M \circledast M \arrow[rr, "m"] & & M & & & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
(M \circledast M) \circledast M \arrow[r, "\alpha^{\circledast}"] \arrow[d, "m \circledast M"'] & M \circledast (M \circledast M) \arrow[r, "M \circledast m"] & M \circledast M \arrow[d, "m"] \\
M \circledast M \arrow[rr, "m"] & & M
\end{tikzcd}
$$

The inequations are

$$
\begin{tikzcd}
J \arrow[r, "\iota"] \arrow[d, equal] \arrow[dr, phantom, "\geq"] & I \arrow[d, "o"] \\
J \arrow[r, "e"] & M
\end{tikzcd}
\quad
\begin{tikzcd}
J \arrow[r, "\Delta"] \arrow[d, equal] & J \odot J \arrow[r, "e \odot e"] \arrow[d, phantom, "\geq"] & M \odot M \arrow[d, "a"] \\
J \arrow[rr, "e"] & & M
\end{tikzcd}
\quad
\begin{tikzcd}
I \circledast I \arrow[r] \arrow[d, "o \circledast o"'] \arrow[dr, phantom, "\geq"] & I \arrow[d, "o"] \\
M \circledast M \arrow[r, "m"] & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
(M \odot M) \circledast (M \odot M) \arrow[r, "\xi"] \arrow[d, "a \circledast a"'] & (M \circledast M) \odot (M \circledast M) \arrow[r, "m \odot m"] \arrow[d, phantom, "\geq"] & M \odot M \arrow[d, "a"] \\
M \circledast M \arrow[rr, "m"] & & M
\end{tikzcd}
$$

[8] We use o, a as mnemonic for zero and addition, e, m for one and multiplication.

A concurrent monoid object that validates all interchange equations as equalities is an ordered duoid object.

Concurrent monoids in the sense of section 2, which are sets with structure, are concurrent monoid objects in the duoidal category $(\mathbf{Poset}, \leq, 1, \times, 1, \times)$.

3.4 Concurrent monads as concurrent monoids

We can now show that concurrent monads are concurrent monoids.

If $\mathbb{C} = (\mathbb{C}, \leq, \mathsf{I}, \otimes)$ is small ordered cocomplete (so the ordered coends are small and exist), then the category $[\mathbb{C}, \mathbb{C}]$ of ordered functors is ordered with $\leq$ on $[\mathbb{C}, \mathbb{C}]$ the pointwise lifting of $\leq$ on $\mathbb{C}$. It carries an ordered duoidal structure $(\mathsf{Id}, \cdot, \mathsf{Jd}, \star)$ where $(\mathsf{Id}, \cdot)$ is the composition ordered strict monoidal structure and $(\mathsf{Jd}, \star)$ the Day convolution ordered monoidal structure

$$\begin{aligned} \mathsf{Jd}Z &= \mathbb{C}(\mathsf{I}, Z) \bullet \mathsf{I} \\ (F \star G)Z &= \int^{X,Y} \mathbb{C}(X \otimes Y, Z) \bullet (FX \otimes GY) \end{aligned}$$

(**Poset** and most ordered categories of interest are not small, so this is restrictive. But if $\mathbb{C}$ is ordered locally finitely presentable, then the full sub- ordered category of $[\mathbb{C}, \mathbb{C}]$ given by ordered finitary functors has the Day convolution monoidal structure existing. If $\mathbb{C}$ is ordered locally presentable, then one can restrict to ordered accessible functors.)

A concurrent monad turns out to be the same as a concurrent monoid object in this ordered duoidal category.

Unpacked, this means that a concurrent monad can be given by an ordered functor $T : \mathbb{C} \to \mathbb{C}$ with

- natural transformations $\eta : \mathsf{Id} \to T$, $\mu : T \cdot T \to T$,
- natural transformations $\mathsf{e} : \mathsf{Jd} \to T$, $\mathsf{m} : T \star T \to T$

satisfying the (in)equations of a concurrent monoid from the previous section.

We refrain from trying to show any calculations here. However, the intuitive reason why this can work is given by these natural bijections between homsets:

$$\begin{array}{c} \mathsf{I} \to T\mathsf{I} \text{ in } \mathbb{C} \\ \hline\hline \mathbb{C}(\mathsf{I}, Z) \to \mathbb{C}(\mathsf{I}, TZ) \text{ in } \mathbf{Poset} \\ \hline\hline \underbrace{\mathbb{C}(\mathsf{I}, Z) \bullet \mathsf{I}}_{\mathsf{Jd}Z} \to TZ \text{ nat. in } Z \text{ in } \mathbb{C} \end{array}$$

and

$$\begin{array}{c} TX \otimes TY \to T(X \otimes Y) \text{ nat. in } X, Y \text{ in } \mathbb{C} \\ \hline\hline \mathbb{C}(X \otimes Y, Z) \to \mathbb{C}(TX \otimes TY, TZ) \text{ nat. in } X, Y, Z \text{ in } \mathbf{Poset} \\ \hline\hline \mathbb{C}(X \otimes Y, Z) \bullet (TX \otimes TY) \to TZ \text{ nat. in } X, Y, Z \text{ in } \mathbb{C} \\ \hline\hline \underbrace{\int^{X,Y} \mathbb{C}(X \otimes Y, Z) \bullet (TX \otimes TY)}_{(T \star T)Z} \to TZ \text{ nat. in } Z \text{ in } \mathbb{C} \end{array}$$

4 CONCURRENT MONADS FOR SHARED STATE

We saw a concurrent monad on $\mathbf{Poset} = (\mathbf{Poset}, \leq, 1, \times)$ for very simplistic shared state concurrency in section 3.2, extending the classical ordered monad for state manipulation.

We will now proceed to concurrent monads for familiar (pre-emptive) interleaving shared state concurrency. We will consider two such concurrent monads, one based on resumptions, the other on finite multisets of traces.

Resumptions. Assume given an ordered set S, which we restrict to be discrete for simplicity (it is not necessary). We consider the ordinary functor T on **Poset** whose object mapping is defined as follows using initial algebras

$$TX = \mu Z.\ \underbrace{X}_{ret} \underbrace{+}_{or} (\underbrace{S \Rightarrow}_{grab} \underbrace{\mathsf{List}}_{branch} (\underbrace{S \times}_{yield} \underbrace{Z}_{repeat}))$$

with the order on TX induced by the order on X and the natural order on $\mathsf{List}Y$ for any ordered set Y given by order-preserving inclusion between lists (all elements of one list must occur in the other, in the same order). Because S is discrete, one can construct the underlying set of TX as an initial algebra in **Set** and then assign it an order according to the order of X. The functorial action of T is obvious. T is ordered because, if $f, g : X \to Y$ satisfy $f \leq g$, then $Tf \leq Tg$.

Intuitively, elements of TX should be understood as resumptions. The idea of resumptions that they are descriptions of computations in terms of small steps. A resumption is always either termination or a small step and then a resumption again. In our case, they are wellfounded trees of a certain kind. A resumption $r \in TX$ is either of the form $\mathsf{inl}\ x$, signifying termination with value x, or of the form $\mathsf{inr}\ k$ where k is a function sending any state to a list of state-resumption pairs. The idea is that, in this case, the resumption consists in "grabbing" a state s from the environment (the current state), then branching nondeterministically and then "yielding" a state s' to the environment (the next state) together with a new resumption. The nondeterminism is there because a resumption may arise from combining two resumptions in parallel, which gives multiple outcomes corresponding to different interleavings. That we use List rather than $\mathcal{M}_\mathrm{f}$ or $\mathcal{P}_\mathrm{f}$ signifies that we have chosen to observe both the multiplicity and order of the outcomes of nondeterministic branching. Our resumptions are much like synchronization trees from concurrency theory, but they also carry return values.

Here is the Haskell code for T as an ordered functor:

```
data Res s x = Ret x | Grab (s → [(s, Res s x)])
instance Functor (Res s) where
  fmap f (Ret x)   = Ret (f x)
  fmap f (Grab k) =
    Grab (\s → [ (s', fmap f r) | (s', r) <- k s ])
instance (Listable s, Eq s) ⇒ OrderedFunctor (Res s) where
  ford (<=.) (Ret x)  (Ret y)  = x <=. y
  ford (<=.) (Ret _)  (Grab _) = False
  ford (<=.) (Grab _) (Ret _)  = False
  ford (<=.) (Grab k) (Grab l) =
    and [helper (k s) (l s) | s <- list ] where
      helper []    _  = True
      helper (_:_) [] = False
      helper ((s,r):ps) ((s',r'):ps') = s == s' && ford (<=.) r r'
                                                && helper ps ps'
                                        || helper ((s,r):ps) ps'
```

A reader knowledgeable about monads will have noticed that we have constructed T like the underlying ordered functor of the free ordered monad on the ordered functor

$$FZ = X + S \Rightarrow \mathsf{List}(S \times Z)$$

Thus we readily have a canonical ordered monad structure on T. Its multiplication, corresponding to sequential composition of computations, is the customary grafting of trees onto a tree. In Haskell, we define

```
instance Mnd (Res s) where
  η x = Ret x
  μ (Ret r)  = r
  μ (Grab k) = Grab (\ s → [ (s', μ r) | (s', r) <− k s ])
```

The lax monoidal functor structure on T is given by interleaving resumptions. Namely, when we combine two resumptions r, r' of the forms inr k and inr ℓ in parallel, this involves a choice of who gets to make the first small step. All remaining steps remain to be made in the continuation. This grows the nondeterminism. The Haskell code is this:

```
merge :: Res s x → Res s y → Res s (x, y)
Ret x `merge` Ret y  = Ret (x, y)
Ret x `merge` Grab l =
    Grab (\ s → [ (s', Ret x `merge` r) | (s', r) <− l s ])
Grab k `merge` Ret y =
    Grab (\ s → [ (s', r `merge` Ret y) | (s', r) <− k s ])
Grab k `merge` Grab l =
    Grab (\ s → [ (s', r `merge` Grab l) | (s', r) <− k s ]
      ++ [ (s', Grab k `merge` r) | (s', r) <− l s ])
instance Lmf (Res s) where
  ψ⁰ = Ret ()
  r `ψ` r' = r `merge` r'
```

This completes the concurrent monad structure on T.

```
instance (Listable s, Eq s) ⇒ ConcurMnd (Res s)
```

All required equations and inequations hold, in particular associativity of ψ and the middle four interchange. That this works with List and does not require quotienting down to $\mathcal{M}_\mathrm{f}$ or $\mathcal{P}_\mathrm{f}$ is a small miracle.

The concurrent monad T is normal, we have $\eta_1 = \psi^0$.

The concurrent monad T supports interleaving shared state in the sense that it admits operations for reading and overwriting the state and for atomizing a computation, satisfying suitable equations.

```
class ConcurMnd t ⇒ SharedState s t | t → s where
  get  :: t s
  put  :: s → t ()
  stitch  :: t x → t x
atomic :: SharedState s t ⇒ (x → t y) → x → t y
atomic k = stitch . k
```

The operations for reading and writing are unsurprising and take one small step: they grab, yield and terminate. Atomizing turns any resumption into such a resumption. This is done by cancelling consecutive yields and grabs: any yielded state that is not final is passed directly into the next grab.

```
instance (Listable s, Eq s) ⇒ SharedState s (Res s) where
  get = Grab (\ s → [(s, Ret s)])
  put s = Grab (\ _ → [(s, Ret ())])
  stitch r = Grab (\ s → stitchAcc (s, r))
    where stitchAcc :: (s, Res s x) → [(s, Res s x)]
          stitchAcc (s, Ret x)  = [(s, Ret x)]
          stitchAcc (s, Grab k) = concat (map stitchAcc (k s))
```

The equations are not the standard ones for state manipulation. E.g., reading following by writing the result is not identity, but it is if the sequence is atomized.

We illustrate these operations in action. We define

```
cat :: SharedState String t ⇒ String → () → t ()
cat w = atomic (const get >>> \ s → put (s ++ w))
  −− = \ () → Grab (\ s → [(s ++ w, Ret ())]) −− equivalently
a = cat "a"
b = cat "b"
c = cat "c"
d = cat "d"
r1 = ((a ||| b) ||| c) (((), ()), ())
r2 = (a ||| (b ||| c)) ((), ((), ()))
r3 = ((a ||| b) >>> (c ||| d)) ((), ())
r4 = ((a >>> c) ||| (b >>> d)) ((), ())
```

Resumptions get very big to show (when we tabulate the functions involved in them), but we can run resumptions against an environment that just passes any state yielded to the next grab and collect what we can observe into a residual resumption that is easily showable. Residual resumptions are an ordered monad, but not a concurrent monad. Running here is in the mathematical sense of the stateful runners of Uustalu [32] and the monad-comonad interaction laws of Katsumata et al. [17].

```
data RRes x = RRet x | RGrab s [(s, Tree x)]  deriving Show
runIsolated :: RRet s x → s → Tree x
runIsolated (Ret x) s = RRet x
runIsolated (Grab k) s =
        RGrab s [ (s', runIsolated r s') | (s', r) <− k s]
```

Both `runIsolated r1 ""` and `runIsolated r2 ""` produce the same result:

```
RGrab "" [
  ("a", RGrab "a" [
    ("ab", RGrab "ab" [("abc",RRet (((),()),()))]),
    ("ac", RGrab "ac" [("acb",RRet (((),()),()))])]),
  ("b", RGrab "b" [
    ("ba", RGrab "ba" [("bac",RRet (((),()),()))]),
    ("bc", RGrab "bc" [("bca",RRet (((),()),()))])]),
  ("c", RGrab "c" [
    ("ca",RGrab "ca" [("cab",RRet (((),()),()))]),
    ("cb", RGrab "cb" [("cba",RRet (((),()),()))])])]
```

But `runIsolated r3 ""` gives

```
RGrab "" [
  ("a",RGrab "a" [
    (""ab",RGrab "ab" [
      ("abc",RGrab "abc" [("abcd",RRet ((),()))]),
      ("abd",RGrab "abd" [("abdc",RRet ((),()))])])]),
  ("b",RGrab "b" [
    ("ba",RGrab "ba" [
      ("bac",RGrab "bac" [("bacd",RRet ((),()))]),
      ("bad",RGrab "bad" [("badc",RRet ((),()))])])])]
```

while `runIsolated r4 ""` gives

```
RGrab "" [
  ("a", RGrab "a" [
    ("ac", RGrab "ac" [
      ("acb", RGrab "acb" [("acbd", RRet ((),()))])]),
    ("ab", RGrab "ab" [
      ("abc", RGrab "abc" [("abcd", RRet ((),()))]),
      ("abd", RGrab "abd" [("abdc", RRet ((),()))])])]),
  ("b", RGrab "b" [
    ("ba", RGrab "ba" [
      ("bac", RGrab "bac" [("bacd", RRet ((),()))]),
      ("bad", RGrab "bad" [("badc", RRet ((),()))])]),
    ("bd", RGrab "bd" [
      ("bda", RGrab "bda" [("bdac", RRet ((),()))])])])]
```

adding two outcomes. One residual resumption is smaller than the other wrt. the order on residual resumptions induced by the order on lists.

Bags of traces. Resumptions are a very compact and very constructive notion of computation manipulating shared state.

A different notion, where branching points are not observable, is given by finite multisets of traces, turned from a concurrent monoid to a concurrent monad by having every trace end with a value. The object mapping of the underlying ordered functor here is

$$TX = \mathcal{M}_\mathrm{f}(T'X) \text{ where } T'X = \mu Z.X + S \times S \times Z$$

The order on TX is induced by the order on X and by the order on $\mathcal{M}_\mathrm{f}Y$ that is induced by the multiset inclusion and the order on Y for any Y.

A trace is either of the form $\mathsf{inl}\,x$ signifying termination with return value x or of the form $\mathsf{inr}(s, s', t)$ where s is a state grabbed, s' is a state yielded and t is a trace again. A computation is a multiset of traces. A trace is completely deterministic, all nondeterministic branching happens at the beginning of the computation—it is the act of choosing of one trace from the finite multiset.

Here is the Haskell code for T as an ordered functor.

```
data Trace s x = R x | G s s (Trace s x)
newtype BoT s x = BoT (Bag (Trace s x)) deriving Eq
instance Functor (Trace s) where
  fmap f (R x) = R (f x)
  fmap f (G s s' t) = G s s' (fmap f t)
instance Eq s ⇒ OrderedFunctor (Trace s) where
  ford (=<) (R x) (R y) = x =< y
  ford (=<) (R _) (G _ _ _) = False
  ford (=<) (G _ _ _) (R _) = False
  ford (=<) (G s0 s t) (G s0' s' t') =
                                s0 == s0' && s == s' && ford (=<) t t'
instance Functor Bag where
  fmap f (B xs) = B (map f xs)
instance OrderedFunctor Bag where
  ford leq (B xs) (B ys) =
      or [ and (zipWith leq xs zs) | zs <- combs (length xs) ys ]
instance Functor (BoT s) where
  fmap f (BoT ts) = BoT (fmap (fmap f) ts)
instance Eq s ⇒ OrderedFunctor (BoT s) where
  ford (=<) (BoT ts) (BoT ts') = ford (ford (=<)) ts ts'
```

The ordered monad structure on T is a combination (from a distributive law) of the ordered monad structures on $\mathcal{M}_\mathrm{f}$ and T' where the monad structure on T' is free. In Haskell,

```
instance Mnd (BoT s) where
  η x = BoT (B [R x])
  μ (BoT (B ts)) = BoT (B (concat (fmap μ' ts))) where
    μ' (R (BoT (B ts))) = ts
    μ' (G s s' t) = fmap (G s s') (μ' t)
```

The ordered lax monoidal functor structure on T is defined by interleaving traces, in Haskell as follows, and the definition of the concurrent monad structure on T is thereby complete.

```
instance Lmf (BoT s) where
  ψ⁰ = BoT (B [R ()])
  BoT (B ts0) `ψ` BoT (B ts1) = BoT (B (concat
         [ t0 `shuffle` t1 | t0 <- ts0, t1 <- ts1 ])) where
    R x `shuffle` R y = [R (x, y)]
    R x `shuffle` G s s' t = fmap (G s s') (R x `shuffle` t)
    G s s' t `shuffle` R y = fmap (G s s') (t `shuffle` R y)
    G s0 s0' t0 `shuffle` G s1 s1' t1
        = fmap (G s0 s0') (t0 `shuffle` G s1 s1' t1)
        ++ fmap (G s1 s1') (G s0 s0' t0 `shuffle` t1)
instance (Listable s, Eq s) ⇒ ConcurMnd (BoT s)
```

Here associativity of ψ and the middle four interchange hold only because we have modelled nondeterminism with $\mathcal{M}_\mathrm{f}$, they would fail for List. (But they are of course satisfied for $\mathcal{P}_\mathrm{f}$.)

The shared state operations are unproblematic to define similarly to the case of resumptions so that they satisfy the required equations. But they are extremely inefficient since the model of collections of traces as computations forces that functions from S functions get represented by their graphs.

```
instance (Listable s, Eq s) ⇒ SharedState s (BoT s) where
  get = BoT (B [ G s s (R s) | s <- list ])
  put s' = BoT (B [ G s s' (R ()) | s <- list ])
  stitch (BoT (B ts)) =
    BoT (B [ G s s' t' | s <- list ,
                         t <- ts ,
                         (s', t') <- stitchAcc s t ]) where
      stitchAcc :: s → Trace s x → [(s, Trace s x)]
      stitchAcc s0 (R x) = [(s0, R x)]
      stitchAcc s0 (G s s' t) =
                         if s0 == s then stitchAcc s' t else []
```

A concurrent monad morphism. There is a morphism τ between the resumption-based and the collection-of-traces based concurrent monads. A morphism of concurrent monads preserves sequential composition properly but parallel composition only oplaxly in that $\psi_{X,Y};\tau_{X\otimes Y} \leq \tau_X \otimes \tau_Y;\psi_{X,Y}$. We only show the Haskell code.

```
class Nat f g where
  tau :: f x → g x
instance Listable s ⇒ Nat (Res s) (BoT s) where
  tau (Ret x) = BoT (B [R x])
  tau (Grab k) = BoT (B [G s s' t | s <- list , (s', r) <- k s,
                                    let BoT (B ts) = tau r, t <- ts ]
class (Mnd t, Mnd r, Nat t r) ⇒ MndMap t r
class (Lmf t, Lmf r, Nat t r) ⇒ LmfMap t r
```

```
class (ConcurMnd t, ConcurMnd r, MndMap t r, LmfMap t r)
                                          ⇒ ConcurMndMap t r
instance Listable s ⇒ MndMap (Res s) (BoT s)
instance Listable s ⇒ LmfMap (Res s) (BoT s)
instance (Listable s, Eq s) ⇒ ConcurMndMap (Res s) (BoT s)
```

5 RELATED WORK

Concurrent monads are due to Rivas and Jaskelioff [29]. In their version concurrent monads were ordered monads (in a different sense than **Poset**-enriched) on **Set**; the Kleisli construction was an ordered category. Paquet and Saville [23] generalized normal concurrent monads in the sense of this paper to pseudomonads on bicategories.

The Freyd categories (a.k.a. effectful categories) of Levy, Power and Thielecke [20] are an axiomatization of aspects of the Kleisli categories of strong monads on a given base monoidal category. They relate to strong monads like concurrent categories relate to concurrent monads. The premonoidal categories of Power and Robinson [28] are related concepts that avoid parametrization in a base and so are the abstract Kleisli categories of Führmann [5]. Heunen and Sigal's [15] duoidally enriched Freyd categories add interesting expressive power to classical Freyd categories. They bear some similarities to concurrent categories, but do not cover them (since they cannot simulate order and inequations). Here 'duoidal enrichment' refers to a generalization over the standard notion of enrichment in a (symmetric closed) monoidal category.

Duoidal categories originate from Balteanu et al. [2]; they have been used or studied specifically by Garner [8], Aguiar and Mahajan [1], Garner and López Franco [9]. They have been called twofold monoidal, bimonoidal. The name 'duoidal' is from Batanin and Markl [3].

Ordered monads in the sense of ordinary monads on **Set** whose underlying functor factors through U : **Poset** $\rightarrow$ **Set** (or even U : **SupLat** $\rightarrow$ **Set**) have been considered by many programming semanticists, e.g., Goncharov and Schröder [12], Katsumata and Sato [18] and Hasuo [14]. They are prominent in monoidal topology, a generalization of topology initiated by Gähler [6] and Seal [30].

Resumptions were invented by Milner [21] and used in domain theory by Plotkin [26]. They were introduced to the monad-based approach to effectful computation by Cenciarelli and Moggi [4]. Goncharov and Schröder [11] studied in great detail a coinductive resumption monad supporting interleaving parallel composition (enabled by the layered structure in that monad) and iteration of general effectful functions. They did not discuss the axioms of parallel composition. Uustalu's [31] semantics of preemptive and cooperative shared state concurrency had in the background a coinductive resumption monad similar to the (inductive) one considered here.

6 CONCLUSIONS

In this paper, we have advocated the concurrent monads of Rivas and Jaskelioff [29] as a principled approach to the semantics of effectful concurrency. We proceeded from concurrent monoids as known from concurrent Kleene algebra and generalized those to concurrent categories by "typing" them. Then we defined concurrent monads so that the Kleisli category of a concurrent monad is a concurrent category. We showed that concurrent monads are concurrent monoid objects in an ordered duoidal category.

To work in ordered category theory, enabling the use lax/oplax functors, was crucial for this development. This made it possible to avoid degeneration of concurrent monads into commutative monads (= lax monoidal monads).

We do not currently know what a free concurrent monad on an ordered endofunctor would look like or how concurrent monads relate to ordered algebraic theories.

We demonstrated that concurrent monads are flexible enough to model resumptions- and collections-of-traces-based semantics of shared state remarkably smoothly, so smoothly, that one is, for example, not forced to switch from the effectively implementable lists-based semantics of nondeterminism to the multisets- or powerset-based semantics to validate the desirable (in)equational axioms in the case of resumptions.

We believe that the models of shared state demonstrated here can be adapted to handle variations such as transactional memory and relaxed memory. We intend to show that this is the case. Neat modelling of message passing and asynchronous communication is a more serious challenge that we would also like to take on. Cooperative concurrency does not fit into concurrent monads since the interchange inequations require that sequential composition can be preempted at the mid-point. It is not clear how to axiomatize cooperative concurrency usefully in a principled way; we would like to find this out.

ACKNOWLEDGMENTS

E.R. was supported by the Estonian Research Council grant no. PSG749. T.U. was supported by the Icelandic Research Fund grant no. 228684-053.

REFERENCES

[1] Marcelo Aguiar and Swapneel Mahajan. 2010. *Monoidal Functors, Species and Hopf Algebras.* Amer. Math. Soc. https://doi.org/10.1090/crmm/029

[2] C. Balteanu, Z. Fiedorowicz, R. Schwänzl, and R. Vogt. 2003. Iterated Monoidal Categories. *Adv. Math.* 176, 2 (2003), 277–349. https://doi.org/10.1016/s0001-8708(03)00065-3

[3] Michael Batanin and Martin Markl. 2012. Centers and Homotopy Centers in Enriched Monoidal Categories. *Adv. Math.* 230, 4–6 (2012), 1811–1858. https://doi.org/10.1016/j.aim.2012.04.011

[4] Pietro Cenciarelli and Eugenio Moggi. Amsterdam. A Syntactic Approach to Modularity in Denotational Semantics. In *Proc. of 5th Biennial Meeting on Category Theory and Computer Science, CTCS '93 (Amsterdam, Sept. 1993).* CWI.

[5] Carsten Führmann. 1999. Direct Models of the Computational Lambda-Calculus. *Electron. Notes in Theor. Comput. Sci.* 20 (1999), 245–292. https://doi.org/10.1016/s1571-0661(04)80078-1

[6] Werner Gähler. 1992. Monadic Topology: A New Concept of Generalized Topology. In *Recent Developments of General Topology and Its Applications: Int. Conf. in Memory of Felix Hausdorff (1868–1942)*, Werner Gähler, Horst Herrlich, and Gerhard Preuß (Eds.). Math. Research, Vol. 67. Akademie Verlag, 136–149.

[7] Nicola Gambino, Richard Garner, and Christina Vasilakopoulou. 2024. Monoidal Kleisli Bicategories and the Arithmetic Product of Coloured Symmetric Sequences. *Docum. Math* 29, 3 (2024), 627–702. https://doi.org/10.4171/dm/950

[8] Richard Garner. 2009. Understanding the Small Object Argument. *Appl. Categ. Struct.* 17, 3 (2009), 247–285. https://doi.org/10.1007/s10485-008-9137-4

[9] Richard Garner and Ignacio López Franco. 2016. Commutativity. *J. Pure Appl. Algebra* 220, 5 (2016), 1707–1751. https://doi.org/10.1016/j.jpaa.2015.09.003

[10] Jay L. Gischer. 1988. The Equational Theory of Pomsets. *Theor. Comput. Sci.* 61, 2–3 (1988), 199–224. https://doi.org/10.1016/0304-3975(88)90124-7

[11] Sergey Goncharov and Lutz Schröder. 2013. A Coinductive Calculus for Asynchronous Side-Effecting Processes. *Inf. Comput.* 231 (2013), 204–232. https://doi.org/10.1016/j.ic.2013.08.012

[12] Sergey Goncharov and Lutz Schröder. 2013. A Relatively Complete Generic Hoare Logic for Order-Enriched effects. In *Proc. of 28th Ann. ACM/IEEE Symp. on*

Logic in Computer Science, LICS 2013 (New Orleans, LA, June 2013). IEEE, 273–282. https://doi.org/10.1109/lics.2013.33

[13] Sergey Goncharov, Lutz Schröder, and Till Mossakowski. 2009. Kleene Monads: Handling Iteration in a Framework of Generic Effects. In *Algebra and Coalgebra in Computer Science: 3rd Int. Conf., CALCO 2009 (Udine, Sept. 2009), Proc.*, Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki (Eds.). Lect. Notes in Comput. Sci., Vol. 5728. Springer, 18–33. https://doi.org/10.1007/978-3-642-03741-2_3

[14] Ichiro Hasuo. 2015. Generic Weakest Precondition Semantics from Monads Enriched with Order. *Theor. Comput. Sci.* 604 (2015), 2–29. https://doi.org/10.1016/j.tcs.2015.03.047

[15] Chris Heunen and Jesse Sigal. 2023. Duoidally Enriched Freyd Categories. In *Relational and Algebraic Methods in Computer Science: 20th Int. Conf., RAMiCS 2023 (Augsburg, Apr. 2023), Proc.*, Roland Glück, Luigi Santocanale, and Michael Winter (Eds.). Lect. Notes in Comput. Sci., Vol. 13896. Springer, 241–257. https://doi.org/10.1007/978-3-031-28083-2_15

[16] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2011. Concurrent Kleene Algebra and Its Foundations. *J. Log. Algebraic Program.* 80, 6 (2011). https://doi.org/10.1016/j.jlap.2011.04.005

[17] Shin-ya Katsumata, Exequiel Rivas, and Tarmo Uustalu. 2020. Interaction Laws of Monads and Comonads. In *Proc. of 35th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS '20 (Saarbrücken, July 2020)*. ACM, 604–618. https://doi.org/10.1145/3373718.3394808

[18] Shin-ya Katsumata and Tetsuya Sato. 2013. Preorders on Monads and Coalgebraic Simulations. In *Foundations of Software Science and Computation Structures: 16th Int. Conf., FoSSaCS 2013 (Rome, March 2013), Proc.*, Frank Pfenning (Ed.). Lect. Notes in Comput. Sci., Vol. 7794. Springer, 145–160. https://doi.org/10.1007/978-3-642-37075-5_10

[19] Anders Kock. 1970. Strong Functors and Monoidal Monads. *Arch. Math.* 23 (1970), 113–120. https://doi.org/10.1007/bf01304852

[20] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling Environments in Call-by-Value Programming Languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/s0890-5401(03)00088-9

[21] Robin Milner. 1975. Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73 (Studies in Logic and the Foundations of Mathematics, Vol. 80)*, H. E. Rose and J. C. Shepherdson (Eds.). North Holland, 157–173. https://doi.org/10.1016/s0049-237X(08)71948-7

[22] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science, LICS '89 (Pacific Grove, CA, June 1989)*. IEEE, 14–23. https://doi.org/10.1109/lics.1989.39155

[23] Hugo Paquet and Philip Saville. 2023. Effectful Semantics in 2-Dimensional Categories: Premonoidal and Freyd Bicategories. In *Proc. of 6th Int. Conf. on Applied Category Theory, ACT 2023 (College Park, MD, July/Aug. 2023)*, Sam Staton and Christina Vasilakopoulou (Eds.). Electron. Proc. in Theor. Comput. Sci., Vol. 397. Open Publishing Assoc., 190–209. https://doi.org/10.4204/eptcs.397.12

[24] Hugo Paquet and Philip Saville. 2024. Effectful Semantics in Bicategories: Strong, Commutative, and Concurrent Pseudomonads. In *Proc. of 39th ACM/IEEE Symp. on Logic in Computer Science, LICS '24 (Tallinn, July 2024)*. ACM.

[25] Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures: 5th Int. Conf., FOSSACS 2002 (Grenoble, Apr. 2002), Proc.*, Mogens Nielsen and Uffe Engberg (Eds.). Lect. Notes in Comput. Sci., Vol. 2303. Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

[26] Gordon D. Plotkin. 1976. A Powerdomain Construction. *SIAM J. Comput.* 5, 3 (1976), 452–487. https://doi.org/10.1137/0205035

[27] John Power. 2002. Premonoidal Categories as Categories with Algebraic Structure. *Theor. Comput. Sci.* 278, 1–2 (2002), 303–321. https://doi.org/10.1016/s0304-3975(00)00340-6

[28] John Power and Edmund Robinson. 1997. Premonoidal Categories and Notions of Computation. *Math. Struct. Comput. Sci.* 5, 4 (1997), 453–468. https://doi.org/10.1017/s0960129597002375

[29] Exequiel Rivas and Mauro Jaskelioff. 2019. Monads with Merging. HAL eprint hal-02150199. https://inria.hal.science/hal-02150199

[30] Gavin J. Seal. 2009. A Kleisli-Based Approach to Lax Algebras. *Appl. Categ. Struct.* 17, 1 (2009), 75–89. https://doi.org/10.1007/s10485-007-9080-9

[31] Tarmo Uustalu. 2013. Coinductive Big-Step Semantics for Concurrency. In *Proc. of 6th Wksh. on Programming Language Approaches to Concurrency and Communication-Centric Software, PLACES 2013 (Rome, March 2013)*, Nobuko Yoshida and Wim Vanderbauwhede (Eds.). Electron. Proc. in Theor. Comput. Sci., Vol. 137. Open Publishing Assoc., 63–78. https://doi.org/10.4204/eptcs.137.6

[32] Tarmo Uustalu. 2015. Stateful Runners for Effectful Computations. *Electron. Notes Theor. Comput. Sci.* 319 (2015), 403–421. https://doi.org/10.1016/j.entcs.2015.12.024