Monads and Modularity

Christoph Lüth[1] and Neil Ghani[2]

[1] FB 3 — Mathematik und Informatik, Universität Bremen
cxl@informatik.uni-bremen.de
[2] Department of Mathematics and Computer Science, University of Leicester
ng13@mcs.le.ac.uk

Abstract. This paper argues that the core of modularity problems is an understanding of how individual components of a large system interact with each other, and that this interaction can be described by a *layer structure*. We propose a uniform treatment of layers based upon the concept of a *monad*. The combination of different systems can be described by the *coproduct* of monads.
Concretely, we give a construction of the coproduct of two monads and show how the layer structure in the coproduct monad can be used to analyse layer structures in three different application areas, namely term rewriting, denotational semantics and functional programming.

1 Introduction

When reasoning about complex systems (such as specifications of large systems, or semantics of rich languages with many different features), modularity and compositionality are crucial properties: *compositionality* allows a large problem to be broken down into parts which can be reasoned about separately, while *modularity* finds criteria under which results concerning these parts combine into results about the overall system.

A prerequisite for modular reasoning is an understanding of how individual components of a large system interact with each other. In modular term rewriting, the key concept is the *layer structure* on the terms of the combined rewrite system, i.e. we can decompose the combined system into *layers* from the component systems. Our basic observation is that this methodology can be generalized by moving to a categorical framework, where layers become the basic concept, described by *monads*, which describe a far wider class of systems than just term rewriting systems, as demonstrated by the examples below. The combination of smaller systems into a larger one is in general described by colimits, but in this paper, we restrict ourselves a natural first step, the *coproduct*.

Monads have been used to describe various formal systems from term rewriting [10,11] to higher-order logic [3] and arbitrary computations [13], just as colimits have been used to describe the combination of specifications and theories [16,17]. A construction of the colimit of monads was given by Kelly [7, Chapter VIII] but the generality of the construction is reflected in its complexity which can be detering even for experienced category theorists, and hence limits the applicability of the construction to modularity problems.

A. Armando (Ed.): FroCoS 2002, LNAI 2309, pp. 18–32, 2002.

Our contribution is to provide alternative constructions which, by restricting ourselves to special cases, are significantly simpler and hence easier to apply in practice. We describe how monads correspond to algebraic structures, coproducts correspond to disjoint unions and how the layer structure in the coproduct monad models the layer structure in the combined system, and apply these ideas to three settings: modular rewriting, modular denotational semantics, and the functional programming language Haskell.

Generality requires abstraction and, as we shall argue later, the use of monads is appropriate for our abstract treatment of layers. We are aware that our categorical meta-language may make our work less accessible to those members of the FroCoS community who are not proficient in category theory. Nevertheless, we have written this paper with the general FroCoS audience in mind, by focusing on ideas, intuitions and concrete examples; proofs using much category theory have been relegated to the appendix. Our overall aim is to apply our general ideas to specific modularity problems and for that we require an exchange of ideas; we hope this paper can serve as a basis for this interaction.

The rest of this paper is structured as follows: we first give a general account of monads and motivate the construction of their coproducts. We examine particular special cases, for which we can give a simplified account. We finish by detailing our three application areas.

2 An Introduction to Monads

In this section, we introduce monads, describe their applications and explain their relevance to a general treatment of layers. Since this is standard material, we refer the reader to general texts [12] for more details. We start with the canonical example of term algebras which we will use throughout the rest of the paper.

Definition 1 (Signature). *A (single-sorted)* signature *consists of a function* $\Sigma : \mathbb{N} \to \boldsymbol{Set}$*. The set of* n-ary operators *of* Σ *is defined* $\Sigma_n = \Sigma(n)$.

Definition 2 (Term Algebra). *Given a signature* Σ *and a set of variables* X*, the* term algebra $T_\Sigma(X)$ *is defined inductively:*

$$\frac{x \in X}{\text{'}x \in T_\Sigma(X)} \qquad \frac{f \in \Sigma_n \quad t_1, \ldots t_n \in T_\Sigma(X)}{f(t_1, \ldots, t_n) \in T_\Sigma(X)}$$

Quotes are used to distinguish a variable $x \in X$ from the term $\text{'}x \in T_\Sigma(X)$. For every set X, the term algebra $T_\Sigma(X)$ is also a set — categorically $T_\Sigma : \mathbf{Set} \to \mathbf{Set}$ is a functor over the category of sets. In addition, for every set of variables X, there is a function $X \to T_\Sigma(X)$ sending each variable to the associated term. Lastly, substitution takes terms built over terms and flattens them, as described by a function $T_\Sigma(T_\Sigma(X)) \to T_\Sigma(X)$. These three pieces of data, namely the construction of a theory from a set of variables, the embedding of variables as terms and the operation of substitution are axiomatised as a monad:

Definition 3 (Monads). *A* monad $\mathsf{T} = \langle T, \eta, \mu \rangle$ *on a category* $\mathcal{C}$ *is given by an endofunctor* $T : \mathcal{C} \to \mathcal{C}$*, called the* action*, and two natural transformations,* $\eta : 1 \Rightarrow T$*, called the* unit*, and* $\mu : TT \Rightarrow T$*, called the* multiplication *of the monad, satisfying the* monad laws*:* $\mu \cdot T\eta = 1 = \mu \cdot \eta_T$*, and* $\mu \cdot T\mu = \mu \cdot \mu_T$*.*

We have already sketched how the term algebra construction T_Σ has an associated unit and multiplication. The equations of a monad correspond to substitution being well behaved, in particular being associative with the variables forming left and right units. In terms of layers, we think of $T_\Sigma(X)$ as a layer of terms over X, the unit converts each variable into a trivial layer and the multiplication allows us to collapse two layers of the same type into a single layer. Monads model a number of other interesting structures in computer science:

Example 1 (More Complex Syntax). Given an algebraic theory $\mathcal{A} = \langle \Sigma, E \rangle$, the free algebra construction defined by $T_\mathcal{A}(X) = T_\Sigma(X)/\!\sim_E$, where $\sim_E$ is the equivalence relation induced by the equations E, is a monad over **Set**.

A many-sorted algebraic theory $\mathcal{A} = \langle S, \Sigma, E \rangle$, where S is a set of sorts, gives rise to a monad on the base category $\mathbf{Set}^S$ which is the category of S-indexed families of sets and S-indexed families of functions between them.

Calculi with variable binders such as the λ-calculus, can be modeled as a monad over $\mathbf{Set}^\mathcal{F}$ which is the category of functors from the category $\mathcal{F}$ of finite ordinals and monotone functions between them, into the category **Set** [4].

Example 2 (Term Rewriting Systems). Term rewriting systems (TRSs) arise as monads over the category **Pre** of preorders, while labelled TRSs arise as monads over the category **Cat** of categories [10,11].

Example 3 (Computational Monads). Moggi proposed the use of monads to structure denotational semantics where TX is thought of as the computations over basic values X [13]; see Sect. 5 below.

Example 4 (Infinitary Structures). Final coalgebras have recently become popular as a model for infinitary structures. The term algebra from Def. 2 is the initial algebra of the functor T_Σ, and just as initial algebras form a monad, so do final coalgebras: for the signature Σ, the mapping T^∞_Σ sending a set X to the set of finite and infinite terms built over X is a monad [5].

From the perspective of modularity, we regard $T(X)$ as an abstraction of a layer. In the examples above, layers are terms, rewrites, or computations; the monad approach allows us to abstract from their particular properties and concentrate on their interaction. Monads provide an abstract calculus for such layers where the actual layer, the empty layers, and the collapsing of two layers of the same type are are taken as primitive concepts.

3 Coproducts of Monads

Recall our aim is to understand the layer structure in modularity problems by understanding the layer structure in the coproduct of monads. The construction

of the coproduct of monads is rather complex and so we motivate our general construction by considering a simple case, namely the coproduct of two term algebra monads. Given two signatures Σ, Ω with corresponding term algebra monads $\mathsf{T}_\Sigma, \mathsf{T}_\Omega$, the coproduct $\mathsf{T}_\Sigma + \mathsf{T}_\Omega$ should calculate the terms built over the disjoint union $\Sigma + \Omega$, i.e. $\mathsf{T}_\Sigma + \mathsf{T}_\Omega = \mathsf{T}_{\Sigma+\Omega}$.[1]

Terms in $T_{\Sigma+\Omega}(X)$ have an inherent notion of layer: a term in $T_{\Sigma+\Omega}$ decomposes into a term from T_Σ (or T_Ω), and strictly smaller subterms whose head symbols are from Ω (or Σ). This suggests that we can build the action of the coproduct $T_{\Sigma+\Omega}(X)$ by successively applying the two actions (T_Σ and T_Ω):

$$\begin{aligned} T_\Sigma + T_\Omega(X) = {} & X + T_\Sigma(X) + T_\Omega(X) + T_\Sigma T_\Sigma(X) + T_\Sigma T_\Omega(X) + \\ & T_\Omega T_\Sigma(X) + T_\Omega T_\Omega(X) + T_\Sigma T_\Omega T_\Sigma(X) + \dots \end{aligned} \tag{1}$$

Crucially, theories are built over variables, and the instantiation of variables builds layered terms. The quotes of Def. 2 can now be seen as encoding layer information within the syntax. For example, if $\Sigma = \{\mathtt{F}, \mathtt{G}\}$ then the term `G'G'x` is an element of $T_\Sigma(T_\Sigma(X))$ and hence has two Σ-layers. This is different from the term `GG'x` which is an element of $T_\Sigma(X)$ and hence has only one Σ-layer.

Equation (1) is actually too simple. In particular there are different elements of the sum which represent the same element of the coproduct monad, and we therefore need to quotient the sum.

Firstly, consider a variable $x \in X$. Then $x \in X$, $'x \in T_\Sigma(X)$, $''x \in T_\Omega T_\Sigma(X)$. By identifying a layered term with its image under the two units, one can identify these layered terms; we call this the *η-quotient*.

Secondly, if $\Omega = \{\mathtt{H}\}$ is another signature, the layered terms $t_1 = \mathtt{GG'x} \in T_\Sigma(X)$ and $t_2 = \mathtt{G'G'x} \in T_\Sigma(T_\Sigma(X))$ are both layered versions of $\mathtt{GG'x} \in T_{\Sigma+\Omega}(X)$. By identifying a layered term containing a repeated layer with the result of collapsing the layer, one identifies these terms (*μ-quotient*).

Finally, in all elements of the sum (1), descending from the root to a leaf in any path we pass through the same number of quotes. Thus, layered terms such as `F(G'x,'H'x)` do not exist in the sum. However, this term is an element of $T_\Sigma(X + T_\Omega(X))$ which indicates that the layer structure of (1) is not the only possible layer structure. In fact, there are a number of different layer structures which we propose, each with uses in different modularity problems.

Summing up, the coproduct monad $\mathsf{T} + \mathsf{R}$ should be constructed pointwise for any set X of variables as a quotient of layered terms. Layered terms are formed solely by constructions over the component monads. This is crucial, as the construction of the coproduct is compositional, and hence properties of T and R can be lifted to $\mathsf{T} + \mathsf{R}$. The equations are essentially given by the unit and multiplication of the components.

For the rest of this paper, we have to make certain technical assumptions about the two monads and the base category (see Appendix A).

[1] This relies on the fact that the mapping of signatures to monads preserves the coproduct, which it does because it is a left adjoint.

3.1 Pointed Functors

A functor $S : \mathcal{C} \to \mathcal{C}$ with a natural transformation $\sigma : 1 \Rightarrow S$ is called *pointed*. Every monad is pointed, so taking the coproduct of pointed functors is a first step towards the construction of the coproduct of monads. In the term algebra example, the natural transformation $\eta_T : 1 \Rightarrow T_\Sigma$ models the variables, and the coproduct of two pointed functors S, T should be the functor which for any given set X returns the union of TX and SX with the variables identified. This construction therefore implements the η-quotient from above.

In **Set**, we identify elements of a set by taking the quotient. Thus, for example to share the variables from X in $T_\Sigma(X) + T_\Omega(X)$, we quotient the set by the equivalence relation generated by $'x \sim 'y$ (note how the term on the left is an element of $T_\Sigma(X)$, whereas the term on the right is an element of $T_\Omega(X)$). Categorically, this process is modelled by a *pushout*:

Definition 4 (Pointed Coproduct). *Given two pointed functors $\langle T, \eta_T \rangle$ and $\langle S, \eta_S \rangle$, their coproduct is given by the functor $Q : \mathcal{C} \to \mathcal{C}$ which maps every object X in $\mathcal{C}$ to the colimit in (2) with the obvious extension of Q to morphisms. Q*

$$\begin{array}{ccc} X & \xrightarrow{\eta_T} & TX \\ {\scriptstyle \eta_R}\downarrow & & \downarrow{\scriptstyle \sigma_T} \\ RX & \xrightarrow[\sigma_R]{} & QX \end{array} \qquad (2)$$

is pointed with $\sigma : 1 \Rightarrow Q$ given by $\sigma_X = \sigma_R \cdot \eta_R = \sigma_T \cdot \eta_T$.

3.2 Non-collapsing Monads

An algebraic theory $\mathcal{A} = \langle \Sigma, E \rangle$ is non-collapsing if none of the equations has a variable as its left or right-hand side. Generalising this to monads, this means that TX can be decomposed into the variables X and non-variable terms T_0X, i.e. $TX = X + T_0X$ for some T_0. More succinctly, this decomposition can be written as an equation on functors, i.e. $T = 1 + T_0$.

Definition 5 (Non-Collapsing Monads). *A monad $\mathsf{T} = \langle T, \eta, \mu \rangle$ is non-collapsing iff $T = 1 + T_0$, with the unit the inclusion $in_1 : 1 \Rightarrow T$ and the other inclusion written $\alpha_T : T_0 \Rightarrow T$. In addition, there is a natural transformation $\mu_0 : T_0T \Rightarrow T_0$ such that $\alpha \cdot \mu_0 = \mu \cdot \alpha_T$.*

Given a signature Σ, the term monad T_Σ is non-collapsing, since every term is either a variable or an operation (applied to subterms). More generally, given an algebraic theory $\langle \Sigma, E \rangle$, the representing monad $\mathsf{T}_{\langle \Sigma, E \rangle}$ is non-collapsing iff neither the left or right hand sides of any equation is a variable.

Lemma 1. *In any category, the pushout of the inclusions $in_1 : 1 \Rightarrow 1 + X$ and $in_1 : 1 \Rightarrow 1 + Y$ is $1 + X + Y$. Given two non-collapsing monads $1 + T_0$ and $1 + R_0$, their pointed coproduct is (Q, q) with $Q = 1 + T_0 + R_0$ and $q : 1 \Rightarrow Q$.*

Proof. The first part can be proved by a simple diagram chase. The second part follows since in a non-collapsing monad the units are the inclusions. □

3.3 Layer Structure 1: Alternating Layers

Our first axiomatisation of layer structure is based upon the idea that, in the coproduct monad, layers alternate between T-layers and R-layers. Since we can decompose the layers of the non-collapsing monads into the variables and the terms, we share the variables between the two monads and only build new non-variable layers on top of other non-variable layers.

Definition 6 (Alternating Layers). *Define the series of functors $A_{T,n}$ and $A_{R,n}$ as follows:*

$$A_{T,0} = 1 \quad A_{T,n+1} = 1 + T_0 A_{R,n} \quad A_{R,0} = 1 \quad A_{R,n+1} = 1 + R_0 A_{T,n}$$

Define natural transformations $a_{T,n} : A_{T,n} \to A_{T,n+1}$ and $a_{R,n} : A_{R,n} \to A_{R,n+1}$

$$a_{T,0} = in_1 \quad a_{T,n+1} = 1 + T_0 a_{R,n} \quad a_{R,0} = in_1 \quad a_{R,n+1} = 1 + R_0 a_{T,n}$$

Finally, define A_R and A_T as the colimits of the chains:

$$A_R = \overset{colim}{\underset{n<\omega}{}} A_{R,n} \quad A_T = \overset{colim}{\underset{n<\omega}{}} A_{T,n}$$

We then have $e_T : 1 \Rightarrow A_T$ and $e_R : 1 \Rightarrow A_R$ defined by the inclusion of $A_{T,0}$ into A_T and A as the pushout:

$$\begin{array}{ccc} 1 & \xrightarrow{e_T} & A_T \\ {\scriptstyle e_R}\downarrow & & \downarrow \\ A_R & \longrightarrow & A \end{array} \tag{3}$$

The functor $A_{T,n}$ can be thought of as alternating, non-variable layers of depth at most n, starting with a T-layer. Thus $A_{T,n+1} = 1 + T_0 A_{R,n}$ says that an alternating, top T-layer term of depth at most $n+1$ is either a variable or contains a non-variable layer T_0 on top of an alternating, top R-layer term of depth at most n. A_R and A_T are alternating layers of arbitrary depth, starting with a R and T-layer, respectively. A contains all alternating layers, starting with either R or T, with the variables shared as we saw in the pointed coproduct construction. That A is (isomorphic to) the coproduct is shown in Sect. A.1.

3.4 Layer Construction 2: Quotiented Layers

In certain situations, the alternating layers will not be appropriate as, for example, one may not want to have to explicitly enforce the alternating criteria on layers. An alternative construction starts with the pointed coproduct of monads $1 + T_0$ and $1 + R_0$ given by $Q = 1 + T_0 + R_0$ with $q : 1 \Rightarrow Q$ given by Lemma 1.

Definition 7 (Q-layers). *With (Q, q) given by Lemma 1, let $Q^* = \operatorname*{colim}_{n<\omega} Q^n$ be the colimit of the ω-chain Q^n with maps $q_n : Q^n \Rightarrow Q^{n+1}$ given by $q_0 = q$ and $q_{n+1} = Q^n q$.*

Of course, Q^* is not the coproduct; it has η-quotienting built in, but no μ-quotienting. For this, we define a map $v^* : Q^*X \to (\mathsf{T}+\mathsf{R})X$ which tells us when two elements of Q^* represent the same element of the coproduct monad, and then we construct normal forms for this equivalence relation. Technically, the map v^* is defined by a family of maps $v_n : Q^n X \to (\mathsf{T}+\mathsf{R})X$ which commute with the q^n (i.e. $v_n \cdot q_n = v_{n-1}$); such a family is called a *cone*. The precise definition of v_n and the quotienting, along with a proof of correctness, can be found in Sect. A.2.

3.5 Layer Structure 3: Non-alternating Layers

A third axiomatisation of layers follows from the observation that every term in the coproduct is either a variable, a T_0-layer over sublayers or an R_0-layer over sublayers. Thus one defines

$$L_0 = 1 \quad L_{n+1} = 1 + T_0 L_n + R_0 L_n$$

As the arguments are similar to those for the quotiented layers, and with space considerations in mind, we only sketch the details. We define $L^* = \operatorname*{colim}_{n<\omega} L_n$, and uses the inclusions of T_0 and R_0 into $\mathsf{T}+\mathsf{R}$ to define a natural transformation $w^* : L^* \Rightarrow \mathsf{T}+\mathsf{R}$ which indicates when two layered term represent the same term in the coproduct monad.

We then define a right inverse for w^* by embedding the alternating layers monad into L^*, which allows us to conclude that the quotient of L^* by the kernel of w^* defines the coproduct monad. The right inverse can be used to construct representatives for each equivalence class of the kernel.

3.6 Collapsing Monads

We have given a number three constructions of the coproduct on non-collapsing monads, each with a different layer structure. Kelly [7, Sect. 27] has shown the construction of colimits of monads, from which we can deduce coproducts of arbitrary monads as a special case. The coproduct is constructed pointwise; given two monads $\mathsf{T} = \langle T, \eta_T, \mu_T \rangle$ and $\mathsf{R} = \langle R, \eta_R, \mu_R \rangle$, the coproduct monad $\mathsf{T} + \mathsf{R}$ maps every object X to the colimit of sequence X_β defined as follows:

$$T + R(X) = \operatorname*{colim}_{\beta<\omega} X_\beta \qquad X_0 = X \qquad X_1 = QX \qquad X_{\beta+1} = colim(D_\beta)$$

where Q, σ_T, σ_R are given by Def. 4, and D_β by the diagram in Fig. 1 with the colimiting morphism $x_\beta : D_\beta \to X_{\beta+1}$ which given the shape of the diagram is a single morphism $x_\beta : QX_\beta \to X_{\beta+1}$ making all arrows in the diagram commute. In principle, D_β defines another layer structure for terms in the coproduct monad but in practice the shape, size and contents of this diagram makes it difficult to reason with directly.

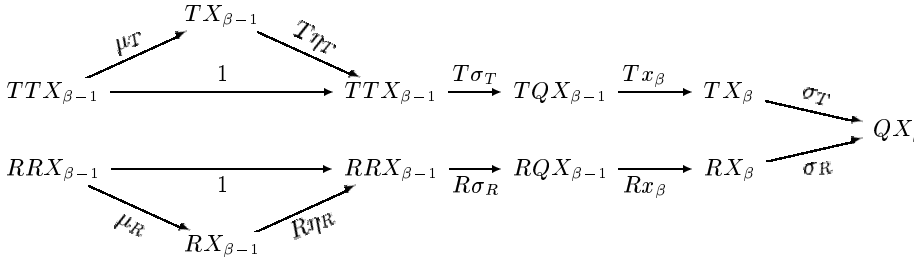

Fig. 1. The diagram defining the coproduct of two monads.

An alternative is to return to the quotiented layers. We can still define a chain Q_n as in Def. 7, together with a map $k^* : Q^n X \to (\mathsf{T} + \mathsf{R})X$; unfortunately, in the absence of the non-collapsing assumption we cannot go further and provide canonical representatives of each equivalence class.

A final, very simple special case is when there is a natural transformation $m : QQ \Rightarrow Q$ which commutes with μ_T and μ_R

$$\sigma_T \cdot \mu_T = m \cdot (\sigma_R * \sigma_R) \qquad \sigma_R \cdot \mu_E = m \cdot (\sigma_R * \sigma_R) \qquad \sigma \cdot m = 1_{QX} \tag{4}$$

then Q is the coproduct, and m its multiplication.

4 Applications I: Modular Rewriting

In this section, we sketch the application of our analysis to modular term rewriting. The results in this section have been presented elsewhere before [10], but the alternating layer presentation from Sect. 3.3 further simplifies our arguments.

The prerequisite of monadic rewriting is the representation of a TRS as a monad. The action of this monad is given by the term reduction algebra:

Definition 8 (Term Reduction Algebra). *For a term rewriting system $\mathcal{R} = \langle \Sigma, R \rangle$, the* term reduction algebra *$T_{\mathcal{R}}(X)$ built over a* preorder *X has as underlying set the term algebra $T_\Sigma(X)$ and as order the least preorder including all instantiations of rules $r \in R$ and the order on X such that all operations $f \in \Sigma$ are monotone.*

The mapping of X to $T_{\mathcal{R}}(X)$ gives rise to a functor $T_{\mathcal{R}} : \mathbf{Pre} \to \mathbf{Pre}$ on the category of preorders. To make this into a monad, we add unit and multiplication as in the case of signatures (see Sect. 2), except that we further have to show that they are monotone. Since the monadic semantics is *compositional*, i.e. $\mathsf{T}_{\mathcal{R}+\mathcal{S}} \cong \mathsf{T}_{\mathcal{R}} + \mathsf{T}_{\mathcal{S}}$, we can prove properties about the disjoint union of TRSs by proving them for the coproduct monad. Of course we also have to translate properties P of a TRS into an equivalent property P' of monads. The obvious way is to require that the action of the monad preserves P.

Definition 9 (Monadic SN). *A monad $\mathsf{T} = \langle T, \eta, \mu \rangle$ on **Pre** is strongly normalising iff whenever the irreflexive part of X is strongly normalising, then so is the irreflexive part of TX.*

To show that this definition makes sense, we show that a TRS $\mathcal{R}$ is SN iff its representing monad $\mathsf{T}_\mathcal{R}$ is SN in the sense of Def. 9; see [10, Prop. 5.1.5]. We can now prove modularity of strong normalisation for non-collapsing TRS [15]. The main lemma will be that the coproduct of two non-collapsing monads $\mathsf{T}+\mathsf{R}$ is SN if T and R are. For this, recall the alternating layers A from Def. 6.

Lemma 2 (Modularity of SN for non-collapsing monads). *Let T, R be non-collapsing, strongly normalising monads, then the monad $\mathsf{T}+\mathsf{R}$ is SN.*

Proof. We use the fact that $T + R(X) = AX$, and show that AX is SN. We first show that A_R and A_T are SN, i.e. that if X is SN so is $A_R X$. Since $A_R = \overset{colim}{n<\omega} A_{R,n}$ this is done by induction over n: the base case is the assumption; for the inductive step, $A_{R,n}X$ is SN by the induction hypothesis, $T_0 A_{R,n}$ is SN since T_0 preserves SN preorders and hence $A_{T,n+1}X = X + T_0 A_{R,n}$ is SN since the disjoint union of SN preorders is SN. Now, $AX \cong TA_R X$, and since A_R is SN, and by assumption T is SN, so is AX. □

Proposition 1 (Modularity of Strong Normalisation). *Strong normalisation is modular for non-collapsing term rewriting systems.*

Proof. Given two strongly normalising TRSs $\mathcal{R}$ and $\mathcal{S}$, then $\mathsf{T}_\mathcal{R}$ and $\mathsf{T}_\mathcal{S}$ are SN. By Lemma 2 it follows that AX is SN whenever X is, and since $AX \cong (\mathsf{T}_\mathcal{R} + \mathsf{T}_\mathcal{S})X$, $\mathsf{T}+\mathsf{R}$ is SN. By compositionality, this means that $\mathsf{T}_{\mathcal{R}+\mathcal{S}}$ is SN, and hence $\mathcal{R}+\mathcal{S}$ is SN, as required. □

The advantage of using monads here is that the main lemma does not talk about term rewriting systems anymore, but about monads. Thus, the theorem applies to any structure which can be modelled by a monad as well, for example if we allow equations as well as rewrite rules.

5 Applications II: Computational Monads

Computational monads [13] provide a categorical framework for expressing computational features independent of the specific computational model we have in mind. The base category provides a basic model of computation, and the computational monad builds additional features, such as exceptions, state and non-determinism:

Example 5 (Exceptions). Let E be an object of $\mathcal{C}$, which are the exceptions. The *exception monad* is given as

$$Ex_E(X) = E + X \qquad \eta_E = in_2 \qquad \mu_E = [in_1, 1]$$

As a second example, consider a monad adding state dependency. In an abstract view, state is just an object $S \in \mathcal{C}$ of our base category:

Example 6 (The State Transformer Monad). Let S be an object of $\mathcal{C}$. The monad $\mathsf{St}_S = \langle St_S, \eta_S, \mu_S \rangle$ is defined as

$$St_S(X) = S \to S \times X \qquad \eta_{S,X}(x) = \lambda s.x \qquad \mu_{S,X}(c) = \lambda s.let\ \langle f, x \rangle = c s\ in\ f\, x$$

A stateful computation maps a state to a successor state and a value. The multiplication composes two stateful computations by inserting the successor state of the first computation as input into the second. The overall result is the result of the second computation.

Finally, if we have a finite powerset functor $\mathbb{P}_{fin} : \mathcal{C} \to \mathcal{C}$ in our base category (e.g. if $\mathcal{C} = \mathbf{Set}$), then we can incorporate non-determinism to our set of models:

Example 7 (The non-determinism monad). The monad $\mathsf{P} = \langle \mathbb{P}_{fin}, \eta_P, \mu_P \rangle$ has the finite powerset functor as its action, with the unit and multiplication defined as follows:

$$\eta_P(X) = \{X\} \qquad \mu_P(X) = \cup_{X_0 \in X} X_0$$

The exception monad does not build any new layers, it only adds constants. Thus, exceptions only ever occur in the lowest layer:

Lemma 3. *The coproduct of Ex_E with any monad $\mathsf{R} = \langle R, \eta_R, \mu_R \rangle$ is given by*

$$SX = R(E + X)$$

The proof of Lemma+3 can be found in the appendix (Sect. A.3). Lemma 3 allows us to combine exceptions with statefulness, leading to $Ex_E + St_S = (S \to S \times (X + E))$, and with non-determinism, resulting in $\mathbb{P}_{fin} + Ex_E = \mathbb{P}_{fin}(X + E)$. Further, we can combine non-determinism with stateful computations. One might think that the action of the combination would be $Q(X) = S \to (S \times \mathbb{P}_{fin} X)$, but this is slightly wrong, since it does not allow non-deterministic computations not depending on the state S; the correct action is given by first calculating $Q(X) = (S \to S \times X) + \mathbb{P}_{fin} X / \sim$ where $\lambda s.\langle s, X \rangle \sim \{X\}$. In other words, every computation is either stateful or non-deterministic. In the coproduct, this is closed under composition, i.e. the computations are interleaved sequences of stateful and non-deterministic computations (appropriately quotiented).

The combination of computational monads has been investigated before, but mainly for special cases [6,8]. Monad transformers have been suggested as a means to combine monads [14,9], but they serve as an organisational tool rather than a general semantic construction like the coproduct described here.

6 Applications III: Haskell

Monads are used extensively in Haskell, which provides the built-in monad `IO a` as well as user-defined ones. The Haskell types are our objects, and the terms our morphisms. Thus, a type constructor t forms a *functor* if for any function `f::a->`

`b` there is a function `fmap :: t a-> t b`, and it is a *monad* if additionally there are functions `eta :: a -> t a` and `mu :: t (t a) -> t a` for all types `a`. These overloaded functions are handled by type classes. Building upon the class `Functor` from Haskell's standard prelude [1, Appx. A], we define

```
class Functor t=> Triple t where
   eta  :: a-> t a
   mu   :: t (t a) -> t a
```

The monad laws cannot be expressed within Haskell, so the tacit assumption is that they are verified externally. In the implementation of the coproduct, we face the difficulty that Haskell does not allow us to write down equations on types. Hence, we representing the equivalence classes, making `eta` and `mu` operate directly on the representatives, which are modelled as follows:

```
data Plus t1 t2 a = T1 (t1 (Plus t1 t2 a))  -- top-T1 layer
                  | T2 (t2 (Plus t1 t2 a))  -- top-T2 layer
                  | Var a                   -- a mere variable
```

Note how this datatype corresponds to the functor L^* from Sect. 3.5. We have to make the type constructor `Plus t1 t2` into a monad by first making it into a functor, and then giving the unit and multiplication. For this, we implement the decision procedure mentioned in Sect. 3.5. Essentially, we recursively collapse adjacent layers wherever possible:

```
instance (Triple t1, Triple t2)=> Triple (Plus t1 t2) where
  eta x      = Var x
  mu (Var t) = t
  mu (T1 t)  = T1 (mu (fmap lift1 (fmap mu t))) where
    lift1 :: Plus t1 t2 x-> t1 (Plus t1 t2 x)
    lift1 (T1 t) = t
    lift1 t      = eta t
  mu (T2 t)  = ... -- analogous to mu (T1 t)
```

Finally, we need the two injections into the coproduct; we only show one:

```
inl :: Triple t1=> t1 a-> Plus t1 t2 a
inl t = T1 (fmap Var t)
```

We can now implement exceptions, state and so on as monads, and compose their computations in the coproduct. This is different from using the built-in `IO` monad, since the type of an expression will be contain precisely those monads the computational features of which are used in this expression.

Technically, monads as implemented by Haskell (in particular the monad `IO`) define a monad by its *Kleisli-category* [12, Sect. VI.5], hence our own type class `Triple` above, rather than the standard `Monad`. One can easily adapt our construction to cover this case. Note that the code above requires extensions to the Haskell 98 standard, such as multi-parameter type classes.

7 Conclusion

The basic hypothesis of this paper has been that when we combine different formal systems, be they algebraic theories, models of computations, or programs, their interaction can be described in terms of their *layers*. We have given an abstract characterisation of layers by monads, and shown how the combination of different monads can be modelled by the coproduct.

We have complimented the general construction of the colimit of monads with alternative, specialised constructions. In particular, we have constructed the coproduct monad based on alternating layers, quotiented layers and non-alternating layers, and employed these constructions for modular term rewriting, modular denotational semantics, and modular functional programming.

Obviously, this work is just the beginning. What needs to be done is to extend our construction to cover e.g. non-collapsing monads, to augment the applications (in particular with regards to Haskell) and to investigate in how far other applications can be covered by our methodology.

References

1. Haskell 98: A non-strict, purely functional language. Available at `http://www.haskell.org`, January 1999.
2. J. Adamek and J. Rosický. *Locally Presentable and Accessible Categories*. LMS Lecture Notes 189, Cambridge University Press, 1994.
3. E. J. Dubuc and G. M. Kelly. A presentation of topoi as algebraic relative to categories or graphs. *Journal for Algebra*, 81:420–433, 1983.
4. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. LICS'99*, pages 193– 202. IEEE Computer Society Press, 1999.
5. N. Ghani, C. Lüth, F. de Marchi, and J. Power. Algebras, coalgebras, monads and comonads. Proc. CMCS'01, *ENTCS* 44:1, 2001.
6. M. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, Dept. Comp. Sci, Dec 1993.
7. G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bulletins of the Australian Mathematical Society*, 22:1– 83, 1980.
8. D. King and P. Wadler. Combining monads. In J. Launchbury and P.M. Samson, editors, *Functional Programming*, Workshops in Computing, 1993.
9. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of the 22nd ACM Symposium on Principles of Programming Languages*. ACM Press, Jan 1995.
10. C. Lüth. *Categorical Term Rewriting: Monads and Modularity*. PhD thesis, University of Edinburgh, 1998.
11. C. Lüth and N. Ghani. Monads and modular term rewriting. In *CTCS'97*, LNAI 1290, pages 69– 86. Springer, Sep 1997.
12. S. Mac Lane. *Categories for the Working Mathematician*, *Graduate Texts in Mathematics* 5. Springer, 1971.
13. E. Moggi. Computational lambda-calculus and monads. In *Proc. LICS'89*. IEEE, Computer Society Press, June 1989.

14. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, 1990.
15. M. Rusinowitch. On the termination of the direct sum of term-rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.
16. D. T. Sannella and R. M. Burstall. Structured theories in LCF. In 8^{th} *Colloquium on Trees in Algebra and Programming*, LNAI 159, pages 377– 391. Springer, 1983.
17. D. T. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, Feb/Mar 1988.

A Correctness Proofs

As blanket assumptions, we assume that the base category is locally finitely presentable (lfp) [2], which in particular means it has all colimits, and that the monads in question are finitary.

If we claim that a construction defines the coproduct monad, how can we to prove this? The answer is that just as one can understand an algebraic theory through its models, one can understand a monad through its *algebras*:

Definition 10 (Algebras for a monad). *An* algebra (X, h) *for a monad* $\mathsf{T} = \langle T, \eta, \mu \rangle$ *on a category* $\mathcal{C}$ *is given by an object* X *in* $\mathcal{C}$*, and a morphism* $h : TX \to X$ *which commutes with the unit and multiplication of the monad, i.e.* $\eta_X = h \cdot \eta_{TX}$ *and* $h \cdot \mu_X = h \cdot Th$.

The category of algebras for T *and morphisms between them is called* $\mathsf{T}-\textbf{\textit{Alg}}$.

We think of a T-algebra (X, h) as being a model with carrier X. The map h ensures that if one builds terms over a such a model, then these terms can be reinterpreted within the model. This is exactly what one is doing in the term algebra case where one assigns to every function symbol f of arity n an interpretation $[\![f]\!] : X^n \to X$. Since monads construct free algebras, we can prove a functor to be equal to a monad if we can prove that the functor constructs free algebras. In particular, we can prove a functor to be the coproduct monad if we can prove it constructs free $\mathsf{T} + \mathsf{R}$-algebras which are defined as follows:

Definition 11 (T+R-algebras). *The category* T+R-$\textbf{\textit{Alg}}$ *has as objects triples* (A, h_t, h_r) *where* (A, h_t) *is a* T*-algebra and* (A, h_r) *is an* R*-algebra. A morphism from* (A, h_t, h_r) *to* (A', h'_t, h'_r) *consists of a map* $f : A \to A'$ *which commutes with the* T *and* R*-algebra structures on* A *and* A'.

There is an obvious forgetful functor $U : \mathsf{T}+\mathsf{R}\text{-}\mathbf{Alg} \to \mathcal{C}$, which takes a $\mathsf{T} + \mathsf{R}$-algebra to its underlying object, and we have the following:

Proposition 2 ([7, Propn. 26.4]). *If the forgetful functor* $U : \mathsf{T}+\mathsf{R}\text{-}\textbf{\textit{Alg}} \to \mathcal{C}$ *functor has a left adjoint* $F : \mathcal{C} \to \mathsf{T}+\mathsf{R}\text{-}\textbf{\textit{Alg}}$*, i.e. if for every object in* $\mathcal{C}$ *there is a* free $\mathsf{T} + \mathsf{R}$*-algebra, then the monad resulting from this adjunction is the coproduct of* T *and* R.

Thus to show that a functor S is the coproduct $\mathsf{T} + \mathsf{R}$, we can show that for every object X, SX is a $\mathsf{T}+\mathsf{R}$-algebra and, moreover, it is the free $\mathsf{T}+\mathsf{R}$-algebra.

A.1 Correctness of the Alternating Layer Construction

Lemma 4. *We have the following isomorphisms:*

$$A_T \cong 1 + T_0 A_R, \qquad A_R \cong 1 + R_0 A_T \tag{5}$$

$$A \cong T A_R, \qquad A \cong R A_T \tag{6}$$

Proof. Isomorphism (5) is shown as follows:

$$\begin{aligned} A_T = \overset{colim}{n<\omega}\ A_{T,n} &= \overset{colim}{n<\omega}\ 1 + T_0 A_{R,n-1} \\ \cong 1 + \overset{colim}{n<\omega}\ T_0 A_{R,n-1} &\cong 1 + T_0 \overset{colim}{n<\omega}\ A_{R,n-1} = 1 + T_0 A_R Q Q \end{aligned}$$

Here, we use interchange of colimits (in the second line) and the fact that T_0 is finitary (which means it preserves colimits of chains).

Since $TA_R = A_R + T_0 A_R$, isomorphism (6) is proven by showing that $A_R + T_0 A_R$ is the pushout of diagram (3). The morphism $A_R \to A_R + T_0 A_R$ is given by the left inclusion; the morphism $m : A_T \to A_R + T_0 A_R$ is given by $e_R + 1$, since by (5) $A_T = 1 + T_0 A_R$. The diagram commutes since e_T is the inclusion $1 \to A_T = 1 + T_0 A_R$. Given any other X and morphisms $g : A_T \to X$, $f : A_R \to X$, we have a unique morphism $!_{f,g} : A_R + T_0 A_R$ given by $!_{f,g} = [f, g_2]$, where g_2 is g on $T_0 A_R$. □

We will now prove that A is the coproduct $\mathsf{T} + \mathsf{R}$, using Proposition 2, by showing that A is the free $T + R$-algebra.

Lemma 5 (A is a $\mathsf{T} + \mathsf{R}$-algebra). *For any $X \in \mathcal{C}$, AX is a $T + R$-algebra.*

Proof. We have to show that there are morphism $h_t^A : TAX \to AX$ and $h_r^A : RAX \to AX$ which satisfy the equations from Def. 10. By (6), we may define h_t^A as

$$TAX \quad = \quad TTA_R X \xrightarrow{\mu_{T,A_R X}} TA_R X \quad = \quad AX$$

and similarly, set $h_r^A = \mu_R$. That h_t^A and h_r^A commute with the unit and multiplication of T and R respectively is easy. □

Lemma 6 (A is the free $\mathsf{T} + \mathsf{R}$-algebra). *The functor $\mathcal{A} : \mathcal{C} \to T{+}R\text{-}\mathbf{Alg}$, mapping X to (AX, h_t^A, h_r^A), is a left adjoint to $U : T{+}R\text{-}\mathbf{Alg} \to \mathcal{C}$.*

Proof. We prove that $e_{A,X} : X \to AX$ is universal to U. That is, given any other $T + R$-algebra (Y, h_t, h_r) and map $f : X \to Y$ in $\mathcal{C}$, there is a unique $T + R$ algebra morphism $f^* : (AX, h_t^A, h_r^A) \to (Y, h_t, h_r)$ such that $U(f^*) \cdot e_{A,X} = f$.

To define a map $f^* : AX \to Y$, it suffices to define maps $f_T : A_T X \to Y$ and $f_R : A_R X \to Y$ such that $f_R \cdot e_R = f_T \cdot e_T$. With $A_T X$ the colimit of $A_{T,n}$, this means f_T is given by a cone $f_{T,n} : A_{T,n} \to Y$ for $n < \omega$. Setting $f_{T,0} = f$ and, with $A_{T,n+1} X = X + T_0 A_{R,n}$, we set $f_{T,n+1} = [f, h_{t,0} \cdot T_0 f_{R,n}]$ where $h_{t,0}$ is the restriction of h to $T_0 Y$. That f_T and f_R are cones is proven by induction while they both clearly equal to f when restricted along e_T and e_R. Thus f^* is well defined. That f^* is an algebra morphism is a routine inductive argument using the fact that h_t and h_r commute with μ_T and μ_R. Finally the equation $f^* \cdot e_{A,X} = f$ has already been commented upon while uniqueness of f^* follows by the uniqueness property of mediating morphisms out of the pushout. □

A.2 Correctness Proofs for the Quotiented Layers

First note we can define a map $v : QX \to (\mathsf{T} + \mathsf{R})X$ by sending TX to its inclusion in $(\mathsf{T}+\mathsf{R})X$ and similarly for RX. Now to define $v_n : Q^n X \to (\mathsf{T}+\mathsf{R})X$ by $v_n = \mu^n_{T+R} \cdot v^n$ where μ^n_{T+R} is the n-fold iteration of multiplication in the coproduct and v^n is the n-fold iteration of v. This clearly forms a cone and hence gives a map $v^* : Q^* X \to (\mathsf{T} + \mathsf{R})X$.

We claim that the quotient of Q^* by the kernel of v^* is the coproduct $\mathsf{T} + \mathsf{R}$. We have already mapped Q^* into $\mathsf{T} + \mathsf{R}$ via v^*. Now we embedd $\mathsf{T} + \mathsf{R}$ in Q^* via A which we have already seen to be $\mathsf{T} + \mathsf{R}$. This embedding effectively decides the kernel of v^*.

First we define maps $s_{T,n} : A_{T,n} \Rightarrow Q^n$ and $s_{R,n} : A_{R,n} \Rightarrow Q^n$ by setting $s_{T,0} = s_{R,0} = 1$ and $s_{T,n+1}$ by

$$A_{T,n+1} \quad = 1 + T_0 A_{R,n} \xrightarrow{[in_1 \cdot q^n,\, in_2 \cdot T_0(s_{R,n})]} Q^n + T_0 Q^n + R_0 Q^n = \quad Q^{n+1}$$

That these maps form cones is easily verified and hence we get a map $s : A \Rightarrow Q^*$. By unwinding these definitions, we obtain $v^* \cdot s = 1$.

Lemma 7. *The quotient of Q^* by the kernel of v^* defines the coproduct monad. Each equivalence class of this quotient has a canonical representative.*

Proof. The existence of s means that v^* is a split epimorphism and hence the quotient of Q^* by the kernel of v^* is the codomain of v^* and hence is $\mathsf{T} + \mathsf{R}$. If $t \in Q^*$, then we define its representative to be $s(v^*(t))$. Thus t is related to u in the kernel of v^* iff $v^*(t) = v^*(u)$ which implies that $s(v^*(t)) = s(v^*(u))$, ie they have the same representative. □

A.3 Proof of Lemma 3

By Prop. 2, it is sufficient to show that SX is the free $Ex_E + \mathsf{R}$-algebra.

Showing that SX is an R-algebra is simple, with the structure map given by μ_R. The structure map $[1, \alpha] : ESX \to SX$ making SX into an Ex_E-algebra is given by $\alpha = Rin_1 \cdot \eta_R$.

The unit of the adjunction is given by $\eta_S = Rin_2 \cdot \eta_R$. To show it is universality from U, assume there is a $R + E$-algebra Y with $\delta : E \to Y$ and $\beta : RY \to Y$, and a morphism $f : X \to Y$. Then we define $!_f : R(E + X) \to Y$, defined as $!_f = \beta \cdot R[\delta, f]$. A simple diagram chase shows that $!_f \cdot \eta_X = f$.

□