Layer by Layer – Combining Monads

Fredrik Dahlqvist, Louis Parlant, and Alexandra Silva*

University College London

Abstract. We develop a modular method to build algebraic structures. Our approach is categorical: we describe the layers of our construct as monads, and combine them using distributive laws.
Finding such laws is known to be difficult and our method identifies precise sufficient conditions for two monads to distribute. We either (i) concretely build a distributive law which then provides a monad structure to the composition of layers, or (ii) pinpoint the algebraic obstacles to the existence of a distributive law and suggest a weakening of one layer that ensures distributivity.
This method can be applied to a step-by-step construction of a programming language. Our running example will involve three layers: a basic imperative language enriched first by adding non-determinism and then probabilistic choice. The first extension works seamlessly, but the second encounters an obstacle, resulting in an 'approximate' language very similar to the probabilistic network specification language ProbNetKAT.

1 Introduction

The practical objective of this paper is to provide a systematic and modular understanding of the design of recent programming languages such as NetKAT [9] and ProbNetKAT [8,28] by re-interpreting their syntax as a layering of monads. However, in order to solve this problem, we develop a very general technique for building *distributive laws between monads* whose applicability goes far beyond understanding the design of languages in the NetKAT family. Indeed, the combination of monads has been an important area of research in theoretical computer science ever since Moggi developed a systematic understanding of computational effects as monads in [25]. In this paradigm – further developed by Plotkin, Power and others in e.g. [26,4] – the question of how to combine computational effects can be treated systematically by studying the possible ways of combining monads. This work can also be understood as a contribution to this area of research.

Combining effects is in general a non-trivial issue, but diverse methods have been studied in the literature. A *monad transformer*, as described in [4], is a way to enrich any theory with a specific effect. These transformers allow a step-by-step construction of computational structures, later exploited by Hudak et al. [21,20]. In [12], Hyland, Plotkin and Power systematized the study of effect combinations

* This work was partially supported by ERC grant ProfoundNet.

by introducing two canonical constructions for combining monads, which in some sense lie at the extreme ends of the collection of possible combination procedures. At one end of the spectrum they define the *sum* of monads which consists in the juxtaposition of both theories with no interaction whatsoever between computational effects. At the other end of the spectrum they define the *tensor* of two monads where both theories are maximally interacting in the sense that "each operator of one theory commutes with each operation of the other" ([12]). In [11] they combine exceptions, side-effects, interactive input/output, non-determinism and continuations using these operations.

In some situations neither the sum nor the tensor of monads is the appropriate construction, and some intermediate level of interaction is required. From the perspective of understanding the design of recent programming languages which use layers of non-determinism and probabilities (e.g. ProbNetKAT), there are two reasons to consider combinations other than the sum or the tensor. First, there is the unavoidable mathematical obstacle which arises when combining sequential composition with non-deterministic choice (see the simple example below), two essential features of languages in the NetKAT family. When combining two monoid operations with the tensor construction, one enforces the equation $(\mathtt{p};\mathtt{q}) + (\mathtt{r};\mathtt{s}) = (\mathtt{p} + \mathtt{r});(\mathtt{q} + \mathtt{s})$ which means, by the Eckmann-Hilton argument, that the two operations collapse into a single commutative operation; clearly not the intended construction. Secondly, and much more importantly, the intended *semantics* of a language may force us to consider specific and limited interactions between its operations. This is the case for languages in the NetKAT family, where the intended trace semantics suggests *distributive laws* between operations, for instance that sequential composition distributes over non-deterministic choice (but not the converse). For this reason, the focus of this paper will be to explicitly construct *distributive laws* between monads.

It is worth noting that existence of distributive laws is a subtle question and having automatic tools to derive these is crucial in avoiding mistakes. As a simple example in which several mistakes have appeared in the literature, consider the composition of the powerset monad $\mathcal{P}$ with itself. Distributive laws of $\mathcal{P}$ over $\mathcal{P}$ were proposed in 1993 by King [15] and in 2007 [23], with a subsequent correction of the latter result by Manes and Mulry themselves in a follow-up paper. In 2015, Klin and Rot made a similar claim [16], but recently Klin and Salamanca have in fact showed that there is no distributive law of $\mathcal{P}$ over itself and explain carefully why all the mistakes in the previous results were so subtle and hard to spot [17]. This example shows that this question is very technical and sometimes counter-intuitive. Our general and modular approach provides a fine-grained method for determining **(a)** if a monad combination by distributive law is possible, **(b)** if it is not possible, exactly which features are broken by the extension and **(c)** suggests a way to fix the composition by modifying one of our monads. In other words, this enables informed design choices on which features we may accept to lose in order to achieve greater expressive power in a language through monad composition.

The original motivation for this work is very concrete and came from trying to understand the design of ProbNetKAT, a recently introduced programming language with non-determinism and probabilities [8,28]. The non-existence of a distributive law between the powerset monad and the distribution monad, first proved by Varacca [30] and discussed recently in [5], is a well known problem in semantics. As we will show, our method enables us to modularly build ProbNetKAT based on the composition of several monads capturing the desired algebraic features. The method derives automatically which equations have to be dropped when adding the probabilistic layer providing a principled justification to the work initially presented in [8,28].

A simple example. Let us consider a set $\mathtt{P}$ of atomic programs, and build a 'minimal' programming language as follows. Since sequential composition is essential to any imperative language we start by defining the syntax as:

$$\mathtt{p} ::= \mathtt{skip} \mid \mathtt{p}\,;\mathtt{p} \mid \mathtt{a} \in \mathtt{P} \tag{1}$$

and ask that the following programs be identified:

$$\mathtt{p}\,;\mathtt{skip} = \mathtt{p} = \mathtt{skip}\,;\mathtt{p} \qquad \text{and} \qquad \mathtt{p}\,;(\mathtt{q}\,;\mathtt{r}) = (\mathtt{p}\,;\mathtt{q})\,;\mathtt{r} \tag{2}$$

The language defined by the operations of (15) and the equations of (16) can equally be described as the application of the *free monoid monad* $(-)^*$ to the set of atomic programs $\mathtt{P}$. If we assign a semantics to each basic program $\mathtt{P}$, the semantics of the extended language can be defined as finite sequences (or traces) of the basic semantics. In a next step, we might want to enrich this basic language by adding a non-deterministic choice operation $+$ and the constant program $\mathtt{abort}$, satisfying the equations:

$$\mathtt{abort}{+}\mathtt{p} = \mathtt{p} = \mathtt{p}{+}\mathtt{abort} \qquad \mathtt{p}{+}\mathtt{p} = \mathtt{p} \qquad \mathtt{p}{+}\mathtt{q} = \mathtt{q}{+}\mathtt{p} \qquad \mathtt{p}{+}(\mathtt{q}{+}\mathtt{r}) = (\mathtt{p}{+}\mathtt{q}){+}\mathtt{r} \tag{3}$$

The signature $(\mathtt{abort}, +)$ and the axioms (17) define join-semilattices, and the monad building free semilattices is the *finitary powerset monad* $\mathcal{P}$. To build our language in a modular fashion we thus want to apply $\mathcal{P}$ on top of our previous construction and consider the programming language where the syntax and semantics arise from $\mathcal{P}(\mathtt{P}^*)$. For this purpose we combine both monads to construct a new monad $\mathcal{P}(-^*)$ by building a distributive law $(-)^*\mathcal{P} \to \mathcal{P}(-)^*$. As explained above, this approach is semantically justified by the intended trace semantics of the language, and will ensure that operations from the inner layer distribute over the outer ones, i.e.

$$\mathtt{p};(\mathtt{q}+\mathtt{r}) = \mathtt{p};\mathtt{q}+\mathtt{p};\mathtt{r} \qquad (\mathtt{q}+\mathtt{r});\mathtt{p} = \mathtt{q};\mathtt{p}+\mathtt{r};\mathtt{p} \qquad \mathtt{p};\mathtt{abort} = \mathtt{abort};\mathtt{p} = \mathtt{abort} \tag{4}$$

Our method proves and relies on the following theorem: if $\mathcal{P}$ preserves the structure of $(-)^*$-algebra defined by (15)-(16), then the composition $\mathcal{P}(-^*)$ has a monad structure provided by the corresponding distributive law. Applying this theorem to our running example, the first step is to lift the signature (15), in other words to define new canonical interpretations in $\mathcal{P}(\mathtt{P}^*)$ for ; and $\mathtt{skip}$. Once

this lifting is achieved, the equations in (16), arising from the inner layer, can be interpreted in $\mathcal{P}(-^*)$. We need to check if they still hold: is the new interpretation of ; still associative? To answer this question, our method makes use of categorical diagrams to obtain precise conditions on our monadic constructs. Furthermore, in the case where equations fail to hold, we provide a way to identify exactly what stands in the way of monad composition. We can then offer tailor-made adjustments to achieve the composition and obtain a 'best approximate' language, with slightly modified monads.

Structure of this paper. Section 2 presents some basic facts about monads and distributive laws and fixes the notation. In Section 3 we recall the well-known fact [29,24] that there exists a distributive law of any polynomial functor over a monoidal **Set**-monad. In particular this shows that *operations* can be lifted by monoidal monads. In fact, the techniques presented in this paper can be extended beyond the monoidal case, but since we won't need such monads in our applications, we will focus on monoidal monads for which the lifting of operations is very straightforward. We then show in Section 4 when *equations* can also be lifted. We isolate two conditions on the lifting monad which guarantee that any equation can be lifted. These two conditions correspond to a monad being *affine* [18] and *relevant* [13]. We also characterise the general form of equations preserved by monads which only satisfy a subset of these conditions. Interestingly, together with the symmetry condition (SYM) which is always satisfied by monoidal **Set**-monads, we recover what are essentially the three structural laws of classical logic (see also [13]). In Section 5 we show how the $*$-free fragment of ProbNetKAT can be built in systematic way by construction distributive laws between the three layers of the language.

2 A primer on monads, algebras and distributive laws

Monads and (Σ, E)-algebras. For the purposes of this paper, we will always consider monads on **Set** ([1,22,25]). The core language described in the introduction is defined by the signature $\Sigma = \{\,;\,, \texttt{skip}\}$ and the set E of equations given by (16). More generally, we view programming languages as algebraic structures defined by a signature $(\Sigma, \mathrm{ar} : \Sigma \to \mathbb{N})$ and a set of equations E enforcing program equivalence. To formalize this we first define a Σ-algebra to be a set X together with an interpretation $[\![\sigma]\!] \colon X^{\mathrm{ar}(\sigma)} \to X$ of each operation $\sigma \in \Sigma$. A Σ-algebra can be conveniently represented as an algebra for the polynomial functor $\mathsf{H}_\Sigma = \coprod_{\sigma\in\Sigma}(-)^{\mathrm{ar}(\sigma)}$ defined by the signature, i.e. as a set X together with a map $\beta \colon \mathsf{H}_\Sigma X \to X$. A Σ-algebra morphism between $\beta \colon \mathsf{H}_\Sigma X \to X$ and $\gamma \colon \mathsf{H}_\Sigma Y \to Y$ is a map $f \colon X \to Y$ such that $\gamma \circ \mathsf{H}_\Sigma f = f \circ \beta$. The category of Σ-algebras and Σ-algebra morphisms is denoted $\mathbf{Alg}(\Sigma)$. In particular, the set $\mathsf{F}_\Sigma X$ of all Σ-terms is a Σ-algebra – the free Σ-algebra over X – and F_Σ is a functor $\mathbf{Set} \to \mathbf{Alg}(\Sigma)$ forming an adjunction

$$\mathsf{F}_\Sigma \dashv \mathsf{U}_\Sigma : \mathbf{Alg}(\Sigma) \to \mathbf{Set} \tag{5}$$

Since it will not lead to any ambiguity we will usually overload the symbol F_Σ to also denote the monad $\mathsf{U}_\Sigma\mathsf{F}_\Sigma : \mathbf{Set} \to \mathbf{Set}$ arising from this adjunction.

Given a Σ-algebra $\mathcal{A}$, a free Σ-term s built over variables in a set V, and a valuation map $v : V \to \mathsf{U}_\Sigma\mathcal{A}$, we define the interpretation $[\![s]\!]_v$ of s in $\mathcal{A}$ recursively in the obvious manner. We say that an equation $s = t$ between free Σ-terms is valid in $\mathcal{A}$, denoted $\mathcal{A} \models s = t$, if for every valuation $v : V \to \mathsf{U}_\Sigma\mathcal{A}$, $[\![s]\!]_v = [\![t]\!]_v$. Given a set E of equations we define a (Σ, E)-algebra as a Σ-algebra in which all the equations in E are valid. We denote by $\mathbf{Alg}(\Sigma, E)$ the subcategory of $\mathbf{Alg}(\Sigma)$ consisting of (Σ, E)-algebras. There exists a functor $\mathsf{F} : \mathbf{Set} \to \mathbf{Alg}(\Sigma, E)$ building free (Σ, E)-algebras which is left adjoint to the obvious forgetful functor:

$$\mathsf{F} \dashv \mathsf{U} : \mathbf{Alg}(\Sigma, E) \to \mathbf{Set} \tag{6}$$

In our running example all monads arise from a finitary syntax, and thus from an adjunction of the type (6).

Eilenberg-Moore categories. An algebra for the monad T is a set X together with an map $\alpha\colon TX \to X$ such that the diagrams in (7) commute. A morphism $(X, \alpha) \xrightarrow{f} (Y, \beta)$ of T-algebras is a morphism $X \xrightarrow{f} Y$ in $\mathbf{Set}$ verifying $\beta \circ Tf = f \circ \alpha$.

$$\begin{array}{ccc} TTX & \xrightarrow{\mu_X} & TX \\ {\scriptstyle T\alpha}\downarrow & & \downarrow{\scriptstyle \alpha} \\ TX & \xrightarrow{\alpha} & X \end{array} \qquad\qquad \begin{array}{ccc} X & \xrightarrow{\eta_X} & TX \\ & {\scriptstyle 1}\searrow & \downarrow{\scriptstyle \alpha} \\ & & X \end{array} \tag{7}$$

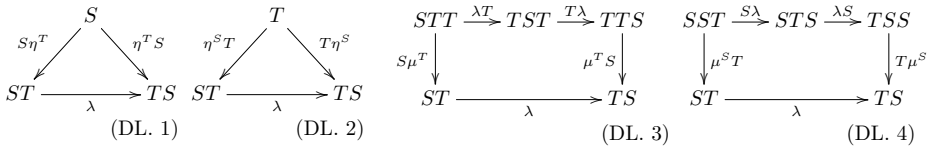
The category of T-algebras and T-algebra morphisms is called the *Eilenberg-Moore* category of the monad T, and denoted $\mathcal{EM}(T)$. There is an obvious forgetful functor $\mathsf{U}_E : \mathcal{EM}(T) \to \mathbf{Set}$ which sends an algebra to its carrier, it has a left adjoint $\mathsf{F}_E : \mathbf{Set} \to \mathcal{EM}(T)$ which sends a set X to the free T-algebra $\mu_X : T^2X \to TX$. Note that the adjunction $\mathsf{F}_E \dashv \mathsf{U}_E$ gives rise to the monad T. A *lifting* of a functor $F : \mathbf{Set} \to \mathbf{Set}$ to $\mathcal{EM}(T)$ is a functor $\hat{F}$ on $\mathcal{EM}(T)$ such that $\mathsf{U}_E \circ \widehat{F} = F \circ \mathsf{U}_E$

Lemma 1 ([22] VI.8. Theorem 1). *For any adjunction of the form* (6)*,* $\mathcal{EM}(\mathsf{UF})$ *and* $\mathbf{Alg}(\Sigma, E)$ *are equivalent categories.*

The functors connecting $\mathcal{EM}(\mathsf{UF})$ and $\mathbf{Alg}(\Sigma, E)$ are traditionally called *comparison functors*, and we will denote them by $M\colon \mathcal{EM}(\mathsf{UF}) \to \mathbf{Alg}(\Sigma, E)$ and $K\colon \mathbf{Alg}(\Sigma, E) \to \mathcal{EM}(\mathsf{UF})$. Consider first the free monad F_Σ for a signature Σ (i.e. the monad generated by the adjunction (5)). The comparison functor $M\colon \mathbf{Alg}(\Sigma) \to \mathcal{EM}(\mathsf{F}_\Sigma)$ maps the free F_Σ-algebra over X, that is $\mu_X^{\mathsf{F}_\Sigma} : \mathsf{F}_\Sigma^2 X \to \mathsf{F}_\Sigma X$ to the free H_Σ-algebra over X which we shall denote by $\alpha_X : \mathsf{H}_\Sigma\mathsf{F}_\Sigma X \to \mathsf{F}_\Sigma X$. It is well-known that α_X is an isomorphism. Moreover, the maps α_X define a natural transformation $\mathsf{H}_\Sigma\mathsf{F}_\Sigma \to \mathsf{F}_\Sigma$. Similarly, in the presence of equations, if we consider the adjunction $\mathsf{F} \dashv \mathsf{U}$ of (6) and the associated monad $T = \mathsf{UF}$, then the comparison functor $M'\colon \mathbf{Alg}(\Sigma, E) \to \mathcal{EM}(T)$ sends the free T-algebra $\mu_X^T : T^2X \to TX$ to an H_Σ-algebra which we shall denote $\rho_X : \mathsf{H}_\Sigma TX \to TX$. Again, the maps ρ_X define a natural transformation

$\mathsf{H}_\Sigma T \to T$, but in general ρ_X is no longer an isomorphism: in the case of monoids and of a set $X = \{x, y, z\}$, we have $\rho_X(x;(y;z)) = \rho_X((x;y);z)$.

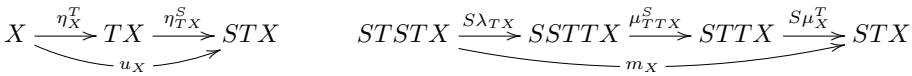
Distributive laws. Let (S, η^S, μ^S) and (T, η^T, μ^T) be monads, a *distributive law of S over T* (see [3]) is a natural transformation $\lambda : ST \to TS$ satisfying:



If λ only satisfies (DL. 2) and (DL. 4), we will say that λ is a distributive law of the the monad S over *functor* T, or in the terminology of [14], an $\mathcal{EM}$-law of S over T. Dually, if λ only satisfies (DL. 1) and (DL. 3), λ is known as a distributive law of the *functor* S over the monad T, or $\mathcal{K}l$-law of S over T [14].

Theorem 1. *[3,14,2] $\mathcal{EM}$-laws $\lambda : SF \to FS$ and liftings of F to $\mathcal{EM}(S)$ are in one-to-one correspondence.*

If there exists a distributive law $\lambda : TS \to ST$ of the monad T over the monad S, then the composition of S and T also forms a monad (ST, u, m), whose unit u and multiplication m are given by:



If $\mathcal{EM}(S) \simeq \mathbf{Alg}(\Sigma, E)$ and $\mathcal{EM}(T) \simeq \mathbf{Alg}(\Sigma', E')$, then a distributive law $ST \to TS$ implements the distributivity of the operations in Σ over those of Σ'.

3 Building distributive laws between monads

In this section we will show how to construct a distributive law $\lambda\colon ST \to TS$ between monads via a *monoidal structure* on T.

3.1 Monoidal monads

Let us briefly recall some relatively well-known categorical notion. A *lax monoidal functor* on a monoidal category $(\mathbf{C}, \otimes, I)$, or simply a monoidal functor[1], is an endofunctor $F : \mathbf{C} \to \mathbf{C}$ together with natural transformations $\psi_{X,Y} : FX \otimes FY \to F(X \otimes Y)$ and $\psi^0 : I \to FI$ satisfying the diagrams:

[1] We will never consider the notion of *strong* monoidal functor, so this terminology should not lead to any confusion.

$$\begin{array}{ccc} FX \otimes I & \xrightarrow{\mathbf{id}_{FX} \otimes \psi^0} & FX \otimes FI \\ \downarrow{\scriptstyle \rho_{FX}} & & \downarrow{\scriptstyle \psi_{X,I}} \\ FX & \xleftarrow{F\rho_X} & F(X \otimes I) \end{array} \quad \text{(MF. 1)}$$

$$\begin{array}{ccc} (FX \otimes FY) \otimes FZ & \xrightarrow{\alpha_{FX,FY,FZ}} & FX \otimes (FY \otimes FZ) \\ \downarrow{\scriptstyle \psi_{X,Y} \otimes \mathrm{id}_{FZ}} & & \downarrow{\scriptstyle \mathrm{id}_{FX} \otimes \psi_{Y,Z}} \\ F(X \otimes Y) \otimes FZ & & FX \otimes F(Y \otimes Z) \\ \downarrow{\scriptstyle \psi_{X \otimes Y, Z}} & & \downarrow{\scriptstyle \psi_{X, Y \otimes Z}} \\ F((X \otimes Y) \otimes Z) & \xrightarrow{F\alpha_{X,Y,Z}} & F(X \otimes (Y \otimes Z))) \end{array} \quad \text{(MF. 3)}$$

$$\begin{array}{ccc} I \otimes FX & \xrightarrow{\psi^0 \otimes \mathbf{id}_{FX}} & FI \otimes FX \\ \downarrow{\scriptstyle \rho'_{FX}} & & \downarrow{\scriptstyle \psi_{I,X}} \\ FX & \xleftarrow{F\rho'_X} & F(I \otimes X) \end{array} \quad \text{(MF. 2)}$$

where α is the associator of $(\mathbf{C}, \otimes, I)$ and ρ, ρ' the right and left unitors respectively. The diagrams (MF. 1), (MF. 2) and (MF. 3) play a key role in the lifting of operations and equations in this section and the next. In particular they ensure that any unital (resp. associative) operation lifts to a unital (resp. associative) operation. We will sometimes refer to ψ as the *Fubini transformation* of F.

A *monoidal monad* T on a monoidal category is a monad whose underlying functor is monoidal for a natural transformation $\psi_{X,Y} : TX \otimes TY \to T(X \otimes Y)$ and $\psi^0 = \eta_I$, the unit of the monad at I, and whose unit and multiplication are monoidal natural transformations, that is to say:

$$\begin{array}{ccc} X \otimes Y & \xrightarrow{\eta_X \otimes \eta_Y} & TX \otimes TY \\ & {\scriptstyle \eta_{X \otimes Y}} \searrow & \downarrow{\scriptstyle \psi_{X,Y}} \\ & & T(X \otimes Y) \end{array} \quad \text{(MM.1)}$$

$$\begin{array}{ccccc} T^2X \otimes T^2Y & \xrightarrow{\psi_{TX,TY}} & T(TX \otimes TY) & \xrightarrow{T\psi_{X,Y}} & TT(X \otimes Y) \\ \downarrow{\scriptstyle \mu_X \otimes \mu_Y} & & & & \downarrow{\scriptstyle \mu_{X \otimes Y}} \\ TX \otimes TY & & \xrightarrow{\qquad\qquad\psi_{X,Y}\qquad\qquad} & & T(X \otimes Y) \end{array} \quad \text{(MM.2)}$$

Moreover, a monoidal monad is called *symmetric monoidal* if

$$\begin{array}{ccc} TX \otimes TY & \xrightarrow{\psi_{X,Y}} & T(X \otimes Y) \\ \downarrow{\scriptstyle \mathsf{swap}_{TX,TY}} & & \downarrow{\scriptstyle T\mathsf{swap}_{X,Y}} \\ TY \otimes TX & \xrightarrow[\psi_{Y,X}]{} & T(Y \otimes X) \end{array} \quad \text{(SYM)}$$

where $\mathsf{swap} : (-) \otimes (-) \to (-) \otimes (-)$ is the argument-swapping transformation (natural in both arguments).

We now present a result which shows that for monoidal categories which are sufficiently similar to $(\mathbf{Set}, \times, 1)$, being monoidal is equivalent to being symmetric monoidal. The criteria on $(\mathbf{C}, \otimes, I)$ in the following theorem are due to [27] and generalize the strength unicity result of [25, Prop. 3.4]. Our usage of the concept of strength in what follows is purely technical, it is the monoidal structure which

is our main object of interest. We therefore refer the reader to e.g. [25] for the definitions of strength and commutative monad.

Theorem 2. *Let $T : \mathbf{C} \to \mathbf{C}$ be a monad over a monoidal category $(\mathbf{C}, \otimes, I)$ whose tensor unit I is a separator of $\mathbf{C}$ (i.e. $f, g : X \to Y$ and $f \neq g$ implies $\exists x : I \to X$ s.th. $f \circ x \neq g \circ x$) and such that for any morphism $z : I \to X \otimes Y$ there exist $x : I \to X, y \to Y$ such that $z = (x \otimes y) \circ \rho_I^{-1}$. Then t.f.a.e.*

(i) There exists a unique natural transformation $\psi_{X,Y} : TX \otimes TY \to T(X \otimes Y)$ making T monoidal
(ii) There exists a unique strength $\mathsf{st}_{X,Y} : X \times TY \to T(X \otimes Y)$ making T commutative
(iii) There exists a unique natural transformation $\psi_{X,Y} : TX \otimes TY \to T(X \otimes Y)$ making T symmetric monoidal

In particular, monoidal monads on $(\mathbf{Set}, \times, 1)$ are necessarily symmetric (and thus commutative). As we will see in the next section (Theorem 7), this symmetry has deep consequences: it means that a large syntactically definable class of equations can always be lifted by monoidal monads.

3.2 Lifting operations

First though, we show that being monoidal allows us to lift *operations*. The following Theorem is well-known and can be found in e.g. [29,24].

Theorem 3. *Let $T : \mathbf{Set} \to \mathbf{Set}$ be a monoidal monad, then for any finitary signature Σ, there exists a distributive law $\lambda^\Sigma : \mathsf{H}_\Sigma T \to T\mathsf{H}_\Sigma$ of the polynomial functor associated with Σ over T.*

The distributive laws $\lambda^\Sigma : \mathsf{H}_\Sigma T \to T\mathsf{H}_\Sigma$ built from a monoidal structure ψ on T in Theorem 3 have the general shape

$$\mathsf{H}_\Sigma TX = \coprod_{s \in \Sigma} (TX)^{\mathrm{ar}(s)} \xrightarrow{\coprod_{s \in \Sigma} \psi_X^{(\mathrm{ar}(s))}} T\mathsf{H}_\Sigma X \tag{8}$$

where $\psi_X^{(0)} = \eta_1^T, \psi_X^{(1)} = \mathrm{id}_X, \psi_X^{(2)} = \psi_{X,X}$. For $k \geq 3$ if we wanted to be completely rigorous we should first give an evaluation order to the k-fold monoidal product $(TX)^k$ – for example evaluating the products from the left, e.g. $(TX)^3 := (TX \otimes TX) \otimes TX$ – and then define $\psi^{(k)} : (TX)^k \to T(X^k)$ accordingly by repeated application of the Fubini transformation ψ – for example defining

$$\psi_X^{(3)} = \psi_{X \otimes X, X} \circ (\psi_{X,X} \times \mathrm{id}) : (TX \otimes TX) \otimes TX \to T((X \otimes X) \otimes X)$$

However, we will in general be interested in a variety of evaluation orders for the tensors (depending on circumstances), and since in **Set** these different evaluation

orders are related by a combination of associators $\alpha_{X,Y,Z}$ which simply re-bracket tuples, we will abuse notation slightly and write

$$\psi_X^{(k)} : (TX)^k \to T(X^k)$$

with the understanding that $\psi_X^{(k)}$ is only defined up to re-bracketing of tuples which is quietly taking place 'under the hood' as called for by the particular situation. The distributive laws defined by Theorem 3 can be extended to distributive laws for the free monad associated with the signature Σ.

Proposition 1. *Given a finitary signature Σ and a monad $T : \mathbf{Set} \to \mathbf{Set}$, there is a one-to-one correspondence between*

(i) distributive laws $\lambda^\Sigma : \mathsf{H}_\Sigma T \to T\mathsf{H}_\Sigma$ of the polynomial functor *associated with Σ over T*
(ii) distributive laws $\rho^\Sigma : \mathsf{F}_\Sigma T \to T\mathsf{F}_\Sigma$ of the free monad *associated with Σ over T*

In particular, by Theorem 1, the distributive law (8) also corresponds to a lifting $\widehat{T}$ of T to $\mathcal{EM}(\mathsf{F}_\Sigma) \simeq \mathbf{Alg}(\Sigma)$. Explicitly, given an F_Σ-algebra $\beta : \mathsf{F}_\Sigma X \to X$, $\widehat{T}(X, \beta)$ is defined as the F_Σ-algebra

$$\mathsf{F}_\Sigma TX \xrightarrow{\rho_X^\Sigma} T\mathsf{F}_\Sigma X \xrightarrow{T\beta} TX \tag{9}$$

Thus whenever T is monoidal, we can 'lift' the operations of Σ, or, in programming language terms, we can define the operations of the outer layer (T) on the language defined by the operations of the inner layer (F_Σ).

3.3 Lifting equations

We now show how to go from a lifting of T on $\mathcal{EM}(\mathsf{F}_\Sigma) \simeq \mathbf{Alg}(\Sigma)$ to a lifting of T on $\mathcal{EM}(S) \simeq \mathbf{Alg}(\Sigma, E)$. More precisely, we will now show how to 'quotient' the distributive law $\rho^\Sigma \colon \mathsf{F}_\Sigma T \to T\mathsf{F}_\Sigma$ into a distributive law $\lambda : ST \to TS$. Of course this is not always possible, but in the next section we will give sufficient conditions under which the procedure described below does work. The first step is to define the natural transformation $q \colon \mathsf{F}_\Sigma \twoheadrightarrow S$ which quotients the free Σ-algebras by the equations of E to build the free (Σ, E)-algebra. At each set X, let EX denote the set of pairs $(s, t) \in \mathsf{F}_\Sigma X$ such that $SX \models s = t$ and let π_1, π_2 be the obvious projections. Then q can be constructed via the coequalizers:

$$EX \underset{\pi_2}{\overset{\pi_1}{\rightrightarrows}} \mathsf{F}_\Sigma X \xrightarrow{q_X} SX \tag{10}$$

By construction q is a component-wise regular epi monad morphism ($q \circ \eta = \eta^S$ and $\mu^S \circ qq = q \circ \mu^T$), and it induces a functor $Q : \mathcal{EM}(S) \to \mathcal{EM}(\mathsf{F}_\Sigma)$ defined by

$$Q(\xi \colon SX \to X) = \xi \circ q_X : \mathsf{F}_\Sigma X \to X, \qquad Q(f) = f$$

which is well defined by naturality of q. This functor describes an embedding, in particular it is injective on objects: if $Q(\xi_1) = Q(\xi_2)$ then $\xi_1 \circ q_X = \xi_2 \circ q_X$, and therefore $\xi_1 = \xi_2$ since q_X is a (regular) epi.

Given two terms $u, v \in \mathsf{F}_\Sigma V$, we will say that a lifting $\widehat{T} : \mathbf{Alg}(\Sigma) \to \mathbf{Alg}(\Sigma)$ preserves the equation $u = v$, or by a slight abuse of notation that the monad T preserves $u = v$, if $\widehat{T}\mathcal{A} \models u = v$ whenever $\mathcal{A} \models u = v$. Similarly, we will say that $\widehat{T}$ sends (Σ, E)-algebras to (Σ, E)-algebras if it preserves all the equations in E. Half of the following result can be found in [6] where a distributive law over a *functor* is built in a similar way.

Lemma 2. *If $q\colon \mathsf{F}_\Sigma \twoheadrightarrow T$ is a component-wise epi monad morphism, ρ^Σ is a distributive law of the monad F_Σ over the monad T and if there exists a natural transformation $\lambda\colon ST \to TS$ such that the following diagram commutes*

$$\begin{array}{ccc} \mathsf{F}_\Sigma T & \xrightarrow{qT} & ST \\ {\scriptstyle \rho^\Sigma}\downarrow & & \downarrow{\scriptstyle \lambda} \\ T\mathsf{F}_\Sigma & \xrightarrow[Tq]{} & TS \end{array} \tag{11}$$

then λ is a distributive law of the monad S over the monad T.

From this lemma we can give an abstract criterion which, when implemented concretely in the next section, will allow us to go from a lifting of T on $\mathcal{EM}(\mathsf{F}_\Sigma) \simeq \mathbf{Alg}(\Sigma)$ to a lifting of T on $\mathcal{EM}(S) \simeq \mathbf{Alg}(\Sigma, E)$.

Theorem 4. *Suppose $T, S : \mathbf{Set} \to \mathbf{Set}$ are finitary monads, that T is monoidal and that $\mathcal{EM}(S) \simeq \mathbf{Alg}(\Sigma, E)$, and let $\widehat{T} : \mathbf{Alg}(\Sigma) \to \mathbf{Alg}(\Sigma)$ be the unique lifting of T defined via Theorems 1,3 and Proposition 1. If $\widehat{T}$ sends (Σ, E)-algebras to (Σ, E)-algebras, then there exists a natural transformation $\lambda : ST \to TS$ satisfying* (11)*, and therefore a distributive law of S over T.*

4 Checking equation preservation

In Section 3 we showed how to build a lifting of $T\colon \mathbf{Set} \to \mathbf{Set}$ to $\widehat{T} : \mathbf{Alg}(\Sigma) \to \mathbf{Alg}(\Sigma)$ using a Fubini transformation ψ via (8) and (9). In this section we provide a sound method to ascertain whether this lifting sends (Σ, E)-algebras to (Σ, E)-algebras, by giving sufficient conditions for the preservation of equations. We assume throughout this section that T is monoidal, in particular T lifts to $\mathbf{Alg}(\Sigma)$ for any finitary signature Σ. We will denote by $\mathsf{U}_\Sigma : \mathbf{Alg}(\Sigma) \to \mathbf{Set}$ the obvious forgetful functor.

4.1 Residual diagrams

We fix a finitary signature Σ and let u, v be Σ-terms over a set of variables V. Recall that the monad T preserves the equation $u = v$ if $\widehat{T}\mathcal{A} \models u = v$ whenever

$\mathcal{A} \models u = v$. If t is a Σ-term, we will denote by $Var(t)$ the *set of variables* in t and by $Arg(t)$ the *list of arguments* used in t ordered as they appear in t. For example, the list of arguments of $t = f(x_1, g(x_3, x_2), x_1)$ is $Arg(t) = [x_1, x_3, x_2, x_1]$.

Let V be a set of variables and $\mathcal{A}$ be a Σ-algebra with carrier A, we define the morphism $\delta^V_{\mathcal{A}}(t) : A^{|V|} \to A^k$ where $k = |Arg(t)|$ as the following pairing of projections:

$$\text{if } Arg(t) = [x_{i_1}, x_{i_2}, x_{i_3}, \dots x_{i_k}] \text{ then } \delta^V_{\mathcal{A}}(t) = \langle \pi_{i_1}, \pi_{i_2}, \pi_{i_3}, \dots \pi_{i_k} \rangle$$

Intuitively, this pairing rearranges, copies and duplicates the variables used in t to match the arguments. Next, we define $\sigma^V_{\mathcal{A}}(t) \colon A^k \to A$ inductively by:

$$\sigma^V_{\mathcal{A}}(x) = \mathrm{id}_A$$
$$\sigma^V_{\mathcal{A}}(f(t_1, \dots, t_i)) = A^k \xrightarrow{\sigma^V_{\mathcal{A}}(t_1) \times \dots \times \sigma^V_{\mathcal{A}}(t_i)} A^i \xrightarrow{f_{\mathcal{A}}} A$$

With $f_{\mathcal{A}}$ the interpretation of $f \in \Sigma$ in $\mathcal{A}$. Finally we define $[\![t]\!]^V_{\mathcal{A}}$ as $\sigma^V_{\mathcal{A}}(t) \circ \delta^V_{\mathcal{A}}(t)$. The following lemma follows easily from the definitions.

Lemma 3. *For any $t \in \mathsf{F}_\Sigma V$, $\delta^V_{\mathcal{A}}(t), \sigma^V_{\mathcal{A}}(t)$, and thus $[\![t]\!]^V_{\mathcal{A}}$, are natural in $\mathcal{A}$.*

We can therefore re-interpret any term $t \in \mathsf{F}_\Sigma V$ as a natural transformation $[\![t]\!]^V : (-)^{(|V|)}\mathsf{U}_\Sigma \to \mathsf{U}_\Sigma$ which is itself the composition of two natural transformations. The first one, $\delta^V(t) : (-)^{|V|}\mathsf{U}_\Sigma \to (-)^k\mathsf{U}_\Sigma$, 'prepares' the variables by swapping, copying and deleting them as appropriate. The second one, $\sigma^V(t) : (-)^k\mathsf{U}_\Sigma \to \mathsf{U}_\Sigma$, performs the evaluation at each given algebra. Of course, the usual soundness and completeness property of term functions still holds.

Lemma 4. *For $\mathcal{A}$ a Σ-algebra and $u, v \in \mathsf{F}_\Sigma V$, $[\![u]\!]^V_{\mathcal{A}} = [\![v]\!]^V_{\mathcal{A}}$ iff $\mathcal{A} \models u = v$.*

Now consider the following diagram:

$$\begin{array}{ccccc}
 & & [\![t]\!]^V_{\widehat{T}} & & \\
 & \overset{\frown}{} & & \overset{\frown}{} & \\
(-)^{(|V|)}\mathsf{U}_\Sigma\widehat{T} & \xrightarrow{\delta^V_{\widehat{T}}(t)} & (-)^k\mathsf{U}_\Sigma\widehat{T} & \xrightarrow{\sigma^V_{\widehat{T}}(t)} & \mathsf{U}_\Sigma\widehat{T} \\
\downarrow{\scriptstyle \psi^{|V|}_{\mathsf{U}_\Sigma}} & \textcircled{r} & \downarrow{\scriptstyle \psi^{(k)}_{\mathsf{U}_\Sigma}} & \textcircled{q} & \downarrow{\scriptstyle \mathsf{U}_\Sigma \mathrm{id}_{\widehat{T}}} \\
T(-)^{|V|}\mathsf{U}_\Sigma & \xrightarrow[T\delta^V(t)]{} & T(-)^k\mathsf{U}_\Sigma & \xrightarrow[T\sigma^V(t)]{} & \mathsf{U}_\Sigma\widehat{T} \\
 & \underset{\frown}{} & & \underset{\frown}{} & \\
 & & T[\![t]\!]^V & &
\end{array} \tag{12}$$

Since $\mathsf{U}_\Sigma \circ \widehat{T} = T \circ \mathsf{U}_\Sigma$ by definition of liftings it is clear that the vertical arrows $\psi^{(|V|)}_{\mathsf{U}_\Sigma}$ and $\psi^{(k)}_{\mathsf{U}_\Sigma}$ are well-typed. We define $Pres(T, t, V)$ as the outer square of Diagram (12) and we call the left-hand square $\textcircled{r}$ the *residual diagram* $\mathcal{R}(T, t, V)$. The following Lemma is at the heart of our method for building distributive laws.

Lemma 5. *If $\mathcal{R}(T,t,V)$ commutes, then $Pres(T,t,V)$ commutes.*

The following soundness theorem follows immediately from Lemma 5.

Theorem 5. *If $u,v \in \mathsf{F}_\Sigma V$ are such that $\mathcal{R}(T,u,V)$ and $\mathcal{R}(T,v,V)$ commute, then T preserves $u=v$.*

Proof. If $\mathcal{A} \models u=v$, then $[\![u]\!]^V_{\mathcal{A}} = [\![v]\!]^V_{\mathcal{A}}$ by Lemma 4 and thus $T[\![u]\!]^V_{\mathcal{A}} \circ \psi_A^{(|V|)} = T[\![v]\!]^V_{\mathcal{A}} \circ \psi_A^{(|V|)}$. Since $\mathcal{R}(T,u,V)$ and $\mathcal{R}(T,v,V)$ commute, so do $Pres(T,u,V)$ and $Pres(T,v,V)$ by Lemma 5, and therefore $[\![u]\!]^V_{\widehat{T}\mathcal{A}} = [\![v]\!]^V_{\widehat{T}\mathcal{A}}$, that is to say $\widehat{T}\mathcal{A} \models u=v$ by Lemma 4.

Therefore residual diagrams act as sufficient conditions for equation preservation. Note that these diagrams only involve ψ, projections and the monad T, sometimes inside pairings. In other words, the actual operations of Σ appearing in an equation have no impact on its preservation. What matters is the variable rearrangement transformations $\delta^V(u)$ and $\delta^V(v)$, and how they interact with the Fubini transformation ψ.

The converse of Theorem 5 does not hold. Consider the powerset monad $\mathcal{P}$ and a Σ-algebra $\mathcal{A}$ with Σ containing a binary operation $\bullet$. Clearly $\widehat{\mathcal{P}}\mathcal{A} \models x \bullet x = x \bullet x$ whenever $\mathcal{A} \models x \bullet x = x \bullet x$, because the equation trivially holds in any Σ-algebra. In other words, it is preserved by $\mathcal{P}$. However $\mathcal{R}(\mathcal{P}, x \bullet x, \{x\})$ does not commute: provided that X has more than one element, it is easy to see that $\mathcal{R}(\mathcal{P}, x \bullet x, \{x\})$ evaluated at X is

$$\begin{array}{ccc} \mathcal{P}A & \xrightarrow{\Delta_{\mathcal{P}A}} & (\mathcal{P}A)^2 \\ {\scriptstyle \mathrm{id}_{\mathcal{P}A}}\downarrow & & \downarrow{\scriptstyle -\times -} \\ \mathcal{P}A & \xrightarrow[\mathcal{P}(\Delta_A)]{} & \mathcal{P}(A^2) \end{array}$$

where Δ is the diagonal transformation and $-\times-$ is the monoidal structure for $\mathcal{P}$ which takes the Cartesian product. This diagram does not commute (in other words $\mathcal{P}$ is not 'relevant', see below).

4.2 Examples of residual diagrams

We need *a priori* two diagrams per equation to verify preservation. However, in many cases diagrams will be trivially commuting. For instance, associativity and unit produce trivial diagrams. For associativity we assume a binary operation $\bullet \in \Sigma$, let $V=\{x,y,z\}$ and compute that $\delta^V_{\mathcal{A}}(x\bullet(y\bullet z)) = \langle \pi_1,\pi_2,\pi_3\rangle : A^3 \to A^3$ which is just id_{A^3}. It follows that $\mathcal{R}(T, x\bullet(y\bullet z), V)$ commutes since $\psi^3 \circ \mathrm{id}_{TA^3} = T\mathrm{id}_{A^3} \circ \psi^3$ which trivially holds. The argument for $(x\bullet y)\bullet z$ is identical, thus associativity is *always* lifted. The same argument shows that units are always lifted as well. This is not completely surprising since we have built-in units and associativity via Diagrams (MF. 1), (MF. 2) and (MF. 3).

Let us now consider commutativity: $x \bullet y = y \bullet x$. In this case, we put $V = \{x, y\}$ and hence $\delta^V_{\mathcal{A}}(x \bullet y) = \mathrm{id}_{\mathcal{A}}$ and $\mathcal{R}(T, x \bullet y, V)$ obviously commutes for the same reason as before. Similarly, it is not hard to check that $\mathcal{R}(T, y \bullet x, V)$ is just diagram (SYM), which we know holds by our assumption that T is monoidal and Theorem 2. It follows that:

Theorem 6. *Monoidal monads preserve associativity, unit and commutativity.*

Some equations are not always preserved by commutative monads, we present here two important examples.

Idempotency: $x \bullet x = x$
$\mathcal{R}(T, x \bullet x, \{x\})$ given by:

$$\begin{array}{ccc} TA & \xleftarrow{\mathrm{id}} & TA \\ {\scriptstyle T\langle \pi_1, \pi_1 \rangle}\downarrow & & \downarrow{\scriptstyle \langle \pi_1, \pi_1 \rangle} \\ T(A^2) & \xleftarrow[\psi]{} & (TA)^2 \end{array}$$

Absorption: $x \bullet 0 = 0$
$\mathcal{R}(T, x \bullet 0, \{x\})$ given by:

$$\begin{array}{ccc} TA & \xleftarrow{\mathrm{id}} & TA \\ {\scriptstyle T!}\downarrow & & \downarrow{\scriptstyle !} \\ T1 & \xleftarrow[\eta_1]{} & 1 \end{array}$$

(13)

These diagrams correspond to classes of monads studied in the literature. The residual diagram for idempotency can be expressed as the equation $\psi_{A,A} \circ \Delta_{TA} = T\Delta_A$, where Δ is the diagonal operator. A monad T verifying this condition is called *relevant* by Jacobs in [13]. Similarly, one easily shows that the commutativity of the absorption diagram is equivalent to the definition of *affine* monads in [18,13].

4.3 General criteria for equation preservation

As shown in lemma 5 and Theorem 5, the interaction between T and the variable rearrangements operated by δ^V can provide a sufficient condition for the preservation of equations. We will focus on three important types of interaction between a monad T and rearrangement operations. First, the residual diagram for commutativity, i.e. Diagram (SYM), which corresponds to saying that 'T preserves variable swapping', i.e. that T is commutative/symmetric monoidal, or in logical terms to the exchange rule. As we have seen, this condition *must* be satisfied in order to simply lift operations, so we must take it as a basic assumption. Second, the residual diagram for idempotency (leftmost diagram of (13)) which corresponds to 'T preserves variable duplications', i.e. that T is *relevant*, or in logical terms to the weakening rule. Finally, the residual diagram for absorption (rightmost diagram of (13)) which corresponds to 'T allows to drop variables', i.e. T is *affine*, or in logical terms to the contraction rule. To each of these residual diagrams corresponds a syntactically definable class of equations which are automatically preserved by a monad satisfying the residual diagram.

Theorem 7. *Let T be a commutative monad. If $Var(u) = Var(v)$ and if variables appear exactly once in u and in v, then T preserves $u = v$.*

Note that this theorem can be found in [23], where this type of equation is called *linear*. Moreover, $\mathcal{P}$ is within the scope of this result, which generalises one direction of Gautam's theorem ([10]). Let us now present original results by first treating the case where variables may appear several times.

Theorem 8. *Let T be a commutative relevant monad. If $Var(u) = Var(v)$, then T preserves $u = v$.*

Commutative relevant monads seem to preserve many algebraic laws. However, in the case where both sides of the equation do not contain the same variables, for instance $x \bullet 0 = 0$, Theorem 8 does not apply. Intuitively, the missing piece is the ability to *drop* some of the variables in V.

Theorem 9. *Let T be a commutative affine monad. If variables appear at most once in u and in v, then T preserves $u = v$.*

Combining the results of Theorems 8 and 9, one gets a very economical – if very strong – criterion for the preservation of *all* equations.

Theorem 10. *Let T be a commutative, relevant and affine monad. For all u and v, T preserves $u = v$.*

Examining the existence of distributive laws between algebraic theories, as well as stating conditions on variable rearrangements, has been studied before in terms of Lawvere Theories (see for instance [7]). Note that for T commutative monad, being both relevant and affine (sometimes called *cartesian*) is equivalent to preserving products, as seen in [18]. This confirms that such a monad T preserves all equations of the underlying algebraic structure, in other words it always has a distributive law with any other monad. This is however a very strong condition. An example of this type of monad is $T(X) = X^Y$ for Y an object of **Set**.

4.4 Weakening the inner layer when composition fails.

In the case where a residual diagram fails to commute, we cannot conclude that the equation lifts from $\mathcal{A}$ to $\widehat{T}\mathcal{A}$. The non-commutativity of the diagram often provides a counter-example which shows that the equation is in fact not valid in $\widehat{T}\mathcal{A}$ (this is the case of idempotency and distributivity in the next section).

However, if our aim is to build a structure combining all operations used to define T and S, then our method can provide an answer, since it allows us to identify precisely which equations fail to hold. Let E' be the subset of E containing the equations preserved by T. A new monad S' can be derived from signature Σ and equations E' using an adjunction of type (6). Since E' only contains equations preserved by T, by theorem 4 the composition TS' creates a monad, and its algebraic structure contains all the constructs derived from the original signature Σ, as well as the new symbols arising from T.

This method for fixing a faulty monad composition follows the idea of loosening the constraints of the *inner* layer, meaning in this case modifying S to construct a monad resembling TS. The best approximate language we obtain has the desired signature, but has lost some of the laws described by S. We illustrate this method in the following section.

5 Application

As sketched in the introduction, our method aims to incrementally build an imperative language: starting with sequential composition, we add a layer providing non-deterministic choice, then a layer for probabilistic choice.

Adding the non-deterministic layer. We start with the simple programming language described in the introduction by the signature (15) and equations (16) – or, equivalently, by the monad $(-)^*$ – and let $\mathtt{A}$ be a set of atomic programs. Our minimal language is thus given by $\mathtt{A}^*$. Note that the free monoid is not commutative and thus in our method it cannot be used as an outer layer, it has to constitute the core of the language we build. More generally, our method provides a simple heuristic for compositional language building: always start with the non-commutative monad.

We now add non-determinism via the finitary powerset monad $\mathcal{P}$, which is simply the free join semi-lattice monad. To build this extension, we want to combine both monads to create a new monad $\mathcal{P}((-)^*)$. As we have shown in Theorem 4, it suffices to build a lifting of monad $\mathcal{P}$ to **Mon**, the category of algebras for the signature (15) and equations (16). For this purpose we apply the method given in section 4.

The first step is lifting $\mathcal{P}$ to the category of $\{\mathtt{skip},\ ;\}$-algebras, which means lifting the operations of $\mathtt{A}^*$ to $\mathcal{P}(\mathtt{A}^*)$ using a Fubini map. It is well-known that the powerset monad is commutative, and it follows in particular that there exists a unique symmetric monoidal transformation $\psi\colon \mathcal{P}\times\mathcal{P}\to\mathcal{P}(-\times-)$ which is given by the Cartesian product: for $U\in\mathcal{P}(X), V\in\mathcal{P}(Y)$, we take $\psi_{X,Y}(U,V)=U\times V$. Using this Fubini transformation, we can now define the interpretation in $\mathcal{P}(\mathtt{A}^*)$ of $\mathtt{skip}$ and $;$ as:

$$\widehat{\mathtt{skip}} = \mathcal{P}(\mathtt{skip})\circ\eta_1(*) = \{\varepsilon\}$$
$$\hat{;} = \mathcal{P}(\,;\,)\circ\psi_{\mathtt{A}^*,\mathtt{A}^*}\colon (\mathcal{P}\mathtt{A}^*)^2\to\mathcal{P}A^*,\quad (U,V)\mapsto\{u\,;v\mid u\in U, v\in V\}$$

To check that this lifting defines a lifting on **Mon**, we need to check that equations (16) hold in $\mathcal{P}(\mathtt{A}^*)$. These equations describe associativity and unit: by Theorem 6, they are always preserved by a strong commutative monad like $\mathcal{P}$.

It follows from Theorem 4 and 5 that we obtain a distributive law $\lambda\colon(\mathcal{P}(-))^*\to\mathcal{P}((-)^*)$ between monads $(-)^*$ and $\mathcal{P}$, hence the composition $\mathcal{P}((-)^*)$ is also a monad, allowing us to apply our method again and potentially add another monadic layer. The language $\mathcal{P}(\mathtt{A}^*)$ contains the lifted versions $\widehat{\mathtt{skip}}$ and $\hat{;}$ of our previous constructs as well as the new operations arising from $\mathcal{P}$, namely

a non-deterministic choice operation $+$, which is associative, commutative and idempotent, and its unit $\texttt{abort}$. Note that since the monad structure on $\mathcal{P}((-)^*)$ is defined by a distributive law of $(-)^*$ over $\mathcal{P}$, the set of equations E is made of the equations (16) arising from $(-)^*$, the equations (17) arising from $\mathcal{P}$, and finally the equations (4) expressing distributivity of operations of $(-)^*$ over those of $\mathcal{P}$. The language we have built so far has the structure of an *idempotent semiring*.

Adding the probabilistic layer. We will now enrich our language further by adding a probabilistic layer. Specifically, we will add the family of probabilistic choice operators $\oplus_\lambda$ for $\lambda \in [0,1]$ satisfying the axioms of convex algebras, i.e.

$$\mathtt{p} \oplus_\lambda \mathtt{p} = \mathtt{p} \qquad \mathtt{p} \oplus_\lambda \mathtt{q} = \mathtt{q} \oplus_{1-\lambda} \mathtt{p} \qquad \mathtt{p} \oplus_\lambda (\mathtt{q} \oplus_\tau \mathtt{r}) = (\mathtt{p} \oplus_{\frac{\lambda}{\lambda+(1-\lambda)\tau}} \mathtt{q}) \oplus_{\lambda+(1-\lambda)\tau} \mathtt{r} \tag{14}$$

From a monadic perspective, we want to examine the composition of monads $\mathcal{D}(\mathcal{P}((-)^*))$. It is known (see [30]) that $\mathcal{D}$ does not distribute over $\mathcal{P}$. We will see that our method confirms this result.

We start by lifting the constants and operations $\{\texttt{skip}, \texttt{abort}, \,;\,, +\}$ of $\mathcal{P}((-)^*)$ by defining a Fubini map $\psi : \mathcal{D}(-) \times \mathcal{D}(-) \to \mathcal{D}(- \times -)$. It is well-known that $\mathcal{D}$ is a commutative monad and that the product of measures defines the Fubini transformation. In the case of finitely supported distributions the product of measures can be expressed simply as follows: given distributions $\mu \in \mathcal{D}X, \nu \in \mathcal{D}Y$, $\psi(\mu,\nu)$ is the distribution on $X \times Y$ defined on singletons $(x,y) \in X \times Y$ by $(\psi(\mu,\nu))(x,y) = \mu(x)\nu(y)$. Theorem 7 tells us that associativity, commutativity and unit are preserved by $\mathcal{D}$. It follows that the associativity of both $\,;\,$ and $+$ is preserved by the lifting operation, and the liftings of $\texttt{skip}$ and $\texttt{abort}$ are their respective units. Furthermore, the lifting of $+$ is commutative.

We know from Theorem 8 that the idempotency of $+$ will be preserved if $\mathcal{D}$ is relevant. It is easy to see that $\mathcal{D}$ is badly non-relevant: consider the set $X = \{a,b\}, a \neq b$ and any measure μ on X which assigns non-zero probability to both a and b. We have:

$$\begin{aligned}
\psi(\Delta_{\mathcal{D}X}(\mu))(a,b) &= (\psi(\mu,\mu))(a,b) \\
&= \mu(a)\mu(b) \neq 0 \\
&= \mu(\emptyset) \\
&= \mu\{x \in X \mid \Delta_X(x) = (a,b)\} \\
&= \mathcal{D}(\Delta_X)(\mu)(a,b)
\end{aligned}$$

It follows that we *cannot* conclude that the lifting $\widehat{\mathcal{D}} : \mathbf{Alg}(\{\texttt{skip}, \texttt{abort}, \,;\,, +\}) \to \mathbf{Alg}(\{\texttt{skip}, \texttt{abort}, \,;\,, +\})$ defined by the product of measures following (8) sends idempotent semirings to idempotent semirings, and therefore we cannot conclude that $\mathcal{D}(\mathcal{P}(-)^*)$ is a monad (in fact we know it isn't). It is very telling that idempotency also had to be dropped in the design of the probabilistic network specification language ProbNetKAT (see [8, Lemma 1]) which is very similar to the language we are trying to incrementally build in this Section.

Requiring that + be idempotent is an algebraic obstacle, so let us now remove it and replace as our inner layer the monad building free idempotent semirings – that is to say $\mathcal{P}(-)^*$ – by the monad building free semirings – that is to say $\mathcal{M}(-)^*$, where $\mathcal{M}$ is the multiset monad ($\mathcal{M}$ can also be described as the free commutative monoid monad). Since we have already checked that the $\mathcal{D}$-liftings of binary operations preserve associativity, units and commutativity, it only remains to check that they preserve the distributivity of ; over +. The equation for distributivity belongs to the syntactic class covered by Theorem 8 since it has the same set of variables on each side (but one of them is duplicated, so we fall outside the scope of Theorems 7 and 9). Since we've just shown that $\mathcal{D}$ is not relevant, it follows that we cannot lift the distributivity axioms. So we must weaken our inner layer even further and consider a structure consisting of two monoids, one of which is commutative. Interestingly, the failure of distributivity was also observed in the development of ProbNetKAT ([8, Lemma 4]), and therefore should not come as a surprise.

Having removed the two distributivity axioms we are left with only the absorption laws to check. In this case the equation has no variable duplication, but has not got the same number of variables on each side of the equation, absorption therefore falls in the scope of Theorem 9, and we need to check if $\mathcal{D}$ is affine. Since $\mathcal{D}1 \simeq 1$, it is trivial to see that $\eta_1 \circ ! = \mathcal{D}!$ and hence $\mathcal{D}$ is affine. By Theorem 9, the absorption law is therefore preserved by the probabilistic extension. It follows that the probabilistic layer $\mathcal{D}$ can be composed with the inner layer consisting of the signature $\{\mathtt{abort}, \mathtt{skip}, \,;\,, +\}$ and the axioms

(i) $\mathtt{p}\,;\mathtt{skip} = \mathtt{skip}\,;\mathtt{p} = \mathtt{p}$
(ii) $(\mathtt{p}\,;\mathtt{q})\,;\mathtt{r} = \mathtt{p}\,;(\mathtt{q}\,;\mathtt{r})$
(iii) $\mathtt{p} + \mathtt{abort} = \mathtt{abort} + \mathtt{p} = \mathtt{p}$
(iv) $\mathtt{p} + \mathtt{q} = \mathtt{q} + \mathtt{p}$
(v) $(\mathtt{p} + \mathtt{q}) + \mathtt{r} = \mathtt{p} + (\mathtt{q} + \mathtt{r})$
(vi) $\mathtt{p}\,;\mathtt{abort} = \mathtt{abort} = \mathtt{abort}\,;\mathtt{p}$

i.e. two monoids, one of them commutative, with the absorption law as the only interaction between the two operations. This structure, combined with the axioms of convex algebras (14) and the distributivity axioms

(Dst i) $\mathtt{p}\,;(\mathtt{q} \oplus_\lambda \mathtt{r}) = (\mathtt{p}\,;\mathtt{q}) \oplus_\lambda (\mathtt{p}\,;\mathtt{r})$
(Dst ii) $(\mathtt{q} \oplus_\lambda \mathtt{r})\,;\mathtt{p} = (\mathtt{q}\,;\mathtt{p}) \oplus_\lambda (\mathtt{r}\,;\mathtt{p})$
(Dst iii) $\mathtt{p}+(\mathtt{q}\oplus_\lambda \mathtt{r}) = (\mathtt{p}+\mathtt{q})\oplus_\lambda(\mathtt{p}+\mathtt{r})$
(Dst iv) $(\mathtt{q}\oplus_\lambda \mathtt{r})+\mathtt{p} = (\mathtt{q}+\mathtt{p})\oplus_\lambda(\mathtt{r}+\mathtt{p})$

forms the 'best approximate language' combining sequential composition, nondeterministic choice and probabilistic choice. Note that the distributive laws above makes good semantic sense, and indeed hold for the semantics of ProbNetKAT. What we have built modularly in this section is essentially the $*$-free and test-free fragment of ProbNetKAT.

6 Discussion and future work.

We have provided a principled approach to building programming languages by incrementally layering features on the top one another. We believe that our

approach is close in spirit to how programming languages are typically constructed, that is to say by an incremental enrichment of the list of features, and to the search for modularity initiated by foundational papers [25] and [20].

Our method has assumed throughout that the monad for the outer layer had to be monoidal/commutative. Our method can in fact be straightforwardly extended to monads satisfying only (MM.1) and (MM.2). In practice however, the generality gained in this way is very limited: only a monoidal monad will lift an associative operation with a left and right unit, and given the importance of sequential composition with $\mathtt{skip}$, the restriction we have placed on our method appears fairly natural and benign.

We must be careful about how layers are composed together: our approach yields distributive interactions between them, but one might want other sorts of interactions. Consider for example the minimal programming language $\mathtt{P}^*$ described in Section 1, and assume that we now want to add a concurrent composition operation $\|$ to this language with the natural axiom $\mathtt{p} \parallel \mathtt{skip} = \mathtt{p} = \mathtt{p} \parallel \mathtt{skip}$. This addition is not as simple as layering described in Section 5, as the new construct has to interact with the core layer in a whole new way: $\mathtt{skip}$ must be the unit of $\|$ as well. In such cases our approach is not satisfactory, and two alternative strategies present themselves to us: we can consider 'larger' layers, for example the combined theory of sequential composition, $\mathtt{skip}$ and $\|$ described above as a single entity. However, the more complex an inner layer is, the less likely it is that an outer layer with lift it in its entirety. Alternatively, we may want to integrate our technique with Hyland and Power's methods ([12]) and combine some layers with sums and tensors, and others with distributive laws, depending on semantic and algebraic considerations.

A comment about our 'approximate language' strategy is also in order. As explained in Section 4, when an equation of the inner layer prevents the existence of a distributive law we choose to remove this equation, i.e. to loosen the inner layer. Another option is in principle possible: we could constrain the outer layer until it becomes compatible with the inner layer. We would obtain in this case a replacement candidate for one of our monads in order to achieve composition. In the case of $\mathcal{D}(\mathcal{P}(-)^*)$ this would be a particularly unproductive idea since the only elements of $\mathcal{D}(\mathcal{P}(-)^*)$ which satisfy the residual diagram for idempotency are Dirac deltas, i.e. we would get back the language $\mathcal{P}(-)^*$.

Another obvious avenue of research is to extend our method to programming languages specified by more than just equations. One example is the so-called 'exchange law' in concurrency theory given by $(\mathtt{p} \parallel \mathtt{r}) \,;\, (\mathtt{q} \parallel \mathtt{s}) \sqsubseteq (\mathtt{p} \,;\, \mathtt{q}) \parallel (\mathtt{r} \,;\, \mathtt{s})$ which involves a native pre-ordering on the collection of programs, i.e. moving from the category of sets to the category of posets. Another example are Kozen's quasi-equations ([19]) axiomatizing the Kleene star operations, for example $\mathtt{p} \,;\, \mathtt{x} \leq \mathtt{x} \Rightarrow \mathtt{p}^* \,;\, \mathtt{x} \leq \mathtt{x}$. This problem is much more difficult and involves moving away from monads and distributive laws altogether since quasi-varieties are in general not monadic categories.

References

1. S. Awodey. *Category theory*. Oxford University Press, 2010.
2. A. Balan and A. Kurz. On coalgebras over algebras. *Theoretical Computer Science*, 412(38):4989–5005, 2011.
3. J. Beck. Distributive laws. In *Seminar on triples and categorical homology theory*, pages 119–140. Springer, 1969.
4. N. Benton, J. Hughes, and E. Moggi. Monads and effects. *Lecture notes in computer science*, 2395:42–122, 2002.
5. F. Bonchi, A. Silva, and A. Sokolova. The power of convex algebras. *arXiv preprint arXiv:1707.02344*, 2017.
6. M. M. Bonsangue, H. H. Hansen, A. Kurz, and J. Rot. Presenting distributive laws. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 95–109. Springer, 2013.
7. E. Cheng. Distributive laws for lawvere theories. *arXiv preprint arXiv:1112.3076*, 2011.
8. N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic netkat. In *European Symposium on Programming Languages and Systems*, pages 282–309. Springer, 2016.
9. N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *ACM SIGPLAN Notices*, volume 50, pages 343–355. ACM, 2015.
10. N.D. Gautam. The validity of equations of complex algebras. *Archiv für Mathematische Logik und Grundlagenforschung*, 3(3-4):117–124, 1957.
11. M. Hyland, P. Levy, G. Plotkin, and J. Power. Combining continuations with other effects. In *Proc. Continuations Workshop*, 2004.
12. M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.
13. B. Jacobs. Semantics of weakening and contraction. *Annals of pure and applied logic*, 69(1):73–106, 1994.
14. B. Jacobs, A. Silva, and A. Sokolova. Trace semantics via determinization. In *CMCS*, volume 12, pages 109–129. Springer, 2012.
15. D. J. King and P. Wadler. Combining monads. In *Functional Programming, Glasgow 1992*, pages 134–143. Springer, 1993.
16. B. Klin and J. Rot. Coalgebraic trace semantics via forgetful logics. In *International Conference on Foundations of Software Science and Computation Structures*, pages 151–166. Springer, 2015.
17. B. Klin and J. Salamanca. Iterated covariant powerset is not a monad. *MFPS XXXIV*, 2018.
18. A. Kock. Bilinearity and cartesian closed monads. *Mathematica Scandinavica*, 29(2):161–174, 1972.
19. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *Proc. 6th Symp. Logic in Comput. Sci.*, pages 214–225, Amsterdam, July 1991. IEEE.
20. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. *Programming Languages and SystemsESOP'96*, pages 219–234, 1996.
21. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.
22. S. Mac Lane. *Categories for the working mathematician*, volume 5. Springer, 2013.

23. E. Manes and P. Mulry. Monad compositions i: general constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.
24. S. Milius, T. Palm, and D. Schwencke. Complete iterativity for algebras with effects. In *CALCO*, volume 9, pages 34–48. Springer, 2009.
25. E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.
26. G. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS*, volume 2, pages 342–356. Springer, 2002.
27. T. Sato. The Giry monad is not strong for the canonical symmetric monoidal closed structure on Meas. *Journal of Pure and Applied Algebra*, 2017.
28. S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva. Cantor meets scott: Semantic foundations for probabilistic networks. *arXiv preprint arXiv:1607.05830*, 2016.
29. A. Sokolova, B. Jacobs, and I. Hasuo. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3, 2007.
30. D. Varacca. *Probability, nondeterminism and concurrency: two denotational models for probabilistic computation.* PhD thesis, BRICS, 2003.