Distributive Laws in Programming Structures

Ondřej Rypáček

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

August 2009

Markétce

Abstract

Generalised Distributive laws in Computer Science are rules governing the transformation of one programming structure into another. In programming, they are programs satisfying certain formal conditions. Their importance has been to date documented in several isolated cases by diverse formal approaches. These applications have always meant leaps in understanding the nature of the subject. However, distributive laws have not yet been given the attention they deserve. One of the reasons for this omission is certainly the lack of a formal notion of distributive laws in their full generality. This hinders the discovery and formal description of occurrences of distributive laws, which is the precursor of any formal manipulation.

In this thesis, an approach to formalisation of distributive laws is presented based on the functorial approach to formal Category Theory pioneered by Lawvere and others, notably Gray. The proposed formalism discloses a rather simple nature of distributive laws of the kind found in programming structures based on lax 2-naturality and Gray's tensor product of 2-categories. It generalises the existing more specific notions of distributive laws. General notions of products, coproducts and composition of distributive laws are studied and conditions for their construction given. Finally, the proposed formalism is put to work in establishing a semantical equivalence between a large class of functional and object-based programs.

Acknowledgements

This thesis would not have been possible without the contribution and support of many people around me whom I'd like to express my thanks and gratitude.

In particular I'd like to thank my supervisor Roland Backhouse for taking me on as his PhD student and for showing me the importance of distributive laws.

My many thanks go to my second supervisor, Henrik Nilsson, for having the time to talk whenever I popped in. It was often just the nudge I needed.

My examiners, Martin Hyland and Thorsten Altenkirch for their insightful comments and suggestions at a very intriguing viva, which contributed greatly to the standard of the text.

Mauro Jaskelioff for being my Categorical and real life friend. I enjoyed and also learned a lot from our arrows scribbled on white boards.

Ralf Lämmel for showing me the importance of commitment to ones own work and for being a wonderful host during my stay in Koblenz.

Neil Ghani for believing in me even in times when I was doing boring stuff and Graham Hutton for words of encouragement always when most needed.

Other members of the FOP group for creating such a motivating academic environment.

Lastly, my parents and my beloved wife for supporting me throughout the study in every way possible.

Contents

Chapter 1

Introduction

> All that you may achieve or discover you will regard as a fragment of a larger pattern of the truth which from the separate approaches every true scholar is striving to descry.
>
> *Abbott L. Lowell*

1.1 Categorical Foundations of Programming

In this thesis, category theory is used to formalise and reason about a pattern frequently occurring in programming, which is dubbed *generalised distributive law*. In this endeavour, we don't start from the syntax of a real or hypothetical programming language but we rather use category theory directly as a mathematical programming language in which programming structures are modelled semantically as formal notions of category theory. We therefore feel obliged to begin with a formal justification of this approach. In the rest of the text we use the term *mathematically structured programming* for the idealised discipline of programming which thus arises.

1.1.1 The internal language of a cartesian closed category

One of the cornerstones of the correspondence between category theory and programming is the correspondence between cartesian closed categories (**CCC**) and simply typed lambda calculi. Following Lambek and Scott in [LS88], this correspondence can be formally described as an equivalence of categories of these two kinds of objects. This can be outlined follows: a simply type lambda calculus with finite pairs is given by

- *types*, among which there are at least 1, the singleton type, and which are closed under the type forming operators $\rightarrow$ for *function space* and $\times$ for *products*. Quite

often one considers also other type constants such as $\mathbb{N}$, the type of natural numbers

- *terms*, which are generated freely from variables and constants belonging to the constant types by term-forming operations including *abstraction*, *application* and *pairing*

- *equations*, among which are the usual congruence and conversion rules together with specific equations for the specific constants.

One can assign to any simply typed lambda calculus, $\mathscr{L}$, a cartesian closed category $\mathbf{C}(\mathscr{L})$ where the objects are types and arrows are terms factorised by the equations. An arrow $f : \prod_{i=1}^{n} X_i \longrightarrow Y$ in $\mathbf{C}(\mathscr{L})$ corresponds to a term where n variables of types X_i are free. Composition in this category corresponds to substitution. The abstraction and application of the lambda calculus correspond to the abstraction and application of the **CCC**.

On the other hand, with any **CCC**, $\mathscr{A}$, one can associate an *internal language*, $\mathbf{L}(\mathscr{A})$, which arises from the underlying graph of the category by interpreting the objects as types, arrows as terms and the equations between arrows as the equations.

It can be shown that the assignments $\mathbf{C}$ and $\mathbf{L}$ are the object parts of functors and that these functors define an equivalence of the categories of lambda calculi and cartesian closed categories. These are the foundations of the functional programmers' understanding of category theory as an *abstract theory of types and functions*.

1.1.2 The 2-categorical paradigm of programming

Modern programming languages are not simply typed but polymorphically typed in the sense that one can form terms and types parameterised by types. Formally, this is known as the polymorphic, or second-order, lambda calculus, or System-F [Gir72, Rey74].

The way the correspondence of simply typed lambda calculi and cartesian-closed categories extends to polymorphic lambda calculi can be very roughly approximated as follows:

- a **CCC**, $\mathscr{K}$, is the universe of types with term- and type-forming operations as described above

- by [RP90], certain type expressions parameterised by types correspond to functors $\mathscr{K} \longrightarrow \mathscr{K}$

- by [Rey83], terms parameterised by types correspond to natural transformations

Categories, functors and natural transformations form a 2-category. Informally, it is a category of categories and functors with the usual units and composition where for

each pair of categories, $\mathscr{A}$, $\mathscr{B}$, the functors from $\mathscr{A}$ to $\mathscr{B}$ together with natural transformations [1] form a category, the so-called *hom-category*. The composition in all hom-categories is the pointwise (so-called *vertical*) composition of natural transformations with components $(\beta \cdot \alpha)_X = \beta_X \cdot \alpha_X$ and composition of functors extends to natural transformations as so-called *horizontal composition* of natural transformations.

Note that the above is not a description of models of System-F, just a statement that the models look like certain 2-categories. See [RP90, Rey83] for further details. In this sense, 2-category theory is an abstract theory of kinds, type constructors and parametrically polymorphic functions.

1.1.3 Mathematically structured programming

Although there are various subtleties in the correspondence between categories and programs, they aren't such that they can't be handled properly should the need arise. The important point is that one can move between the mathematical and programming worlds and use categorical techniques to reason about and construct programs. Moreover, by comparing patterns that occur in programming to formal notions of category theory one can discover, understand and develop sound mathematical foundations of informal notions in programming. This has proven hugely successful and remains one of the driving forces of progress in programming, pure functional programming in particular.

A shining example is the success of *monads* as notions of computations with side effects, an idea due to Moggi [Mog91]. Monads have since become a major organising principle in pure functional programming and form the formal foundations of procedural behaviour in pure functional languages such as HASKELL. Of similar significance is the so-called *initial algebra semantics* as a semantics of inductive datatypes and functions defined by induction. Coalgebras and coinduction, on the other hand, formalise state based systems [JR97, Rut00], examples of which are objects in object-oriented programming [Rei95, Jac96]. These are just a few examples of the major role category theory plays in the development of the mathematics of programming.

In this thesis category theory is used to formalise and reason about distributive laws in programming. Our starting point is the notion of a *functor with (additional) structure*. An example is a monad, which comprises of a category $\mathscr{C}$, an endofunctor $\mathbf{T} : \mathscr{C} \longrightarrow \mathscr{C}$ and two natural transformations – the additional structure – $\mu : \mathbf{T}^2 \Longrightarrow \mathbf{T}$, $\eta : \mathbf{1} \Longrightarrow \mathbf{T}$. Moreover, μ and η must satisfy additional equations, namely that $\mu \cdot \mathbf{T}\mu = \mu \cdot \mu_{\mathbf{T}}$ and $\mu \cdot \mathbf{T}\eta = 1_{\mathbf{T}} = \mu \cdot \eta_{\mathbf{T}}$. These are called *coherence properties*.

One can also view functors with structure as constructive definitions of functors satisfying certain properties witnessed by the additional structure. For instance, a monad,

[1] Natural transformations, α, from $\mathbf{F}$ to $\mathbf{G}$, both of type $\mathscr{A} \longrightarrow \mathscr{B}$ are denoted $\alpha : \mathbf{F} \Longrightarrow \mathbf{G} : \mathscr{A} \longrightarrow \mathscr{B}$, or just $\alpha : \mathbf{F} \Longrightarrow \mathbf{G}$.

$\mathbf{T}$, on a discrete category (a set) is a closure operator where functoriality of $\mathbf{T}$ gives monotonicity, η witnesses extensibility and μ idempotence. The coherence properties ensure coherence of the witnesses in the sense that any two proofs of a property obtained by composition of the witnesses are equal.

In programming, functors with structure appear in various forms depending on the language features.

Good examples are HASKELL's *type classes* or *signatures* in SML, which are both general mechanisms for defining arbitrary such notions. In other languages and paradigms the correspondence is less explicit nevertheless in principle a functor with structure is always a parametric type equipped with a collection of parametrically polymorphic functions which can be used for programming with all instances of the type. The equations the additional structure must satisfy are most often ignored in practice and supposedly checked by the programmer on the side. The situation is different in the dependently typed setting, e.g. in COQ, EPIGRAM or AGDA, which allows the equations to be expressed as well.

1.2 Distributive Laws

1.2.1 Distributivity in programming structures

Datatype-generic programming (a.k.a. polytypic programming) [BJJM99, Gib07, Bir88, MFP91, JJ97, HJL06] has arisen in the 1990's from the observation that many programs are generic transformations of datatypes whose behaviour is in the most part determined by the structure of the datatype. The goal of the discipline is to minimise, eliminate or automatically generate such programs in order to increase maintainability, minimise the chance of error and to allow programmers to shift focus towards the specific parts of a programming task.

There are two main approaches to generic programming. Both are based on the categorical understanding of regular datatypes as carriers of initial algebras of polynomial functors, so-called *shape functors*. One of the approaches is characterised by definitions of functions by induction on the polynomial structure of shape functors [JJ97, HJL06]. The other approach, the so-called *algebra of programming* approach [BdM96, MFP91, Bir88], is characterised by the use of higher-order structured recursion operators parametrised by a shape functor instead of ad-hoc pattern matching.

The generic programming community, being on the lookout for higher-order patterns in programming, have pointed out [HB97, Mee98] that many interesting parametric functions on polymorphic datatypes are of types resembling the following:

$$\alpha : \mathbf{FG} \Longrightarrow \mathbf{GF} \ , \tag{1.1}$$

for functors $\mathbf{F}$, $\mathbf{G}$, thought of as constructors of parametric datatypes. A basic example is the function `zip` taking a pair of lists to a list of pairs[2]

```
zip :: forall x y.([x],[y]) -> [(x,y)]
```

Clearly, not all functions of the above type classify as correct implementations of the expected behaviour. The expected behaviour of `zip` is characterised by the conditions that the list of first components of the result is the first component of the input, and likewise for second components. Similarly, any `zip` should be length preserving, which corresponds to the requirement for `zip` to be *coherent* with the structure of lists given by its two constructors: a zip of empty lists is an empty list and a `zip` of a *cons* is the pair of the heads prepended before the `zip` of the tails.

Functions such as `zip` for arbitrary regular datatypes were studied by Hoogendijk and Backhouse in [HB97] and called *zips*. The work is carried out in the allegorical (relational) setting which makes it easy to deal with partiality. For instance, if all lists in a tree have *the same length*, n, one can apply the function

```
zipTree :: forall x. Tree [x] -> [Tree x]
```

which yields a list of length n of trees of the same shape, i.e. it is shape preserving. Note that no such natural transformation exists in the category of sets and total functions, Set, even though `Tree` and `[ ]` (lists) are definable. This is not an issue in allegories and it is shown that *zips* are definable for all regular datatypes and that they always preserve shape. Although these are useful insights, the allegorical setting makes the results difficult to relate to programming and other categorical results.

Later, Meertens in [Mee98] defines *functor pullers* as polymorphic functions swapping the order of two functors. An example of a puller that exists for any category $\mathscr{C}$ with products and an endofunctor List is

$$\mathsf{unzip} \;:\; [\,](\times) \Longrightarrow (\times)[\,]^2$$

which is in HASKELL the function

```
unzip :: forall x y. [(x,y)] -> ([x],[y])
```

Meertens also shows a construction of functor pullers for regular datatypes modulo some additional conditions. The construction is carefully worked out for regular datatypes and is therefore directly applicable to functional programming. However, no theoretical insight is provided as to the nature of the construction, which makes it difficult to generalise. Moreover, the structure of $\mathbf{H}$ is not considered. This is significant as shown e.g. in [MP08, GdSO09] where examples of natural transformations of type (1.1) where $\mathbf{G}$ is a so-called *applicative functor* are shown.

[2]The "has type" relation in HASKELL is denoted by `::`, lists of elements of type x are denoted `[x]`

In [MP08], the notion of an `iFunctor` is defined as a datatype for which there exists a natural transformation like (1.1) where $\mathbf{G}$ is any applicative functor. Practical applications of `iFunctor` to parsing and datatype traversal are shown. An example of such a traversal over lists is a generalisation of another similar natural transformation, sequence, previously defined for monads, which takes a list of applicative functors to an applicative functor on lists:

```
sequence :: Applicative i => forall a . [i a] -> i [a]        .
```

In summary, there is a mounting evidence of the importance of such natural transformations in programming and the need for their proper formal understanding. There seems to be an underlying pattern in these and other similar examples, which we set out to identify and study in this thesis under the name *generalised distributive law*.

1.2.2 Distributive laws of monads and related notions

A starting point for a categorical study of natural transformations like (1.1), which are in some sense *coherent* with $\mathbf{F}$ and $\mathbf{G}$ must clearly be Beck's notion of a *distributive law (of monads)*[3] [Bec69]. Monads in category theory can be used to formalise algebraic theories such as monoids or groups. A distributive law of monads is defined as follows.

Definition 1.2.1 (Distributive Law of Monads)**.** *Let $\mathscr{C}$ be a category, and $(\mathbf{T} : \mathscr{C} \longrightarrow \mathscr{C}, \mu, \eta)$ and $(\mathbf{U} : \mathscr{C} \longrightarrow \mathscr{C}, \nu, \zeta)$ a pair of monads. A distributive law of $\mathbf{T}$ over $\mathbf{U}$ is a natural transformation $\lambda : \mathbf{TU} \Longrightarrow \mathbf{UT}$ such that:*

$$\begin{array}{rclcrcl} \lambda \cdot \eta\mathbf{U} & = & \mathbf{U}\eta & \qquad & \lambda \cdot \mu\mathbf{U} & = & \mathbf{U}\mu \cdot \lambda\mathbf{T} \cdot \mathbf{T}\lambda \\ \lambda \cdot \mathbf{T}\zeta & = & \zeta\mathbf{T} & & \lambda \cdot \mathbf{T}\nu & = & \nu\mathbf{T} \cdot \mathbf{U}\lambda \cdot \lambda\mathbf{U} \end{array} \tag{1.2}$$

The relation of categorical distributive laws of monads to distributive laws in elementary algebra, which gives the former its name, is the following. A distributive law in elementary algebra is usually defined by the equality, for all x, y, z:

$$(x + y) \times z \;=\; (x \times z) + (y \times z) \tag{1.3}$$

where $\times$ and $+$ are the usual multiplication and addition. Operationally, one may view the left-hand side as addition (in "$x + y$", and "$z + 0$") followed by multiplication. On the other hand, the right-hand side prescribes multiplication followed by addition.

Let us for any set X write $\mathbf{S}X$ for the set of well-bracketed terms formed from the symbol $+$ and variables from X (formal sums). Similarly, let $\mathbf{P}X$ denote the set of well-bracketed terms formed from the symbol $\times$ and variables from X (formal

[3] Beck calls it just a *distributive law* but as several derived notions have been defined since then, and we introduce their generalisation we add "...of monads" to disambiguate it.

products). Then the set $\mathbf{PS}X$ is the set of formal products of formal sums. So to speak, where "multiplication follows addition". Similarly, $\mathbf{SP}X$ is the set of formal sums of formal products where "addition follows multiplication". Now, a distributive law of multiplication over addition amounts to a uniform rewriting rule turning products of sums into sums of products. Formally, there is a natural collection of functions:

$$\lambda_X : \mathbf{PS}X \Longrightarrow \mathbf{SP}X \tag{1.4}$$

which respects the associativity and unit of substitution. Note that the opposite direction, from $\mathbf{SP}X$ to $\mathbf{PS}X$, corresponds to factorisation, which is not always possible.

Now, the assignments $\mathbf{S}$, $\mathbf{P}$, extend to endofunctors on the category of sets, which together with substitution and the injection of variables as trivial terms form monads. Namely, $\mathbf{S}$ is the free abelian group monad, $\mathbf{P}$ is the free monoid monad. The collection λ_X then becomes a formal distributive law in the sense of Def. 1.2.1. Distributive laws provide an important tool for combination of algebraic theories, represented as monads. In the example above, the distributive law links $\mathbf{S}$ and $\mathbf{P}$ to a ring monad, $\mathbf{SP}$.

Following Beck, several similar definitions of formal categorical distributive laws have been proposed. Namely, by duality one obtains the definition of a comonad from the definition of a monad, and immediately a definition of a distributive law of comonads by dualisation of Def. 1.2.1. One can also forget the structure on either of the monads (comonads) and correspondingly one half of the coherence conditions (1.2) to obtain a distributive law involving one monad (comonad) and an endofunctor with no additional structure. Importantly, one can also combine the notions to obtain a mixed distributive law of a monad over a comonad which appear in the definition of a bialgebra[Swe69]. Such distributive laws and bialgebras have been used by Turi and Plotkin in [TP97] to categorically formalise adequacy of denotational and structural operational semantics.

1.2.3 Towards generalised distributive laws

Distributive laws of monads and the related notions are relevant to the programming case. Firstly, any polynomial functor on a locally ω-cocomplete category such as Set has a least fixed point and therefore there exists the free monad over $\mathbf{F}$, denoted $\mathbf{F}^*$ and defined as

$$\mathbf{F}^*X \ =_{\mathrm{def}} \ \mu Y.X + \mathbf{F}Y \ .$$

In Set, for any set X, $\mathbf{F}^*X$ is the set of free terms over the signature $\mathbf{F}$ with variables from X. Distributive laws of free monads over comonads have been considered by the author of this thesis and Ralf Lämmel [LR08] as the basis for a semantics of distributed component-based systems whose graph of components is tree-shaped and generated by

a polynomial endofunctor and whose behaviour is given by a polynomial endofunctor.

However, as discussed in [LR08] distributive laws of monads over comonads are too general to guarantee properties such as preservation of the shape of the graph of components as the coherence conditions (1.2) only guarantee coherence with substitution. Moreover, the example of applicative functors spoils every hope as they are strictly more general than monads, yet, their coherent distributive laws are interesting. Any theory of distributivity in programming structures based on the notion of a distributive law of monads would have to leave them out.

The above argument shows not only that distributive laws of monads are not the suitable notion of distributive laws in programming structures but also that there is no such single notion. To cover all cases, we must define a notion of a distributive law which is parameterised by theories of functors with structure. Instances of such generalised distributive laws must define an underlying natural transformation, which is coherent with both theories of functors with structure. The question however arises, whether such a general notion isn't too general to be useful. The following section shows that there is something that can be said for all notions of distributive laws of functors with structure.

1.2.4 Operations on distributive laws

In the case of distributive-law-like natural transformations, certain constructions, such as products, coproducts or composition have been done repeatedly for each notion.

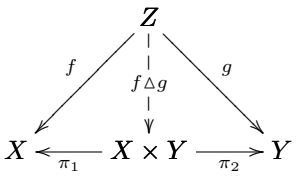
For example, for endofunctors $\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$ on a category $\mathscr{C}$ with products, and distributive laws $\lambda : \mathbf{HF} \Longrightarrow \mathbf{FH}$, $\kappa : \mathbf{HG} \Longrightarrow \mathbf{GH}$, one can define the distributive law $\lambda \boxtimes \kappa$ by the universal property of products in $\mathscr{C}$ as follows:

$$\begin{array}{ccccc} \mathbf{HF} & \xleftarrow{\mathbf{H}\pi_1} & \mathbf{H}(\mathbf{F}\times\mathbf{G}) & \xrightarrow{\mathbf{H}\pi_2} & \mathbf{HG} \\ \downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle \lambda\boxtimes\kappa} & & \downarrow{\scriptstyle \kappa} \\ \mathbf{FH} & \xleftarrow[\pi_1\mathbf{H}]{} & (\mathbf{F}\times\mathbf{G})\mathbf{H} & \xrightarrow[\pi_2\mathbf{H}]{} & \mathbf{GH} \end{array} \tag{1.5}$$

For monads, a similar construction has been considered by Ernie Manes and Philip Mulry [MM07]. For applicative functors, the same has been done in [GdSO09]. In the latter two cases, as both monads and applicative functors have additional structure, it must be shown that the resulting distributive law is coherent with this structure. The proofs are a bit tedious but straightforward by naturality, the preconditions, and universal properties. They are similar in nature but different in the details. If we understood the higher-order nature of these constructions, we could be able to simplify or eliminate the proofs. This is illustrated in the following examples.

Consider monads $(\mathbf{F}, \mu, \eta)$, $(\mathbf{G}, \mu', \eta')$. Because $\mathscr{C}$ has products, there exists a so-

called *cartesian-product monad* defined as $(\mathbf{F} \times \mathbf{G}, (\mu \cdot \pi_1\pi_1) \vartriangle (\mu' \cdot \pi_2\pi_2), \eta \vartriangle \eta')$, where $f \vartriangle g$ denotes the universal arrow of the product in the following situation[4]:



Let λ, κ be as above but in addition coherent with the structure of the monads, $\mathbf{F}$, $\mathbf{G}$, i.e., they in addition satisfy the second line of (1.2) appropriately renamed. It then holds that the definition (1.5) gives a distributive law of $\mathbf{H}$ over the cartesian-product monad. Explicitly, for

$$\begin{aligned} \nu &=_{\text{def}} (\mu \cdot \pi_1\pi_1) \vartriangle (\mu' \cdot \pi_2\pi_2) \\ \zeta &=_{\text{def}} \eta \vartriangle \eta' \end{aligned}$$

one can establish the following:

$$(\lambda \boxtimes \kappa) \cdot \mathbf{H}\zeta = \zeta\mathbf{H} \tag{1.6}$$

$$(\lambda \boxtimes \kappa) \cdot \mathbf{H}\nu = \nu\mathbf{H} \cdot (\mathbf{F} \times \mathbf{G})(\lambda \boxtimes \kappa) \cdot (\lambda \boxtimes \kappa)(\mathbf{F} \times \mathbf{G}) \ . \tag{1.7}$$

For illustration, we show the proof of (1.7), the proof of (1.6) is similar. To this end, observe that in the diagram in Fig. 1.1 the numbered squares and the hexagon commute, where (1) commutes by definition of $\lambda \boxtimes \kappa$, (2) and (4) by definition of ν and naturality, (3) is coherence of λ with μ, and the outside hexagon commutes by naturality and definition of $\lambda \boxtimes \kappa$. It follows that

$$\begin{aligned} \pi_1\mathbf{H} \cdot \lambda \boxtimes \kappa \cdot \mathbf{H}\nu &= \mu\mathbf{H} \cdot \mathbf{F}\lambda \cdot \lambda\mathbf{F} \cdot \mathbf{H}{\pi_1}^2 \\ &= \mu\mathbf{H} \cdot {\pi_1}^2\mathbf{H} \cdot (\mathbf{F} \times \mathbf{G})(\lambda \boxtimes \kappa) \cdot (\lambda \boxtimes \kappa)(\mathbf{F} \times \mathbf{G}) \\ &= \pi_1\mathbf{H} \cdot \nu\mathbf{H} \cdot (\mathbf{F} \times \mathbf{G})(\lambda \boxtimes \kappa) \cdot (\lambda \boxtimes \kappa)(\mathbf{F} \times \mathbf{G}) \end{aligned}$$

A similar argument shows that

$$\pi_2\mathbf{H} \cdot \lambda \boxtimes \kappa \cdot \mathbf{H}\nu = \pi_2\mathbf{H} \cdot \nu\mathbf{H} \cdot (\mathbf{F} \times \mathbf{G})(\lambda \boxtimes \kappa) \cdot (\lambda \boxtimes \kappa)(\mathbf{F} \times \mathbf{G})$$

and therefore

$$\lambda \boxtimes \kappa \cdot \mathbf{H}\nu = \nu\mathbf{H} \cdot (\mathbf{F} \times \mathbf{G})(\lambda \boxtimes \kappa) \cdot (\lambda \boxtimes \kappa)(\mathbf{F} \times \mathbf{G}) \ .$$

Note that if $\mathbf{H}$ is not just an endofunctor but an endofunctor with structure, coherence

[4]The standard notation for this arrow in category theory is $\langle f, g \rangle$.

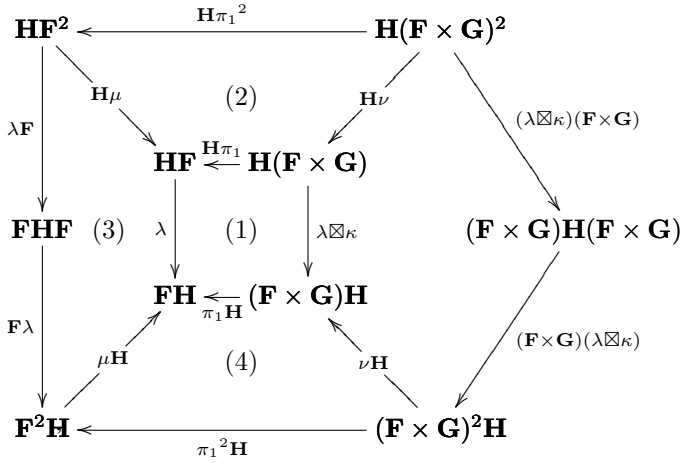


Figure 1.1: Coherence of $\lambda \boxtimes \kappa$ with μ

with its structure must be established as well.

For any notion of a distributive law, just in the case of products one must establish similar coherence properties by similar proofs. The proofs are similar in nature, but the details are different. For instance, an applicative functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{C}$ comes with a natural transformation $\gamma_{X,Y} : \mathbf{F}X \times \mathbf{F}Y \longrightarrow \mathbf{F}(X \times Y)$. A distributive law, λ, of an endofunctor, $\mathbf{H}$, over $\mathbf{F}$ must be coherent with γ in the sense that the following diagram commutes

$$\begin{array}{ccccc}
\mathbf{H}(\mathbf{F}X \times \mathbf{F}Y) & \xrightarrow{\mathbf{H}\pi_1 \vartriangle \mathbf{H}\pi_2} & \mathbf{HF}X \times \mathbf{HF}Y & \xrightarrow{\lambda_X \times \lambda_Y} & \mathbf{FH}X \times \mathbf{FH}Y \\
\downarrow{\scriptstyle \mathbf{H}\gamma_{X,Y}} & & & & \downarrow{\scriptstyle \gamma_{\mathbf{H}X,\mathbf{H}Y}} \\
\mathbf{HF}(X \times Y) & \xrightarrow[\lambda_{X\times Y}]{} & \mathbf{FH}(X \times Y) & \xrightarrow[\mathbf{F}(\mathbf{H}\pi_1 \vartriangle \mathbf{H}\pi_2)]{} & \mathbf{F}(\mathbf{H}X \times \mathbf{H}Y)
\end{array}$$

Note that this is by far not the whole definition of an applicative functor and not all coherence conditions on an applicative functor but for illustration purposes we have singled out this case.

For another applicative functor $\mathbf{G}$, one can define the structure of an applicative functor on $\mathbf{F} \times \mathbf{G}$. In particular, when $\mathbf{G}$ comes with a $\gamma'_{X,Y} : \mathbf{G}X \times \mathbf{G}Y \longrightarrow \mathbf{G}(X \times Y)$, one can define the arrow

$$\begin{aligned}
\nu_{X,Y} \; : (\mathbf{F} \times \mathbf{G})X \times (\mathbf{F} \times \mathbf{G})Y &\xrightarrow{\tau} \mathbf{F}X \times \mathbf{F}Y \times \mathbf{G}X \times \mathbf{G}Y \\
&\xrightarrow{\gamma_{X,Y} \times \gamma'_{X,Y}} \mathbf{F}(X \times Y) \times \mathbf{G}(X \times Y) \equiv (\mathbf{F} \times \mathbf{G})(X \times Y) \;\; ,
\end{aligned} \tag{1.8}$$

where τ is the obvious isomorphism. For another distributive law $\kappa : \mathbf{HG} \Longrightarrow \mathbf{GH}$, where $\mathbf{G}$ is an applicative functor, (1.5) defines the underlying arrow of a product

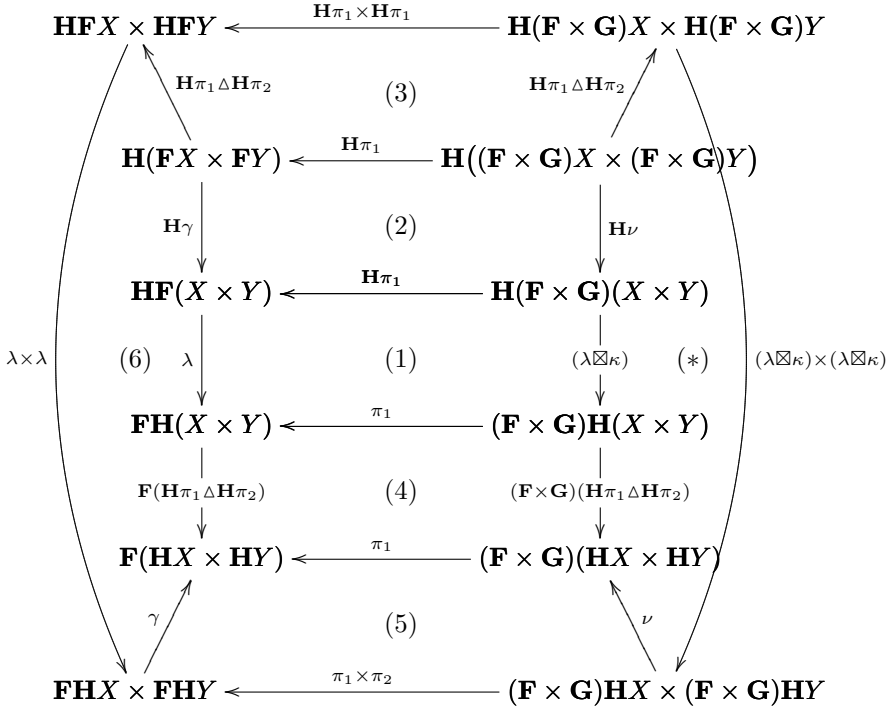


Figure 1.2: Left projection on the coherence of ν

distributive law. To see this, one must establish all coherence conditions for the product. In particular, to establish coherence of $\lambda \boxtimes \kappa$ with γ, observe that in the diagram in Fig. 1.2 all numbered squares commute, where (1) commutes by the definition of $\lambda \boxtimes \kappa$, (6) is a precondition, (2) and (5) commute by definition (1.8), (3) and (4) commute by naturality, and the outside square commutes by definition of $\lambda \boxtimes \kappa$ and naturality. Commutativity of $(*)$ follows by a similar argument as before and a symmetrical diagram for π_2.

There is a common pattern in the two proofs and in the sense they are both simple, which is formalised in this thesis. Without a proper understanding of the pattern, one has no other option than to go through the tedious details for each notion of a distributive law, or omit the proofs of such coherence conditions, as it is has been done before. Moreover, there are cases when omission is not an option, such as computer aided theorem proving.

1.2.5 Generalised distributive laws

2-Category theory is a theory of ordinary categories where one can formally model ordinary-categorical structures. It is therefore the formal setting where various notions of functors with structure, arising both from category theory and programming, can be studied as formal mathematical objects.

In this thesis, functors with structure are modelled as 2-functors into Cat, where the functors are thought of as *models*, their domain 2-categories as *theories*. This turns functors with structure into formal objects of 2-category theory and makes them available for categorical manipulation.

The idea of modelling mathematical theories as categories and their models as functors into suitable *base categories* is due to Lawvere [Law68]. Its significance can be hardly overestimated. It gave birth to the whole field of categorical model theory, which includes various notions of sketches [Ehr68, Wel93, BW85, BW99] as categorical counterparts of the usual set-theoretical definitions by tuples. Categories of models of sketches can also by described axiomatically as so-called locally presentable and accessible categories [GU71, MP90, AR94].

Importantly, once a mathematical notion is described by a theory (a category) its models can be taken also in other than the most obvious categories as long as these categories have enough structure, such as finite products, limits, etc. These can be in particular certain functor categories. For instance, a monad on $\mathscr{C}$ is a monoid in the category of endofunctors on $\mathscr{C}$ and the two coherence conditions for μ and η arise as the associativity and unit conditions of a monoid. This is a key principle also in this thesis as we use models of a theory of functors with structure in a suitable 2-category of models of another theory of functors with structure. This is the formal basis for our notion of generalised distributive laws.

Once distributive laws become objects in a category, one can study the constructions on distributive laws such as products, coproducts or composition as formal constructions on objects in the category. Our categories of generalised distributive laws are 2-categories of 2-functors parameterised by a notion of a theory. We are therefore interested in the constructions parametric in the theory and it turns out that many of the usual constructions arise as such. For instance, the proofs of coherence in Figs (1.1) and (1.2) turn out to be just projections in the corresponding functor categories – 2-categorical *lax natural transformations*.

1.3 Contributions

In this thesis we develop a new notion of a *generalised distributive law*, which is specifically designed to be close to the definitions in (mostly functional) programming languages. Our starting point is mathematically structured programming – a discipline where one seeks parallels between mathematical and programming structures in order to construct better and more dependable programs. The broader intent of this thesis is to demonstrate that mathematical distributive laws deserve a more prominent place in the toolbox of a computer scientist. Because too often mathematics plays a role in programming only at the very beginning to guide the definition of a programming structure and at the end to verify the result, rather than to guide the programming process.

To this end we make original contributions to the state of the art of distributive laws in mathematically structured programing by giving answers to the following questions:

- What is the formal relation of the distributive-law-like polymorphic programs and formal distributive laws in category theory?
- How are the different notions of distributive laws related ?
- Is there a repeating pattern in the constructions of products, coproducts and composition of distributive laws that occur in mathematically structured programming?
- Is it possible to simplify or completely eliminate the proofs of coherence conditions in such constructions?

This is achieved in the following concrete contributions:

1. In Chapter 4, *generalised distributive laws* of two functors with structure are defined, which subsume previously defined specific notions in category theory and mathematically structured programming. The notion is essentially equivalent to J.W.Gray's notion of a quasi-functor of two variables which provides it with strong theoretical foundations and a useful characterisation (Theorem 4.1.3). Our contribution lies in demonstrating the connection of the various notions of coherent distributive-law-like polymorphic programs in mathematically structured programming and Gray's quasi-functors.

2. We show that in this setting one can iterate the construction to obtain distributive laws of more than two notions of functors with structure (Sect. 4.3), and that such iterated distributive laws correspond to Gray's notion of a quasi-functor of n variables. This observation yields a general higher-dimensional characterisation theorem (Theorem 4.3.7).

3. We develop a general approach to products, coproducts and compositions of distributive laws in a way close to their informal definitions in programming. Our analysis discloses the abstract nature of these constructions, their correctness and their coherence properties (Chapter 5). Moreover, we eliminate the proofs of coherence conditions altogether for a class of functors with structure (Theorem 5.1.9).

4. The relevance of the notions introduced by us is documented in many examples throughout the text. In Chapter 6, we collect several key examples in mathematically structured programming. For McBride and Paterson's *applicative functors* we make use of the fact that they are internally higher-dimensional distributive

laws to show their notions of distributive laws, products, compositions and so-called *crushes*. We also show how Meertens's functor pullers are distributive laws of n-ary functors.

5. We contribute to the existing body of examples of distributive laws in programming by developing a notion of equivalence of functional and component-based programs based on distributive laws (Sect. 6.3). We believe it is an important instance of distributive laws, which also contributes to the mathematics of component-based programming.

1.4 Outline

In Chapter 2 the basic notions of 2-category theory are reviewed. The material is standard, nothing of substance there is original. The reader familiar with 2-category theory might wish to skip this section and start reading from Chapter 3 and refer to Chapter 2 only as needed following references in the main text.

In Chapter 3, functors with structure are formalised as functors from certain 2-categories serving as *theories*. The elementary notions are developed and supported by examples. The development starts in Sect. 3.1 by a discussion of formal objects and arrows modelled as functors into a base 2-category. The section is slow-paced and serves mostly to plant the correct intuitions about the functorial approach and to introduce some basic notation. In Sect. 3.2, theories and models of functors with structure are defined, and Sect. 3.3 introduces their morphisms and discusses the two kinds of 2-categories of functors with structure. Formal forgetful functors, defined in Sect. 3.4, are in the functorial setting given by precomposition with inclusions of theories. They allow us, in particular, to define the notions of formal domain, codomain and underlying arrows of functors with structure. The chapter concludes by Sect. 3.5, which introduces important examples of functors with structure which are used and further developed in the remaining text.

In Chapter 4, the key notion of the thesis, a *generalised distributive law*, is defined in Sect. 4.1, Def. 4.1.1. Section 4.2 introduces some useful notation used for generalised distributive laws. Section 4.3 introduces and investigates the important case of *higher-dimensional distributive laws*, i.e. distributive laws where more than two notions of functors with structure are involved. Section 4.5 discusses examples.

In Chapter 5, constructions on generalised distributive laws are studied, namely their products (Sect. 5.1), coproducts (Sect. 5.2) and composition (Sect. 5.3).

In Chapter 6, we collect important examples of distributive laws in programming. Namely we interpret functor pullers and so-called idiomatic functors (`iFunctors`) as generalised distributive laws. We illustrate how the constructions on distributive laws developed in Chapter 5 underlie the constructions previously defined in the literature

for these examples. Finally, we give an original example of an application of distributive laws to formal comparison of functional and component-based programming (previously published by us in [LR08]). The chapter is self-contained to a large degree making it possible for the reader interested in the applications to start reading here and refer to the previous theoretical chapters as needed.

In Chapter 7 we conclude and discuss related and future work.

Chapter 2

Preliminaries

In this chapter, the basic notions of 2-category theory are reviewed. The material is standard, nothing of substance here is original. The reader is assumed to be already familiar with the basic notions of ordinary category theory such as category, functor, natural transformation, adjunction, limits and colimits. Good first introductions to category theory are [Awo06, BW99], Mac Lane's textbook [Mac97] is more advanced.

The exposition of 2-categories in the following text is slanted towards the notions of enriched category theory [Kel82]. In short, a 2-category in Def. 2.2.1 is introduced as a category enriched over a category. In contrast, the more basic, and at first sight more direct approach is to start from 0-cells, 1-cells and 2-cells with two kinds of composition, horizontal and vertical, satisfying certain coherence conditions. The enriched approach is more principled and is thus suitable for the advanced notions gradually introduced later. Namely, it provides the right guidance for the correct definitions of 2-functors in Def. 2.2.5, 2-natural transformations and modifications in Defs 2.2.7 and 2.2.9. Secondly, it leads directly to the definition of 3-categories in Sect. 2.2.5, which will be needed soon. Most importantly, the enriched point of view allows us to define categories that are "very much like but not quite" 2-categories in Sect. 2.4.

2.1 Ordinary Categories

This section collects some basic notation used in the rest of the text.

Categories, $\mathscr{C}, \mathscr{D}, \ldots$, have *objects* $X, Y, \ldots$ and *arrows* $f, g, \ldots$. The set of objects of $\mathscr{C}$ is denoted $|\mathscr{C}|$. The set membership relation $\in$ is occasionally used as $X \in \mathscr{C}$ instead of the formally correct $X \in |\mathscr{C}|$.

Arrows have a *domain* (or source) and a *codomain* (or target). An arrow with domain X and codomain Y is denoted $f : X \longrightarrow Y$.

Composition of arrows $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ is denoted $g \cdot f$, or just gf. Identities are denoted 1_X, or X, or just 1 in diagrams where the object is clear from

context.

Functors between categories are denoted $\mathbf{F}, \mathbf{G}, \ldots$. Their action on objects is denoted $\mathbf{F}(X)$ or $\mathbf{F}X$, the same goes for arrows. Constant functors, i.e. functors factoring through the terminal one-object category, 1, are denoted $\underline{Y}$ for an object Y in $\mathscr{D}$. Formally:

$$\underline{Y} =_{\text{def}} \mathscr{C} \xrightarrow{!} 1 \xrightarrow{Y} \mathscr{D}$$

where the label Y is overloaded to denote also the point in $\mathscr{D}$ picking Y.

Natural transformations between functors are denoted $\alpha, \beta, \ldots$, their components $\alpha_X, \alpha_Y, \ldots$. A natural transformation from $\mathbf{F}$ to $\mathbf{G}$, where $\mathbf{F}, \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$, are denoted either $\alpha : \mathbf{F} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ or by the two-dimensional notation as $\mathscr{C} \underset{\mathbf{G}}{\overset{\mathbf{F}}{\underset{\Downarrow\alpha}{\rightarrow}}} \mathscr{D}$.

For natural transformations α, β, the natural transformation with components $\beta_X \cdot \alpha_X$ is denoted $\beta \cdot \alpha$. This is called *vertical composition*. So-called *horizontal composition* of natural transformations, $\mathscr{C} \underset{\mathbf{G}}{\overset{\mathbf{F}}{\underset{\Downarrow\alpha}{\rightarrow}}} \mathscr{D}$, $\mathscr{D} \underset{\mathbf{I}}{\overset{\mathbf{H}}{\underset{\Downarrow\beta}{\rightarrow}}} \mathscr{E}$, is the natural transformation whose components are $\beta_{\mathbf{G}X} \cdot \mathbf{H}(\alpha_X)$, or equivalently $\mathbf{I}(\alpha_X) \cdot \beta_{\mathbf{F}X}$. It is denoted $\beta \circ \alpha$ or just $\beta\alpha$. For each functor $\mathbf{F}$, the identity natural transformation, i.e. the natural transformation with components 1_X, is denoted $1_{\mathbf{F}}$, or just $\mathbf{F}$. Functors from $\mathscr{C}$ to $\mathscr{D}$ and natural transformations with vertical composition form a category $\mathscr{D}^{\mathscr{C}}$.

Small categories and functors form a category, Cat. Composition in this category, i.e. composition of functors, is denoted $\mathbf{G} \circ \mathbf{F}$ or $\mathbf{GF}$. Note that this notation is consistent with the meaning given to it in the previous paragraph.

The category Cat has products $\mathscr{C} \times \mathscr{D}$, with objects (X, Y) and arrows (f, g). It also has coproducts $\mathscr{C} + \mathscr{D}$, and exponents $\mathscr{D}^{\mathscr{C}}$. In an arbitrary category, $\mathscr{C}$, the set of arrows between objects X and Y is denoted $\mathscr{C}(X, Y)$.

When $\mathscr{C}$ has products, $\times$, there is a natural isomorphism

$$(\ \vartriangle\ ) : \mathscr{C}(X, Y) \times \mathscr{C}(X, Z) \longrightarrow \mathscr{C}(X, Y \times Z)\ , \tag{2.1}$$

which is used as an infix operator as in $f \vartriangle g$. Likewise, when $\mathscr{C}$ has coproducts, $+$, there is a natural isomorphism

$$(\ \triangledown\ ) : \mathscr{C}(X, Z) \times \mathscr{C}(Y, Z) \longrightarrow \mathscr{C}(X + Y, Z)\ . \tag{2.2}$$

Usually, $f \vartriangle g$ is denoted $\langle f, g \rangle$ and $f \triangledown g$ is denoted $[f, g]$. We stray from this convention because the square brackets will have a more important and no less standard role.

We adhere to the convention, already used in (2.1) and (2.2) above, that the name of a functor meant to be used as an infix operator is typeset in parentheses when not

applied to arguments.

2.2 2-Categories

In this section we give a succinct summary of the basic notions of 2-category theory which are essential in the rest of the text. Although the material as presented here might be a bit terse, the rest of the thesis serves as an accessible and detailed example of the notions in the area of computer science. The reader might find it more accessible to skip to Chapter 3 at the first reading and refer to this section on as needed basis following many references there.

2.2.1 Definition

A 2-category $\mathscr{C}$ is a category (so-called *underlying category*), $\mathscr{C}_0$, whose hom-sets are not just sets but categories. Moreover, the composition in $\mathscr{C}_0$ and in each hom-category interact appropriately. A formal definition follows.

Definition 2.2.1 (2-Category)**.** *A 2-category $\mathscr{C}$ consists of*

1. *a class $|\mathscr{C}|$*
2. *for each pair $A, B \in |\mathscr{C}|$ a category $\mathscr{C}(A, B)$*
3. *for each triple $A, B, C \in |\mathscr{C}|$ a bifunctor*

$$c_{A,B,C} : \mathscr{C}(A, B) \times \mathscr{C}(B, C) \longrightarrow \mathscr{C}(A, C)$$

4. *for each $A \in |\mathscr{C}|$ a functor $u_A : 1 \longrightarrow \mathscr{C}(A, A)$, where 1 is the terminal one-object category.*

This data is required to satisfy the following coherence conditions (Figure 2.1):

$$c_{A,C,D} \cdot (c_{A,B,C} \times 1) \;\cong\; c_{A,B,D} \cdot (1 \times c_{B,C,D})$$
$$c_{A,A,B} \cdot (u_A \times 1) \;\cong\; 1 \;\cong\; c_{A,B,B} \cdot (1 \times u_B)$$

We use the following conventions:

1. Elements of $|\mathscr{C}|$ are called 0-cells or *objects*. They are usually denoted $A, B, C, \ldots$. In case we explicitly have the 2-category of categories, functors and natural transformations in mind (Example 2.2.2), we also use script letters $\mathscr{C}, \mathscr{D}, \ldots$, which are otherwise reserved for categories. Instead of $A \in |\mathscr{C}|$ we often write just $A \in \mathscr{C}$ to save notational clutter.

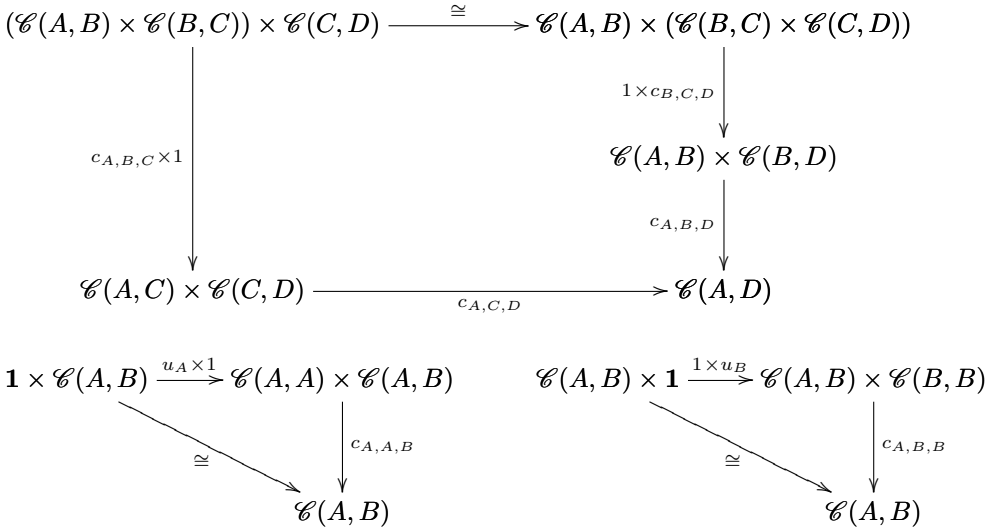


Figure 2.1: Associativity of composition and unit axioms

2. Objects of hom-categories $\mathscr{C}(A,B)$ are called 1-cells or *arrows*, and are denoted $f,g,h,\ldots$ or $\mathbf{F},\mathbf{G},\mathbf{H},\ldots$, which otherwise denote functors.

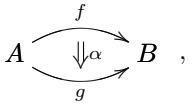
3. Arrows of hom-categories are called 2-cells and denoted with Greek letters $\alpha,\beta,\gamma,\ldots$. This is the same for natural transformations.

4. The action of c is called *horizontal composition* and is denoted $\circ$ or just by juxtaposition when there's no danger of confusion.

5. On the other hand, composition of 2-cells is called *vertical composition* and is always denoted by $\cdot$. So, $\beta\alpha$ always means $\beta\circ\alpha$ rather than $\beta\cdot\alpha$.

6. For each 1-cell $f\in\mathscr{C}(A,B)$ there is a unit 2-cell on f denoted 1_f. We adhere to the usual convention that identities on objects and 1-cells are denoted just by the name of the object or 1-cell, so 1_f can also be denoted just f.

7. In diagrams, the usual convention is used of drawing 2-cells as double arrows running orthogonally to 1-cells as in



,

or $A \xrightarrow[g]{f \atop \Downarrow\alpha} B$ in running text. In writing, we usually chain all typing information, so α above has type $f\Longrightarrow g : A\longrightarrow B$.

8. Horizontal composition is usually pictured just by *pasting* 2-cells next to each other (see (2.3) below).

9. Likewise, vertical composition is pictured just by pasting 2-cells one on top of the other, as in (2.3). Note that there is a way of making sense of an arbitrary 2-categorical diagram of vertices (0-cells), edges (1-cells) and polygons (2-cells). See section 2.2.2.

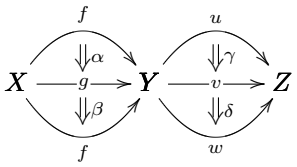
10. From now on, we use the term *ordinary category* for a category which is not a 2-category. In general, the adjective *ordinary* can be used for anything that is explicitly non-2-categorical, such as "ordinary functor", "ordinary natural transformation", etc.

The coherence conditions of Figure 2.1 clearly make $|\mathscr{C}|$ and 1-cells of $\mathscr{C}$ into a category, so-called *underlying category of* $\mathscr{C}$, denoted $\mathscr{C}_0$. Composition in $\mathscr{C}_0$ is given by the action of c on objects of hom-categories; units are given by $u(*)$, where $*$ is the sole object of 1.

Note that the above definition of a 2-category elegantly entails the more elementary definitions where one starts from 0-cells, 1-cells and 2-cells with three kinds of composition that interact in a good way (are coherent). In particular, it is easy to show that horizontal composition respects identities,

$$1_g \circ 1_f \;=\; 1_{gf}$$

and that horizontal and vertical composition commute – so called Interchange Law:



$$(\delta \cdot \gamma) \circ (\beta \cdot \alpha) \;=\; (\delta \circ \beta) \cdot (\gamma \circ \alpha) \tag{2.3}$$

Example 2.2.2 (The 2-category of categories)**.** Small ordinary categories, functors and natural transformations form a 2-category; denoted Cat. Here, for a functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$ and a natural transformation $\alpha : \mathbf{G} \Longrightarrow \mathbf{H} : \mathscr{D} \longrightarrow \mathscr{E}$ (see Fig. 2.2), the horizontal composition of α after the identity on $\mathbf{F}$, $\alpha \circ \mathbf{F}$, coincides with the pointwise application $\alpha_{\mathbf{F}}$:

$$(\alpha \circ \mathbf{F})_X \;\equiv\; \alpha_{(\mathbf{F}X)}$$

Similarly, the horizontal post-composition $\mathbf{I} \circ \alpha$ concides with pointwise functor appli-

cation:

$$(\mathbf{I} \circ \alpha)_X \equiv \mathbf{I}(\alpha_X)$$

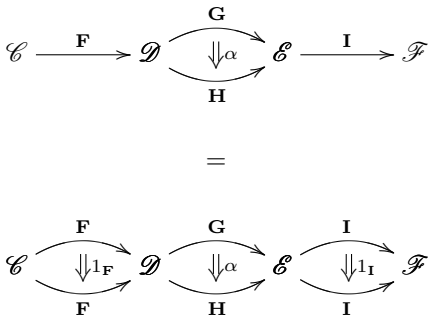


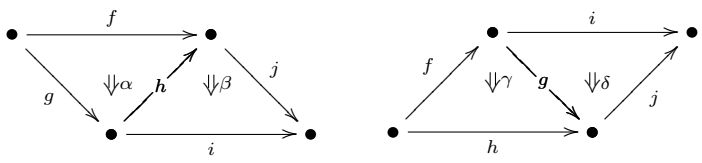
Figure 2.2: Horiziontal composition and identities

Example 2.2.3 (Ordinary categories)**.** Any ordinary category, $\mathscr{C}$, is trivially a 2-category, where each hom-category $\mathscr{C}(A, B)$ is discrete.

2.2.2 Pasting

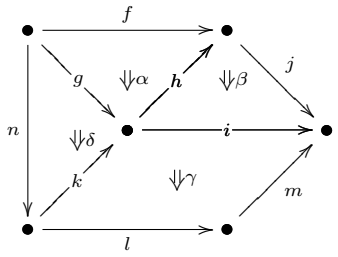
In ordinary category theory, one often draws a diagram of objects and arrows saying "this diagram commutes". Formally this means that for any two objects, A, B, two arbitrary paths from A to B are equal. The validity of this statement rests on associativity of composition of arrows, because any composable path of arrows $\cdot \xrightarrow{f_1} \cdot \xrightarrow{f_2} \cdots \xrightarrow{f_n} \cdot$ defines a unique arrow $f_n \cdots f_1$. Geometrically, the composition gf of arrows $g : B \longrightarrow C$ and $f : A \longrightarrow B$ is obtained by "pasting" g after f at the common boundary, B.

The composition of paths of arrows in a diagram generalises in the case of 2-categories to *pasting* of 2-cells, introduced by Benabou [Bén67]. The two basic situations are



The left-hand side diagram defines by pasting the 2-cell $(\beta g) \cdot (j\alpha) : jf \Longrightarrow ig$. The right-hand side diagram defines by pasting the 2-cell $(j\gamma) \cdot (\delta f) : if \Longrightarrow jh$. In

general, one can use pasting in more general situations such as:



Which defines by pasting the 2-cell

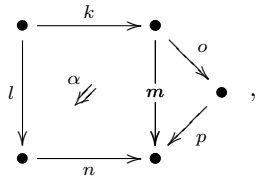
$$jf \xRightarrow{j\alpha} jhg \xRightarrow{\beta g} ig \xRightarrow{i\delta} ikn \xRightarrow{\gamma n} mln \tag{2.4}$$

Note that the same diagram can be usually pasted in many different ways. In the case above, the following is an alternative order of pasting the 2-cells:

$$jf \xRightarrow{j\alpha} jhg \xRightarrow{jh\delta} jhkn \xRightarrow{\beta kn} ikn \xRightarrow{\gamma n} mln \tag{2.5}$$

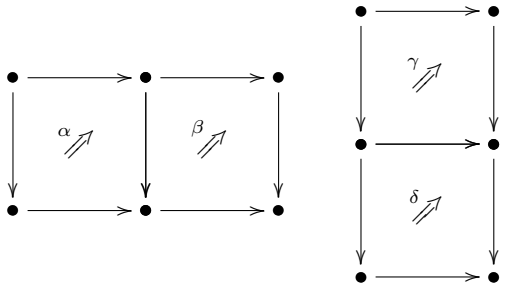
However, (2.4) = (2.5) by associativity of vertical and horizontal composition, and interchange. This doesn't mean that an arbitrary diagram pastes to give a single 2-cell, however, if it does, it does so in a unique way [Pow90].

We adhere to the convention of denoting identity 2-cells by *empty space* in diagrams, such as in



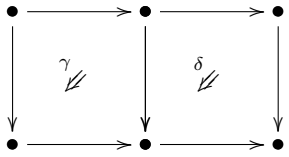
which means just that $m = po$, and so the diagram denotes a 2-cell $pok = mk \xRightarrow{\alpha} nl$.

Among all pasting situations the following two are of special importance:



In this situation, the 2-cell on the left is denoted $\beta \boxplus \alpha$ and the operation is called

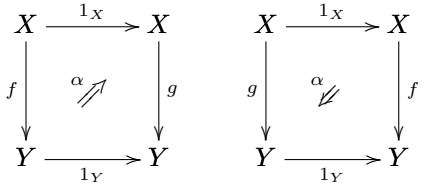
horizontal pasting. The 2-cell on the right is denoted $\delta \boxminus \gamma$ (note the order of δ and γ) and the operation is called *vertical pasting*. The orientation of the drawing is not important, so $\delta \boxminus \gamma$ can be equivalently pictured as



It follows immediately from geometrical considerations that

$$(\beta \boxplus \alpha) \boxminus (\delta \boxplus \gamma) \quad = \quad (\beta \boxminus \delta) \boxplus (\alpha \boxminus \gamma)$$

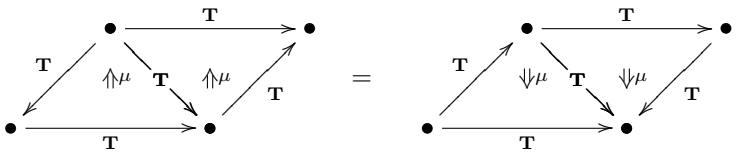
Remark 2.2.4. Note that for the purposes of pasting, a 2-cell $X \xrightarrow[\Downarrow\alpha]{f \atop g} Y$ can be considered as either of the following squares.



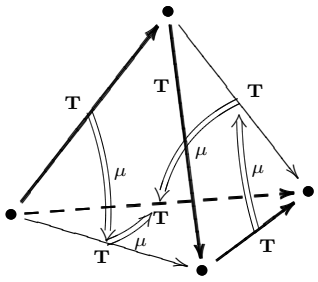
Now, in analogy to the ordinary case, any diagram of 2-cells can be proclaimed (required) to commute. More formally, for a diagram of 2-cells to commute means that for any two objects, A, B, and two composable paths, f, g, of arrows from A to B, an arbitrary pair of 2-cells from f to g obtained by pasting is equal. This necessarily leads to three-dimensional diagrams, which we draw with joy whenever it is sensible and enlightening. The formal details can be found in [Pow90]. Here we give just an example of how exactly the following equation

$$\mu \cdot \mathbf{T}\mu \;=\; \mu \cdot \mu\mathbf{T} \tag{2.6}$$

corresponds to the 2-categorical pasting situation in Fig. 2.3. To this end, observe that commutativity of (2.7) can be equivalently written as an equality of the two 2-cells arising from splitting its surface along the bold path.



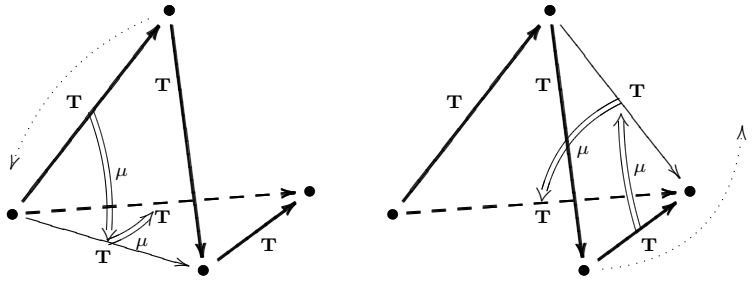
(2.8)



(2.7)

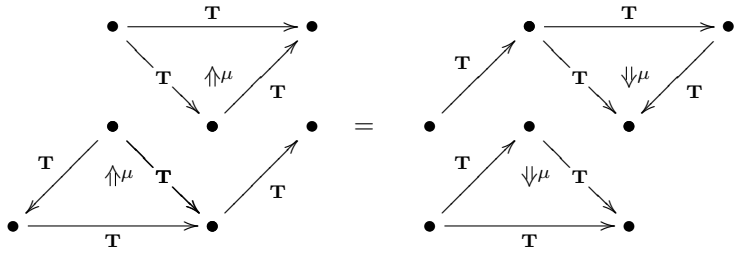
Figure 2.3: Coherence for the multiplication of a monad

In detail, first split (2.7) along the bold path:



(2.9)

Now, by straightening the surface as indicated one obtains (2.8). To see that (2.8) is actually (2.6) by pasting, one has to split the parallelogram along the diagonal arrow, $\mathbf{T}$, and paste the two halves together. Here, pasting corresponds to occurrences of $\cdot$ in (2.6).



(2.10)

Note the identity on $\mathbf{T}$ hanging on either side of μ in the diagram above. This is necessary to make the types of the top and bottom halves match: the type of the boundary in each case is $\mathbf{TT}$.

2.2.3 2-Functors, 2-natural transformations and modifications

In the previous section, 2-categories are introduced as categories where homsets are categories. This paves the way to straightforward definitions of functors on 2-categories: 2-functors, and natural transformations on them: 2-natural transformations.

Definition 2.2.5 (2-Functor)**.** *For 2-categories $\mathscr{C}$ and $\mathscr{D}$, a 2-functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$ is given by the following:*

1. *a function $|\mathbf{F}| : |\mathscr{C}| \longrightarrow |\mathscr{D}|$. We usually write just $\mathbf{F}$ for this function when there's no danger of confusion.*
2. *for each pair $A, B \in |\mathscr{C}|$, an ordinary functor $\mathbf{F}_{A,B} : \mathscr{C}(A, B) \longrightarrow \mathscr{D}(\mathbf{F}A, \mathbf{F}B)$*

such that the following coherence conditions hold, where composition and unit in $\mathscr{D}$ are d and v, respectively (Figure 2.4):

$$\begin{aligned} d_{FA,FB,FC} \cdot (\mathbf{F}_{A,B} \times \mathbf{F}_{B,C}) &= \mathbf{F}_{A,C} \cdot c_{A,B,C} \\ \mathbf{F}_{A,A} \cdot u_A &= v_{FA} \end{aligned}$$

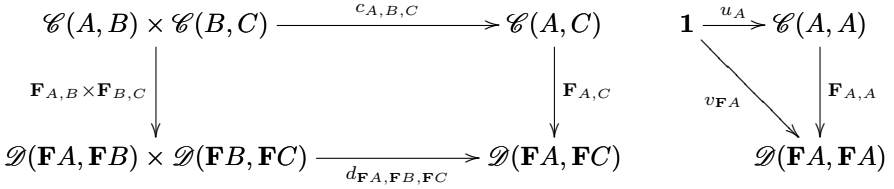


Figure 2.4: Coherence of 2-functor $\mathbf{F} : (\mathscr{C}, c, u) \longrightarrow (\mathscr{D}, d, v)$

We often write just $\mathbf{F}$ instead of $\mathbf{F}_{A,B}$. It's immediate that $\mathbf{F}$ defines a functor on the underlying categories denoted $\mathbf{F}_0 : \mathscr{C}_0 \longrightarrow \mathscr{D}_0$. In order to introduce natural transformations on 2-functors, the homset notation, $\mathscr{C}(A, B)$, must be extended from a mere syntactical notation to a proper functor as follows:

Definition 2.2.6 (Homfunctors)**.** *For an object in $f \in \mathscr{C}(A, B)$, the functor $\mathscr{C}(X, f) : \mathscr{C}(X, A) \longrightarrow \mathscr{C}(X, B)$ is defined by*

$$\begin{aligned} \mathscr{C}(X, f)(g) &\equiv f \circ g \\ \mathscr{C}(X, f)(\alpha) &\equiv 1_f \circ \alpha \end{aligned}$$

Similarly for $\mathscr{C}(f, Y) : \mathscr{C}(B, Y) \longrightarrow \mathscr{C}(A, Y)$:

$$\begin{aligned} \mathscr{C}(f, Y)(g) &\equiv g \circ f \\ \mathscr{C}(f, Y)(\alpha) &\equiv \alpha \circ 1_f \end{aligned}$$

That this defines functors is easy to verify.

The definition of 2-natural transformation is now rather compact.

Definition 2.2.7 (2-Natural transformation)**.** *For 2-functors* $\mathbf{F}, \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$*, a* 2-natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ *is given by:*

- *for each* $A \in |\mathscr{C}|$ *a 1-cell* $\theta_A : \mathbf{F}A \longrightarrow \mathbf{G}A$ *of* $\mathscr{D}$ *such that for each pair* $A, B \in |\mathscr{C}|$*, the following holds:*

$$\begin{array}{ccc} \mathscr{C}(A,B) & \xrightarrow{\mathbf{G}_{A,B}} & \mathscr{D}(\mathbf{G}A, \mathbf{G}B) \\ \downarrow{\scriptstyle \mathbf{F}_{A,B}} & & \downarrow{\scriptstyle \mathscr{D}(\theta_A, 1_{\mathbf{G}B})} \\ \mathscr{D}(\mathbf{F}A, \mathbf{F}B) & \xrightarrow[\mathscr{D}(1_{\mathbf{F}A}, \theta_B)]{} & \mathscr{D}(\mathbf{F}A, \mathbf{G}B) \end{array} \qquad \mathscr{D}(1_{\mathbf{F}A}, \theta_B) \cdot \mathbf{F}_{A,B} \;=\; \mathscr{D}(\theta_A, 1_{\mathbf{G}B}) \cdot \mathbf{G}_{A,B} \tag{2.11}$$

The following observation characterises 2-natural transformations in more elementary terms.

Observation 2.2.8. A 2-natural transformation θ as above is equivalent to the following:

1. θ is an ordinary natural transformation on the underlying functors $\mathbf{F}_0, \mathbf{G}_0 : \mathscr{C}_0 \longrightarrow \mathscr{D}_0$. To see this, consider an $f \in \mathscr{C}(A, B)$ in the upper left corner in (2.11). Going down and right takes it to $\theta_B \circ \mathbf{F}f$ while going right and down takes it to $\mathbf{G}f \circ \theta_A$. Commtativity of (2.11) therefore gives commutativity of the following diagram in $\mathscr{D}_0$:

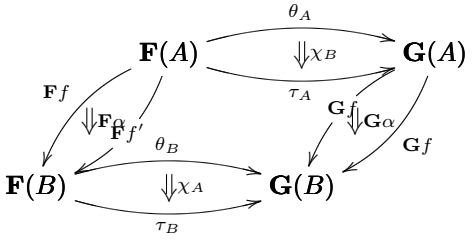
$$\begin{array}{ccc} \mathbf{F}A & \xrightarrow{\theta_A} & \mathbf{G}A \\ \downarrow{\scriptstyle \mathbf{F}f} & & \downarrow{\scriptstyle \mathbf{G}f} \\ \mathbf{F}B & \xrightarrow[\theta_B]{} & \mathbf{G}B \end{array}$$

2. Similarly, for a 2-cell $\alpha : f \Longrightarrow g : A \longrightarrow B$, the bottom route gives $\theta_B \circ \mathbf{F}\alpha$; the top route gives $\mathbf{G}\alpha \circ \theta_A$:

$$\begin{array}{ccc} \mathbf{F}A & \xrightarrow{\theta_A} & \mathbf{G}A \\ {\scriptstyle \mathbf{F}f}\searrow \Downarrow{\scriptstyle \mathbf{F}\alpha} \swarrow{\scriptstyle \mathbf{F}g} & & {\scriptstyle \mathbf{G}f}\searrow \Downarrow{\scriptstyle \mathbf{G}\alpha} \swarrow{\scriptstyle \mathbf{G}g} \\ \mathbf{F}B & \xrightarrow[\theta_B]{} & \mathbf{G}B \end{array} \qquad \theta_B \circ \mathbf{F}\alpha \;=\; \mathbf{G}\alpha \circ \theta_A \tag{2.12}$$

So in elementary terms, a 2-natural transformation is a natural transformation on the underlying functors that in addition satisfies a 2-naturality condition (2.12) with respect to 2-cells. It is straightforward to define vertical and horizontal composition of 2-natural transformations (Figures 2.5 and 2.6) and to prove this data defines a 2-category. Moreover, one can define morphisms of 2-natural transformations – *modifications*.

Definition 2.2.9 (Modification)**.** *Let $\theta, \tau : \mathbf{F} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ be 2-natural transformations. A modification $\chi : \theta \Longrightarrow \tau$ is given by a collection of 2-cells of $\mathscr{D}$, $\chi_A : \theta_A \Longrightarrow \tau_A$ such that for every 2-cell $A \overset{f}{\underset{f'}{\Downarrow \alpha}} B$ in $\mathscr{C}$ the following equality holds in $\mathscr{D}$:*



$$\chi_A \circ \mathbf{F}\alpha \;=\; \mathbf{G}\alpha \circ \chi_B \tag{2.13}$$

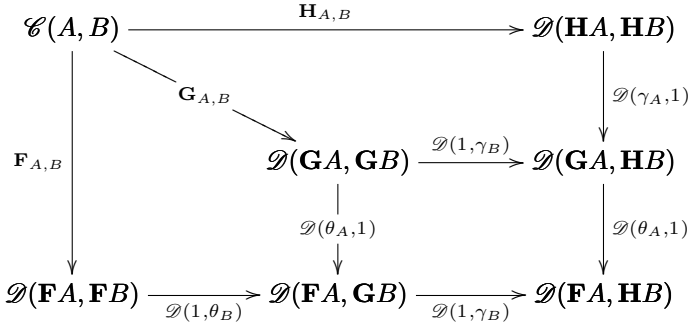


Figure 2.5: Vertical composition of 2-natural transformations

2.2.4 Duality

For an ordinary category, $\mathscr{C}$, there is an opposite category, $\mathscr{C}^{\mathsf{op}}$, such that objects of $\mathscr{C}^{\mathsf{op}}$ are objects of $\mathscr{C}$ and there is an arrow $f^{\mathsf{op}} : X \longrightarrow Y$ of $\mathscr{C}^{\mathsf{op}}$ for each arrow $f : Y \longrightarrow X$ of $\mathscr{C}$. An ordinary functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$ is by definition the same thing as a functor $\mathbf{F}^{\mathsf{op}} : \mathscr{C}^{\mathsf{op}} \longrightarrow \mathscr{D}^{\mathsf{op}}$ and a natural transformation $\alpha : \mathbf{F} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ is by definition a natural transformation $\alpha^{\mathsf{op}} : \mathbf{G}^{\mathsf{op}} \Longrightarrow \mathbf{F}^{\mathsf{op}} : \mathscr{C}^{\mathsf{op}} \longrightarrow \mathscr{D}^{\mathsf{op}}$ because

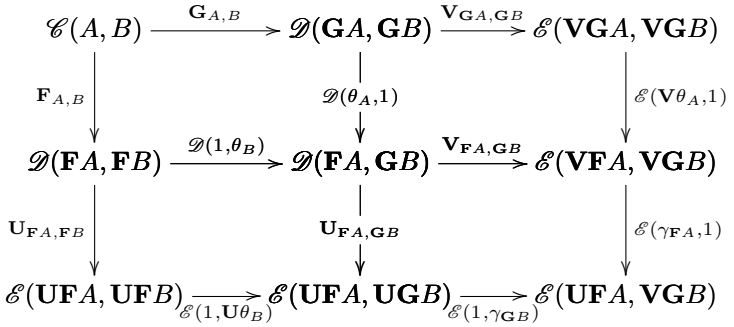
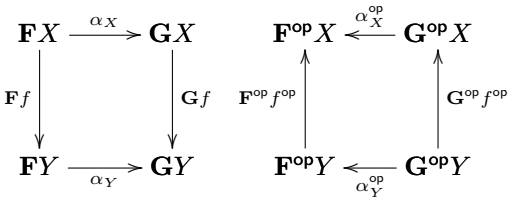
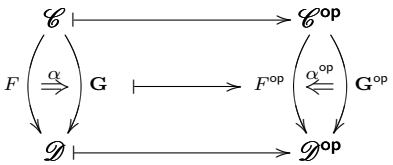


Figure 2.6: Horizontal composition of 2-natural transformations

the following two commuting squares are just different denotations of the same data:



It's easy to check that this defines an ordinary endo-isomorphism $\mathsf{op} : \mathsf{Cat} \longrightarrow \mathsf{Cat}$.



These considerations apply to 2-categories in two ways as follows. For an arbitrary 2-category $\mathscr{C}$, one can define two derived categories. Both categories have the same objects but differ in hom-categories.

Definition 2.2.10 (Horizontally opposite category)**.** *The category $\mathscr{C}^{\mathsf{op}}$ is defined by:*

$$\begin{aligned} |\mathscr{C}^{\mathsf{op}}| &=_{\mathrm{def}} |\mathscr{C}| \\ \mathscr{C}^{\mathsf{op}}(A,B) &=_{\mathrm{def}} \mathscr{C}(B,A) \ , \end{aligned}$$

and the rest follows straightforwardly.

So the category $\mathscr{C}^{\mathsf{op}}$ has reversed 1-cells but its 2-cells have the same directions w.r.t. $\mathscr{C}$. One can also reverse the 2-cells of a 2-category as follows.

Definition 2.2.11 (Vertically opposite category)**.** *The category $\mathscr{C}^{\mathsf{co}}$ is defined by:*

$$\begin{aligned} |\mathscr{C}^{\mathsf{co}}| &=_{\mathrm{def}} |\mathscr{C}| \\ \mathscr{C}^{\mathsf{co}}(A,B) &=_{\mathrm{def}} \mathscr{C}(A,B)^{\mathsf{op}} \ , \end{aligned}$$

and the rest follows straightforwardly.

So $\mathscr{C}^{\mathsf{co}}$ has reversed or 2-cells, but 1-cells are the same. The following then follows directly from the definitions.

Lemma 2.2.12.

1. $\mathscr{A}^{\mathsf{opop}} \equiv \mathscr{A}$
2. $\mathscr{A}^{\mathsf{coco}} \equiv \mathscr{A}$
3. $\mathscr{A}^{\mathsf{coop}} \equiv \mathscr{A}^{\mathsf{opco}}$

The effect the opposite-forming operations have on 2-functors and modifications is summarised at the end of the next section after we have discussed their categories.

2.2.5 The 3-category of 2-categories

Fix a pair of 2-categories, $\mathscr{C}$, $\mathscr{D}$. The 2-functors from $\mathscr{C}$ to $\mathscr{D}$, 2-natural transformations and modifications form a 2-category, $\mathscr{D}^{\mathscr{C}}$. Moreover, for any three 2-categories, $\mathscr{C}$, $\mathscr{D}$, $\mathscr{E}$, composition of 2-functors and horizontal composition of 2-natural transformations define a 2-functor $c_{\mathscr{C}\mathscr{D}\mathscr{E}} : \mathscr{D}^{\mathscr{C}} \times \mathscr{E}^{\mathscr{D}} \longrightarrow \mathscr{E}^{\mathscr{C}}$, which satisfies the diagrams in Fig. 2.1, appropriately renamed. This means formally that 2-categories, 2-functors, 2-natural transformations and modifications form a 3-category, denoted $\mathsf{2Cat}$. The formal definition of a 3-category follows:

Definition 2.2.13 (3-Category)**.** *A 3-category $\mathscr{C}$ consists of*

1. *a class $|\mathscr{C}|$ of* objects
2. *for each pair $A, B \in |\mathscr{C}|$ a 2-category $\mathscr{C}(A,B)$*
3. *for each triple $A, B, C \in |\mathscr{C}|$ a 2-bifunctor $c_{A,B,C} : \mathscr{C}(A,B) \times \mathscr{C}(B,C) \longrightarrow \mathscr{C}(A,C)$*
4. *for each $A \in |\mathscr{C}|$ a 2-functor $u_A : 1 \longrightarrow \mathscr{C}(A,A)$*

This data is required to satisfy the coherence conditions in Figure 2.1 where the diagrams are considered as diagrams of 2-functors.

Remark 2.2.14. Clearly, there is a common theme going on between Definitions 2.2.1 and 2.2.13, which leads to the, now obvious, definition of an n-category. Moreover, there is an even more general notion of a $\mathscr{V}$-category, for a monoidal category $\mathscr{V}$, defined as a set, $|\mathscr{C}|$, together with an assignment of

1. a hom-set-like object $\mathscr{C}(X, Y) \in \mathscr{V}$ to each pair of objects $X, Y \in |\mathscr{C}|$
2. an object $I \in \mathscr{V}$ and an arrow $u : I \longrightarrow \mathscr{C}(X, X)$ for all $X \in \mathscr{V}$
3. for each triple X, Y, Z, an arrow c of $\mathscr{V}$ of type $\mathscr{C}(X, Y) \otimes \mathscr{C}(Y, Z) \longrightarrow \mathscr{C}(X, Z)$, where $\otimes$ is the tensor of $\mathscr{V}$

This data must satisfy conditions similar to those of a 2-category that ensure that u is the unit of composition c, and composition is associative.

Enriched categories generalise n-categories, and many other more and less similar notions. The theory of enriched categories, and the associated notions, have been studied extensively. Most notably, G. M. Kelly in [Kel82] gives a comprehensive introduction to Enriched Category Theory.

Finally, note that the 3-category $\mathsf{2Cat}$ is cartesian closed with the closed structure given by:

terminal object: the one-object 2-category, 1.

binary product: the 2-category $\mathscr{A} \times \mathscr{B}$ for any two 2-categories $\mathscr{A}$ and $\mathscr{B}$. It is the category of pairs of 0-cells, 1-cells and 2-cells with the obvious first and second projections. This forms a binary product in $\mathsf{2Cat}$.

exponential: given by the right adjoint in $(-) \times \mathscr{A} \dashv (-)^{\mathscr{A}}$ for every $\mathscr{A}$, where the counit $\epsilon : \mathscr{B}^{\mathscr{A}} \times \mathscr{A} \longrightarrow \mathscr{B}$ is called *evaluation*.

The following lemma summarises how the exponential interacts with the dualities.

Lemma 2.2.15.

1. $(\mathscr{B}^{\mathscr{A}}) \equiv \left((\mathscr{B}^{\mathsf{op}})^{\mathscr{A}^{\mathsf{op}}}\right)^{\mathsf{op}}$
2. $(\mathscr{B}^{\mathscr{A}}) \equiv \left((\mathscr{B}^{\mathsf{co}})^{\mathscr{A}^{\mathsf{co}}}\right)^{\mathsf{co}}$

Proof. Just expand the definitions. □

2.2.6 Adjunction

Adjuncions are ordinarily defined for functors and natural transformations. Here, we describe the notion of adjunction w.r.t. an arbitrary 2-category, $\mathscr{K}$. It arises simply by spelling out the usual definition in terms of the objects, arrows and 2-cells of, $\mathscr{K}$, in place of Cat.

Definition 2.2.16 (Adjunction)**.** *In a 2-category $\mathscr{K}$, 1-cells $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$ are said to be* adjoint *iff there are 2-cells $\eta : 1 \Longrightarrow gf$ and $\epsilon : fg \Longrightarrow 1$ that satisfy (see Fig. 2.7):*

$$\begin{aligned} \epsilon f \cdot f\eta &= 1_f \\ g\epsilon \cdot \eta g &= 1_g \end{aligned}$$

Any of

$$\begin{gathered} f \dashv g \\ (f, g, \eta, \epsilon) \\ f : X \rightleftarrows Y : g \end{gathered}$$

is used as a notation for the same adjunction.

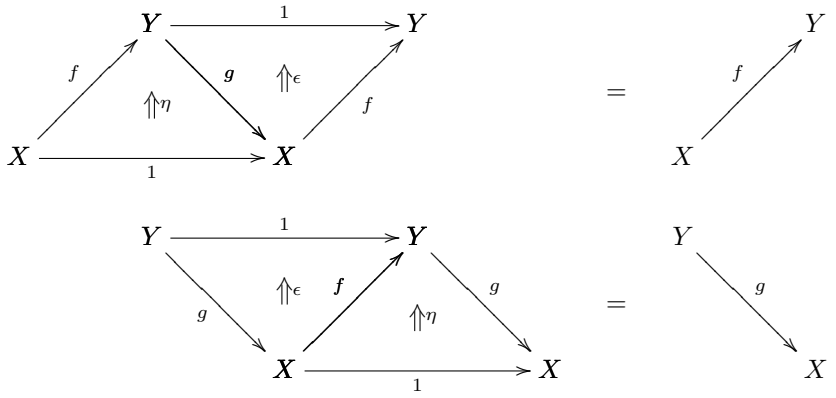


Figure 2.7: The triangle identities in a 2-category

Definition 2.2.16 makes it possible to generalise all other ordinary notions defined by an adjunction to an arbitrary base 2-category $\mathscr{K}$. Before we investigate products and coproducts arising in this way in detail, we mention the following simple but useful fact. Its proof is immediate from Fig. 2.7.

Theorem 2.2.17 (Adjunctions and duality)**.** *The following are equivalent in a 2-category $\mathscr{K}$:*

1. *(f, g, η, ϵ) is an adjunction in $\mathscr{K}$*
2. *(g, f, η, ϵ) is an adjunction in $\mathscr{K}^{\mathsf{op}}$*
3. *(g, f, ϵ, η) is an adjunction in $\mathscr{K}^{\mathsf{co}}$.*

2.2.7 Products

This section works out the details and notation of a notion of products on an object of an arbitrary 2-category, $\mathscr{K}$, made possible by the notion of adjunction in an arbitrary 2-category.

We start by fixing some notation. In a 2-category $\mathscr{K}$, an object $X \times Y$ is a *product* of objects X and Y iff there is for any Z in $\mathscr{K}$ a natural isomorphism of hom-categories

$$\mathscr{K}(Z, X) \times \mathscr{K}(Z, Y) \;\cong\; \mathscr{K}(Z, X \times Y) \;, \tag{2.14}$$

natural in Z. The left-to-right direction of (2.14) is denoted $(\triangle)$ and written as an infix operator. The right to left direction of (2.14) is given by horizontal post-composition with projections $\pi_1 : X \times Y \longrightarrow X$ and $\pi_2 : X \times Y \longrightarrow Y$. This is summarised in Fig. 2.8.

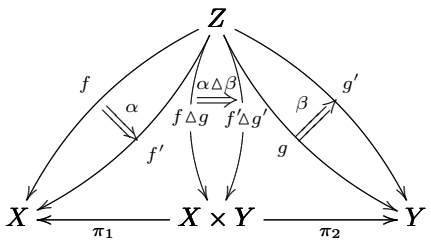


Figure 2.8: Product $X \times Y$ of X, Y, in $\mathscr{K}$

On ordinary category, $\mathscr{C}$, has products iff there is a right adjoint, $(\times) : \mathscr{C} \times \mathscr{C} \longrightarrow \mathscr{C}$, to the doubling functor $\Delta : \mathscr{C} \longrightarrow \mathscr{C} \times \mathscr{C}$. Formally, when $X, Y : 1 \longrightarrow \mathscr{C}$ are objects in $\mathscr{C}$, $X \times Y$ is defined as:

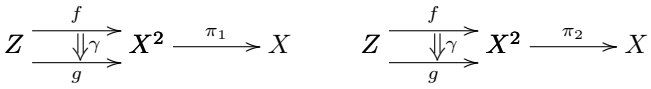
$$1 \xrightarrow{X \triangle Y} \mathscr{C} \times \mathscr{C} \xrightarrow{(\times)} \mathscr{C}$$

The functor Δ is usually defined by its action on objects and arrows as $\Delta(X) =_{\mathrm{def}} (X, X)$ and $\Delta(f) =_{\mathrm{def}} (f, f)$, but equivalently it is just $\Delta =_{\mathrm{def}} (1_{\mathscr{C}} \triangle 1_{\mathscr{C}})$ in Cat. This allows us to generalise the notion of products *in a category* to products *on an object* in $\mathscr{K}$ as follows.

Definition 2.2.18 (Product on an object of a 2-category)**.** *In an arbitrary 2-category $\mathscr{K}$, with object $X \in \mathscr{K}$, such that there exists a product $X \times X \in \mathscr{K}$:*

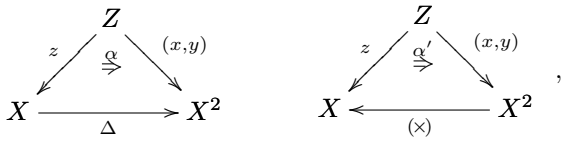
1. *The arrow $1_X \triangle 1_X : X \longrightarrow X \times X$ is denoted Δ_X.*
2. *It is said that X* has products *iff there is an adjunction $\Delta_X \dashv (\times)_X$ in $\mathscr{K}$*

In the following text, the notation X^2 is often used instead of $X \times X$. By definition, the unit and counit in the adjunction in $\mathscr{K}$ above are just 2-cells of $\mathscr{K}$ of types $\eta : 1_X \Longrightarrow (\times) \circ \Delta : X \longrightarrow X$ and $\epsilon : \Delta \circ (\times) \Longrightarrow 1_X : X \times X \longrightarrow X \times X$ that satisfy the triangle identities. By (2.14) for X^2, an arbitrary 2-cell $\gamma : f \Longrightarrow g : Z \longrightarrow X^2$ is isomorphic to the pair of 2-cells:

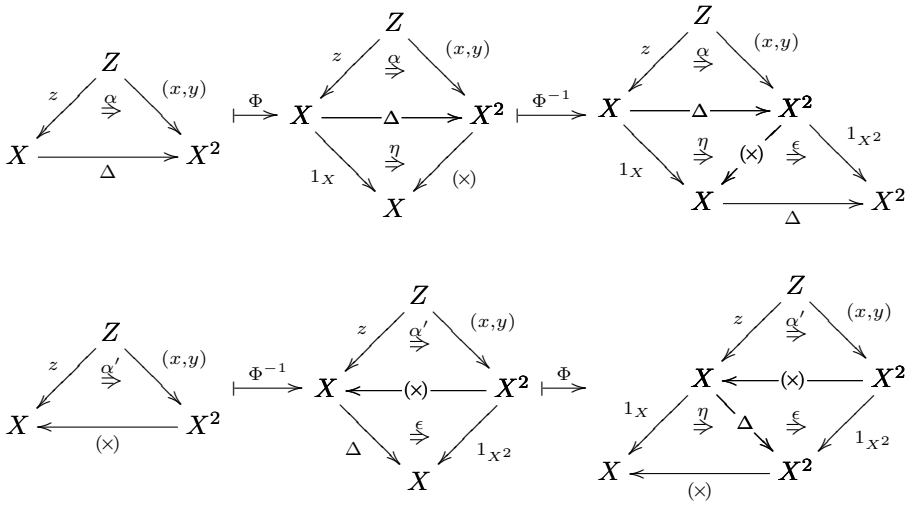


In the special case of ϵ, $\pi_1 \circ \epsilon$ is denoted ${\pi_1}^X$, or just π_1 when there is no danger of confusion. Similarly for $\pi_2 \circ \epsilon$ and ${\pi_2}^X$.

A product on X in $\mathscr{K}$ defines an isomorphism, Φ, of 2-cells of types $\alpha : \Delta \circ z \Longrightarrow (x, y)$ and $\alpha' : z \Longrightarrow (\times) \circ (x, y)$



which is given by pasting with the unit, η, and counit, ϵ, of the adjunction as follows:



That these are indeed inverse follows from the triangle identities. More commonly, $\Delta \circ z$ is written (z, z) and $(\times) \circ (x, y)$ is written $x \times y$, and so we have the familiar isomorphism

$$\frac{(z, z) \Longrightarrow (x, y)}{z \Longrightarrow x \times y \ .} \tag{2.15}$$

This justifies the use of the pointwise notation for general objects with products regardless of whether the objects are internally defined as sets.

2.2.8 Coproducts

The considerations of the previous section about products dualise for coproducts. In particular, the following definition is a dual of Def. 2.2.18.

Definition 2.2.19 (Coproduct on an object of a 2-category)**.** *In an arbitrary category $\mathscr{K}$, with object $X \in \mathscr{K}$, such that there exists a product, $X \times X$, it is said that an X,* has coproducts *iff there is an adjunction $\oplus_X \dashv \Delta_X$.*

By Theorem 2.2.17, X in $\mathscr{K}$ has products iff X in $\mathscr{K}^{\mathsf{co}}$ has coproducts.

2.3 Lax natural transformations and functor categories

2.3.1 Lax natural transformations

The notion of a 2-natural transformation (Def. 2.2.7) is too strict for many applications. Whereas a 2-functor is a collection of ordinary functors, a 2-natural transformation is *not* a collection of ordinary natural transformations, it is just a collection of 1-cells required to commute strictly (Fig. 2.9). It seems more natural for a notion of a natural

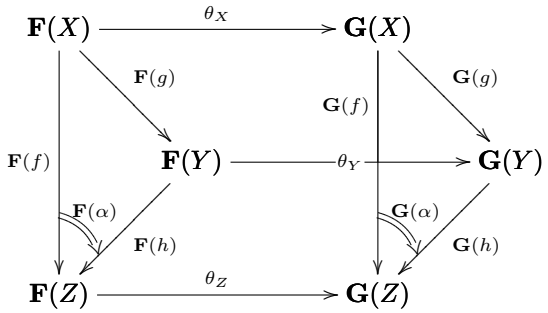


Figure 2.9: A 2-natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$

transformation of 2-functors to be defined in terms of 1-cells *and* 2-cells which are coherent with $\mathbf{F}$ and $\mathbf{G}$ rather than to rely on 1-cells to do the whole job. Indeed, there is a notion of a natural transformation of 2-functors which fits the 2-categorical setting better and of which a 2-natural transformation is a special case.

Definition 2.3.1 (Lax natural transformation)**.** *For 2-functors $\mathbf{F}, \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$, a* lax natural transformation *$\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ is defined by the following data (Fig. 2.10):*

1. *for each $A \in |\mathscr{C}|$ a 1-cell $\theta_A : \mathbf{F}A \longrightarrow \mathbf{G}A$ of $\mathscr{D}$*

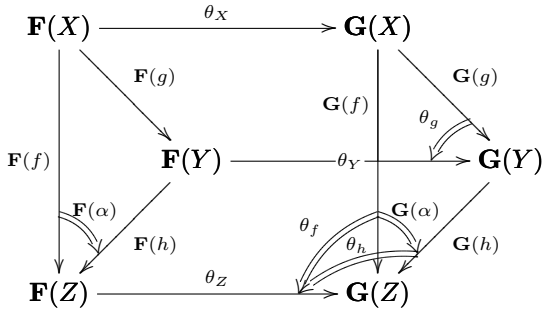


Figure 2.10: A lax natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$

2. *for each pair $A, B \in |\mathscr{C}|$, an ordinary natural transformation*

$$\begin{array}{ccc} \mathscr{C}(A,B) & \xrightarrow{\mathbf{G}_{A,B}} & \mathscr{D}(\mathbf{G}A, \mathbf{G}B) \\ {\scriptstyle \mathbf{F}_{A,B}}\downarrow & \swarrow{\scriptstyle \theta_{A,B}} & \downarrow{\scriptstyle \mathscr{D}(\theta_A, 1_{\mathbf{G}B})} \\ \mathscr{D}(\mathbf{F}A, \mathbf{F}B) & \xrightarrow[\mathscr{D}(1_{\mathbf{F}A}, \theta_B)]{} & \mathscr{D}(\mathbf{F}A, \mathbf{G}B) \end{array} \tag{2.16}$$

$$\theta_{A,B} \,:\, \mathscr{D}(\theta_A, 1_{\mathbf{G}B}) \cdot \mathbf{G}_{A,B} \Longrightarrow \mathscr{D}(1_{\mathbf{F}A}, \theta_B) \cdot \mathbf{F}_{A,B} \ ,$$

such that the components $(\theta_{A,B})_f$

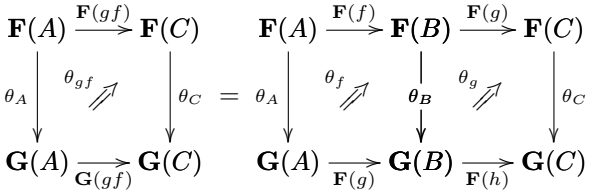
$$\begin{array}{ccc} \mathbf{F}(A) & \xrightarrow{\theta_A} & \mathbf{G}(A) \\ {\scriptstyle \mathbf{F}(f)}\downarrow & \swarrow{\scriptstyle (\theta_{A,B})_f} & \downarrow{\scriptstyle \mathbf{G}(f)} \\ \mathbf{F}(B) & \xrightarrow[\theta_B]{} & \mathbf{G}(B) \end{array} \tag{2.17}$$

in addition satisfy the following conditions (where $(\theta_{A,B})_f$ is denoted just θ_f):

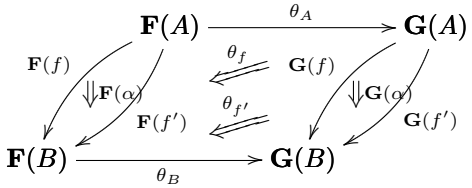
(a) $\theta_{1_A} = 1_{\theta_A}$

$$\begin{array}{ccc} \mathbf{F}(A) & \xrightarrow{\theta_A} & \mathbf{G}(A) \\ {\scriptstyle \mathbf{F}(1)}\downarrow & (\theta_{1_A}) & \downarrow{\scriptstyle \mathbf{G}(1)} \\ \mathbf{F}(A) & \xrightarrow[\theta_A]{} & \mathbf{G}(A) \end{array}$$

(b) for $f : A \longrightarrow B$, $g : B \longrightarrow C$, $\theta_{gf} = \theta_g \boxplus \theta_f$



Note that the naturality of (2.16) amounts to the following coherence condition of the components θ_f and $\theta_{f'}$ with 2-cells $\begin{array}{ccc} A & \xrightarrow{1_A} & A \\ {\scriptstyle f'}\downarrow & \overset{\alpha}{\swarrow} & \downarrow{\scriptstyle f} \\ B & \xrightarrow[1_B]{} & B \end{array}$:



$$\theta_f \boxminus \mathbf{F}(\alpha) = \mathbf{G}(\alpha) \boxminus \theta_{f'} \tag{2.18}$$

The above Definition 2.3.1 is very succinct and economical and we leave it as a useful exercise for the reader to work out the details and verify that the components of $\theta_{A,B}$ in (2.16) are indeed 2-cells (2.20) and that the naturality of (2.16) is equivalent to the commutativity of (2.18).

Notation 2.3.2. In the following text we use the following conventions:

- 2-cells $\alpha : fg \Longrightarrow hi$ for 1-cells f, g, h, i of the appropriate types will be called *laxly commuting squares*. Note however that there is no *commutativity* condition as such, it is merely a 2-cell which can be pictured as a square with a diagonal 2-cell. An example is (2.20).
- a "natural transformation" always means a "lax natural transformation", whereas 2-natural transformations are also called strict natural transformations of 2-functors.

The definitions of horizontal and vertical composition for lax natural transformations (Figs 2.11 and 2.12) relax straightforwardly the strict version (Figs 2.5 and 2.6). Note that a strict 2-natural transformation Def. 2.2.7 is clearly an instance of Def. 2.3.1.

Remark 2.3.3. It is also possible to relax the definition of 2-functors by relaxing the composition and unit axioms from equality to natural transformations ($\mathbf{F}g \circ \mathbf{F}f \Rightarrow \mathbf{F}(g \circ f)$ and $1 \Rightarrow \mathbf{F}1$).

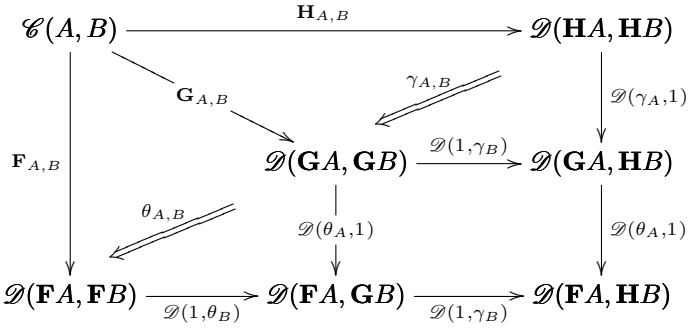


Figure 2.11: Vertical composition of lax natural transformations $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ and $\gamma : \mathbf{G} \Longrightarrow \mathbf{H}$

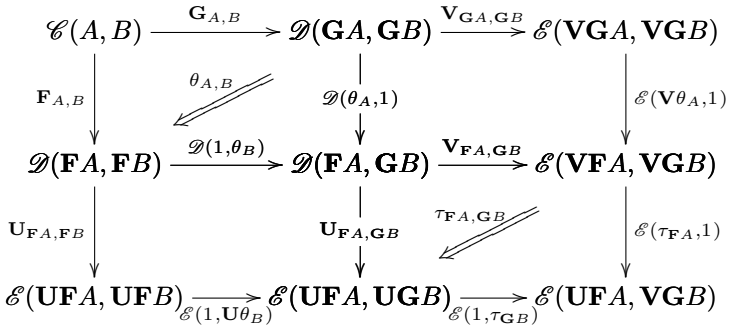


Figure 2.12: Horizontal composition of lax natural transformations $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ and $\tau : \mathbf{U} \Longrightarrow \mathbf{V}$

Reversing the direction of the family (2.16) gives rise to an alternative notion of lax natural transformations which is called *forward*, whereas the one in Def. 2.3.1 is called *reverse*. To remember which is which note that this nomenclature reflects the relative direction of the 2-components θ_f with respect to the 1-components θ_A, θ_B (see 2.17 and 2.20). For completeness, a formal definition of a forward lax natural transformations follows.

Definition 2.3.4 (Forward lax natural transformation)**.** *For 2-functors $\mathbf{F}, \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$, a* forward lax natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ *is defined by the following data:*

1. *for each $A \in |\mathscr{C}|$ a 1-cell $\theta_A : \mathbf{F}A \longrightarrow \mathbf{G}A$ of $\mathscr{D}$*

2. *for each pair $A, B \in |\mathscr{C}|$, there is an ordinary natural transformation*

$$\begin{array}{ccc} \mathscr{C}(A,B) & \xrightarrow{\mathbf{G}_{A,B}} & \mathscr{D}(\mathbf{G}A,\mathbf{G}B) \\ {\scriptstyle \mathbf{F}_{A,B}}\downarrow & \overset{\theta_{A,B}}{\swarrow} & \downarrow{\scriptstyle \mathscr{D}(\theta_A,1_{\mathbf{G}B})} \\ \mathscr{D}(\mathbf{F}A,\mathbf{F}B) & \xrightarrow[\mathscr{D}(1_{\mathbf{F}A},\theta_B)]{} & \mathscr{D}(\mathbf{F}A,\mathbf{G}B) \end{array} \tag{2.19}$$

$$\theta_{A,B} \,:\, \mathscr{D}(1_{\mathbf{F}A},\theta_B)\cdot\mathbf{F}_{A,B} \Longrightarrow \mathscr{D}(\theta_A,1_{\mathbf{G}B})\cdot\mathbf{G}_{A,B} \ ,$$

such that the components θ_f, laxly commuting squares

$$\begin{array}{ccc} \mathbf{F}(A) & \xrightarrow{\theta_A} & \mathbf{G}(A) \\ {\scriptstyle \mathbf{F}(f)}\downarrow & \overset{\theta_f}{\swarrow} & \downarrow{\scriptstyle \mathbf{G}(f)} \\ \mathbf{F}(B) & \xrightarrow[\theta_B]{} & \mathbf{G}(B) \end{array} \ , \tag{2.20}$$

satisfy:

(a) $\theta_{1_A} \,=\, 1_{\theta_A}$

(b) for $f : A \longrightarrow B$, $g : B \longrightarrow C$, $\theta_{gf} \,=\, \theta_g \boxminus \theta_f$

By naturality of (2.19), components of $\theta_{A,B}$ satisfy for 2-cells $A \overset{f}{\underset{f'}{\Downarrow\alpha}} B$:

$$\begin{array}{ccccc} & & \mathbf{F}(A) & \xrightarrow{\theta_A} & \mathbf{G}(A) & & \\ & {\scriptstyle \mathbf{F}(f)}\swarrow & \overset{\theta_f}{\Rightarrow} & {\scriptstyle \mathbf{G}(f)}\searrow & & \\ & \Downarrow{\scriptstyle \mathbf{F}(\alpha)} & & \Downarrow{\scriptstyle \mathbf{G}(\alpha)} & & \\ & {\scriptstyle \mathbf{F}(f')}\searrow & \overset{\theta_{f'}}{\Rightarrow} & {\scriptstyle \mathbf{G}(f')}\swarrow & & \\ \mathbf{F}(B) & & \xrightarrow[\theta_B]{} & & \mathbf{G}(B) & & \end{array} \tag{2.21}$$

$$\mathbf{G}(\alpha) \boxplus \theta_f \,=\, \theta_{f'} \boxplus \mathbf{F}(\alpha)$$

Horizontal and vertical composition dualise accordingly.

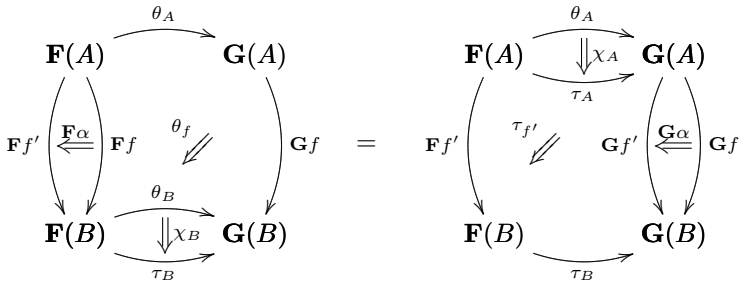
2.3.2 Lax Modifications

The notion of a modification (Def. 2.2.9) is relaxed accordingly.

Definition 2.3.5 (Modification of reverse lax natural transformations)**.** *Let $\theta, \tau : \mathbf{F} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ be reverse lax natural transformations. A modification $\chi : \theta \Rrightarrow \tau$ is given by a collection of 2-cells of $\mathscr{D}$, $\chi_A \,:\, \theta_A \Longrightarrow \tau_A$, such that for every 2-cell*

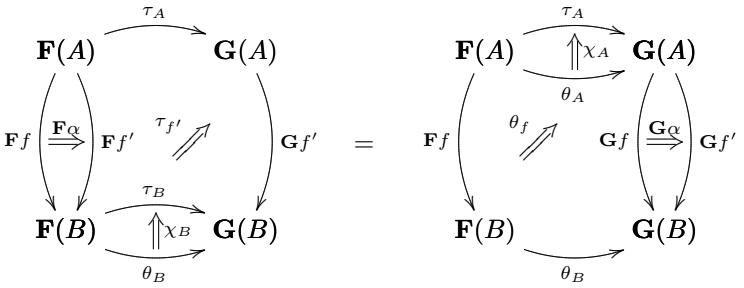
$A \xrightarrow[f']{\overset{f}{\Downarrow\alpha}} B$ *in* $\mathscr{C}$ *the following equality holds in* $\mathscr{D}$*:*

$$\chi_B \boxplus (\theta_f \boxminus \mathbf{F}\alpha) = (\mathbf{G}\alpha \boxminus \tau_{f'}) \boxplus \chi_A \tag{2.22}$$



And the forward variant is obvious, here spelled out for easy reference:

Definition 2.3.6 (Modification of forward lax natural transformations)**.** *Let* $\theta, \tau : \mathbf{F} \longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ *be* forward *lax natural transformations. A modification* $\chi : \theta \longrightarrow \tau$ *is given by a collection of 2-cells of* $\mathscr{D}$*,* $\chi_A : \theta_A \Longrightarrow \tau_A$ *such that for every 2-cell* $A \xrightarrow[f']{\overset{f}{\Downarrow\alpha}} B$ *in* $\mathscr{C}$ *the following equality holds in* $\mathscr{D}$*:*



$$\chi_B \boxminus (\tau_{f'} \boxplus \mathbf{F}\alpha) = (\mathbf{G}\alpha \boxplus \theta_f) \boxminus \chi_A \tag{2.23}$$

The following is an accessible example of a lax modification. In particular note that the components of a modification are 2-cells which must be in some sense coherent with the components of the lax natural transformations the modification is on.

Example 2.3.7. A type constructor in a parametrically typed programming language can be seen as a functor $\mathbf{F} : \star \longrightarrow \star$ where $\star$ is a universe of types, for instance the category of sets. Therefore a type constructor is a an arrow in the 2-category Cat.

1. First, consider the category 1 with just one object, $\mathbf{o}$, and no non-identity 1- and 2-cells. A 2-functor from 1 into Cat picks an object – a category. In particular, let $\underline{\star}$ denote the functor such that $\underline{\star}(\mathbf{o}) = \star$.

A lax natural transformation $\theta : \underline{\star} \longrightarrow \underline{\star}$ has just one 1-component, $\theta_{\mathbf{o}} : \star \longrightarrow \star$, in Cat. This is an ordinary functor.

Consider two type constructors (arrows $\star \longrightarrow \star$ in Cat): $\mathbf{T}$ for binary trees with data in the leaves and $\mathbf{L}$ for lists. For the lax natural transformations τ and λ such that $\tau_{\mathbf{o}} = \mathbf{T}$ and $\lambda_{\mathbf{o}} = \mathbf{L}$, a modification $\tau \longrightarrow \lambda$ is given by a 2-cell $\alpha : \mathbf{T} \Longrightarrow \mathbf{L}$. This is a natural transformation in Cat such as the parametrically polymorphic function `flatten` taking a tree to a list of its leaves in the left-to-right order.

2. Now consider a category, $\mathscr{C}$, generated freely from finite products on one object, $\mathbf{o}$. That is, $\mathscr{C}$ has objects 1, a terminal object, $\mathbf{o}$, $\mathbf{o}^2$, $\mathbf{o}^3$, etc., and the respective projections as 1-cells. In addition let there be a product on $\mathbf{o}$ in $\mathscr{C}$ (see Sect. 2.2.7 for details). The 2-category Cat has products so one can consider a a product preserving 2-functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathsf{Cat}$. The image $\mathbf{F}(\mathbf{o})$ in Cat is a category with products. Set has products so $\underline{\star}$ has an extension from 1 to $\mathscr{C}$, hereby called $\mathbf{S}$.

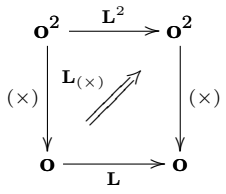
   Do the lax natural transformations λ and τ considered before have an extension to $\mathscr{C}$ as well? Firstly, as there is more structure in $\mathscr{C}$ there is now difference between the forward and reverse notions of lax natural transformations. For the case of λ, we chose the forward direction. By definition Def. 2.3.4, we need the following:

   (a) For each object $X \in \mathscr{C}$ a component $\lambda_X : \mathbf{S}(X) \longrightarrow \mathbf{S}(X)$. But all objects in $\mathscr{C}$ are of the form $\mathbf{o}^i$, for $i \in \mathbb{N}$, and we have that $\mathbf{S}(\mathbf{o}) = \star$ and that $\mathbf{S}$ is product preserving so $\mathbf{S}(\mathbf{o}^i) = \star^i$ and if $\lambda_{\mathbf{o}} = \mathbf{L} : \star \longrightarrow \star$ we have that the component $\lambda_X : \star^i \longrightarrow \star^i$ must be $\mathbf{L}^i$.

   (b) For each arrow $f : X^i \longrightarrow X^j$, λ must have a 2-component

$$\lambda_f : \mathbf{L}^j \cdot \mathbf{S}(f) \Longrightarrow \mathbf{S}(f) \cdot \lambda^i \ .$$

   We can define the component by induction on the structure of f because arrows are generated freely. For a projection π, λ_π is an identity. For a tuple $g \vartriangle h$, $\lambda_{g \vartriangle h} = \lambda_g \vartriangle \lambda_h$, and similarly for composition, etc. The interesting case is the product on $\mathbf{o}$. The nontrivial component is $\lambda_{(\times)}$ on the arrow $(\times) : \mathbf{o}^2 \longrightarrow \mathbf{o}$ which is a part of the definition of products on $\mathbf{o}$ in $\mathscr{C}$. By definition this component has the following type:



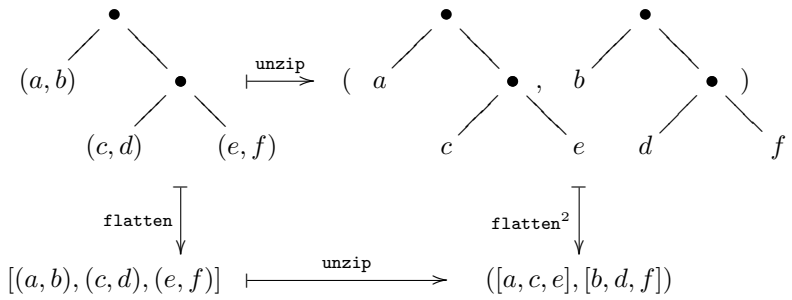
When $\mathbf{L}$ is interpreted as the type constructor of lists, a parametric function of this type is `unzip` taking a list of pairs into a pair of lists of the same length. This function is moreover coherent with the product structure on $\star$ (projections, tupling, etc.) so it fits as the required component $\lambda_{(\times)}$ of a lax natural transformation λ.

The same analysis works for τ and trees represented by the functor $\mathbf{T}$, where `unzip` for trees takes trees where data are pairs to a pair of trees of the same shape.

3. Thus we have two forward lax natural transformations: λ for lists and τ for trees, which are now explicitly coherent with products in $\star$. A modification $\alpha : \mathbf{T} \longrightarrow \mathbf{L}$ is given by a component $\alpha_{\mathbf{o}}$, the rest is implied by the product structure. In order to use `flatten` for this as before, coherence condition (2.23) must be established. Explicitly `flatten` must be shown to be coherent with products and with `unzip`. Coherence with products is trivial as before. To establish coherence with `unzip`, one must establish the following:

$$\texttt{unzip} \,.\, \texttt{flatten} \;=\; \texttt{flatten}^2 \,.\, \texttt{unzip}$$

The following example illustrates this property:



We have therefore established that `flatten` is a (the nontrivial) component of a modification of trees into lists when these are considered as coherent datatypes on a category with products.

2.3.3 2-Categories of functors and lax natural transformations

For a pair of 2-categories, $\mathscr{C}$, $\mathscr{D}$, functors $\mathscr{C} \longrightarrow \mathscr{D}$, lax natural transformations of either kind and the corresponding lax modifications form a 2-category [Gra74]. We introduce the following notation for these 2-categories.

Notation 2.3.8. For arbitrary 2-categories $\mathscr{C}$, $\mathscr{D}$,

1. the 2-category of 2-functors $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$, *forward* lax natural transformations and forward lax modifications is denoted $[\![\mathscr{C}, \mathscr{D}]\!]$.

2. the 2-category of 2-functors $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$, *reverse* lax natural transformations and reverse lax modifications is denoted $[\mathscr{C}, \mathscr{D}]$.

For any fixed $\mathscr{C}$, $\mathscr{D}$, the categories $[\![\mathscr{C}, \mathscr{D}]\!]$ and $[\mathscr{C}, \mathscr{D}]$ are not equivalent. They have the same set of objects, though, which is sometimes denoted $|\mathscr{C}, \mathscr{D}|$. Formally, $|\mathscr{C}, \mathscr{D}| =_{\text{def}} |[\![\mathscr{C}, \mathscr{D}]\!]| = |[\mathscr{C}, \mathscr{D}]|$. However, they are related by the dualities $(-)^{\mathsf{op}}$ and $(-)^{\mathsf{co}}$ as described by the following theorem.

Theorem 2.3.9. *For any 2-categories $\mathscr{C}$, $\mathscr{D}$, the following equivalences hold:*

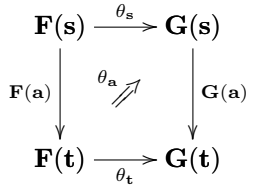
$$[\![\mathscr{C}, \mathscr{D}]\!] \cong [\mathscr{C}^{\mathsf{co}}, \mathscr{D}^{\mathsf{co}}]^{\mathsf{co}} \tag{2.24}$$

$$[\![\mathscr{C}, \mathscr{D}]\!] \cong [\mathscr{C}^{\mathsf{op}}, \mathscr{D}^{\mathsf{op}}]^{\mathsf{op}} \tag{2.25}$$

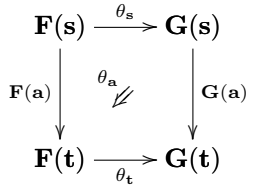
$$[\mathscr{C}, \mathscr{D}] \cong [\![\mathscr{C}^{\mathsf{co}}, \mathscr{D}^{\mathsf{co}}]\!]^{\mathsf{co}} \tag{2.26}$$

$$[\mathscr{C}, \mathscr{D}] \cong [\![\mathscr{C}^{\mathsf{op}}, \mathscr{D}^{\mathsf{op}}]\!]^{\mathsf{op}} \tag{2.27}$$

The proof can be found in [Gra74]. It is a not technically difficult but a bit tedious exposition from the definitions. To understand the intuition behind these equivalences, consider first the case when $\mathscr{C} \equiv 2$, the 2-category with 2 distinct objects, $\mathbf{s}$, $\mathbf{t}$, and one nonidentity arrow, $\mathbf{a}$, between them. Arrows $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$ in $[\![2, \mathscr{D}]\!]$ are laxly commuting squares with a 2-cell of $\mathscr{D}$ going diagonally.



In $[2, \mathscr{D}]$, arrows are laxly commuting squares of arrows of $\mathscr{D}$ with the 2-cell reversed.



So, arrows in $[\![2, \mathscr{D}]\!]$ are just arrows in $[2, \mathscr{D}^{\mathsf{co}}]$. However, when 2 above is replaced by a 2-category, $\mathscr{C}$, with nontrivial 2-cells, $[\![\mathscr{C}, \mathscr{D}]\!] \ncong [\mathscr{C}, \mathscr{D}^{\mathsf{co}}]$ because reversing 2-cells in $\mathscr{D}$ affects not only the direction of the diagonal 2-cell but all 2-cells in the image of $\mathscr{C}$. This can be corrected by reversing 2-cells in $\mathscr{C}$. But still, $[\![\mathscr{C}, \mathscr{D}]\!] \ncong [\mathscr{C}^{\mathsf{co}}, \mathscr{D}^{\mathsf{co}}]$, as 2-cells in this category (modifications) are reversed with respect to $[\![\mathscr{C}, \mathscr{D}]\!]$ because

they are defined in terms of 2-cells of $\mathscr{D}$. This can corrected again by considering $[\mathscr{C}^{\mathsf{co}}, \mathscr{D}^{\mathsf{co}}]^{\mathsf{co}}$, and we are done. This justifies (2.24) and (2.25); (2.26) and (2.27) are equivalent by duality.

As for the cartesian structure of these categories, it holds that each $[\mathscr{C}, \mathscr{D}]$ has products if $\mathscr{D}$ has products. The product is defined point wise as for the strict case. In particular, for a pair of lax natural transformations $\theta : \mathbf{H} \Longrightarrow \mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$, $\gamma : \mathbf{H} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$, the mediating lax natural transformation has components

$$(\theta \vartriangle \gamma)_X \ =_{\mathrm{def}} \ \theta_X \vartriangle \gamma_X \qquad\qquad (\theta \vartriangle \gamma)_f \ =_{\mathrm{def}} \ \theta_f \vartriangle \gamma_f$$

$$\begin{array}{ccc} \mathbf{H}(X) & \xrightarrow{(\theta \vartriangle \gamma)_X} & \mathbf{F}(X) \times \mathbf{G}(X) \\ {\scriptstyle \mathbf{H}(f)}\downarrow & \swarrow\!\!\!\swarrow {\scriptstyle (\theta \vartriangle \gamma)_f} & \downarrow{\scriptstyle \mathbf{F}(f) \times \mathbf{G}(f)} \\ \mathbf{H}(Y) & \xrightarrow[(\theta \vartriangle \gamma)_Y]{} & \mathbf{F}(Y) \times \mathbf{G}(Y) \end{array}$$

The projections $\pi_{1_{\mathbf{F},\mathbf{G}}} : \mathbf{F} \times \mathbf{G} \Longrightarrow \mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$ and $\pi_{2_{\mathbf{F},\mathbf{G}}} : \mathbf{F} \times \mathbf{G} \Longrightarrow \mathbf{G} : \mathscr{C} \longrightarrow \mathscr{D}$ are strict, i.e. $\pi_{1_{\mathbf{F},\mathbf{G},X}} : (\mathbf{F} \times \mathbf{G})(X) \longrightarrow \mathbf{F}(X)$ is just the projection $\pi_{1_{\mathbf{F}(X),\mathbf{G}(X)}} : \mathbf{F}(X) \times \mathbf{G}(X) \longrightarrow \mathbf{F}(X)$ and $\pi_{1_{\mathbf{F},\mathbf{G},f}} = 1$; similarly for π_2.

$$\begin{array}{ccc} \mathbf{F}(X) \times \mathbf{G}(X) & \xrightarrow{\pi_1} & \mathbf{F}(X) \\ {\scriptstyle \mathbf{F}(f) \times \mathbf{G}(f)}\downarrow & & \downarrow{\scriptstyle \mathbf{F}(f)} \\ \mathbf{F}(Y) \times \mathbf{G}(Y) & \xrightarrow[\pi_1]{} & \mathbf{F}(Y) \end{array}$$

Remark 2.3.10. By Theorem 2.3.9, we could have done with just one notion of lax natural transformation and eliminated the occurrences of the other in exchange for exponents and brackets. However, this certainly wouldn't help readability and as the two notions, forward and reverse, are equally valid and complex we prefer to keep both and use Theorem 2.3.9 to relate the results.

2.4 Gray's tensor product of 2-categories

Although for any pair of 2-categories the functors between them, lax natural transformations and modifications form a 2-category, these aren't the hom-categories in a 3-category whose objects are 2-categories. In fact 2-categories, 2-functors and lax natural transformations don't even form a 2-category. The problem is the supposed multiplication

$$c_{\mathscr{C},\mathscr{D},\mathscr{E}} : [\mathscr{C}, \mathscr{D}] \times [\mathscr{D}, \mathscr{E}] \longrightarrow [\mathscr{C}, \mathscr{E}] \ , \tag{2.28}$$

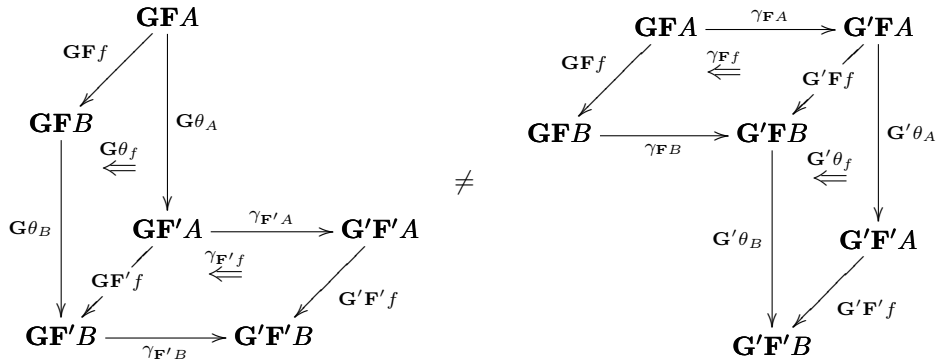
which must be a functor (see Defs 2.2.1 and 2.2.13). Let us try to construct such a functor. On objects, $c_{\mathscr{C},\mathscr{D},\mathscr{E}}$ must act by horizontal composition sending a pair $\mathbf{F}$, $\mathbf{G}$, to $\mathbf{G} \circ \mathbf{F}$. For a pair of arrows $\theta : \mathbf{F} \Longrightarrow \mathbf{F}' : \mathscr{C} \longrightarrow \mathscr{D}$ and $\gamma : \mathbf{G} \Longrightarrow \mathbf{G}' : \mathscr{D} \longrightarrow \mathscr{E}$, there is an arrow $(\theta, \gamma) : (\mathbf{F}, \mathbf{G}) \longrightarrow (\mathbf{F}', \mathbf{G}')$ in $[\mathscr{C}, \mathscr{D}] \times [\mathscr{D}, \mathscr{E}]$, pictured in the following commutative diagram in $[\mathscr{C}, \mathscr{D}] \times [\mathscr{D}, \mathscr{E}]$:

$$\begin{array}{ccc} (\mathbf{F},\mathbf{G}) & \xrightarrow{(\mathbf{F},\gamma)} & (\mathbf{F},\mathbf{G}') \\ {\scriptstyle (\theta,\mathbf{G})}\downarrow & \searrow^{(\theta,\gamma)} & \downarrow{\scriptstyle (\theta',\mathbf{G})} \\ (\mathbf{F}',\mathbf{G}) & \xrightarrow[(\mathbf{F}',\gamma')]{} & (\mathbf{F}',\mathbf{G}') \end{array} \tag{2.29}$$

This diagram (2.29) would have to be taken by (2.28) to a commutative diagram in $[\mathscr{C}, \mathscr{E}]$. By definition, the action of c on $(\theta, \mathbf{G})$ must be a lax natural transformation of type $\mathbf{GF} \longrightarrow \mathbf{GF}'$. It's easy to see from functoriality and the definition of $[\mathscr{C}, \mathscr{D}] \times [\mathscr{D}, \mathscr{E}]$ that this must be horizontal composition. Similarly for $(\mathbf{F}, \gamma)$. So one has the following situation:

$$\begin{array}{ccc} (\mathbf{F},\mathbf{G}) & \xrightarrow{(\mathbf{F},\gamma)} & (\mathbf{F},\mathbf{G}') \\ {\scriptstyle (\theta,\mathbf{G})}\downarrow & \searrow^{(\theta,\gamma)} & \downarrow{\scriptstyle (\theta,\mathbf{G}')} \\ (\mathbf{F}',\mathbf{G}) & \xrightarrow[(\mathbf{F}',\gamma)]{} & (\mathbf{F}',\mathbf{G}') \end{array} \quad \longmapsto \quad \begin{array}{ccc} \mathbf{GF} & \xrightarrow{\gamma\mathbf{F}} & \mathbf{G}'\mathbf{F} \\ {\scriptstyle \mathbf{G}\theta}\downarrow & \neq & \downarrow{\scriptstyle \mathbf{G}'\theta} \\ \mathbf{GF}' & \xrightarrow[\gamma\mathbf{F}']{} & \mathbf{G}'\mathbf{F}' \end{array} \tag{2.30}$$

The right-hand-side square in (2.30) doesn't commute. To see this, consider components of $(\gamma\mathbf{F} \cdot \mathbf{G}\theta)_f$ and $(\mathbf{G}'\theta \cdot \gamma\mathbf{F})_f$, respectively, for an $f : A \longrightarrow A'$ in $\mathscr{C}$.



So there is no such functor, but there is a solution. In terms of enriched category theory [Kel82], a 2-category is a Cat-category where objects are 2-categories and Cat is the symmetric monoidal category of small categories where the tensor product is

the cartesian product of categories $(\times) : \mathsf{Cat} \times \mathsf{Cat} \longrightarrow \mathsf{Cat}$. This definition gives rise to the requirement of a composition functor in the form (2.28) which doesn't exist for the above reasons. However, there is a category-like structure, a $2\mathsf{Cat}_\otimes$-category, whose objects are 2-categories and hom-categories are categories $[\mathscr{C}, \mathscr{D}]_0$ for each pair of 2-categories $\mathscr{C}$, $\mathscr{D}$. Composition in this category has type:

$$c_{\mathscr{C},\mathscr{D},\mathscr{E}} \ : \ [\mathscr{C}, \mathscr{D}] \otimes [\mathscr{D}, \mathscr{E}] \longrightarrow [\mathscr{C}, \mathscr{E}] \ , \tag{2.31}$$

where $\otimes : 2\mathsf{Cat}_0 \times 2\mathsf{Cat}_0 \longrightarrow 2\mathsf{Cat}_0$ is so-called *Gray's tensor product* of 2-categories. Objects in this category are 2-categories, arrows are 2-functors and 2-cells are lax natural transformations. Horizontal composition is horizontal composition of 2-functors and natural transformations, vertical composition is vertical composition of natural transformations. Informally, Gray's solution to coherence of composition is in relaxing strict commutativity to lax commutativity by inserting syntactically constructed diagonal 2-cells. Formally, $2\mathsf{Cat}_\otimes$ is the monoidal category whose underlying category is $2\mathsf{Cat}_0$ and tensor product $\otimes$, Gray's tensor product of 2-categories which is discussed below. Gray in [Gra74] further shows that the resulting $2\mathsf{Cat}_\otimes$-category is bimonoidal closed – i.e. there are right adjoints to the left and right sections of $\otimes$:

$$\begin{aligned} \mathscr{C} \otimes - \quad &\longrightarrow\!\dashv \quad [\mathscr{C}, -] \\ - \otimes \mathscr{D} \quad &\longrightarrow\!\dashv \quad [\![\mathscr{D}, -]\!] \end{aligned}$$

Explicitly, the functor 2-categories $[\![\mathscr{C}, \mathscr{D}]\!]$ and $[\mathscr{C}, \mathscr{D}]$ play the role of internal hom-categories, similarly to the exponential categories $\mathscr{D}^{\mathscr{C}}$ in the strict case.

We have no intention of going into the full details of the construction of $\otimes$. The following theorem characterises $\otimes$ in terms of its relation to the categories $[\![\mathscr{C}, \mathscr{D}]\!]$ and $[\mathscr{C}, \mathscr{D}]$, which are essential for the development of the rest of the thesis.

Theorem 2.4.1 ([Gra74])**.** *There exists a functor* $\otimes : 2\mathsf{Cat}_0 \times 2\mathsf{Cat}_0 \longrightarrow 2\mathsf{Cat}_0$ *together with a natural family of natural isomorphisms*

$$\begin{aligned} \alpha_{\mathscr{C},\mathscr{D},\mathscr{E}} \ &: \ (\mathscr{C} \otimes \mathscr{D}) \otimes \mathscr{E} \longrightarrow \mathscr{C} \otimes (\mathscr{D} \otimes \mathscr{E}) \\ \lambda_{\mathscr{C}} \ &: \ (1 \otimes \mathscr{C}) \longrightarrow \mathscr{C} \\ \rho_{\mathscr{C}} \ &: \ (\mathscr{C} \otimes 1) \longrightarrow \mathscr{C} \ , \end{aligned}$$

which are respectively the associativity*,* left *and* right units *of* $\otimes$*. Moreover,*

1. *for any* $\mathscr{C}$*,* $\mathscr{D}$*:*

$$\mathscr{C} \otimes (-) \quad \longrightarrow\!\dashv \quad [\mathscr{C}, -] \tag{2.32}$$

$$(-) \otimes \mathscr{D} \quad \longrightarrow\!\dashv \quad [\![\mathscr{D}, -]\!] \tag{2.33}$$

2. for any $\mathscr{C}$, $\mathscr{D}$, $\mathscr{E}$ there are natural isomorphisms of categories:

$$[\mathscr{C} \otimes \mathscr{D}, \mathscr{E}] \quad \cong \quad [\mathscr{D}, [\mathscr{C}, \mathscr{E}]] \tag{2.34}$$

$$[\![\mathscr{C} \otimes \mathscr{D}, \mathscr{E}]\!] \quad \cong \quad [\![\mathscr{C}, [\![\mathscr{D}, \mathscr{E}]\!]]\!] \tag{2.35}$$

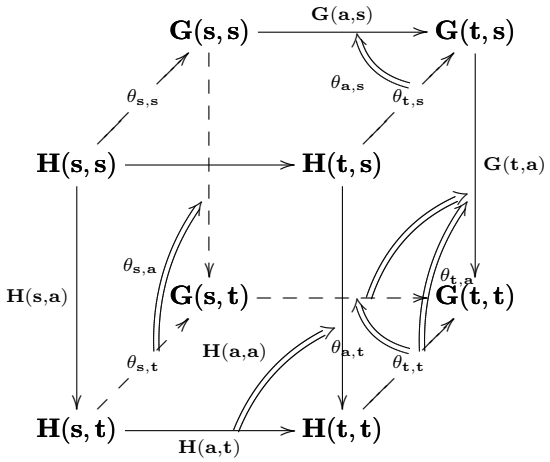
As mentioned before, (2.32) and (2.33) establish a bimonoidal closed category structure on $2\mathsf{Cat}_0$ and are essential for establishing the rest, but are not needed as such in the rest of the thesis.

Items (2.34) and (2.35) provide an insight into the nature of $\otimes$ without us having to go into an explicit definition. In particular, take $\mathscr{C} = \mathscr{D} = 2$, where 2 is the two-object-one-arrow category, as before. Now, an object in $[\![2, [\![2, \mathscr{C}]\!]]\!]$ for any $\mathscr{C}$, i.e. a functor $\mathbf{H} : 2 \longrightarrow [\![2, \mathscr{C}]\!]$, is a pair of arrows of $\mathscr{C}$: $\mathbf{H}(\mathbf{s}) : \mathbf{H}(\mathbf{s})(\mathbf{s}) \xrightarrow{\mathbf{H}(\mathbf{s})(\mathbf{a})} \mathbf{H}(\mathbf{s})(\mathbf{t})$ and $\mathbf{H}(\mathbf{t}) : \mathbf{H}(\mathbf{t})(\mathbf{s}) \xrightarrow{\mathbf{H}(\mathbf{t})(\mathbf{a})} \mathbf{H}(\mathbf{t})(\mathbf{t})$, and a forward lax natural transformation $\mathbf{H}(\mathbf{a}) : \mathbf{H}(\mathbf{s}) \longrightarrow \mathbf{H}(\mathbf{t})$. In summary, it is the following diagram in $\mathscr{C}$:

$$\begin{array}{ccc} \mathbf{H}(\mathbf{s},\mathbf{s}) & \xrightarrow{\mathbf{H}(\mathbf{a},\mathbf{s})} & \mathbf{H}(\mathbf{t},\mathbf{s}) \\ {\scriptstyle \mathbf{H}(\mathbf{s},\mathbf{a})}\downarrow & \swarrow{\scriptstyle \mathbf{H}(\mathbf{a},\mathbf{a})} & \downarrow{\scriptstyle \mathbf{H}(\mathbf{t},\mathbf{a})} \\ \mathbf{H}(\mathbf{s},\mathbf{t}) & \xrightarrow[\mathbf{H}(\mathbf{a},\mathbf{t})]{} & \mathbf{H}(\mathbf{t},\mathbf{t}) \end{array} , \tag{2.36}$$

where $\mathbf{H}$ is uncurried. By (2.34), (2.36) is the image of a functor $\mathbf{H} : 2 \otimes 2 \longrightarrow \mathscr{C}$. Compare with (2.29). Similarly, an object of $[2, [2, \mathscr{C}]]$, and equivalently a functor $2 \otimes 2 \longrightarrow \mathscr{C}$, is a laxly commuting square (2.36) with $\mathbf{H}(\mathbf{a}, \mathbf{a})$ reversed.

Arrows in $[\![2, [\![2, \mathscr{C}]\!]]\!]$ are laxly commuting cubes



Its not difficult to figure out what 2-cells in $[\![2, [\![2, \mathscr{C}]\!]]\!]$ are. As for more interesting

categories with 2-cells, such as $\mathbf{X} =_{\text{def}} \mathbf{s} \xrightarrow[\Downarrow\alpha]{\mathbf{a}} \mathbf{t}$, we get by a similar analysis that objects in $[\![\mathbf{X}, [\![\mathbf{X}, \mathscr{C}]\!]]\!]$ are laxly commuting "cushions", such as the one pictured in (2.23), where $\mathbf{F} =_{\text{def}} \mathbf{H}(\mathbf{s}, -)$, $\mathbf{G} =_{\text{def}} \mathbf{H}(\mathbf{t}, -)$, $A =_{\text{def}} \mathbf{s}$, $B =_{\text{def}} \mathbf{t}$, $f =_{\text{def}} \mathbf{a}$, $g =_{\text{def}} \mathbf{a}'$, etc.

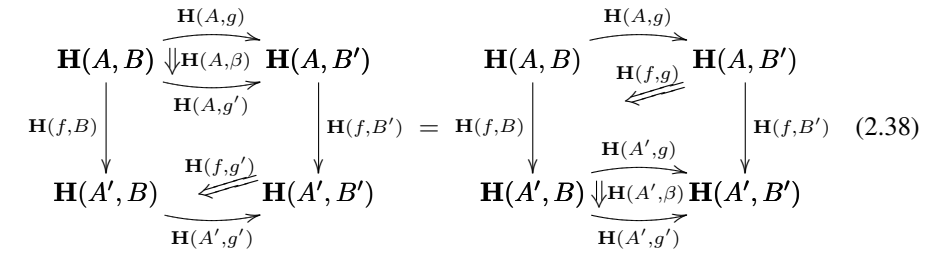
Finally, the following isomorphism is important. It describes how the two monoidal structures given by the adjunctions (2.32) and (2.33) interact. The details are in [Gra74].

Theorem 2.4.2 (Transposition)**.** *For any 2-categories $\mathscr{C}$, $\mathscr{D}$, $\mathscr{E}$:*

$$[\mathscr{C}, [\![\mathscr{D}, \mathscr{E}]\!]] \quad \cong \quad [\![\mathscr{D}, [\mathscr{C}, \mathscr{E}]]\!] \tag{2.37}$$

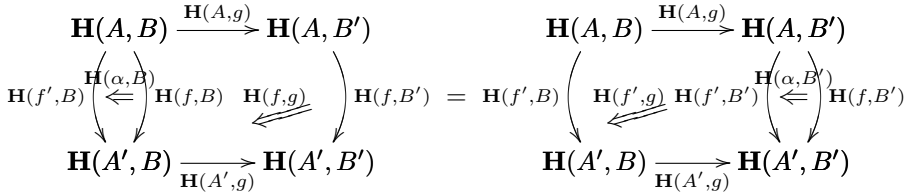
In the special case of $\mathscr{C} = \mathscr{D} = 2$, Theorem 2.4.2 states formally that it doesn't matter whether the square (2.36) is considered as a forward morphism of arrows $\mathbf{H}(\mathbf{s}, -) \longrightarrow \mathbf{H}(\mathbf{t}, -)$ going from left to right, or a reverse morphism $\mathbf{H}(-, \mathbf{s}) \longrightarrow \mathbf{H}(-, \mathbf{t})$ of arrows going from top to bottom.

In the general case, to see the isomorphism at work on objects of $[\mathscr{C}, [\![\mathscr{D}, \mathscr{E}]\!]]$ take an $\mathbf{F} : \mathscr{C} \longrightarrow [\![\mathscr{D}, \mathscr{E}]\!]$. Its value on $A \in \mathscr{C}$ is a functor $\mathbf{H}(A, -) : \mathscr{D} \longrightarrow \mathscr{E}$; its value on an $f : A \longrightarrow A'$ is a lax natural transformation $\mathbf{H}(f, -) : \mathbf{H}(A, -) \longrightarrow \mathbf{H}(A', -)$ with components $\mathbf{H}(f, B)$, $\mathbf{H}(f, B')$ and $\mathbf{H}(f, g)$ for each $g : B \longrightarrow B'$ in $\mathscr{D}$, such that the following diagram commutes for all $\gamma : g \Longrightarrow g' : B \longrightarrow B'$:



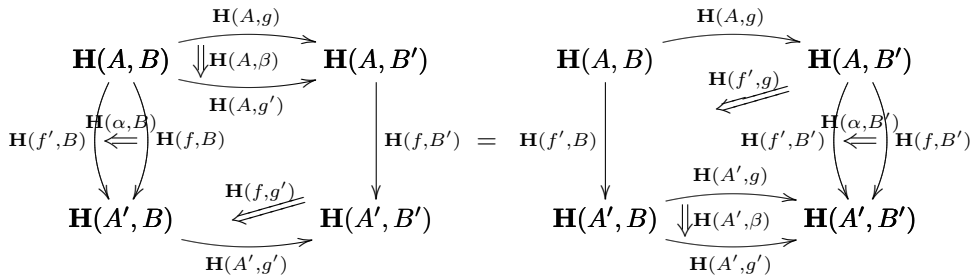
(2.38)

Further, there is a modification $\mathbf{F}(\alpha, -)$ for each $\alpha : f \Longrightarrow f' : A \longrightarrow A'$ such that the following holds:



(2.39)

Relations (2.38) and (2.39) combine to a single diagram:



(2.40)

Clearly, (2.38) and (2.39) are just special cases of (2.40). In summary, $\mathbf{F}$ assigns to every f, f', g, g', α, β the diagram (2.40). On the other hand, a functor $\mathbf{G} : \mathscr{D} \longrightarrow [\mathscr{C}, \mathscr{E}]$ assigns to each $B \in \mathscr{D}$ a functor, $\mathbf{H}(-, B) : \mathscr{C} \longrightarrow \mathscr{E}$, and to each $g : B \longrightarrow B'$ a reverse lax natural transformation $\mathbf{H}(-, g)$ such that for each $\alpha : f \Longrightarrow f' : A \longrightarrow A'$ the diagram (2.39) commutes. Diagram (2.38) depicts a modification $\mathbf{H}(-, \beta)$ given by the action of $\mathbf{G}$ on a 2-cell $\beta : g \Longrightarrow g' : B \longrightarrow B'$. That is, also $\mathbf{G}$ assigns to every f, f', g, g', α, β the diagram (2.40).

The assignment (2.40) to the α, β, as above is called *quasi-functor of two variables* by Gray.

Definition 2.4.3 (Quasi-functor of two variables)**.** *For 2-categories, $\mathscr{C}, \mathscr{D}, \mathscr{E}$,* quasi-functor of two variables $\mathbf{H} : \mathscr{C} \times \mathscr{D} \rightsquigarrow \mathscr{E}$ *consists of families of 2-functors*

$$\begin{aligned} \mathbf{H}(A, -) &: \mathscr{D} \longrightarrow \mathscr{E} \qquad \textit{for all } A \in |\mathscr{C}| \\ \mathbf{H}(-, B) &: \mathscr{C} \longrightarrow \mathscr{E} \qquad \textit{for all } B \in |\mathscr{D}| \end{aligned}$$

such that

$$\mathbf{H}(A, -)(B) \;=\; \mathbf{H}(-, B)(A) \;=_{\mathrm{def}}\; \mathbf{H}(A, B)$$

together with for all $f : A \longrightarrow A'$, $g : B \longrightarrow B'$, a 2-cell:

$$\mathbf{H}(f, g) \;:\; \mathbf{H}(f, B') \circ \mathbf{H}(A, g) \Longrightarrow \mathbf{H}(A', g) \circ \mathbf{H}(f, B) \tag{2.41}$$

such that (2.40) commutes and moreover the assignment of (2.41) to f, g, respects units and composition. Explicitly:

1. $\mathbf{H}(1, g) \;=\; 1_g \;, \quad \mathbf{H}(f, 1) \;=\; 1_f$
2. $\mathbf{H}(f'f, g) \;=\; \mathbf{H}(f', g) \;\boxplus\; \mathbf{H}(f, g)$
3. $\mathbf{H}(f, g'g) \;=\; \mathbf{H}(f, g') \;\boxminus\; \mathbf{H}(f, g)$

Theorem 2.4.4 ([Gra74])**.** *There is an isomorphism between quasi-functors of two variables* $\mathscr{C} \times \mathscr{D} \rightsquigarrow \mathscr{E}$ *and 2-functors* $\mathscr{D} \longrightarrow [\mathscr{C}, \mathscr{E}]$.

For the right definition of a natural transformation of quasi-functors, the above isomorphism extends to an isomorphism of 2-categories.

Remark 2.4.5. Note that the notion of a quasi-functor of two variables is the more fundamental one although introduced here later, whereas Gray's tensor product is defined precisely in such a way as to allow one to treat quasi-functors of two variables as 2-functors. This is analogical to the correspondence between functions of two variables and functions of one variable which is a cartesian product.

Chapter 3

Functors with Structure

In this chapter, functors with structure are formalised as functors from certain 2-categories serving as *theories*. The elementary notions are developed and supported by examples. The development starts in Sect. 3.1 by a discussion of formal objects and arrows modelled as functors into a base 2-category. The section is slow-paced and serves mostly to plant the correct intuitions about the functorial approach and to introduce some basic notation. In Sect. 3.2, theories and models of functors with structure are discussed and an approximate definition is given, and Sect. 3.3 introduces their morphisms and discusses the two kinds of 2-categories of functors with structure. Formal forgetful functors, defined in Sect. 3.4, are in the functorial setting given by precomposition with inclusions of theories. They allow us in particular to define the notions of formal domain, codomain and underlying arrows of functors with structure. The chapter concludes by Sect. 3.5, which introduces important examples of functors with structure which are used and further developed in the remaining text.

3.1 Formal Objects and Arrows

In this section the formal definition of functors with structure, to come in Def. 3.2.1, is motivated by the simple example of formal objects and arrows in a 2-category, which are formalised as functors from the simple one-arrow category, 2. Nothing substantially new is presented in this section; the idea of using functors from chosen categories 1, 2, 3 and 4, for formalisation of category theory comes from Lawvere [Law66]. The purpose of this section is to introduce the idea of functorial formalisation of structures in a category in order to prepare the reader for the more general development that follows.

3.1.1 Formal objects and arrows

Let 1 be the 2-category with one object, denoted $\mathbf{o}$. Similarly, let 2 be the 2-category with exactly one nonidentity arrow between two two distinct objects. There are exactly two functors from 1 to 2. One, labelled $\mathbf{s}$, maps $\mathbf{o}$ in 1 to the domain of the arrow in 2. The second one, labelled $\mathbf{t}$, maps $\mathbf{o}$ to the codomain of the arrow.

$$\mathsf{1} \underset{\mathbf{t}}{\overset{\mathbf{s}}{\rightrightarrows}} \mathsf{2} \tag{3.1}$$

So the domain of the arrow is $\mathbf{s}(\mathbf{o})$, also denoted just $\mathbf{s}$, the codomain is $\mathbf{t}(\mathbf{o})$ or just $\mathbf{t}$. The arrow is denoted $\mathbf{a}$.

Notation 3.1.1. It is convenient to use the labels $\mathbf{s}$, $\mathbf{t}$, $\mathbf{a}$ for any category, $\mathscr{K}$, with a distinguished functor $\mathbf{F} : 2 \longrightarrow \mathscr{K}$, which is clear from the context. Then, $\mathbf{s}$ stands for $\mathbf{F} \cdot \mathbf{s}$, $\mathbf{t}$ stands for $\mathbf{F} \cdot \mathbf{t}$ and $\mathbf{a}$ stands for $\mathbf{F}$. This makes it possible to speak about the image of $\mathbf{s}$ and $\mathbf{t}$ under $\mathbf{F}$, without explicitly naming $\mathbf{F}$.

In general, for any 2-category $\mathscr{K}$, a functor from 1 picks one object of $\mathscr{K}$ and any object of $\mathscr{K}$ defines a unique functor from 1 to $\mathscr{K}$ – the functor that picks it. That is, 2-functors from 1 to $\mathscr{K}$ are exactly objects of $\mathscr{K}$. Similarly, an arrow of $\mathscr{K}$ is exactly a functor $2 \longrightarrow \mathscr{K}$. In the following text, we work up to this equivalence and use the terms *formal objects* and *formal arrows* only to emphasise when functors are meant. In mathematical notation, the distinction is made using the numeral brackets, $\ulcorner - \urcorner$, as follows:

Notation 3.1.2.

1. Whenever $\mathscr{K}$ is a category and x is an object in it, $\ulcorner x \urcorner$ denotes the unique functor $1 \xrightarrow{\ulcorner x \urcorner} \mathscr{K}$ whose value on $\mathbf{o}$ is x.
2. For an arrow f in $\mathscr{K}$, $\ulcorner f \urcorner$ denotes the unique functor $2 \xrightarrow{\ulcorner f \urcorner} \mathscr{K}$ whose value on $\mathbf{a}$ is f.
3. The brackets are dropped when there is no danger of confusion.

Endomorphisms in $\mathscr{K}$ are arrows with the same domain and codomain. Formally, let $(\mathsf{O}, \mathbf{a})$ be the coequaliser of (3.1), i.e. in the following diagram $\mathbf{a}$ is a coequaliser arrow:

$$\mathsf{1} \underset{\mathbf{t}}{\overset{\mathbf{s}}{\rightrightarrows}} \mathsf{2} \xrightarrow{\mathbf{a}} \mathsf{O} \tag{3.2}$$

The category O is equivalent to a nonidentity arrow on one object.

The unique arrow from 1 to O is denoted $\mathbf{o}$. Functors $\mathsf{O} \longrightarrow \mathscr{K}$ are called *formal loops* in $\mathscr{K}$.

3.1.2 Categories of formal objects and arrows

As mentioned above, the set of formal objects in $\mathscr{K}$, formally $|1, \mathscr{K}|$, is isomorphic to $|\mathscr{K}|$, the set of objects of $\mathscr{K}$. Likewise, the set of formal arrows in $\mathscr{K}$ is isomorphically the set of arrows of $\mathscr{K}$; formally $|2, \mathscr{K}|$. These sets form the sets of objects in the functor categories $[\![1, \mathscr{K}]\!]$, $[1, \mathscr{K}]$, $[\![2, \mathscr{K}]\!]$ and $[2, \mathscr{K}]$. In this section we discus the simple structure of these categories.

First, consider a forward lax natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G} : 1 \longrightarrow \mathscr{K}$, i.e. an arrow in $[\![1, \mathscr{K}]\!]$. By definition, θ is determined by one component $\theta_{\mathbf{o}} : \mathbf{Fo} \longrightarrow \mathbf{Go}$. Consequently, a modification $\alpha : \theta \Longrightarrow \gamma : \mathbf{F} \Longrightarrow \mathbf{G} : 1 \longrightarrow \mathscr{K}$ is determined by a single 2-cell

$$\begin{array}{ccc} & \overset{\theta_{\mathbf{o}}}{\curvearrowright} & \\ \mathbf{Fo} & \alpha \Downarrow & \mathbf{Go} \\ & \underset{\gamma_{\mathbf{o}}}{\curvearrowright} & \end{array}$$

Note that the same holds when θ is a reverse natural transformation, because 1 is trivial. So formally there is the following equivalence, for any $\mathscr{K}$:

$$[\![1, \mathscr{K}]\!] \;\cong\; \mathscr{K} \;\cong\; [1, \mathscr{K}] \tag{3.3}$$

Arrows of $[\![2, \mathscr{K}]\!]$ and $[2, \mathscr{K}]$ are just slightly more complicated, as is observed below. And, although 2 is still very simple, $[\![2, \mathscr{K}]\!] \ncong [2, \mathscr{K}]$ for some $\mathscr{K}$.

Observation 3.1.3.

1. An *object* in $[\![2, \mathscr{K}]\!]$ is a functor $\mathbf{F} : 2 \longrightarrow \mathscr{K}$. This functor picks an object of $\mathscr{K}$ for each of the two objects of 2, and an arrow of $\mathscr{K}$ between them.

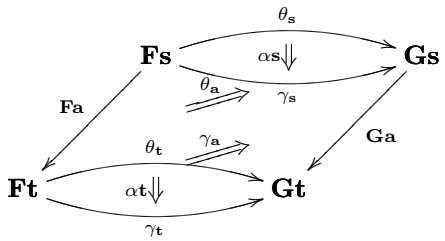
$$\mathbf{Fs} \xrightarrow{\mathbf{Fa}} \mathbf{Ft}$$

2. An *arrow* in $[\![2, \mathscr{K}]\!]$ between functors $\mathbf{F}$ and $\mathbf{G}$ is a lax 2-natural transformation $\theta : \mathbf{F} \Longrightarrow \mathbf{G}$. That is, for the pair of objects of 2, there is a pair of arrows of $\mathscr{K}$, $\theta_{\mathbf{s}} : \mathbf{Fs} \longrightarrow \mathbf{Gs}$ and $\theta_{\mathbf{t}} : \mathbf{Ft} \longrightarrow \mathbf{Gt}$. And for the arrow $\mathbf{a}$ of 2, a 2-cell, $\theta_{\mathbf{t}} \cdot \mathbf{Fa} \overset{\theta_{\mathbf{a}}}{\Longrightarrow} \mathbf{Ga} \cdot \theta_{\mathbf{s}}$.

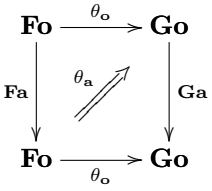
$$\begin{array}{ccc} \mathbf{Fs} & \xrightarrow{\theta_{\mathbf{s}}} & \mathbf{Gs} \\ {\scriptstyle \mathbf{Fa}}\downarrow & \overset{\theta_{\mathbf{a}}}{\nearrow} & \downarrow{\scriptstyle \mathbf{Ga}} \\ \mathbf{Ft} & \xrightarrow[\theta_{\mathbf{t}}]{} & \mathbf{Gt} \end{array} \tag{3.4}$$

Because 2 has no nontrivial 2-cells, there are no other nontrivial coherence condition on 1-cells.

3. A *2-cell* $\alpha : \theta \Longrightarrow \gamma : \mathbf{F} \longrightarrow \mathbf{G}$ is a modification given by a pair of 2-cells $\alpha_{\mathbf{s}} : \theta_{\mathbf{s}} \Rightarrow \gamma_{\mathbf{s}}$ and $\alpha_{\mathbf{t}} : \theta_{\mathbf{t}} \Rightarrow \gamma_{\mathbf{t}}$ such that $\theta_{\mathbf{a}} \boxminus \alpha_{\mathbf{s}} = \alpha_{\mathbf{t}} \boxminus \gamma_{\mathbf{a}}$



4. In $[\![\mathsf{O}, \mathscr{K}]\!]$, the objects are the same. But because there is only one object in O, a morphism in $[\![\mathsf{O}, \mathscr{K}]\!]$, i.e. a formal morphism of endomorphisms, consists of *one* (not two) 1-cells, and a diagonal 2-cell.



2-cells follow the same pattern.

3.1.3 Forgetful functors

Any formal arrow, $\mathbf{F} : 2 \longrightarrow \mathscr{K}$, determines two formal objects – its domain and codomain. These are given formally by precomposition of $\mathbf{F}$ with $\mathbf{s}$ and $\mathbf{t}$, respectively:

$$1 \xrightarrow{\mathbf{s}} 2 \xrightarrow{\mathbf{F}} \mathscr{K} \qquad \text{the domain of } \mathbf{F}$$

$$1 \xrightarrow{\mathbf{t}} 2 \xrightarrow{\mathbf{F}} \mathscr{K} \qquad \text{the codomain of } \mathbf{F}$$

The assignment $f \mapsto f \cdot \mathbf{s}$, for any $f : 2 \longrightarrow \mathscr{K}$ extends to a contravariant functor

$$[\![\mathbf{s}, \mathscr{K}]\!] \; : \; [\![2, \mathscr{K}]\!] \longrightarrow [\![1, \mathscr{K}]\!]$$

sending formal arrows in $\mathscr{K}$ to their domains, formal objects in $\mathscr{K}$. Similarly for $\mathbf{t}$, and also for $[-, \mathscr{K}]$.

Similarly, there is a pair of contravariant homfunctors:

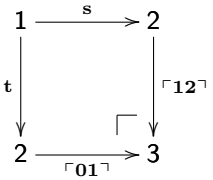
$$[\![\mathbf{a}, 1]\!] : [\![\mathsf{O}, \mathscr{K}]\!] \longrightarrow [\![2, \mathscr{K}]\!]$$
$$[\mathbf{a}, 1] : [\mathsf{O}, \mathscr{K}] \longrightarrow [2, \mathscr{K}] \ ,$$

sending loops to their arrows. Note that this shows formally that endomorphisms are morphisms, when one forgets that the domain and codomain are the same.

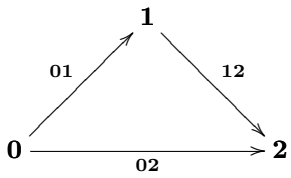
3.1.4 Composition

For any pair of composable arrows, $f : X \longrightarrow Y$, $g : Y \longrightarrow Z$, of an ordinary category $\mathscr{C}$, there exists a composite $gf : X \longrightarrow Z$ in $\mathscr{C}$. This must hold also for formal arrows, i.e. for functors $\mathbf{F} : 2 \longrightarrow \mathscr{C}$ and $\mathbf{G} : 2 \longrightarrow \mathscr{C}$, such that $\mathbf{Ft} = \mathbf{Gs} =_{\text{def}} Y$, there must be a functor, $\mathbf{G} \circ \mathbf{F} : 2 \longrightarrow \mathscr{C}$ such that $(\mathbf{G} \circ \mathbf{F})\mathbf{s} = \mathbf{Fs}$ and $(\mathbf{G} \circ \mathbf{F})\mathbf{t} = \mathbf{Gt}$. This assignment, $(\mathbf{F}, \mathbf{G}) \mapsto \mathbf{G} \circ \mathbf{F}$, must be associative and respect formal identities. Lawvere in [Law68] shows such a notion of composition, which we present in this section.

Let 3 be the category given by the following pushout in Cat:

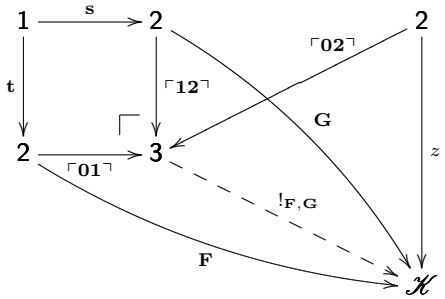


It is a property of the category of categories (an axiom of a category) that in addition to the two arrows defined by the pushout in the above diagram, there is exactly one other arrow, $\ulcorner\mathbf{02}\urcorner : 2 \longrightarrow 3$, picking the composition of $\mathbf{01}$ and $\mathbf{12}$ in 3. So, 3 is a triangle:



Now, any two formal arrows $\mathbf{F}$ and $\mathbf{G}$ in $\mathscr{K}$ are said to be *composable* iff $\mathbf{Ft} = \mathbf{Gs}$. For any such pair of formal arrows, the universal arrow of the pushout precomposed with $\ulcorner\mathbf{02}\urcorner$ gives the composite $\mathbf{GF}$. In summary, it is a property of the category of categories Cat, that for two arrows $\mathbf{F}, \mathbf{G} : 2 \longrightarrow \mathscr{K}$, z is their composite *iff* the

following diagram commutes.



(3.5)

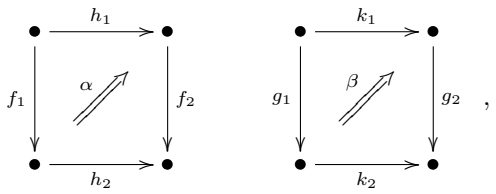
This property defines a family of functions

$$\circ_{X,Y,Z} : |2, \mathscr{K}|_{X,Y} \times |2, \mathscr{K}|_{Y,Z} \longrightarrow |2, \mathscr{K}|_{X,Z} \ , \tag{3.6}$$

where $|2, \mathscr{K}|_{x,y}$ denotes the set of formal arrows from x to y. It is easy to show that this operation is associative and respects units and therefore defines a notion of composition in category where objects are formal objects of $\mathscr{K}$, arrows are formal arrows of $\mathscr{K}$. It is easy to check this category is isomorphic to $\mathscr{K}$.

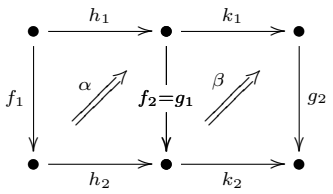
We skip the trivial example of composition of functors as formal arrows in Cat. The following is an example of formal arrows in a category substantially different from Cat.

Example 3.1.4. In Observation 3.1.3, (2), we observed that arrows in $[\![2, \mathscr{K}]\!]$ are laxly commuting squares (3.4). It follows from the development so far that these are the objects of the category $[2, [\![2, \mathscr{K}]\!]]$. For two lax squares, α, β:



(3.7)

such that the codomain of the first is the domain of the second: $f_2 = g_1$, their formal composition corresponds to horizontal pasting $\beta \boxplus \alpha$:

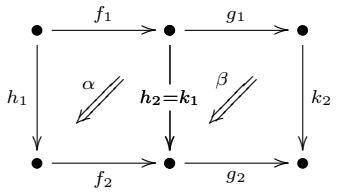


On the other hand, suppose $h_2 = k_1$ in (3.7). Let $\tilde{\alpha}$, $\tilde{\beta}$, denote the transposes of α, β,

under the isomorphism (2.37). We have

$$[2, [\![\mathbf{t}, \mathscr{K}]\!]](\alpha) \;=\; [\![\mathbf{t}, [2, \mathscr{K}]]\!](\tilde{\alpha}) \;=\; \ulcorner h_1 \urcorner$$

Similarly, $[\![\mathbf{s}, [2, \mathscr{K}]]\!](\tilde{\beta}) \;=\; \ulcorner k_1 \urcorner$, and so $\tilde{\alpha}$ and $\tilde{\beta}$ are composable. Their composition corresponds to vertical pasting of squares, $\tilde{\beta} \boxminus \tilde{\alpha}$:



3.2 Functors with Structure

In this section, the notion of a *functor with structure* is approximated by the very coarse Definition 3.2.1. The approach follows the intuitions outlined in the previous section, only generalised to arbitrary 2-categories in place of 2. We give the definition first and discuss some important issues later.

Definition 3.2.1.

1. *a* theory of a functor with structure *is a tuple,* $(\mathscr{T}, a)$*, where* $\mathscr{T}$ *is a small 2-category and* a *is a functor* $a : 2 \longrightarrow \mathscr{T}$*.*

2. *a* morphism of theories *from* $(\mathscr{T}, a)$ *to* $(\mathscr{U}, b)$ *is a functor* $\mathbf{H} : \mathscr{T} \longrightarrow \mathscr{U}$ *such that* $\mathbf{H} \cdot a \;=\; b$*.*

3. *a* (model of) a functor with structure *in a 2-category,* $\mathscr{K}$*, is a functor* $\mathbf{F} : \mathscr{T} \longrightarrow \mathscr{K}$*, where* $\mathscr{T}$ *is a theory of a functor with structure. Thus formally it is a tuple* $(\mathscr{T}, a, \mathbf{F})$*.*

The role of $\mathscr{K}$ in item (3) above is that of a universe of interpretation. It is often Cat. Although later it becomes essential that the definition is abstract in $\mathscr{K}$, the categories in which functors with structure are interpreted are usually built from Cat in such a way that the objects of a theory can be thought of as categories, arrows as functors and 2-cells as natural transformations.

Remark 3.2.2. In Definition 3.2.1 we made a huge step from functors with no additional structure to anything that can have the slightest chance of being called a functor with structure because it simply has a chosen arrow, a, and some "additional structure" which is given by the rest of an arbitrary 2-category, $\mathscr{T}$. However, quite clearly not all 2-categories are meaningful theories of functors with structure. Moreover, there is no connection between the arrow and the additional structure in our definition so the

"additional structure" is allowed to have nothing to with the "functor". In this thesis, we make no attempt to rectify these points. We simply present examples which are much more specific than allowed by the above definition and at the same time we present a very general mechanism to deal with *arbitrary* functors with structure in the above sense. The mechanism is based on the notion of Gray's tensor product. The startling point is that the very general mechanism works exceptionally well for the specific examples. We support this claim by general characterisation theorems, which provide a general frame for the good behaviour. All of this comes in the next chapter.

In the rest of this chapter we analyse Definition 3.2.1 and present examples. Let us fix some notation first.

Notation 3.2.3.

- For a theory $(\mathscr{T}, a)$, a is called the *arrow* of $\mathscr{T}$.
- The theory is often denoted by the name of the category, $\mathscr{T}$. In that case a must be implicit from the context and in a mathematical notation it is denoted simply $\mathbf{a}$.
- Models of a theory $\mathscr{T}$ are called $\mathscr{T}$-functors.
- The term *endofunctor* is used instead of *functor* when a factors through O.

Example 3.2.4 (Functors)**.** The tuple $(2, 1)$ is a theory of a functor with structure, where the additional structure is empty. Any arrow in $\mathscr{K}$ defines a model of 2 in $\mathscr{K}$. Similarly, $(\mathsf{O}, 1)$ is a theory of an endofunctor with structure.

3.2.1 Theories generated by a 2-sketch

As discussed, the introduced notions of a *theory* and its *model* are very weak. However, quite often and in fact in all examples considered in this thesis a theory is built freely from the syntax of a mathematical definition. The gap between the definition – a finite piece of syntax – and the theory – a potentially infinite 2-category – is bridged by the theory of 2-sketches, which is briefly outlined in this section.

Ordinary sketches have been invented by Charles Ehresmann [Ehr68] as a middle ground between Category Theory and the usual mathematical definitions. This is also the role they play in this text. The theory of sketches has been further extensively developed. A good introduction with applications to computer science is [BW99], a more comprehensive text is [BW85]. Charles Wells in [Wel93] provides an outline with extensive references. Sketches are classified according to the kinds of the *other requirements* allowed and form a hierarchy according to expressivity with *linear sketches* – where only equations are allowed – at the bottom, *finite-product sketches* – where objects can be required to be finite products – somewhere in the middle, and *general*

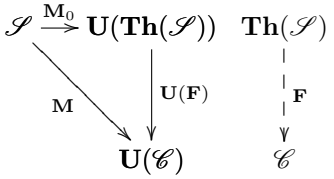
sketches – where arbitrary limits and colimits are allowed – at the top. In all cases the *theory of a sketch* is informally a category freely generated by the underlying graph of the sketch, factorised by the equations and such that the additional requirements are met (e.g. formal products become real products in the theory).

The theory of ordinary sketches extends to the 2-dimensional and more generally enriched context [Kel82, PW92]. In the 2-dimensional case, a definition of a functor with structure can be thought of as a finite or denumerable collection of objects, arrows and 2-cells, the so-called *underlying 2-graph*. Just as in the ordinary case, this data is required to satisfy certain equations and other requirements (such as requirements that certain objects are products or coproducts of other objects). Formally these constitute a so-called *2-sketch*.

In the rest of this section we discuss these points in detail.

Ordinary finite-product sketches

Formally, the theory in the finite-product (**FP**) case is specified as follows. Let an **FP**-sketch be a graph with a collection of formal diagrams in the graph and formal product cones. Let $\mathbf{Sktch_{FP}}$ be a category of **FP** sketches with morphisms defined as to preserve the formal diagrams and product cones, and let $\mathbf{U} : \mathsf{Cat} \longrightarrow \mathbf{Sktch_{FP}}$ be the *underlying sketch functor*, which takes a category, $\mathscr{C}$, to its underlying graph, all commuting diagrams to formal diagrams in the sketch, product cones to formal product cones in the sketch. A model of a sketch, $\mathscr{S}$, in a category $\mathscr{C}$ is a sketch morphism $\mathbf{M} : \mathscr{S} \longrightarrow \mathbf{U}(\mathscr{C})$. Then a theory of a sketch $\mathscr{S} \in \mathbf{Sktch}$ is a category $\mathbf{Th}(\mathscr{S})$ together with a model $\mathbf{M}_0 : \mathscr{S} \longrightarrow \mathbf{U}(\mathbf{Th}(\mathscr{S}))$, which is universal in the sense that $\mathbf{M}_0$ implies a natural isomorphism between models of $\mathscr{S}$ in a $\mathscr{C}$ and finite-product preserving functors $\mathbf{F} : \mathbf{Th}(\mathscr{S}) \longrightarrow \mathscr{C}$. In $\mathbf{Sktch_{FP}}$ this is pictured as follows:



(3.8)

The following characterisation of the theory of an **FP** sketch is important, for more details see [BW85].

Theorem 3.2.5. *The theory* $\mathbf{Th}(\mathscr{S})$ *is defined up to equivalence by the following properties:*

1. $\mathbf{Th}(\mathscr{S})$ *has all finite products*
2. $\mathbf{M}_0$ *takes every diagram in* $\mathscr{S}$ *to a commutative diagram in* $\mathbf{Th}(\mathscr{S})$

3. $\mathbf{M}_0$ *takes every formal product cone to a product cone in* $\mathbf{Th}(\mathscr{S})$

4. *No proper subcategory of* $\mathbf{Th}(\mathscr{S})$ *includes an image of* $\mathbf{M}_0$ *and satisfies (1)-(3).*

This gives in the *ordinary case* a mechanism to define uniquely a category by a finite formal specification – a sketch, and also to define functors from this category by finite enumeration of cases – the sketch model $\mathbf{M}$.

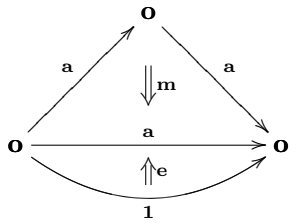
Finite-product 2-sketches

A generalisation of the ordinary case to 2-categories formally underpins a transition from an essentially 2-categorical mathematical definition to a 2-category – a theory of a functor with structure in the sense of Def. 3.2.1. In this thesis we specifically consider finite-product 2-sketches, i.e. essentially **FP** sketches with additional formal 2-edges between *paths* of 2-edges and diagrams on them. These can been defined formally as instances of more general enriched sketches, as Kelly shows in [Kel82], Chapter 6. As 2-categories are just Cat-categories, the kind of sketch we are interested in is a Cat-sketch in Kelly's sense. Explicitly, in detail and an accessible form the construction of a 2-sketch and its theory is shown in [PW92].

We don't go into more details here, just note that a theory of an **FP** 2-sketch is characterised precisely as in the ordinary case, diagram (3.8) and Theorem 3.2.5, with the exception that everything is considered 2-categorically.

This is the formal sense in which we define 2-functors from a theory of a 2-sketch (usually an infinite object) by describing the action of the 2-functor on the components of the 2-sketch and assume the rest is generated freely. This also applies to 2-natural transformations.

Example 3.2.6 (Monads)**.** An example of an essentially 2-categorical definition is the definition of a *monad*. By rewriting the definition as a graph with 2-cells (so-called *computad* by Ross Street [Str76]), one obtains the following:



The definition also prescribes 2-diagrams on the 2-cells expressing that

$$\mathbf{m} \cdot (\mathbf{m} \circ \mathbf{a}) \;=\; \mathbf{m} \cdot (\mathbf{a} \circ \mathbf{m}) \tag{3.9}$$

$$\mathbf{m} \cdot (\mathbf{e} \circ \mathbf{a}) \;=\; 1_{\mathbf{o}} \;=\; \mathbf{m} \cdot (\mathbf{a} \circ \mathbf{e}) \tag{3.10}$$

Moreover, there are implicit 1-diagrams (diagrams on the arrows), which must specify that the arrow denoted $\mathbf{1}$ becomes an identity.

A theory of this sketch is a 2-category which is generated freely by the graph and equations. The theory is denoted M and $\mathbf{a}$ denotes the inclusion $2 \longrightarrow \mathsf{O} \longrightarrow \mathsf{M}$, and also the inclusion $\mathsf{O} \longrightarrow \mathsf{M}$. A model of M in $\mathscr{K}$ is given by a 1-cell for $\mathbf{a}$, a pair of 2-cells for $\mathbf{m}$ and $\mathbf{e}$ such that (3.9) and (3.10) hold in $\mathscr{K}$ It is obvious that a model of M in Cat is a monad.

3.3 Morphisms of functors with structure

In summary, a theory of a functor with structure in full generality is a 2-category with a chosen arrow. The models of a theory $\mathscr{T}$ in $\mathscr{K}$ are 2-functors. What is the corresponding notion of a morphism?

In the well known ordinary case, theories are ordinary categories, models are functors, morphisms are natural transformations. A 2-category is a collection of ordinary categories connected by horizontal composition and therefore, as explained in Sect. 2.3, the corresponding notion of a natural transformation of 2-functors, which is a *coherent collection of ordinary natural transformations*, is lax a natural transformation. The following definition fixes this nomenclature.

Definition 3.3.1 (Morphism of functors with structure)**.** *Let $(\mathscr{T}, a)$ be a theory of a functor with structure; let $\mathscr{K}$ be a 2-category.*

1. *A category of* models *of $\mathscr{T}$ in $\mathscr{K}$ is the hom-category $[\mathscr{T}, \mathscr{K}]$ or $[\![\mathscr{T}, \mathscr{K}]\!]$. When in need of distinguishing them, the former is called* reverse*, while the latter* forward*.*

2. *A* morphism *of functors with structure from $\mathbf{F}$ to $\mathbf{G}$ with theory $\mathscr{T}$ is an arrow in $[\mathscr{T}, \mathscr{K}]$ or in $[\![\mathscr{T}, \mathscr{K}]\!]$. Again, the former is a* reverse*, the latter a* forward *morphism.*

Remark 3.3.2. Our *reverse morphisms* are also called *colax morphisms* and our *forward morphisms* are called just *lax morphisms* in some other literature. However, as everything is already lax in this thesis, we believe this nomenclature could be misleading so we opted for a symmetry where just *morphism* without adjectives means either of them.

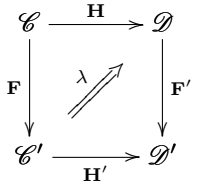
Similarly, our *forward category of models* is related to what is called a *comodel* in [PS04]. Namely, a category of *comodels* of a Lawvere theory $\mathscr{L}$ is an opposite category of its models in $\mathscr{C}^{\mathsf{op}}$ for an ordinary category $\mathscr{C}$. This generalises to the 2-categorical case by the following isomorphism:

$$[\mathscr{A}, \mathscr{K}^{\mathsf{co}}]^{\mathsf{co}} \cong [\![\mathscr{A}^{\mathsf{co}}, \mathscr{K}]\!] \ ,$$

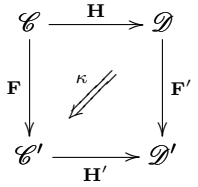
So a forward category of models of $\mathscr{A}^{\mathsf{co}}$ could also be called a category of comodels of $\mathscr{A}$.

Example 3.3.3. The case of (endo)functors with no additional structure is illuminating. See also Observation 3.1.3.

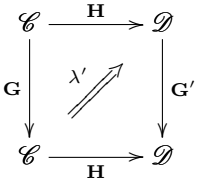
1. Models of 2 in Cat are just functors, $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{C}'$, $\mathbf{F}' : \mathscr{D} \longrightarrow \mathscr{D}'$. A forward morphism $\mathbf{F} \longrightarrow \mathbf{G}$, an arrow in $[\![2, \mathsf{Cat}]\!]$, is given by a pair of functors $\mathbf{H} : \mathscr{C} \longrightarrow \mathscr{D}$ and $\mathbf{H}' : \mathscr{C}' \longrightarrow \mathscr{D}'$ and a natural transformation $\lambda : \mathbf{H}'\mathbf{F} \Longrightarrow \mathbf{F}'\mathbf{H}$.



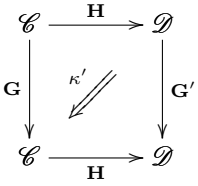
2. A reverse morphism of the same functors $\mathbf{F}$, $\mathbf{F}'$, i.e. an arrow $\mathbf{F} \longrightarrow \mathbf{F}'$ in $[2, \mathsf{Cat}]$, is given by $\mathbf{H}$ and $\mathbf{H}'$ as above and a natural transformation $\kappa : \mathbf{F}'\mathbf{H} \Longrightarrow \mathbf{H}'\mathbf{F}$.



3. When endofunctors $\mathbf{G} : \mathscr{C} \longrightarrow \mathscr{C}$ and $\mathbf{G}' : \mathscr{D} \longrightarrow \mathscr{D}$ are considered instead, i.e. models of O in Cat, a forward morphism in $[\![\mathsf{O}, \mathsf{Cat}]\!]$ is given by a single functor $\mathbf{H} : \mathscr{C} \longrightarrow \mathscr{D}$ and a natural transformation $\lambda : \mathbf{H}\mathbf{G} \Longrightarrow \mathbf{G}'\mathbf{H}$.



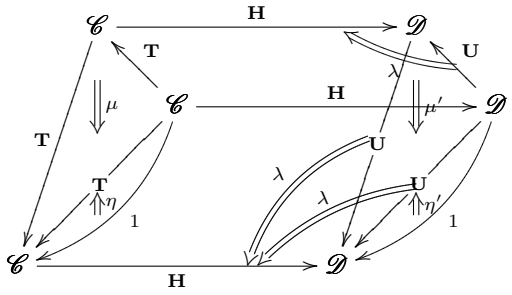
4. A reverse morphism $\mathbf{G} \longrightarrow \mathbf{G}'$ in $[\mathsf{O}, \mathsf{Cat}]$ has the natural transformation reversed:



The following example illustrates morphisms of a more complicated theory with

nontrivial 2-cells.

Example 3.3.4. Let $\mathscr{C}, \mathscr{D} \in \mathsf{Cat}$ be ordinary categories, and $\mathbf{T}$ and $\mathbf{U}$ a pair of monads. Formally, $\mathbf{T}, \mathbf{U}$ are functors $\mathsf{M} \longrightarrow \mathsf{Cat}$. Let the components of $\mathbf{T}$ and $\mathbf{U}$ be $(\mathbf{T} : \mathscr{C} \longrightarrow \mathscr{C}, \mu, \eta)$ and $(\mathbf{U} : \mathscr{D} \longrightarrow \mathscr{D}, \mu', \eta')$, respectively. A morphism $\mathbf{T} \longrightarrow \mathbf{U}$ in $[\mathsf{M}, \mathsf{Cat}]$ is by definition given by a functor $\mathbf{H} : \mathscr{C} \longrightarrow \mathscr{D}$ and a natural transformation $\lambda : \mathbf{UH} \Longrightarrow \mathbf{HT}$ such that the following diagram commutes:



(3.11)

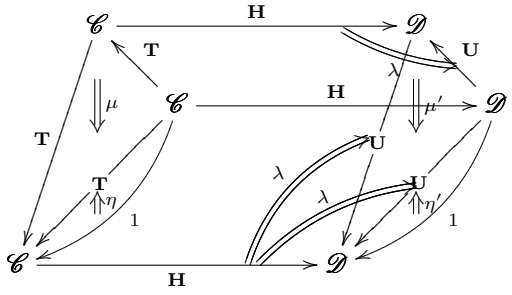
Equivalently,

$$\begin{aligned} (\lambda \boxplus \lambda) \boxminus \mu &= \mu' \boxminus \lambda \\ \eta' \boxminus \lambda &= \eta \end{aligned} \tag{3.12}$$

The reader might be familiar with (3.12) in the more elementary form:

$$\begin{aligned} \mathbf{H}\mu \cdot \lambda\mathbf{T} \cdot \mathbf{U}\lambda &= \lambda \cdot \mu'\mathbf{H} \\ \lambda \cdot \eta'\mathbf{H} &= \mathbf{H}\eta \end{aligned} \tag{3.13}$$

The forward case, i.e. a morphism $\mathbf{T} \longrightarrow \mathbf{U}$ in $[\![\mathsf{M}, \mathsf{Cat}]\!]$ is obtained by reversing the direction of λ in (3.11). This is three-dimensionally the following diagram



(3.14)

Note that strictly speaking there is also a "silent" reversal of the identity 2-cell belong-

ing to the square $\begin{array}{ccc} \mathscr{C} & \xrightarrow{\mathbf{H}} & \mathscr{D} \\ {\scriptstyle 1}\downarrow & & \downarrow{\scriptstyle 1} \\ \mathscr{C} & \xrightarrow[\mathbf{H}]{} & \mathscr{D} \end{array}$. The form (3.12) changes by the reversal to the following:

$$\begin{aligned} \mu' \boxplus (\lambda \boxminus \lambda) &= \lambda \boxplus \mu \\ \lambda \boxplus \eta &= \eta' \end{aligned}$$

and correspondingly (3.13) becomes

$$\begin{aligned} \mu' \mathbf{H} \cdot \mathbf{U}\lambda \cdot \lambda \mathbf{T} &= \lambda \cdot \mathbf{H}\mu \\ \lambda \cdot \mathbf{H}\eta &= \eta' \mathbf{H} \end{aligned}$$

Remark 3.3.5. The forward and reverse categories of functors with structure are related by the dualities of Theorem 2.3.9 as follows:

$$[\mathscr{T}, \mathscr{K}] \cong [\![\mathscr{T}^{\mathsf{co}}, \mathscr{K}^{\mathsf{co}}]\!]^{\mathsf{co}}$$

3.4 Forgetful Functors

As before for arrows, precomposition of models with inclusion corresponds to forgetting parts of the structure. In particular, the following notions are valid for all functors with structure.

Definition 3.4.1. *Let* $\mathbf{F} : \mathscr{T} \longrightarrow \mathscr{K}$ *be a functor with structure with theory* $(\mathscr{T}, a)$. *Then the*

1. underlying arrow *of* $\mathbf{F}$ *is the formal arrow defined as:*

$$\mathbf{F} \cdot a : 2 \longrightarrow \mathscr{K}$$

   This arrow is often denoted just $\mathbf{Fa}$.

2. domain *(or* source*), and* codomain *(or* target*) are the formal objects of* $\mathscr{K}$ *defined respectively as*

$$\begin{aligned} \mathbf{F} \cdot a \cdot \mathbf{s} &: 1 \longrightarrow \mathscr{K} \\ \mathbf{F} \cdot a \cdot \mathbf{t} &: 1 \longrightarrow \mathscr{K} \end{aligned}$$

   These arrows are often denoted just $\mathbf{Fs}$ *and* $\mathbf{Ft}$.

3. *if* $\mathbf{F}$ *is an endofunctor with structure, the* underlying object *of* $\mathbf{F}$ *is the formal*

object

$$\mathbf{F} \cdot a \cdot \mathbf{t} \;=\; \mathbf{F} \cdot a \cdot \mathbf{s} : 1 \longrightarrow \mathscr{K}$$

This arrow is often denoted just **Fo**.

In general, there is an obvious notion of *underlying "something" functor*, defined by precomposition with inclusions. Formally, for a functor $f : \mathscr{A} \longrightarrow \mathscr{T}$ of theory $(\mathscr{T}, a)$ and a category $\mathscr{A}$, one obtains by precomposition from a model, $\mathbf{F}$, of $\mathscr{T}$ a functor, $\mathbf{F} \circ f$ from $\mathscr{A}$. This assignment extends to a functor, the contravariant representable functor

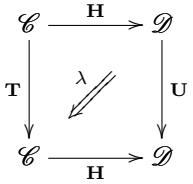
$$[\![f, \mathscr{K}]\!] : [\![\mathscr{T}, \mathscr{K}]\!] \longrightarrow [\![\mathscr{A}, \mathscr{K}]\!] \ , \tag{3.15}$$

for any $\mathscr{K}$. When f is an inclusion of categories, it makes sense to call (3.15) a *forgetful functor*.

Moreover, when $\mathscr{A}$ above is a part of a theory $(\mathscr{A}, a')$ and f is a theory morphism, $[\![f, \mathscr{K}]\!]$ is a *reinterpretation functor*. It takes models and compatible morphisms of $\mathscr{T}$ to models and compatible morphisms of $\mathscr{A}$.

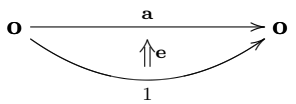
By the dualities, all of the above applies to $[-, -]$.

Example 3.4.2. The functor $[\mathbf{a}, \mathsf{Cat}] : [\mathsf{M}, \mathsf{Cat}] \longrightarrow [2, \mathsf{Cat}]$ takes monads to their underlying functors. Its action on arrows of $[\mathsf{M}, \mathsf{Cat}]$, monad morphisms, is that it takes (3.11) to



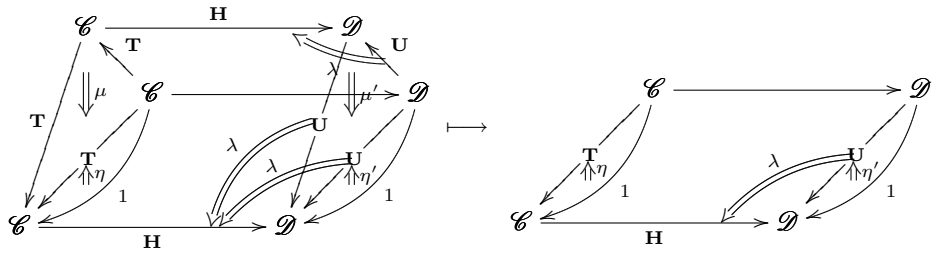
Similarly, $[\mathbf{o}, \mathsf{Cat}] : [\mathsf{M}, \mathsf{Cat}] \longrightarrow [1, \mathsf{Cat}]$ gives the underlying category for each monad. On arrows, it takes morphisms (3.11) to the underlying functor between the underlying categories, $\mathbf{H} : \mathscr{C} \longrightarrow \mathscr{D}$.

Example 3.4.3 (Pointed Functors)**.** A so-called *pointed functor* is an endofunctor $f : \mathscr{C} \longrightarrow \mathscr{C}$ with a natural transformation $\eta : 1_{\mathscr{C}} \Longrightarrow f$. Formally, the theory for a *pointed functor* is 2 with an extra nonidentity 2-cell, $\mathbf{e} : 1_{\mathbf{o}} \Longrightarrow \mathbf{a}$:



The category generated by this diagram, the theory of pointed functors, which is denoted Pt, is included in the obvious way in the theory of monads, M. The inclusion is

called $\mathbf{pt} : \mathsf{Pt} \hookrightarrow \mathsf{M}$ and by definition it preserves $\mathbf{a}$. Therefore, there is the *underlying pointed functor* $\mathbf{T} \cdot \mathbf{pt} \ : \ \mathsf{Pt} \longrightarrow \mathscr{K}$, for every monad $\mathbf{T} : \mathsf{M} \longrightarrow \mathsf{Cat}$. On arrows, the action of $[\mathbf{pt}, \mathsf{Cat}]$ is:



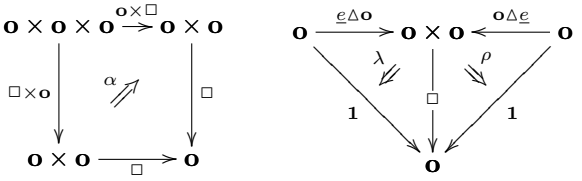
3.5 Examples

The following examples show some basic instances of functors with structure arising directly from rewriting definitions of categorical notions as sketches. The definitions can be found in Category-theoretical textbooks such as [Mac97] or [BW99].

Example 3.5.1 (Comonads)**.** The theory of a *comonad* is M^{co}, i.e. the theory for monads (Ex. 3.2.6) with 2-cells reversed. A model of M^{co} in Cat is clearly a comonad.

Example 3.5.2 (Monoidal categories)**.** A *monoidal category* is a category, $\mathscr{C}$, equipped with a binary operation, $\Box : \mathscr{C} \times \mathscr{C} \longrightarrow \mathscr{C}$, which is associative and has a left and right unit. One often considers the associativity and unit laws relaxed from strict equality to existence of isomorphisms, α, λ, ρ. Moreover one often considers a weaker *lax* variant, where α, λ and ρ above are not isomorphisms but just unidirectional natural transformations. This makes sense in particular in Computer Science where the associativity law would be realised (witnessed) by an algorithm taking $x\Box(y\Box z)$ to $(x\Box y)\Box z$. The adjective *strong* is then used for the case with isomorphisms.

These definitions can be generalised to take place in an arbitrary 2-category instead of Cat and sketched as follows (with the equations omitted):



(3.16)

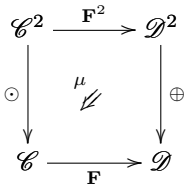
Here it is assumed that the vertices $\mathbf{o} \times \mathbf{o}$, $\mathbf{o} \times \mathbf{o} \times \mathbf{o}$ become real products of the model of $\mathbf{o}$ of the theory. The theory of this sketch is denoted Mon. The arrow of Mon is the arrow $\Box : \mathbf{o} \times \mathbf{o} \longrightarrow \mathbf{o}$. A product preserving model of Mon in Cat is a lax monoidal category (more precisely, the value of the model on $\mathbf{o}$ is).

To obtain a strong monoidal category from (3.16), its reversed copy would have to be included as well, together with equations stating that the two are inverse.

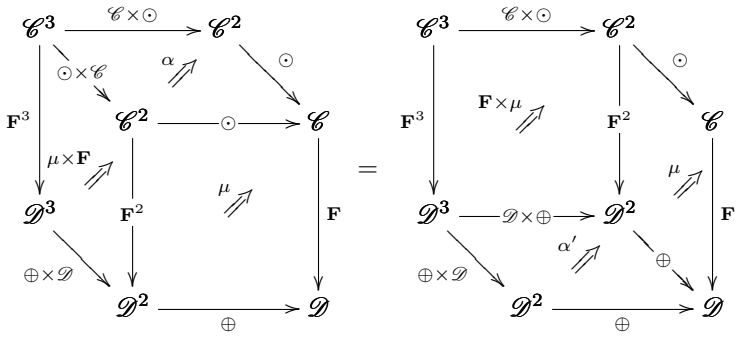
A morphism of lax monoidal categories is a lax monoidal functor. The following example shows that all the usual coherence conditions follow from this definition, see Sect. 3.3.

Example 3.5.3 (Lax Monoidal Functors)**.** For monoidal categories $\mathscr{C}$ and $\mathscr{D}$ (Ex. 3.5.2), a lax monoidal functor is a functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$ that is laxly coherent with the monoidal structures on $\mathscr{C}$ and $\mathscr{D}$. Explicitly, this is a morphism in $[\mathsf{Mon}, \mathsf{Cat}]$, whose

underlying arrow is pictured below:

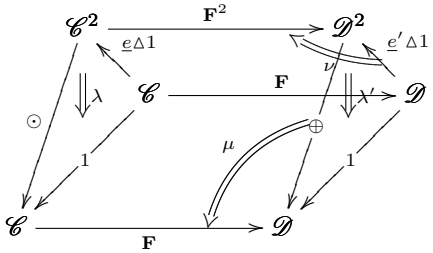


The usual coherence conditions (see [Mac97]) follow from this definition. The coherence with associativity, α, is the following condition :



(3.17)

Coherence with the left units, λ, λ' is a lax morphism of lax triangles:

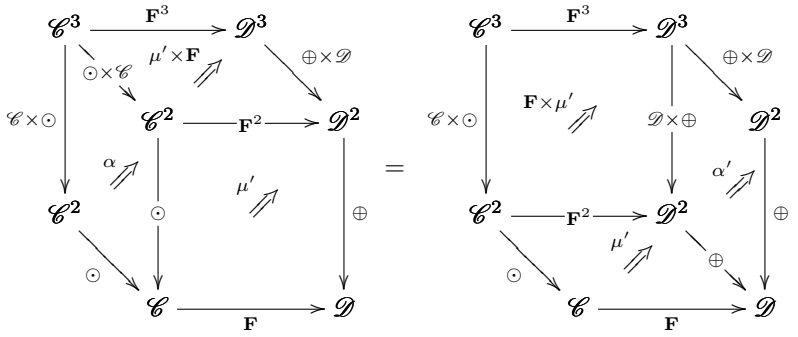


(3.18)

Right units are symmetrical. Note that (3.18) contains another 2-cell, $\nu : \underline{e}' \triangle \mathbf{F} \Longrightarrow \mathbf{F}\underline{e} \triangle \mathbf{F}$, which splits into $\nu_1 : \underline{e}' \Longrightarrow \mathbf{F}\underline{e}$, i.e. a morphism on units, and $\nu_2 : \mathbf{F} \Longrightarrow \mathbf{F}$, which is necessarily the identity by the definition of a lax natural transformation.

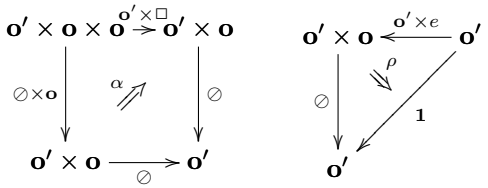
Example 3.5.4 (Forward Lax Monoidal Functors)**.** Example 3.5.3 can be reversed by considering arrows in $[\![\mathsf{Mon}, \mathsf{Cat}]\!]$. The resulting notion is given by a 2-cell (natural transformation) $\mu' : \mathbf{F} \circ \odot \Longrightarrow \oplus \circ \mathbf{F}^2$. For instance, the coherence with α changes

appropriately:



(3.19)

Example 3.5.5 (Right Monoidal Action)**.** A notion similar to (3.5.2) above is that of a right *action of a monoidal category*. In an arbitrary 2-category, $\mathscr{K}$, let X be a monoidal object, i.e. an underlying object of a model of a monoidal category. A right action of X on another object, Y, is given by a 1-cell $\oslash : Y \times X \longrightarrow Y$, which is associative and has a right unit. Formally, this is sketched as follows:



(3.20)

It is assumed that the sketch also contains a copy of (3.16) specifying that $\mathbf{o}$ is monoidal, and a reversed copy together with equations making α, ρ into formal isomorphisms, as discussed above. In a *lax right monoidal action* this is omitted and α, ρ are just 2-cells. The arrow of the theory of this sketch is $\oslash : \mathbf{o}' \times \mathbf{o} \longrightarrow \mathbf{o}'$. The image of $\oslash$ under a product preserving functor, $\mathbf{F}$, into $\mathscr{K}$ is a *lax right action* of $\mathbf{F}(\mathbf{o})$ on $\mathbf{F}(\mathbf{o}')$. The theory of the described sketch is denoted RAct.

Morphisms of certain monoidal actions are so-called *strong functors*. The details will be discussed in the next section, after we have introduced distributive laws as generalised morphisms.

Example 3.5.6. An adjoint in an arbitrary 2-category $\mathscr{K}$ is just an image of Def. 2.2.16 regarded as a sketch. There are two ways to regard an adjunction as a functor with structure, depending on which adjoint, left or right, is considered as *the arrow* of the functor with structure. Formally, there are two theories of functors with structure, LAdj and RAdj of *left adjoints* and *right adjoints*. They have the came category but the arrow of the theory, i.e. the inclusion from 2 differs.

Chapter 4

Generalised Distributive Laws

Reynolds in [Rey83] showed that parametric functions in polymorphic lambda calculus correspond to natural transformations in category theory. Among all natural transformations that occur in programming, we are interested in those of types similar to $\mathbf{FG} \Longrightarrow \mathbf{G'F}$ and $\mathbf{F'G} \Longrightarrow \mathbf{GF}$, which are coherent with any additional structure on the $\mathbf{F}$'s and $\mathbf{G}$'s, because they repeatedly occur in many seemingly unrelated situations in computer science and the related category theory.

Superficially, one can notice the resemblance to homomorphisms in abstract algebra, namely to coherence conditions of the form $f(x \cdot y) = f(x) \bullet f(y)$. This connection has been formalised in the previous chapter where it was shown that such natural transformations are the underlying 2-cells of coherent morphisms of functors with structure. Quite often, though, all functors bear an additional structure and one then wants the natural transformation to be coherent with all of them. Such a situation gives rise to the notion of a distributive law of one functor with structure over another.

In category theory, an example was described by Jon Beck in [Bec69]: a natural transformation $\lambda : \mathbf{TU} \Longrightarrow \mathbf{UT}$ where $\mathbf{T}$, $\mathbf{U}$ are monads and λ is coherent with the structure of the monads is called a *distributive law* by Beck. When the monads are those of an abelian group and a multiplicative monoid, respectively, a distributive law categorically formalises the usual algebraic notion of a distributive law joining them into (the monad of) a ring. Many variants of the notion, which are straightforwardly derived from Beck's notion, have been studied since then. This includes cases when $\mathbf{U}$ above is a comonad, or either of $\mathbf{T}$ or $\mathbf{U}$ is just a (co)pointed functor.

In this chapter, a notion of a *generalised distributive law* of functors with structure is introduced. The generalisation is in the type of the structure on the underlying functors. The notion is defined and it is shown that Beck's distributive laws of monads are an instance. It is also shown, and it follows rather straightforwardly from the definition and the general 2-categorical results of Gray, that a generalised distributive law of functors with structure $\mathbf{T}, \mathbf{T}'$ with theory $\mathscr{T}$ over functors with structure $\mathbf{U}$, $\mathbf{U}'$ with

theory $\mathscr{U}$ is at the same time a coherent morphism of the $\mathbf{T}$'s and a coherent morphism of the $\mathbf{U}$'s.

The point of generalising Beck's distributive laws is that there are important cases of functors with structure, which are not subsumed by monads or comonads, but whose distributive laws make sense. And although the theory becomes weaker, there are still interesting properties of generalised distributive laws irrespective of monads. An important example is the case of higher-dimensional distributive laws where there are more than two functors with structure involved, which is studied in Sect. 4.3.

The rest of this chapter comprises of detailed examples, which illustrate that the notion of generalised distributive laws conceptually unifies many diverse notions in computer science.

4.1 Definition and Characterisation

This chapter is about the following definition.

Definition 4.1.1 (Generalised distributive law)**.** *Let $\mathscr{T}$, $\mathscr{U}$ be theories of functors with structure, $\mathscr{K}$ a 2-category. A* (generalised) distributive law of $\mathscr{T}$ over $\mathscr{U}$ in $\mathscr{K}$ *is a 2-functor $\mathbf{F} : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ where $\otimes$ is Gray's tensor product of 2-categories (Sect. 2.4).*

The intuition and nomenclature of informal distributive laws is that a distributive law of functor $\mathbf{F}$ over functor $\mathbf{G}$, both with some additional structure, is a natural transformation $\lambda : \mathbf{FG} \Longrightarrow \mathbf{G'F'}$, where $\mathbf{F}$ and $\mathbf{F'}$, $\mathbf{G}$ and $\mathbf{G'}$ have pairwise the same kind of additional structure, such that some coherence conditions expressing the coherence of λ with the additional structure hold. This is honoured by our generalised definition in the sense that when $\mathscr{T}$ and $\mathscr{U}$ have only one arrow, $\mathbf{a}$, it follows, from the fact that 2-functors of type $\mathscr{C} \otimes \mathscr{D} \longrightarrow \mathscr{E}$ are isomorphic to quasi functors of two variables $\mathscr{C} \times \mathscr{D} \rightsquigarrow \mathscr{E}$ and by the definition of a quasi functor (Def. 2.4.3), that a distributive law of such $\mathscr{T}$ over $\mathscr{U}$ in Cat is a natural transformation $\mathbf{F}(\mathbf{a}, \mathbf{t}) \circ \mathbf{F}(\mathbf{s}, \mathbf{a}) \Longrightarrow \mathbf{F}(\mathbf{t}, \mathbf{a}) \circ \mathbf{F}(\mathbf{a}, \mathbf{s})$. In the general case, one obtains for each arrow $t : A \longrightarrow A' \in \mathscr{T}$ and $u : B \longrightarrow B' \in \mathscr{U}$ a 2-cell $\alpha_{t,u} : \mathbf{F}(t, B') \circ \mathbf{F}(A, u) \Longrightarrow \mathbf{F}(A', u) \circ \mathbf{F}(t, B)$. In this sense, $\mathbf{F}$ is taking the model of $\mathscr{T}$ over the model of $\mathscr{U}$.

Remark 4.1.2. Note that the arrows of the functors with structure involved in a distributive law are not explicitly mentioned in the definition of a distributive law. It plays a bookkeeping role allowing us to point to the 2-cell (often natural transformation) "swapping the order of two functors", appropriately generalised.

More importantly, though, it serves as a reminder (or a seed of) of a notion of a structure of the functors. Later, in Section 4.3 we realise that it is important not to forget about it. See Remark 4.3.4 if you like to take a to peek ahead.

We summarise the above considerations precisely in the following characterisation theorem.

Theorem 4.1.3 (Characterisation)**.** *A distributive law of $\mathscr{T}$ over $\mathscr{U}$, in $\mathscr{K}$, formally $\lambda : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$, is equivalently:*

1. *a model of $\mathscr{U}$ in $[\mathscr{T}, \mathscr{K}]$, explicitly a functor $\lambda^{\downarrow} : \mathscr{U} \longrightarrow [\mathscr{T}, \mathscr{K}]$; so-called* vertical distributive law

2. *a model of $\mathscr{T}$ in $[\![\mathscr{U}, \mathscr{K}]\!]$, explicitly a functor $\lambda^{\rightarrow} : \mathscr{T} \longrightarrow [\![\mathscr{U}, \mathscr{K}]\!]$; so-called* horizontal distributive law

3. *It follows that λ is a reverse morphism of $\mathscr{T}$-functors and at the same time a forward morphism of $\mathscr{U}$-functors and is in this sense coherent with the additional structures of both $\mathscr{T}$ and $\mathscr{U}$.*

Proof. The result follows immediately from the isomorphisms in Theorem 2.4.1. Namely, a functor $\lambda : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ is an object in $[\mathscr{T} \otimes \mathscr{U}, \mathscr{K}] \cong [\mathscr{U}, [\mathscr{T}, \mathscr{K}]]$. Similarly for $\lambda \in [\![\mathscr{T} \otimes \mathscr{U}, \mathscr{K}]\!] \cong [\![\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]]\!]$. The morphisms of functors with structure in (3) are obtained as the underlying arrows of (2) and (1) because $\mathscr{T}$ and $\mathscr{U}$ are theories of functors with structure, i.e. with an inclusion $2 \longrightarrow \mathscr{T}, \mathscr{U}$. □

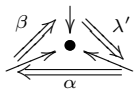
Remark 4.1.4. Note that a distributive law $\lambda : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ is an object in two other 2-categories, namely $[\![\mathscr{U}, [\mathscr{T}, \mathscr{K}]]\!]$ and $[\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]]$. These last two categories are equivalent by (2.37), however, they are not equivalent to either of $[\![\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]]\!]$ or $[\mathscr{U}, [\mathscr{T}, \mathscr{K}]]$ by any of the theorems or dualities. To understand the differences between these three nonequivalent categories, first note that objects in $[\mathscr{A}, [\mathscr{B}, \mathscr{K}]]$ are the same as objects in $[\![\mathscr{B}, [\![\mathscr{A}, \mathscr{K}]\!]]\!]$ and in $[\![\mathscr{A}, [\mathscr{B}, \mathscr{K}]]\!] \cong [\mathscr{B}, [\![\mathscr{A}, \mathscr{K}]\!]]$. In all three cases they comprise of laxly commuting squares

$$\begin{array}{ccc} \lambda_{A,B} & \xrightarrow{\lambda_{A,g}} & \lambda_{A,B'} \\ {\scriptstyle \lambda f,B}\downarrow & \overset{\lambda_{f,g}}{\nearrow\!\!\nearrow} & \downarrow{\scriptstyle \lambda f,B'} \\ \lambda_{A',B} & \xrightarrow[\lambda A',g]{} & \lambda_{A',B'} \end{array}$$

for each pair of arrows $f : A \longrightarrow A'$ in $\mathscr{A}$ and $g : B \longrightarrow B'$. The morphisms in all three categories are laxly commuting cubes, but the difference lies in the direction of the diagonal 2-cells defining the components of the morphisms. Figure 4.1 illustrates morphisms in $[\mathscr{A}, [\mathscr{B}, \mathscr{K}]]$, $[\![\mathscr{B}, [\![\mathscr{A}, \mathscr{K}]\!]]\!]$ and $[\![\mathscr{A}, [\mathscr{B}, \mathscr{K}]]\!]$, respectively.

Note that by purely geometrical considerations, one would expect there to be four, not three different notions of morphisms between lax squares. However, it is clearly

seen that the fourth variant missing from Fig. 4.1, i.e. when when α is reverse, β forward, results in a cycle of 2-cells



around the front-bottom-right corner. And there is a similar diagram around the back-top-left corner. Thus the cube doesn't make sense as coherence condition.

This answers the question posed in [PW99] as to why there are only six not eight (there, duals are counted too) categories of distributive laws of a monad over a comonad.

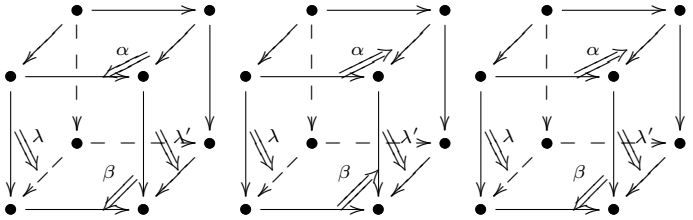


Figure 4.1: Morphisms of distributive laws are laxly commuting cubes; here depicted as a morphism $\lambda \longrightarrow \lambda'$ in $[\mathscr{A}, [\mathscr{B}, \mathscr{K}]]$, $[\![\mathscr{B}, [\![\mathscr{A}, \mathscr{K}]\!]]\!]$, $[\![\mathscr{A}, [\mathscr{B}, \mathscr{K}]]\!]$, respectively from left to right; 2-cells on opposing hidden faces have the same direction.

The following example illustrates the decomposition of distributive laws into morphisms as carried out in Theorem 4.1.3. It also relates generalised distributive laws to Beck's distributive laws of monads.

Example 4.1.5 (Distributive laws of Monads)**.** Let $\mathscr{C}$ be an ordinary category, and $(\mathbf{T}, \mu, \eta)$ and $(\mathbf{U}, \nu, \zeta)$ a pair of monads on $\mathscr{C}$. In [Bec69] a distributive law of $\mathbf{T}$ over $\mathbf{U}$ is defined as a natural transformation $\lambda : \mathbf{TU} \Longrightarrow \mathbf{UT}$ such that:

$$\begin{array}{lll} \lambda \cdot \eta\mathbf{U} &= \mathbf{U}\eta & \qquad \lambda \cdot \mu\mathbf{U} &= \mathbf{U}\mu \cdot \lambda\mathbf{T} \cdot \mathbf{T}\lambda \\ \lambda \cdot \mathbf{T}\zeta &= \zeta\mathbf{T} & \qquad \lambda \cdot \mathbf{T}\nu &= \nu\mathbf{T} \cdot \mathbf{U}\lambda \cdot \lambda\mathbf{U} \end{array} \tag{4.1}$$

A distributive law $\lambda : \mathsf{M} \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$ is a distributive law of a monad over a monad in the above sense. In order to see this, observe that commutativity of the diagram in Fig. 4.2 is equivalent to equations (4.1). At the same time, the left-hand side of the diagram pictures an endomorphism $(\mathbf{T}, \lambda) : \mathbf{U} \longrightarrow \mathbf{U}$ (see Ex. 3.3.4), and the right-hand side pictures modifications μ and η, all in $[\![\mathsf{M}, \mathscr{K}]\!]$. Therefore[1], Fig. 4.2 is a model of a monad in $[\![\mathsf{M}, \mathscr{K}]\!]$. By Theorem 4.1.3, this is the same as a distributive law $\lambda : \mathsf{M} \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$. Similarly, the right-hand side pictures a reverse morphism $(\mathbf{U}, \lambda) : \mathbf{T} \longrightarrow \mathbf{T}$ and the left-hand side 2-modifications in $[\mathsf{M}, \mathscr{K}]$. These two views correspond to points (1) and (2) in Theorem 4.1.3.

[1] Although it remains to establish the associativity and unit laws.

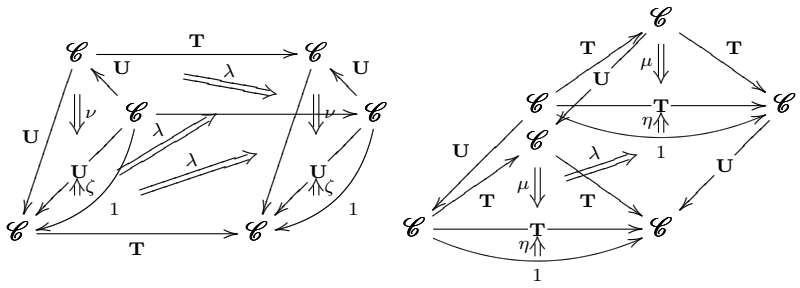


Figure 4.2: Distributive law $\lambda : \mathsf{M} \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$ of a monad over a monad.

Example 4.1.6. For comonads $(\mathbf{D} : \mathscr{C} \longrightarrow \mathscr{C}, \delta, \epsilon)$ and $(\mathbf{E} : \mathscr{C} \longrightarrow \mathscr{C}, \gamma, \xi)$ a distributive law of $\mathbf{D}$ over $\mathbf{E}$ is usually defined as a natural transformation $\lambda : \mathbf{DE} \Longrightarrow \mathbf{ED}$ such that the duals of (4.1) hold. To see that $\lambda : \mathsf{M}^{\mathsf{co}} \otimes \mathsf{M}^{\mathsf{co}} \longrightarrow \mathsf{Cat}$ is a distributive law of a comonad over a comonad, it is obviously possible to expand all definitions again. On the other hand, one can see this easily by duality from the previous example of monads:

$$[\mathsf{M}, [\![\mathsf{M}, \mathscr{K}^{\mathsf{co}}]\!]] \;\cong\; [\mathsf{M}, [\mathsf{M}^{\mathsf{co}}, \mathscr{K}]^{\mathsf{co}}] \;\cong\; [\![\mathsf{M}^{\mathsf{co}}, [\mathsf{M}^{\mathsf{co}}, \mathscr{K}]]\!]^{\mathsf{co}}$$

So a distributive law of monads in $\mathscr{K}^{\mathsf{co}}$ is a distributive law of comonads in $\mathscr{K}$. It follows that the coherence conditions of the latter are just dual of those of the former.

The definition of generalised distributive laws, seen through Theorem 4.1.3, entails the result of Ross Street about formal monads and distributive laws of monads in [Str72]. Street explicitly defines the category of monads and their morphisms in an arbitrary 2-category, $\mathscr{K}$. This category is denoted $\mathsf{Mnd}(\mathscr{K})$. His definition of $\mathsf{Mnd}(\mathscr{K})$ is such that

$$\mathsf{Mnd}(\mathscr{K}) \;\equiv\; [\mathsf{M}, \mathscr{K}]$$

in the present notation. Street further observes that a monad in $\mathsf{Mnd}(\mathscr{K})$ is a distributive law of a monad over a monad. In the present notation this is just

$$\mathsf{Mnd}(\mathsf{Mnd}(\mathscr{K})) \equiv [\mathsf{M}, [\mathsf{M}, \mathscr{K}]] \cong [\mathsf{M} \otimes \mathsf{M}, \mathscr{K}]$$

The significance of our generalisation form $\mathsf{Mnd}(\mathscr{K})$ to the functor category $[\mathsf{M}, \mathscr{K}]$ is that one can start to formally study distributive laws of other kinds than monads and the relations between the different notions. And it turns out that the generalised notion of a distributive law is meaningful and useful. This is demonstrated in the rest of this thesis.

4.2 Some Useful Notation

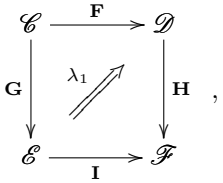
Any distributive law $\lambda : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ has two underlying arrows, and therefore two source and two target functors with structure. The following definition fixes the notation.

Definition 4.2.1. *For a distributive law $\lambda : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$, equivalently $\lambda^{\rightarrow} : \mathscr{T} \longrightarrow [\![\mathscr{U}, \mathscr{K}]\!]$, or $\lambda^{\downarrow} : \mathscr{U} \longrightarrow [\mathscr{T}, \mathscr{K}]$, by the action of the contravariant forgetful functors one obtains the following:*

1. horizontal domain *and* codomain *as the domain and codomain, respectively, of* $\lambda^{\rightarrow}$
2. vertical domain *and* codomain *as the domain and codomain, respectively, of* $\lambda^{\downarrow}$
3. underlying horizontal morphism *as the underlying arrow of* $\lambda^{\rightarrow}$
4. underlying vertical morphism *as the underlying arrow of* $\lambda^{\downarrow}$
5. underlying 2-cell *as the underlying arrow of the underlying horizontal (or equivalently vertical) morphism of* λ

Example 4.2.2. Here are some illustrating examples.

1. A distributive law of 2 over 2 in Cat (i.e. when $\mathscr{T}$ and $\mathscr{U}$ are just 2 above) is given by the following data:



   where $\mathscr{C}$, $\mathscr{D}$, $\mathscr{E}$, $\mathscr{F}$ are categories, $\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$, $\mathbf{I}$ are ordinary functors and λ_1 is an ordinary natural transformation.

   (a) $\lambda_1^{\rightarrow}$ is the morphism $(\mathbf{F}, \lambda_1, \mathbf{I}) : \mathbf{G} \longrightarrow \mathbf{H}$ in $[\![2, \mathsf{Cat}]\!]$

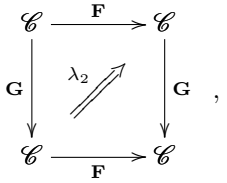
   (b) $\lambda_1^{\downarrow}$ is the morphism $(\mathbf{G}, \lambda_1, \mathbf{H}) : \mathbf{F} \longrightarrow \mathbf{I}$ in $[2, \mathsf{Cat}]$

   (c) The horizontal domain is $\mathbf{G}$, codomain $\mathbf{H}$, the vertical domain is $\mathbf{F}$ and codomain $\mathbf{I}$

   (d) The underlying horizontal and vertical morphisms are in this case the same as $\lambda_1^{\rightarrow}$ and $\lambda_1^{\downarrow}$, respectively.

   (e) The underlying 2-cell is just λ_1

2. A distributive law of O over O in Cat is given by the following data:



where $\mathscr{C}$ is a category, $\mathbf{F}$, $\mathbf{G}$ are ordinary functors and λ_2 is an ordinary natural transformation $\lambda_2 : \mathbf{FG} \Longrightarrow \mathbf{GF}$.

   (a) $\lambda_2^{\rightarrow}$ is the morphism $(\mathbf{F}, \lambda_2) : \mathbf{G} \longrightarrow \mathbf{G}$ in $[\![\mathsf{O}, \mathsf{Cat}]\!]$

   (b) $\lambda_2^{\downarrow}$ is the morphism $(\mathbf{G}, \lambda_2) : \mathbf{F} \longrightarrow \mathbf{F}$ in $[\mathsf{O}, \mathsf{Cat}]$

   (c) The horizontal domain and codomain is $\mathbf{G}$, the vertical domain and codomain is $\mathbf{F}$

   (d) The underlying horizontal and vertical morphisms are also in this case the same as $\lambda_2^{\rightarrow}$ and $\lambda_2^{\downarrow}$, respectively.

   (e) The underlying 2-cell is just λ_2

3. For λ as in Ex. 4.1.5

   (a) Both horizontal domain and codomain is the monad $\mathbf{U}$

   (b) Both vertical domain and codomain is the monad $\mathbf{T}$

   (c) The underlying horizontal and vertical morphisms are pictured in Fig. 4.2 left and right, respectively

   (d) The underlying 2-cell is the natural transformation $\lambda_3 : \mathbf{TU} \Longrightarrow \mathbf{UT}$

Notation 4.2.3. A distributive law λ of $\mathscr{T}$ over $\mathscr{U}$ with vertical source $\mathbf{T}$, vertical target $\mathbf{T}'$, horizontal source $\mathbf{U}$, horizontal target $\mathbf{U}'$ is denoted

$$(\mathscr{U}, \mathbf{U}) \Big| \frac{\overset{(\mathscr{T},\mathbf{T})}{}}{\underset{(\mathscr{T},\mathbf{T}')}{\lambda}} \Big| (\mathscr{U}, \mathbf{U}') \quad , \quad \text{or just} \quad \mathbf{U} \Big| \frac{\overset{\mathbf{T}}{}}{\underset{\mathbf{T}'}{\lambda}} \Big| \mathbf{U}' \quad ,$$

when $\mathscr{U}$, $\mathscr{T}$ are clear from the context.

The underlying 2-cell of λ, of type $\mathbf{T}'\mathbf{a} \circ \mathbf{Ua} \Longrightarrow \mathbf{U}'\mathbf{a} \circ \mathbf{Ta}$ is denoted $|\overline{\lambda}|$.

Example 4.2.4. The distributive laws of Example 4.2.2 are denoted respectively:

1. $(2, \mathbf{G}) \Big| \frac{\overset{(2,\mathbf{F})}{}}{\underset{(2,\mathbf{I})}{\lambda_1}} \Big| (2, \mathbf{H})$ or just $\mathbf{G} \Big| \frac{\overset{\mathbf{F}}{}}{\underset{\mathbf{I}}{\lambda_1}} \Big| \mathbf{H}$

2. $(\mathsf{O}, \mathbf{G}) \Big| \frac{\overset{(\mathsf{O},\mathbf{F})}{}}{\underset{(\mathsf{O},\mathbf{F})}{\lambda_2}} \Big| (\mathsf{O}, \mathbf{G})$ or just $\mathbf{G} \Big| \frac{\overset{\mathbf{F}}{}}{\underset{\mathbf{F}}{\lambda_2}} \Big| \mathbf{G}$

3. $(\mathsf{M},\mathbf{U})|\,\dfrac{\overset{(\mathsf{M},\mathbf{T})}{}}{\underset{(\mathsf{M},\mathbf{T})}{\lambda_3}}\,|(\mathsf{M},\mathbf{U})$ or just $\mathbf{U}|\,\dfrac{\overset{\mathbf{T}}{}}{\underset{\mathbf{T}}{\lambda_3}}\,|\mathbf{U}$

4.3 Higher-dimensional Distributive Laws

A distributive law of $(\mathscr{T},a)$ over $(\mathscr{U},b)$ in $\mathscr{K}$ is a model of $\mathscr{T}$ in the category of models of $\mathscr{U}$ in $\mathscr{K}$. This construction can be iterated by considering $\mathscr{K}$ itself to be a category of models of a functor with structure in a $\mathscr{K}'$. And so on. The following is a preliminary definition of what we have in mind.

Definition 4.3.1 (Iterated Distributive Law)**.** *For a 2-category $\mathscr{K}$, and a theory of functors with structure $(\mathscr{T},a)$,*

1. *a category of* 1-iterated distributive laws *in $\mathscr{K}$ is a category of models of $\mathscr{T}$ in $\mathscr{K}$, i.e either $[\mathscr{T},\mathscr{K}]$ or $[\![\mathscr{T},\mathscr{K}]\!]$.*

2. *a category of* n-iterated distributive laws*, for $n>1$, in $\mathscr{K}$ is a model of $\mathscr{T}$ in a category of (n-1)-iterated distributive laws.*

Note that a functor $\mathbf{F}:\mathscr{U}\longrightarrow\mathscr{K}$ assigns to each $g:B\longrightarrow B'$ in $\mathscr{U}$ an arrow $\mathbf{F}(g)$ in $\mathscr{K}$. A functor $\mathbf{H}$ in $[\![\mathscr{T},[\![\mathscr{U},\mathscr{K}]\!]]\!]$ assigns to each pair of arrows $g:B\longrightarrow B'$ in $\mathscr{U}$ and $f:A\longrightarrow A'$ in $\mathscr{T}$ a laxly commuting square in $\mathscr{K}$:

$$\begin{array}{ccc}
\mathbf{H}(A,B) & \xrightarrow{\mathbf{H}(f,B)} & \mathbf{H}(A',B) \\
{\scriptstyle \mathbf{H}(A,g)}\downarrow & \swarrow\!\!\!\swarrow\ {\scriptstyle \mathbf{H}(f,g)} & \downarrow{\scriptstyle \mathbf{H}(A',g)} \\
\mathbf{H}(A,B') & \xrightarrow[\mathbf{H}(f,B')]{} & \mathbf{H}(A',B')
\end{array}$$

See (2.39) and (2.38). Now, in the case when $\mathscr{K}$ above is a functor category, say $[\![\mathscr{V},\mathscr{K}]\!]$, so $\mathbf{H}\in[\![\mathscr{T},[\![\mathscr{U},[\![\mathscr{V},\mathscr{K}]\!]]\!]]\!]$, one obtains a laxly commuting square in the functor category $[\![\mathscr{V},\mathscr{K}]\!]$. This is a lax cube. The vertices are the functors $\mathbf{H}(A,B,-)$, for $A\in\mathscr{T}$, $B\in\mathscr{U}$. The four sides are the lax natural transformations $\mathbf{H}(f,B,-)$, $\mathbf{H}(f,B',-)$, $\mathbf{H}(A,g,-)$, $\mathbf{H}(A',g,-)$, for f and g as above. The diagonal is the modification $\mathbf{H}(f,g,-)$, with a component $\mathbf{H}(f,g,C)$ for every $C\in\mathscr{V}$, such that for any $g:C\longrightarrow C'$ the cube in Fig. 4.3 commutes. Informally, it seems, and it is straightforward to verify from the definitions, that ordinary (2-iterated) distributive laws are given by lax squares, 3-iterated distributive laws by lax cubes, 4-iterated distributive laws by lax tesseracts (Fig. 4.6), etc., for all tuples of arrows. In this sense, iterated distributive laws are higher-dimensional.

Note that written out as diagrams of arrows in $\mathscr{K}(\mathbf{H}(f,g,h),\mathbf{H}(f',g',h'))$, (4.2)

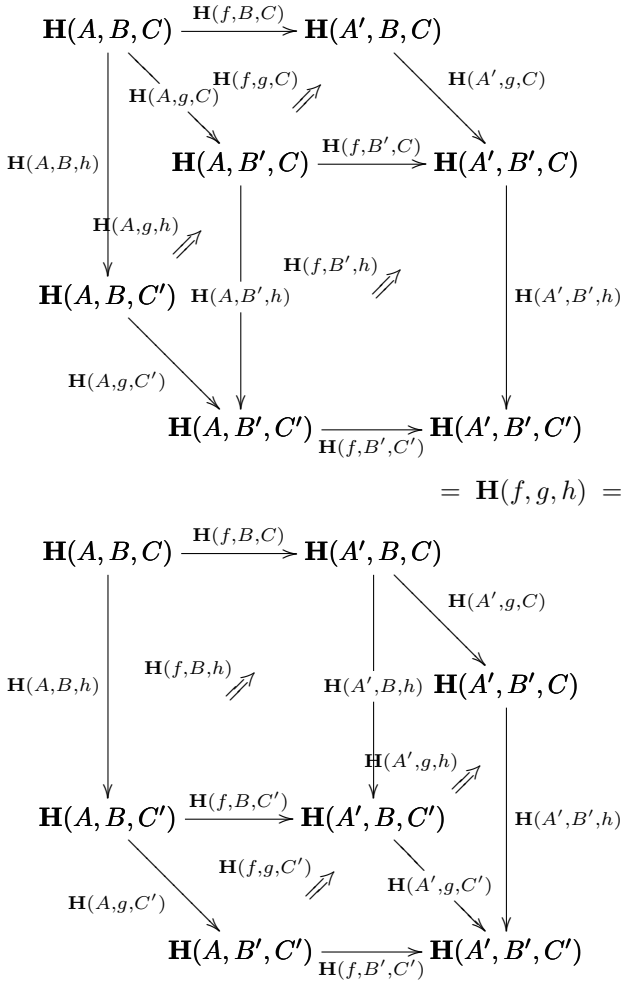


Figure 4.3: Yang-Baxter equation

is

$$\mathbf{H}(A',B',h)\mathbf{H}(f,g,C)\cdot\mathbf{H}(f,B',h)\mathbf{H}(A,g,C)\cdot\mathbf{H}(f,B',C')\mathbf{H}(A,g,h) \;=\; \mathbf{H}(A,g,h)\mathbf{H}(f,B,C)\cdot\mathbf{H}(A',g,C')\mathbf{H}(f,B,h)\cdot\mathbf{H}(f,g,C')\mathbf{H}(A,B,h) \quad (4.3)$$

This is a variant of so-called *Yang-Baxter equation*. It is perhaps better known and recognisable as the hexagon in Fig. 4.4.

Example 4.3.2 (Tesseract)**.** A 4-iterated distributive law $\mathbf{T} : 2 \otimes 2 \otimes 2 \otimes 2 \longrightarrow \mathscr{K}$ is a tesseract (4-dimensional cube), Fig. 4.6. It arises as a morphism of lax cubes, say inside-to-outside. The morphism has by definition a component, a lax cube, for each

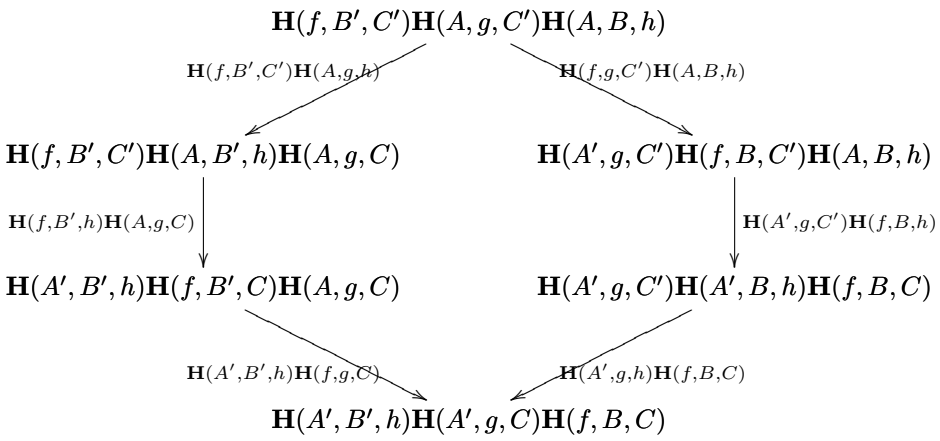


Figure 4.4: Yang-Baxter equation

face of the source cube, which connects the source face to the corresponding target face.

So how can one characterise general n-iterated distributive laws for $n > 3$ and for other combinations of categories $[\![-,-]\!]$, $[-,-]$? Is iteration a good approach to distributive laws of more than two theories, as it has already been observed that even in the 2-iterated case not all combinations of categories $[\![-,-]\!]$ and $[-,-]$ and permutations of theories yield different distributive laws. Just as in the case of 2-iterated distributive laws, it would be useful to have a normal form of an n-iterated distributive law. As $\otimes$ is associative, it is natural to seek such a normal form as a functor from an iterated Gray's tensor.

Definition 4.3.3 (Higher-dimensional Distributive Law)**.** *For theories* $(\mathscr{T}_i, a_i),\ i \leq n$*, and 2-category* $\mathscr{K}$*, an* n-dimensional distributive law of $\mathscr{T}_n$ over $\mathscr{T}_{n-1}$, etc., in $\mathscr{K}$ *is a 2-functor*

$$\lambda\ :\ \mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_n \longrightarrow \mathscr{K} \tag{4.4}$$

Remark 4.3.4. Even though $\otimes$ is associative it is not the case that any n-dimensional distributive law is equivalent to a 2-dimensional distributive law. The problem lies in the arrows of the theories as there is no canonical way to pick an arrow in $\mathscr{T}_i \otimes \mathscr{T}_{i+1}$ for a pair of theories $(\mathscr{T}_i, a_i)$, $(\mathscr{T}_{i+1}, a_{i+1})$.

We now investigate the relationship of higher-dimensional and iterated distributive

laws. It is immediate by associativity of $\otimes$ and iteration of (2.34) and (2.35) that

$$[\![\mathscr{T}_n \otimes \cdots \otimes \mathscr{T}_1, \mathscr{K}]\!] \cong [\![\mathscr{T}_n, [\![\ldots, [\![\mathscr{T}_1, \mathscr{K}]\!]]\!]]\!] \tag{4.5}$$

$$[\mathscr{T}_n \otimes \cdots \otimes \mathscr{T}_1, \mathscr{K}] \cong [\mathscr{T}_1, [\ldots, [\mathscr{T}_n, \mathscr{K}]]] \tag{4.6}$$

Similarly, the following is easy and useful.

Lemma 4.3.5. *Any functor (4.4), is a functor from $\mathscr{T}_i$, for any $1 \leq i \leq n$.*

Proof. There is the following chain of isomorphic sets of functors:

$$\begin{aligned}
\lambda \in\ & |[\![\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_n, \mathscr{K}]\!]| \\
\cong\ & |[\![(\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}) \otimes (\mathscr{T}_i \otimes \cdots \otimes \mathscr{T}_n), \mathscr{K}]\!]| \\
\cong\ & |[\![\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}, [\![\mathscr{T}_i \otimes \cdots \otimes \mathscr{T}_n, \mathscr{K}]\!]]\!]| \\
=\ & |[\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}, [\![\mathscr{T}_i \otimes \cdots \otimes \mathscr{T}_n, \mathscr{K}]\!]]| \\
\cong\ & |[\![\mathscr{T}_i \otimes \cdots \otimes \mathscr{T}_n, [\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}, \mathscr{K}]]\!]| \\
\cong\ & |[\![\mathscr{T}_i, [\![\mathscr{T}_{i+1} \otimes \cdots \otimes \mathscr{T}_n, [\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}, \mathscr{K}]]\!]]\!]
\end{aligned}$$

Objects in the last category are functors from $\mathscr{T}_i$ to $[\![\mathscr{T}_{i+1} \otimes \cdots \otimes \mathscr{T}_n, [\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_{i-1}, \mathscr{K}]]\!]$. Note that starting from $[\mathscr{T}_1 \otimes \cdots \otimes \mathscr{T}_n, \mathscr{K}]$ gives the same result under the isomorphism (2.37). □

But what about categories where $[-,-]$ and $[\![-,-]\!]$ are arbitrarily nested, such as

$$[\![\mathscr{T}, [\mathscr{U}, [\mathscr{V}, [\![\mathscr{D}, \mathscr{K}]\!]]]]\!] \ ? \tag{4.7}$$

Interestingly, any such iterated category is equivalent to a category of functors of the form (4.4). For example, using (2.34), (2.35) and (2.37), one can normalise (4.7) as follows:

$$\begin{aligned}
[\![\mathscr{T}, [\mathscr{U}, [\mathscr{V}, [\![\mathscr{D}, \mathscr{K}]\!]]]]\!] &\cong [\![\mathscr{T}, [\mathscr{V} \otimes \mathscr{U}, [\![\mathscr{D}, \mathscr{K}]\!]]]\!] \\
&\cong [\![\mathscr{T}, [\![\mathscr{D}, [\mathscr{V} \otimes \mathscr{U}, \mathscr{K}]]\!]]\!] \\
&\cong [\![\mathscr{T} \otimes \mathscr{D}, [\mathscr{V} \otimes \mathscr{U}, \mathscr{K}]]\!]
\end{aligned} \tag{4.8}$$

Objects in (4.8) are the same as objects in

$$[\mathscr{T} \otimes \mathscr{D}, [\mathscr{V} \otimes \mathscr{U}, \mathscr{K}]] \cong [\mathscr{V} \otimes \mathscr{U} \otimes \mathscr{T} \otimes \mathscr{D}, \mathscr{K}] \tag{4.9}$$

In general, let $\mathbf{Fun_f}(\mathscr{T}, \mathscr{U})$ stand for $[\![\mathscr{T}, \mathscr{U}]\!]$ and $\mathbf{Fun_r}(\mathscr{T}, \mathscr{U})$ stand for $[\mathscr{T}, \mathscr{U}]$ in the following theorem.

Theorem 4.3.6 (Higher-dimensional Normal Form)**.** *For theories $\mathscr{T}_i,\ i \leq n$, $2 < n$, and a 2-category $\mathscr{K}$, for any category*

$$\mathscr{H} \equiv \mathbf{Fun}_{x_1}(\mathscr{T}_1, \ldots, (\mathbf{Fun}_{x_n}(\mathscr{T}_n, \mathscr{K}))), \quad x_i \in \{\mathbf{f}, \mathbf{r}\} \tag{4.10}$$

there exists a permutation $s \in \mathbf{S}_n$ such that exactly one of the following holds:

1. *when $x_i = \mathbf{f}$, for all $i \leq n$,*

$$\mathscr{H} \cong \mathbf{Fun}_{\mathbf{f}}(\mathscr{T}_{s(1)} \otimes \cdots \otimes \mathscr{T}_{s(n)}, \mathscr{K})$$

2. *when $x_i = \mathbf{r}$, for all $i \leq n$,*

$$\mathscr{H} \cong \mathbf{Fun}_{\mathbf{r}}(\mathscr{T}_{s(1)} \otimes \cdots \otimes \mathscr{T}_{s(n)}, \mathscr{K})$$

3. *otherwise there exists a $k,\ 1 \leq k < n$ in*

$$\mathscr{H} \cong \mathbf{Fun}_{\mathbf{r}}(\mathscr{T}_{s(1)} \otimes \cdots \otimes \mathscr{T}_{s(k)}, \mathbf{Fun}_{\mathbf{f}}(\mathscr{T}_{s(k+1)} \otimes \cdots \otimes \mathscr{T}_{s(n)}, \mathscr{K}))$$

Proof. The theorem is proven by induction on the nesting depth of the definition (4.10). It clearly holds for $\mathscr{H}$'s of nesting depth 2. Let the nesting depth of $\mathscr{H}$ be n and let the inductive hypothesis hold for terms of nesting depth $< n$. Then depending on the shape of $\mathscr{H}$, exactly one of the following holds for arbitrary $\mathscr{T}$, $\mathscr{U}$, $\mathscr{V}$, $\mathscr{D}$, by (2.34)-(2.37):

1. $\mathscr{H} \equiv [\mathscr{T}, [\mathscr{U}, \mathbf{Fun}_x(\mathscr{V}, \mathscr{D})]] \cong [\mathscr{U} \otimes \mathscr{T}, \mathbf{Fun}_x(\mathscr{V}, \mathscr{D})]$
2. $\mathscr{H} \equiv [\mathscr{T}, [\![\mathscr{U}, [\mathscr{V}, \mathscr{D}]]\!]] \cong [\![\mathscr{U}, [\mathscr{T}, [\mathscr{V}, \mathscr{D}]]]\!] \cong [\![\mathscr{U}, [\mathscr{V} \otimes \mathscr{T}, \mathscr{D}]]\!]$
3. $\mathscr{H} \equiv [\mathscr{T}, [\![\mathscr{U}, [\![\mathscr{V}, \mathscr{D}]\!]]\!]] \cong [\mathscr{T}, [\![\mathscr{U} \otimes \mathscr{V}, \mathscr{D}]\!]]$
4. $\mathscr{H} \equiv [\![\mathscr{T}, [\mathscr{U}, [\mathscr{V}, \mathscr{D}]]]\!] \cong [\![\mathscr{T}, [\mathscr{V} \otimes \mathscr{U}, \mathscr{D}]]\!]$
5. $\mathscr{H} \equiv [\![\mathscr{T}, [\mathscr{U}, [\![\mathscr{V}, \mathscr{D}]\!]]]\!] \cong [\![\mathscr{T}, [\![\mathscr{V}, [\mathscr{U}, \mathscr{D}]]\!]]\!] \cong [\![\mathscr{T} \otimes \mathscr{V}, [\mathscr{U}, \mathscr{D}]]\!]$
6. $\mathscr{H} \equiv [\![\mathscr{T}, [\![\mathscr{U}, \mathbf{Fun}_x(\mathscr{V}, \mathscr{D})]\!]]\!] \cong [\![\mathscr{T} \otimes \mathscr{U}, \mathbf{Fun}_x(\mathscr{V}, \mathscr{D})]\!]$

Clearly, the nesting depth in each step thought of as a rewriting rule from left to right strictly decreases, so the induction hypothesis can be applied. □

We now turn to the question of construction of higher-dimensional distributive laws. Luckily, and somewhat interestingly, the complexity of distributive laws doesn't grow with iteration, as formulated below. The most interesting part of the following theorem, the characterisation (4) can be de facto found already in [Gra74] for arbitrary 2-functors.

Theorem 4.3.7 (Higher-dimensional Characterisation)**.** *For theories* $(\mathscr{T}_i, a_i),\ 1 \leq i \leq n$*, the following are equivalent:*

1. *2-functors (4.4)*
2. *for any* $k,\ 1 \leq k \leq n$*, 2-functors*

$$\lambda^{\rightarrow} \ : \ \mathscr{T}_1 \otimes \cdots \mathscr{T}_k \longrightarrow [\![\mathscr{T}_{k+1} \otimes \cdots \otimes \mathscr{T}_n, \mathscr{K}]\!]$$
$$\lambda^{\downarrow} \ : \ \mathscr{T}_{k+1} \otimes \cdots \otimes \mathscr{T}_n \longrightarrow [\mathscr{T}_1 \otimes \cdots \mathscr{T}_k, \mathscr{K}]$$

3. *for some permutation* $s \in \mathbf{S}_n$*, an iterated distributive law of* $\mathscr{T}_{s(n)}$ *over* $\mathscr{T}_{s(n-1)}$*, etc. in* $\mathscr{K}$*. Formally:*

$$\lambda \ \in \ \mathbf{Fun}_{x_1}(\mathscr{T}_{s(1)}, \ldots, (\mathbf{Fun}_{x_n}(\mathscr{T}_{s(n)}, \mathscr{K}))), \quad x_i \in \{\mathbf{f}, \mathbf{r}\}$$

4. *a family of 2-dimensional distributive laws*

$$\mathbf{H}(A_1, \ldots, A_{i-1}, -, A_{i+1}, \ldots, A_{j-1}, -, A_{j+1}, \ldots, A_n) : \mathscr{T}_i \otimes \mathscr{T}_j \longrightarrow \mathscr{K}$$

   for all $i < j$ *and all choices of objects* $A_i \in \mathscr{T}_i$*. This family must agree on objects in the obvious way, and for all triples of indices* $i < j < k$ *and morphisms* $f_i : A_i \longrightarrow A_i'$*,* $f_j : A_j \longrightarrow A_j'$*,* $f_k : A_k \longrightarrow A_k'$ *the diagram in Fig. 4.5 must commute, where the variables constant throughout the diagram are omitted.*

Proof. Equivalence of (1) and (2) is trivial by associativity of $\otimes$ and the characterisation Theorem 4.1.3. Equivalence of (1) and (3) is the Normal Form Theorem, 4.3.6.

The interesting part is equivalence of (1) and (4), which has been shown by Gray for arbitrary 2-functors. We won't repeat the details here, they can by found in Chapter 4 of [Gra74], namely see Definition I,4.6. of quasi-functors of n-variables[2] the remark following it and Theorem I,4.7. □

Example 4.3.8 (Tesseract, continues)**.** By Theorem 4.3.7 a tesseract (Ex. 4.3.2) is defined by sections $\mathbf{T}(-,-,\mathbf{s},\mathbf{s})$, $\mathbf{T}(-,-,\mathbf{s},\mathbf{t})$ $\mathbf{T}(-,-,\mathbf{t},\mathbf{s})$, $\mathbf{T}(-,\mathbf{s},-,\mathbf{s})$, ... The total number of such combinations is $\binom{4}{2} \times 4 \ = \ 24$. This corresponds to the number of rectangular faces in it: six for the inside cube, six for the outside cube and altogether twelve connecting faces. The number of cubes, i.e. functors like $\mathbf{T}(-,-,-,\mathbf{s})$, $\mathbf{T}(-,-,-,\mathbf{t})$, $\mathbf{T}(-,-,\mathbf{s},-), \ldots$, is $\binom{4}{3} \times 2 \ = \ 8$: inside, outside, and the six connecting them. As for the coherence condition, the theorem requires that for any six faces, which constitute a cube geometrically, the corresponding 2-cells commute.

[2] which are similar in the obvious way to the case for two variables

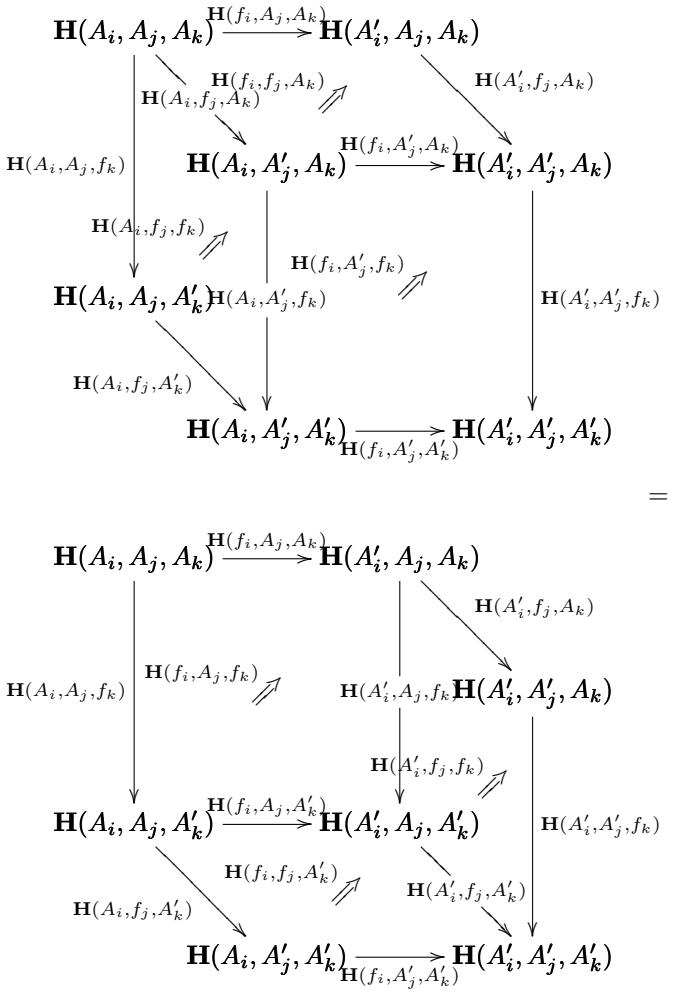


Figure 4.5: Coherence of n-dimensional distributive law

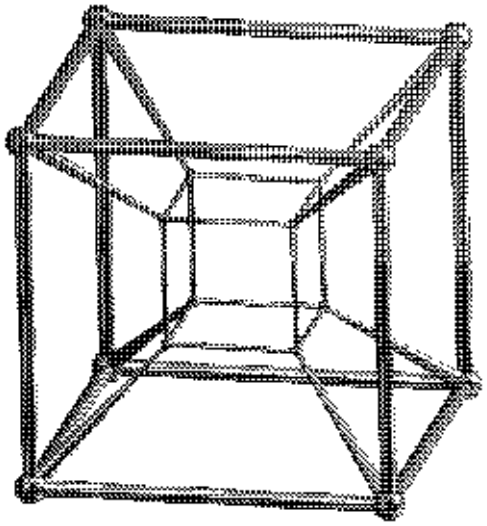


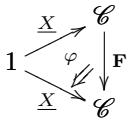
Figure 4.6: Tesseract

Example 4.3.9. An instance of Theorem 4.3.7 for $\mathscr{T}_i = \mathsf{M}$, for all i, is worked out by Cheng in [Che07]. The author shows that a distributive law of n monads can be constructed from a collection of pair-wise distributive laws of two monads such that the condition in Fig. 4.5 holds for the underlying functors of the monads. This is used to combine n monads on a category, and an example why such combined monads are interesting is given. The proof there is carried out explicitly from first principles for monads just with the use of Street's iterative characterisation of distributive laws of monads in 2-categories. The interested reader might therefore find there an interesting exposition for the special case of M.

4.4 Algebras, Coalgebras, Bialgebras

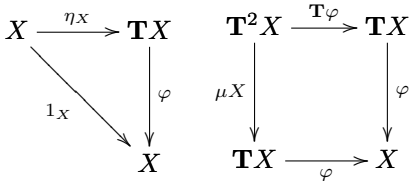
Interestingly, the notion of a generalised distributive law subsumes also the notions of algebras, coalgebras, bialgebras and similar related notions both for endofunctors and monads. In the formulation below it is essential that for any purely equational theory $\mathscr{T}$ and an object $X \in \mathscr{K}$, the functor $\underline{X} : \mathscr{T} \longrightarrow \mathscr{K}$ sending all objects of $\mathscr{T}$ to X and all arrows of $\mathscr{T}$ to 1_X is a model of $\mathscr{T}$ in $\mathscr{K}$. We give a few motivating examples first before introducing a general formulation.

Example 4.4.1 (Algebras for Endofunctors)**.** An algebra of an ordinary endofunctor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{C}$ is an arrow $\varphi : \mathbf{F}X \longrightarrow X$ of $\mathscr{C}$ for some category $\mathscr{C}$ and $X \in \mathscr{C}$. It is easy to check from the definitions that this is the same as a morphism $\varphi : \underline{1} \longrightarrow \ulcorner \mathbf{F} \urcorner$ in $[\mathsf{O}, \mathsf{Cat}]$, where 1 is the terminal one-object category. Recall that $\ulcorner \mathbf{F} \urcorner$ denotes the formal arrow $\mathsf{O} \longrightarrow \mathsf{Cat}$ corresponding to $\mathbf{F}$.



(4.11)

Example 4.4.2 (Algebras for Monads)**.** For a monad $(\mathbf{T}, \mu, \eta) : \mathsf{M} \longrightarrow \mathsf{Cat}$, an algebra of $\mathbf{T}$ is an algebra of the underlying endofunctor $\mathbf{T} : \mathscr{C} \longrightarrow \mathscr{C}$ together with the following coherence conditions, which take place in $\mathscr{C}$:

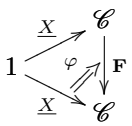


(4.12)

When one observes that (4.12) is equivalent to the diagram in Fig. 4.7 it is immediate that it is just a morphism $\varphi : \underline{1} \longrightarrow (\mathbf{T}, \mu, \eta)$ in $[\mathsf{M}, \mathsf{Cat}]$.

Note that contracting the left-hand-side triangle of identities in Fig. 4.7 to a single point yields a tetrahedron and compare this to Fig. 2.3. Going from Fig. 2.3 to Fig. 4.7 using also the unit of the monad is the construction of a free $\mathbf{T}$-algebra for a given X.

Example 4.4.3 (Coalgebras)**.** Similarly, coalgebras for endofunctors are arrows $\underline{1} \longrightarrow \ulcorner \mathbf{F} \urcorner$ in $[\![\mathsf{O}, \mathsf{Cat}]\!]$. To see this, just reverse the 2-cell in (4.11) to obtain



Similarly, one obtains a colagebra for a comonad from Fig. 4.7 by reversing all 2-cells.

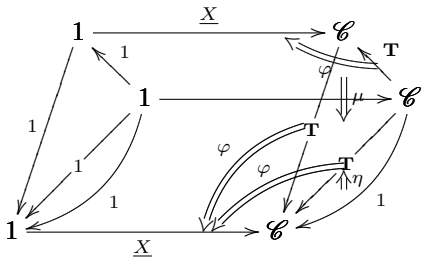


Figure 4.7: Algebra for a monad

Remark 4.4.4. It's easy to check that when 1 is a terminal object of $\mathscr{K}$, $\underline{1}$ is the terminal object both in $[\mathscr{T}, \mathscr{K}]$ and $[\![\mathscr{T}, \mathscr{K}]\!]$. This means that in categorical terms algebras and coalgebras are *points* (elements) of models of theories.

4.4.1 Generalised Algebras and Coalgebras

The previous examples and Remark 4.4.4 suggest the following definition.

Definition 4.4.5 (Generalised algebra)**.** *Let $\mathscr{K}$ be a 2-category with terminal object* 1.

1. *A* generalised algebra *for a functor with structure $\mathbf{T} : \mathscr{T} \longrightarrow \mathscr{K}$ (a* $\mathbf{T}$-algebra*) is an arrow $\phi : \underline{1} \longrightarrow \mathbf{T}$ in $[\mathscr{T}, \mathscr{K}]$.*

2. *A* morphism *of $\mathbf{T}$-algebras ϕ to ψ is a modification $\phi \longrightarrow \psi$.*

3. *A* generalised coalgebra *for a functor with structure $\mathbf{T} : \mathscr{T} \longrightarrow \mathscr{K}$ (a* $\mathbf{T}$-coalgebra*) is an arrow $\phi : \underline{1} \longrightarrow \mathbf{T}$ in $[\![\mathscr{T}, \mathscr{K}]\!]$. The notion of a morphism dualises accordingly.*

We leave for the reader to check that the definition of a modification indeed yields the correct notions of morphisms for Examples 4.4.1, 4.4.2 and 4.4.3. The following example shows that some interesting notions combing algebras and distributive laws in the usual sense arise as generalised algebras in the above sense.

Example 4.4.6. 1. For a distributive law of monads, $\mathbf{G}|\frac{\mathbf{F}}{\frac{\kappa}{\mathbf{F}}}|\mathbf{G}$, a *distributive algebra* [Bec69] is a pair of algebras $\phi : \mathbf{F}X \longrightarrow X$, $\psi : \mathbf{G}X \longrightarrow X$, such that

$$\mathbf{FG}X \xrightarrow{\mathbf{F}\psi} \mathbf{F}X \xrightarrow{\phi} X \;=\; \mathbf{FG}X \xrightarrow{\kappa X} \mathbf{GF}X \xrightarrow{\mathbf{G}\phi} \mathbf{G}X \xrightarrow{\psi} X \tag{4.13}$$

Is is easy to check that this is precisely a morphism $\underline{1} \longrightarrow \kappa$ in $[\mathsf{M} \otimes \mathsf{M}, \mathsf{Cat}]$. Equivalently, it is a morphism $\underline{\underline{1}} \longrightarrow \kappa^{\downarrow}$ in $[\mathsf{M}, [\mathsf{M}, \mathsf{Cat}]]$, a $\kappa^{\downarrow}$-algebra.

2. Dually, a *distributive coalgebra* for a distributive law of comonads κ is given by a pair of coalgebras satisfying the dual of (4.13). By duality we obtain directly from the above example that it is a morphism $\underline{1} \longrightarrow \kappa$ in $[\![\mathsf{M}^{\mathsf{co}} \otimes \mathsf{M}^{\mathsf{co}}, \mathsf{Cat}]\!]$ and therefore a $\kappa^{\rightarrow}$-coalgebra $\underline{1} \longrightarrow \kappa^{\rightarrow}$ in $[\![\mathsf{M}^{\mathsf{co}}, [\![\mathsf{M}^{\mathsf{co}}, \mathsf{Cat}]\!]]\!]$.

A fundamental property of algebras is the existence of initial algebras, which is a question studied well for 2 and M. Dually, one is interested in terminal coalgebras of various endofunctors and comonads. In the remaining text, the following notation for initial algebras and terminal coalgebras is used, whenever the respective objects exist.

Notation 4.4.7. Let $\mathbf{T}$ be a functor with structure with theory $\mathscr{T}$.

1. the (generalised) initial $\mathbf{T}$-algebra, i.e. the initial object in the category of generalised $\mathbf{T}$-algebras, is denoted $\mu_{\mathbf{T}}$
2. the unique morphism out of the initial algebra is denoted $(\!|\, \varphi \,|\!)$ for a $\mathbf{T}$-algebra φ.
3. the (generalised) terminal $\mathbf{T}$-coalgebra is denoted $\nu_{\mathbf{T}}$
4. the unique morphism into the terminal coalgebra is denoted $[\!(\, \psi \,)\!]$ for a $\mathbf{T}$-coalgebra ψ.

Remark 4.4.8. In the algebra of functional programming [BdM96], a rule called *fusion* discussed in more detail in (6.15) is based solely on the initiality (terminality) properties. Hence, it generalises to generalised algebras (coalgebras).

4.4.2 Bialgebras

For the reasons explained in remark 4.3.4, neither is a distributive κ-algebra a κ-algebra (only a $\kappa^{\rightarrow}$-algebra) nor is a distributive κ-coalgebra a κ-coalgebra (only a $\kappa^{\downarrow}$-coalgebra). However, just as there are three non-equivalent ways to form a functor category from two theories, there is a third nonequivalent notion similar to generalised algebras and coalgebras of a distributive law of $\mathscr{T}$ over $\mathscr{U}$. This has been called a *bialgebra* when $\mathscr{T} \equiv \mathsf{M}$ and $\mathscr{U} \equiv \mathsf{M}^{\mathsf{co}}$.

Definition 4.4.9 (Generalised bialgebra)**.** *Let* $\mathbf{U}|\frac{\mathbf{T}}{\kappa}|\mathbf{U}$ *be a generalised distributive law in* $\mathscr{K}$*. A* generalised κ-bialgebra *is an arrow* $1 \longrightarrow \kappa^{\downarrow}$ *in* $[\![\mathscr{U}, [\mathscr{T}, \mathscr{K}]]\!]$ *– a generalised* $\kappa^{\downarrow}$*-coalgebra – and equivalently and arrow* $1 \longrightarrow \kappa^{\rightarrow}$ *in* $[\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]]$ *– a generalised* $\kappa^{\rightarrow}$*-algebra.*

Example 4.4.10. The following is obtained straightforwardly from the general definition.

1. For a distributive law $(\mathsf{O}, \mathbf{G})|\overset{(\mathsf{O},\mathbf{F})}{\underset{(\mathsf{O},\mathbf{F})}{\overline{\kappa}}}|(\mathsf{O}, \mathbf{G})$, so that $\mathbf{F}$ and $\mathbf{G}$ are endofunctors, a κ-bialgebra is an $\mathbf{F}$-algebra ϕ and a $\mathbf{G}$-coalgebra ψ such that the following pentagram commutes:

$$\begin{array}{ccccc} \mathbf{F}X & \xrightarrow{\mathbf{F}\psi} & \mathbf{F}\mathbf{G}X & & \\ & & & \searrow^{\kappa_X} & \\ \downarrow{\scriptstyle \phi} & & & & \mathbf{G}\mathbf{F}X \\ & & & & \downarrow{\scriptstyle \mathbf{G}\phi} \\ X & & \xrightarrow[\psi]{} & & \mathbf{G}X \end{array} \tag{4.14}$$

2. For $\mathscr{T} \equiv \mathsf{M}$ and $\mathscr{U} \equiv \mathsf{M}^{\mathsf{co}}$ with underlying functors $\mathbf{F}$ and $\mathbf{G}$, respectively, κ-bialgebra is the same as above but where everything is understood with the additional structure. So e.g. $\mathbf{G}\phi \cdot \kappa X$ is an $\mathbf{F}$-algebra coherent with M, $\kappa_X \cdot \mathbf{F}\psi$ is a $\mathbf{G}$-coalgebra, etc.

Remark 4.4.11. Although the mentioned general formulation of (co)algebras parameterised in a theory $\mathscr{T}$ allowed us to conceptually unify some notions related to (co)algebras of endofunctors and (co)monads and distributive laws, the question remains if there exists any other interesting theory for which the notion of an algebra makes sense.

4.5 Examples

Remark 4.5.1. Some of the following examples are presented up to isomorphisms on objects, so we write, e.g. $\mathscr{C}^2 \times \mathscr{D}^2 \xrightarrow{f^2} \mathscr{C}^2$ for $f : \mathscr{C} \times \mathscr{D} \longrightarrow \mathscr{C}$. However, note that we don't *identify* objects up to isomorphisms, we just don't write the isomorphisms in the diagrams.

Example 4.5.2 (Simple distributive laws)**.** A toy example interesting for its simplicity is that of natural transformations of functors with no additional structure. A *simple distributive law* is a distributive law of 2 over 2. By the Characterisation Theorem 4.1.3, a simple distributive law in $\mathscr{K}$ is a 2-cell $\kappa : fg \Longrightarrow g'f'$ in $\mathscr{K}$.

Example 4.5.3 (Distributive laws involving monads and comonads)**.** Distributive laws of monads over monads have already been discussed in detail in Ex. 4.1.5. Formally, they are functors $\mathsf{M} \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$. The other known notions of distributive laws involving monads and comonads are:

$\mathsf{M} \otimes \mathsf{M}^{\mathsf{co}} \longrightarrow \mathsf{Cat}$	monads over comonads
$\mathsf{M}^{\mathsf{co}} \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$	comonads over monads
$\mathsf{M}^{\mathsf{co}} \otimes \mathsf{M}^{\mathsf{co}} \longrightarrow \mathsf{Cat}$	comonads over comonads

Example 4.5.4 (Eilenberg-Moor lifting and Kleisli extension of functors)**.** For a functor $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$, and monads $\mathbf{H}$, $\mathbf{K}$, the functor $\tilde{\mathbf{F}} : \mathscr{C}^{\mathbf{H}} \longrightarrow \mathscr{D}^{\mathbf{K}}$ is called an *Eilenberg-Moore lifting of* $\mathbf{F}$ if the following square commutes, where $\mathscr{C}^{\mathbf{T}}$ is the Eilenberg-Moore category of algebras of a monad $\mathbf{T}$ on $\mathscr{C}$:

$$\begin{array}{ccc} \mathscr{C}^{\mathbf{H}} & \xrightarrow{\tilde{\mathbf{F}}} & \mathscr{D}^{\mathbf{K}} \\ {\scriptstyle U^{\mathbf{H}}}\downarrow & & \downarrow{\scriptstyle U^{\mathbf{K}}} \\ \mathscr{C} & \xrightarrow[\mathbf{F}]{} & \mathscr{D} \end{array} \tag{4.15}$$

Eilenberg-Moore liftings (4.15) are in 1-1 correspondence with distributive laws of monads over a functor [MM07] typed

$$\lambda \ : (2, \mathbf{F}) | \frac{\overset{(\mathsf{M},\mathbf{H})}{\lambda}}{\underset{(\mathsf{M},\mathbf{K})}{}} | (2, \mathbf{F})$$

These are formally functors $\lambda : \mathsf{M} \otimes 2 \longrightarrow \mathsf{Cat}$. Note that because $\mathbf{F}$ is not an endofunctor, there are two monads involved in this distributive law. In other words, Eilenberg-Moore liftings are reverse morphisms of monads $\mathbf{H} \longrightarrow \mathbf{K}$.

Similarly, so-called *Kleisli extension of* $\mathbf{F}$ is a functor $\breve{\mathbf{F}} : \mathscr{C}_{\mathbf{H}} \longrightarrow \mathscr{C}_{\mathbf{K}}$ such that the following commutes:

$$\begin{array}{ccc} \mathscr{C}_{\mathbf{H}} & \xrightarrow{\breve{\mathbf{F}}} & \mathscr{D}_{\mathbf{K}} \\ {\scriptstyle \mathbf{i}_{\mathbf{H}}}\uparrow & & \uparrow{\scriptstyle \mathbf{i}_{\mathbf{K}}} \\ \mathscr{C} & \xrightarrow[\mathbf{F}]{} & \mathscr{D} \end{array} \quad , \tag{4.16}$$

where $\mathscr{C}_{\mathbf{H}}, \mathscr{C}_{\mathbf{K}}$ are the Kleisli categories of monads $\mathbf{H}$, $\mathbf{K}$, and $\mathbf{i}$ are inclusions. Kleisli extensions (4.16) are in 1-1 correspondence with distributive laws of a functor over monads

$$(\mathsf{M}, \mathbf{H}) | \overset{(2,\mathbf{F})}{\underset{(2,\mathbf{F})}{\overset{\lambda}{\overline{\quad}}}} | (\mathsf{M}, \mathbf{K}) \ ,$$

formally, functors $\lambda : 2 \otimes \mathsf{M} \longrightarrow \mathsf{Cat}$. These are forward morphisms of monads $\mathbf{H} \longrightarrow \mathbf{K}$.

Example 4.5.5 (Lax Monoidal Functors)**.** A lax monoidal functor was defined in Ex. 3.5.3 as a morphism in $[\mathsf{Mon}, \mathscr{K}]$. Equivalently, it is a distributive law $\mu : \mathsf{Mon} \otimes 2 \longrightarrow \mathsf{Cat}$

Example 4.5.6 (Strength)**.** Strong functors with different additional structures play an essential role in the mathematics of functional programming. A strong monad [Koc72] $(\mathbf{T} : \mathscr{C} \longrightarrow \mathscr{C}, \mu, \eta)$ is a monad equipped with a natural transformation $\sigma_{X,Y} : \mathbf{T}X \times Y \longrightarrow \mathbf{T}(X \times Y)$ coherent with the multiplication and unit of the monad, associativity α and right unit, π_1, of the product. For the latter two we have in $\mathscr{C}$ the following diagrams of natural transformations:

$$\begin{array}{ccc} \mathbf{F}X \times 1 & \xrightarrow{\sigma_{X,1}} & \mathbf{F}(X \times 1) \\ {\scriptstyle \pi_1}\downarrow & \swarrow_{\mathbf{F}\pi_1} & \\ \mathbf{F}X & & \end{array} \qquad \begin{array}{ccc} \mathbf{F}X \times (Y \times Z) & \xrightarrow{\alpha_{\mathbf{F}X,Y,Z}} & (\mathbf{F}X \times Y) \times Z \\ {\scriptstyle \sigma_{X,Y\times Z}}\downarrow & & \downarrow{\scriptstyle \sigma_{X,Y}\times Z} \\ \mathbf{F}(X \times (Y \times Z)) & & \mathbf{F}(X \times Y) \times Z \\ & {\scriptstyle \mathbf{F}\alpha_{X,Y,Z}}\searrow & \downarrow{\scriptstyle \sigma_{X\times Y,Z}} \\ & & \mathbf{F}((X \times Y) \times Z) \end{array} \tag{4.17}$$

Moggi in [Mog91] demonstrated the relevance of strong monads to the semantics of computation. In particular, strength allows one to perform the computation of functors at the level of arrows, as is common in functional languages.

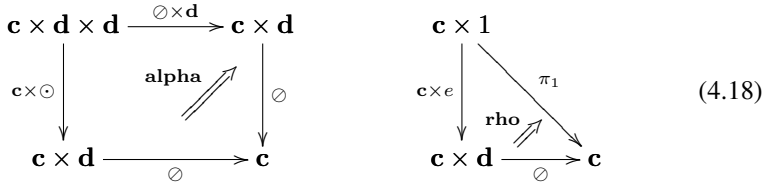
But examples of strong functors with structure other than monads exist. For instance, in [MP08] the authors introduce so-called *applicative functors*, also known as *idioms*, which are, according to one of the equivalent definitions, lax monoidal functors with strength. Applicative functors are not monads in general. See Ex. 4.5.8 bellow. Here we illustrate how the various notions of strong functors with structure all arise

as distributive laws of a *right action* over a functor with structure. For foundations of strength the reader is refered to the analysis by Cockett and Spencer in [CS91].

Let $\mathscr{C}$ and $\mathscr{D}$ be cartesian categories [3]. A right action of $\mathscr{D}$ on $\mathscr{C}$ is a functor $\oslash : \mathscr{C} \times \mathscr{D} \longrightarrow \mathscr{C}$ such that there are natural isomorphisms:

$$\begin{array}{ll} \alpha_{X,Y_1,Y_2} & : X \oslash (Y_1 \odot Y_2) \longrightarrow (X \oslash Y_1) \oslash Y_2 \\ \rho_X & : X \oslash 1 \longrightarrow X \\ \upsilon_{Y,X} & : X \oslash Y \longrightarrow X \oplus (1 \oslash Y) \ , \end{array}$$

where $\oplus$ is the product in $\mathscr{C}$ and $\odot$ is the product in $\mathscr{D}$. These isomorphisms must be coherent with the monoidal structure of $\mathscr{C}$ and $\mathscr{D}$, details of which are irrelevant at this point. A category, $\mathscr{C}$, with a right action by $\mathscr{D}$ is called $\mathscr{D}$*-strong* in [CS91]. All of this can be specified in a finite product sketch, where from the first two lines above we get the following two diagrams:



(4.18)

The theory of this sketch was called RAct in Example 3.5.5, where $\mathbf{c}$ and $\mathbf{d}$ were considered to be just monoidal. However, to obtain the correct notion of strength, which is given by a morphism of $\mathscr{D}$-strong categories for a fixed $\mathscr{D}$ it is not enough to consider an arbitrary morphism between two arbitrary models $\mathbf{F}$, $\mathbf{G}$: RAct $\longrightarrow$ Cat. Not even when both $\mathbf{F}$ and $\mathbf{G}$ send $\mathbf{d}$ to the same $\mathscr{D}$ in Cat. The fact that $\mathscr{D}$ is constant throughout the morphism must be made explicit in the construction.

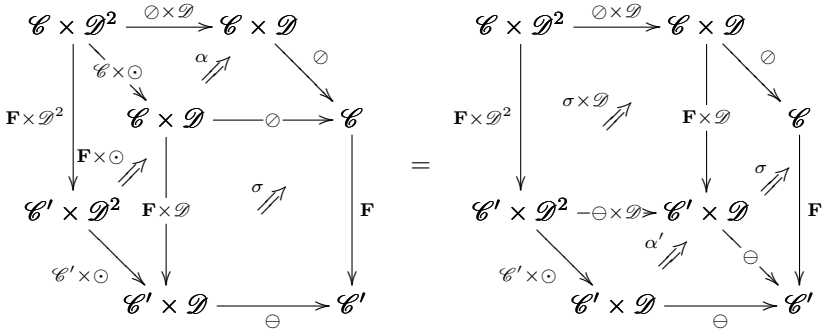
To this end, we must consider a subcategory of those product preserving models of RAct and their morphisms where the image of $\mathbf{d}$ together with the whole monoidal structure on it is fixed to a given monoidal category $\mathbf{D}$: Mon $\longrightarrow$ Cat such that $\mathbf{D}(\mathbf{d}) = \mathscr{D}$. This subcategory is formally given by the following 2-equaliser in 2Cat:

$$[\mathsf{RAct}, \mathsf{Cat}]_{\mathbf{D}} \xrightarrow{\ i\ } [\mathsf{RAct}, \mathsf{Cat}] \underset{\underline{\mathbf{D}}}{\overset{[d,\mathsf{Cat}]}{\rightrightarrows}} [\mathsf{Mon}, \mathsf{Cat}] \quad ,$$

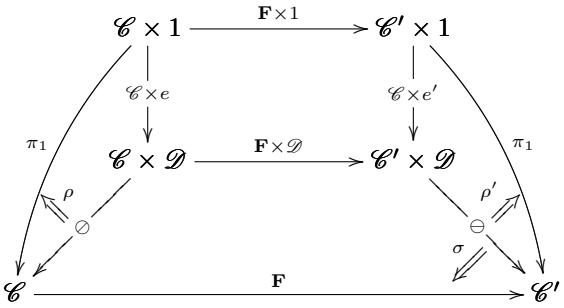
where $[\mathsf{RAct}, \mathsf{Cat}]$ is considered to be the subcategory of *product preserving* 2-functors and d is the inclusion Mon $\longrightarrow$ RAct picking the sub-2-sketch for the monoidal structure of $\mathbf{d}$ within RAct. This construction clearly generalises to an arbitrary $\mathscr{K}$ and an object D in it in place of Cat and $\mathscr{D}$.

[3] Certainly any monoidal structure would do for the analysis below but the monoidal structure of a product gives what is called a *strong functor* in [CS91].

A morphism in $[\mathsf{RAct}, \mathsf{Cat}]_{\mathbf{D}}$, i.e. an object of $[2, [\mathsf{RAct}, \mathsf{Cat}]_{\mathbf{D}}]$ – a distributive law – is expanded below going from top to bottom. Note that it is a particular object in $[2, [\mathsf{RAct}, \mathsf{Cat}]]$ by the inclusion $[2, i]$. The strong functor being defined is here called $\mathbf{F}$.



(4.19)



(4.20)

Now, (4.19) and (4.20) are just diagrams for point free versions of (4.17) for $\mathscr{C}' = \mathscr{D} = \mathscr{C}$ and all products equivalent to the binary product, $\times$, in $\mathscr{C}$.

To obtain a strong *monad* rather than just a strong endofunctor it suffices to consider monads in $[\mathsf{RAct}, \mathsf{Cat}]_{\mathbf{D}}$, formally $\lambda : \mathsf{M} \longrightarrow [\mathsf{RAct}, \mathsf{Cat}]_{\mathbf{D}}$, i.e. particular distributive laws of RAct over M. The underlying arrow of λ is a single arrow $\oslash : \mathscr{C} \times \mathscr{D} \longrightarrow \mathscr{C}$, which is a morphism, σ, of the monad on $\mathscr{C}$.

Remark 4.5.7. In the above example, instead of considering a category of all (product-preserving) models of RAct and then forming its subcategory, we could have defined explicitly the notion of a model and its morphism which fixes the image of a certain sub-theory to a chosen sub-model. This would make explicit what could be called a "2-sketch with constants".

On a general level, this example demonstrates the importance of the notion of additional structure of the theories, which we ignored in our preliminary definition of functor with structure (Def. 3.2.1) but which is of utmost importance. In fact, the whole theory we are presenting in this thesis is a germ of a theory of coherence conditions of such different additional structures.

Note that because of such issues, which we leave for the future work, whenever we

say "x is *a* distributive law of y over z" we mean "some" rather than "any". In other words, most often not all objects and arrows in a particular functor category have the desired properties, but some of them do.

Example 4.5.8 (Applicative functors)**.** In [MP08], McBride and Paterson define the notion of *applicative functors* (a.k.a. idioms) as a notion weaker than monads, which is common in functional programming and useful for organisation of side-effects. In [GdSO09] the authors use applicative functors to formalise so-called *iterator pattern* in object-oriented programming.

Formally, applicative functors are *lax monoidal endofunctors with strength*, where the monoidal action, $\mu_{X,Y} : \mathbf{F}(X) \times \mathbf{F}(Y) \Longrightarrow \mathbf{F}(X \times Y)$ is coherent with the strength $\sigma_{X,Y} : \mathbf{F}X \times Y \Longrightarrow \mathbf{F}(X \times Y)$ by the means of the following coherence equation:

$$\begin{array}{ccc}
\mathbf{F}(X_1) \times \mathbf{F}(X_2) \times Y_1 \times Y_2 & \xrightarrow{\cong} & \mathbf{F}(X_1) \times Y_1 \times \mathbf{F}(X_2) \times Y_2) \\
\downarrow{\scriptstyle \mu \times Y_1 \times Y_2} & & \downarrow{\scriptstyle \sigma \times \sigma} \\
\mathbf{F}(X_1 \times X_2) \times Y_1 \times Y_2 & & \mathbf{F}(X_1 \times Y_1) \times \mathbf{F}(X_2 \times Y_2) \\
\downarrow{\scriptstyle \sigma} & & \downarrow{\scriptstyle \mu} \\
\mathbf{F}(X_1 \times X_2 \times Y_1 \times Y_2) & \xrightarrow[\cong]{} & \mathbf{F}(X_1 \times Y_1 \times X_2 \times Y_2)
\end{array} \tag{4.21}$$

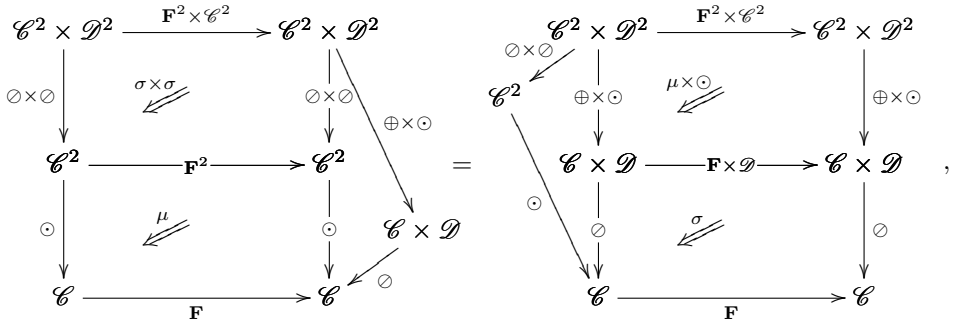
To see how this is a distributive law just notice that (4.21) is the coherence condition in Fig. 4.5. It follows that applicative functors are 3-dimensional distributive laws.

In detail and full generality, let $\mathscr{C}$, $\mathscr{D}$ be monoidal categories with monoidal actions $\oplus$, $\odot$. Let $\oslash : \mathscr{C} \times \mathscr{D} \longrightarrow \mathscr{C}$ be a right action such that it distributes over $\oplus$ and $\odot$:

$$(X_1 \oplus X_2) \oslash (Y_1 \odot Y_2) \;=\; (X_1 \oslash Y_1) \oplus (X_2 \oslash Y_2) \tag{4.22}$$

Further, let $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{C}$ be a monoidal functor with monoidal action $\mu_{X_1,X_2} : \mathbf{F}(X_1) \oplus \mathbf{F}(X_2) \longrightarrow \mathbf{F}(X_1 \oplus X_2)$, i.e. we have a $\lambda_1 : \mathsf{O} \longrightarrow [\mathsf{Mon}, \mathscr{K}]$, and at the same time a strong functor with strength $\sigma_{X,Y} : (\mathbf{F}(X) \oslash Y) \longrightarrow \mathbf{F}(X \oslash Y)$, i.e. a $\lambda_2 : \mathsf{O} \longrightarrow [\mathsf{RAct}, \mathscr{K}]$. For $\mathscr{C} = \mathscr{D}$ and $\odot = \oplus = \oslash = \odot$, (4.22) is trivially satisfied and this is a $\lambda_3 : \mathsf{Mon} \longrightarrow [\mathsf{RAct}, \mathscr{K}]$. In such a case (4.21) is an instance of

the following diagram:



(4.23)

which is easily seen to be a Yang-Baxter equation and therefore, by Theorem 4.3.7, λ_1, λ_2 and λ_3 define a 3-dimensional distributive law

$$\mathbf{H} : \mathsf{O} \otimes \mathsf{Mon} \otimes \mathsf{RAct} \longrightarrow \mathscr{K} \quad ,$$

or equivalently, because (4.22) is an identity, an

$$\mathbf{H}' : \mathsf{O} \otimes \mathsf{RAct} \otimes \mathsf{Mon} \longrightarrow \mathscr{K} \quad , \mathbf{H} \cong \mathbf{H}' \quad .$$

In the further text, we define

$$\mathsf{Af} \ =_{\mathrm{def}} \ \mathsf{O} \otimes \mathsf{RAct} \otimes \mathsf{Mon}$$

Chapter 5

Constructions on Distributive Laws

In this chapter we study constructions on generalised distributive laws, namely their products, coproducts and composition. These are operations frequently considered for the various notions of distributive laws. From the computational perspective, if one equates distributive laws with programs, constructions on distributive laws become constructions on programs. The requirement that the constructions are closed under the additional structure becomes closure w.r.t. specification, in this analogy.

The general approach to the constructions rests on Theorem 4.1.3, which states that a distributive law $\mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ is equivalently a $\mathscr{T}$-functor and a $\mathscr{U}$-functor. Thus one can reduce constructions on distributive laws to constructions on functors with structure. This is indeed what has been done in the specific cases elsewhere [MM07, MP08, GdSO09]. The necessary ingredients for this strategy to succeed are (1.) a generalisation of the intended construction on $\mathscr{T}$- and $\mathscr{U}$-functors to an arbitrary 2-category (2.) establishing that the base category, $[\![\mathscr{U}, \mathscr{K}]\!]$ or $[\mathscr{T}, \mathscr{K}]$, has the required structure to support this construction.

For example, a so-called *cartesian-product monad* on the same category always exists if the category has products. Distributive laws $\lambda, \kappa : \mathscr{U} \otimes \mathsf{M} \longrightarrow \mathscr{K}$ of $\mathscr{U}$-functor over a monad are monads, $\lambda^{\downarrow}$, $\kappa^{\downarrow} : \mathsf{M} \longrightarrow [\mathscr{U}, \mathscr{K}]$, in the functor category $[\mathscr{U}, \mathscr{K}]$. It is easy to generalise the construction of the cartesian-product monad. If the common underlying object of $\lambda^{\downarrow}$ and $\kappa^{\downarrow}$ has products in $[\mathscr{U}, \mathscr{K}]$ then one can construct the product as the pointwise product of λ and κ as the cartesian-product monad of $\lambda^{\downarrow}$ and $\kappa^{\downarrow}$.

A similar argument leads to coproducts and composition of distributive laws, which we also consider and we illustrate by examples that the notions arising in this way are the usual notions considered previously for specific examples of distributive laws.

Moreover, we illustrate how one can make use of the internal iterative nature of certain theories of functors with structure. For instance, applicative functors are themselves 3-dimensional distributive laws $2 \otimes \mathsf{Mon} \otimes \mathsf{RAct} \longrightarrow \mathscr{K}$ (Ex. 4.5.8). This fact can be used for combining applicative functors, e.g. to define their composition. The key result that makes this work is Theorem 4.3.7, which allows one to rearrange the theories in a suitable way.

This chapter illustrates that the iterative nature of distributive laws is a useful tool for reasoning with distributive laws. Its purpose is not to present new results about distributive laws but to describe in a fundamental way these and similar constructions, which could lead to a generalisation of existing or discovery of new constructions on distributive laws.

5.1 Products

In this section a general approach to the construction of products of distributive laws is developed. We construct products of distributive laws as products of functors with structure, providing the functor category has enough structure. In this instance we investigate the situation when the required structure is just that of *products on objects* in the base 2-category. This is a 2-categorical generalisation of the usual requirement for the underlying category of functors with structure to have all products.

5.1.1 Pointwise products of functors with structure

A pair of ordinary functors $f, g : \mathscr{C} \longrightarrow \mathscr{D}$ has a product $f \parallel g : \mathscr{C} \longrightarrow \mathscr{D}$ providing that $\mathscr{D}$ has products, $\times$. In that case, $f \parallel g$ is defined by the assignment on objects and arrows:

$$(f \parallel g)(x) =_{\mathrm{def}} f(x) \times g(x) \tag{5.1}$$

In order to generalise (5.1) from Cat to an arbitrary 2-category, $\mathscr{K}$, we must rewrite it in a point-free fashion as

$$\mathscr{C} \xrightarrow{f \vartriangle g} \mathscr{D}^2 \xrightarrow{(\times)} \mathscr{D} \ ; \tag{5.2}$$

this is well defined in Cat because Cat itself has products, and $\mathscr{D}$ has products, which are given by the adjunction $\Delta \dashv (\times)$.

Notation 5.1.1. Because we are dealing with products on two levels, we adhere to the nonstandard notation that, with respect to a known base 2-category with products, $\mathscr{K}$, $\times$ is used for the products in $\mathscr{K}$, i.e. $f \times g : X \times X' \longrightarrow Y \times Y'$, whereas $f \parallel g$ stands for (5.2) when it makes sense.

For any other bracketed and circular operators denoting arrows in $\mathscr{K}$ of type $X^2 \longrightarrow X$, we interpret them as operators as follows: for an arrow, $\Box : X^2 \longrightarrow X$ and arrows of $\mathscr{K}$, $f, g : Z \longrightarrow X$, the notation $f \Box g$ means formally the arrow:

$$Z \xrightarrow{f \vartriangle g} X^2 \xrightarrow{\Box} X \ .$$

For any pair of 2-functors $\mathbf{F}, \mathbf{G} : \mathscr{T} \longrightarrow \mathscr{K}$ into a 2-category, $\mathscr{K}$, with products, $\times$, there always exists the functor $\mathbf{F} \parallel \mathbf{G} : \mathscr{T} \longrightarrow \mathscr{K}$. It clearly holds that for any 2-cell α in $\mathscr{T}$, $(\mathbf{F} \parallel \mathbf{G})(\alpha) = \mathbf{F}(\alpha) \times \mathbf{G}(\alpha)$.

The assignment $\mathbf{F}, \mathbf{G} \mapsto \mathbf{F} \parallel \mathbf{G}$ extends to a 2-functor on either of $\mathscr{K}^{\mathscr{T}}$, $[\mathscr{T}, \mathscr{K}]$ and $[\![\mathscr{T}, \mathscr{K}]\!]$, defines a product in the respective 2-categories. We record this fact as the following theorem, its proof is straightforward from the definitions.

Theorem 5.1.2. *For a 2-category $\mathscr{K}$, the categories $[\mathscr{T}, \mathscr{K}]$ and $[\![\mathscr{T}, \mathscr{K}]\!]$ have products for all $\mathscr{T}$ if and only if $\mathscr{K}$ has products.*

These products are called *pointwise products*.

Example 5.1.3 (Pointwise product monad)**.** For example, for a pair of monads $\mathbf{T} \equiv (t : \mathscr{C} \longrightarrow \mathscr{C}, \mu, \eta)$, $\mathbf{U} \equiv (u : \mathscr{D} \longrightarrow \mathscr{D}, \nu, \zeta)$, this defines the monad $\mathbf{T} \parallel \mathbf{U} \equiv (t \times u : \mathscr{C} \times \mathscr{D} \longrightarrow \mathscr{C} \times \mathscr{D}, \mu \times \nu, \eta \times \zeta)$ for the corresponding formal monads, $\mathbf{T}$, $\mathbf{U}$, with underlying arrows t, u, respectively.

5.1.2 Products on objects in functor categories

Consider a theory $\mathscr{T}$, a base 2-category $\mathscr{K}$ with products, and the diagram (5.2) in $[\mathscr{T}, \mathscr{K}]$. By Theorem 5.1.2, $[\mathscr{T}, \mathscr{K}]$ has products so the diagram

$$X \xrightarrow{f \vartriangle g} Y^2 \xrightarrow{(\times)} Y \tag{5.3}$$

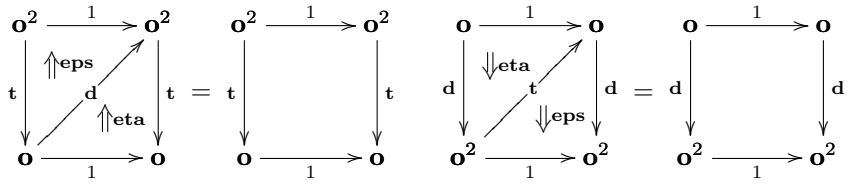
makes sense in it providing there is an arrow $(\times) : Y^2 \longrightarrow Y$ which is a formal right adjoint to the arrow $\Delta_Y \equiv 1_Y \vartriangle 1_Y$, i.e. a product on Y in $[\mathscr{T}, \mathscr{K}]$. This shows that if there is a product on Y in $[\mathscr{T}, \mathscr{K}]$, there is a product for any pair of arrows $f, g : X \longrightarrow Y$ in $[\mathscr{T}, \mathscr{K}]$, i.e. distributive laws $\mathscr{T} \otimes 2 \longrightarrow \mathscr{K}$.

Similar considerations apply for products of functors with structure other then arrows. For instance, the cartesian-product monad is defined from the products in $\mathscr{C}$ and it holds for the underlying arrow that $(\mathbf{T} \times \mathbf{U})(\mathbf{a}) = \mathbf{T}(\mathbf{a}) \parallel \mathbf{U}(\mathbf{a})$.

In both cases, the key notion is that of a product on an object of a 2-category, where the 2-category is a functor category. The rest of this section investigates the construction of such products on objects for arbitrary $\mathscr{T}$ and $\mathscr{K}$ and it is shown that products on objects of $\mathscr{K}$ *always* lift to products on objects of $[\mathscr{T}, \mathscr{K}]$.

We later observe that the situation is not symmetrical for $[\mathscr{T}, \mathscr{K}]$ and $[\![\mathscr{T}, \mathscr{K}]\!]$. We can employ the dualities, though. This is discussed in Sect. 5.2 below.

Remark 5.1.4. Note that formally a product on an object X in $\mathscr{K}$ is a product preserving model, $\mathbf{F} : \mathsf{Prod} \longrightarrow \mathscr{K}$, of the theory, called Prod, of the following sketch



(5.4)

in $\mathscr{K}$ such that $\mathbf{F}(\mathbf{o}) = X$, where $\mathbf{d} \equiv 1 \vartriangle 1$.

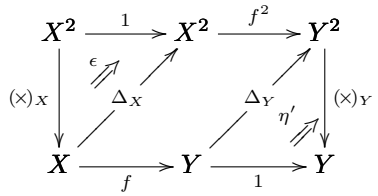
We start by looking at some simple examples.

Arrows, 2

In the following lemma, products on objects of $[2, \mathscr{K}]$, equivalently arrows $f : X \longrightarrow Y$ of $\mathscr{K}$, are constructed.

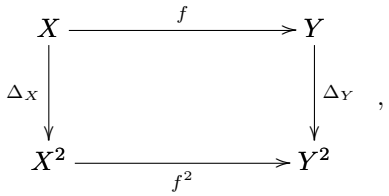
Lemma 5.1.5. *In a 2-category $\mathscr{K}$, let X and Y be objects with products given by adjunctions $(\Delta_X, (\times)_X, \eta, \epsilon)$ and $(\Delta_Y, (\times)_Y, \eta', \epsilon')$, respectively. Then in the category $[2, \mathscr{K}]$ there is a* product *on any $f : X \longrightarrow Y$.*

Proof. First, define an arrow $(\times)_f : f^2 \longrightarrow f$ in $\mathscr{K}$ as follows:



(5.5)

In order to show that this is a right adjoint to the arrow:



, (5.6)

which is denoted $\Delta_f : f \longrightarrow f^2$, one needs 2-cells

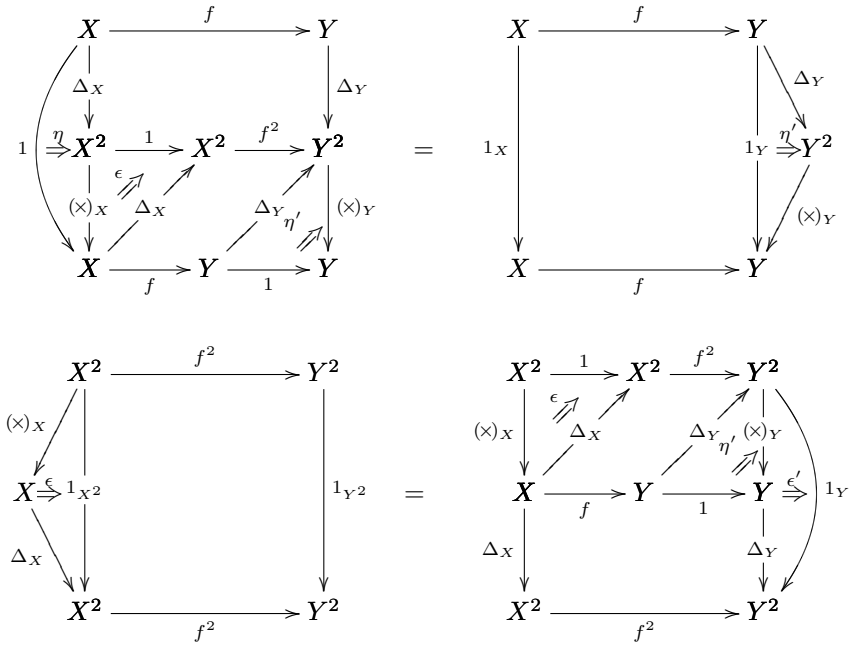
$$\eta_f : 1_f \longrightarrow (\times)_f \Delta_f \qquad \epsilon_f : \Delta_f (\times)_f \longrightarrow 1_{f^2} \tag{5.7}$$

in $[2, \mathscr{K}]$. These are modifications with components $\eta_X : 1_X \longrightarrow (\times)_X \Delta_X$ and $\eta_Y : 1_Y \longrightarrow (\times)_Y \Delta_Y$, and $\epsilon_X : \Delta_X (\times)_X \longrightarrow 1_X$ and $\epsilon_Y : \Delta_Y (\times)_Y \longrightarrow 1_Y$. The

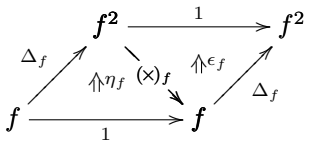
obvious choice is

$$\eta_X := \eta \qquad \eta_Y := \eta' \qquad \epsilon_X := \epsilon \qquad \epsilon_Y := \epsilon' \tag{5.8}$$

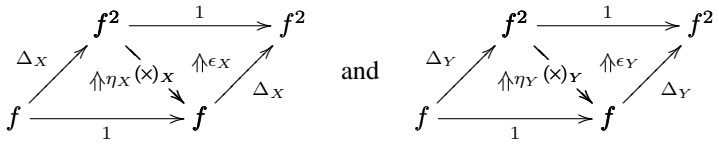
These must, by definition of modification, satisfy coherence conditions rendered in $\mathscr{K}$ as follows:



To see that the equalities hold, note that the 2-cells η and ϵ in the top left corner cancel each other out and the rest are identities; similarly for η' and ϵ' in the bottom right corner in the bottom row. This establishes that the components (5.8) indeed define modifications (5.7), i.e. 2-cells in $[2, \mathscr{K}]$. Note that because 2 has no nontrivial 2-cells, there are no other coherence conditions on the components of the modifications. It remains to establish that these 2-cells satisfy the triangle identities of an adjunction. They are inherited from the triangle identities of the components (5.8) because the components of



are just



□

Example 5.1.6. Apply Lemma 5.1.5 for $\mathscr{K} := \mathsf{Cat}$, $X := \mathscr{C}$ and $Y := \mathscr{D}$, categories with products. For a functor $\mathbf{H} : \mathscr{C} \longrightarrow \mathscr{D}$, the arrow $(\times)_{\mathbf{H}}$ is the natural transformation $(\mathbf{H}\pi_1 \vartriangle \mathbf{H}\pi_2)_{X,Y} : \mathbf{H}(X \times Y) \longrightarrow (\mathbf{H}X) \times (\mathbf{H}Y)$; $\Delta_{\mathbf{H}}$ is just the natural transformation with components $\Delta_{\mathbf{H}X} : \mathbf{H}X \longrightarrow \mathbf{H}X \times \mathbf{H}X$.

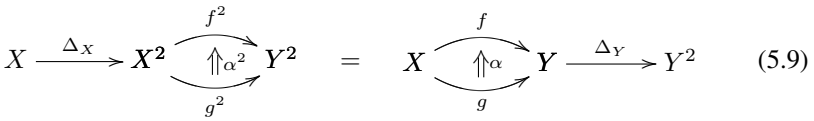
Monads, M

The definition of $(\times)_f$ in (5.5) is valid even in the case when f is not just an arrow, but a monad in $\mathscr{K}$. In other words, $(\times)_f : f^2 \longrightarrow f$ is coherent with the structure of monad on f.

Lemma 5.1.7. *In a 2-category $\mathscr{K}$, let X and Y be objects with products given by adjunctions $(\Delta_X, (\times)_X, \eta, \epsilon)$ and $(\Delta_Y, (\times)_Y, \eta', \epsilon')$, respectively. Then in the category $[\mathsf{M}, \mathscr{K}]$ there is a* product *on any formal monad with underlying arrow $f : X \longrightarrow Y$.*

The following fact is useful in the proof.

Lemma 5.1.8.

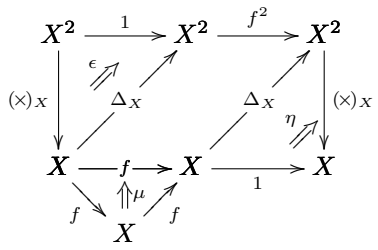


(5.9)

Proof. This is an elementary property of products. □

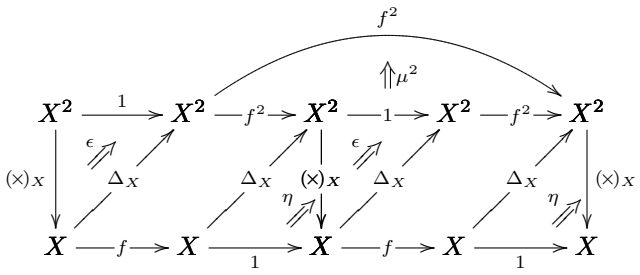
Proof of Lemma 5.1.7. Let $(f : X \longrightarrow X, \mu, \xi)$ be a monad in $\mathscr{K}$; formally, an object in $[\mathsf{M}, \mathscr{K}]$. We keep the definitions (5.5) and (5.6) of $(\times)_f : f^2 \longrightarrow f$ and $\Delta_f : f \longrightarrow f^2$. In addition it must proven that they constitute morphisms of monads, i.e. that they are coherent with μ and ξ. To show the coherence with μ is to establish the following

equality (where $f^2 \equiv f \times f$, not $f \circ f$):



(5.10)

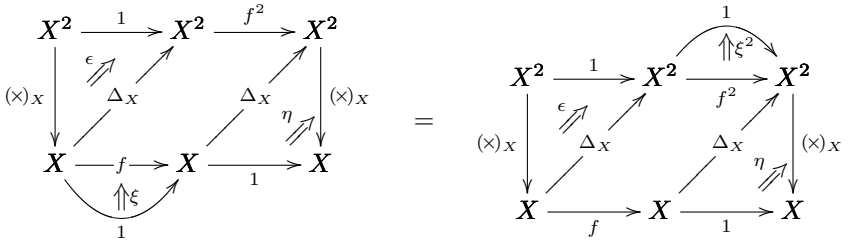
=



(5.11)

But the middle stripe $\epsilon\Delta_X \cdot \Delta_X\eta$ in (5.11) is equal to identity, by the triangle identity of the adjunction. The result is just (5.9), flanked on both sides by the same diagrams.

Establishing coherence with ξ is straightforward; namely



,

follows again from (5.9). □

Generalisation

Note that in the proof of Lemma 5.1.7, nothing specific about monads is used. Everything follows from the triangle identities of the adjunctions defining products on objects. And indeed, as the following theorem shows, such a construction, namely the assignment of the square (5.5) to any arrow f lifts products on objects of $\mathscr{K}$ to products on objects in $[\mathscr{T}, \mathscr{K}]$.

Theorem 5.1.9 (Product Lifting)**.** *Let $\mathscr{K}$ be a category with chosen products, $\times$, let $\mathscr{T}$ be a category, $\mathbf{T} : \mathscr{T} \longrightarrow \mathscr{K}$ a functor. Moreover, let there be for any $X \in \mathscr{T}$ a*

functor $\mathbf{P}_X : \mathsf{Prod} \longrightarrow \mathscr{K}$ *such that*

$$\mathbf{P}_X(\mathbf{o}) \;=\; \mathbf{T}(X) \;. \tag{5.12}$$

Moreover, each $\mathbf{P}_X$ *must preserve the chosen product. Then there exists a product preserving functor*

$$\tilde{\mathbf{T}} : \mathsf{Prod} \longrightarrow [\mathscr{T}, \mathscr{K}]$$

such that $\tilde{\mathbf{T}}(\mathbf{o}) \;=\; \mathbf{T}$*, i.e.* $\tilde{\mathbf{T}}$ *is a product on* $\mathbf{T}$ *in* $[\mathscr{T}, \mathscr{K}]$.

Proof. In the following proof, the notation $(\times)_X, \Delta_X, \eta^X, \epsilon^X$ is used instead of $\mathbf{P}_X(\mathbf{t})$, $\mathbf{P}_X(\mathbf{d}), \mathbf{P}_X(\mathbf{eta}), \mathbf{P}_X(\mathbf{eps})$. Moreover, the index, X, is often omitted when it is clear from the context.

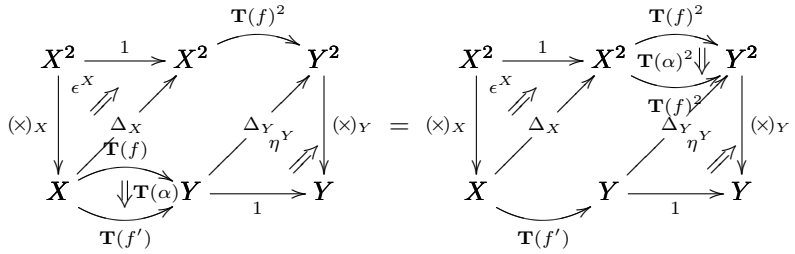
We define a quasi-functor $\mathbf{H} : \mathscr{T} \otimes \mathsf{Prod} \rightsquigarrow \mathscr{K}$, with the required properties, the rest follows by Theorem 2.4.4. To this end, by Def. 2.4.3, one must define the following:

1. for each $X \in \mathscr{T}$ a 2-functor $\mathbf{H}(X, -) : \mathsf{Prod} \longrightarrow \mathscr{K}$. Put $\mathbf{H}(X, -) =_{\mathrm{def}} \mathbf{P}_X$.
2. (a) for the object $\mathbf{o} \in \mathsf{Prod}$, a 2-functor $\mathbf{H}(-, \mathbf{o}) : \mathscr{T} \longrightarrow \mathscr{K}$. Put $\mathbf{H}(-, \mathbf{o}) =_{\mathrm{def}} \mathbf{T}$.
   (b) for the object $\mathbf{o}^2 \in \mathsf{Prod}$, a 2-functor $\mathbf{H}(-, \mathbf{o}^2) : \mathscr{T} \longrightarrow \mathscr{K}$. Put $\mathbf{H}(-, \mathbf{o}) =_{\mathrm{def}} \mathbf{T} \parallel \mathbf{T}$.
3. the equation $\mathbf{H}(X, -)(\mathbf{o}) \;=\; \mathbf{H}(-, \mathbf{o})(X)$ follows from definitions and (5.12). The equation $\mathbf{H}(X, -)(\mathbf{o}^2) \;=\; \mathbf{H}(-, \mathbf{o}^2)(X)$ is established as follows:

$$\begin{aligned}
\mathbf{H}(X, -)(\mathbf{o}^2) &= \mathbf{P}_X(\mathbf{o}^2) && \textit{definition} \\
&= \mathbf{P}_X(\mathbf{o})^2 && \textit{preservation of products} \\
&= \mathbf{T}(X)^2 && (5.12) \\
&= (\mathbf{T} \parallel \mathbf{T})(X) && \textit{definition} \\
&= \mathbf{H}(-, \mathbf{o}^2)(X) && \textit{definition}
\end{aligned}$$

4. For $f : X \longrightarrow X'$ in $\mathscr{T}$ and $\mathbf{t} : \mathbf{o}^2 \longrightarrow \mathbf{o}$ in Prod, we define 2-cells: $\mathbf{H}(f, \mathbf{t})$ essentially as in (5.5) (up to an appropriate renaming); define the 2-cell $\mathbf{H}(f, \mathbf{d})$ essentially as in (5.6). $\mathbf{H}(f, 1)$ is just the identity. All other arrows, g, in Prod are generated freely from $\mathbf{t}$ and $\mathbf{d}$, we extend the definition of $\mathbf{H}(f, g)$ accordingly, i.e. we set $\mathbf{H}(f, kl) =_{\mathrm{def}} \mathbf{H}(f, k) \boxminus \mathbf{H}(f, l)$.
5. Next, it must be shown that for the above definitions (2.40), or equivalently (2.39) and (2.38) commute, for all 2-cells in $\mathscr{T}$ and Prod. The commutativity for 2-cells in Prod, $\mathbf{eta}$, $\mathbf{eps}$, is shown in 5.1.5. This establishes (2.39). Now consider

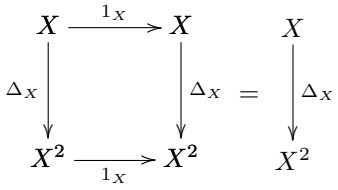
a 2-cell $\alpha : f \Longrightarrow f' : X \longrightarrow Y$ in $\mathscr{T}$, to establish (2.38) for $g \equiv \mathbf{t}$ is to show



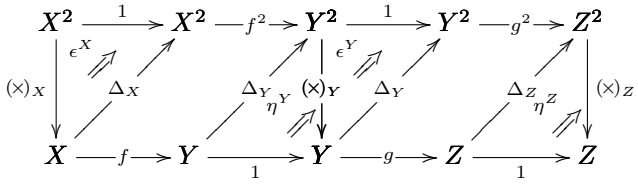
(5.13)

This is just Lemma 5.1.8. The equation (2.38) for $g \equiv \mathbf{t}$ is literally Lemma 5.1.8. The rest follows freely.

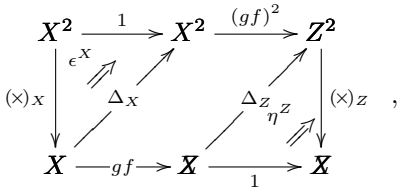
6. That $\mathbf{H}$ respects identities, i.e. $\mathbf{H}(1_X, \mathbf{t}) = 1_{(\times)_X}$ is a triangle identity of the adjunction $\Delta_X \dashv (\times)_X$; $\mathbf{H}(1_X, \mathbf{d}) = 1_{\Delta_X}$ is trivial:



7. That $\mathbf{H}$ respects horizontal composition is to show:



=



which follows by $\Delta_Y \dashv (\times)_Y$.

8. That $\mathbf{H}$ respects vertical composition is by definition, as it is defined freely on the components $\mathbf{t}$, $\mathbf{d}$.

□

This is an important theorem giving a way of lifting products on objects of $\mathscr{K}$ to products on objects of $[\mathscr{T}, \mathscr{K}]$, and consequently to products of distributive laws. Note that often $\mathscr{K}$ is just Cat, and all objects in the range of a functor with structure, $\mathbf{T}$, are the same category with products, say Set. It is therefore automatic that all objects in the image of $\mathscr{T}$ have products on them.

Note that $\tilde{\mathbf{T}}$ is product preserving if $\mathbf{T}$ is, providing that the products on objects in $\mathscr{K}$ are defined pointwise as follows:

Observation 5.1.10. Products $\Delta_X \dashv (\times)_X$ and $\Delta_Y \dashv (\times)_Y$ on X, Y in $\mathscr{K}$ with products give rise to a product $\Delta_{X\times Y} \dashv (\times)_{X\times Y}$ on $X \times Y$ defined as

$$(\times)_{X\times Y} =_{\text{def}} (X \times Y)^2 \xrightarrow{\alpha} X^2 \times Y^2 \xrightarrow{(\times)_X \times (\times)_Y} X \times Y$$
$$\Delta_{X\times Y} =_{\text{def}} X \times Y \xrightarrow{\Delta_X \times \Delta_Y} X^2 \times Y^2 \xrightarrow{\alpha^{-1}} (X \times Y)^2 \ ,$$

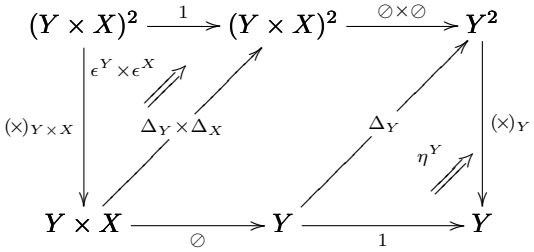
where α is the obvious isomorphism. The unit and counit are just $\eta_{X\times Y} =_{\text{def}} \eta_X \times \eta_Y$, $\epsilon_{X\times Y} =_{\text{def}} \epsilon_X \times \epsilon_Y$.

In the following text we take the liberty in the diagrams of identifying objects up to isomorphism.

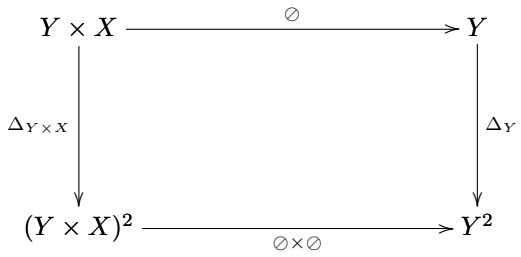
As an example, of the application of Theorem 5.1.9, we expand a part of the definition of a product on a right action.

Example 5.1.11 (Right actions, RAct)**.** Any $\mathbf{S} \in [\mathsf{RAct}, \mathscr{K}]$ has by Theorem 5.1.9 products in $[\mathsf{RAct}, \mathscr{K}]$ defined as follows.

Let $X, Y \in \mathscr{K}$ be defined by $Y \times X \equiv \mathbf{S}(\mathbf{s})$. By definition of RAct, X and Y necessarily have products in $\mathscr{K}$, which are given by adjunctions $(\Delta_X, (\times)_X, \eta^X, \epsilon^X)$ and $(\Delta_Y, (\times)_Y, \eta^Y, \epsilon^Y)$. The product on $\mathbf{S}$ is defined as follows where the isomorphism $(Y \times X)^2 \cong Y^2 \times X^2$ is not pictured. The following two squares define the underlying arrows of $(\times)_{X\,\mathbf{S}}$ and $\Delta_{\mathbf{S}}$, respectively.

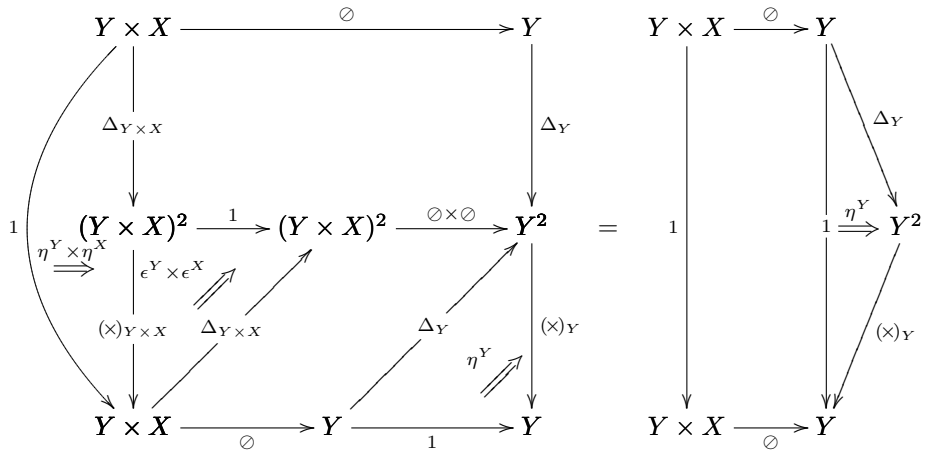


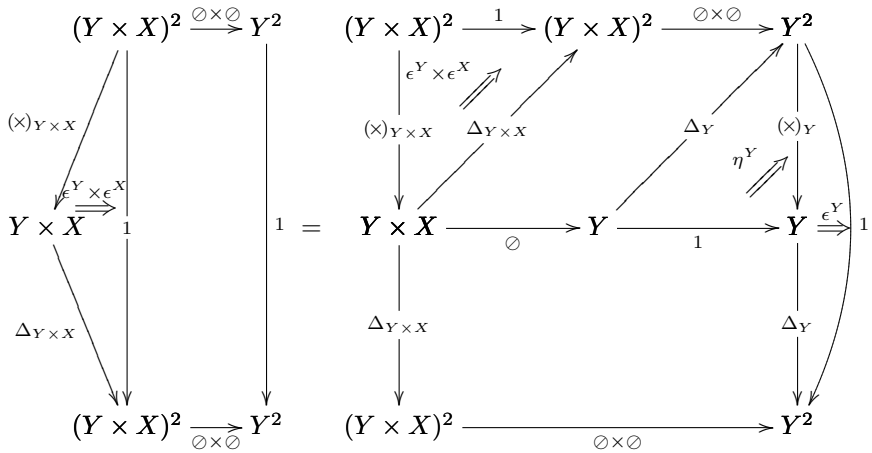
(5.14)



(5.15)

The modifications η and ϵ with components η^X, η^Y, ϵ^X and ϵ^Y are defined next. The coherence conditions for these modifications follow from the triangle identities of the adjunctions on X and Y as follows:





So do the triangle identities. This defines only the morphisms on the underlying arrow, $\oslash$, what remains is the coherence with associativity, α, and right unit rid. All of this follows by the lifting theorem.

5.1.3 Products of Distributive Laws

Having defined a way of lifting products on objects in 2-categories $[\mathscr{T}, \mathscr{K}]$, for all $\mathscr{T}$'s, we can move on to products of distributive laws $\mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$ for various $\mathscr{U}$'s. In the following text, known constructions of products of chosen functors with structure are detailed w.r.t. an arbitrary base 2-category $\mathscr{K}$. This leads, together with the results of the previous section, to notions of products of distributive laws.

Notation 5.1.12. A product of distributive laws λ, κ, is denoted $\lambda \boxtimes \kappa$.

Products of Functors

It was indicated in (5.2) how the pointwise construction of products $f \parallel g$ of functors $f, g : \mathscr{C} \longrightarrow \mathscr{D}$ can be generalised to an arbitrary 2-category $\mathscr{K}$. It wasn't shown why this defines a product in the subcategory of arrows from $\mathscr{C}$ to $\mathscr{D}$. This is now proven formally in a great detail. Namely, we show that a product on X defines a product on arrows from Z to X, for any Z. Formally, $\mathscr{K}(Z, X)$ has products for all Z iff X has products in $\mathscr{K}$. The point of the following lengthy exposition is not so much in proving the obvious but in providing a detailed translation of the abstract categorical notion of products in a hom-category to the lowest-level building blocks, 2-cells, which can already be directly interpreted as programs.

To this end, recall from Def. 2.2.18, that an object $X \in \mathscr{K}$ has products if there exists a 2-categorical adjunction $(\Delta_X, (\times)_X, \epsilon, \eta)$. We start by a general lemma, which shows that the action of the covariant homfunctor $\mathscr{K}(Z, -)$ for any 2-category $\mathscr{K}$ and object $Z \in \mathscr{K}$ preserves adjoints.

Lemma 5.1.13. *In a 2-category $\mathscr{K}$, let $f : X \rightleftarrows Y : g$ be an adjunction with unit η and counit ϵ. Then for any $Z \in \mathscr{K}$, $\mathscr{K}(Z, f) : \mathscr{K}(Z, X) \rightleftarrows \mathscr{K}(Z, Y) : \mathscr{K}(Z, g)$ is an adjunction of ordinary functors, with unit $\mathscr{K}(Z, \eta)$ and counit $\mathscr{K}(Z, \epsilon)$.*

Proof. Fix a Z. The action of $\mathscr{K}(Z, f) : \mathscr{K}(Z, X) \longrightarrow \mathscr{K}(Z, Y)$ on arrows and 2-cells (i.e. the objects and arrows of the *ordinary* category $\mathscr{K}(Z, X)$) is:

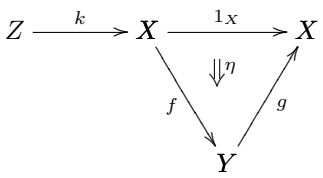
$$Z \underset{l}{\overset{k}{\underset{\Downarrow\alpha}{\rightarrow}}} X \;\mapsto\; Z \underset{l}{\overset{k}{\underset{\Downarrow\alpha}{\rightarrow}}} X \xrightarrow{f} Y$$

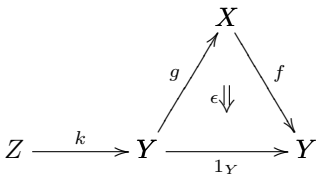
Similarly for $\mathscr{K}(Z, g) : \mathscr{K}(Z, Y) \longrightarrow \mathscr{K}(Z, X)$:

$$Z \underset{l}{\overset{k}{\underset{\Downarrow\alpha}{\rightarrow}}} Y \;\mapsto\; Z \underset{l}{\overset{k}{\underset{\Downarrow\alpha}{\rightarrow}}} Y \xrightarrow{g} X$$

The components $\mathscr{K}(Z, \eta)_k$, $\mathscr{K}(Z, \epsilon)_k$, of the ordinary natural transformations $\mathscr{K}(Z, \eta) : \mathscr{K}(Z, X) \Longrightarrow \mathscr{K}(Z, gf)$ and $\mathscr{K}(Z, \epsilon) : \mathscr{K}(Z, fg) \Longrightarrow \mathscr{K}(Z, Y)$ are the following

2-cells of $\mathscr{K}$:

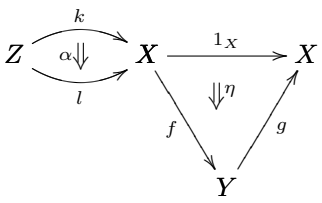


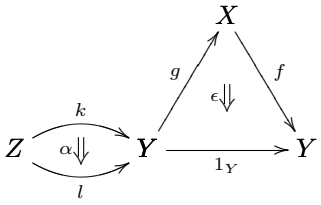


Naturality of $\mathscr{K}(Z,\eta)$ and $\mathscr{K}(Z,\epsilon)$ amounts to showing for any $Z \overset{k}{\underset{l}{\Downarrow\alpha}} X$:

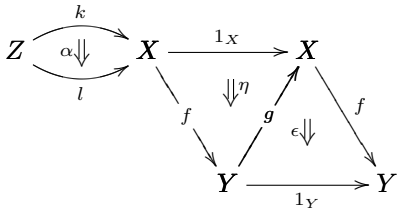
$$\begin{aligned}(gf \circ \alpha) \cdot (\eta \circ k) &= (\eta \circ l) \cdot \alpha \\ (\epsilon \circ l) \cdot (fg \circ \alpha) &= \alpha \cdot (\epsilon \circ k) \ ,\end{aligned}$$

which is proven by the following diagram:



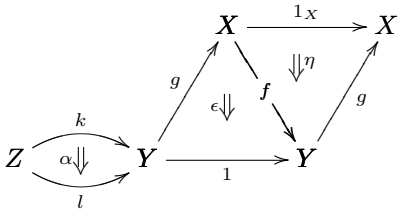


Now the proof of the triangle identities amounts to establishing the following two identities:



$= \quad f \circ \alpha$

,



$= \quad g \circ \alpha$

which follow readily from the triangle identities for η and ϵ. □

The desired theorem about products now follows from the isomorphism $\mathscr{K}(Z, X^2) \cong \mathscr{K}(Z,X)^2$ implied by the product on X.

Lemma 5.1.14. *Let X be an object with products in $\mathscr{K}$ as in Def. 2.2.18. Then for any $Z \in \mathscr{K}$, $\mathscr{K}(Z,X)$ has products.*

Proof. Fix $Z \in \mathscr{K}$. The existence of the product X^2 in $\mathscr{K}$ implies an isomorphism $\Psi : \mathscr{K}(Z,X)^2 \to \mathscr{K}(Z,X^2)$. The adjunction $\Delta : \mathscr{K}(Z,X) \rightleftarrows \mathscr{K}(Z,X)^2 : (\times)$ defining the product in $\mathscr{K}(Z,X)$ is up to Ψ just the adjunction $\mathscr{K}(Z,\Delta_X) \dashv \mathscr{K}(Z,(\times)_X)$, as shown below:

$$\mathscr{K}(Z,X) \xrightarrow{\mathscr{K}(Z,\Delta_X)} \mathscr{K}(Z,X)^2 \;\cong\; \mathscr{K}(Z,X^2) \xrightarrow{\mathscr{K}(Z,(\times)_X)} \mathscr{K}(Z,X) \quad (5.16)$$

Explicitly:

$$\begin{aligned} \Delta &= \Psi \circ \mathscr{K}(Z,\Delta_X) \\ (\times) &= \mathscr{K}(Z,(\times)_X) \circ \Psi^{-1} \end{aligned} \quad (5.17)$$

□

The following observation makes explicit the sense in which such products are defined essentially pointwise.

Observation 5.1.15. The action of $(\times)$ in (5.17) on pairs of arrows $f, g : Z \longrightarrow X$ after unfolding all definitions is:

$$Z \xrightarrow{f \vartriangle g} X^2 \xrightarrow{(\times)_X} X \quad (5.18)$$

So we are ready to construct products of formal arrows in $\mathscr{K}$ between two objects Z, X, and consequently of distributive laws $\lambda, \kappa : \mathscr{T} \otimes 2 \longrightarrow \mathscr{K}$, equivalently 2-functors $\lambda^{\downarrow}, \kappa^{\downarrow} : 2 \longrightarrow [\mathscr{T},\mathscr{K}]$. We want to consider the category of formal arrows λ such that $\lambda^{\downarrow}(\mathbf{s}) = \mathbf{Z}$ and $\lambda^{\downarrow}(\mathbf{t}) = \mathbf{X}$ for $\mathbf{Z}$, $\mathbf{X} \in [\mathscr{T},\mathscr{K}]$. The arrows are those reverse lax natural transformations $\theta : f \Longrightarrow g : 2 \longrightarrow [\mathscr{T},\mathscr{K}]$ such that $\theta_{\mathbf{s}} = 1_{\mathbf{X}}$, $\theta_{\mathbf{t}} = 1_{\mathbf{Z}}$ and $\theta_{\mathbf{a}}$ is a 2-cell of $\mathscr{K}$ of type $1_{\mathbf{Z}} \circ f \Longrightarrow g \circ 1_{\mathbf{X}}$. The 2-cells in this 2-category are modifications but their components are forced by the coherence conditions of the definition to be identities. So this is equivalent to the ordinary homcategory $[\mathscr{T},\mathscr{K}](\mathbf{Z},\mathbf{X})$. By Lemma 5.1.14 this category has products if $\mathbf{X}$ has products in $[\mathscr{T},\mathscr{K}]$. By the Product Lifting theorem, this is whenever $\mathscr{K}$ has products on the relevant objects. The following proposition summarises this result.

Theorem 5.1.16. *Let λ, κ be distributive laws $(2,\mathbf{F})|\overset{(\mathscr{T},\mathbf{H})}{\underset{(\mathscr{T},\mathbf{H}')}{\overline{\quad\lambda\quad}}}|(2,\mathbf{F}')$, $(2,\mathbf{G})|\overset{(\mathscr{T},\mathbf{H})}{\underset{(\mathscr{T},\mathbf{H}')}{\overline{\quad\kappa\quad}}}|(2,\mathbf{G}')$, such that the objects $\mathbf{H}'(\mathbf{s})$ and $\mathbf{H}'(\mathbf{t})$ have products in $\mathscr{K}$. Then there is a distributive law $\mathbf{F} \times \mathbf{G}|\overset{\mathbf{H}}{\underset{\mathbf{H}'}{\overline{\quad\lambda\boxtimes\kappa\quad}}}|\mathbf{F}' \times \mathbf{G}'$, which is the product in the subcategory of vertical distributive laws $\mathbf{H} \longrightarrow \mathbf{H}'$.*

Example 5.1.17. Distributive laws of endofunctors, $\lambda,\ \kappa : \mathsf{O} \otimes \mathsf{O} \longrightarrow \mathsf{Cat}$, such that $\mathbf{F}|\frac{\mathbf{H}}{\lambda}{\mathbf{H}}|\mathbf{F}$ and $\mathbf{G}|\frac{\mathbf{H}}{\kappa}{\mathbf{H}}|\mathbf{G}$ are just natural transformations $\lambda : \mathbf{HF} \Longrightarrow \mathbf{FH}$ and $\kappa : \mathbf{HG} \Longrightarrow \mathbf{GH}$. Their product $\mathbf{F} \times \mathbf{G}|\frac{\mathbf{H}}{\lambda\boxtimes\kappa}{\mathbf{H}}|\mathbf{F} \times \mathbf{G}$ given by Theorem 5.1.16 is a natural transformation $\lambda \boxtimes \kappa : \mathbf{H}(\mathbf{F} \times \mathbf{G}) \Longrightarrow (\mathbf{F} \times \mathbf{G})\mathbf{H}$ defined as follows:

$$\begin{array}{ccccc}
\mathbf{HF} & \xleftarrow{\mathbf{H}\pi_1} & \mathbf{H}(\mathbf{F}\times\mathbf{G}) & \xrightarrow{\mathbf{H}\pi_2} & \mathbf{HG} \\
\downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle \lambda\boxtimes\kappa} & & \downarrow{\scriptstyle \kappa} \\
\mathbf{FH} & \xleftarrow[\pi_1\mathbf{H}]{} & (\mathbf{F}\times\mathbf{G})\mathbf{H} & \xrightarrow[\pi_2\mathbf{H}]{} & \mathbf{GH}
\end{array} \tag{5.19}$$

$$\lambda \boxtimes \kappa =_{\mathrm{def}} (\lambda \cdot \mathbf{H}\pi_1) \vartriangle (\kappa \cdot \mathbf{H}\pi_2)$$

By Observation 5.1.15, it can be also expanded as

$$\mathbf{H} \xrightarrow{\Delta} \mathbf{H}^2 \xrightarrow{\lambda\times\kappa} \mathbf{H}^2 \xrightarrow{(\times)_{\mathbf{H}}} \mathbf{H}$$

in $[\mathsf{O}, \mathscr{K}]$. Unfolding the definitions of morphisms in this category yields the following diagram in Cat:

$$\begin{array}{ccccccc}
C & \xrightarrow{\Delta_C} & C^2 & \xrightarrow{\mathbf{F}\times\mathbf{G}} & C^2 & \xrightarrow{(\times)_C} & C \\
\downarrow{\scriptstyle \mathbf{H}} & & \downarrow{\scriptstyle \mathbf{H}^2} & \swarrow{\scriptstyle \lambda\times\kappa} & \downarrow{\scriptstyle \mathbf{H}^2} & \swarrow{\scriptstyle (\times)_{\mathbf{H}}} & \downarrow{\scriptstyle \mathbf{H}} \\
C & \xrightarrow[\Delta_C]{} & C^2 & \xrightarrow[\mathbf{F}\times\mathbf{G}]{} & C^2 & \xrightarrow[(\times)_C]{} & C
\end{array}$$

This can readily be seen to be the same as (5.19).

Similarly, one obtains by Theorem 5.1.16 distributive laws of monads, strong functors, etc. over functors. Explicitly, this means that when $\mathbf{H}$ itself has some additional structure, the morphism (5.19) is coherent with it.

Products of Monads

The fact that monads have products is known and standard. Following is a presentation of the product in an arbitrary 2-category $\mathscr{K}$. Note the 2-categorical sleekness of the proofs of the coherence conditions.

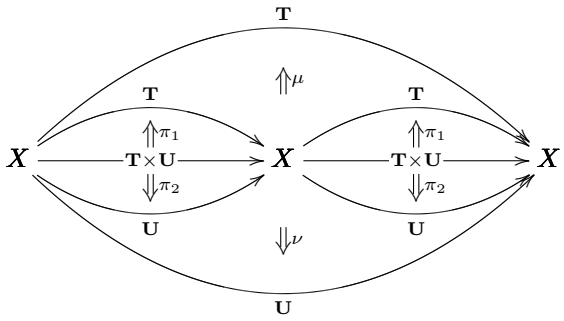
Lemma 5.1.18. *For any two monads* $(\mathbf{T} : X \longrightarrow X, \mu, \eta)$ *and* $(\mathbf{U} : X \longrightarrow X, \nu, \zeta)$ *in a 2-category* $\mathscr{K}$ *with products on* X *the following comprises a monad*

$$(\mathbf{T} \times \mathbf{U} : X \longrightarrow X, \mu \cdot \pi_1\pi_1 \vartriangle \nu \cdot \pi_2\pi_2, \eta \vartriangle \zeta) \tag{5.20}$$

This is a cartesian-product monad.

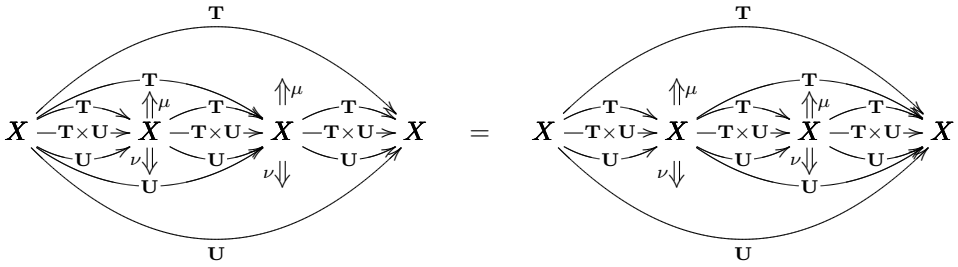
Proof. In order to establish that (5.20) defines a monad, the coherence conditions of multiplication and unit must be established. Below, just associativity of multiplication is shown, units are even simpler. Note that use is made of the isomorphisms of 2-cells implied by the adjunction $\Delta_X \dashv (\times)_X$ in $\mathscr{K}$ (see Sect. 2.2.7 for details).

First, note that when the definition of multiplication in (5.20) is pictured in the following diagram:



(5.21)

associativity of (5.21) becomes the following equation:



It is clearly seen that the top half of the picture is associativity of μ while the bottom half of the picture is associativity of ν, i.e. the preconditions. □

The following is now a simple corollary.

Theorem 5.1.19. *Let λ, κ be distributive laws $(\mathsf{M}, \mathbf{T})|\overline{\underset{(\mathscr{T},\mathbf{H}')}{\overset{(\mathscr{T},\mathbf{H})}{\lambda}}}|(\mathsf{M}, \mathbf{T}')$, $(\mathsf{M}, \mathbf{U})|\overline{\underset{(\mathscr{T},\mathbf{H}')}{\overset{(\mathscr{T},\mathbf{H})}{\kappa}}}|(\mathsf{M}, \mathbf{U}')$ such that $\mathbf{H}'(\mathbf{s})$ and $\mathbf{H}'(\mathbf{t})$ have products in $[\mathscr{T}, \mathscr{K}]$. Then there is a distributive law $\mathbf{T} \times \mathbf{U}|\overline{\underset{\mathbf{H}'}{\overset{\mathbf{H}}{\lambda \boxtimes \kappa}}}|\mathbf{T}' \times \mathbf{U}'$, where $\mathbf{T} \times \mathbf{U}$, $\mathbf{T}' \times \mathbf{U}'$ are cartesian products of monads given by Lemma 5.1.18. This is a product in the category of vertical distributive laws $\mathbf{H} \longrightarrow \mathbf{H}'$.*

Example 5.1.20 (Products of strong monads)**.** Consider strong monads in $\mathscr{K}$, i.e. distributive laws $(\mathsf{M}, \mathbf{T})|\overline{\underset{(\mathsf{RAct},\oslash)}{\overset{(\mathsf{RAct},\oslash)}{\lambda}}}|(\mathsf{M}, \mathbf{T})$, $(\mathsf{M}, \mathbf{U})|\overline{\underset{(\mathsf{RAct},\oslash)}{\overset{(\mathsf{RAct},\oslash)}{\kappa}}}|(\mathsf{M}, \mathbf{U})$ (see Ex. 4.5.6).

It follows that the underlying categories already have products and so by Lemmas 5.1.18, their product $\mathbf{T} \times \mathbf{U}|\frac{\oslash}{\lambda\boxtimes\kappa}|\mathbf{T} \times \mathbf{U}$ exists. This is by definition a strong monad.

The following are examples of constructions of products of functors with structure obtained as products of distributive laws.

Lax monoidal functors

A lax monoidal functor, $\mathbf{F}$, in $\mathscr{K}$ is formally $\mathbf{F} : 2 \longrightarrow [\mathsf{Mon}, \mathscr{K}]$ (see Ex. 3.5.3) and objects of $[\mathsf{Mon}, \mathscr{K}]$ have products whenever $\mathbf{F}(\mathbf{t})(\mathbf{o})$ does in $\mathscr{K}$, by lifting. One can therefore construct products of lax monoidal functors as products of arrows in $[\mathsf{Mon}, \mathscr{K}]$ whenever $\mathbf{F}(\mathbf{t})(\mathbf{o})$ has products in $\mathscr{K}$.

Corollary 5.1.21. *For all lax monoidal functors,* $\mathbf{F}, \mathbf{G} : 2 \longrightarrow [\mathsf{Mon}, \mathscr{K}]$*, on the same monoidal category with products* $(\times)$ *in* $\mathscr{K}$*, the product* $\mathbf{F} \parallel \mathbf{G} : 2 \longrightarrow [\mathsf{Mon}, \mathscr{K}]$ *exists, such that* $(\mathbf{F} \parallel \mathbf{G})(\mathbf{a}) = \mathbf{F}(\mathbf{a}) \times \mathbf{G}(\mathbf{a})$.

Applicative functors

Applicative functors (Ex. 4.5.8) are lax monoidal endofunctors with strength. So the situation is the same as for lax monoidal functors with the difference that applicative functors are defined on monoidal categories where the monoidal action is the product. It follows that applicative functors on the same category always have products.

Remark 5.1.22. We invite the reader to expand the definitions of projections of products of monads and applicative functors in $[2, \mathscr{K}]$ and compare them to Figs 1.1 and 1.2.

5.2 Coproducts

Section 5.1 dualises by the following two facts (see Sect. 2.2.6):

1. products in $\mathscr{K}$ are coproducts in $\mathscr{K}^{\mathsf{op}}$
2. $\mathbf{F} \dashv \mathbf{G}$ in $\mathscr{K}$ if and only if $\mathbf{G} \dashv \mathbf{F}$ in $\mathscr{K}^{\mathsf{co}}$

We summarise the results for a future reference.

5.2.1 Pointwise coproducts

It follows that categories $[\![\mathscr{T}, \mathscr{K}]\!]$ and $[\mathscr{T}, \mathscr{K}]$ have coproducts for all $\mathscr{K}$ with coproducts. Coproducts in $[\![\mathscr{T}, \mathscr{K}]\!]$, as well as in $[\mathscr{T}, \mathscr{K}]$ are called *pointwise coproducts*.

Example 5.2.1.

1. The coproduct of functors $\mathbf{F} : \mathscr{C} \longrightarrow \mathscr{D}$, $\mathbf{G} : \mathscr{C}' \longrightarrow \mathscr{D}'$, considered as formal arrows in $[\![2, \mathsf{Cat}]\!]$, is the usual coproduct of functors $\mathbf{F}+\mathbf{G} : \mathscr{C}+\mathscr{C}' \longrightarrow \mathscr{D}+\mathscr{D}'$ as arrows in Cat.

2. For two comonads $(\mathbf{D} : \mathscr{C} \longrightarrow \mathscr{C}, \delta, \varepsilon)$, $(\mathbf{E} : \mathscr{D} \longrightarrow \mathscr{D}, \gamma, \xi)$, the coproduct comonad $\mathbf{D} + \mathbf{E}$ is defined as $(\mathbf{D} + \mathbf{E} : \mathscr{C} + \mathscr{D} \longrightarrow \mathscr{C} + \mathscr{D}, \delta + \gamma, \varepsilon + \xi)$. This is precisely the free coproduct of $\mathbf{D}$ and $\mathbf{E}$ as formal monads $\mathbf{D}, \mathbf{E} \in [\![\mathsf{M}^{\mathsf{co}}, \mathsf{Cat}]\!]$.

5.2.2 Coproducts on objects in functor categories

The development of Sect. 5.1.2 dualises by the equivalence $[\mathscr{T}, \mathscr{K}] \cong [\![\mathscr{T}^{\mathsf{co}}, \mathscr{K}^{\mathsf{co}}]\!]^{\mathsf{co}}$. As $\mathscr{K}$ has products if and only if $\mathscr{K}^{\mathsf{co}}$ does, one can lift coproducts on objects of $\mathscr{K}$ to coproducts on objects of $[\![\mathscr{T}, \mathscr{K}]\!]$ for all $\mathscr{T}$. This is summarised below.

Corollary 5.2.2. *Let $\mathscr{K}$ be a category with chosen products, $\times$, let $\mathscr{T}$ be a category, $\mathbf{T} : \mathscr{T} \longrightarrow \mathscr{K}$ a functor. Moreover, let there be for any $X \in \mathscr{T}$ a functor $\mathbf{P}_X : \mathsf{Prod}^{\mathsf{co}} \longrightarrow \mathscr{K}$ such that*

$$\mathbf{P}_X(\mathbf{o}) \;=\; \mathbf{T}(X) \;. \tag{5.22}$$

Moreover, each $\mathbf{P}_X$ must preserve the chosen product. Then there exists a product preserving functor

$$\tilde{\mathbf{T}} : \mathsf{Prod}^{\mathsf{co}} \longrightarrow [\![\mathscr{T}, \mathscr{K}]\!]$$

such that $\tilde{\mathbf{T}}(\mathbf{o}) \;=\; \mathbf{T}$, i.e. $\tilde{\mathbf{T}}$ is a coproduct on $\mathbf{T}$ in $[\![\mathscr{T}, \mathscr{K}]\!]$.

5.2.3 Coproducts of distributive laws

Corollary 5.2.2 allows us to construct coproducts of distributive laws of functors with structure, which have coproducts whenever the underlying 2-category does, such as functors or comonads. As before, these are just coproducts of functors with structure in the respective forward category of functors. This follows simply by duality of propositions 5.1.14, 5.1.18 and 5.1.21. This is summarised bellow.

Notation 5.2.3. A coproduct of distributive laws λ, κ, is denoted $\lambda \boxplus \kappa$.

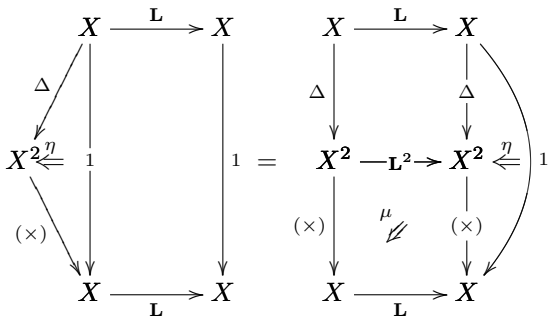
Corollary 5.2.4. *Let $\mathscr{K}$ be a 2-category with coproducts on objects whenever it matters. Let $\mathscr{U}$ be any theory of a functor with structure. Then for $\mathscr{T} \in \{2, \mathsf{M}^{\mathsf{co}}, \mathsf{Mon}\}$, distributive laws $(\mathscr{U}, \mathbf{H})|\frac{(\mathscr{T},\mathbf{F})}{\lambda}|(\mathscr{U}, \mathbf{H}')$, $(\mathscr{U}, \mathbf{H})|\frac{(\mathscr{T},\mathbf{G})}{\kappa}|(\mathscr{U}, \mathbf{H}')$, have coproducts $(\mathscr{U}, \mathbf{H})|\frac{(\mathscr{T},\mathbf{F}+\mathbf{G})}{\lambda\boxplus\kappa}|(\mathscr{U}, \mathbf{H}')$. Here, the fact is used that $2^{\mathsf{co}} \equiv 2$ and*

$\mathsf{Mon}^{\mathsf{co}} \equiv \mathsf{Mon}$.

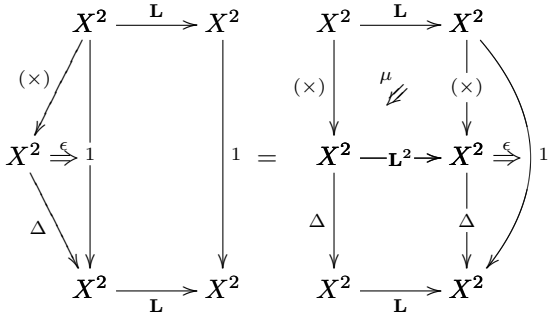
5.2.4 Constructions which are not for free

It is not clear, how one could reverse the proof of Theorem 5.1.9 so as to lift products to objects in $[\![\mathscr{T}, \mathscr{K}]\!]$ and dually coproducts to objects in $[\mathscr{T}, \mathscr{K}]$. However, such constructions for specific theories of functors with structure may, of course, exist.

Example 5.2.5 (Applicative functors)**.** For instance, an applicative functor (Ex. 4.5.8) is a lax monoidal endofunctor on a category with products, which has strength. Strength (Ex. 4.5.6) is a special kind of a morphism of a right action on a category, where all monoidal structures, and the right action are products on the categories. Therefore the strength fixes the monoidal structure of the lax monoidal endofunctor to be a product. The bottom line is that an applicative functor is an endofunctor coherent with the product on the underlying category. Formally this means that the following diagrams commute, for any applicative functor with underlying functor $\mathbf{L}$:



(5.23)



(5.24)

Coherence (5.23) and (5.24) are making η, ϵ into components of modifications, $\eta_{\mathbf{L}}$, $\epsilon_{\mathbf{L}}$. The fact that μ is the underling 2-cell of a forward morphism $\mathbf{L}^2 \longrightarrow \mathbf{L}$, i.e. that μ is coherent with the structure on $\mathbf{L}$, follows by definition and transposition because $\mathbf{L}$ is a reverse morphism $(\times) \longrightarrow (\times)$. As before, these modifications, $\eta_{\mathbf{L}}$, $\epsilon_{\mathbf{L}}$, satisfy the triangle identities pointwise and we therefore have a product on $\mathbf{L}$ in $[\![\mathbf{L}, \mathscr{K}]\!]$, for any applicative functor $\mathbf{L}$.

5.3 Composition

This section continues the theme of the previous sections in that a composition of distributive laws is considered as a composition of functors with structure. It follows that distributive laws can be possibly composed in two ways: horizontally and vertically. First, however, before diving into any further details, it must be made precise what is meant by *composition of functors with structure* and *composition of distributive laws*.

Definition 5.3.1 (Composition of functors with structure)**.** *For a theory of functors with structure, $(a, \mathscr{T})$, and its two models, $\mathbf{F}, \mathbf{G} : \mathscr{T} \longrightarrow \mathscr{K}$ in $\mathscr{K}$, such that $\mathbf{F}(\mathbf{t}) = \mathbf{G}(\mathbf{s})$, $\mathbf{H}$ is* a composition of $\mathbf{G}$ after $\mathbf{F}$*, also denoted $\mathbf{G} \circ \mathbf{F}$, if $\mathbf{H}$ is a model of $\mathscr{T}$ such that*

$$\mathbf{H}(\mathbf{a}) \;=\; \mathbf{G}(\mathbf{a}) \circ \mathbf{F}(\mathbf{a})$$

In words, a composition of models $\mathbf{G}$, $\mathbf{F}$ of $\mathscr{T}$ is a model of $\mathscr{T}$ such that its underlying arrow is composition of the underlying arrows of $\mathbf{G}$, $\mathbf{F}$. According to this relaxed specification, some functors with structure compose in more than one way, as shown below. On the other hand, some functors with structure don't compose at all.

The particular notion of composition depends on the particularities of the functor with structure at hand so there is a little hope of giving a general definition. However, we analyse the following three cases to provide some useful guidance to construction of compositions.

Firstly, arrows always compose. Secondly, it is well known that monads $\mathbf{T}$, $\mathbf{U}$, compose to give a monad whose underlying arrow is a composition of the underlying arrows of $\mathbf{T}$, $\mathbf{U}$, provided there is a third distributive law of $\mathbf{U}$ over $\mathbf{T}$. Dually, a similar theorem holds for comonads. Thirdly, some other nontrivial functors with structure compose and we analyse the reason.

5.3.1 Arrows

When one of the theories, $\mathscr{T}$, $\mathscr{U}$, in Def. 4.1.1 is just 2, distributive laws can be composed as morphisms of functors with structure. This corresponds to pasting, horizontal or vertical, of the underlying 2-cells, and all 2-cells that comprise the morphism of the other non-trivial functors with structure. The notation $\boxplus$, $\boxminus$, is overloaded for composition of such distributive laws in the horizontal and vertical directions, respectively. The following definition formalises this precisely.

Definition 5.3.2. *For a pair of distributive laws $\lambda, \kappa : \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K}$,*

1. *when $\mathscr{T} \equiv 2$ and $\mathbf{U}|\frac{\mathbf{H}}{\lambda}|\mathbf{U}'$ and $\mathbf{U}'|\frac{\mathbf{I}}{\kappa}|\mathbf{U}''$, the distributive law $\kappa \boxplus \lambda$ is defined as the composite $\mathbf{U}|\frac{\mathbf{IH}}{\kappa^{\rightarrow} \circ \lambda^{\rightarrow}}|\mathbf{U}''$*

2. *when* $\mathscr{U} \equiv 2$ *and* $\mathbf{F}|\overset{\mathbf{T}}{\underset{\mathbf{T}'}{\overline{\underline{\ \lambda\ }}}}|\mathbf{F}'$ *and* $\mathbf{G}|\overset{\mathbf{T}'}{\underset{\mathbf{T}''}{\overline{\underline{\ \kappa\ }}}}|\mathbf{G}'$*, the distributive law* $\kappa \boxminus \lambda$ *is defined as the composite* $\mathbf{GF}|\overset{\mathbf{T}}{\underset{\mathbf{T}''}{\overline{\underline{\ \kappa^{\downarrow} \circ \lambda^{\downarrow}\ }}}}|\mathbf{G}'\mathbf{F}'$

Clearly:

$$|\overline{\underline{\kappa \boxplus \lambda}}| \;=\; |\overline{\underline{\kappa}}| \boxplus |\overline{\underline{\lambda}}| \tag{5.25}$$

and similarly

$$|\overline{\underline{\kappa \boxminus \lambda}}| \;=\; |\overline{\underline{\kappa}}| \boxminus |\overline{\underline{\lambda}}| \tag{5.26}$$

Example 5.3.3 (Eilenberg-Moor and Kleisli liftings)**.** Eilenberg-Moore liftings (see Ex. 4.5.4), $\mathbf{F}|\overset{\mathbf{H}}{\underset{\mathbf{K}}{\overline{\underline{\ \lambda\ }}}}|\mathbf{F}$ and $\mathbf{G}|\overset{\mathbf{K}}{\underset{\mathbf{L}}{\overline{\underline{\ \kappa\ }}}}|\mathbf{G}$ compose by vertical pasting to $\mathbf{GF}|\overset{\mathbf{H}}{\underset{\mathbf{L}}{\overline{\underline{\ \kappa \boxminus \lambda\ }}}}|\mathbf{GF}$. Similarly Kleisli liftings $\mathbf{H}|\overset{\mathbf{F}}{\underset{\mathbf{F}}{\overline{\underline{\ \lambda\ }}}}|\mathbf{K}$, $\mathbf{K}|\overset{\mathbf{G}}{\underset{\mathbf{G}}{\overline{\underline{\ \kappa\ }}}}|\mathbf{L}$ compose horizontally to $\mathbf{H}|\overset{\mathbf{GF}}{\underset{\mathbf{GF}}{\overline{\underline{\ \kappa \boxplus \lambda\ }}}}|\mathbf{L}$.

Pasting of distributive laws is useful for proving that composition of underling functors preserves additional structure.

Example 5.3.4 (Strong functors compose)**.** A strong functor is a reverse morphism of right monoidal actions (see Ex. 4.5.6). Formally, strong functors are arrows in $[\mathsf{RAct}, \mathsf{Cat}]$, equivalently distributive laws $\mathbf{F}|\overset{(\times\mathscr{C})}{\underset{(\times\mathscr{C})}{\overline{\underline{\ \sigma\ }}}}|\mathbf{F}$. It follows that the composition of strong endofunctors on the same category is strong:

$$\mathbf{F}|\overset{(\times\mathscr{C})}{\underset{(\times\mathscr{C})}{\overline{\underline{\ \sigma\ }}}}|\mathbf{F} \boxminus \mathbf{G}|\overset{(\times\mathscr{C})}{\underset{(\times\mathscr{C})}{\overline{\underline{\ \tau\ }}}}|\mathbf{G} \;=\; \mathbf{GF}|\overset{(\times\mathscr{C})}{\underset{(\times\mathscr{C})}{\overline{\underline{\ \tau \boxminus \sigma\ }}}}|\mathbf{GF}$$

It follows from (5.26) that the composite strength is

$$(\tau \boxminus \sigma)_{X,Y} \;\equiv\; \mathbf{GF}X \times Y \xrightarrow{\tau_{\mathbf{F}X,Y}} \mathbf{G}(\mathbf{F}X \times Y) \xrightarrow{\mathbf{G}\tau_{X,Y}} \mathbf{GF}(X \times Y)\ , \tag{5.27}$$

which is precisely the usual definition of a composition of strengths. The abstract approach however also immediately establishes the coherence conditions, which would otherwise have to be verified by hand.

Similarly, it follows that applicative functors compose (see Ex. 4.5.8).

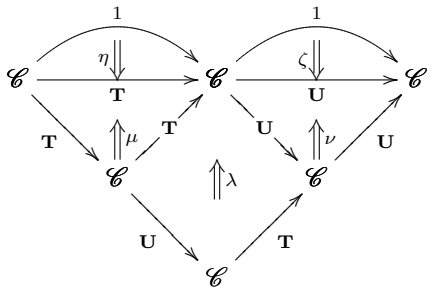


Figure 5.1: Composition of monads by a distributive law

5.3.2 Monads

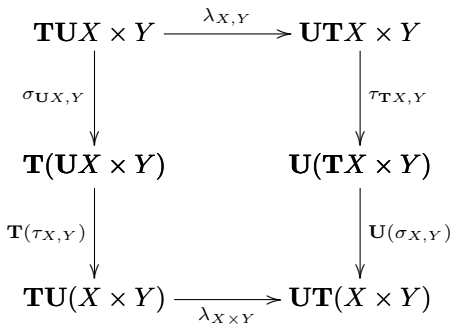
We now discuss monads (and dually comonads) as an example of functors with structure that compose nontrivially.

A distributive law λ in $\mathscr{K}$ of a monad $\mathbf{T}$ over a monad $\mathbf{U}$, both on an object X of $\mathscr{K}$, gives rise to a monad $\mathbf{UT}$ on X in $\mathscr{K}$, defined in Fig. 5.1. It follows that in order to define composites of distributive laws of or over monads, an iterative distributive law is needed. This follows from Theorem 4.1.3 as follows: a pair of distributive laws $(\mathscr{U},\mathbf{H})|\frac{(\mathsf{M},\mathbf{T})}{\lambda}|(\mathscr{U},\mathbf{H})$, $(\mathscr{U},\mathbf{H})|\frac{(\mathsf{M},\mathbf{U})}{\kappa}|(\mathscr{U},\mathbf{H})$ is equivalently a pair of monads $\lambda^{\rightarrow}$, $\kappa^{\rightarrow} : \mathsf{M} \longrightarrow [\![\mathscr{U},\mathscr{K}]\!]$. By the mentioned result about composition of monads, $\lambda^{\rightarrow}$ and $\kappa^{\rightarrow}$ compose if there is a distributive law $(\mathsf{M},\mathbf{U})|\frac{(\mathsf{M},\mathbf{T})}{\gamma^{\rightarrow}}|(\mathsf{M},\mathbf{U})$ in $[\![\mathscr{U},\mathscr{K}]\!]$, equivalently $\gamma : \mathsf{M}\otimes\mathsf{M}\otimes\mathscr{U} \longrightarrow \mathscr{K}$.

A similar argument applies to distributive laws of $\mathscr{T}$ over monads, and by duality for comonads.

Example 5.3.5 (Composition of strong monads)**.** Consider two monads, $\mathbf{T}$, $\mathbf{U}$, in $\mathscr{K}$, which compose via a distributive law λ of $\mathbf{T}$ over $\mathbf{U}$. In the case when $\mathbf{T}$ and $\mathbf{U}$ are strong, in order for the composite, $\mathbf{UT}$, to be strong, λ must be coherent with the

strength as follows:



This requirement arises from Theorem 4.3.7 as the condition in Fig. 4.5 for the arrows $\mathbf{a}$, $\mathbf{a}$, $\oslash$ in the theories M, M, RAct, respectively. The other coherence conditions are trivial in this case.

Chapter 6

Distributive Laws in Programming

In this chapter we collect examples of distributive laws in mathematically structured programming, some of which have appeared before in the literature [Mee98, GdSO09, MP08], however, without them being formally understood as distributive laws of one functor with structure over another. Our presentation improves upon these examples by showing the formal sense in which they are distributive laws. This allows us to demonstrate the sense in which the constructions previously carried out informally on them are constructions on distributive laws in a formal sense.

The last section of this chapter, Sect. 6.3, desribes our original contribution to the collection of examples of distributive laws in computer science, where distributive laws are used to compare functional programs on inductive datatypes to component-based systems, which are interpreted coinductively.

6.1 Distributive Laws of Polynomial Functors

Polynomial functors of arity $n \in \mathbb{N}$ are functors of type $\mathscr{C}^n \longrightarrow \mathscr{C}$ inductively generated by the grammar in Fig. 6.1, for a category $\mathscr{C}$ with products ($\times$) and coproducts ($+$).

Example 6.1.1 (Shape Functors)**.** The type of lists of A's in a suitable $\mathscr{C}$, for each $A \in \mathscr{C}$, is defined as the fixed-point

$$\mathsf{List}_A \;=_{\mathrm{def}}\; \mu Y.\, 1 + A \times Y \tag{6.1}$$

The fixed point exists for a suitable $\mathscr{C}$ with enough structure, e.g. when $\mathscr{C}$ is locally ω-cocomplete. The right-hand side of (6.1) defines a polynomial functor $\mathsf{ListF}_A : \mathscr{C} \longrightarrow$

$$\frac{X \in \mathscr{C}}{\underline{X}}\ \text{(constant)} \qquad \frac{}{\pi_i : \mathscr{C}^n \longrightarrow \mathscr{C} \quad ,\ i < n}\ \text{(projection)}$$

$$\frac{\mathbf{F}_1 : \mathscr{C}^n \longrightarrow \mathscr{C} \quad \mathbf{F}_2 : \mathscr{C}^n \longrightarrow \mathscr{C}}{\mathbf{F}_1 \times \mathbf{F}_2 : \mathscr{C}^n \longrightarrow \mathscr{C}}\ \text{(products)}$$

$$\frac{\mathbf{F}_1 : \mathscr{C}^n \longrightarrow \mathscr{C} \quad \mathbf{F}_2 : \mathscr{C}^n \longrightarrow \mathscr{C}}{\mathbf{F}_1 + \mathbf{F}_2 : \mathscr{C}^n \longrightarrow \mathscr{C}}\ \text{(coproducts)}$$

$$\frac{\mathbf{G} : \mathscr{C}^k \longrightarrow \mathscr{C} \quad \mathbf{F}_i : \mathscr{C}^n \longrightarrow \mathscr{C} \quad ,\ 1 < i \leq k}{\mathbf{G} \circ (\mathbf{F}_1 \vartriangle \cdots \vartriangle \mathbf{F}_k) : \mathscr{C}^n \longrightarrow \mathscr{C}}\ \text{(composition)}$$

Figure 6.1: Polynomial functors

$\mathscr{C}$ as $\underline{1} + (\underline{A} \times 1)$, for an $A \in \mathscr{C}$, and we have $\mathsf{List}_A = \mu(\mathsf{ListF}_A)$ where μ is the least-fixed-point operator $\mu : \mathscr{C}^{\mathscr{C}} \longrightarrow \mathscr{C}$.

Definition 6.1.2 (Datatypes)**.** *When the definition of* List_A *is abstracted in* A*, one obtains a regular datatype, i.e. a datatype which is given by a polynomial functor parameterised by a type. Formally, regular functors arise by adding the following rule to Fig. 6.1:*

$$\frac{\mathbf{F} : \mathscr{C}^{n+1} \longrightarrow \mathscr{C} \quad \textit{is polynomial}}{\tau\mathbf{F} \ =_{\mathrm{def}}\ \vec{X} \mapsto \mu Y.\, \mathbf{F}(\vec{X}, Y) \quad : \mathscr{C}^n \longrightarrow \mathscr{C}}\ \textit{(type functors)} \tag{6.2}$$

The least fixed point in the definition $\tau\mathbf{F}$ *in (6.2) comes with a natural isomorphism :*

$$\mathsf{in}_\tau : \mathbf{F} \circ (1^n \vartriangle \tau\mathbf{F}) \Longrightarrow \tau\mathbf{F} \ : \mathscr{C}^n \longrightarrow \mathscr{C} \tag{6.3}$$

Example 6.1.3. One can define a binary polynomial functor $\mathsf{ListF} : \mathscr{C}^2 \longrightarrow \mathscr{C}$ as

$$\mathsf{ListF} \ =_{\mathrm{def}}\ \underline{1} + (\pi_1 \times \pi_2)$$

and define the parametric regular datatype of lists $\mathsf{List} \ =_{\mathrm{def}}\ \tau\mathsf{ListF}$. Explicitly, this is given by the assignment

$$X \ \mapsto \ \mu Y.\, \underline{1} + X \times Y$$

The following is a generalisation of similar constructions of distributive laws over polynomial functors that can be found in [Mee98, HB97]. The formulation below is fully generic in the additional structure of the other functor, providing some general requirements are met. This is achieved by giving the definition entirely in terms of

distributive laws and their constructions.

Lemma 6.1.4. *Let* $\mathbf{F} : \mathscr{C}^n \longrightarrow \mathscr{C}$ *be a polynomial functor and let* $\mathbf{U}$ *be a* $\mathscr{U}$*-functor, for any theory* $\mathscr{U}$*, with the underlying arrow of type* $\mathscr{C}^m \longrightarrow \mathscr{C}$*. Let there be a*

- *coproduct on* $\mathbf{U}$ *in* $[\![\mathscr{U}, \mathsf{Cat}]\!]$
- product $(\times)$ *on* $\mathbf{U}$ *in* $[\![\mathscr{U}, \mathsf{Cat}]\!]$
- *distributive law* $\mathbf{U}^n| \dfrac{\dfrac{\underline{X}^m}{\eta x}}{\underline{X}} |\mathbf{U}$ *for any constant* $\underline{X}$ *occurring in* $\mathbf{F}$

Then there exists a distributive law $\mathbf{U}^n| \dfrac{\dfrac{\mathbf{F}^m\alpha}{\lambda}}{\mathbf{F}} |\mathbf{U}$ *where* $\alpha : (\mathscr{C}^m)^n \longrightarrow (\mathscr{C}^n)^m$ *is the obvious isomorphism natural in* n *and* m*.*

Proof. We proceed by induction on the structure of $\mathbf{F}$ as follows:

1. for $\mathbf{F} \equiv \underline{X}$, put $\lambda =_{\mathrm{def}} \mathbf{U}^n| \dfrac{\dfrac{\underline{X}^m\alpha}{\eta x}}{\underline{X}} |\mathbf{U}$.

2. for $\mathbf{F} \equiv \pi_i$, put $\lambda =_{\mathrm{def}} \mathbf{U}^n| \dfrac{\dfrac{\pi_i^m\alpha}{\pi_i}}{\pi_i} |\mathbf{U}$, the i-th projection $\pi_i : \mathbf{U}^n \longrightarrow \mathbf{U}$ in $[\![\mathbf{U}, \mathscr{K}]\!]$ given by Corollary 5.1.2.

3. for $\mathbf{F} \equiv \mathbf{F}_1 \times \mathbf{F}_2$, by induction there exist $\mathbf{U}^n| \dfrac{\dfrac{\mathbf{F}_i^m\alpha}{\lambda_i}}{\mathbf{F}_i} |\mathbf{U},\ i \in \{1, 2\}$. These are just arrows $\lambda_i^{\rightarrow} : \mathbf{U}^n \longrightarrow \mathbf{U}$, $[\![\mathscr{U}, \mathsf{Cat}]\!]$, and as there is a product $(\times)$ on $\mathbf{U}$ in $[\![\mathscr{U}, \mathsf{Cat}]\!]$, the following distributive law

$$\mathbf{U}^n \xrightarrow{\lambda_1^{\rightarrow} \vartriangle \lambda_2^{\rightarrow}} \mathbf{U}^2 \xrightarrow{(\times)} \mathbf{U}$$

   is a product of arrows in $[\![\mathscr{U}, \mathsf{Cat}]\!]$ by Lemma 5.1.14.

4. for $\mathbf{F} \equiv \mathbf{F}_1 + \mathbf{F}_2$, by induction there exist $\mathbf{U}^n| \dfrac{\dfrac{\mathbf{F}_i^m\alpha}{\lambda_i}}{\mathbf{F}_i} |\mathbf{U},\ i \in \{1, 2\}$. So we can put $\lambda =_{\mathrm{def}} \mathbf{U}^n| \dfrac{\dfrac{(\mathbf{F}_1+\mathbf{F}_2)^m\alpha}{\lambda_1 \boxplus \lambda_2}}{\mathbf{F}_1+\mathbf{F}_2} |\mathbf{U}$, which is defined by Corollary 5.2.4.

5. for $\mathbf{F} \equiv \mathbf{G}\,(\mathbf{F}_1 \vartriangle \cdots \vartriangle \mathbf{F}_k)$, by the induction hypothesis we have $\mathbf{U}^n| \dfrac{\dfrac{\mathbf{F}_i^m\alpha}{\lambda_i}}{\mathbf{F}_i} |\mathbf{U}$ and $\mathbf{U}^k| \dfrac{\dfrac{\mathbf{G}^m\alpha'}{\kappa}}{\mathbf{G}} |\mathbf{U}$, where $\alpha' : (\mathscr{C}^m)^k \longrightarrow (\mathscr{C}^k)^m$. Therefore, we also have

$\mathbf{U}^n|\,\dfrac{\dfrac{\mathbf{F}_1^m\,\alpha\triangle\cdots\triangle\mathbf{F}_k^m\,\alpha}{\overrightarrow{\lambda_1}\triangle\cdots\triangle\overrightarrow{\lambda_k}}}{\mathbf{F}_1\triangle\cdots\triangle\mathbf{F}_k}\,|\mathbf{U}^k$. Now because trivially

$$\mathbf{G}^m\,\alpha'\,(\mathbf{F}_1^m\,\alpha\,\triangle\,\cdots\,\triangle\,\mathbf{F}_k^m\,\alpha)\;=\;(\mathbf{G}\,(\mathbf{F}_1\,\triangle\,\cdots\,\triangle\,\mathbf{F}_k))^m\,\alpha$$

we can put

$$\lambda\;=_{\mathrm{def}}\;\mathbf{U}^n|\,\frac{\dfrac{(\mathbf{G}\,(\mathbf{F}_1\triangle\cdots\triangle\mathbf{F}_k))^m\,\alpha}{(\overrightarrow{\lambda_1}\triangle\cdots\triangle\overrightarrow{\lambda_k})\boxplus\kappa}}{\mathbf{G}\,(\mathbf{F}_1\triangle\cdots\triangle\mathbf{F}_k)}\,|\mathbf{U}$$

□

In the rest of this section we give details of how Lemma 6.1.4 corresponds to the existing instances given before elsewhere.

6.1.1 Functor Pulling

In [Mee98], Lambert Meertens defines a *functor puller* to be a natural transformation of type

$$p_{\mathbf{F},\mathbf{H}}\;:\;\mathbf{F}\mathbf{H}^n\Longrightarrow\mathbf{H}\mathbf{F}^m\alpha \tag{6.4}$$

for an n-ary functor $\mathbf{F}$ and an m-ary functor $\mathbf{H}$, where α is the natural isomorphism $\alpha : (\mathscr{C}^m)^n \longrightarrow (\mathscr{C}^n)^m$.

Define $2_{n,1}$ to be the theory generated by the sketch

$$\mathbf{o}^n\longrightarrow\mathbf{o}\;.$$

Up to α, (6.4) is a distributive law of $2_{n,1}$ over $2_{m,1}$ in Cat.

$$\begin{array}{ccc}(\mathscr{C}^m)^n\cong(\mathscr{C}^n)^m & \xrightarrow{\mathbf{F}^m} & \mathscr{C}^m\\ \Big\downarrow{\scriptstyle\mathbf{H}^n} & \overset{p_{\mathbf{F},\mathbf{H}}}{\Longrightarrow} & \Big\downarrow{\scriptstyle\mathbf{H}}\\ \mathscr{C}^n & \xrightarrow[\mathbf{F}]{} & \mathscr{C}\end{array} \tag{6.5}$$

Meertens shows that functor pullers exist for all regular datatypes $\mathbf{F}$. For the special case of functors pullers, i.e. when $\mathbf{F}$ in (6.5) is just a functor with no additional structure, it's easy to extend the construction in Lemma 6.1.4 to *type functors* by defining a $\mathbf{H}^n|\,\dfrac{\tau\mathbf{F}^m}{\tau\lambda}\,|\mathbf{H}$, for $\mathbf{H}^n|\,\dfrac{\mathbf{F}^m}{\lambda}\,|\mathbf{H}$ by initiality of $\tau\mathbf{F}$.

Degenerated examples of pullers are so-called *crushes*, functions of type $\mathbf{F}X\longrightarrow$

X for any regular functor $\mathbf{F}$ and a constant X. An example of a crush is

$$\texttt{sum} : \texttt{List}(\mathbb{Z}) \longrightarrow \mathbb{Z} \ .$$

Example 6.1.5 (Applicative Functor Pullers)**.** It is shown in Sect. 5.2.4 that applicative functors have products as objects in $[\![\mathsf{Af}, \mathsf{Cat}]\!]$. By Corollary 5.2.2 they also have coproducts in $[\![\mathsf{Af}, \mathsf{Cat}]\!]$. Moreover, applicative functors have units, i.e. for each applicative $\mathbf{L}$, there is a natural transformation $\eta_X : X \longrightarrow \mathbf{L}X$. To see this, consider ν_1 in (3.18), which in the case of applicative functors, i.e. $\otimes \equiv \oplus \equiv (\times)$ and $e = e' = 1$, reduces to a point $u : 1 \longrightarrow \mathbf{F}1$. Together with strength, σ, one obtains η_X above as

$$X \cong 1 \times X \xrightarrow{u \triangle X} \mathbf{L}(1) \times X \xrightarrow{\sigma_{1,X}} \mathbf{L}(1 \times X) \xrightarrow{\mathbf{L}\lambda} \mathbf{L}(X)$$

Components of this natural transformation are distributive laws $\mathbf{L}|\ \frac{\underline{X}}{\underline{\eta_X}}\ |\mathbf{L}$. It follows by Lemma 6.1.4 that distributive laws of polynomial endofunctors over applicative functors always exist.

In [MP08], `iFunctor` is defined as a functor which distributes over all applicative functors. Several examples of `iFunctor` are shown which are regular datatypes, such as `Tree` (trees) or `Expr` (expressions). Lemma 6.1.4 therefore almost explains these examples, except for the fixed points for which we would need a lifting theorem similar to Theorem 5.1.9 or Corollary 5.2.2.

6.2 Zips and Traversals

6.2.1 Unzip

The function called `unzip` in HASKELL takes a list of length n of pairs into a pair of lists of equal length n of the first and second components. The type of the function in the Standard HASKELL Library (SHL) is:

```
unzip :: [(a,b)] -> ([a],[b])
```

It is easily provable from the implementation (not shown) that `unzip` satisfies the equations:

$$\mathsf{List}\,\pi_1 \ = \ \pi_1 \cdot \texttt{unzip}_{A,B} \tag{6.6}$$

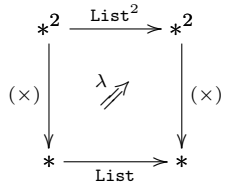
$$\mathsf{List}\,\pi_2 \ = \ \pi_2 \cdot \texttt{unzip}_{A,B} \tag{6.7}$$

$$\tag{6.8}$$

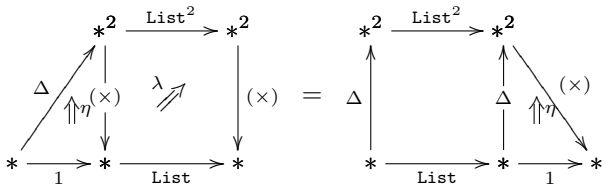
These equations characterise `unzip` by stating formally that first component of the result is the list of first components of the argument. And similarly for the second

components. This defines `unzip` formally as a distributive law of lists over pairs.

To this end, let `List` : $* \longrightarrow *$ be the type constructor of lists, i.e. the functor that takes a type, s, to the type of finite lists of elements of s and $(\times) : *^2 \longrightarrow *$ the cartesian product. Then `unzip` is a natural transformation λ in

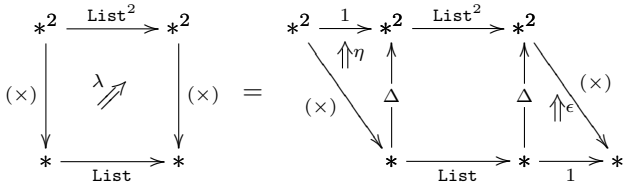


Let $(\Delta, (\times), \eta, \epsilon)$ be the adjunction defining $(\times)$ in $*$. Now, observe that (6.6) and (6.7) are formally the equations:



(6.9)

By pasting $\epsilon : \Delta(\times) \Longrightarrow 1$ on the left of both sides in (6.9), and using the triangle identities we get a definition of `unzip`:



(6.10)

This defines `unzip` as so-called *mate* [KS74] of the identity 2-cell under the adjunction $(\Delta, (\times), \eta, \epsilon)$. To see that `unzip` is coherent with the structure of the monad on `List`, please consult Sect. 5.1, Equation (5.5) and the proofs of coherence that follow.

6.2.2 Sequence

Any instance of the function

```
sequence :: Monad m => [m a] -> m [a]
```

taking a list of monads into a monad on lists, which is defined in SHL, is a distributive law of lists over the monad, `m`. Note this doesn't define `sequence` uniquely. There are at least two distributive laws of the given type: one sequencing the effects of the monads left to right, and the other one sequencing them in the reverse order.

Similarly, one can define

```
sequence :: Applicative i => [i a] -> i [a]
```

for an applicative functor `i`. This is just a distributive law of lists over `i` as an applicative functor. Formally, a functor $\mathsf{L} \otimes \mathsf{Af} \longrightarrow \mathscr{K}$, where L is a theory of lists. These are both examples of distributive laws of regular functors (the list) which are not pullers in the sense of [Mee98] as the structure of the other functor is important.

Remark 6.2.1. Functions such as

```
transpose :: [[x]] -> [[x]]
repeat :: x -> [x]
zip :: ([x],[y]) -> [(x,y)]
```

i.e. matrix transposition, infinite duplication, and list zipping, are essentially distributive laws for lists considered as monads or applicative functors. The importance of this observation is that such distributive laws can be constructed generically by operations on distributive laws from primitive components as demonstrated above. This brings along properties of these functions, which hold automatically once they've been constructed in such a high-level way. There is a problem though, which didn't appear for `unzip`, say, that none of the above functions is total as they are only defined on arguments of the correct shape. The solution, which we leave for the future work, would be to make the types more precise by making information about shape a part of the datatype. In this way, one could be precise and define for instance a *total* function $\texttt{zip}_n$ on pairs of lists of length n producing a list of pairs of length n.

6.3 Adequacy of Functional and Component-based Programming

In the current programming practice, there are two prevailing programming paradigms: component-based programming (CBP), with the notable example of object-oriented programming[1] (OOP), and functional programming (FP). In this chapter, a novel application of distributive laws to formally compare programs in these two paradigms is presented.

Informal comparisons of the paradigms have been carried out before. Most notably, Cook in [Coo91] argues that the two styles are characterised by a different grouping of programming fragments, namely by data and behaviour, and that this leads to a mismatch in extensibility. This is dubbed *the expression problem* by Cook. Other notable contributions are Buchlovsky and Thielecke's paper [BT06] non the so-called *Visitor Pattern* in object-oriented programming (see [GHJV94]). Jeremy Gibbons argues

[1] The term *component-based programming* is used rather than the much more common *object-oriented programming*, which is its widespread example. The term conveys our focus on a small subset of what comprises the discipline of object-oriented programming.

in [Gib06] that higher-order datatype-generic programs are in many instances the FP equivalents of patterns in object-oriented software design. In [GdSO09], he and Bruno Oliveira argue that what we call a distributive law of a dataype over an applicative functor is the correct formal counterpart of so-called *Iterator pattern* in OOP.

Implicit in all such comparisons of programs in the two paradigms is a notion of their equivalence as one must compare "the same program" written in two styles. In this chapter we develop such a notion of equivalence, which rests on the notion of a distributive law.

We start by describing informally the paradigm of component-based programming, with the aim of it being later interpreted coalgebraically. Then we remind the reader of the *initial algebra semantics* of inductive functional programming and *coalgebraic semantics* of state based systems. Our main contribution is in the observation that in the interesting cases both the algebra, which defines an recursive function, and the coalgebra, which defines a component based system, arise from a single distributive law of *data over behaviour*. This makes it possible to apply category theory to establish a bisimulation relation between the two original programs.

6.3.1 Component-based Programming

Omitting many technical details, the *component-based* approach can be characterised as programming with discrete software components. Each component has an *interface* and a *local state*. Components are organised into oriented graphs. In more detail, each component has an *address* and as a part of the local state it keeps a collection of addresses of other components, so-called *pointers*. This defines a directed graph on addresses of components.

At the runtime, the components can send *messages* to other components, and can be sent messages by other components and from the *environment*, say by the user operating a user interface (keyboard, mouse). All possible senders of messages to a component are called *clients*. The messages each component can *receive* are specified in the interface of each component. A specification of a message contains an identifier, an *arity* and a *return type*. The *arity* of each message specifies additional data that can be used to *parameterise* the message. The return type specifies the type of data that is being returned to the sender in response to the message. This data is also called a *return value*. Both parameters and return values can include pointers. In order to calculate the return value, a component can execute calculations on the local state, alter the local state, send messages to other components and incorporate their return values to the calculation. It can also *create new components* and store the pointers to them in the local state and thus alter the graph of components.

In summary, a *component-based program* consists of a specification of the components that can appear in the system. The specification of each component consists of an

interface, type of the local state and the code executed in response to each valid message in the interface. The latter is usually realised in a low-level structural programming language providing basic arithmetics, conditional statements and loops, string manipulation, message invocation and component creation. In addition, a specification of the initial graph of components is required, which is often external, or fixed upfront. The initial graph is often trivial: just one component, which is gradually expanded during the execution of the program. In such a case, the initial component has a chosen message whose execution, or rather the process of calculation of its return value is equivalent with the execution of the whole program. In other examples, the graph is fixed throughout the runtime (component creation is prohibited) and the execution of the program is a collection of responses to messages[2] being sent to its components by the environment.

Encapsulation One of the cornerstones of CBP is so-called *encapsulation*, which is the property that any access or update to the local state of a component can take place only within the execution of a message of the component. In other words, components cannot access the local state of other components directly. This allows, for instance, for the graph of components to be distributed over a network, with the components implemented in different programming languages and executed on different physical machines.

Encapsulation has a major impact on the programming style of CBP.

Examples Examples of CBP systems are object-oriented programming, and distributed programming systems such as COM or CORBA.

The nonspecification of data structures Most component based programming systems lack mechanisms for specification for higher-level datatypes. It means that all datatypes are primitive such as integers, characters and pointers, and there is no way to specify new composite datatypes. Instead the primitive datatypes, wrapped as local state in components, are linked by pointers as pieces of data into graphs which are used to model more complicated data structures. For instance, a *finite list of integers* is modelled as a linear chain of components, each carrying an integer as its local state and a pointer to the next component.

$$[n_1] \longrightarrow [n_2] \longrightarrow \cdots \longrightarrow [n_k]$$

Although this allows one to model datatypes, CBP in general lacks any means of higher-order specification of the organisation of components. This means that during the runtime of a program components can enter in communication with any other

[2] often asynchronous

component they happen to be linked to. The changes to the graph of components and the admissible configurations are implied by the operational *behaviour* of the system rather than specified and enforced. This also means that datatypes modelled as components are unspecified to a large extent.

In this sense, CBP constrains behaviour but doesn't constrain structure. Behaviour is constrained by the means of interfaces. On the other hand, structure (the graph of components) is unconstrained. For instance, even though one can model lists as a collection of components ordered in a chain, there is no formal way of enforcing this organisation of the components. So a list is merely a component that behaves like a list in the sense that the messages sent to it allow other components to add elements – new components into the chain – and look up and remove components, and that this communication is coherent with the expected behaviour of lists: e.g. after sending a message "add 3" a subsequent message "find 3" returns "yes", and a message "remove 3" results in all messages "find 3" returning "no" until "3" is again added. In this sense,

Component-based programming is programming of data from behaviour.

6.3.2 Functional programming

In contrast, functional programming (FP) is centred around specifications of data, so-called datatypes. Datatypes are specified in a high-level algebraic way by enumeration of *constructors* that generate the set of *elements* of the datatype. One then defines *functions* that are applied to elements of datatypes. This is done by *pattern matching* on values of a datatype and by specification of a value of the function for each pattern. The values are in general other functions applied to arguments.

In contrast to CBP, there can be arbitrarily many functions defined on each datatype. When the definition of a datatype changes all functions pattern matching on the datatype must change as well. In this sense, datatypes come first, functions later.

A *functional program* is a collection of all definitions of datatypes and functions on the datatypes. The meaning of a functional program is the value of a chosen function on chosen arguments (e.g., the function called "main" applied to the command-line arguments). With respect to a functional program, a value of a chosen datatype can be assigned a *behaviour* informally as all values reachable by application of functions in the program to the value and recursively to the subsequent values of the datatype arising in this process. Examples of FP languages are Haskell, ML or Clean. In summary, in the sense described above:

Functional programming is programming of behaviour on data.

6.3.3 Initial algebra semantics

The following is an overview of some elementary facts about the so-called *initial algebra semantics* of datatypes in functional programming[GTW78, Awo06]. Nothing new of substance is presented in this section. The reader is reminded of section Sect. 4.4.1, where the notions summarised below in the usual fashion are given a uniform treatment by distributive laws.

First, recall that an *algebra of an endofunctor* $\mathbf{F}$ is an arrow $\varphi : \mathbf{F}X \longrightarrow X$, where X is called the *carrier of* φ. Algebras are often denoted by a tuple, (φ, X), where X is the carrier of φ. A morhpism from φ to $\psi : \mathbf{F}Y \longrightarrow Y$ is an arrow $f : X \longrightarrow Y$ such that the square $\psi \cdot \mathbf{F}f = f \cdot \varphi$ commutes.

$$\begin{array}{ccc} \mathbf{F}X & \xrightarrow{\mathbf{F}f} & \mathbf{F}Y \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\psi} \\ X & \xrightarrow[f]{} & Y \end{array} \tag{6.11}$$

Such squares compose by horizontal pasting, and with the obvious unit form a category, $\mathbf{F}$-$\mathbf{Alg}$.

The initial object in $\mathbf{F}$-$\mathbf{Alg}$, if it exists, is an algebra

$$\mathsf{in}_{\mathbf{F}} : \mathbf{F}\mu_{\mathbf{F}} \longrightarrow \mu_{\mathbf{F}}$$

It is necessarily an isomorphism by the well-known Lambeck's Lemma. Initiality of $\mathsf{in}_{\mathbf{F}}$ means explicitly that for any other $\mathbf{F}$-algebra (φ, X), there exists a unique algebra morphism, $(\!|\, \varphi \,|\!) : \mu_{\mathbf{F}} \longrightarrow X$, sometimes called the *fold of* φ. More precisely, $(\!|\, \varphi \,|\!)$ is the underlying arrow in the algebra morhpism

$$\begin{array}{ccc} \mathbf{F}\mu_{\mathbf{F}} & \xrightarrow{\mathbf{F}(\!|\, \varphi \,|\!)} & \mathbf{F}X \\ {\scriptstyle\mathsf{in}_{\mathbf{F}}}\downarrow & & \downarrow{\scriptstyle\varphi} \\ \mu_{\mathbf{F}} & \xrightarrow[(\!|\, \varphi \,|\!)]{} & X \end{array} \tag{6.12}$$

Algebras of functors have a long time ago been identified as a good formal basis for mathematics of functional programming (see e.g. [GTW78, Mal90, MFP91, BdM96]). The key fact behind their usefulness in this context is that *regular datatypes*, roughly datatypes defined by enumeration of a finite number of all admissible constructors of new elements of a datatype from old elements and constants, correspond to initial algebras of polynomial functors. Here, the role of the polynomial functor is to define the *signature* (or *shape*) of the datatype, i.e. the sum of alternatives of the datatype.

```
data List a = Nil | Cons a (List a)

foldList :: b -> (a -> b -> b) -> List a -> b
foldList m f Nil = m
foldList m f (Cons x xs) = f x (foldList m f xs)


data Expr a = Var a | Val Int | Add (Expr a) (Expr a)

foldExpr :: (a -> b) -> (Int -> b) -> (b -> b -> b) ->
         Expr a -> b
foldExpr f g h (Var a) = f a
foldExpr f g h (Val n) = g n
foldExpr f g h (Add t u) = h (foldExpr t) (foldExpr u)
```

Figure 6.2: The inductive datatypes of lists and expressions

Figure 6.2 shows examples of an inductive datatypes in the functional programming language HASKELL. Here, the datatype `Expr a` is defined to consist of elements labelled `Var` standing for variables, which carry data of type `a` (a type parameter in the definition), of elements labelled `Val` standing for terminals carrying integers, and of elements labelled `Add`, which carry a pair of other (previously constructed) expressions. Nothing else is in the datatype. Such a datatype can be modelled for each object A as an *initial algebra of an endofunctor*, $\mathbf{E}_A$, where the functor (called *shape functor*) acts on a bicartesian closed category, $\mathscr{C}$, of "types and functions" such as Set. It captures the arities of all constructors as follows:.

$$\mathbf{E}_A X \;=\; A + \mathbb{Z} + X \times X \tag{6.13}$$

Here, A is an object in $\mathscr{C}$ standing for the collection of variables in the expression; $+$ stands for coproduct in $\mathscr{C}$ (disjoint union in Set) and $\times$ is the product (cartesian product in Set). Therefore, the assignment (6.13) expresses formally the fact that elements in $\mathbf{E}_A X$ are either variables from A, natural numbers or elements of $X \times X$. We define

$$\mathtt{Expr}_A \;=_{\mathrm{def}}\; \mu_{\mathbf{E_A}} \tag{6.14}$$

for each $A \in \mathscr{C}$. Now, the carrier, $\mathtt{Expr}_A$, of the *initial algebra* of $\mathbf{E}_A$ is the type of expressions constructed inductively from the empty type, 0, by a finite iteration of application of the constructors. In other words, it is the least solution of the domain-theoretic equation

$$\mathtt{Expr}_A \;\cong\; A + \mathbb{Z} + (\mathtt{Expr}_A)^2$$

Functions on inductive datatypes are often defined by structural recursion, that is, by

specifying a collection of functions, one for each constructor of the datatype, which are used recursively to collapse each level of constructors in an inductively defined element of the datatype into a value. In the categorical setting, this amounts to giving an arrow of type $\mathbf{E}_A(X) \longrightarrow X$ – an $\mathbf{E}_A$-algebra. The universal arrow out of the initial algebra into the algebra then corresponds to the recursive application of the algebra at each level. This follows from an operational reading of (6.12), which is equivalent to the following, because $\mathsf{in}_{\mathbf{E}_A}$ is an isomorphism:

$$(\!|\, \varphi \,|\!) \;=\; \varphi \cdot \mathbf{E}_A(\!|\, \varphi \,|\!) \cdot {\mathsf{in}_{\mathbf{E}_A}}^{-1}$$

In words, an element of $\mathtt{Expr}_A$ is first taken apart to reveal one level of constructors, then $(\!|\, \varphi \,|\!)$ is applied to subterms, and finally φ is applied to the results to construct a new X. Figure 6.2 gives a definition of a *combinator* `foldExpr` that does exactly this for any $\mathbf{E}_A$-algebra. The reader can observe that `foldExpr` operates by case analysis on elements of the datatype, and that it is structurally recursive[3].

The uniqueness of the universal arrow out of the initial algebra corresponds in the programming-language interpretation to uniqueness of the function defined by structural recursion. Explicitly, a function defined by structural recursion is uniquely determined by its value on each of the constructors. Equivalently, any algebra morphism out of the initial algebra is uniquely determined by its target algebra. It therefore suffices to compare algebras in order to reason about equality of functions defined by structural recursion.

An important reasoning principle stemming from uniqueness is so-called *fusion* which makes it possible to express the composition of an algebra morphism after a fold as a fold. Formally:

$$f \cdot (\!|\, \varphi \,|\!) \;=\; (\!|\, \psi \,|\!) \qquad \Leftarrow\; f : \varphi \longrightarrow \psi \tag{6.15}$$

6.3.4 Algebras for Monads and Equations

Similar considerations apply to algebras for monads; see [Awo06, Mac97]. In particular, for any polynomial endofunctor $\mathbf{F}$ on a locally ω-cocomplete category $\mathscr{C}$, there is the free monad, $\mathbf{F}^* : \mathscr{C} \longrightarrow \mathscr{C}$ defined on X as the least fixed point

$$\mathbf{F}^*X \;=\; \mu Y.X + \mathbf{F}Y \;\;, \tag{6.16}$$

where μ binds Y in the rest of the expression. The assignment (6.16) extends to a functor in the obvious way and defines a monad (see [Awo06]). For $\mathscr{C} \equiv \mathsf{Set}$, and $\mathbf{F}$ representing a single-sorted signature Σ, $\mathbf{F}^*X$ is the set of finite terms constructed from

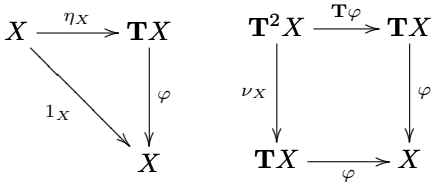
[3]Haskell and other functional languages don't ensure that functions such as `foldExpr` are proper structurally recursive functions.

Σ on the set of variables X. The multiplication of the free monad, μ, is substitution, and unit, η, is the interpretation of variables as trivial terms.

An example of a free monad is $\mathtt{Expr}_X$ when considered as functor in X. Formally,

$$\mathtt{Expr}_X \equiv (\mathbb{N} + (-)^2)^*(X)$$

In general, for any monad $(\mathbf{T}, \nu, \eta)$, a $\mathbf{T}$-algebra, φ, is an algebra of the underlying functor satisfying the additional coherence conditions, for any X:



(6.17)

Morphisms of $\mathbf{T}$-algebras are again commuting squares (6.11). The category of $\mathbf{T}$-algebras is denoted $\mathbf{T}$-$\mathbf{Alg}$. The forgetful functor $\mathbf{U}^{\mathbf{T}}$, sending an algebra to its carrier, and a morphism to the underlying arrow on carriers has a left adjoint, $\mathbf{F}^{\mathbf{T}} \dashv \mathbf{U}^{\mathbf{T}}$ defined as

$$X \quad \longmapsto \quad \mathbf{T}^2 X \xrightarrow{\nu_X} \mathbf{T}X \tag{6.18}$$

$$f \quad \longmapsto \quad \mathbf{T}f \tag{6.19}$$

Because $\mathbf{F}^{\mathbf{T}} : \mathscr{C} \longrightarrow \mathbf{T}$-$\mathbf{Alg}$ is a left adjoint, if $\mathscr{C}$ has an initial object, 0,

$$\mathbf{F}^{\mathbf{T}}(0) \;\equiv\; \mathbf{T}^2 0 \xrightarrow{\mu_{\mathbf{T}0}} \mathbf{T}0$$

is an initial object in $\mathbf{T}$-$\mathbf{Alg}$, i.e. an initial $\mathbf{T}$-algebra. As before, (the underlying arrow of) an initial algebra morphism is denoted $(\!|\, \varphi \,|\!)$, where φ is an algebra of the monad.

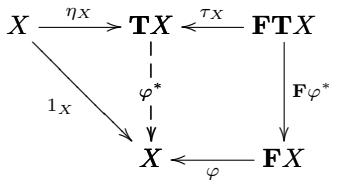
Importantly, algebras of a free monad, $\mathbf{F}^*$ are equivalent to $\mathbf{F}$-algebras by the assignment:

$$\mathbf{F}X \xrightarrow{\varphi} X \quad \longmapsto \quad (\!|\, 1_X \triangledown \varphi \,|\!) =_{\mathrm{def}} \varphi^* \tag{6.20}$$

$$\mathbf{F}^*Y \xrightarrow{\psi} Y \quad \longmapsto \quad \mathbf{F}Y \xrightarrow{\mathbf{F}\eta} \mathbf{F}\mathbf{F}^*Y \xrightarrow{\tau_Y} \mathbf{F}^*Y \xrightarrow{\psi} Y \;, \tag{6.21}$$

where $\tau : \mathbf{F}\mathbf{F}^* \Longrightarrow \mathbf{F}^*$ is the inclusion of terms with at least one level of symbols from $\mathbf{F}$ as arbitrary terms. It's worth noting that $\mathbf{F}^*0 \cong \mu\mathbf{F}$; set theoretically: the set of ground terms. In (6.20), the fold is out of the initial algebra (6.16), i.e., φ^* is defined

by the universal arrow in the following situation:



Operationally, the free algebra, φ^*, can be thought of as being defined by iteration of φ.

The story of terms and monads continues for monads not generated just freely, but modulo a set of equations on terms. Such monads are called *term monads* and denoted $\mathbf{T}_\Sigma$, where $\mathbf{T}$ stands for "terms" and Σ is a single sorted signature. Even though the equivalence of Σ-algebras and $\mathbf{T}_\Sigma$-algebras doesn't hold for this case, one can still see functions defined by initiality as being defined by structural recursion on the terms modulo the equations.

6.3.5 Coalgebraic component-based programming

The formalities in the previous section dualise for coalgebras and comonads. In summary:

1. A coalgebra (X, ψ) of an endofunctor, $\mathbf{G} : \mathscr{C} \longrightarrow \mathscr{C}$ is an arrow $\psi : X \longrightarrow \mathbf{G}X$

2. The final coalgebra is given by the greatest fixed point of the equation

$$\nu_{\mathbf{G}} \cong \mathbf{G}\nu_{\mathbf{G}} \ ,$$

   and it is often denoted $\mathsf{out}_{\mathbf{G}}$, or just out, and has type $\nu_{\mathbf{G}} \longrightarrow \mathbf{G}\nu_{\mathbf{G}}$. It is an isomorphism.

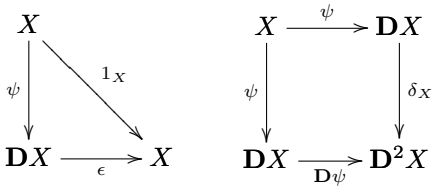
3. The underlying arrow of a morphism $\psi \longrightarrow \mathsf{out}_{\mathbf{G}}$ is denoted $[\!(\, \psi \,)\!]$ and called the *unfold* of ψ.

4. The cofree comonad over $\mathbf{G}$ is defined by the greatest fixed point

$$\mathbf{G}_\infty X \ =_{\mathrm{def}} \ \nu Y.X \times \mathbf{G}Y$$

5. A coalgebra, ψ, of a comonad $(\mathbf{D}, \delta, \epsilon)$ is a coalgebra of the underlying functor

satisfying in addition the dual of (6.17):



(6.22)

The category of $\mathbf{D}$-coalgebras is denoted $\mathbf{D}$-$\mathbf{CoAlg}$.

6. The forgetful functor $\mathbf{U_D} : \mathbf{D}\text{-}\mathbf{CoAlg} \longrightarrow \mathscr{C}$ has a right adjoint $\mathbf{U_D} \dashv \mathbf{F_D}$ defined by

$$\mathbf{F_D}(X) \;=\; \mathbf{D}X \xrightarrow{\delta_X} \mathbf{D}^2X$$

   As right adjoints preserve limits, $\mathbf{F_D}(1)$ is the final $\mathbf{D}$-coalgebra.

7. $\mathbf{G}_\infty(1) \cong \nu_\mathbf{G}$, when this makes sense.

It is well known [Jac96, Rei95, Rut00] that coalgebras can be used to give semantics to objects in object-oriented programming, or state-based systems in general, of which our component-based systems are an instance.

Consider the following definition of an interface of a component in a hypothetical component-based language, which is similar in nature to the mainstream object-oriented languages.

```
interface IExpr
  attribute eval:() -> Int
  procedure mult:Int -> ()
```

In general and somewhat informally, interfaces in the language are given by a collection of *attributes* and *procedures*, called collectively *methods*. *Attributes* are values, observations, potentially parametrised, which don't change the local state. *Procedures* are sequences of commands that accept a parameter, change the local state accordingly and possibly return an observable value. The above example describes an interface of a component representing expressions, where the attribute `eval` returns the value of the expression and the procedure `mult` presumably[4] multiplies the expression by a parameter by altering the local state of the expression, i.e. changing the expression in place rather than returning a copy.

[4]Guessing from the name. There is nothing in the interface specifying the meaning of `mult`.

Such an interface can be modelled categorically as an endofunctor, $\mathbf{G} : \mathscr{C} \longrightarrow \mathscr{C}$ defined on a bicartesian closed category $\mathscr{C}$ as:

$$\mathbf{G} \;=_{\mathrm{def}}\; \prod_{i \leq n} \mathbf{G}_m \tag{6.23}$$

where each $\mathbf{G}_m$ stands for one method, m, in the interface. It is defined as follows:

attributes are represented by constants – function spaces B^A for an attribute

```
attribute a: A -> B     ,
```

where A and B are the interpretations of the type symbols `A`, `B` as objects in $\mathscr{C}$.

procedures are represented by the functor $(- \times B)^A$ for a declaration

```
procedure: A -> B
```

So a $(- \times B)^A$-coalgebra,

$$\psi : X \longrightarrow (X \times B)^A \;\cong\; X \times A \longrightarrow X \times B$$

is thought of as a function taking the current state and a parameter, $(x, a) \in X \times A$, to a new state and a return value in $X \times B$.

In summary, `IExpr` corresponds to the functor:

$$\mathbf{G}X \;\equiv\; \mathbb{Z} \times X^{\mathbb{Z}} \tag{6.24}$$

As described before, in a component based system, each component has potentially a different interface and the components are linked by pointers, sending each other messages. For the time being, we describe the standard coalgebraic semantics of just one object. This is itself a state-based system.

Fix an endofunctor $\mathbf{G}$ describing the *interface* of a state-based system. A colagebra $\psi : X \longrightarrow \mathbf{G}X$ specifies one step of the system with state in X, producing all possible observable behaviour in $\mathbf{G}X$. Dually to the case of algebras where all possible constructors are collected into one functor, here all possible behaviour is collected in a functor $\mathbf{G}$. The final $\mathbf{G}$-coalgebra, $\mathtt{out}_{\mathbf{G}} : \nu_{\mathbf{G}} \longrightarrow \mathbf{G}\nu_{\mathbf{G}}$, if it exists, represents the abstract system with the given interface $\mathbf{G}$. This system is capable of any behaviour specified by $\mathbf{G}$. Given any concrete system $\psi : X \longrightarrow \mathbf{G}X$, the arrow

$$[\![\psi]\!] : X \longrightarrow \nu_{\mathbf{G}}$$

into the final coalgebra can be thought of as giving for any element $x \in X$, a state, $[\![\psi]\!](x) \in \nu_{\mathbf{G}}$, of the universal system. The state is such that a run of the abstract

system, $\mathsf{out}_{\mathbf{G}}$, in this state simulates ψ in state x in the sense that the two systems produce exactly the same behaviour. I.e., an outside observer wouldn't be able to distinguish them by observations only. So $[\![\psi]\!]$ can be thought of as a *constructor* of an abstract system from an initial state x with behaviour ψ.

This models faithfully encapsulation in CBP because one can no longer observe the concrete state in $\nu_{\mathbf{G}}$ in the same sense as one cannot observe the true nature (implementation details) of components in CBP. All that is observable is the behaviour $\mathbf{G}(\nu\mathbf{G})$. Moreover, the uniqueness of $[\![\psi]\!]$ means that any behaviour of ψ in state x is represented by a unique element of $\nu_{\mathbf{G}}$, so one can use equality in $\nu_{\mathbf{G}}$ to compare behaviour. Informally, two elements $x, y \in \nu_{\mathbf{G}}$ are equal if and only if they have the same observable behaviour.

The arrow, $\mathsf{out}_{\mathbf{G}}$ of the final coalgebra performs one step of the abstract system in all possible methods. For $\mathbf{G}$ of the form (6.23),

$$\nu_{\mathbf{G}} \xrightarrow{\mathsf{out}_{\mathbf{G}}} \mathbf{G}\nu_{\mathbf{G}} \xrightarrow{\pi_i} \mathbf{G}_i\nu_{\mathbf{G}}$$

retrieves the i-th method of the interface that $\mathbf{G}$ represents. For $\mathbf{G}_i \equiv (- \times B)^A$, $x \in \nu_{\mathbf{G}}$, $a \in A$, the arrow

$$1 \xrightarrow{x \triangle a} \nu_{\mathbf{G}} \times A \xrightarrow{\mathsf{out}_{\mathbf{G}} \times A} \mathbf{G}\nu_{\mathbf{G}} \times A \xrightarrow{\pi_i \times A} \mathbf{G}_i\nu_{\mathbf{G}} \times A \xrightarrow{\mathbf{app}} \nu_{\mathbf{G}} \times B \ , \tag{6.25}$$

where $\mathbf{app}$ is application given by the closed structure of $\mathscr{C}$, can be thought of as sending message i to x, with the argument a. The result is the pair (*new state*, *return value*). The situation is similar for attributes.

From this point onwards the standpoint is taken that datatypes are formally carriers, $\mu_{\mathbf{F}}$, of initial $\mathbf{F}$-algebras, for some functor $\mathbf{F}$, and that well defined functions on datatypes are defined by structural recursion, i.e. categorically they are folds $(\!| \varphi |\!)$ for some algebra $\varphi : \mathbf{F}X \longrightarrow X$. From this point of view many interesting functions in FP are necessarily folds, or they factor through a fold. We observe that often the algebras, $\varphi : \mathbf{F}X \longrightarrow X$ of folds $(\!| \varphi |\!)$ have the form $\psi \cdot \kappa_X : \mathbf{FG}X \longrightarrow \mathbf{GF'}X \longrightarrow \mathbf{G}X$ for some distributive law $\kappa : \mathbf{FG} \Longrightarrow \mathbf{GF'}$ and a $\psi : \mathbf{F'}X \longrightarrow X$. This is investigated and illustrated below.

6.3.6 Programming expressions, in two ways

As an introduction to the more general development to follow, we show an example of one problem and two solutions, a functional one and a component-based one. A categorical semantics is given to both, an algebraic and a coalgebraic one, respectively, which is later compared formally and a notion of their equivalence is shown.

```
data TExpr = CVal Int | CAdd TExpr TExpr

feval (CVal n) = n
feval (CAdd t u) = (feval t) + (feval u)

fmult x (CVal n) = CVal n*x
fmult x (CAdd t u) = CAdd (fmult x t) (fmult x u)
```

Figure 6.3: Expressions

Expressions, functionally

Figure 6.3 shows a datatype of ground expressions `TExpr` and a pair of functions: `feval` evaluating an expression to a number and `fmult` multiplying each expression by an integer parameter, x. We give this program a categorical semantics as an arrow in a bicartesian closed category $\mathscr{C}$. The category must be such that the following functor has a least fixed point (e.g. ω-cocomplete).

$$\mathbf{E} =_{\text{def}} \mathbb{Z} + (-)^2 \quad , \tag{6.26}$$

where $\mathbb{Z}$ is an object of $\mathscr{C}$ with monoidal functions $(\times) : \mathscr{C}^2 \longrightarrow \mathscr{C}$ and $(+) : \mathscr{C}^2 \longrightarrow \mathscr{C}$; 2 is the two object category, so $X^2 \cong X \times X$. In the following text the semantic brackets $[\![,]\!]$ are used informally to denote interpretations of symbols in the program as arrows and objects in $\mathscr{C}$. Types are interpreted as objects. So formally we can write:

$$[\![\texttt{TExpr}]\!] \;=\; \mu_{\mathbf{E}}$$

meaning that the datatype `TExpr` is interpreted as the carrier of the initial algebra of $\mathbf{E}$. The functions `feval` and `fmult` are interpreted as the pair of arrows

$$\mu_{\mathbf{E}} \xrightarrow{[\![\texttt{feval}]\!] \triangle [\![\texttt{fmult}]\!]} \mathbb{Z} \times {\mu_{\mathbf{E}}}^{\mathbb{Z}} \quad . \tag{6.27}$$

Now observe that both functions are structurally recursive, so they are formally interpreted as folds. As for `feval`, the $\mathbf{E}$-algebra,

$$\varepsilon : \mathbb{Z} + \mathbb{Z}^2 \longrightarrow \mathbb{Z} \ ,$$

corresponding to the meaning of the definition is the pair

$$\begin{aligned} \varepsilon_1 &=_{\text{def}} \mathbb{Z} \xrightarrow{1_{\mathbb{Z}}} \mathbb{Z} \\ \varepsilon_2 &=_{\text{def}} \mathbb{Z}^2 \xrightarrow{(+)} \mathbb{Z} \ . \end{aligned}$$

Altogether:

$$[\![\texttt{feval}]\!] \;=\; \mu_{\mathbf{E}} \xrightarrow{(\!|\, 1_{\mathbb{Z}} \triangledown (+) \,|\!)} \mathbb{Z} \quad . \tag{6.28}$$

As for `fmult`, first note that the constructors, `CVal`, `CAdd`, are interpreted as

$$\begin{aligned} [\![\texttt{CVal}]\!] \;&=\; \mathbb{Z} \xrightarrow{\iota_1} \mathbf{E}(\mu_{\mathbf{E}}) \xrightarrow{\mathsf{in}_{\mathbf{E}}} \mu_{\mathbf{E}} \\ [\![\texttt{CAdd}]\!] \;&=\; {\mu_{\mathbf{E}}}^2 \xrightarrow{\iota_2} \mathbf{E}(\mu_{\mathbf{E}}) \xrightarrow{\mathsf{in}_{\mathbf{E}}} \mu_{\mathbf{E}} \end{aligned}$$

And it is worth noting that

$$[\![\texttt{CVal}]\!] \triangledown [\![\texttt{CAdd}]\!] \;=\; \mathsf{in}_{\mathbf{E}} \quad . \tag{6.29}$$

The algebra, ψ, for `fmult` has type $\psi : \mathbb{Z} + (\mu_{\mathbf{E}}{}^{\mathbb{Z}})^2 \longrightarrow \mu_{\mathbf{E}}{}^{\mathbb{Z}}$. The left component is clearly simply

$$\psi_1 \;=_{\mathrm{def}}\; \mathbb{Z} \xrightarrow{(\times)^\wr} \mathbb{Z}^{\mathbb{Z}} \xrightarrow{[\![\texttt{CVal}]\!]^{\mathbb{Z}}} \mu_{\mathbf{E}}{}^{\mathbb{Z}} \;\;,$$

where $(-)^\wr$ is the currying isomorphism $\mathscr{C}(X \times Y, Z) \longrightarrow \mathscr{C}(X, Z^Y)$, for all X, Y, Z. The right component of ψ is

$$\psi_2 \;=_{\mathrm{def}}\; (\mu_{\mathbf{E}}{}^{\mathbb{Z}})^2 \xrightarrow{\tau_{\mu_{\mathbf{E}}}} (\mu_{\mathbf{E}}{}^2)^{\mathbb{Z}} \xrightarrow{[\![\texttt{CAdd}]\!]^{\mathbb{Z}}} \mu_{\mathbf{E}}{}^{\mathbb{Z}} \;\;,$$

where

$$\tau : ((-)^{\mathbb{Z}})^2 \Longrightarrow ((-)^2)^{\mathbb{Z}} \tag{6.30}$$

is the obvious natural isomorphism. Altogether, we have:

$$[\![\texttt{fmult}]\!] \;=\; \mu_{\mathbf{E}} \xrightarrow{(\!|\, [\![\texttt{CVal}]\!]^{\mathbb{Z}} \cdot (\times)^\wr \;\triangledown\; [\![\texttt{CAdd}]\!]^{\mathbb{Z}} \cdot \tau_{\mu_{\mathbf{E}}} \,|\!)} \mu_{\mathbf{E}}{}^{\mathbb{Z}} \tag{6.31}$$

Expressions, component-based

We now turn to component-based systems. An example is presented (see Fig. 6.4) and interpreted coalgebraically, which has the same intended meaning as the functional program in Fig. 6.3.

Let us first explain the contents of Fig. 6.4. It contains the definitions of two components, `IVal` and `IAdd`, in a hypothetical component-based language. Both components are declared to adhere to the interface `IExpr`, which is defined to accept two messages. One is an *attribute*, i.e. it doesn't update the local state, it is just a value. The type of the return value of *eval* is `Int`, the type of integers. The messages doesn't accept

```
interface IExpr
  attribute eval:() -> Int
  procedure mult:Int -> ()

component IVal implements IExpr
  state value:Int
  attribute eval = value
  procedure mult(x) { value := value * x; }

component IAdd implements IExpr
  state t,u: IExpr
  attribute eval = t.eval + u.eval
  procedure mult(x) { t.mult(x); u.mult(x); }
```

Figure 6.4: Implementation of `Expr`

any parameters. A second message defined in `IExpr` is `mult`. It is parameterised by `Int` and returns a value of the unit type, which is equivalent to returning no interesting value. But remember, as `mult` is a *procedure*, it can change the local state of the components it is sent to.

An implementation of a component consists of a specification of the type of the local state. It is `Int` in the case of `IVal` and a pair of pointers to components with interface `IExpr` in the case of `IAdd`.

Each component must also define an *implementation* for each attribute and method in the interface. As attributes are pure expressions, the symbol = is used to separate the expression on the right from its name on the left. For the component `IVal`, the expression defining the value of `eval` is just `value`, i.e. the variable holding the local state. For `IAdd`, the expression defining the value of `eval` is

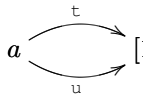
```
t.eval() + u.eval()
```

Here, the dot denotes message sending so the expression `t.eval()` denotes the return value of the message `eval` sent to the component pointed to by the state variable `t`. The two return values, integers, are added by $(+)$.

Implementations of methods are sequences of commands separated by `;`. The sequence itself is enclosed in braces $\{, \}$. For `IVal`, `mult` updates the local state by the operator `:=` to the value of the current local state multiplied by the value of the parameter `x`. The implementation of `mult` for `IAdd` contains two commands: one sending message `mult` with parameter `x` to the component pointed to by state variable `t`, the second one sending message `mult` with parameter `x` to the component pointed to by `u`. Note that this sequence doesn't change the local state of the component itself, but possibly of the components pointed to by `t` and `u`.

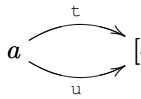
Coalgebraic semantics of `IExpr`

In order to interpret Fig. 6.4 coalgebraically, we impose the requirement that graph of components is a finite tree of components `IVal` and `IAdd`. This assumption is not expressed in the program. However, both acyclicity and lack of *sharing*, i.e. the fact that there are no two distinct paths in the tree of components with the same source and target, are clearly in line with the intended meaning of the program. Firstly, in the presence of cycles the processing of recursive messages wouldn't be terminating. If sharing was present, the meaning of `mult` as a function assigning expressions to expression would be rather weird. Just consider the following graph with one `IAdd` component, a, where both `t` and `u` are pointing to the same `IVal` component, v. Then in the initial state (6.32):



(6.32)

where the local state of v is in square brackets, the value of `eval` sent to a is 2. Sending message $(\texttt{mult}\ 2)$ to a results in the state:



(6.33)

because v is sent $(\texttt{mult}\ 2)$ twice, first through `t` and then through `u`. In this state, sending `eval` to a results in 8. Informally speaking, multiplying an expression with value 2 by 2 resulted in an expression with value 8. So clearly by prohibiting cycles and sharing in this program doesn't loose any desirable behaviour; finiteness is a natural requirement.

Next, the definitions of `IVal` and `IAdd` are interpreted as coalgebras. It has been already explained in what sense `IExpr` corresponds to the functor $\mathbf{G}$ defined in (6.24). So each of the components, `IVal`, `IAdd`, is interpreted as an $\mathbf{G}$-coalgebra.

For `IVal`, the local state has type `Int`, and we put

$$[\![\texttt{Int}]\!] =_{\text{def}} \mathbb{Z}$$

so we are looking for a coalgebra

$$[\![\texttt{IVal}]\!] : \mathbb{Z} \longrightarrow \mathbf{G}\mathbb{Z}$$

In general, as the local state of a component is implicitly defined to be a tuple, a reference to a local-state variable is interpreted as a projection. Formally, for a component

with state declaration

$$\texttt{state}\ x_1 : t_1, \ldots, x_n : t_n \qquad ,$$

where x_i are names of variables and t_i are names of types, a reference x_i is interpreted simply as

$$[\![x_i]\!] =_{\text{def}} \pi_i$$

Now, as $\mathbf{G}\mathbb{Z}$ is a product, $\mathbb{Z} \times \mathbb{Z}^{\mathbb{Z}}$, $[\![\texttt{IVal}]\!]$ splits into two arrows:

$$[\![\texttt{IVal}]\!]_{\texttt{eval}} =_{\text{def}} \mathbb{Z} \xrightarrow{1_{\mathbb{Z}}} \mathbb{Z}$$

$$[\![\texttt{IVal}]\!]_{\texttt{mult}} =_{\text{def}} \mathbb{Z} \xrightarrow{(\times)^{\imath}} \mathbb{Z}^{\mathbb{Z}}$$

and

$$[\![\texttt{IVal}]\!] =_{\text{def}} \mathbb{Z} \xrightarrow{[\![\texttt{IVal}]\!]_{\texttt{eval}} \vartriangle [\![\texttt{IVal}]\!]_{\texttt{mult}}} \mathbb{Z} \times \mathbb{Z}^{\mathbb{Z}} \tag{6.34}$$

As for `IAdd`, the local state should contain two pointers to other components. So how should pointers be represented? There are known approaches to programming and reasoning with pointers [] but here we take a different, more abstract approach. We observe, that as long as the graph of pointers is a tree and as long as there is no notion of global state, i.e. all state updates resulting from a message sent to a component x in the tree are localised within the subtree rooted at x, one can replace a pointer to x with the abstract component simulating x in its current state. So the state of the `IAdd` is formally ${\nu_{\mathbf{G}}}^2$.

As discussed in Sect. 6.3.5, sending any message $.m_i$ is interpreted as the i-th projection of the observation obtained by $\texttt{out}_{\mathbf{G}}$. In particular, the messages `eval` and `mult` are interpreted as follows:

$$[\![\texttt{.eval}]\!] =_{\text{def}} \nu_{\mathbf{G}} \xrightarrow{\texttt{out}_{\mathbf{G}}} \mathbf{G}\nu_{\mathbf{G}} \xrightarrow{\pi_1} \mathbb{Z}$$

$$[\![\texttt{.mult}]\!] =_{\text{def}} \nu_{\mathbf{G}} \xrightarrow{\texttt{out}_{\mathbf{G}}} \mathbf{G}\nu_{\mathbf{G}} \xrightarrow{\pi_2} {\nu_{\mathbf{G}}}^{\mathbb{Z}}$$

It clearly holds that

$$[\![\texttt{.eval}]\!] \vartriangle [\![\texttt{.mult}]\!] = \texttt{out}_{\mathbf{G}} \tag{6.35}$$

Lastly, from the assumption of no sharing, message sending to components pointed to by different state variables commutes. Formally, for state variables t, u, and arbitrary messages m, n:

$$[\![t.m(x); u.n(y)]\!] = [\![u.n(y); t.m(x)]\!] \quad \Leftarrow \quad t \neq u$$

It is therefore safe to interpret such a sequence of message sending commands in parallel.

Remark 6.3.1. More precisely, one can define for a pair of components u, v, with interfaces U, V, the component (u, v) with interface, $U \otimes V$, which has a message (m, n) for each pair of messages m in U and n in V, and such that sending the message (m, n) to (u, v) is equivalent to both $u.m\,; v.n$ and $v.n\,; u.m$. See [RBN06] for the author's Type-theoretic investigation of such constructions on state-based components.

Putting this all together, the interpretation of `IAdd` is a colagebra defined as follows:

$$\begin{aligned}
[\![\texttt{IAdd}]\!]_{\texttt{eval}} &=_{\text{def}} {\nu_{\mathbf{G}}}^2 \xrightarrow{[\![\texttt{.eval}]\!]^2} \mathbb{Z}^2 \xrightarrow{(+)} \mathbb{Z} \\
[\![\texttt{IAdd}]\!]_{\texttt{mult}} &=_{\text{def}} {\nu_{\mathbf{G}}}^2 \xrightarrow{[\![\texttt{.mult}]\!]^2} ({\nu_{\mathbf{G}}}^{\mathbb{Z}})^2 \xrightarrow{\tau_{\nu_{\mathbf{G}}}} ({\nu_{\mathbf{G}}}^2)^{\mathbb{Z}} \\
[\![\texttt{IAdd}]\!] &=_{\text{def}} {\nu_{\mathbf{G}}}^2 \xrightarrow{(+)\cdot[\![\texttt{.eval}]\!]^2 \vartriangle \tau_{\nu_{\mathbf{G}}} \cdot [\![\texttt{.mult}]\!]^2} \mathbb{Z} \times ({\nu_{\mathbf{G}}}^2)^{\mathbb{Z}} \quad , \qquad (6.36)
\end{aligned}$$

where τ is as in (6.30).

We have defined coalgebras (6.34), (6.36) interpreting the program in Fig. 6.4. In order to define an interpretation of a system of components, we introduce some notation: $(x, \texttt{IVal})$ denotes a component `IVal` with state x, and $((t, u), \texttt{IAdd})$ denotes a component `IAdd` pointing to components t and u. Now define interpretation of components as:

$$[\![(x, \texttt{IVal})]\!] =_{\text{def}} (\!|\, [\![\texttt{IVal}]\!] \,|\!)(x) \quad : \nu_{\mathbf{G}} \qquad (6.37)$$

$$[\![((t, u), \texttt{IAdd})]\!] =_{\text{def}} (\!|\, [\![\texttt{IAdd}]\!] \,|\!)([\![t]\!], [\![u]\!]) \quad : \nu_{\mathbf{G}} \qquad (6.38)$$

Abstracting on the state of components, x in (6.37), (t, u) in (6.38), one obtains the functions

$$\mathbb{Z} \xrightarrow{[\![(-, \texttt{IVal})]\!]} \nu_{\mathbf{G}}$$

$${\nu_{\mathbf{G}}}^2 \xrightarrow{[\![(-, \texttt{IAdd})]\!]} \nu_{\mathbf{G}}$$

As the graph of components is a tree, one can observe that a configuration of a system is exactly a term of type $\mu_{\mathbf{E}}$ and the definition of interpretation of components, (6.37), (6.38), is structurally recursive and therefore defines the semantic function

$$\Phi =_{\text{def}} \mu_{\mathbf{E}} \xrightarrow{(\!|\, [\![(-, \texttt{IVal})]\!] \triangledown [\![(-, \texttt{IAdd})]\!] \,|\!)} \nu_{\mathbf{G}} \qquad (6.39)$$

In summary, a system of components whose graph of components is a tree is interpreted as the behaviour of the component in the root of the tree, which is obtained inductively.

Comparing the functional and component-based programs

First, observe that (6.27) is in fact a $\mathbf{G}$-coalgebra so there is the arrow

$$\Psi \ =_{\mathrm{def}} \ \mu_{\mathbf{E}} \xrightarrow{\ (\!|\, [\![\texttt{feval}]\!] \vartriangle [\![\texttt{fmult}]\!] \,|\!)\ } \nu_{\mathbf{G}} \quad , \tag{6.40}$$

which assigns to a term its $\mathbf{G}$-behaviour obtained by iteratively applying `feval` and `fmult` to intermediate results. We can now hope to be able to establish

$$\Psi \ = \ \Phi \quad : \mu_{\mathbf{E}} \longrightarrow \nu_{\mathbf{G}} \tag{6.41}$$

This doesn't seem so hopeless as one can observe that Ψ and Φ are defined in terms of exactly the same components. Namely, after unfolding all definitions in (6.41) one obtains:

$$\begin{aligned}
\Psi \ &= \ [\!(\, (\!|\, 1_{\mathbb{Z}} \triangledown (+) \,|\!) \vartriangle (\!|\, \mathsf{in}_{\mathbf{E}}{}^{\mathbb{Z}} \cdot \iota_1{}^{\mathbb{Z}} \cdot (\times)^{\wr} \triangledown \mathsf{in}_{\mathbf{E}}{}^{\mathbb{Z}} \cdot \iota_2{}^{\mathbb{Z}} \cdot \tau_{\mu_{\mathbf{E}}} \,|\!) \,)\!] \\
\Phi \ &= \ (\!|\, [\!(\, 1_{\mathbb{Z}} \vartriangle (\times)^{\wr} \,)\!] \triangledown [\!(\, (+) \cdot \pi_1{}^2 \cdot \mathsf{out}_{\mathbf{G}}{}^2 \vartriangle \tau_{\nu_{\mathbf{G}}} \cdot \pi_2{}^2 \cdot \mathsf{out}_{\mathbf{G}}{}^2 \,)\!] \,|\!)
\end{aligned}$$

Then it follows from basic properties of products and coproducts that for natural transformations $\kappa, \kappa' : \mathbf{EG} \longrightarrow \mathbf{GE}$, defined as follows:

$$\kappa \ =_{\mathrm{def}} \ (1_{\mathbb{Z}} \triangledown (+)) \vartriangle (\iota_1{}^{\mathbb{Z}} \cdot (\times)^{\wr} \triangledown \iota_2{}^{\mathbb{Z}} \cdot \tau) \tag{6.42}$$

$$\kappa' \ =_{\mathrm{def}} \ (1_{\mathbb{Z}} \vartriangle (\times)^{\wr}) \triangledown ((+) \cdot \pi_1{}^2 \vartriangle \tau \cdot \pi_2{}^2) \tag{6.43}$$

it holds that

$$\begin{aligned}
\Psi \ &= \ [\!(\, (\!|\, \mathbf{G}(\mathsf{in}_{\mathbf{E}}) \cdot \kappa_{\mu_{\mathbf{E}}} \,|\!) \,)\!] \\
\Phi \ &= \ (\!|\, [\!(\, \kappa'_{\nu_{\mathbf{G}}} \cdot \mathbf{E}(\mathsf{out}_{\mathbf{G}}) \,)\!] \,|\!)
\end{aligned}$$

Finally, it is straightforward to show that

$$\kappa \ = \ \kappa' \tag{6.44}$$

and it therefore remains to establish

$$[\!(\, (\!|\, \mathbf{G}(\mathsf{in}_{\mathbf{E}}) \cdot \kappa_{\mu_{\mathbf{E}}} \,|\!) \,)\!] \ = \ (\!|\, [\!(\, \kappa_{\nu_{\mathbf{G}}} \cdot \mathbf{E}(\mathsf{out}_{\mathbf{G}}) \,)\!] \,|\!) \quad : \mu_{\mathbf{E}} \longrightarrow \nu_{\mathbf{G}} \quad . \tag{6.45}$$

But (6.45) is known (see e.g.[TP97]).

6.3.7 Adequacy, distributively

In this section, the introductory example of `TExpr` and `IExpr` is recast in terms of distributive laws of endofunctors. This formulation then forms the basis of a generalisation in terms of generalised distributive laws.

Expressions and distributive laws of endofunctors

First, note that all non-generic components that occur in 6.42 are

$$1_{\mathbb{Z}} \;:\; \mathbb{Z} \longrightarrow \mathbb{Z} \tag{6.46}$$

$$(+) \;:\; \mathbb{Z}^2 \longrightarrow \mathbb{Z} \tag{6.47}$$

$$(\times)^{\imath} \;:\; \mathbb{Z} \longrightarrow \mathbb{Z}^{\mathbb{Z}} \tag{6.48}$$

$$\tau \;:\; ((-)^{\mathbb{Z}})^2 \Longrightarrow ((-)^2)^{\mathbb{Z}} \quad . \tag{6.49}$$

These are all the following distributive laws of endofunctors, where most of the functors are constant.

$$\underline{\mathbb{Z}}\,|\,\frac{\frac{\mathbb{Z}}{}}{\frac{1_{\mathbb{Z}}}{\mathbb{Z}}}\,|\,\underline{\mathbb{Z}}$$

$$\underline{\mathbb{Z}}\,|\,\frac{\frac{(-)^2}{}}{\frac{(+)}{(-)^2}}\,|\,\underline{\mathbb{Z}}$$

$$(-)^{\mathbb{Z}}\,|\,\frac{\frac{\mathbb{Z}}{}}{\frac{(\times)^{\imath}}{\mathbb{Z}}}\,|\,(-)^{\mathbb{Z}}$$

$$(-)^{\mathbb{Z}}\,|\,\frac{\frac{(-)^2}{}}{\frac{\tau}{(-)^2}}\,|\,(-)^{\mathbb{Z}}$$

Then observe that (6.42) is equivalently

$$\underline{\mathbb{Z}} \times (-)^{\mathbb{Z}}\,|\,\frac{\frac{\underline{\mathbb{Z}}+(-)^2}{}}{\frac{(1_{\mathbb{Z}} \boxplus (+)) \boxtimes ((\times)^{\imath} \boxplus \tau)}{\underline{\mathbb{Z}}+(-)^2}}\,|\,\underline{\mathbb{Z}} \times (-)^{\mathbb{Z}}$$

and (6.43) is equivalently

$$\underline{\mathbb{Z}} \times (-)^{\mathbb{Z}}\,|\,\frac{\frac{\underline{\mathbb{Z}}+(-)^2}{}}{\frac{(1_{\mathbb{Z}} \boxtimes (\times)^{\imath}) \boxplus ((+) \boxtimes \tau)}{\underline{\mathbb{Z}}+(-)^2}}\,|\,\underline{\mathbb{Z}} \times (-)^{\mathbb{Z}}$$

for the product and coproduct of distributive laws, $\boxtimes$, $\boxplus$. Equation (6.44) then reduces to

$$(\alpha \boxplus \beta) \boxtimes (\gamma \boxplus \delta) \;=\; (\alpha \boxtimes \gamma) \boxplus (\beta \boxtimes \delta) \tag{6.50}$$

for distributive laws, α, β, γ, δ, of endofunctors on the same category. This is proven by so called *abides rule* for product and coproduct [MFP91], namely it holds that:

$$(k \triangledown l) \vartriangle (m \triangledown n) \;=\; (k \vartriangle m) \triangledown (l \vartriangle n) \;\; .$$

Bialgebras

The proof of (6.45) can be carried out quite straightforwardly by a rudimentary calculation involving properties of products, coproducts and, importantly, fusion. On the other hand, [TP97] shows a more abstract proof based on the notion of *bialgebra*, which was introduced in Sect. 4.4.2 in terms of generalised distributive laws.

Recall from Sect. 4.4.2 that a when λ is a distributive law of $\mathscr{T}$ over $\mathscr{U}$ in $\mathscr{K}$, a λ-bialgebra φ is an arrow $\underline{1} \longrightarrow \lambda$ in $[\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]]$ or equivalently in $[\![\mathscr{U}, [\mathscr{T}, \mathscr{K}]]\!]$. As $\mathscr{T}$ and $\mathscr{U}$ are theories of functors with structure there exist the forgetful 2-functors

$$[\mathscr{T}, [\![\mathbf{o}, \mathscr{K}]\!]] \;:\; [\mathscr{T}, [\![\mathscr{U}, \mathscr{K}]\!]] \longrightarrow [\mathscr{T}, \mathscr{K}] \tag{6.51}$$

$$[\![\mathscr{U}, [\mathbf{o}, \mathscr{K}]]\!] \;:\; [\![\mathscr{U}, [\mathscr{T}, \mathscr{K}]]\!] \longrightarrow [\![\mathscr{U}, \mathscr{K}]\!] \tag{6.52}$$

which forget either the coalgebra or algebra part, respectively, of a bialgebra. In the cases of Ex. 4.4.10, i.e. for endofunctors, monads and comonads, the following is important and known.

Theorem 6.3.2. *Consider the components, ordinary functors:* $[\mathsf{M}, [\![\mathbf{o}, \mathsf{Cat}]\!]](\underline{1}, \kappa)$ *and* $[\![\mathsf{M}^{\mathsf{co}}, [\mathbf{o}, \mathsf{Cat}]]\!](\underline{1}, \kappa)$. *These functors have a right and left adjoint, respectively.*

Proof. See [TP97], Theorems 7.2 and 7.3. □

Corollary 6.3.3. *For a* $(\mathsf{M}^{\mathsf{co}}, \mathbf{D})|\overline{\underset{(\mathsf{M},\mathbf{T})}{\overset{(\mathsf{M},\mathbf{T})}{\kappa}}}|(\mathsf{M}^{\mathsf{co}}, \mathbf{D})$ *in* Cat*, the category* κ*-bialgebras in* Cat*, formally* $[\![\mathsf{M}^{\mathsf{co}}, [\mathsf{M}, \mathsf{Cat}]]\!](1, \kappa)$ *has an initial and a terminal object.*

Proof. Because right adjoints preserve limits and left adjoints preserve colimits, the terminal κ-bialgebra is obtained by the action of the right adjoint on the trivial terminal $\mathbf{T}$-algebra, $\mathbf{T}1 \longrightarrow 1$. Likewise, the initial κ-bialgebra is obtained by the action of the left adjoint on the trivial initial $\mathbf{D}$-coalgebra, $0 \longrightarrow \mathbf{D}0$. □

The initial κ-bialgebra given by Corollary 6.3.3 is

$$\mathbf{T}^2 0 \xrightarrow{\mu_0} \mathbf{T}0 \xrightarrow{\mathbf{T}!_{\mathbf{D}0}} \mathbf{T}\mathbf{D}0 \xrightarrow{\kappa_0} \mathbf{D}\mathbf{T}0 \tag{6.53}$$

The terminal κ-bialgebra is

$$\mathbf{TD}1 \xrightarrow{\kappa_1} \mathbf{DT}1 \xrightarrow{\mathbf{D}!_{\mathbf{T}1}} \mathbf{D}1 \xrightarrow{\delta_1} \mathbf{D}^2 1 \tag{6.54}$$

It follows that there is a unique morphism from (6.53) to (6.54), whose underlying arrow has type

$$\mathbf{T}0 \longrightarrow \mathbf{D}1 \tag{6.55}$$

When everything is considered up to the isomorphism between (co)algebras of an endofunctor and (co)algebras of its (co)free (co)monad,

$$\mathbf{F}\text{-}\mathbf{Alg} \;\cong\; \mathbf{F}^*\text{-}\mathbf{Alg} \qquad\qquad \mathbf{G}\text{-}\mathbf{CoAlg} \;\cong\; \mathbf{G}_\infty\text{-}\mathbf{CoAlg} \tag{6.56}$$

one obtains (6.45). We interpret this as the following result about functional and component-based programs.

Theorem 6.3.4. *Consider fixed polynomial functors $\mathbf{F}$, $\mathbf{G}$; a functional program P thought of as implementing an abstract datatype with interface $\mathbf{G}$ by the tuple of functions $[\![P]\!] : \mu_{\mathbf{F}} \longrightarrow \mathbf{G}\mu_{\mathbf{F}}$; a component-based program, R, whose admissible graph of components has shape $\mu_{\mathbf{F}}$ and where all components have an interface corresponding to $\mathbf{G}$, formally $[\![G]\!] : \mathbf{F}\nu_{\mathbf{G}} \longrightarrow \nu_{\mathbf{G}}$. Then if there is a distributive law $\mathbf{G}_\infty | \overset{\mathbf{F}^*}{\underset{\mathbf{F}^*}{\overline{\underline{\kappa}}}} | \mathbf{G}_\infty$ such that*

$$[\![P]\!] = (\!|\, \mathbf{G}(\mathsf{in}_{\mathbf{F}^*}) \cdot \kappa\mu_{\mathbf{F}} \,|\!)$$

and

$$[\![R]\!] = [\!(\, \kappa\nu_{\mathbf{G}} \cdot \mathbf{F}\mathsf{out}_{\mathbf{G}} \,)\!]$$

where $[\![\,]\!]$ has the meaning described above, then P and R are observationally indistinguishable when applied to the same element of $\mu_{\mathbf{F}}$ serving as the initial value of the abstract datatype, and a description of the graph of components, respectively.

For our case of comparison of functional and component-based programming, it is important to note that something has been gained by the generalisation from functors to their free monads and comonads. Because, although (6.56) holds, it is not the case that distributive laws of endofunctors are isomorphic to distributive laws of the free monad over the cofree comonad. Clearly, one can lift every distributive law of endofunctors to a distributive law of a monad over a comonad by iteration and coiteration. But one can do more: Turi and Plotking in [TP97] show that natural transformations

$$\mathbf{F}(\mathbf{1} \times \mathbf{G}) \Longrightarrow \mathbf{G}\mathbf{F}^* \tag{6.57}$$

give rise to distributive laws $\mathbf{G}_\infty | \frac{\mathbf{F}^*}{\mathbf{F}^*} | \mathbf{G}_\infty$. So do natural transformations

$$\mathbf{F}\mathbf{G}_\infty \Longrightarrow \mathbf{G}(\mathbf{1} + \mathbf{F}) \tag{6.58}$$

To see how (6.57) and (6.58) correspond to programs, first recall that a polynomial $\mathbf{F}$ describes a the signature of a datatype, or equivalently the graph of components. A polynomial $\mathbf{G}$ stands for behaviour, equivalently interface. And it was shown in the introduction how one can think of a natural transformation $\mathbf{FG} \Longrightarrow \mathbf{GF}$ as of a program operating by case analysis, always matching on one constructor and recursing to the subterms; equivalently, delegating a message to the immediate subcomponents. In this light, (6.57) is taking results for subterms, possibly ignoring them, to results with new subterms, which can be deeper than one layer. In the functional case, this allows definitions like

```
mult x (Add t u) =  Add t u
```

or

```
mult x (Add t u) = Add (Add t t) (Add u u)    .
```

In the component-based case, the first corresponds to

```
method mult(v) { }    ,
```

i.e. not passing a message to child components and

```
method mult(v) { t := new Add(t,t); u := new Add(u,u); }   ,
```

i.e. construction of new components. However note that the type $\mathbf{F}^*$ includes the pure variable term, with no constructors from $\mathbf{F}$ in it. This has no counterpart in current mainstream object-oriented programming, nor in the simplified component-based programming as we have described it. The same applies for functions changing the head constructor of a clause, e.g.:

```
mult x (Add t u) = Val x    ,
```

which are perfectly legal in functional programming. In component-based programming this would correspond to a change of the type of the component that currently processes the message. This is allowed in some languages, but not in the mainstream class-based ones.

On the other hand, (6.58) provides for iterative recursive invocation, such as

```
mult x (Add t u) = Add (mult x (mult x t)) (mult x (mult x u))
```

and omitting a constructor in the result

```
mult x (Add t u) = t
```

in the functional case. In the component-based reading, this corresponds to

```
method mult(x) { t.mult(x); t.mult(x); u.mult(x); u.mult(x); }
```

The second case, i.e. omitting a constructor has already been discussed.

So clearly, moving to monads has enlarged the class of programs on either side we are able to formally relate by distributive laws.

Remark 6.3.5. The generalisation used here to conceptually unify thy functorial and monadic case poses the intriguing question whether similar adequacy results exist for other notions of distributive law than of a monad over a comonad.

6.3.8 Algebra of distributive laws and programming

Theorem 6.3.4 effectively identifies distributive laws of free monads over cofree comonads with a certain subclass of programs which can be interpreted both a as functional and component based programs. It follows that the operations on distributive laws described in Chapter 5 can be used as operations on programs. Some examples were presented in the previous section. In the rest of this section we outline the interpretations of the formal constructions in the respective programming paradigms.

Combining structure

> *The coproduct $\boxplus$ of distributive laws corresponds to combination of structure.*

Functionally, programs p on $\mu_{\mathbf{E}}$ and q on $\mu_{\mathbf{E}'}$ give a program $p \boxplus q$ on $\mu_{\mathbf{E}+\mathbf{E}'}$. For instance, say we wanted to extend expressions `TExpr` with constructor

```
data TExpr = ... | CMult TExpr TExpr
```

for representing multiplicative expressions. Consequently, the functions `feval` and `fmult` must be extended with cases for the new constructor, `CMult`.

```
    ...
feval (CMult t u) = (feval t) * (feval u)


    ...
fmult x (CMult t u) = CMult (fmult x t) u
```

This bit of code corresponds to a distributive law $\kappa \equiv \mathbb{Z} \times (-)^{\mathbb{Z}} | \frac{(-)^2}{(\star)\boxtimes\gamma} | \mathbb{Z} \times (-)^{\mathbb{Z}}$, where γ could be defined as $\lambda\, f\, g\, x\,.\ (fx, g1)$. This isn't however precisely what is written in the code, but rather

```
fmult x (CMult t u) = CMult (fmult x t) (fmult 1 u)  .
```

However, the type of γ, namely $((-)^{\mathbb{Z}})^2 \Longrightarrow ((-)^2)^{\mathbb{Z}}$ or even the whole κ, forces us to invoke the recursive call, because of naturality. This is where the monadic generalisation comes in handy, because the form (6.57) allows us reuse the previous subterm and to define just a γ' of type

$$((-) \times (-)^{\mathbb{Z}})^2 \Longrightarrow ((-)^2)^{\mathbb{Z}}$$

as

$$\gamma' =_{\mathrm{def}} \lambda\,(t, f)(u, g)\,.\,\lambda\,x\,.\,(fx, u)$$

which can be lifted to a distributive law of the free monad of $(-)^2$ over the cofree comonad of $(-)^{\mathbb{Z}}$. Such a distributive law can be then combined by $\boxplus$ with the lifted κ, because the sum of free monads is the free monad of sum. The complete program is given by a distributive law of a free monad over cofree comonad.

$$(\mathbb{Z} \times (-)^{\mathbb{Z}})_\infty | \frac{\overset{(\mathbb{Z}+(-)^2+(-)^2)^*}{\tilde{\kappa} \boxplus \tilde{\kappa}'}}{(\mathbb{Z}+(-)^2+(-)^2)^*} | (\mathbb{Z} \times (-)^{\mathbb{Z}})_\infty \ ,$$

where $\tilde{\alpha}$ denotes the iteration of a distributive law $\mathbf{G}| \frac{\overset{\mathbf{F}}{\alpha}}{\mathbf{F}} |\mathbf{G}$ to a distributive law $\mathbf{G}_\infty | \frac{\overset{\mathbf{F}^*}{\tilde{\alpha}}}{\mathbf{F}^*} |\mathbf{G}_\infty$. Note that in terms of changes to source code, changes to the datatype is an expensive operation in FP as cases for the new constructors of the datatype must be added to each function in p.

In the component-based case, $p \boxplus q$ corresponds to putting two sets of definitions of components together to form one large system. In the above example, κ corresponds to the definition of a new component

```
component CMult implements IExpr
  state t,u: IExpr
  attribute eval = t.eval * u.eval
  procedure mult(x) = { t.mult(x); }
```

In terms of changes to source code this is simple, as the two programs are just put together.

Combination of behaviour

> *The product $\boxtimes$ of distributive laws corresponds to combination of behaviour.*

Functionally, when p implements a collection of functions $\mathbf{G}$ and q implements a collection of functions $\mathbf{G}'$, $p \boxtimes q$ implements both $\mathbf{G}$ and $\mathbf{G}'$. For instance, say we wanted to implement in addition the function `print` producing a character string in `Str`.

This can be given as a distributive law

$$\mathtt{Str}\Big|\frac{\mathbb{Z}+(-)^2+(-)^2}{\rho}\Big|\mathtt{Str}$$

i.e. a natural transformation (an algebra indeed) of type $\mathbb{Z} + \mathtt{Str}^2 + \mathtt{Str}^2 \Longrightarrow \mathtt{Str}$. The details are obvious. Now one readily obtains the distributive law

$$(\mathbb{Z} \times (-)^{\mathbb{Z}} \times \mathtt{Str})_\infty \Big|\frac{(\mathbb{Z}+(-)^2+(-)^2)^*}{(\tilde{\kappa}\boxplus\tilde{\kappa}')\boxtimes\tilde{\rho}}\Big|(\mathbb{Z} \times (-)^{\mathbb{Z}} \times \mathtt{Str})_\infty$$

In terms of source code, the functional case is simple as the definition of `print` is just added to the others. In the component-based case, the old implementations of components must be merged with the new implementations.

This dichotomy has been described by Cook [Coo91] as so-called *expression problem*. In the case of distributive laws, there is no difference.

Note that a rudimentary description of products, coproducts, and also composition of components in the Type-theoretical setting has been carried out by the author in [RBN06].

Chapter 7

Conclusion

7.1 Conclusions

7.1.1 Summary

The intent of this thesis was to contribute to the understanding of certain parametrically polymorphic programs, which, when understood categorically as natural transformations, resemble categorical distributive laws in that they essentially swap the order of two functors. We have analysed the situation and defined a generalised distributive law essentially as a 2-functor

$$\lambda \;:\; \mathscr{T} \otimes \mathscr{U} \longrightarrow \mathscr{K} \tag{7.1}$$

where $\otimes$ is Gray's lax tensor product of 2-categories and $\mathscr{T}$ and $\mathscr{U}$ are 2-categories standing for theories of the functors with structure involved in the distributive law. In this situation it is said that λ is a distributive law of a $\mathscr{T}$-functor over a $\mathscr{U}$-functor. Our requirement on the theories, $\mathscr{T}$, $\mathscr{U}$, was only that a theory has a chosen arrow, which is understood as the underlying arrow of the functor with structure. The characteristic functor-swapping natural transformation arises from this definition as the 2-cell given by (2.41) of Def. 2.4.3 for the chosen arrows. Much of the rest of the presented results about distributive laws, namely the coherence with both functors with structure and the ability to treat a distributive law both as a model of $\mathscr{T}$ and as a model of $\mathscr{U}$ follows from fundamental properties of 2-categories which hold in full generality. We therefore claim that the characterisation (7.1) is *a fundamental characterisation of distributive-law-like programs in mathematically structured programming*. It hasn't been known before in such a refined form and by introducing it we have clarified the previously vague understanding of such programs.

The clarification and demonstration of the connection of distributive-law-

like programs and Gray's lax tensor product of 2-categories is the main contribution of this thesis.

In Chapter 5, we investigated constructions of products, coproducts and composition of distributive laws in this general setting and showed how the usual specific constructions arise as instances of the generic case. This alleviates the burden of proving coherence conditions in many cases.

In Chapter 6, we outlined how our generalised definition matches the previously defined notions and in Sect. 6.3 we show how distributive laws of "structure over behaviour" connect inductive functional programming and coinductive component-based programming.

7.1.2 Critical Evaluation

The following is a summary of our contributions and non-contributions.

1. In Chapter 3, we defined *functors with structure* (Def. 3.2.1) as 2-functors into an arbitrary base 2-category from a 2-category with a chosen arrow called a *theory*.

2. In Sect. 3.2.1, we recalled that it follows from general results about 2-categories generated by a 2-sketch, that a model in $\mathscr{K}$ of a theory generated by a 2-sketch is equivalent to a model of the sketch in $\mathscr{K}$, i.e. that it is valid to define such a model by enumeration of the values of the model on the components of the sketch. Similarly for morphisms. This results in a notion of a functor with structure which is very similar to functors with structure in the informal sense, as one can essentially turn a categorical definition of a functor with structure into a 2-sketch to obtain a formal functor with structure. In this context we considered finite-product 2-sketches, i.e. computads with equations on 1- and 2-cells, and products on 0-cells.

3. In Chapter 4, we defined *generalised distributive laws* (Def. 4.1.1) parameterised by a pair of *theories* of functors with structure essentially as 2-functors from the Gray's lax tensor product of the two theories to a base 2-category.

4. This definition is characterised by Theorem 4.1.3, according to which a distributive law is equivalently a model of $\mathscr{T}$ in the forward category of models of $\mathscr{U}$, and a model of $\mathscr{U}$ in the reverse category of models of $\mathscr{T}$. This corresponds to the general intuition and other results about distributive laws (see also Sect. 7.2).

5. We further defined an n-ary version of the definition of a distributive law (Def. 4.3.3) which has an iterated characterisation as a model in an iterated category of models. It also comes with a strong characterisation theorem (Theorem 4.3.7)

which allows one to construct an arbitrary-dimensional distributive law from 2-dimensional distributive laws which pairwise satisfy only a 3-dimensional coherence condition (Fig. 4.4). This is a useful tool for the construction of models and distributive laws whose theories arise from simpler theories by Gray's tensor product.

In these results, our contribution lies in the identification and collecting of the relevant results and their presentation in an accessible geometrical style and thus illustrating the connection of distributive-law-like programs and Gray's tensor product of 2-categories. This arises from the connection of Beck's distributive laws of monads and Gray's tensor product, which is obvious as soon as the iterative case is known. Namely, from the fact that $\mathsf{M} \longrightarrow [\mathsf{M}, \mathscr{K}]$ is a distributive law of monads, the transition to $\mathsf{M} \otimes \mathsf{M} \longrightarrow \mathscr{K}$ has been proven by Gray in full generality. It is therefore straightforward to generalise M to an arbitrary 2-category $\mathscr{T}$ to arrive at our notion of a generalised distributive law. The rest of our results, the characterisation theorems and the theory of higher dimensional distributive laws are just Gray's results from [Gra74].

In Chapter 5 we consider some common constructions on generalised distributive laws:

6. In Chapter 5, we describe a general approach to products, coproducts (by duality) and composition of distributive laws based on the iterative characterisation of distributive laws as models of functors with structure in a category of models of a functor with structure.

7. To this end we prove a general product lifting theorem (Theorem 5.1.9) showing that products on objects of the base 2-category can be lifted to products on objects in the functor category.

8. In Sect. 5.3, we make use of the fact that certain theories of functors with structure are defined as Gray's product of one or more simpler theories where one of them is 2, to define their composition.

The material in this chapter is our own. Our contribution lies in the development of the constructions in the chosen lax approach, which gives rise to constructions which are very close to the definitions found in programming. In fact, we demonstrate in examples that they are literally the same modulo the usual encoding and conventions that take place when translating 2-categorical concepts to 1-categorical ones [1]. This serves our purpose of linking programming and mathematics.

On a more general level, Chapter 5 illustrates that our lax 2-categorical approach to distributive laws is viable. It is our belief that other interesting constructions on distributive laws could be introduced to programming by interpreting other results about lax natural transformations such as lax adjoints or Kan extensions.

[1] e.g. the use of pointwise definitions instead of the point-free 2-categorical style

Finally, in Chapter 6, we apply the developed theory to examples in programming as follows:

9. We illustrate how Meertens's functor pullers and McBride and Patterson's `iFunctors` arise as instances of the same scheme of a distributive law of a polynomial functor over an arbitrary functor with structure.

10. In Sect. 6.3, we establish a semantical equivalence of inductive functional and coinductive component-based programs that arise in a canonical way from a distributive law.

Although in (9) we haven't completely subsumed the mentioned notions because we haven't considered regular datatypes and shape, we have shown enough for the rest to be straightforward. Namely, one has to verify that fixed points of endofunctors lift from $\mathscr{K}$ to $[\![\mathscr{T}, \mathscr{K}]\!]$, for an arbitrary $\mathscr{K}$, similar to the lifting of products in Theorem 5.1.9. Preservation of shape will then follow from coherence.

Finally, although turning to [HB97] at an informal level we haven't formally explained the allegorical approach thereof. It seems plausible that such an explanation would involve a 2-category of allegories, relators and their natural transformations. We provide a more thorough comparison in Sect. 7.2.

7.2 Related Work

Distributive Laws in Datatype-generic Programming The direct predecessors of our work have already been mentioned in the introduction. Here we offer a more thorough analysis.

Hoogendijk and Backhouse in [HB97, Hoo97] analyse the situation of *commuting relators*, so-called zips, in the allegorical setting. As the work is carried out in a different formal setting, we offer just an informal comparison of the requirements on zips in [HB97], Section 4, with Def. 2.4.3 of a quasi-functor of two variables. One can see that if Alg is a suitable 2-categorical universe of allegories, $\mathscr{F}_n$ is a theory of n-ary regular datatypes, $\mathbf{F} : \mathscr{F}_n \longrightarrow \mathsf{Alg}$ and $\mathbf{G} : \mathscr{F}_m \longrightarrow \mathsf{Alg}$ are two allegorical datatypes with underlying arrows $f : \mathscr{C}^n \longrightarrow \mathscr{C}$ and $g : \mathscr{C}^m \longrightarrow \mathscr{C}$, respectively, then what is called, modulo notation, $\mathsf{zip}\,(f, g)$ in [HB97] is essentially a 2-functor $\mathbf{H} : \mathscr{F}_n \otimes \mathscr{F}_m \longrightarrow \mathsf{Alg}$ such that $\mathbf{H}(\mathbf{o}, -) = \mathbf{G}, \mathbf{H}(-, \mathbf{o}) = \mathbf{F}$.

- The "proper natural transformation indexed by an $l * k$ matrix of types" given by a $\mathsf{zip}\,(f, g)$ corresponds the underlying 2-cell $\mathbf{H}(\mathbf{a}, \mathbf{a})$.

- The comments regarding arbitrary arity corresponds to the requirement that $\mathscr{F}_k$ be a single sorted finite product 2-theory with an underlying arrow of type $\mathbf{o}^k \longrightarrow \mathbf{o}$.

- The requirement of preservation of projections up to a transposition, τ, correspond to $\mathbf{F}$, $\mathbf{G}$ being strongly finite-product preserving, i.e. the product preservation is only up to the isomorphism $\tau : (X^n)^m \cong (X^m)^n$.

- Preservation of shape is coherence with the structure of $\mathscr{F}_k$

- The requirement that $\mathsf{zip}\,(f, g)$ and $\mathsf{zip}\,(f, h)$ be coherent with any transformation $\alpha : g \Longrightarrow h$, i.e. equation (37) in [HB97]:

$$\alpha f \cdot \mathsf{zip}\,(f, h) \;=\; \mathsf{zip}\,(f, g) \circ f\alpha \tag{7.2}$$

  is similar to our Equation (2.38). Equation (2.39) follows from a symmetrical requirement for the first argument of zip.

- It is moreover required in [HB97], (38), that (7.2) be relaxed to an inclusion of relations. This requirement follows from the nondeterminism and partiality in the relational setting. This could be captured by not considering a 2-categorical, i.e. a Cat-enriched setting, but a $\mathsf{Cat}_{\subseteq}$-enriched setting.

- Section 4.2 in [HB97] spells out coherence of $\mathbf{H}$ with composition and units, which can be readily seen to be our items (1)-(3) in Def. 2.4.3.

In summary, we conjecture that allegorical zips can be made precise in our setting (up to nondeterminacy and partiality) as 2-functors out of Gray's lax tensor of theories of regular datatypes into a suitable 2-category of allegories.

Meertens's *functor pullers* [Mee98, Mee96] can be seen as a categorical version of so-called half-zips in [HB97], which are like zips with half of the symmetrical requirements on a quasi functor dropped.

The theme of a traversal of a datatype, $\mathbf{F}$, by a functor, $\mathbf{T}$, standing for the collected result or *effect* later appears for different functors of effects. For instance, in [MBJ98] natural transformations of type $\mathbf{F}\mathbf{T}^m \Longrightarrow \mathbf{T}\mathbf{F}$ are used for a monad $\mathbf{T}$. These must be coherent with the structure of the monad and preserving the shape of $\mathbf{F}$. Later, McBride and Patterson observe in [MP08], that one doesn't need a monad to define a traversal but that an applicative functor is enough for many applications such as parsing or crushing (evaluation). The same kind of distributive-law-like polymorphic program is associated in [GdSO09] with the semantics of traversals in object-oriented programming, so-called *iterators*.

In all these cases, the additional structure is fixed in the case of the other functor, $\mathbf{T}$, and there is a notion of a datatype, $\mathbf{F}$, and its shape, which must be preserved. We generalise this situation to a pair of functors with structure and natural transformations which must be coherent with both of them.

Distributive laws à la Beck Once the connection of such distributive-law-like programs and Beck's distributive laws of monads [Bec69] is made, together with the observation that the single notion alone won't suffice, one is naturally interested in what has been done in terms of its generalisation.

In [PW99], Power and Watanabe give an overview of the notions of distributive laws of monads and comonads and study their relations. In [LPW00], the spectrum of derived distributive laws is extended to the weaker distributive laws of pointed endofunctors and plain endofunctors. In both cases, Street's iterative characterisation of Beck's monads in [Str72] is used to define categories of such distributive laws. This is used, among other things, to systematically enumerate and study the relations of the mentioned notions of distributive laws. Our 2-functorial characterisation of these categories improves on theirs in exhibiting the general nature of these and similar comparisons and therefore some of the specific characterisation theorems proven in [LPW00] become just instances of one general characterisation theorem, Theorem 4.1.3. See also Remark 4.1.4. We believe this is helpful as it separates the specifics from the generics.

Recently Cheng in [Che07] studies iterated distributive laws of monads. Again, Street's characterisation theorem is used and it is remarked that a monad in $\mathscr{K}$ is a 2-functor from a suitable 2-category, Δ, and that $\Delta \longrightarrow [\Delta, \mathscr{K}] \cong \Delta \otimes \Delta \longrightarrow \mathscr{K}$, in our notation. The main theorem in the paper is a characterisation of iterated distributive laws of monads, which becomes an instance of the Higher-dimensional Characterisation Theorem 4.3.7 in this thesis for the specific case of Δ. See also Ex. 4.3.9.

As for constructions on distributive laws, Manes and Mulry in [MM07] give a comprehensive overview for monads. Our treatment covers some of the constructions exhibited there in a greater generality while eliminating the proofs of coherence carried out explicitly. On the other hand, their collection is more complete.

Lawvere Theories, etc. The transition form Street's 2-category $\mathsf{Mnd}(\mathscr{K})$ of monads in $\mathscr{K}$ to $[\mathsf{M}, \mathscr{K}]$ is of crucial importance because it allows one to make the step to $[\mathscr{T}, \mathscr{K}]$ and then to $[\mathscr{U}, [\mathscr{T}, \mathscr{K}]] \cong [\mathscr{T} \otimes \mathscr{U}, \mathscr{K}]$ for all theories $\mathscr{T}$, $\mathscr{U}$. This completes the connection of certain parametrically polymorphic programs on datatypes, through Beck's distributive laws of monads to categorical model theory. This, we believe, brings the programs to their true foundations and opens up a chasm of further possibilities.

The leading example in this area of research is the research programme of Hyland, Plotkin and Power into the semantics of computational effects [PP02, HPP06, HLPP07, PP08]. From the starting point of Moggi's *notions of computations* as monads, the authors argue e.g. in [PP02] that monads in the context of computational effects are a notion derived from *operations* (such as `read` and `write` for computations with input and output) while the operations should be considered primitive. One then specifies

operations in a suitable notion of a *sketch*, from which arises a *theory* with a corresponding notion of a *model*. The notion of theory, the central semantical notion used in this line of work is that of a *countably enriched Lawvere theory* [Pow99], and the corresponding notion of a model is a finite cotensor preserving V-functor. When V is Cat, one obtains a countable Lawvere 2-theory, for which there is a notion of a sketch, which allows one to specify notions such as monoidal categories or categories with a monad, considered also in this thesis. The corresponding notion of a model considered there is however that of a *pseudo* V-functor and *pseudo* V-natural transformation, as opposed to our lax notions.

7.3 Future Work

Functors with additional structure In the thesis we studied a certain notion of a combination of functors with structure which is at the same time a coherent morphism of both (all) functors with structure involved. We avoided the question of the "correct" notion of the additional structure of functors simply because the range of examples presented here indicates that there is no such single notion which would be at the same time general and strong enough. In this thesis we simply use different specific approaches to different specific examples.

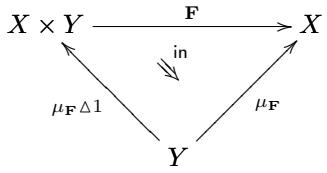
In order to address this shortcoming it seems that we would need to consider functors with structure parameterised by a notion of additional structure such as *linear*, *finite-product*, *with-constants*, etc. One can then expect that more complex additional structures of functors involved in a distributive law will arise from simpler additional structures, among other things by distributive laws of a simpler kind. Such a situation was encountered here in the example of applicative functors.

There seems to be a myriad of further structure when one takes further the simple idea presented in this thesis. We believe it deserves to be studied in order to develop deep theoretical understanding and useful mathematics of coherence properties, which present in their more complicated forms a major hurdle in areas such as higher-dimensional category theory.

In [KPT99] a general framework for sketches in the enriched biclosed monoidal setting is described, which seems to be a good starting point for such an investigation.

Regular distributive laws An obvious shortcoming of this work on the practical front is that we haven't described distributive laws of datatypes. Section 5 illustrates a general approach to definitions of combinators on distributive laws – by lifting combinators on functors with structure. We considered products, coproducts and composition, but not fixed points. This is certainly desirable in order to define distributive laws of all regular datatypes over functors with structure. It would involve a lifting of

collections of initial diagrams



,

in a $\mathscr{K}$, which represent regular functors for a polynomial $\mathbf{F}$, to diagrams in $[\![\mathscr{T}, \mathscr{K}]\!]$.

Transpositions – shape preservation After we have defined generalised distributive laws of all regular datatypes over endofunctors, it shouldn't be difficult to show that they preserve the shape of the datatype. However, a problem arises in defining distributive laws of datatypes over datatypes in the total setting as in order for a *function* like

```
zip :: ([x],[y]) -> [(x,y)]
```

to be well defined, the two lists must have the same length. Similarly for

```
zipTree :: (Tree x, Tree y) -> Tree (x,y)
transpose :: [[x]] -> [[x]]
```

where the two trees must have the same shape and all lists in the lists of lists must have the same length. Otherwise the functions are ill-defined[2].

More formally, let $\mathcal{S}_{\mathbf{T}}$ denote the *set of shapes* of datatype $\mathbf{T}$, and let $\mathbf{T}_s$ denote the subtype of $\mathbf{T}$ of elements of shape s. Then for any $s \in \mathcal{S}_{\mathtt{Tree}}$, $m, n \in \mathbb{N} = \mathcal{S}_{[\,]}$, there are distributive laws

$$\begin{aligned} \mathtt{zip}_s &: (\times) \circ \mathtt{Tree}_s^2 \Longrightarrow \mathtt{Tree}_s \circ (\times)^2 \\ \mathtt{transpose}_{m,n} &: [[-]_n]_m \Longrightarrow [[-]_m]_n \end{aligned}$$

This collection is moreover natural in the shapes. It would be desirable to make such shape constraints a part of the type and express formally that the above distributive laws are a family natural in the shape, rather than ad hoc instances. In order to carry this out in the total setting one would have to turn to dependent types, where one can be more precise about the types and express such shape constraints.

Abbott, Altenkirch and Ghani develop in [AAG03] an approach to datatypes, which is fundamentally built on a notion of shape. Datatypes are formalised as *containers* of data, which have *shape* (a type of positions) and an assignment of a datum to each position. Categorically, datatypes are indexed by a category of shapes, $\mathcal{S}$, i.e. they are represented as functors $\mathcal{S} \longrightarrow *$, where $*$ is a category of types. It would be interesting to see distributive laws in this setting.

[2]The way this is handled in practice is by truncating the arguments to the same shape.

Dependently typed distributive laws There is no doubt that distributive laws appear in programming with dependent types. How does the theory apply to that setting, or what is needed to make it work?

An example why it might be useful was given above when we indexed types by shapes but one could go further and consider the general dependently typed setting. For instance, this could describe situations when shape is not only preserved but also changed in a systematic way.

Distributive proofs No less interesting would be to consider distributive laws of proofs under the Curry-Howard isomorphism. A distributive law of functors with structure can be seen as an underlying natural transformation together with witnesses of proofs of coherence conditions. Composition and other operations on distributive laws construct proofs of coherence for the results from the proofs for the arguments. In our case, these proofs were just simple equational ones. This could be taken further.

For instance, consider the function

$$\mathbf{T}_s(X)^2 \xrightarrow{\mathsf{zip}} \mathbf{T}_s(X^2) \xrightarrow{\mathsf{eq}_X} \mathbf{T}_s(\mathbb{B}) \xrightarrow{\mathsf{and}} \mathbb{B} \ , \tag{7.3}$$

where eq_X is boolean equality on X and and is the lifting of $\mathsf{and} : \mathbb{B}^2 \longrightarrow \mathbb{B}$ to $\mathbf{T}$'s. The function (7.3) is a boolean equality of trees which assigns true to a pair of trees with equal elements. How can this be defined with propositional equality, and in general for other inductive/coinductive structures and general propositions? For instance, it could be that inductive proofs of proposition $\mathbf{P}$ about an inductive type $\mathbf{T}$ are defined in terms of a distributive law of $\mathbf{T}$ over $\mathbf{P}$.

Nomenclature

2-Cateogories

Distributive Laws

Theories

Algebras, Coalgebras

References

[AAG03] Michael Abbott, Thorsten Altenkirch and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003.

[AR94] Jiří Adámek and Jiří Rosický. *Locally Presentable and Accessible Categories*. Cambridge University Press, 1994.

[Awo06] Steve Awodey. *Category Theory*. Clarendon Press, 2006.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall PTR, September 1996.

[Bec69] Jon Beck. Distributive laws. *Lecture Notes in Mathematics*, 80:119–140, 1969.

[Bén67] Jean Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77. Springer-Verlag, Berlin, 1967.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag, 1988. NATO ASI Series F Volume 55. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring and Lambert Meertens. Generic programming. An introduction. In Jos'e N. Oliveira S.D. Swierstra, Pedro R. Henriques, editor, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998*, volume 1608 of *LNCS*, pages 28–115. Springer Verlag, 1999.

[BT06] Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. In *21st Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 309–329. Elsevier, 2006.

[BW85] Michael Barr and Charles Wells. *Toposes, Triples and Theories*, volume 278 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, New York, 1985.

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science*. CRM, 3rd edition, 1999.

[Che07] Eugenia Cheng. Iterated distributive laws, 2007.

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178. Springer-Verlag, 1991.

[CS91] J. Robin B. Cockett and D. Spencer. Strong categorical datatypes i, 1991.

[Ehr68] Charles Ehresmann. Esquisses et types des structures algébriques. *Bul. Inst. Polit. Iaşi*, XIV, 1968.

[GdSO09] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19:377–402, July 2009.

[GHJV94] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gib06] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In Ralf Hinze, editor, *Workshop on Generic Programming*, September 2006.

[Gib07] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[Gir72] Jean-Yves Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur*. Thèse d'état, Université Paris 7, June 1972.

[Gra74] John W. Gray. *Formal category theory : adjointness for 2-categories*, volume 391 of *Lecture Notes in Mathematics*. Springer-Verlag, New York, 1974.

[GTW78] Joseph A. Goguen, James Thatcher and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[GU71] Peter Gabriel and Friedrich Ulmer. *Lokal Präsentierbare Kategorien*, volume 221 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1971.

[HB97] Paul F. Hoogendijk and Roland Backhouse. When do datatypes commute? In *CTCS '97: Proceedings of the 7th International Conference on Category Theory and Computer Science*, pages 242–260, London, UK, 1997. Springer-Verlag.

[HJL06] Ralf Hinze, Johan Jeuring and Andres Löh. Comparing approaches to generic programming in haskell. Technical Report UU-CS-2006-022, Department of Information and Computing Sciences, Utrecht University, 2006.

[HLPP07] Martin Hyland, Paul Blain Levy, Gordon Plotkin and John Power. Combining algebraic effects with continuations. *Theor. Comput. Sci.*, 375(1-3):20–40, 2007.

[Hoo97] Paul F. Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[HPP06] Martin Hyland, Gordon Plotkin and John Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, 357(1):70–99, 2006.

[Jac96] Bart Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer and H. J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.

[JJ97] Patrik Jansson and Johan Jeuring. Polyp - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[Kel82] G. M. Kelly. *Basic concepts of enriched category theory*. Number 64 in London Mathematical Society Lecture Note Series. Cambridge Univ. Press, 1982.

[Koc72] Anders Kock. Strong functors and monoidal monads. *Arch. Math*, 23:113–120, 1972.

[KPT99] Yoshiki Kinoshita, John Power and Makoto Takeyama. Sketches. *Journal of Pure and Applied Algebra*, 143:275–291, 1999.

[KS74] G. M. Kelly and Ross Street. *Review of the elements of 2-categories*, volume 420 of *Lecture Notes in Mathematics*. Springer Berlin / Heidelberg, 1974.

[Law66] F. William Lawvere. The category of categories as a foundation for mathematics. In S. et al. Eilenberg, editor, *Proceedings of the Conference on Categorical Algebra*, pages 1–20, La Jolla, 1966. Springer-Verlag: Berlin, Heidelberg and New York.

[Law68] F. William Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories*. PhD thesis, University of Colombia, 1963,1968.

[LPW00] Marina Lenisa, John Power and Hiroshi Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *ENTCS*, 33, 2000.

[LR08] Ralf Lämmel and Ondrej Rypacek. The expression lemma. In *Proceedings of Mathematics of Program Construction (MPC) 2008*, LNCS. springer, July 2008.

[LS88] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic (Cambridge Studies in Advanced Mathematics)*. Cambridge University Press, March 1988.

[Mac97] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. springer, New York, 2nd edition, 1997. (1st ed., 1971).

[Mal90] Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Department of Computing Science, Groningen University, 1990.

[MBJ98] Eugenio Moggi, Gianna Bellè and Barry Jay. Monads, shapely functors and traversals. Technical Report DISI-TR-98-06, Università di Genova, 1998. abstract appeared in WGP'98.

[Mee96] Lambert Meertens. Calculate polytypically! In *PLILP'96, volume 1140 of LNCS*, pages 1–16. Springer-Verlag, 1996.

[Mee98] Lambert Meertens. Functor pulling. In *Workshop on Generic Programming (WGP98*, 1998.

[MFP91] Erik Meijer, Maarten M. Fokkinga and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[MM07] Ernie Manes and Philip Mulry. Monad compositions i: General constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MP90] Michael Makkai and Robert Paré. *Accessible Categories: the Foundations of Categorical Model Theory*. Number 104 in Contemporary Mathematics. American Mathematical Society, 1990.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[Pow90] John Power. A 2-categorical pasting theorem. *Journal of Algebra*, 129:439–445, 1990.

[Pow99] John Power. Enriched lawvere theories. *Theory and Applications of Categories*, 6(7):83–93, 1999.

[PP02] Gordon Plotkin and John Power. Computational effects and operations: An overview, 2002.

[PP08] Gordon Plotkin and John Power. Tensors of comodels and models for operational semantics. *Electron. Notes Theor. Comput. Sci.*, 218:295–311, 2008.

[PS04] J. Power and O. Shkaravska. From comodels to coalgebras: State and arrays. In *CMCS'2004: 7th Intl. Workshop on Coalgebraic Methods in Computer Science*, volume 106, Barcelona, Spain, March 2004. Elsevier, ENTCS.

[PW92] John Power and Charles Wells. A formalism for the specification of essentially algebraic structures in 2-categories. *Mathematical Structures in Computer Science*, 2:1–28, 1992.

[PW99] John Power and Hiroshi Watanabe. Distributivity for a monad and a comonad. *Electr. Notes Theor. Comput. Sci.*, 19, 1999.

[RBN06] Ondrej Rypacek, Roland Backhouse and Henrik Nilsson. Type-theoretic design patterns. In *ACM SIGSOFT Workshop on Generic Programming 2006*. ACM Press, 2006.

[Rei95] Horst Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Paris, France*, pages 513–523. Elsevier, 1983.

[RP90] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic lambda calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 127–152. Addison-Wesley, Reading, MA, 1990.

[Rut00] Jan Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[Str72] Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 1972.

[Str76] Ross Street. Limits indexed by category-valued 2-functors. *J. Pure Appl. Algebra*, 8:149–181, 1976.

[Swe69] M. Sweedler. *Hopf Algebras*. Mathematics Lecture Note Series. W. A. Benjamin, Inc., New York, 1969.

[TP97] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings 12th Annual IEEE Symposium on Logic in Computer Science, LICS'97, Warsaw, Poland, 29 June – 2 July 1997*, pages 280–291. IEEE Press, 1997.

[Wel93] Charles Wells. Sketches: Outline with references. 1993.