Composing Monads Using Coproducts

Christoph Lüth
FB 3 — Mathematik und Informatik
Universität Bremen
cxl@informatik.uni-bremen.de

Neil Ghani
Dept. Mathematics and Computer Science
University of Leicester
ng13@mcs.le.ac.uk

Abstract

Monads are a useful abstraction of computation, as they model diverse computational effects such as stateful computations, exceptions and I/O in a uniform manner. Their potential to provide both a modular semantics and a modular programming style was soon recognised. However, in general, monads proved difficult to compose and so research focused on special mechanisms for their composition such as distributive monads and monad transformers.

We present a new approach to this problem which is general in that nearly all monads compose, mathematically elegant in using the standard categorical tools underpinning monads and computationally expressive in supporting a canonical recursion operator. In a nutshell, we propose that two monads should be composed by taking their *coproduct*. Although abstractly this is a simple idea, the actual construction of the coproduct of two monads is non-trivial. We outline this construction, show how to implement the coproduct within Haskell and demonstrate its usage with a few examples. We also discuss its relationship with other ways of combining monads, in particular distributive laws for monads and monad transformers.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Functional Programming

General Terms

Algorithms, Languages, Theory

1 Introduction

It has long been a goal of research within the theoretical computer science community to provide a modular semantics for programming languages. In detail, one would like to give a semantics for individual features of a programming language such as *exception handling*, *non-determinism* and *state-based computation* and, by suitably composing these, obtain a semantics for the programming language as a whole. If successful, reasoning about large pieces of software could be achieved by reasoning about smaller (and hence more tractable) subcomponents and then lifting these results to the original problem.

One exciting possibility was Moggi's proposal [17] to use monads to structure denotational semantics. Moggi pointed out that a number of computational features (including all of those mentioned above) could be modelled by monads. Wadler took Moggi's ideas further by showing how one could actually program with monads. For example monads can be used to support imperative features within a purely functional language. Soon, however, it became clear that despite the undoubted value of monads from both the semantic and programming perspectives, composing monads would prove to be a significant challenge. Put briefly, monads just do not seem to compose in any general manner. Thus, a variety of different methods to combine monads were proposed, most notably in the use of *distributivity laws* and *monad transformers*. While being useful in specific situations, these theories do not cover all situations, and furthermore, as we comment later, can sometimes seem rather adhoc.

What these approaches have in common is the observation that monads are functors carrying extra structure. Thus the obvious way to compose a monad T and a monad R is to use the functorial composition $R{\cdot}T$. Unfortunately, $R{\cdot}T$ is not a monad and one can see both monad transformers and distributivity laws as an attempt to coax $R{\cdot}T$ into a monad. An alternative point of view is to observe that, just as functors are the objects of the functor category, so monads are the objects of the category of monads and monad morphisms. Then the canonical way of putting two objects together is take their coproduct, and this is what this paper is about.

Our interest in this topic arose from previous research of ours which used the coproduct of two monads as a framework for modularity [14] and, in particular, modular term rewriting [12, 13]. In modular term rewriting, terms built over two signatures are decomposed into terms built over each of the two signatures, called *layers*. Within these papers, we argued that the layer structure is the key concept of modularity, and should be taken as a primitive notion rather than as a derived concept. The relevance of monads is that they provided an abstract and axiomatic formulation of the notion of a layer. When we tried to apply this work to the combination of monads within functional programming we found that we had used several assumptions which important computational monads did not satisfy. Thus, the composition of computational monads, in particular from the functional programmer's perspective, became a challenge for us, and this paper is the result of our research. Its

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

concrete contributions are:

- to explain the construction of the coproduct of two monads using the idea of layers to describe how the individual monads are interleaved to form the coproduct monad;
- to explain how the coproduct can be regarded as an abstract datatype with a canonical recursion operator similar to `fold`;
- to explain that, unlike current approaches, virtually all monads can be composed via the coproduct construction;
- to explain how the coproduct of two monads can be implemented in Haskell. This is non-trivial since the coproduct is not a free datatype but rather a quotient;
- to explain how the coproduct approach relates to current approaches. For example, in the presence of *strong distributivity* laws, the functorial composite of two monads is not just a monad, but also the coproduct of the two monads in question.

To summarise, we feel that coproducts provide the right general framework for composing monads. The fact that almost all monads compose in this way and that the coproduct has a universal property and an associated recursion operator are two powerful arguments in its favour over current approaches. However, there is a price to be paid, namely that we are keeping track of all possible layers. As we show later, one can regard monad transformers and distributivity laws as an attempt to squash all these layers into one specific layer which, when possible, results in a data structure which is potentially easier to manage. However, even in such situations, the coproduct analysis still contributes to our understanding, e.g. in reminding us that however we compose monads, there should be an associated recursion operator.

Our aim here is to explain our ideas to the functional programming community. Consequently we keep the categorical jargon down to a minimum, focusing on intuitions and examples rather than detailed proofs; however, at times we need to be technically exact (in particular in Sect. 4.2). Then, a basic knowledge of category theory (such as categories, functors and natural transformations) will be helpful.

The remainder of this paper is structured as follows: in Sect. 2 we introduce monads, both as term algebras and computational monads as found in Haskell, and discuss how distributivity laws and monad transformers have been used to compose monads. In Sect. 3, we construct the coproduct of two monads and give some examples of its use. In Sect. 4, we evaluate the construction and in particular show how distributive monads form a special case.

2 A Brief History of Monads

Monads originally arose within category theory as models of term algebras. Such monads provide good intuitions as to how to construct the coproduct of two monads and so we begin with them. Since this is standard material, we refer the reader to general texts [15] for more details.

2.1 Monads and Term Algebras

Term algebras are built from signatures which are defined as follows:

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

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

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

Quotes are used to distinguish a variable $x \in X$ from the term ${}'x \in T_\Sigma(X)$ and can be seen as introducing layer information into terms. As we shall see later, when constructing the coproduct of two monads, this layer structure is the central concept. 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 x to the associated term ${}'x$. 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. 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 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. If 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, such as (many-sorted) algebraic theories, calculi with variable binders [4], term rewriting systems [12], and, via computational monads [17], state-based computations, exceptions, continuations etc. Several of these computational monads are used throughout this paper and their definition is given in Appendix A. Importantly, these applications involve base categories other than **Set** (in fact, every monad on **Set** can be considered as an algebraic theory [16]), and possibly even enrichment. Even in this abstract setting, monads can be considered as a generalised form of algebraic theories [9, 21]. We do not pursue the theory in its full generality here, but should bear in mind that although we draw our intuitions from monads on **Set**, this is just a particular case which we do well not to restrict ourselves to.

In the rest of this paper, we often regard $T(X)$ abstractly as a *layer* — in the examples above, layers were terms, rewrites, or computations; monads then abstract from the nature of a layer and rather provide a 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. The coproduct construction we detail later will then consist of all interleavings of layers from the component monads.

We end this section with two minor technical comments about *finitariness* and *enrichment*; while not important for the basic understanding, they are essential for the mathematical correctness.

Firstly, a monad is *finitary* if its action on infinite objects is determined by its action on finite objects. For example, for a finitary signature Σ and an infinite set X of variables, we have

$$T_\Sigma(X) = \bigcup_{X_0 \subseteq X, X_0 \text{ finite}} T_\Sigma(X_0)$$

Categorically, this property is defined in terms of preservation of directed colimits, but we shall not need this level of detail; however, some of our constructions will only be valid for finitary monads, so we introduce the terminology on an informal level here. For the precise definitions, see [1]. All known computational monads are finitary, except for the continuation monad [19].

Secondly, when reasoning about the semantics of functional languages, we usually pass from sets to order structures such as complete partial orders (cpos). Categorically, this means not only a change of the base category, but imposing an order structure on the morphisms as well (see e.g. [23]; crucially the set of functions between two cpo's forms in turn a cpo.) This is called *enrichment* [8]. We note that all of the theory in the following can be enriched [9], but prefer to keep the presentation simple.

The definition of a monad given above is unlike what readers may have seen in a functional language like Haskell. Here is how the two are related.

2.2 Monads in Haskell

In the programming language Haskell, a monad is given by two operations `>>=` and `return`, which form a *type class*:

```
class Monad m where
  (>>=)  :: m a-> (a-> m b)-> m b
  return :: a  -> m a
```

This is the so-called *Kleisli-category* of a monad:

Definition 4. For a monad $\mathsf{T} = \langle T, \eta, \mu \rangle$, the associated *Kleisli-category* $\mathcal{C}_T$ has the same objects as $\mathcal{C}$. Each morphism $f : X \to TY$ in $\mathcal{C}$ gives a morphism $f^\dagger : X \to Y$ in $\mathcal{C}_T$, with composition is given $g^\dagger \cdot f^\dagger = (\mu \cdot Tg \cdot f)^\dagger$ and identity $1_X = \eta_X^\dagger : X \to TX$.

The Kleisli-category for a signature Σ has as objects sets (of variables or terms), and as morphisms *assignments* of the form $\sigma : X \to T_\Sigma(Y)$. Composition is by variable substitution, i.e. given $\tau : Y \to T_\Sigma(Z)$, we have $\tau \cdot \sigma : X \to T_\Sigma(Z)$ as $\tau \cdot \sigma(x) \stackrel{def}{=} t[\tau(y)/y]$ for $\sigma(x) = t$.

The Kleisli-category can be considered the free syntactic category generated from the signature, and as such it has some attractive properties (for example, unification can be expressed elegantly as a coequaliser [22]). However, we prefer to work with Def. 3 since it is unclear how to form the coproduct of two Kleisli-categories directly. It is most certainly not the coproduct of the two categories, which would just be their disjoint union. In contrast, as we have already seen, the presentation of monads in Def. 3 supports the intuition based on layers which turns out to be the central concept in constructing the coproduct of two monads.

Fortunately, Def. 3 and Def. 4 are equivalent in the sense that given one we can always calculate the other in a bijective manner. For example, given a monad T, one constructs functors $F_T : \mathcal{C} \to \mathcal{C}_T, G_T : \mathcal{C}_T \to \mathcal{C}$ which on objects are respectively the identity and T, while on morphisms:

$$\begin{aligned} F_T(f : X \to Y) &= (\eta_Y \cdot f)^\dagger X \to TY \\ G_T(f : X \to TY) &= (\mu_Y \cdot Tf) : TX \to TY \end{aligned}$$

The functor F_T is left adjoint to G_T, and their composition results in a monad which is isomorphic to T [15, Theorem VI.5.1]. We can directly express this in Haskell as follows:[1]

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

instance Triple f=> Monad f where
  x >>= y = (mu . (fmap y)) x
  return  = eta

instance (Functor m, Monad m) => Triple m where
  eta = return
  mu x = x >>= id
```

Of course, we could have defined a new class `Monad` or `Triple` which has all of `(>>=)`, `eta` and `mu`, but the advantage of this approach is that we can use Haskell's syntactic sugar when writing down monads which we combine using the `Triple` class, and we can further use Haskell's predefined monads as instance of `Triple`, combining the best of both worlds.

2.3 Composing Monads

Given a monad modelling exceptions and a monad modelling state transformations, can we derive a monad modelling computations which can either raise exceptions or modify the state. More generally, given monads T and R which we may think of as performing T-computations and R-computations respectively, is there a monad which performs both? We finish this section by discussing two approaches to this problem.

Distributivity: A first guess would be to consider if the functorial composite $T \cdot R$ is a monad. This would require a multiplication of the form $T \cdot R \cdot T \cdot R \Rightarrow T \cdot R$ but there is no reason one should exist. One solution [6, 10] is to restrict attention to those monads for which there is a natural transformation $\lambda : R \cdot T \Rightarrow T \cdot R$ and then, given some coherence laws, the multiplication can be constructed as

$$T \cdot R \cdot T \cdot R \xRightarrow{T\lambda_R} T \cdot T \cdot R \cdot R \xRightarrow{\mu_T \mu_R} T \cdot R$$

Such a natural transformation λ is called a *distributive law* [2] and ensures $T \cdot R$ is a monad. A practical example occurs with the exception monad which distributes over any other monad. That is, if E is a fixed object of exceptions and T is a monad, then the assignment mapping X to $T(X+E)$ is also a monad.

From our layer based perspective, the functorial composite $T \cdot R$ corresponds to a T-layer over an R-layer. In general, combining a monad T with a monad R should not include just this specific layering, but all layerings, e.g. $T \cdot R \cdot T$ or $R \cdot T \cdot R$. Thus it is not surprising that in general $T \cdot R$ is not a monad. Now, a distributive law corresponds to an interchange law which permutes T-layers over R-layers and hence allows us to squash an arbitrary layering into the

[1] `Triple` is another word for monad [2]. This code will produce overlapping and undecidable class instances, and will be rejected by `ghc` and `hugs` unless the appropriate options are given.

specific layering $T{\cdot}R$. We later formalise this observation by showing that when there is a *strong distributive law*, not only is $T{\cdot}R$ a monad, but it is also the coproduct monad.

Monad Transformers: Monad transformers provide a partial answer to the question of composing monads when there is no distributivity law present. In a nutshell, a *monad transformer* [18] is a pointed endofunctor on the category $\mathbf{Mon}(\mathcal{C})$ of monads over a fixed base category. That is, a functor $F : \mathbf{Mon}(\mathcal{C}) \to \mathbf{Mon}(\mathcal{C})$ mapping every monad T to a monad $F(\mathsf{T})$, and for every monad T a natural family of monad morphisms $\alpha_{\mathsf{T}} : \mathsf{T} \Rightarrow F\mathsf{T}$. We think of the functor F transforming a monad T to the monad $F(\mathsf{T})$, while α_{T} ensures the monad T is sitting inside $F(\mathsf{T})$. Given that the identity functor is a monad, we may thus regard F as adding to every monad T the monad $F(1)$.

For example, the definition of the exception monad transformer [11] is

```
type ExnT m a = m (Exn a)
```

As we have already seen, there is a distributivity law present and in this case, the action of the monad transformer takes any monad T to TExn. However, the definition of the state monad transformer is

```
type StateT s m a = s -> m (s, a)
```

There is no distributive law here between the monad *m* and the state monad which allows us to permute one layer over the other. Nevertheless, one can see that the monad *m* has been partially permuted past the pairing operation although not past the outer function space. So, monad transformers allow us a more fine-grained control of the combination allowing some form of mixing of the layers between the state monad and an arbitrary monad *m*.

In our opinion, the concept of a monad transformer is rather elegant but the definition is too general to support an adequate meta-theory. By this we mean that given a monad, it is not clear whether it is possible, and if so how, to define an associated monad transformer. For example,there is no monad transformer for the list monad associated to the monadic treatment of non-determinism [11]. Another disadvantage is that we have to pick an order in which to combine the monads, which does make a considerable difference. Moreover, when adding a new monad transformer we have to consider the possible combinations with all existing monad transformers separately, and this number of combinations grows quadratically. Thus, monad transformers are not really modular.

To summarise, both distributivity laws and monad transformers attempt to define a composite monad by squashing all the different layers obtained by interleaving the component monads into one specific layer. As we shall see, the coproduct approach simply keeps all the layers. This shift in emphasis means we buy generality since we can always compose monads. Of course the price to be paid is that the data structure has to manage all layers and this imposes some computational overhead.

3 The Coproduct of Monads

Having discussed the current approaches to composing monads, we now explain our approach based upon coproducts. We first discuss abstractly what the coproduct means and show how the associated universal property provides a recursion operator. We then discuss the construction of the coproduct of two monads before implementing it in Haskell and discussing its correctness briefly.

3.1 Using the Coproduct

Given two monads $\mathsf{T}_1, \mathsf{T}_2$, their *coproduct* can be thought of as the smallest monad containing both monads. Formally, the coproduct of two monads is simply the coproduct in the category of monads. For definitiveness we give a formal definition including the relevant *universal property*:

Definition 5. Given two monads T and R, their *coproduct* is a monad denoted $\mathsf{T}+\mathsf{R}$ such that

- there are monad morphisms $in_1 : \mathsf{T} \to \mathsf{T}+\mathsf{R}$ and $in_2 : \mathsf{R} \to \mathsf{T}+\mathsf{R}$;
- for any other monad S and monad morphisms $\alpha : \mathsf{T} \to \mathsf{S}$, $\beta : \mathsf{R} \to \mathsf{S}$, there is a unique monad morphism $[\alpha,\beta] : \mathsf{T}+\mathsf{R} \to \mathsf{S}$, called the *mediating morphism*, such that

$$[\alpha,\beta]{\cdot}in_1 = \alpha \qquad [\alpha,\beta]{\cdot}in_2 = \beta \tag{1}$$

A monad morphism is just a natural transformation commuting with the unit and multiplication [2, Sect. 3.6]. The universal property allows us to treat the coproduct of two monads as an abstract datatype; so given monads `Triple t1, Triple t2`, we have a type `Plus t1 t2 a` and two injections:

```
inl :: Triple t1=> t1 a-> Plus t1 t2 a
inr :: Triple t2=> t2 a-> Plus t1 t2 a
```

Morever, given any two monad morphism from `t1` and `t2` respectively to another monad `s`, we have a function out of the coproduct. The monad morphisms have the type `t1 a-> s a`, but each has to be defined uniformly for all `a`, hence the type of `coprod` uses rank-2 polymorphism and is not standard Haskell98 anymore:

```
coprod ::(Triple t1, Triple t2, Triple s)=>
         (forall a.t1 a-> s a)->
         (forall a.t2 a-> s a)-> Plus t1 t2 a-> s a
```

Just like `fold` on lists is given by the universal property (i.e. freeness, initiality) of the datatype `[a]`, `coprod` is given by the universal property of the coproduct, and just like `fold`, it can be used to implement recursive functions on the coproduct.

We now consider a small example. Assume that we have an exception monad `Exn` (see Sect. A.2). Using this monad, we can define a function

```
check :: Char-> Exn Char
check c = if isPrint c || isSpace c
           then return (toLower c)
           else raise "Illegal character"
```

which takes its argument to lower case, but raises an exception if the character is neither printable nor a white space. We now want to combine this exception-raising function with a state monad `Store` (see Sect. A.3) to implement a function which imperatively counts the occurrence of a particular character in a string, raising an exception if the string contains non-printing non-space characters. The type of the function will be

```
count :: Ref Int-> Char-> String->
         Plus Exn (Store Int) Int
```

as it uses both `Store` (of which `Ref` is a part) and `Exn`. Within the function `count`, we can use both stateful computations and exception-raising computations, injecting them into the coproduct using `inl` and `inr`:

```
count r _ []     = inr (readRef r)
count r x (c:cs) =
  do c <- inl (check c)
     if c == x then inr (incRef r)
        else return ()
     count r x cs
```

To run this computation, we have to map `Store` and `Exn` to another monad using `coprod`. Note that we have not defined `coprod` yet (we will do so in Sect. 3.2); the point here is that to *use* the coproduct all we need is the universal property, as given by the mediating morphism `coprod`. This follows good programming practice in hiding the implementation of an abstract data type from the programmer. The simplest monad possible is the identity monad `Id`, as defined in the Appendix A.1. We can use the identity monad as a target monad if for both `t1` and `t2`, we have maps `t1 a -> a, t2 a-> a`. For the identity monad we get a special case of `coprod`:

```
coprod':: (Triple t1, Triple t2)=>
          (forall a. t1 a-> a)->
          (forall a. t2 a-> a)-> Plus t1 t2 a-> a
coprod' f g x = r where
   Id r = coprod (Id. f) (Id. g) x
```

To evaluate exceptions, we catch all exceptions which might occur, and to evaluate stateful computations, we have the `runSt` combinator. Then, we can run the `count` computation with

```
run :: Char-> String-> Int
run x s = coprod' catch runSt
             (do r<- inr (newRef 0)
                 count r x s)
```

Now we want to augment the function by having it print a dot each time it encounters an occurrence of the character it is supposed to count. Thus, we add the predefined `IO` monad to the type by replacing `Store Int` with `Plus (Store Int) IO`, and the type becomes

```
count' :: Ref Int-> Char-> String->
          Plus Exn (Plus (Store Int) IO) Int
```

The code remains largely unchanged, except that we need to change the injections of the stateful computations to `inr.inl`. This change seems inconvenient at this point, but we will show how to avoid it later (see Sect. 3.4):

```
count' r _ []     = inr (inl (readRef r))
count' r x (c:cs) =
  do c <- inl (check c)
     if c == x then do inr (inl (incRef r))
                       inr (inr (putStr "."))
        else return ()
     count' r x cs
```

This example also shows that we can use both self-defined and built-in monads. To run this example, we cannot use the identity monad as the target anymore, since we can not get out of the `IO` monad; hence, we use `IO` as the target monad:

```
run' :: Char-> String-> IO Int
run' x s =
   coprod (return.catch)
          (coprod (return. runSt) id)
                  (do r<- inr (inl (newRef 0))
                      count' r x s)
```

Summing up, if we have two different monads and we want to implement a computation using both of these, we can use the coproduct of the two monads. The universal property of the coproduct gives the mediating morphism `coprod`, which allows us to define functions (like `run` and `run'` above) out of the coproduct, by defining functions `f` and `g` on the component monads. So, we can have more than one function out of the coproduct monad, but `coprod f g` is determined uniquely by `f` and `g`.

Now that we have seen how to use the coproduct in a practical situation, we turn to the actual construction of the coproduct monad.

3.2 Constructing the Coproduct

To motivate our general construction, we consider the simple case of the coproduct of two term algebra monads. Given two signatures Σ and Ω with corresponding term algebra monads T_Σ and T_Ω, the coproduct $\mathsf{T}_\Sigma + \mathsf{T}_\Omega$ should calculate the terms built over the disjoint union of the signatures, $\Sigma + \Omega$, i.e. $\mathsf{T}_\Sigma + \mathsf{T}_\Omega = \mathsf{T}_{\Sigma+\Omega}$.[2] Terms in $T_{\Sigma+\Omega}(X)$ have an inherent notion of layer, as a term in $T_{\Sigma+\Omega}$ either is a variable or decomposes into a term from T_Σ (or T_Ω), and strictly smaller subterms with head symbols from Ω (or Σ respectively). 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:

$$
\begin{aligned}
(T_\Sigma + T_\Omega)(X) = \;& X + T_\Sigma(X) + T_\Omega(X) + \\
& T_\Sigma T_\Sigma(X) + T_\Sigma T_\Omega(X) + \\
& T_\Omega T_\Sigma(X) + T_\Omega T_\Omega(X) + \\
& T_\Sigma T_\Omega T_\Sigma(X) + \ldots
\end{aligned} \tag{2}
$$

In terms of Haskell, each layer corresponds to a particular computational feature, and the coproduct allows arbitrary interleavings of computations from the two monads. This reflects one of our key points, namely that, in general, $T \cdot R$ is too simple a data structure to represent the interaction of T and R since one only has a T-layer sitting above one R-layer.

Unfortunately, equation (2) is too simple in that different elements of the sum represent the same element of the coproduct monad. To see this, note that variables from X are contained in each of the summands in the right hand side. Similarly, in the example of Sect. 3.1, two or more computations from the same monad are layered above each other and should be composed using the multiplication from that monad. Therefore, the coproduct is a quotient of the sum in equation (2).

Kelly [7, Sect. 27] has shown the construction of colimits of ranked monads, from which we can deduce coproducts of monads as a special case. Rank is a generalisation of finitariness to higher cardinals which basically allows operations of infinitary arity provided they are bounded above by the rank of the monad. Roughly, the construction of the coproduct for such monads proceeds in two steps: we first construct the coproduct of pointed functors, and then the coproduct of two monads.

A *pointed functor* is an endofunctor $S : \mathcal{C} \to \mathcal{C}$ with a natural transformation $\sigma : 1 \Rightarrow S$ (this is called *premonad* in [6]). 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

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

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.

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 $'x \sim 'x$ where the quote of the left-hand side injects the variable into $T_\Sigma(X)$, whereas the quote on the right injects the variable into $T_\Omega(X)$. Categorically, this process is modelled by a *pushout*:

Definition 6. Given two pointed functors $\langle T, \eta_T\rangle$ and $\langle R, \eta_R\rangle$, 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 (3)

$$\begin{tikzcd} X \arrow[r, "\eta_T"] \arrow[d, "\eta_R"'] & TX \arrow[d, "\sigma_T"] \\ RX \arrow[r, "\sigma_R"'] \arrow[ur, phantom, "\ulcorner", very near end] & QX \end{tikzcd} \tag{3}$$

The coproduct of monads is constructed pointwise as well: the coproduct monad maps each object X to the colimit of a specific diagram.

Definition 7. Given two finitary 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:

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

where Q, σ_T, σ_R are given by Def. 6, and D_β is the diagram in Fig. 1 with the colimiting morphism $x_\beta : D_\beta \to X_{\beta+1}$. Given the shape of the diagram, x_β is a single morphism $x_\beta : QX_\beta \to X_{\beta+1}$ making all arrows in the diagram commute.

$$\begin{tikzcd}
 & TX_{\beta-1} \arrow[dr, "T\eta_T"] & & & & \\
TTX_{\beta-1} \arrow[ur, "\mu_T"] \arrow[rr, "1"] & & TTX_{\beta-1} \arrow[r, "T\sigma_T"] & TQX_{\beta-1} \arrow[r, "Tx_\beta"] & TX_\beta \arrow[dr, "\sigma_T"] & \\
 & & & & & QX_\beta \\
RRX_{\beta-1} \arrow[rr, "1"'] \arrow[dr, "\mu_R"'] & & RRX_{\beta-1} \arrow[r, "R\sigma_R"'] & RQX_{\beta-1} \arrow[r, "Rx_\beta"'] & RX_\beta \arrow[ur, "\sigma_R"'] & \\
 & RX_{\beta-1} \arrow[ur, "R\eta_R"'] & & & &
\end{tikzcd}$$

Figure 1. The diagram defining the coproduct.

Note that the two triangles on the left of Fig. 1 are *not* the unit laws of the two monads T, R (see Def. 3), otherwise the diagram would be trivial.

Def.3 has been given for finitary monads, and can easily be extended to monads with rank, using transfinite induction. However, it is unclear how this would work out in practical terms, so we restrict ourselves to finitary monads here. As has been pointed out earlier, all known computational monads are finitary except the continuation monad, which actually does not even have a rank. Despite being a general answer to the construction of the coproduct of two monads, the usefulness of Fig. 1 is limited in practice since the shape, size and contents of the diagram make it difficult to reason with directly. We now turn to our contribution, namely an alternative construction which trades less generality for greater simplicity.

3.3 Implementing the Coproduct

Definitions 6 and 7 show the main difficulty we are faced when implementing the coproduct: namely, that the coproduct is not a free datatype, but a *quotient*. (The usual construction of the colimit of a diagram is to take the coproduct of all the objects appearing in the diagram, and quotient it by the relation described by the arrows [15, V.2].)

The solution to this is to choose a representing type for the coproduct, treat it as an abstract datatype with the three operations from Sect. 3.1 (and the monad operations), and have the operations operate on representatives of the equivalence class. Thus, we need a *decision procedure* for this equivalence. Unfortunately, in general this is impossible: for example if the monads are modelling algebraic theories (ie term algebras quotiented by equations), then this amounts to asking if two elements $s,t \in TX$ are equal under the equations of the theory. This is in general undecidable.

However, in order to decide the equivalence from Fig. 1, we do not need to decide the full equational theory of the two monads involved, but merely when an element t is in the image of the unit η, i.e. $t = \eta(s)$. Such a layer is called a *variable layer* and the following example demonstrates their importance: an element of $\mathsf{T}_\Sigma\mathsf{T}_\Omega\mathsf{T}_\Sigma(X)$ consists of a Σ-layer over a Ω-layer which is itself over a Σ-layer. If the middle Ω-layer is a variable layer, then in the quotient this element will be equal to the element of $\mathsf{T}_\Sigma\mathsf{T}_\Sigma(X)$ consisting of the two Σ-layers. In turn, this element will be equal to the result of applying the multiplication of T_Σ. Our construction concerns itself with *layered monads* for which this question is decidable. The name is chosen to indicate that for such monads we can tell if an element is a proper layer or a variable layer. Concretely a monad is *layered* if there is a function $\eta_X^{-1} : TX \to 1+X$ for all X, which for t in TX either returns an element x of X such that $\eta(x) = t$, or returns the canonical element $*$ of 1, if there is no such x. (In other words, η^{-1} is partial.) While the coproduct for any two monads with rank exists, and is given by Def. 7, our construction will only apply to layered monads but, for these monads, our construction is simpler and hence easier to reason about.

Definition 8. A *layered monad* is a monad $\mathsf{T} = \langle T, \eta, \mu\rangle$ such that there is a natural transformation $\eta_X^{-1} : TX \Rightarrow 1+X$, which is a partial left inverse for η_X, i.e. for all X, $\eta_X^{-1} \cdot \eta_X = in_1$ (where $in_1 : X \to X+1$ is the inclusion).

Layered monads allow us to decide the equivalence on QX from Def. 6 as follows: s is equivalent to t iff $\eta^{-1}(s) = \eta^{-1}(t) \neq *$. The type class of layered monads is a straightforward extension of the `Triple` class; the codomain $X+1$ of η^{-1} corresponds to Haskell's `Maybe` type. In fact, we will only consider layered monads, hence we will add this function to the class:

```
class Functor t=> Triple t where
  ...
  etaInv :: t a-> Maybe a
```

It might be tempting to provide `etaInv _ = Nothing` as a default definition, but that would be wrong in being semantically incorrect.

Analysing Fig. 1, we can see the coproduct will either consist of elements of the form TX_β or RX_β, where X_β is again a representation of the coproduct, or for the base case, $X_0 = X$. Hence given two monads (triples `t1` and `t2`), the coproduct will be represented by a recursive datatype which contains either a layer from `t1`, or from `t2`, or a variable:

```
data Plus t1 t2 a = T1 (t1 (Plus t1 t2 a))
                  | T2 (t2 (Plus t1 t2 a))
                  | Var a
```

where `t1` and `t2` are instances of `Triple`.[3] The coproduct will be a quotient of this datatype by the equivalence relation generated by the diagram D_β (Fig. 1). To understand this equivalence, we come back to equation (2) where there are three forms of equalities. In the following, let $\Sigma = \{\mathtt{F},\mathtt{G}\}$ and $\Omega = \{\mathtt{H}\}$.

- For a variable $x \in X$, we have $x \in X$, $'x \in T_\Sigma(X)$ and $''x \in T_\Omega T_\Sigma(X)$ (and many more); all of these should denote the same term. This equivalence is generated by Def. 6, and the arrows σ_T, σ_Q in the diagram. For the implementation, we need to use the map `etaInv` to detect and remove these variable layers.
- The 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 equivalent to $\mathtt{GG'x} \in T_{\Sigma+\Omega}(X)$. By collapsing the two layers, one identifies these terms. This equivalence is generated by the arrows μ_T, μ_R in the diagram while the implementation will need to check for repeated layers and use the relevant multiplication to collapse such layers.
- The sum (2) over-simplified matters by considering only terms which, when descending from the root to any leaf, we pass through the same number of quotes. However, terms such as `F('G'x,'H'x)` in $T_\Sigma(T_\Sigma X + T_\Omega(X))$ must also be considered. Given that the symbol `G` comes from the same signature as `F`, we create a repeated Σ-layer so that the `F` and `G` can be collapsed together. In effect, we try to create a Σ-sublayer underneath the top Σ-layer to which the multiplication can then be applied.

Thus, a normal form for the equivalence generated from D_β — called a *witness* — should be a term which has no variable layers and no subterms whose top symbol comes from the same monad as the term as a whole. Drawing together all the three quotients just mentioned, all of the following terms have the same witness, namely t_3:

$$\begin{aligned} t_1 &= \mathtt{F('G'x,\ 'H'y)} \\ t_2 &= \mathtt{'F(''G'x,\ ''H'y)} \\ t_3 &= \mathtt{F(G'x,\ 'H'y)} \end{aligned}$$

Note that for legibility we do not distinguish between the quotes associated to Σ and those associated to Ω. Thus from Def. 2, we have that if $x \in X$, then $'x \in T_\Sigma(X)$; hence, if $t \in T_\Sigma(X)$, then $'t \in T_\Sigma(T_\Sigma(X))$, or $'t \in T_\Omega(T_\Sigma(X))$. Thus, the quotes syntactically represent the layer information (which would otherwise only be present implicitly).

In order to calculate the witness of a term, we recursively strip away all unnecessary quotes and collapse layers wherever possible. To define the function calculating the witness, we use the fact that the datatype `Plus` is an initial algebra $\mu Y.FY$ of the functor $FY = X + TY + RY$ (X are the variables), and we can give a recursive function $\mu Y.FY \to A$ out of this datatype by giving an F-algebra structure to A, i.e. a map $FA \to A$, which in turn means three functions $X \to A$, $TA \to A$, $RA \to A$. As a higher-order function, this is called `fold` (just like its counterpart on lists):

```
fold :: (Functor t1, Functor t2)=>
        (a-> b)-> (t1 b-> b)-> (t2 b-> b)->
          Plus t1 t2 a -> b
fold e f1 f2 (Var a) = e a
fold e f1 f2 (T1 t)  = f1 (fmap (fold e f1 f2) t)
fold e f1 f2 (T2 t)  = f2 (fmap (fold e f1 f2) t)
```

Defining the function `strip` which strips away all unnecessary quotes is straightforward:

```
strip1 :: Triple t1=>
          t1 (Plus t1 t2 a) -> Plus t1 t2 a
strip1 t = case etaInv t of
               Just x  -> x
               Nothing -> T1 t

strip2 :: Triple t2=>
          t2 (Plus t1 t2 a) -> Plus t1 t2 a
strip2 t = case etaInv t of
               Just x  -> x
               Nothing -> T2 t

strip :: (Triple t1, Triple t2)=>
          Plus t1 t2 a-> Plus t1 t2 a
strip = fold Var strip1 strip2
```

Collapsing layers is achieved by the multiplication $\mu : TT \Rightarrow T$, so for example $\mu(\mathtt{F('G'x,''H'y)}) = t_3$ above. But we also have to collapse the term t_1 to t_3, and t_1 is not element of $T_\Sigma(T_\Sigma(X))$, but of $T_\Sigma(T_\Sigma(X) + T_\Omega(X))$. In other words, we could collapse in the first argument, but not the second. However, the second argument is equivalent to the term `''H'x`, which is in $T_\Sigma(T_\Omega(X))$, so t_1 is equivalent to $t_1' = \mathtt{F('G'x,\ ''H'x)} \in T_\Sigma(T_\Sigma(X + T_\Omega(X)))$ which now has a repeated layer to which the multiplication can be applied to give t_3. This latter process we call *lifting* as we are raising a sub-layer.

All of this motivates the following definition: to calculate the witness, we first recursively calculate the witness of the subterms; then lift the top sublayer to create a repeated layer if possible; then apply the multiplication; and finally strip of the top layer if it is a variable layer. Of these, the recursive calculation is achieved by defining the witness function in terms of `fold` from above, which applies its argument functions recursively to all subterms:

```
lift1 :: Triple t1=>
         Plus t1 t2 a-> t1 (Plus t1 t2 a)
lift1 (T1 t) = t
lift1 t      = eta t

wit1 :: Triple t1=>
        t1 (Plus t1 t2 a) -> Plus t1 t2 a
wit1 t = strip1 (mu (fmap lift1 t))

lift2 :: Triple t2=>
         Plus t1 t2 a-> t2 (Plus t1 t2 a)
lift2 (T2 t) = t
lift2 t      = eta t

wit2 :: Triple t2=>
```

[3] Haskell would allow us to put class constraints on the variables `t1` and `t2` here, but this only constrains the types of the constructors, which is a bit pointless.

```
       t2 (Plus t1 t2 a) -> Plus t1 t2 a
wit2 t = strip2 (mu (fmap lift2 t))

wit :: (Triple t1, Triple t2)=>
       Plus t1 t2 a-> Plus t1 t2 a
wit = fold Var wit1 wit2
```

We can now implement `eta` and `mu`, and hence make `Plus` an instance of `Triple`. For this, we need to start by making it an instance of `Functor`:

```
instance (Functor t1, Functor t2)=>
         Functor (Plus t1 t2) where
  fmap f = fold (Var. f) T1 T2
```

The simplest definition of `mu` would be

```
mu :: (Triple t1, Triple t2)=>
      Plus t1 t2 (Plus t1 t2 a)-> Plus t1 t2 a
mu = wit. fold id T1 T2
```

But the argument of `mu` consists of `Plus t1 t2` terms built over `Plus t1 t2` terms, and we can assume that these are already in normal form, so we only need to compute the witnesses of the "upper" layer.

```
instance (Triple t1, Triple t2)=>
         Triple (Plus t1 t2) where
  eta x          = Var x
  etaInv (Var x) = Just x
  etaInv (T1 t)  = Nothing
  etaInv (T2 t)  = Nothing
  mu             = fold id wit1 wit2
```

All that remains are now the injections into the coproduct, and the unique mediating morphism. The injections are simple (we only give one):

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

For the definition of the unique mediating morphism, we recursively evaluate the layers of coproduct in the target monad. Given monad morphisms `f` or `g`, we apply `f` and `g` to each layer of the coproduct and compose the resulting computation with the multiplication of the target monad:

```
coprod :: (Triple t1, Triple t2, Triple s)=>
          (forall a. t1 a-> s a)->
          (forall a. t2 a-> s a)-> Plus t1 t2 a-> s a
coprod f g = fold eta (mu. f) (mu. g)
```

Note that `f` and `g` above have to be monad morphisms, i.e. commute with unit and multiplication. Just like the monad laws, we cannot denote this in Haskell, so it is an external assumption that the programmer is responsible for.

To sum up, the coproduct of two monads with rank, a mild technical condition, always exists and is given by Def. 7. This definition, however, is very abstract, and difficult to reason with directly, so we have given a simple implementation for a large class of monads called layered monads. To be precise, this implementation works for all *finitary layered monads*, which include all usual computational monads except for the continuation monad.

In our implementations above, we have striven for clarity, not efficiency. For example, the `fmap` operation for the coproduct is quadratic in the number of layers, since it uses `fold`. Here is a more efficient version, which is linear in the number of layers:

```
instance (Functor t1, Functor t2)=>
         Functor (Plus t1 t2) where
  fmap f (T1 t)  = T1 (fmap (fmap f) t)
  fmap f (T2 t)  = T2 (fmap (fmap f) t)
  fmap f (Var x) = Var (f x)
```

The witness operation as given above is quadratic in the number of layers, and hence the multiplication (`mu`) of the coproduct is quadratic in the number of layers of the upper monad (i.e. in the (`>>=`) operation of the Kleisli category, the right argument). That could be improved, since we do not need to recompute the whole witness of the upper layer, but merely need to check whether we can collapse layers. However, in a typical situation the upper layers will only consist of one layer anyway (see the example from Sect. 3.1), so the present definition seems a good mix of simplicity and efficiency. An optimised implementation should be linear in the number of layers, since in principle we only need to check whether the new layer which is added can be collapsed with any of the top layers from the term it is added to, but we have not pursued the matter further yet.

3.4 Monad Transformers Revisited

We have claimed that monad transformers can be seen as squashing the different layers in the coproduct monad into one particular monad. While the definition is elegant, there seems little metatheoretic support, e.g. it is not clear when and how one can define a monad transformer for a specific monad. A related problem is that monad transformers can not be used for to combine existing monads (like the ubiquitous `IO` monad).

Having said that, the coproduct gives us a canonical monad transformer, which has some practical relevance.

THEOREM 3.1. *Given any monad T, the functor $T + \_$ which takes any other monad S to the coproduct with T is a monad transformer.*

PROOF. The coproduct construction is functorial. Pointedness requires for every monad S a monad morphism $\mathsf{S} \Rightarrow \mathsf{T} + \mathsf{S}$. This can be taken to be the inclusion of the coproduct and then naturality follows from the naturality of inclusions, as does the fact that it is a monad morphism. $\square$

We can use this to introduce some conventions which make the coproduct more readily usable. Recall that in Sect. 3.1 above, as we added the `IO` monad to the coproduct, we had to replace some existing injections, because we changed the type of the expression from `Plus Exn (Store Int) Int` to `Plus Exn (Plus (Store Int) IO) Int`. Clearly, this is inconvenient; if we want to add a monad to an already existing computation, we do not want to have to change existing injections.

Now, the category $\mathbf{Mon}(\mathcal{C})$ of monads over $\mathcal{C}$ has an initial object, namely the identity monad `Id` (see Sect. A.1), and in any category with coproducts and and an initial object 0, we have $X + 0 \cong X$.

In other words, for every monad T, we have $\mathsf{T} + \mathtt{Id} \cong \mathsf{T}$. Thus, if we have a function which has the result type `t a`, we can always replace this with the type `Plus t Id a`, and insert the injections `inl e` for terms `e:: t a`. If we now want to add a second monad `s`, all we need is to change the type to `Plus t (Plus s Id a)`; previously existing code pertaining to the monad `t` remains unchanged, while new code is transformed in that we need to write `inr (inl e')` for expressions `e':: s a`. To add a third monad `m`, we change the type to `Plus t (Plus s (Plus m Id))`, and write `inr (inr (inl e))` etc. Thus, the monad `Id` serves as a placeholder for future extensions and these extensions do not alter the currently existing code.[4]

As a brief example, assume that above we would have started with a recursive version of the `count` function. This would have type

```
count :: Char-> String-> Plus Exn Id Int
```

Then, the first extension would be to add the imperative counting, leading to the type

```
count :: Ref Int-> Char-> String->
         Plus Exn (Plus (Store Int) Id) Int
```

and finally, with the addition of `IO`, we would obtain

```
count :: Ref Int-> Char-> String->
         Plus Exn (Plus (Store Int) (Plus IO Id)) Int
```

but unlike above, the second version would already use `inr.inl` to embed the stateful computations, so when adding the `IO` monad we would not have to change the existing code.

In Haskell, this would need some syntactic sugar to reduce the clutter (we want to read and write `t` instead of `Plus t Id`, write perhaps `in`*n* for `inr`^{n}`(inl t)` etc.), akin to the `do` notation.

4 Properties of the Coproduct

Of course, we have to justify the constructions of the previous section. A full formal correctness proof would be very categorical and hence outside the scope of this paper, but we will sketch how one goes about proving correctness. Moreover, the definitions introduced in this section will also be used later on.

4.1 The Coproduct as a Free Algebra

First off, note that the datatype `Plus t1 t2 a` is *not* the coproduct of the two monads, i.e. $(T+R)(X) \ncong$ `Plus t1 t2 a`. As has been pointed out, `Plus t1 t2 a` only represents the coproduct which is actually a quotient of `Plus t1 t2 a`

The straightforward way to prove this quotient is the coproduct monad would be to first show that the monad laws hold for the quotient, and then prove that on the equivalence classes, the injections `inl` and `inr` are monad morphisms, that `coprod f g` is a monad morphism, that it is unique, and that it satisfies equations (1). This is a lot of work; fortunately, from the categorical constructions we employ, there is an easier way. This alternate proof rests on the idea that one can understand a monad through its *algebras*:

Definition 9. 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.

$$1_X = h{\cdot}\eta_X \qquad h{\cdot}\mu_X = h{\cdot}Th \tag{4}$$

[4] Note that the quadratic complexity of the `mu` operator mentioned above does not have a detrimental effect here, since by its construction the `Id` monad does not actually build any proper layers— everything is a variable.

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

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 10. The category T+R-$\mathbf{Alg}$ 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 $U : \mathsf{T}$+R-$\mathbf{Alg} \to \mathcal{C}$, which takes a $\mathsf{T}+\mathsf{R}$-algebra to its underlying object, and we have the following:

PROPOSITION 4.1 ([7, PROP. 26.4]). *If the forgetful functor $U : \mathsf{T}$+R-$\mathbf{Alg} \to \mathcal{C}$ functor has a left adjoint $F : \mathcal{C} \to \mathsf{T}$+$\mathsf{R}$-$\mathbf{Alg}$, 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; in other words, if there is any other $\mathsf{T}+\mathsf{R}$-algebra (A,h'_t,h'_h) and a morphism $f : X \to A$, then there is a unique algebra morphism $!_f : X \to Y$.

Prop. 4.1 shows that action of the coproduct monad creates free algebras. Functions defined using such morphisms are sometimes called *catamorphisms* [3], and the canonical example is the free datatype of lists, and functions using `fold` on lists. So, `coprod` is for the coproduct of monads what `fold` is for lists, and hence we can use it in this way as we did in Sect. 3.1.

4.2 Distributivity Revisited

From our perspective, composing two monads T and R means combining all possible interleavings of layers from the component monads in $\mathsf{T}+\mathsf{R}$. In contrast, the approach of distributivity is to consider only one possible layering, namely $T{\cdot}R$ consisting of a T-layer above an R-layer. However, in the presence of a *strong distributivity law*, the monad $T{\cdot}R$ actually is the coproduct. Thus an alternative analysis is that distributivity is a special situation in which all layers can be squashed into the layer $T{\cdot}R$. First the relevant definitions [2]:

Definition 11. Given two monads $\mathsf{T} = \langle T,\eta,\mu\rangle, \mathsf{S} = \langle S,\zeta,\xi\rangle$, a *distributive law* is a natural transformation $\lambda : T{\cdot}S \Rightarrow S{\cdot}T$ satisfying four coherence laws [2, Sect. 9.2].

The four coherence laws state that λ respects the unit and multiplication of the monads, e.g. $\xi_T = \lambda{\cdot}T\xi$ or $S\mu{\cdot}\lambda_T{\cdot}T\lambda = \lambda{\cdot}\mu_S$.

Given such a distributive law λ, we have the *compatible monad* [2, Prop. 9.2.2] $\mathsf{T} = \langle S{\cdot}T, \eta^*, \mu^*\rangle$ with $\eta^* = S\eta{\cdot}\zeta, \mu^* = S\mu{\cdot}\xi_{TT}{\cdot}S\lambda_T$.

However, with a slightly stronger distributive law, the compatible monad is also the coproduct of T and S.

Definition 12. Given two monads $\mathsf{T} = \langle T, \eta, \mu \rangle, \mathsf{S} = \langle S, \zeta, \xi \rangle$, a *strong* distributive law is a distributive law $\lambda : T \cdot S \Rightarrow S \cdot T$ such that, for any $\mathsf{T}+\mathsf{S}$-algebra (X, α, β), diagram (5) commutes:

$$\begin{tikzcd} TSX \arrow[r, "T\beta"] \arrow[d, "\lambda"'] & TX \arrow[dd, "\alpha"] \\ STX \arrow[d, "S\alpha"'] & \\ SX \arrow[r, "\beta"] & X \end{tikzcd} \tag{5}$$

THEOREM 4.2. *Given two monads $\mathsf{T} = \langle T, \eta, \mu \rangle, \mathsf{S} = \langle S, \zeta, \xi \rangle$ and a strong distributive law $\lambda : T \cdot S \Rightarrow S \cdot T$, the compatible monad $\mathsf{T} = \langle S \cdot T, \eta^*, \mu^* \rangle$ is the coproduct of T and S.*

PROOF. To show the theorem, we first give T a $\mathsf{T}+\mathsf{S}$-algebra structure, and then show it is a free algebra. By Prop. 4.1, it is then the coproduct $\mathsf{T}+\mathsf{S}$.

The algebra structure is given by two maps

$$SSTX \xrightarrow{\xi_T} STX \qquad TSTX \xrightarrow{\lambda_T} STTX \xrightarrow{S\mu} STX$$

Equations (4) easily follow from the coherence laws of the distributive law, and the unit laws of the monads.

We now have to show that the algebra structure is free. To this end, we show that given any other $\mathsf{T}+\mathsf{S}$-algebra (K, α, β) and a morphism $f : X \to K$, there is a unique morphism $!_f : STX \to K$ such that $!_f \cdot \eta^* = f$, and $!_f$ is a $\mathsf{T}+\mathsf{S}$-algebra morphism.

The unique morphism is defined as $!_f = \beta \cdot S\alpha \cdot STf$. Diagram (6) shows that $!_f \cdot \eta^* = f$, where the squares are naturality squares, and the triangles are the left equation of (4) for the $\mathsf{T}+\mathsf{S}$-algebra (K, α, β).

$$\begin{tikzcd} X \arrow[r, "\zeta"] \arrow[d, "f"'] & SX \arrow[r, "S\eta"] \arrow[d, "Sf"] & STX \arrow[d, "STf"] \\ K \arrow[r, "\zeta"] \arrow[dr, "1"'] & SK \arrow[r, "S\eta"] \arrow[d, "\beta"] \arrow[dr, "1"] & STK \arrow[d, "S\alpha"] \\ & K \arrow[dr, "1"'] & SK \arrow[d, "\beta"] \\ & & K \end{tikzcd} \tag{6}$$

To show that $!_f$ is an algebra morphism, we have to show that

$$\begin{tikzcd} SSTX \arrow[r, "S!_f"] \arrow[d, "\xi_T"'] & SK \arrow[d, "\beta"] \\ STX \arrow[r, "!_f"] & K \end{tikzcd} \qquad \begin{tikzcd} TSTX \arrow[r, "T!_f"] \arrow[d, "\lambda"'] & TK \arrow[dd, "\alpha"] \\ STTX \arrow[d, "S\mu"'] & \\ STX \arrow[r, "!_f"] & K \end{tikzcd} \tag{7}$$

The proofs are simple diagram chases, which we omit here; the first uses naturality of ξ and (4) for (K, α); for the second, we additionally need strong distributivity (5).

Now assume we have another $\mathsf{T}+\mathsf{S}$-algebra morphism $m : STX \to K$ s.t. $m \cdot \eta^* = f$, and consider diagram (8); the lower two parallelo-

$$\begin{tikzcd} STX \arrow[rr, "ST\zeta"] \arrow[dr, "S\zeta_T"'] \arrow[dd, "1"'] & & STSX \arrow[dl, "S\lambda"'] \arrow[dr, "STS\eta"] & \\ & SSTX \arrow[dr, "SST\eta"] \arrow[dd, "1"] & & STSTX \arrow[dl, "S\lambda"'] \arrow[d, "STm"] \\ STX \arrow[dr, "S\zeta_T"'] \arrow[dd, "1"'] & & SSTTX \arrow[dl, "SS\mu"'] & STK \arrow[ddll, "S\alpha"] \\ & SSTX \arrow[dl, "\xi_T"'] \arrow[d, "Sm"] & & \\ STX \arrow[d, "m"'] & SK \arrow[dl, "\beta"] & & \\ K & & & \end{tikzcd} \tag{8}$$

grams are (7) with m for $!_f$ (since m is a $\mathsf{T}+\mathsf{S}$-algebra morphism), the triangles on the left are the unit laws of the monads, the diamond is naturality of λ, and the triangle on the top coherence of λ. Now, the arrow on the left-hand face of the diagram is m, and it equals the arrows on the right-hand face of the diagram, and we have (with the assumption $f = m \cdot \eta^* = m \cdot S\eta \cdot \zeta$)

$$\begin{aligned} m &= \beta \cdot S\alpha \cdot STm \cdot STS\eta \cdot ST\zeta \\ &= \beta \cdot S\alpha \cdot ST(m \cdot S\eta \cdot \zeta) \\ &= \beta \cdot S\alpha \cdot STf \\ &= !_f \end{aligned}$$

making $!_f$ unique as required. □

It should be pointed out that the strong distributivity requirement is more than a mere technical condition. For example, consider the two monads given by signatures $\Sigma = \{\mathtt{F}\}, \Omega = \{\mathtt{G}\}$ with a unary operation each. The terms in $T_\Sigma(X)$ are of the form $\mathtt{F}^n(x)$ (i.e. $\mathtt{F}$ applied n times). Then clearly there is a distributive law $\lambda : T_\Sigma T_\Omega \Rightarrow T_\Omega T_\Sigma$, which takes $\mathtt{F}^n(\mathtt{G}^m(x)$ to $\mathtt{G}^m(\mathtt{F}^n(x)$. However, there is no strong distributive law as this would require that taking any carrier set with any two unary operations, these should commute.

The relevance of Theorem 4.2 is that even when working with distributivity laws, we are close to working with the coproduct. The prime example here is if one of the monads in question is the exception monad, then we always have a strong distributivity law $\lambda_{\mathtt{Exn}} : \mathtt{Exn} \cdot T \Rightarrow T \cdot \mathtt{Exn}$, hence for any other monad T, we have

$$T + \mathtt{Exn}(X) = T(\mathtt{Exn}(X)) \tag{9}$$

5 Conclusion

This paper has introduced the coproduct of two monads as their canonical combination: the smallest non-interacting combination of their computational effects. From the general categorical construction [7], we have derived an implementation in Haskell for a wide class of monads, the so-called *layered* monads.

Computations in the coproduct monad can be thought of as sequences of steps in the two component monads. In particular, the monads retain their laziness. Thus, `Store a + Store b` is sequences of steps in `Store a` and `Store b`; this is however not the same as the monad `Store (a, b)`, since in the latter we can store and retrieve tuples of `(a,b)`, which is not possible in the coproduct.

We have investigated the relationship of other combinations based on distributivity, and have shown that if the two monads are strongly distributive, then their coproduct coincides with the functorial composition. Comparison with monad transformers has shown that the coproduct is a more general, flexible approach with a stronger metatheory.

In Haskell (and in particular the Glasgow Haskell Compiler, GHC), the `IO` monad is the mother of all monads; it has mutable references (i.e. state transformers), input/output, concurrency, exceptions and much more. In our approach, we would suggest to dismantle the `IO` monad into its constituting monads, and combine them with the coproduct as needed. This has the advantage that the effects of most monads can be localised, so using e.g. state threads or exceptions in one part of the program will not make the type of every function using this part monadic, as is currently the case with the `IO` monad. Of course, the `IO` monad can be recovered (and continued to be used) as the coproduct of all its components. It remains to be investigated whether the optimisations currently afforded to the `IO` monad by GHC can still be carried out in our style, but it seems unlikely this should not be the case, as all that would be done is to provide a more precise typing.

We are confident that this approach will scale to the combination of more than a handful monads, not least because the coproduct is only the simplest way of combining monads. If we allow arbitrary *colimits* of monads, this will allow shared computation effects (for example, shared state or shared exceptions), and imposed equations such as the distributivity laws (by using coequalisers). This situation is not uncommon; for example, when combining a state monad with another monad, we typically do not want a stateful computation, followed by a computation from the other monad, followed by another stateful computation, as then the second stateful computation knows nothing about the first, and in particular cannot access its state; we would want the stateful computation to distribute over the other computation, so the second stateful computation can use the state left by the first stateful computation. This is related to recent work by Plotkin, Power and Hyland [5], where they describe a commutative combination $\otimes$ which corresponds to an imposed distributivity law in our sense. Technically, their work uses so-called Lawvere theories but these are equivalent to monads [20].

In summary, the coproduct of monads is a simple, modular way of combining two monads. It is more general than distributivity and monad transformers, is applicable in nearly all cases, and based on sound mathematical foundations. We believe it should the functional programmers' first choice when combining monads.

6 References

[1] J. Adamek and J. Rosický. *Locally Presentable and Accessible Categories*. LMS Lecture Notes 183. Cambridge University Press, 1994.

[2] M. Barr and C. Wells. *Toposes, Triples and Theories*. Grundlehren der mathematischen Wissenschaften 278. Springer Verlag, 1985.

[3] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[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] Martin Hyland, Gordon Plotkin, and John Power. Combining computational effects: Commutativity and sum. In *TCS 2002, 2nd IFIP International Conference on Computer Science*, Montreal, 2002.

[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] G. M. Kelly. *Basic Concepts of Enriched Category Theory*, LMS Lecture Notes 64. Cambridge University Press, 1982.

[9] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalizers, and presentations of finitary monads. *Journal for Pure and Applied Algebra*, 89:163– 179, 1993.

[10] David King and Philip Wadler. Combining monads. In J. Launchbury and P.M. Samson, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing Series, Ayr, July 1992. Springer Verlag.

[11] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM Press, Jan 1995.

[12] C. Lüth. *Categorical Term Rewriting: Monads and Modularity*. PhD thesis, University of Edinburgh, 1998.

[13] C. Lüth and N. Ghani. Monads and modular term rewriting. In *Category Theory in Computer Science CTCS'97*, LNCS 1290, pages 69– 86. Springer Verlag, September 1997.

[14] C. Lüth and N. Ghani. Monads and modularity. In *Frontiers of Combining Systems FroCoS'02*, LNAI 2309, pages 18–32. Springer Verlag, 2002.

[15] S. Mac Lane. *Categories for the Working Mathematician*, *Graduate Texts in Mathematics* 5. Springer Verlag, 1971.

[16] Ernest G. Manes. *Algebraic Theories*, *Graduate Texts in Mathematics* 26. Springer Verlag, 1976.

[17] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, Computer Society Press, June 1989.

[18] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, 1990.

[19] Gordon Plotkin and John Power. Notions of computation determine monads. In M. Nielsen and U. Engeberg, editors,

Proc. FOSSACS 2002, LNCS 2303, pages 342– 356, 2002.

[20] John Power. Enriched Lawvere theories. *Theories and Applications of Categories* 6:83–93, 2000.

[21] E. Robinson. Variations on algebra: monadicity and generalisations of equational theories. Technical Report 6/94, Sussex Computer Science, 1994.

[22] D. E. Rydeheard and J. G. Stell. Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms. In *Category Theory and Computer Science*, LNCS 283, pages 114– 139. Springer Verlag, 1987.

[23] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):763–783, 1982.

A Some Useful Monads

This appendix contains the definitions for the monads as used in the text.

A.1 The identity monad

The identity monad is very simple:

```
newtype Id a = Id a

instance Functor Id where
  fmap f (Id x) = Id (f x)
instance Triple Id where
  eta x         = Id x
  etaInv (Id x) = Just x
  mu  (Id x)    = x
```

The identity monad is the initial object of $\mathbf{Mon}(C)$, as for any other monad $\mathsf{T} = \langle T, \eta, \mu \rangle$, the unit is monad morphism from `Id` to T; this gives a unique monad morphism $!_T : \mathtt{Id} \to \mathsf{T}$.

The identity monad is trivially finitary.

A.2 The exception monad

The exception monad adds an error element to the base. In category theory, this is also known as the *lifting monad*.

```
type Error = String

data Exn a = Exn Error | Base a

instance Functor Exn  where
     fmap f (Base x) = Base (f x)
     fmap f (Exn e)  = Exn e

instance Triple Exn where
     eta                 = Base
     etaInv (Base a)     = Just a
     etaInv (Exn e)      = Nothing
     mu (Exn e)          = Exn e
     mu (Base (Exn e))   = Exn e
     mu (Base (Base b))  = Base b
```

The identity monad is finitary: if we add an error element to all finite subsets of an infinite set of variables and take their union, all the exceptions are unified, leaving us with an infinite set of variables and one error element. An easier way to see this is that the exception monad is the free monad on a signature consisting only of one constant.

A.3 The store monad

The store monad is a simple state transformer monad. The state is a list, and references are indices into the list. This is not very efficient, but simple. We leave out the implementations of `newRef` etc, since they are standard.

```
data Store a b = St ([a]-> ([a], b))

instance Functor (Store a) where
     fmap f (St s0) = St (\s-> let (s1, x)= s0 s
                               in  (s1, f x))

instance Triple (Store a) where
     eta a         = St (\s-> (s, a))
     etaInv (St s) = Nothing
     mu  (St s0)   = St (\s-> let (s1, St c)= s0 s
                            in  c s1)
data Ref a     = Ref a Int

newRef   :: a -> Store a (Ref a)
readRef  :: Ref a-> Store a a
writeRef :: Ref a -> a-> Store a ()
runSt    :: Store a b-> b

incRef :: Num a=> Ref a-> Store a ()
incRef r = do i <- readRef r
              writeRef r (i+1)
```

The state transformer monad, of which this store monad is a particular example, is finitary provided the underlying state is finite (a reasonable assumption). So the store monad is finitary as long as we restrict ourselves to finite lists of stores.

The full sources for the code used in this paper can be found at `http://www.informatik.uni-bremen.de/~cxl/papers/icfp02-src.tar.gz`.