Karlsruhe Institute of Technology

Modeling Uncertain Data using Monads and an Application to the Sequence Alignment Problem

Bachelor Thesis of

Alexander Kuhnle

At the Department of Informatics
Institute for Program Structures
and Data Organisation (IPD)

Reviewer: Prof. Dr.-Ing. Gregor Snelting
Advisors: Dr. Olaf Klinke (DKFZ Heidelberg)
Dipl.-Math. Dipl.-Inform. Joachim Breitner

Duration: June 5, 2013 – September 5, 2013

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practise.

Karlsruhe, September 2, 2013

....
(Alexander Kuhnle)

Abstract

Bioinformatics applies algorithms of text processing to the problem of DNA/protein sequencing to great success. But in many places one has to cope with uncertain data. Most of the contemporary techniques either ignore this fact or have trouble to get a grip on all the occuring uncertainties. This work presents an approach to model uncertain data and calculate sequence alignments based on this model.

We show that monads pose a general way to describe various kinds of uncertainty. Using the framework of monads with minor additional structure we introduce the concept of sequences with inherent uncertainty. Finally, we transfer a suffix tree construction algorithm to the new setting of uncertainty to provide an efficient way of calculating alignments. This algorithm uses the previously developed general interfaces of uncertainty monads and uncertain sequences and hence is able to cover a wide range of situations with different kinds of uncertainty, including the former case of ignoring any uncertainty.

Contents

1 Preface **3**
1.1 Motivation . . . 3
1.2 Outline . . . 5
1.3 Preliminary Remarks . . . 7

2 The Concept of Monads **8**
2.1 Monads in Category Theory . . . 8
2.1.1 Functors & Natural Transformations . . . 8
2.1.2 Monads & Kleisli Triples . . . 9
2.1.3 Monads with Null . . . 11
2.1.4 Additive & Measurable Functors, Functors with Inverses . . . 12
2.2 Monads in Computer Science . . . 14
2.2.1 The Use of Monads . . . 14
2.2.2 Implementation in Haskell . . . 15
2.3 Composing Monads . . . 17
2.3.1 The Problem of Monad Composition . . . 17
2.3.2 Distributive Laws . . . 18

3 Monads and Uncertainty **20**
3.1 A Theory of Uncertainty . . . 20
3.1.1 Terminology . . . 21
3.1.2 Uncertainty Monads . . . 22
3.1.3 Equality of Uncertain Values . . . 23
3.2 Examples of Uncertainty Monads . . . 25
3.2.1 A Remark on Indeterminism and Composability . . . 27
3.3 Properties . . . 29

4 Uncertain Sequences **31**
4.1 Definition of Uncertain Sequences . . . 31
4.2 Implementation in Haskell . . . 34
4.2.1 The Sequence Type Class . . . 34
4.2.2 Sequence Instances . . . 35
4.3 Working with Uncertain Sequences . . . 37
4.3.1 Predicates & Functions . . . 37
4.3.2 Properties . . . 37
4.3.3 An Example . . . 39

5 Suffix Trees to Uncertain Sequences **40**
5.1 About Suffix Trees 40
5.2 The Suffix Tree Construction Algorithm 41
5.2.1 Determining Suffixes 41
5.2.2 An Intermediate Version of a Suffix Tree 42
5.2.3 The Final Suffix Tree 43
5.2.4 An Example, revisited 47
5.3 Further Remarks 48
5.3.1 Correctness via QuickCheck 48
5.3.2 Space Consumption 49

6 Related Work **53**
6.1 The Haskell Classes 54

7 Conclusion **55**

8 Appendix **57**

1 Preface

1.1 Motivation

Before we go into detail about the content of this work we depict various problems and challenges which led to the considerations presented here. These come from biology and are, more precisely, related to DNA/protein sequencing. Bioinformatics is a relatively new field of science which tries to solve problems computationally by applying well-known text-processing algorithms to them. Our work is primarily a contribution to bioinformatics.

Reverse Transcription

Proteins are chains of amino acids which are encoded in the DNA. For higher organisms there are 21 amino acids. In DNA these are represented as triplets (codons) of the four different nucleic acids (including a start and a stop triplet). Hence potentially $4^3 = 64$ amino acids could be encoded. Instead, for many amino acids there are several representations as codons. When trying to reverse translate a protein, i.e. a sequence of amino acids, to the corresponding sequence of nucleic acids in the DNA, one should take these ambiguities into account. The result of such a reverse transcription actually is a sequence not having definite codons but a collection of possible codons for each position.

Polymorphisms

Every human possesses a unique DNA. This is because small mutations distinguish the genome of individual humans, although most of these mutations do not have a significant effect (see also the last example). There seem to be sites where mutations occur more frequently. Furthermore, this process seems to be influenced by factors like gender or the region of origin. There are attempts to cataloguize such polymorphisms in the healthy population[1]. Thus, nowadays one has both a single human reference genome and a database of well-known deviations from this reference. Sequence alignment algorithms usually first calculate the best possible alignments to this reference and afterwards examine the leftover differences using the known polymorphisms. It would be preferable to

[1] For instance, see www.1000genomes.org.

merge this two-step process into one by using a suitable structure to represent the reference with inherent polymorphisms, if of course the runtime and required space remain reasonable.

Splicing

In the construction of a protein a gene is first copied from a chromosome, resulting in a new sequence of nucleic acids called messenger RNA. Afterwards, this sequence is edited. There are regions of the gene which contain the codons specifying the protein (exons) and hence are not modified, and there are regions which are possibly cut out (introns). This process is called splicing. It is sometimes used to gain a wider variety of similar proteins from one single gene by exploiting combinatorial complexity[2]. A similar mechanism can also be found in human immune cells at the level of chromosomal DNA or, more precisely, in the formation of mature immune cells out of precursor immune cells. To get correct alignments, one hence needs to allow gaps to take care of possibly excised parts. This obviously involves higher computational costs. Once all the splicing variants are known, one could instead include the information in a reference sequence which at the end of an exon would be able to give all possible successor sequences taking potentially missing introns into account.

Repetitive Elements

There are some sequences which occur more frequently in the genome. These repetitions may serve as spacers, and an organism can change the number of repititions quite easily. However, in the construction of a reference sequence one has to decide on this number. On the one hand short sequencing reads possibly result in many alignments where usually one single reported occurence is required. On the other hand the comparison of sequences might result in a great evolutionary distance[3] due to repetitive regions although these seemingly large differences arise from relatively small changes of the number of repeats. Thus a sequence dynamically allowing to vary the repetition of such regions would be favourable.

[2] Mainly, the reason for splice variants is unknown, though.
[3] At least if the evolutionary distance is measured using a simple model, e.g. the Levenshtein distance.

1.2 Outline

All the examples presented in 1.1 have in common that one wants to compute alignments to sequences which include some kind of uncertainty. We saw sequences for which the actual values cannot be given precisely, some sequences have positions for which the successor sequence is not definite, while for other sequences the expected values can be given more exactly using specific properties of the treated sequence (e.g. gender of the individual). These examples constitute the motivation to deal with sequences and alignments in the context of uncertainty.

In this work we present an approach that tries to model uncertain sequences with the help of monads. We develop a framework to represent uncertain data using monads and generalize the concept of a sequence on the basis of this framework. Finally we exemplarily transfer a classic alignment algorithm using suffix trees to the new concept of uncertain sequences. To give an outline of what one can expect of the following sections, we want to illustrate the essential considerations that underlie this work.

A sequence is an ordered collection of elements and we assume sequences to be finite. So every item either has a successor or it is the last item of the sequence. We also call a sequence a "string". Such a sequence is not interfered by any real uncertainty, i.e. analyzing a predicate on this sequence results in a definite positive resp. negative answer. The trivial uncertainty hence can be expressed by using the logical values *true* and *false*.

When calculating an alignment of a string to a reference sequence we basically determine whether the former string is a substring of the reference. There might be multiple positions where the string occurs in the reference as a substring. The idea of a suffix tree is to give an efficient method to calculate alignments. It is achieved by first computing all suffixes of the reference and then representing them in the suffix tree. Thus the suffix tree starts at every position in the reference simultaneously and the alignment is reduced to determining whether the string is a prefix of the suffix tree.

This short outline reveals some aspects that ought to be covered in the following sections:

- At first the matter of uncertainty needs clarification: how can uncertainty be modeled and what properties have to be satisfied? In our approach we describe uncertainty by the use of monads. The categorical concept of a monad and some additional structures are presented in section 2, while their interpretation in the context of uncertainty are the subject of section 3.
- The rough idea of some properties of uncertainty can already be discerned (and are also presented in sections 2 and 3). They should at least be able to express the *true/false* uncertainty of the logical values. While the *true* value can be seen in correspondence to the unit operator of a monad, we introduce the concept of a

monad with a null operator to also have a counterpart to the *false* value. Furthermore, we expect a way to measure uncertainty, e.g. to say to which uncertainty a string is substring of an uncertain sequence, and we require to have a way to aggregate uncertainty, e.g. in the case of multiple occurences of a substring. These observations motivate the structures of measurable resp. additive functors. The idea of combined uncertainties via composing monads arises from the fact that a suffix tree can again be seen as an uncertain sequence. More precisely, a suffix tree can be seen as a kind of *meta-sequence* which takes a sequence and adds the uncertainty of potentially starting at any position within the sequence.

- Another subject is the concept of an uncertain sequence, which is discussed in section 4. It is not entirely clear how a sequence can be properly generalized to a sequence with uncertainty. Moreover, to work with such sequences we need functions which allow us to do at least the basic queries to get a kind of *head value* resp. *tail sequence*. To actually talk about alignments one also needs to reasonably define the terms of a *substring*, *prefix* and *suffix* regarding this new concept of uncertain sequences.

- In section 5 we finally use all the structures developed in the former sections to transfer the idea of alignment using a suffix tree to uncertain sequences. The result is a suffix tree construction algorithm for sequences of an arbitrary kind of uncertainty which fulfill a few minimal requirements.

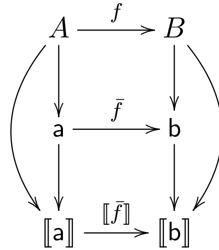
While being concerned with these issues, it is useful and also necessary to look back upon the classic sequences and logical values every now and then, as they should always be a trivial special case of our attempt of generalization.

1.3 Preliminary Remarks

In section 2 we introduce structures of category theory, which are fundamental for our approach to model uncertainty. It will become apparent that in all of our considerations we aim to talk only about a special kind of categories. As some of the structures require such a peculiar category, we want to restrict our view of a category from the beginning. Some of the definitions, e.g. a monad, can of course also be given in a more general context, but for this work the restricted view is sufficient.

The categories we are interested in are subcategories of the category of sets, i.e. their objects are sets and their morphisms are mappings between the sets. Moreover we assume a specific singleton object $\mathbb{1}$ with the only element $1 \in \mathbb{1}$ to be an object of these categories. We further suppose that for every object A we have a morphism $\tau_A \colon A \to \mathbb{1}$ explicitly given.

Since we want to use the categorical concepts to implement a suffix tree construction algorithm for uncertain sequences, we should be able to give an implementation of the categorical structures. For this we assume that objects A, B can be properly embedded in types a, b, and that a morphism $f \colon A \to B$ between the objects has a corresponding function $\bar{f} :: \mathsf{a} \to \mathsf{b}$ between values of the types. By proper we mean that our semantical interpretation of elements of the categorical objects is consistent with the interpretation of the corresponding values of the types. This can be summarized by the following diagram:



This assumption justifies that on the one hand we do not go into detail concerning the precise implementation of the categorical concepts presented in section 2, while on the other hand in later sections for some functions we only give their implementation.

We use the functional programming language Haskell, but other functional languages are equally applicable. So on the level of types we have the category of Haskell types $\mathcal{H}$. At this we stick to the Haskell-typic notation of types, functions and values. The 0-tuple () is taken to be the type corresponding to the object $\mathbb{1}$, the value () :: () is seen as the element $1 \in \mathbb{1}$ and the functions const () :: $\mathsf{a} \to$ () correlate to the morphisms $\tau_A \colon A \to \mathbb{1}$.

Our implementation is based on the Glasgow Haskell Compiler (GHC) in version 7.6.3. We use the following extensions: ExistentialQuantification, FlexibleContexts, FlexibleInstances, FunctionalDependencies, MultiParamTypeClasses, OverlappingInstances, ScopedTypeVariables, UndecidableInstances. Due to name conflicts the module Prelude has to be explicitly imported via import Prelude hiding (Monad, (≫=)). If one also uses the module Control.Monad, it has to be imported via import Control.Monad hiding (Monad, join, (≫=), mfilter).

2 The Concept of Monads

In the following we give a brief introduction to the concept of monads both in category theory and in computer science. We start with the categorical definition of a monad and the equivalent structure of a Kleisli triple. At the same time we are going to introduce additional properties monads resp. functors may have, which will be important for later sections. We proceed by giving an outline of the occurence and use of monads in computer science. Subsequently, our implementation of monads in the Haskell programming language is presented. Finally, the matter of monad composition will be discussed and we will present a feasible combination technique.

2.1 Monads in Category Theory

The origin of monads lies in category theory, an abstract and conceptual branch of mathematics. We will not go into details and instead assume that the reader is familiar with basic definitions and notations of category theory. An introduction can be found in [1]. For the sake of completeness we start with the definition of functors and natural transformations.

2.1.1 Functors & Natural Transformations

Definition 1 (Functor)
Given two categories $\mathcal{C}$ and $\mathcal{D}$, a ***functor*** $\mathrm{F}\colon \mathcal{C} \to \mathcal{D}$ *from $\mathcal{C}$ to $\mathcal{D}$ is a mapping which for each object A in $\mathrm{Obj}_{\mathcal{C}}$ gives an object $\mathrm{F}A$ in $\mathrm{Obj}_{\mathcal{D}}$ and associates to each morphism $f\colon A \to B$ in $\mathrm{Mor}_{\mathcal{C}}$ a morphism $\mathrm{F}f\colon \mathrm{F}A \to \mathrm{F}B$ in $\mathrm{Mor}_{\mathcal{D}}$. The functor preserves the identity morphism and morphism composition, i.e. the following two conditions hold for any $f\colon A \to B, g\colon B \to C$:*

- (F1) : $\mathrm{F}\,\mathrm{id}_A = \mathrm{id}_{\mathrm{F}A}$,
- (F2) : $\mathrm{F}(g \circ f) = \mathrm{F}g \circ \mathrm{F}f$.

Definition 2 (Natural Transformation)
Given two functors F and G from categories $\mathcal{C}$ to $\mathcal{D}$, a ***natural transformation*** $\alpha\colon \mathrm{F} \to \mathrm{G}$ *from F to G is a mapping which for each object A in $\mathrm{Obj}_{\mathcal{C}}$ gives a morphism $\alpha_A\colon \mathrm{F}A \to \mathrm{G}A$ such that for any morphism $f\colon A \to B$ in $\mathrm{Mor}_{\mathcal{C}}$ the following diagram commutes[1]:*

$$\begin{array}{ccc} \mathrm{F}A & \xrightarrow{\mathrm{F}f} & \mathrm{F}B \\ {\scriptstyle \alpha_A}\downarrow & (\cdot) & \downarrow{\scriptstyle \alpha_B} \\ \mathrm{G}A & \xrightarrow[\mathrm{G}f]{} & \mathrm{G}B \end{array}$$

[1]We will give the abbreviation $(\cdot)$ representing the condition stated by the diagram in the respective definition where a natural transformation is introduced.

2.1.2 Monads & Kleisli Triples

Before we formally define the term of a monad, we try to explain the idea of a monad by giving two examples[2].

(1) Given a set A we can always form the powerset $\mathcal{P}(A)$ over A. For any mapping $f\colon A \to B$ we are able to define a corresponding mapping $\mathcal{P}(f)\colon \mathcal{P}(A) \to \mathcal{P}(B)$ by applying f to the elements of a set $X \in \mathcal{P}(A)$. Additionally, A can be embedded in $\mathcal{P}(A)$ by assigning to each element $x \in A$ the singleton $\{x\} \in \mathcal{P}(A)$. Finally, a set of sets $X' \in \mathcal{P}(\mathcal{P}(A))$ can be joined to a set $X \in \mathcal{P}(A)$ via $X := \bigcup X'$. Note that embedded singletons are trivially joined, $\bigcup\{X\} = X$ and $\bigcup\big\{\,\{x\} \mid x \in X\big\} = X$ for $X \in \mathcal{P}(A)$, and that the order of joining does not matter, $\bigcup\big(\bigcup X\big) = \bigcup\big\{\bigcup x \mid x \in X\big\}$ for $X \in \mathcal{P}(\mathcal{P}(\mathcal{P}(A)))$.

(2) For every set A we can form the set of R-annotated values $R \times A$ over A with annotation set R. We can transfer any mapping $f\colon A \to B$ to a mapping of annotated values via $f \mapsto \big((r,x) \mapsto (r, f(x))\big)$. If $(R, \cdot^R, 1^R)$ is a monoid, then A can be embedded in $R \times A$ via $x \mapsto (1^R, x)$, and a twice annotated element $(r, (r', x)) \in R \times (R \times A)$ can be joined to a single annotated element $(r \cdot^R r', x) \in R \times A$. The properties of a monoid, i.e. associativity and neutral element, are the reason why embedded elements trivially join and the order of joining is irrelevant.

Although these examples are quite different at the first sight, they both reveal the same basic structure. They form a new set $\mathrm{M}A$ over a given set A, the elements of the former set can be embedded in the new one, $A \to \mathrm{M}A$, mappings $f\colon A \to B$ can be transferred, $\mathrm{M}f\colon \mathrm{M}A \to \mathrm{M}B$, and if the former set already was of the form $\mathrm{M}A$, it can be joined, $\mathrm{MM}A \to \mathrm{M}A$. The joining operation is also called multiplication. A monoidal structure becomes apparent in the sense that the joining operator is associative and the embeddings join trivially. In the definition of a monad the illustrated relations such structures inhere are formalized.

Definition 3 (Monad)
*A **monad** on a category $\mathcal{C}$ is a triple (M, η, μ) consisting of*

- *an endofunctor $\mathrm{M}\colon \mathcal{C} \to \mathcal{C}$,*
- *a (M1) natural transformation $\eta\colon \mathrm{id}_{\mathcal{C}} \to \mathrm{M}$ called **unit** operator and*
- *a (M2) natural transformation $\mu\colon \mathrm{MM} \to \mathrm{M}$ called **join** operator*

such that the following diagrams commute:

$$\begin{array}{ccccc} \mathrm{M}A & \xrightarrow{\eta_{\mathrm{M}A}} & \mathrm{M}^2A & \xleftarrow{\mathrm{M}\eta_A} & \mathrm{M}A \\ {\scriptstyle \mathrm{id}_{\mathrm{M}A}} & \text{(M3)} & \downarrow{\scriptstyle \mu_A} & \text{(M4)} & {\scriptstyle \mathrm{id}_{\mathrm{M}A}} \\ & \longrightarrow & \mathrm{M}A & \longleftarrow & \end{array} \qquad \begin{array}{ccc} \mathrm{M}^3A & \xrightarrow{\mu_{\mathrm{M}A}} & \mathrm{M}^2A \\ {\scriptstyle \mathrm{M}\mu_A}\downarrow & \text{(M5)} & \downarrow{\scriptstyle \mu_A} \\ \mathrm{M}^2A & \xrightarrow[\mu_A]{} & \mathrm{M}A \end{array}$$

An alternative way to describe the same structure is by using Kleisli triples. The difference lies in the intuition behind this term. Instead of the join operator one now considers a lifting operator which allows to lift a mapping $A \to \mathrm{M}B$ to a corresponding mapping $\mathrm{M}A \to \mathrm{M}B$. For our examples the lifting should be straightforward as we already saw that for $f\colon A \to \mathrm{M}B$ we

[2] It is to note that there are other ways to motivate the concept of a monad. So the examples concurrently give a hint of our understanding of a monad.

can get $\mathrm{M}f\colon \mathrm{M}A \to \mathrm{MM}B$ and via joining $\mathrm{M}A \to \mathrm{M}B$. This simultaneously shows the idea why indeed the two concepts are equivalent.

Definition 4 (Kleisli triple)
*A **Kleisli triple** on a category $\mathcal{C}$ is a triple $(\mathrm{M}, \eta, \text{-}^*)$ consisting of*

- *a mapping $\mathrm{M}\colon \mathrm{Obj}_\mathcal{C} \to \mathrm{Obj}_\mathcal{C}$,*
- *a morphism $\eta_A\colon A \to \mathrm{M}A$ for each A in $\mathrm{Obj}_\mathcal{C}$ and*
- *a **lift** operator -^* which for each morphism $f\colon A \to \mathrm{M}B$ gives a morphism $f^*\colon \mathrm{M}A \to \mathrm{M}B$. f^* is also called **(Kleisli) lifting of** f.*

The lift operator has to satisfy the following three conditions for any $f\colon A \to B, g\colon B \to C$:

- (K1) : $\eta_A^* = \mathrm{id}_{\mathrm{M}A}$,
- (K2) : $f^* \circ \eta_A = f$,
- (K3) : $(g^* \circ f)^* = g^* \circ f^*$.

A Kleisli triple $(\mathrm{M}, \eta, \text{-}^*)$ gives rise to a new category, the **Kleisli category** $\mathcal{C}^\mathrm{M}$. Its objects $\mathrm{Obj}_\mathcal{C}^\mathrm{M} := \mathrm{Obj}_\mathcal{C}$ are the objects of $\mathcal{C}$ and its morphisms between objects A and B are the morphisms $\mathrm{Mor}_\mathcal{C}^\mathrm{M}(A, B) := \mathrm{Mor}_\mathcal{C}(A, \mathrm{M}B)$. The identity of A is given by η_A, and the Kleisli composition of $f\colon A \to \mathrm{M}B$ and $g\colon B \to \mathrm{M}C$ is $(f \ggg g) := g^* \circ f\colon A \to \mathrm{M}C$. This issue is closely related to the monoidal structure of a monad we mentioned when presenting our examples. We will also call $\ggg$ the **bind** operator. As our objects are sets, we can define another operator **apply** $\gg=$. For any value $x \in \mathrm{M}A$ and any morphism $f\colon A \to \mathrm{M}B$ let $(x \gg= f) := f^*(x) \in \mathrm{M}B$.

Theorem 1 (Equivalence of monads and Kleisli triples)
The definition of monads and Kleisli triples are equivalent, i.e. for any monad there is a corresponding Kleisli triple and vice versa.

Proof. Let (M, η, μ) be a monad, then by $f^* := \mu_B \circ \mathrm{M}f$ for $f\colon A \to \mathrm{M}B$ we get the corresponding Kleisli triple $(\mathrm{M}, \eta, \text{-}^*)$:

$$
\begin{aligned}
&\text{(K1)} && \eta_A^* && \overset{\text{(Def)}}{=} \mu_A \circ \mathrm{M}\eta_A \overset{\text{(M4)}}{=} \mathrm{id}_{\mathrm{M}A} \\
&\text{(K2)} && f^* \circ \eta_A && \overset{\text{(Def)}}{=} \mu_B \circ \mathrm{M}f \circ \eta_A \overset{\text{(M1)}}{=} \mu_B \circ \eta_{\mathrm{M}B} \circ f \overset{\text{(M3)}}{=} \mathrm{id}_{\mathrm{M}B} \circ f \overset{\text{(Id)}}{=} f \\
&\text{(K3)} && (g^* \circ f)^* && \overset{\text{(Def)}}{=} \mu_C \circ \mathrm{M}(\mu_C \circ \mathrm{M}g \circ f) \overset{\text{(F2)}}{=} \mu_C \circ \mathrm{M}\mu_C \circ \mathrm{M}(\mathrm{M}g) \circ \mathrm{M}f \\
&&&&& \overset{\text{(M5)}}{=} \mu_C \circ \mu_{\mathrm{M}C} \circ \mathrm{M}(\mathrm{M}g) \circ \mathrm{M}f \overset{\text{(M2)}}{=} \mu_C \circ \mathrm{M}g \circ \mu_B \circ \mathrm{M}f \overset{\text{(Def)}}{=} g^* \circ f^*
\end{aligned}
$$

Let now $(\mathrm{M}, \eta, \text{-}^*)$ be a Kleisli triple, then by extending M to an endofunctor via $\mathrm{M}f := (\eta_B \circ f)^*$ for $f\colon A \to B$ and by $\mu_A := \mathrm{id}_{\mathrm{M}A}^*$ we get the corresponding monad (M, η, μ):

$$
\begin{aligned}
&\text{(F1)} && \mathrm{M}\,\mathrm{id}_A && \overset{\text{(Def)}}{=} (\eta_A \circ \mathrm{id}_A)^* \overset{\text{(Id)}}{=} \eta_A^* \overset{\text{(K1)}}{=} \mathrm{id}_{\mathrm{M}A} \\
&\text{(F2)} && \mathrm{M}(g \circ f) && \overset{\text{(Def)}}{=} (\eta_C \circ g \circ f)^* \overset{\text{(K2)}}{=} ((\eta_C \circ g)^* \circ \eta_B \circ f)^* \overset{\text{(K3)}}{=} (\eta_C \circ g)^* \circ (\eta_B \circ f)^* \\
&&&&& \overset{\text{(Def)}}{=} \mathrm{M}g \circ \mathrm{M}f
\end{aligned}
$$

$$
\begin{array}{lll}
\text{(M1)} & \mathrm{M}f \circ \eta_A & \stackrel{\text{(Def)}}{=} (\eta_B \circ f)^* \circ \eta_A \stackrel{\text{(K2)}}{=} \eta_B \circ f \\
\text{(M2)} & \mathrm{M}f \circ \mu_A & \stackrel{\text{(Def)}}{=} (\eta_B \circ f)^* \circ \mathrm{id}^*_{\mathrm{M}A} \stackrel{\text{(K3)}}{=} ((\eta_B \circ f)^* \circ \mathrm{id}_{\mathrm{M}A}) \stackrel{\text{(Id)}}{=} ((\eta_B \circ f)^*)^* \\
& & \stackrel{\text{(Id)}}{=} (\mathrm{id}_{\mathrm{M}B} \circ (\eta_B \circ f)^*)^* \stackrel{\text{(K2)}}{=} (\mathrm{id}^*_{\mathrm{M}B} \circ \eta_{\mathrm{M}B} \circ (\eta_B \circ f)^*)^* \\
& & \stackrel{\text{(K3)}}{=} \mathrm{id}^*_{\mathrm{M}B} \circ (\eta_{\mathrm{M}B} \circ (\eta_B \circ f)^*)^* \stackrel{\text{(Def)}}{=} \mu_B \circ \mathrm{M}(\eta_B \circ f)^* \stackrel{\text{(Def)}}{=} \mu_B \circ \mathrm{M}(\mathrm{M}f) \\
\text{(M3)} & \mu_A \circ \eta_{\mathrm{M}A} & \stackrel{\text{(Def)}}{=} \mathrm{id}^*_{\mathrm{M}A} \circ \eta_{\mathrm{M}A} \stackrel{\text{(K2)}}{=} \mathrm{id}_{\mathrm{M}A} \\
\text{(M4)} & \mu_A \circ \mathrm{M}\eta_A & \stackrel{\text{(Def)}}{=} \mathrm{id}^*_{\mathrm{M}A} \circ (\eta_{\mathrm{M}A} \circ \eta_A)^* \stackrel{\text{(K3)}}{=} (\mathrm{id}^*_{\mathrm{M}A} \circ \eta_{\mathrm{M}A} \circ \eta_A)^* \stackrel{\text{(K2)}}{=} (\mathrm{id}_{\mathrm{M}A} \circ \eta_A)^* \\
& & \stackrel{\text{(Id)}}{=} \eta_A^* \stackrel{\text{(K1)}}{=} \mathrm{id}_{\mathrm{M}A} \\
\text{(M5)} & \mu_A \circ \mathrm{M}\mu_A & \stackrel{\text{(Def)}}{=} \mathrm{id}^*_{\mathrm{M}A} \circ (\eta_{\mathrm{M}A} \circ \mathrm{id}^*_{\mathrm{M}A})^* \stackrel{\text{(K3)}}{=} (\mathrm{id}^*_{\mathrm{M}A} \circ \eta_{\mathrm{M}A} \circ \mathrm{id}^*_{\mathrm{M}A})^* \stackrel{\text{(K2)}}{=} (\mathrm{id}_{\mathrm{M}A} \circ \mathrm{id}^*_{\mathrm{M}A})^* \\
& & \stackrel{\text{(Id)}}{=} (\mathrm{id}^*_{\mathrm{M}A})^* \stackrel{\text{(Id)}}{=} (\mathrm{id}^*_{\mathrm{M}A} \circ \mathrm{id}_{\mathrm{MM}A})^* \stackrel{\text{(K3)}}{=} \mathrm{id}^*_{\mathrm{M}A} \circ \mathrm{id}^*_{\mathrm{MM}A} \stackrel{\text{(Def)}}{=} \mu_A \circ \mu_{\mathrm{M}A}
\end{array}
$$

□

This theorem justifies that for the rest of this work we will subsume both concepts under the term "monad", i.e. we assume a monad to have both join and lift/bind/apply operator. Moreover, when talking about a monad (or any kind of monadic/functorial structure yet to define) we only write the endofunctor M and assume that the corresponding operators are implicitly given by $\eta^{\mathrm{M}}, \mu^{\mathrm{M}}$, and so on. Usually we will drop the $^{\mathrm{M}}$ in these and the following notations if there is no risk of confusion.

2.1.3 Monads with Null

Coming back to our two examples we observe that the monads have a second way to embed elements, where for this embedding the elements do not behave like a unit value but like a null value to the multiplication. We call a monad with such a null embedding a monad with null.

(1) In parallel with the embedding $A \to \mathcal{P}(A)$ we have a second "embedding" given by $x \mapsto \emptyset$. This of course is no real embedding, but it can be seen as an embedding in the way that we take an element $x \in A$ and give a representation in $\mathcal{P}(A)$ expressing that the element is irrelevant. The null property can be seen for $\emptyset \in \mathcal{P}(\mathcal{P}(A))$ by $\bigcup \emptyset = \emptyset \in \mathcal{P}(A)$ and $\left\{ \bigcup x \mid x \in \emptyset \right\} = \emptyset \in \mathcal{P}(A)$.

(2) If R is a monoid with a null element[3] 0^R, then via $x \mapsto (0^R, x)$ we get an element in $R \times A$ for which the annotation will stay 0^R after any multiplication: $(r \cdot^R 0^R, x) = (0^R \cdot^R r, x) = (0^R, x)$ for any $r \in R$.

Definition 5 (Monad with Null)
A ***monad with null*** *is a monad* M *together with a* (N1) *natural transformation* $\nu \colon \mathrm{id}_{\mathcal{C}} \to \mathrm{M}$ *called* ***null*** *operator, such that the following two diagrams commute for any* $f \colon A \to \mathrm{M}A$*:*

$$
\begin{array}{ccc}
A & \xrightarrow{\nu_A} & \mathrm{M}A \\
{\scriptstyle f}\downarrow & \text{(N2)} & \uparrow{\scriptstyle \mu_A} \\
\mathrm{M}A & \xrightarrow[\nu_{\mathrm{M}A}]{} & \mathrm{M}^2A
\end{array}
\qquad
\begin{array}{ccc}
A & \xrightarrow{\nu_A} & \mathrm{M}A \\
{\scriptstyle f}\downarrow & \text{(N3)} & \uparrow{\scriptstyle \mu_A} \\
\mathrm{M}A & \xrightarrow[\mathrm{M}\nu_A]{} & \mathrm{M}^2A
\end{array}
$$

[3] A **monoid with null element** is a monoid $(R, \cdot^R, 1^R)$ with an element $0^R \in R$, such that $0^R \cdot^R r = r \cdot^R 0^R = 0^R$ for all $r \in R$.

2.1.4 Additive & Measurable Functors, Functors with Inverses

Beside the multiplication we might have an additive structure. In our example of a powerset this is the union operation $\cup\colon \mathcal{P}(A) \times \mathcal{P}(A) \to \mathcal{P}(A)$ which aggregates the elements of two sets to a single set. We call this property full additivity as the union is defined for any underlying object A. For the annotated values $R \times A$ we get a weaker form of additivity, namely if R has an addition $+^R$. We can add two annotated values by adding their annotation, but we can only do this without loss of information if their inner values are equal. So in particular annotated values over $\mathbb{1}$ are additive without restriction. This property is called additivity. The term "strong additivity" describes a generalization of additivity and will only matter for us in the context of monad composition.

Definition 6 (Additive Functor)
An endofunctor M *over a category* $\mathcal{C}$ *is called* ***additive*** *(**strongly additive**, **fully additive**) if for* $X = \mathbb{1}$ *(any object* $X = \mathrm{N}\mathbb{1}$ *with an additive endofunctor* N *over* $\mathcal{C}$*, any object* $X = A$*) we are given an associated binary operation* $+\colon \mathrm{M}X \times \mathrm{M}X \to \mathrm{M}X$ *called* ***addition****, which is associative on the elements of* $\mathrm{M}X$*. If the functor is part of a monad with null, the elements* $\nu_X(x)$ *for* $x \in X$ *are required to be neutral elements of the addition.*

The special object $\mathbb{1}$ also plays an important role in our next definition. Elements of $\mathrm{M}\mathbb{1}$ can tell us only about the "M-aspect" as the only inner value 1 is trivial. So $\mathcal{P}(\mathbb{1}) = \{\emptyset, \{1\}\}$ states that a set is either empty or not, and accordingly $R \times \mathbb{1} \cong R$ gives us the annotation. We therefore see morphisms of the form $\mathrm{M}A \to \mathrm{M}\mathbb{1}$ as measuring the M-part of a value. Again we have the generalization of strong measurability which will only be important for monad composition.

Definition 7 (Measurable Functor)
A ***measurable*** *functor is an endofunctor* M *over a category* $\mathcal{C}$ *together with a morphism* $\omega_A\colon \mathrm{M}A \to \mathrm{M}\mathbb{1}$ *for any object* A*.* ω *is called* ***measure*** *operator. If the functor is part of a monad (with null), the following diagram(s) have to commute:*

$$
\begin{array}{ccc}
A & \xrightarrow{\tau_A} & \mathbb{1} \\
\eta_A \downarrow & \text{(U1)} & \downarrow \eta_{\mathbb{1}} \\
\mathrm{M}A & \xrightarrow[\omega_A]{} & \mathrm{M}\mathbb{1}
\end{array}
\qquad
\begin{array}{ccc}
A & \xrightarrow{\tau_A} & \mathbb{1} \\
\nu_A \downarrow & \text{(U2)} & \downarrow \nu_{\mathbb{1}} \\
\mathrm{M}A & \xrightarrow[\omega_A]{} & \mathrm{M}\mathbb{1}
\end{array}
$$

A ***strongly measurable*** *functor is an endofunctor together with a morphism* $\omega_{\mathrm{N};A}\colon \mathrm{MN}A \to \mathrm{MN}\mathbb{1}$ *for any object* A *and any measurable and additive endofunctor* N*. In the case of a monad (with null) the following diagram(s) commute:*

$$
\begin{array}{ccc}
\mathrm{N}A & \xrightarrow{\omega_A^{\mathrm{N}}} & \mathrm{N}\mathbb{1} \\
\eta_{\mathrm{N}A}^{\mathrm{M}} \downarrow & \text{(SU1)} & \downarrow \eta_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \\
\mathrm{MN}A & \xrightarrow[\omega_{\mathrm{N};A}^{\mathrm{M}}]{} & \mathrm{MN}\mathbb{1}
\end{array}
\qquad
\begin{array}{ccc}
\mathrm{N}A & \xrightarrow{\omega_A^{\mathrm{N}}} & \mathrm{N}\mathbb{1} \\
\nu_{\mathrm{N}A}^{\mathrm{M}} \downarrow & \text{(SU2)} & \downarrow \nu_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \\
\mathrm{MN}A & \xrightarrow[\omega_{\mathrm{N};A}^{\mathrm{M}}]{} & \mathrm{MN}\mathbb{1}
\end{array}
$$

We do not dwell on the next definition. The structure of a functor with inverses will only be important in section 5.2 and we will give a more detailed explanation of its intention there. Nevertheless, it is easily seen that the inverses of the powerset monad are trivial (as $\mathcal{P}(\mathbb{1})$ only consists of trivial elements), and if we have a group R for annotation, the inverses of the annotated values are given by inverting the annotation.

Definition 8 (Functor with Inverses)
*A **functor with inverses (with strong inverses)** is an endofunctor* M *over a category* $\mathcal{C}$ *together with a morphism* $\text{-}^{-1} \colon \mathrm{M}X \to \mathrm{M}X$ *for* $X = \mathbb{1}$ *(*$X = \mathrm{N}\mathbb{1}$ *for any additive and measurable functor with inverses). If a functor with inverses is part of a monad (with null), the following holds for any* $u \in \mathrm{M}\mathbb{1}$ *(*$u \neq \nu_{\mathbb{1}}(1)$*):*

$$u^{-1} \ggg= (\mathrm{const}\, u) = u \ggg= (\mathrm{const}\, u^{-1}) = \eta_{\mathbb{1}}(1)\ , \quad \nu_{\mathbb{1}}(1)^{-1} = \nu_{\mathbb{1}}(1).$$

If a functor with strong inverses is part of a monad (with null), the following holds for any $u \in \mathrm{M}\mathrm{N}\mathbb{1}$ *(*$u \neq \nu^{\mathrm{M}}_{\mathrm{N}\mathbb{1}}(\eta^{\mathrm{N}}_{\mathbb{1}}(1))$*):*

$$u^{-1} \ggg= (\mathrm{const}\, u) = u \ggg= (\mathrm{const}\, u^{-1}) = \eta^{\mathrm{M}}_{\mathrm{N}\mathbb{1}}(\eta^{\mathrm{N}}_{\mathbb{1}}(1))\ , \quad \left(\nu^{\mathrm{M}}_{\mathrm{N}\mathbb{1}}(\eta^{\mathrm{N}}_{\mathbb{1}}(1))\right)^{-1} = \nu^{\mathrm{M}}_{\mathrm{N}\mathbb{1}}(\eta^{\mathrm{N}}_{\mathbb{1}}(1)).$$

We require only a functor in the last three definitions as in fact the presented structures are not related to the monad structure in the first place. But in the case that we have a monad the new structures fit together with the monad operators. If the functor of a monad has one of these properties, we will hence say that the monad has the property too. Observe also that strong additivity/measurability/inverses implies additivity/measurability/inverses via applying it to the trivial identity monad (which is additive, measurable, and has inverses; see 3.2).

2.2 Monads in Computer Science

2.2.1 The Use of Monads

Although monads were originally motivated by abstract algebra, they proved to be a general and uniform concept for describing a wide range of features of a programming language.

Moggi [14] was the first to suggest a categorical approach to structure the semantics of computations by using monads. In his attempt he looks at the category of types, where the objects are types and the morphisms are functions between types. Monadic types in this setting are understood as computations of the underlying type, while the monad itself describes the nature of the computation, e.g. possible occurrence of side-effects, dependency of the result, or indeterministic calculation. Programs are seen as the Kleisli morphisms, i.e. they have exact input values but computations as result. The unit operation is the inclusion of values into trivial computations, and the Kleisli lifting allows to apply programs on computations. In particular, it becomes possible to chain programs to a new program.

Adopting the basic ideas of Moggi, Wadler [19] then showed that one can actually program with monads. More specificly, he explained why this concept is a suitable framework to structure functional programs. Functional languages are appreciated for their strict relation to lambda calculus. This makes functional programs easy to reason since the data flow is explicit and there are no side-effects. But the upside is also the downside. Since there are no side-effects, it is inconvenient to have features like global variables, exceptions, or I/O. From a functional point of view these effects can be implemented by, for example, passing the required "global" variable as an extra parameter through all the functions up to the place where its value is needed. Wadler explains why the concept of monads solves many of the problems in an elegant way without leaving the purity of functional programming. On the other hand, it even gives useful features which would not be possible without the use of monads (at least not with the same simplicity).

Wadler shows how monads can be implemented in a functional programming language and he gives a clearer comprehension syntax for monads [18] inspired by the list comprehension syntax. For both Wadler and Moggi a monad given in the form of a Kleisli triple seems more appropriate for considerations in computer science. This is because a Kleisli triple explicitly explains the chaining of functions, an issue more related to programming than the "joining" of two layers of computational effects. Nevertheless, the categorical definition of a monad is often more suitable when it comes to prove things, and so the equivalence is useful.

Since then monads have been successfully applied to different situations. Many of the early works are concerned with the construction of modular interpreters, [12, 19, 15]. In Haskell a commonly used monad is the I/O monad. Kiselyov *et al.* [11] show how monads can be used to model backtracking computations. Erwig and Kollmansberger [4] use the probability distribution monad as a foundation of their domain-specific language for probabilistic programming. These are just a few examples of how widespread the field of possible applications of monads is. Our approach to use monads to model uncertain values is explained in the next section 3 and afterwards in section 4 and 5 it is applied to calculate alignments to uncertain sequences.

2.2.2 Implementation in Haskell

We show exemplarily how the definition of a monad can be transferred to a suitable type in Haskell. So let M be a monad. The endofunctor M assigns to each object A a corresponding monadic object $\mathrm{M}A$ and to each morphism $A \to B$ a morphism $\mathrm{M}A \to \mathrm{M}B$. In Haskell we get a special class of type constructors. A type constructor m forms a new type m a for any type a. All the type constructors of our subclass have to provide a function fmap :: (a → b) → m a → m b transferring the morphisms. This is the definition of an endofunctor over $\mathcal{H}$ and altogether looks as follows:

```
class Functor m where
    fmap :: (a → b) → m a → m b
```

Every monad is an endofunctor and hence monads over $\mathcal{H}$ are a subclass of this type class. The unit operator transcribes to a function unit :: a → m a and the join operator accordingly to join :: m (m a) → m a. Since monads and Kleisli triples are equivalent, we will aggregate both in one definition and get a third function lift :: (a → m b) → m a → m b. Theorem 1 states that it is sufficient if either join or lift is implemented. So we can give a default implementation of the other. But as there might be more efficient ways to implement the same function, the possibility to overwrite the default implementation is left open. All in all we get the following definition of a monad:

```
class (Functor m) ⇒ Monad m where
    unit :: a → m a
    join :: m (m a) → m a
    join = lift id
    lift :: (a → m b) → m a → m b
    lift f = join . (fmap f)
```

We left out the conditions given in the categorical definitions so far. In fact, it is not possible to add these restrictions to the Haskell implementation as the compiler cannot check whether they are satisfied for a given instance. So we only state them and require to carefully check instances of this class. At this point we recall the diagram presented in 1.3. The laws a monad has to satisfy are well-known as the "monad laws"[19]:

```
(F1)  fmap id ≡ id
(F2)  fmap (f . g) ≡ (fmap f) . (fmap g)
(M1)  (fmap f) . unit ≡ unit . f
(M2)  (fmap f) . join ≡ join . (fmap (fmap f))
(M3)  join . unit ≡ id
(M4)  join . (fmap unit) ≡ id
(M5)  join . (fmap join) ≡ join . join
(M6)  lift f ≡ join . (fmap f)
```

Although these laws cover all conditions for a monad, they can alternatively be given for the lift operator:

```
(K1)  lift unit ≡ id
(K2)  (lift f) . unit ≡ f
(K3)  lift ((lift g) . f) ≡ (lift g) . (lift f)
(K4)  fmap f ≡ lift (unit . f)
(K5)  join ≡ lift id
```

Next we can transfer the definition of the bind operator and the apply operator:

```
(⋙) :: (Monad m) ⇒ (a → m b) → (b → m c) → a → m c
f ⋙ g = (lift g) . f
(≫=) :: (Monad m) ⇒ m a → (a → m b) → m b
(≫=) = flip lift
```

The monad laws yield the following equivalences where the former two state the reason for the Kleisli category being a category:

```
Identity (K1, K2):  f ⋙ unit ≡ f ≡ unit ⋙ f
Associativity (K3):  (f ⋙ g) ⋙ h ≡ f ⋙ (g ⋙ h)
Identity (K1):  mx ≫= unit ≡ mx
Associativity (K3):  mx ≫= (f ⋙ g) ≡ (mx ≫= f) ≫= g
```

The other definitions can be transcribed to Haskell in the same way. Thus we do not comment on them any further and just give their implementation:

```
class (Monad m) ⇒ MonadWithNull m where
    mnull :: a → m a

class (Functor m) ⇒ Additive m where
    (+++) :: m () → m () → m ()

class (Additive m) ⇒ StronglyAdditive m where
    (+!+) :: (Additive n) ⇒ m (n ()) → m (n ()) → m (n ())

class (Functor m) ⇒ FullyAdditive m where
    plus :: m a → m a → m a

class (Functor m) ⇒ Measurable m where
    measure :: m a → m ()
    measure = fmap (const ())

class (Measurable m) ⇒ StronglyMeasurable m where
    smeasure :: (Measurable n, Additive n) ⇒ m (n a) → m (n ())
    smeasure = fmap measure

class (Functor m) ⇒ Invertible m where
    inverse :: m () → m ()

class (Invertible m) ⇒ StronglyInvertible m where
    sinverse :: (Additive n, Measurable n, Invertible n) ⇒ m (n ()) → m (n ())
```

2.3 Composing Monads

It was mentioned that monads provide a uniform way of modeling many useful features. But a monad only provides one explicit feature, and even though we can change the monad and consequently the effect in a properly[4] written program, it often is required to combine the effects of different monads. One simple but elaborate possibility is to implement the desired combination as a new monad every time. This of course is not useful when arbitrary combinations of monads are necessary. So the question arises whether there are more general ways to compose monads.

2.3.1 The Problem of Monad Composition

In fact, the question of composition turns out to be difficult, and no general composition method without restrictions can be found. A closer look at the composability reveals the problem. Proofs can be found in the appendix.

It is easyly seen that endofunctors M, N can always be composed to a new endofunctor $\mathrm{M} \circ \mathrm{N}$. If both functors are additive and M is strongly additive, the composition can be turned to an additive functor again. Basically the same works for measurable functors. If both functors are measurable, M is strongly measurable, and N is additive, then the composition is also a measurable functor. The case of functors with inverses is again the same.

The problem arises when we also try to compose two monads to a new monad. While the unit operator is no problem, trying to define the join/lift operator in the same simple way we have seen for the other structures does not work. The join operator illustrates the problem. We cannot apply one of the two join operators of the composing monads M, N to an object $\mathrm{MNMN}A$ since the monads alternate on each layer. In fact, the intuition that there is no natural way of composing two monads and preserve the monadic structure turns out to be right. There are different approaches to get a less general way of composing. We briefly present some of the early efforts in computer science.

King and Wadler [10] propose a library of primitive monads categorized by their composition technique. Every two monad can then be combined using the appropriate technique. They have particularly shown that the special kind of “container” monads (list, set, tree, bag) can all be composed with another monad in a similar way.

Jones and Duponcheel [9] try to find restrictions of the monads necessary to give a general technique of composition. Given two monads they distinguish four different cases and for each give a special helper function that makes the composition possible. Additionally, for each technique they state properties the two composing monads have to fulfill so that the method can be applied.

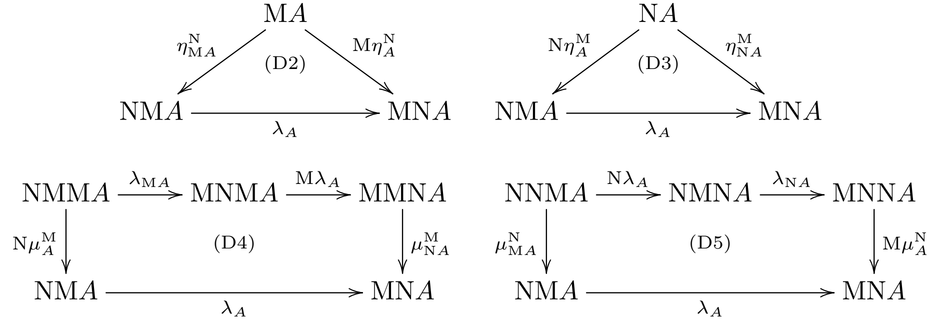
The probably most widespread approach is the concept of monad transformers, e.g. proposed by Liang, Hudak and Jones [12], but actually also the first attempt of King and Wadler [10] can be seen in view of monad transformers. A monad transformer basically is a monad which composes with every other monad. Quite a few of the more popular monads can be turned to a monad transformer (Option, Reader, State,...). It should furthermore be mentioned that the result of a composed monad using a monad transformer for `m` does not need to be of the form `m (n a)`, see for example the well-known state transformer: `s → m (s, a)` instead of `s → (s, m a)`.

[4] I.e. parametric in the monad, as far as possible

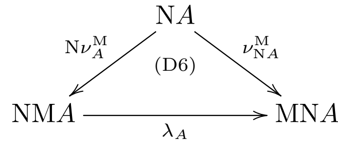
2.3.2 Distributive Laws

We now present a possible composition technique more precisely[5]. It is the approach we also use in our implementation. Beck [2] introduced distributive laws between two monads and showed that, given such a law, the composition of these two monads indeed is a monad again. We start by giving the definition of a distributive law.

Definition 9 (Distributive Law between Monads)
Given two monads $(\mathrm{M}, \eta^{\mathrm{M}}, \mu^{\mathrm{M}})$ *and* $(\mathrm{N}, \eta^{\mathrm{N}}, \mu^{\mathrm{N}})$ *over a category* $\mathcal{C}$*, a* ***distributive law*** *between* M *and* N *is a (D1) natural transformation* $\lambda\colon \mathrm{NM} \to \mathrm{MN}$*, such that the following four diagrams commute:*



If the monad is a monad with null, then the following diagram has to commute as well:



The following theorem states the desired outcome that the existence of a distributive law implies the existence of a composed monad.

Theorem 2 (Monad Composition)
Let $(\mathrm{M}, \eta^{\mathrm{M}}, \mu^{\mathrm{M}})$ *and* $(\mathrm{N}, \eta^{\mathrm{N}}, \mu^{\mathrm{N}})$ *be two monads. Given a distributive law* $\lambda\colon \mathrm{NM} \to \mathrm{MN}$ *for these two monads, their composition* $\mathrm{M} \circ \mathrm{N}$ *yields a monad by:*

$$(\mathrm{M} \circ \mathrm{N}\,,\ \eta^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_A\,,\ \mathrm{M}\mu^{\mathrm{N}}_A \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}\lambda_{\mathrm{N}A}).$$

If M *is a monad with null, then* $\mathrm{M} \circ \mathrm{N}$ *also is a monad with null via* $\nu^{\mathrm{M}\circ\mathrm{N}}_A := \nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_A$.

Proof. The proof can be found in the appendix.

□

For some monads the existence of a distributive law does not depend on the other monad, i.e. they commute with every other monad. Because of their universal composability these monads

[5] The forth method of composition presented by Jones and Duponcheel [9] corresponds to that approach. They use a helper function `swap :: n (m a) → m (n a)` to get a composition monad `m (n a)`.

yield monad transformers, and indeed many of the popular examples of monad transformers can also be given using such a universal distributive law.

In our implementation we define a new type representing the composition of functors/monads. We will use the abbreviation m ∘ n for Composition m n in the following code snippets. The distributive law is given as a type class providing the associated natural transformation.

```
newtype Composition m n a = Compose { decompose :: m (n a) }

class (Monad m, Monad n) ⇒ DistributiveLaw m n where
  dstr :: n (m a) → m (n a)
```

The following table summarizes the conditions the composed types m and n have to satisfy such that the composition type m ∘ n is instance of a certain type class:

Composition m ∘ n	Outer m	Inner n	Requires distributive law m n
Functor	Functor	Functor	–
Monad	Monad	Monad	Yes
Monad with null	Monad with null	Monad	Yes
Additive functor	Strongly additive functor	Additive functor	–
Strongly additive functor	Strongly additive functor	Strongly additive functor	–
Fully additive functor	Fully additive functor	Functor	–
Measurable functor	Strongly measurable functor	Measurable functor	–
Strongly measurable functor	Strongly measurable functor	Strongly measurable functor	–
Functor with inverses	Functor with strong inverses	Functor with inverses	–
Functor with strong inverses	Functor with strong inverses	Functor with strong inverses	–

3 Monads and Uncertainty

This section explains how monads can be used to describe uncertainty. For this purpose we are first going to introduce some basic terms, then present the definition of an uncertainty monad and take a closer look at uncertainties and monadic values. Afterwards we are ready to give and properly explain examples of uncertainty monads. As these monads are all well-known, we will focus on describing the new uncertainty aspect. In the last part we will generalize the concept of a property, i.e. we will show a way to reason about the conformance of objects to certain properties in the context of uncertainty.

3.1 A Theory of Uncertainty

When monads were introduced by Moggi [14], he described monads as notions of computation, as the monad determines the kind of abstraction a result of a computation of a certain type underlies. Looking at popular examples of monads, one can see that some of them can be summarized under a more precise description of their effects. There are monads modeling a possibly failing computation (option monad, exception monad), monads giving a computation a possibly indefinite result (indeterminism monad, probability distribution monad), some making the result of a computation dependent on a value (reader monad, state monad). When focusing on a certain subset of monads with a common more specialized structure one can reasonably make use of the additional feature. This is done by introducing a new subclass of monads with the desired operations. An example could be a monad that, given a list of possible results, chooses a suitable (for the respective monad) representation of the situation.

```
class (Monad m) ⇒ ChoiceMonad m where
    choose :: [a] → m a
```

Possible instances are the option monad giving the head element of the list if there is any, the indeterminism monad taking the list as possible results or the probability distribution monad giving a uniform distribution over the elements of the list.

We are also interested in a special kind of monad, namely the monads that can be seen as describing a kind of uncertainty. By doing so we do not want to be too focused on a special uncertainty (e.g. the probability distribution monad would be a suitable “prototype uncertainty”), but instead try to be as modest as possible when formulating our restrictions a monad has to fulfill to be a proper “notion of uncertainty”. It turns out that quite many of the popular examples of monads have an explanation as a special type of uncertainty.

Before we come to the concept of an uncertainty monad we first introduce an appropriate terminology to prepare reasoning about uncertainty. The new terms hopefully clearify our semantical understanding of these objects in the following considerations.

3.1.1 Terminology

We think of an object A of $\mathrm{Obj}_{\mathcal{C}}$ as a **value space** and the corresponding monadic object $\mathrm{M}A$ then is the **space of uncertain values (over A)** with elements called **uncertain values**. A Kleisli morphism $A \to \mathrm{M}B$ we will also call an **uncertain map (from A to B)**. The special uncertain map η_A we will also name **certain map** and, respectively, the uncertain map ν_A **impossible map**. The composition $f \ggg g$ of two uncertain maps $f\colon A \to \mathrm{M}B$ and $g\colon B \to \mathrm{M}C$ then is the uncertain map f **under the condition of** g, and applying an uncertain map $f\colon A \to \mathrm{M}B$ to an uncertain value $x \in \mathrm{M}A$ via $x \gg= f$ is the uncertain value x **under the condition of** f.

In our considerations the object $\mathbb{1}$ plays an important role. This is because it consists of only one single value and thus does not hold any information. So the uncertain values $\mathrm{M}\mathbb{1}$ over $\mathbb{1}$ represent "context-free" information about the monadic part of a monadic value and can be seen as "pure" uncertainties. We will therefore call $\mathrm{M}\mathbb{1}$ the **space of uncertainties** and its elements **uncertainties**. There are two special uncertainties, the **certainty** $\eta_{\mathbb{1}}(1)$ and the **impossibility** $\nu_{\mathbb{1}}(1)$. The space of uncertainties has an algebraic structure dependent on the properties of the considered monad.

Proposition 1 (Algebraic Structure of Uncertainties)
Let M *be a monad. The uncertainties equipped with the* ***multiplication*** $\cdot\colon \mathrm{M}\mathbb{1} \times \mathrm{M}\mathbb{1} \to \mathrm{M}\mathbb{1}$ *given by* $p \cdot q := p \gg= (\mathrm{const}\, q)$ *form a monoid with the certainty as neutral element. If the monad has inverses, we even have a group. If the monad is additive, the uncertainties with the* ***addition*** *also form a semigroup. If the monad is a monad with null, then the monoid has the impossibility as a* ***null element****, and given also the additivity of the monad, this null element is the neutral element to the additive semigroup, so we have a semiring*[1]*.*
Furthermore, the multiplicative monoid (with null) of uncertainties ***operates on the uncertain values*** $\mathrm{M}A$ *via* $\cdot_A\colon \mathrm{M}\mathbb{1} \times \mathrm{M}A \to \mathrm{M}A$ *given by* $p \cdot_A x := p \gg= (\mathrm{const}\, x)$*.*

Proof. We will use the fact that every element $p \in \mathrm{M}\mathbb{1}$ is equivalent to the only mapping $\mathrm{const}\, p$ of the form $\mathbb{1} \to \mathrm{M}\mathbb{1}$ with result p. So we take $(*)$ $p \equiv \mathrm{const}\, p$. The operation $\cdot\colon \mathrm{M}\mathbb{1} \times \mathrm{M}\mathbb{1} \to \mathrm{M}\mathbb{1}$ is associative by definition. The neutral element is given by $(*)$ $\eta_{\mathbb{1}}(1) = \eta_{\mathbb{1}}$ as an immediate result of the unit operator being the neutral element of the bind operator. The same holds for the null element $(*)$ $\nu_{\mathbb{1}}(1) = \nu_{\mathbb{1}}$. The inverses again are given by definition. The operation $+\colon \mathrm{M}\mathbb{1} \times \mathrm{M}\mathbb{1} \to \mathrm{M}\mathbb{1}$ is associative by definition. The neutral element $\nu_{\mathbb{1}}(1)$ is given by definition too.
So finally we have to show that the uncertainties operate on the uncertain values. This is because for any two uncertainties $p, q \in \mathrm{M}\mathbb{1}$ and any uncertain value $x \in \mathrm{M}A$ it holds:

$$
\begin{aligned}
\eta_{\mathbb{1}}(1) \cdot_A x &\overset{(\mathrm{Def})}{=} \eta_{\mathbb{1}}(1) \gg= (\mathrm{const}\, x) \overset{(*)}{=} \eta_{\mathbb{1}} \ggg (\mathrm{const}\, x) \overset{(\mathrm{K2})}{=} \mathrm{const}\, x \overset{(*)}{=} x \\
\nu_{\mathbb{1}}(1) \cdot_A x &\overset{(\mathrm{Def})}{=} \nu_{\mathbb{1}}(1) \gg= (\mathrm{const}\, x) \overset{(*)}{=} \nu_{\mathbb{1}} \ggg (\mathrm{const}\, x) \overset{(\mathrm{N2})}{=} \nu_{\mathbb{1}} \overset{(*)}{=} \nu_{\mathbb{1}}(1) \\
(p \cdot q) \cdot_A x &\overset{(\mathrm{Def})}{=} (p \gg= (\mathrm{const}\, q)) \gg= (\mathrm{const}\, x) \overset{(\mathrm{K3})}{=} p \gg= ((\mathrm{const}\, q) \ggg (\mathrm{const}\, x)) \\
&\overset{(*)}{=} p \gg= (q \gg= (\mathrm{const}\, x)) \overset{(\mathrm{Def})}{=} p \cdot_A (q \cdot_A x)
\end{aligned}
$$

□

[1] When talking about a **semiring (ring)** we take it to be both an additive monoid and a multiplicative monoid (group) where the additive neutral element is the null element of the multiplication. We do not require commutativity and the distributivity of the multiplication over the addition.

3.1.2 Uncertainty Monads

Definition 10 (Uncertainty Monad)
An ***uncertainty monad*** M *over* $\mathcal{C}$ *is a measurable monad with null. The measure operator* ω *we will also call* **uncertainty** *operator. The uncertainty monad is called* ***(fully) additive*** *if the monad is (fully) additive.*

One might also consider a weaker form of uncertainty monads for which no monad with null is required. This is possibly the weakest decent structure to describe uncertainties in the way we do. But we will see that for most of the following concepts the null operator is indispensable. Thinking of uncertainties, the fact that beside a certain map there also has to exist an impossible map seems a coherent postulation.

The uncertainty operator ω is understood as giving the entire uncertainty bound to an uncertain value by forgeting about the context, i.e. the type and its values. While the monad specifies the behavior of uncertain values and uncertain maps, the uncertainty operator determines the nature of the bare uncertainty. The uncertainty of indeterminism, for instance, might be distinguished between either having possible results or having no result. Everything else, primarily the specific result values, is then seen as dependent on the situation. Another interpretation might consider the amount of possible results as the characterizing quality of indeterminism. We see that indeterminism can be understood either as a binary or as a cardinal concept focusing on the bare uncertainty. So definitely the uncertainty operator plays a significant role for the monad with regard to the modeled uncertainty.

While fully additive uncertainty monads will be of no special interest in this section, additive monads will as they give an additive structure to the uncertainties of this monad. Adding two uncertainty values is understood as aggregating the uncertainty represented by the values. It can be seen as an "or" operation in the sense that we can get an uncertainty of the situation in which we either have a first uncertainty *or* an independent second uncertainty. A suitable "and" operation is the multiplication of uncertainties using the apply operator. This makes sense since a situation with a first uncertainty *and* an independent second uncertainty can be described as having the second uncertainty *under the condition* of the first. By associating certainty with true and impossibility with false we get the logical values as an inherent structure. And indeed the operations "and" and "or" correspond to the logical operations $\wedge$ and $\vee$. So an additive uncertainty monad has at least the expressiveness of the two logical values.

Every monad itself implicitly gives a possible description of uncertainty. It can be seen as the maximal expressiveness of the monad, while a different choice of the uncertainty operator reduces this expressiveness by forming some kind of quotient. The following proposition states the inherent free uncertainty monad of a monad.

Proposition 2 (Free Uncertainty Monad)
Every monad with null M *induces an uncertainty monad by making the functor* M *measurable via* $\omega_A := \mathrm{M}\tau_A$*. The so obtained monad is called* ***free uncertainty monad over*** M.

Proof.

$$\text{(U1)} \quad \omega_a \cdot \eta_A \overset{\text{(Def)}}{=} \mathrm{M}\tau_A \circ \eta_A \overset{\text{(M1)}}{=} \eta_{\mathbb{1}} \circ \tau_A \qquad \text{(U2)} \quad \omega_a \cdot \nu_A \overset{\text{(Def)}}{=} \mathrm{M}\tau_A \circ \nu_A \overset{\text{(N1)}}{=} \nu_{\mathbb{1}} \circ \tau_A$$

□

3.1.3 Equality of Uncertain Values

We will call a morphism of the form $A \to \mathrm{M}\mathbb{1}$ **uncertainty measure (over A)**. There are always two trivial uncertainty measures, namely the **certain measure** $\eta_{\mathbb{1}} \circ \tau_A$ and the **impossible measure** $\nu_{\mathbb{1}} \circ \tau_A$.

Beside the two trivial measures, there is a class of measures giving certainty only to some values. These are induced by predicates, i.e. functions of the form $p\colon A \to \mathrm{Bool}$. The trivial measures then are the corresponding measures to the trivial predicates true resp. false.

$$\chi_p\colon A \to \mathrm{M}\mathbb{1}, \quad x \mapsto \chi_p(x) := \begin{cases} (\eta_{\mathbb{1}} \circ \tau_A)(x) & \text{if } p(x) \\ (\nu_{\mathbb{1}} \circ \tau_A)(x) & \text{else} \end{cases}.$$

We will come back to uncertainty measures later or, more precisely, to a special kind of monad-independent measures which will call a property. Nevertheless, we want to do a few informal considerations and in particular broach the quotient structure of an uncertainty monad that we mentioned before. For this we assume that we can check two uncertainties for equality, i.e. we have a function of the form $=^{\mathrm{M}}\colon \mathrm{M}\mathbb{1} \times \mathrm{M}\mathbb{1} \to \mathrm{Bool}$. Actually, beside the following theoretical considerations the property of a (computable) equality check for uncertainties can be of practical use as shall see later in section 5.3.

A proper definition of equality for a type may take two values as equal if they behave equally, i.e. they give the same answer for every predicate on the corresponding type. By extending this idea on uncertain values we can define that two uncertain values $x, y \in \mathrm{M}A$ are **equal (in view of uncertainty)**, if their uncertainty is the same for the predicate-induced uncertainty measure of any predicate $p\colon A \to \mathrm{Bool}$:

$$\omega_{\mathbb{1}}(x \ggg= \chi_p) \ =^{\mathrm{M}} \ \omega_{\mathbb{1}}(y \ggg= \chi_p).$$

It can be seen that the equality of uncertain values depends on the uncertainty operator. An arbitrary uncertainty monad now might take some uncertain values as equal which are not equal for the corresponding free uncertainty monad. This shall be clarified by the following example for the two different concepts of indeterminism we already mentioned when explaining the uncertainty operator. We denote the binary indeterminism by Binary and the cardinal by Cardinal.

Consider the possible result of six randomly taken red or black marbles:

$$x = \big[r, b, r, r, r, r\big], \quad y = \big[r, r, b, r, b, b\big]$$

We can measure the uncertainty and get the expected results. Binary says that there was at least one marble while Cardinal says that there were exactly six marbles:

$$\omega^{\mathrm{Binary}}(x) = \omega^{\mathrm{Binary}}(y) = \big[1\big], \quad \omega^{\mathrm{Cardinal}}(x) = \omega^{\mathrm{Cardinal}}(y) = \big[1, 1, 1, 1, 1, 1\big].$$

It can now easily be seen that for the binary indeterminism these two values are the same since every predicate on $\{r, b\}$ is true for *at least* one item of x if and only if it is true for *at least* one item of y. Therefore holds:

$$\big[r, b, r, r, r, r\big] =^{\mathrm{Binary}} \big[r, r, b, r, b, b\big].$$

But for the cardinal indeterminism any non-trivial predicate states the inequality of the values, for example let $p = \left(x \mapsto x \stackrel{?}{=} b\right)$:

$$\omega^{\text{Cardinal}}\big(\big[r, b, r, r, r, r\big] \ggg \chi_p\big) = \omega^{\text{Cardinal}}\big(\big[1\big]\big) = \big[1\big]$$
$$\neq \big[1, 1, 1\big] = \omega^{\text{Cardinal}}\big(\big[1, 1, 1\big]\big) = \omega^{\text{Cardinal}}\big(\big[r, r, b, r, b, b\big] \ggg \chi_p\big).$$

So we have:

$$\big[r, b, r, r, r, r\big] \neq^{\text{Cardinal}} \big[r, r, b, r, b, b\big].$$

This shows that a less strict uncertainty operator, e.g. Binary compared to Cardinal, takes more values as equal when seen in the context of uncertainty. In fact, Cardinal is the free uncertainty monad over the list monad while Binary is a quotient uncertainty monad of the list monad where multiple occurence is equal to single occurence. Observe also that the order of the elements never matters. After applying the measure there is only the value 1 and so measuring implies forgetting the order.

We can say more about these two uncertainty monads. One can see that binary indeterminism basically is equivalent to the powerset monad (order and multiple occurence do not matter), and cardinal indeterminism accordingly is equivalent to the multiset monad (only order does not matter). The structure of the list uncertainty monad caused by a certain choice of the uncertainty operator hence results in a corresponding quotient structure of the list monad. Here we have sets resp. multisets as a quotient of a list. While for theoretical considerations this is not significant, it is a problem that in general it is not possible to implement such quotient types[2]. To solve this issue we give a suitable semantic interpretation of uncertain values (recall the diagram in 1.3). Given two uncertain values $x, y \in \mathrm{M}A$ we define **semantic equality** in the following way:

$$[\![x]\!] \equiv^{\mathrm{M}} [\![y]\!] \quad :\Leftrightarrow \quad \forall\, p\colon A \to \mathrm{Bool}: \;\; \omega_{\mathbb{1}}(x \gg= \chi_p) \;=^{\mathrm{M}}\; \omega_{\mathbb{1}}(y \gg= \chi_p).$$

We reduce the comparison of uncertain values to applying predicates and comparing the resulting uncertainties. In doing so we get the desired quotient structure at least on the level of interpretation.

[2]Here one needs a way to check equality of values, but this is not possible in general (e.g. functions).

3.2 Examples of Uncertainty Monads

Now we present some examples of uncertainty monads. We do not want to go into details concerning the formal definition and implementation. The examples shall instead be explained by reference to their behavior and interpretation in the context of uncertainty. All of the presented monads are additive uncertainty monads, except the identity and environment monad which are missing the null operator. But both the identity monad can be turned to a monad with null (the result is our option monad) and the environment monad (with the help of the option monad).

Identity Monad

The identity monad Id is the trivial monad with $\mathrm{Id} = \mathrm{id}_{\mathcal{C}}$ and $\eta^{\mathrm{Id}} = \mu^{\mathrm{Id}} = \mathrm{id}$. Seen in the context of Kleisli triples we have $\mathcal{C}^{\mathrm{Id}} = \mathcal{C}$, so the Kleisli composition is equal to the composition of $\mathcal{C}$. The identity monad obviously has a distributive law for any other monad. In the context of uncertainty the identity monad is describing the trivial uncertainty with only certainty represented by 1. The unique uncertainty operator is given by τ, the unique additive structure is $1 + 1 = 1$, and the inverse of 1 is again 1.

Option Monad

The option monad Option is the most trivial monad with null over the identity monad in the sense that we have $\mathrm{Option}\,A := {!A} \,\dot\cup\, \{0_A\}$, i.e. we add a new null element for each object. We have $\eta_A^{\mathrm{Option}}(a) = {!a}$ and $\nu_A^{\mathrm{Option}}(a) = 0_A$. 0_A is the null object of the join operator, so its result is null for the null value and the identity else: $\mu_A^{\mathrm{Option}}(0_{\mathrm{Option}\,A}) = 0_A$ and $\mu_A^{\mathrm{Option}}(!x) = x$ for $x \in \mathrm{Option}\,A$. The option monad has a distributive law $\lambda^{\mathrm{M}} \colon \mathrm{Option}\,\mathrm{M} \to \mathrm{M}\,\mathrm{Option}$ for any other monad M via $\lambda_A^{\mathrm{M}}(0_{\mathrm{M}A}) := \eta^{\mathrm{M}}_{\mathrm{Option}\,A}(0_A)$ and $\lambda_A^{\mathrm{M}}(!x) := (\mathrm{M}\,\eta_A^{\mathrm{Option}})(x)$ for any $x \in \mathrm{M}A$.

The option monad models the uncertainty with two values, certainty $\eta_{\mathbb{1}}^{\mathrm{Option}}(1) = {!1}$ and impossibility $\nu_{\mathbb{1}}^{\mathrm{Option}}(1) = 0_{\mathbb{1}}$. Obviously, there is only one unique uncertainty operator with $\omega_A^{\mathrm{Option}}(!x) = {!1}$ and $\omega_A^{\mathrm{Option}}(0_A) = 0_{\mathbb{1}}$, namely the operator of the free uncertainty monad over Option. We get the additive structure by using the trivial additive structure of the underlying identity monad and extend it by taking the new null element as the neutral element, i.e. $0_{\mathbb{1}} + x = x + 0_{\mathbb{1}} = x$ for $x \in \mathrm{Option}\,\mathbb{1}$. The two uncertainties are their own inverses with respect to the multiplication.

The emerging ring structure corresponds to the ring structure of the logical values. This is not surprising as the boolean values are the formalization of the behavior of the two logical values which themselves describe a kind of uncertainty, namely the bivalent classic logic. In classic logic one either has a certainly positive or a certainly negative, i.e. an impossibly positive, result, in perfect accordance to the uncertainties of the option monad. The option monad is the most trivial extension of the identity monad forming an additive uncertainty monad. Hence we will also call it the **trivial uncertainty monad**.

List/Multiset Monad

The list monad List is a fully additive monad (with null). Formally, given an object A it forms the free monoid $\mathrm{List}\,A$ over the set A with the unit operation $\eta_A^{\mathrm{List}}(x) = [x] \in \mathrm{List}\,A$ and the null operation $\nu_A^{\mathrm{List}}(x) = [] \in \mathrm{List}\,A$ for any $x \in A$. A morphism $f \colon A \to B$ is transformed to the only monoid homomorphism $\mathrm{List}\,f \colon \mathrm{List}\,A \to \mathrm{List}\,B$ obeying $\mathrm{List}\,f([x]) = [f(x)]$ for any $x \in A$.

The join operation is obtained by applying the inner monoidal structure to collapse two layers of free monoid constructions. The list monad is fully additive by construction via the monoidal operation.

Examining the list monad from the uncertainty's point of view we first observe that the space of uncertainties List $\mathbb{1}$ is the free monoid over a singleton set and hence can be treated (up to isomorphism) as the commutative monoid of the natural numbers $\mathbb{N}_0$. The uncertainty operator ω_A^{List} of the free uncertainty monad over List then is the unique monoid homomorphism from List A to List $\mathbb{1} \cong \mathbb{N}_0$ with $\omega_A^{\mathrm{List}}([x]) = [1] \equiv 1$ for any $x \in A$. The actual semiring structure of List $\mathbb{1}$ indeed is equivalent to the semiring $\mathbb{N}_0$ too.

The list monad corresponds to the cardinal indeterminism which we discussed before in section 3.1. When we defined equality in view of uncertainty, we saw that we actually consider lists modulo permutation. This is because the commutativity of the uncertainties List $\mathbb{1} \cong \mathbb{N}_0$ influences equality in the way that the order is forgotten. Hence our list monad rather is the multiset monad, i.e. the monad forming the free commutative monoid over a set. Nevertheless, we stick to the list monad as explained here and instead do not care about the ordinal structure of a list on the level of semantic interpretation.

Indeterminism/Powerset Monad

The indeterminism monad Indet uses the list monad List (without any uncertainty monad structure) as basis, i.e. all monad operations and full additivity are defined equally. The difference lies in the uncertainty operator and the additive structure. We define ω^{Indet} via $\omega_A^{\mathrm{Indet}}([]) = [] \equiv 0$ and $\omega_A^{\mathrm{Indet}}(x) = [1] \equiv 1$ for any $[] \neq x \in \mathrm{Indet}\, A$. So we have only two uncertainties Indet $\mathbb{1} = \{[], [1]\}$. This corresponds to the identification of every number greater than zero in $\mathbb{N}_0 \cong$ List $\mathbb{1}$. The structure of the uncertainties is equivalent to the uncertainties of the option monad.

The indeterminism monad is correlated to the binary indeterminism in section 3.1. Furthermore, our indeterminism monad basically is the powerset monad which we gave as an example in the introduction of the categorical terms in section 2.1. So we have $\mathcal{P}(\mathbb{1}) = \{\emptyset, \{1\}\} \cong$ Bool with $\cup, \cap$ as $\vee, \wedge$. On the level of interpretation we can say that indeterminism just states which results one has to expect. But it does not give any information about the relation of the results, e.g. any kind of likelihood of the single results (as multiple occurence in the case of the list monad is). But again we stick to lists in our implementation since using a set would restrict the monad to equatable types. Our definition of semantic equality of uncertain values yields the actual structure of the powerset monad.

Another way to look at the indeterminism monad is to take it as a monad forming a fully additive monad (with null) over another monad (here the identity monad) with regard to the additive structure of the underlying monad. The uncertainty operator then gives the uncertainty of a list of uncertain values and, using the fact that the inner uncertainties are additive, sums them up to a singleton list with the aggregated inner uncertainty. The procedure is similar to the join operator. We have a free monoid over the additive monoid of the inner uncertainties and use this fact to collapse the free monoid.

Annotation Monad

The annotation monad Ann endows values with an annotation of a certain set R (which is given together with the monad). We already explained this monad in section 2.1 when motivating the categorical concepts. Observe that the annotation monad has a distributive law $\lambda^{\mathrm{M}} \colon \mathrm{Ann}\,\mathrm{M} \to$

M Ann for any other monad M by pulling back the annotation, i.e. $\lambda_A^{\mathrm{M}}((p,x)) := \big(\mathrm{M}\,[y \mapsto (p,y)]\big)(x)$ for any $x \in \mathrm{M}A$.

In the context of uncertainty we see the annotation as some kind of quality score of the value. The multiplication of the monoid explains how iterated scores can be expressed in a single score, the neutral element resp. the null element gives the quality of certainty resp. impossibility, the addition is seen as the union of quality scores, and the inverses allow to invert uncertainties. A famous example of annotations are probability values. They are understood as giving the chance of correctness of a value. The corresponding ring structure is given by $(\mathbb{R}_{\geq 0}, \cdot, +)^3$. Another example are integer scores denoting deviation from the optimal solution. Here the corresponding ring structure is given by $(\mathbb{Z}, +, \min)$.

Environment Monad

The environment monad Env makes a value dependent on the selection of another value of a set E (which is given together with the monad). So $\mathrm{Env}\,A = \{f \mid f\colon E \to A\}$ for an object A. We embed elements $x \in A$ using the constant function $\eta_A^{\mathrm{Env}}(x) = \mathrm{const}\,x$. The join operator is given by $\mu_A^{\mathrm{Env}}(f) = \big(x \in A \mapsto (f(x))(x)\big)$ for $f \in \mathrm{Env}(\mathrm{Env}\,A)$.

There is only a single uncertainty $\tau_E = \mathrm{const}\,1$ as we have seen for the identity monad. So the environment monad has trivial additive and measurable structure and trivial inverses. As it is also lacking a null operator, it is not yet useful for our purposes. But its composition with the option monad $\mathrm{Env} \circ \mathrm{Option}$ resolves this deficit. This yields a monad again as the option monad composes with every outer monad. Basically, the composition is an extension of the environment monad as the option monad only adds a single null value. So the properties of the environment monad are the same for the composition. But we are now able to define a null operator via $\nu_A^{\mathrm{Env}\,\circ\,\mathrm{Option}}(x) = \eta_{\mathrm{Option}\,A}^{\mathrm{Env}} \circ \nu_A^{\mathrm{Option}} = \mathrm{const}\,0_A$ for all $x \in A$.

Distribution Monad, Mismatch Monad A famous monad is the probability distribution monad. The fact that probability distributions form a monad is well-known. Basic expositions are given by Giry [7] and Jones, Plotkin [8]. Our distribution monad Dist more precisely is the monad of finite distributions, not necessarily probability distributions. The objects of $\mathrm{Dist}\,A$ are finite distributions over A and hence can be seen as finite subsets of $\mathbb{R}_{\geq 0} \times A$. Accordingly, we define $\mathrm{Dist} := \mathrm{Indet} \circ \mathrm{Ann}$ with annotation ring $(\mathbb{R}_{\geq 0}, \cdot, +)$. The composition using the universal distributive law of the annotation monad explains the structure of this monad, as we have seen in 2.3. The mismatch monad Mism has exactly the same structure, i.e. it is defined $\mathrm{Mism} := \mathrm{Indet} \circ \mathrm{Ann}$, but with annotation ring $(\mathbb{Z}, +, \min)$.

3.2.1 A Remark on Indeterminism and Composability

We have not said anything about distributive laws concerning the indeterminism monad so far (beside the universal laws of other monads). And indeed there are no general constructions. At the first glance this does not matter as most of the other monads have such a construction. But the indeterminism monad plays an important role in the algorithm presented in the last section 5.2, and as some important monads, namely the distribution monad and the indeterminism monad itself, do not compose yet, it is worth to look at the subject a little closer.

[3] Actually, probabilities are associated with $[0, 1]$, but to gain additivity we take a broader understanding of probability.

Our claim is that the identity $\mathrm{id}_{\mathrm{Indet}(\mathrm{Indet}\,A)} : \mathrm{Indet}(\mathrm{Indet}\,A) \to \mathrm{Indet}(\mathrm{Indet}\,A)$ is a suitable distributive law considering our definition of equality for monadic values. The following three conditions have to be satisfied:

$$\eta_{\mathrm{Indet}\,A} = \mathrm{Indet}\,\eta_A\ , \quad \mu_{\mathrm{Indet}\,A} = \mathrm{Indet}\,\mu_A\ , \quad \nu_{\mathrm{Indet}_A} = \mathrm{Indet}\,\nu_A.$$

We only have two truly different uncertainties $\mathrm{Indet}(\mathrm{Indet}\,\mathbb{1}) \equiv \{\emptyset, \{\{1\}\}\}$ since $[\![\emptyset]\!] \equiv^{\mathrm{Indet}} [\![\{\emptyset\}]\!]$ and $[\![\{1, \emptyset\}]\!] \equiv^{\mathrm{Indet}} [\![\{1\}]\!]$. So despite the inner configuration of a two-layered set the only thing that matters for the uncertainty is the existence of an element not caring about order and number of occurence. This is similar to the observation we made for indeterminism at the end of section 3.1. The unit and join operator do not affect the existence of an element but just change the inner configuration of the list/set, while the null operator removes all elements. So indeed the conditions hold and $\mathrm{id}_{\mathrm{Indet}(\mathrm{Indet}\,A)}$ is a distributive law.

This fact has an interesting consequence. If we want to compose a monad K with a composition of two monads $\mathrm{M} \circ \mathrm{N}$ using distributive laws, it can be shown that one can define a suitable distributive law provided that the following holds:

- There are distributive laws $\lambda^{\mathrm{M}} : \mathrm{MK} \to \mathrm{KM}$ and $\lambda^{\mathrm{N}} : \mathrm{NK} \to \mathrm{KN}$.
- The distributive laws satisfy the **Yang-Baxter** equation (with $\lambda\colon \mathrm{NM} \to \mathrm{MN}$ denoting the distributive law of the composition $\mathrm{M} \circ \mathrm{N}$):

$$\lambda_{\mathrm{K}A} \circ \mathrm{M}\lambda^{\mathrm{N}}_A \circ \lambda^{\mathrm{M}}_{\mathrm{N}A} = \mathrm{N}\lambda^{\mathrm{M}}_A \circ \lambda^{\mathrm{N}}_{\mathrm{M}A} \circ \mathrm{K}\lambda_A.$$

The proof of (D1)-(D4) and (D6) can be found in the appendix. The Yang-Baxter equation is only necessary in the proof of (D5), which we omit. It can be found in [3].

Applied to the situation of $\mathrm{Indet} \circ \mathrm{Dist} = \mathrm{Indet} \circ (\mathrm{Indet} \circ \mathrm{Ann})$ we see that the Yang-Baxter equation trivially holds (λ now denotes the distributive law between Indet and Ann[4]):

$$\begin{aligned} \lambda_{\mathrm{Indet}\,A} \circ \mathrm{Indet}\,\lambda_A \circ \mathrm{id}_{\mathrm{Indet\,Indet\,Ann}\,A} &= \mathrm{Ann}\,\mathrm{id}_{\mathrm{Indet\,Indet}\,A} \circ \lambda_{\mathrm{Indet}\,A} \circ \mathrm{Indet}\,\lambda_A \\ \Leftrightarrow \quad \lambda_{\mathrm{Indet}\,A} \circ \mathrm{Indet}\,\lambda_A &= \lambda_{\mathrm{Indet}\,A} \circ \mathrm{Indet}\,\lambda_A. \end{aligned}$$

Trying it the other way round, i.e. $\mathrm{Dist} \circ \mathrm{Indet} = (\mathrm{Indet} \circ \mathrm{Ann}) \circ \mathrm{Indet}$, we need a distributive law $\mathrm{Indet\,Ann} \to \mathrm{Ann\,Indet}$ which we do not have. So the composition only works in one way.

The interesting point in this observation is that the question of composability of these two monads is much-discussed, see for instance the work of Varacca [17] or Gibbons [5]. Our composition is different from the composition for instance presented by Gibbons. However, we get the same result that the composition only works in the form $\mathrm{Indet} \circ \mathrm{Dist}$. The relationship shall only be noted here, but we will not go into detail. However, we will see that our way of composing indeterminism and distributions yields decent results when applying it to sequence alignment in the following sections.

[4]The same works not only for Ann but for any monad with a distributive law for the indeterminism monad.

3.3 Properties

In the last part of this section we want to revisit uncertainty measures as they will be useful for the application of monads shown in the following chapters. We are especially interested in measures, or more precisely measure constructions, which are independent from the underlying monad. We will call these measures properties as such measures are indeed suitable definitions of properties in the context of arbitrary uncertainty.

Definition 11 (Properties and Conformance)
The following two trivial uncertainty measures are ***properties (over** A**).***

- ***Certain Property:***

$$\textsc{Certain} := x \mapsto (\eta_{\mathbb{1}} \circ \tau_A)(x)$$

- ***Impossible Property:***

$$\textsc{Impossible} := x \mapsto (\nu_{\mathbb{1}} \circ \tau_A)(x)$$

If P *and* Q *are properties over* A*, the following again are properties over* A*:*

- ***Conditional Combination:*** *Given a predicate* $p\colon A \to Bool$

$$p \mathbin{?} \mathrm{P} : \mathrm{Q} := x \mapsto \begin{cases} \mathrm{P}(x) & \textit{if } p(x) \\ \mathrm{Q}(x) & \textit{else} \end{cases}$$

- ***And-Combination:***

$$\mathrm{P} \wedge \mathrm{P} := x \mapsto \mathrm{P}(x) \cdot \mathrm{Q}(x)$$

- ***Or-Combination***[5]*:*

$$\mathrm{P} \vee \mathrm{Q} := x \mapsto \mathrm{P}(x) + \mathrm{Q}(x)$$

- ***Computation:*** *Given an uncertain map* $\textsc{Comp}\colon A \to \mathrm{M}A$

$$\mathrm{P} \mid \textsc{Comp} := \omega \circ (\textsc{Comp} \ggg \mathrm{P})$$

Note that all constructions are parametric in the monad as long as we have an (additive) uncertainty monad. Given an uncertain value $x \in \mathrm{M}A$*, the uncertainty* $x \gg= P$ *is the* ***conformance of** x **with** P*.*

Why can these properties indeed be seen as a generalization of what we understood as a property so far? When talking about properties of values we usually think of a set of values A for which the property is defined as a more specific description. The property then indicates which values apply to this description, and so can be seen as a predicate $A \to Bool$, i.e. a property underlies the uncertainty of boolean values. We already saw in 3.2 that an uncertainty monad describing the boolean values is the option monad. So we can write the property in the way $A \to \mathrm{Option}\,\mathbb{1}$ and get a property for the option monad.

[5] Both the *and* and *or* combination are not necessarily commutative.

In the case of an arbitrary type of uncertainty a property does not necessarily just apply or not (these two cases are only special cases). We perhaps can say more about the conformance of a value with a property. Hence a property for any kind of uncertainty monad is of the form $A \to \mathrm{M}\mathbb{1}$ and it is up to the expressiveness of the monad what can be said about the conformance. We require the monad to be a monad with null in order to have at least the expressiveness of the boolean values. Since the or-combination needs the monad to be additive, for simplicity we will implicitly assume additive uncertainty monads in the remaining sections if required. The option monad, i.e. the most trivial monad with null over the identity monad, is the simplest (additive) uncertainty monad giving the least expressiveness of uncertainty. This indeed is the most trivial uncertainty monad in correspondence to the classic situation with the uncertainty of boolean values.

We will conclude this section by giving our implementation of properties in Haskell. Examples of properties can be found at the end of the next section in 4.3.

```
data Property m a =
    Certain
    | Impossible
    | Cond (a → Bool) (Property m a) (Property m a)
    | And (Property m a) (Property m a)
    | Or (Property m a) (Property m a)
    | Comp (Property m a) (a → m a)
```

To evaluate an instance of the type `Property m a` and get the underlying uncertainty measure `a → m ()` we need an additional evaluation function.

```
evaluate :: (AdditiveUncertaintyMonad m) ⇒ Property m a → a → m ()
evaluate Certain _ = certainty
evaluate Impossible _ = impossibility
evaluate (Cond pred p1 p2) x = if pred x then evaluate p1 x else evaluate p2 x
evaluate (p1 `And` p2) x = (evaluate p1 x) *** (evaluate p2 x)
evaluate (p1 `Or` p2) x = (evaluate p1 x) +++ (evaluate p2 x)
evaluate (Comp p f) x = uncertainty ((f >>> (evaluate p)) x)
```

4 Uncertain Sequences

In the following section we look at uncertain sequences. We first consider classic deterministic sequences and see how they can be generalized to uncertain sequences with the help of monads. The classic sequence should of course be a special case using the trivial uncertainty monad. We then give the definition of a sequence class in Haskell and several instances. Finally we will present basic functions and properties of an uncertain sequence.

4.1 Definition of Uncertain Sequences

To start reasoning about uncertain sequences we first have to think about what we understand by the term "uncertain sequence". While the term "sequence" is quite clear, it might not be obvious how uncertainty can be properly integrated.

We start with the term "sequence". A sequence is an ordered collection of elements. That is, every item in the sequence either has a successor or it is the last item in the sequence. A suitable data type modeling such a collection is the linked list type. This type provides functions that allow to check for an empty list and get the head value resp. the tail list if it is not empty. In functional languages a list usually is given as a type constructor taking a type `a` and in return gives the type `[a]` of lists of values of that type. Here an example of a list over two values $\{a, b\}$:

$$\big[a, b, a, a, b\big]$$

An obvious attempt to have sequences with values of some kind of uncertainty is to take a list of monadic values `[m a]` where `m` is a monad modeling the desired uncertainty. We can extend our example:

$$\big[a, \begin{matrix} a \\ b \end{matrix}, a, a, b\big]$$

In this sequence the second item is either an 'a' or a 'b'. We can model the situation using the indeterminism monad, so more correctly it should look as follows:

$$\big[[a], [a, b], [a], [a], [b]\big]$$

Although this attempt represents some kind of uncertain sequence, it does not describe the desired range of possible uncertainties. In fact, it only describes sequences where the uncertainty of each item is independent from every other, so in particular the successor of an item is not influenced by the uncertainty of the current value. But of course one might want to consider cases where they are not independent. Taking our above example it might be the case that if the second item is an 'a', the next item is a 'b', and if it is a 'b', the next is an 'a':

$$\big[a, \begin{matrix} a, b \\ b, a \end{matrix}, a, b\big]$$

It can easily be seen that this cannot be modeled by just having indeterministic values. Thus changing the parameter of the list type does not have the full desired effect. And indeed a closer look at the definition of the list type reveals a more natural way of modeling uncertain sequences.

In type theory the list type is described as an algebraic data type of the following form:

```
List a = 1 + (a × List a)
```

In words, a list either is empty, represented by the unit type, or it has a first item, the head of the list, and a successor sequence, the tail of the list. It can be seen that the definition of a list is very similar to the type theoretic definition of the option monad[1]:

```
Maybe a = 1 + a
```

We have seen in the last section in 3.2 that the option type is the trivial uncertainty monad. Since in our consideration the list type models a sequence with certain values, the similarity of the option monad and the list type is not surprising. A certain sequence either has a definite first value and a successor sequence or it has not, i.e. it is empty. So the type theoretic definition of a list with respect to the underlying monad looks like:

```
List a = Maybe (a × List a)
```

A natural attempt to generalize the monad of a sequence is to parametrize the list definition by a monad and compose it with the option monad.

```
Sequence m a = (Maybe ∘ m) (a × Sequence m a)
```

From the monadic point of view we could, instead of trivially forming a monad with null by composition with the option monad, just require the monad of a sequence to be a monad with null. The classic list type then is the special sequence using the most trivial monad with null, the option monad.

```
Sequence m a = (MonadNull m) => m (a × Sequence m a)
```

Although this definition basically describes uncertain sequences in the way we wanted, there is one problem. A sequence as well as a list is a recursively defined type, so to reason about it we usually use inductive arguments. Particularly, we need a base case. For lists this is the empty list and we have ways to check a list for emptiness. In the case of our definition of an uncertain sequence we need to assume a way to check a monadic value whether it is "empty", i.e. impossible.

However, requiring a function of the form isnull :: m a → Bool of a monad with null turns out to be a grave requirement. When working with arbitrary monads it is impossible to "look into" the monadic value and see the "inner value(s)". In the context of uncertainty this is comprehensible. It would not be truly uncertain if one could see what an uncertain value looks like and possibly act on this knowledge. But a function isnull allows exactly such an insight. For instance, we can define a function checking whether the "inner values" of a monadic value fulfill a particular predicate:

[1] The option monad in Haskell is the type Maybe a.

```
forall :: (MonadWithNull m) ⇒ (a → Bool) → m a → Bool
forall pred mx = isnull (mx ≫= (\x → if pred x then mnull x else unit x))
```

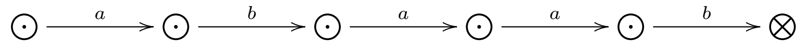
Fortunately, we do not need the assumption of such a function and so should carefully think whether to insist on it. The necessity had its origin in the merging of the option monad with the monad representing the underlying uncertainty of the sequence. However, the possibility to end was less a characterizing aspect of the uncertainty than a property of the sequence. Thus we define a sequence in which the monad only is responsible for the uncertainty of the values and their successors while the information of being empty is part of the sequence:

```
Sequence m a = 1 + m (a × Sequence m a)
```

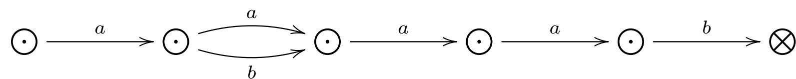
Finally, this is the most appropriate definition. It is now left to show that the examples we discussed above are all properly covered by this definition of an uncertain sequence. We will do this and simultaneously introduce a way to visualize uncertain sequences. It is to mention that the visualization of course only works for monads which can be visualized. For our examples this is the case.

Given a sequence we either have an empty sequence or a monadic value of value-successor-pairs. All examples of monads we introduced in the last section either annotate a value or have contain values which themselves possibly are annotated[2]. We denote empty sequences by $\otimes$ and non-empty sequences by $\odot$. Every value-successor-pair is given by an arrow pointing from the current non-empty sequence to the successor sequence. This arrow is labeled with the current value and, if any, the annotation. In the following the examples are recapitulated and visualized. Note that we always get some kind of tree.

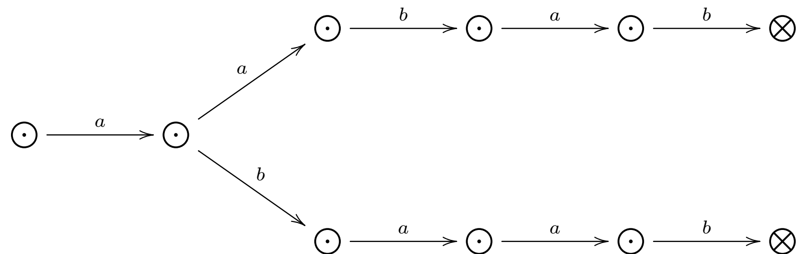
- The first example of a classic list:



- The example of an indeterministic list with independent values (we combine the two branches after the second item, but of course we could also give them as distinct branches to get a tree):



- The example of an indeterministic list with dependent values:



[2]For the environment monad we annotate the value with its fiber which always is finite in our examples.

4.2 Implementation in Haskell

Taking our type theoretic definition we can easily transfer it to Haskell.

```
data Tree m a = Leaf | Tree { branches :: m (a, Tree m a) }
```

Basically, this is the most general structure of an uncertain sequence, i.e. a sequence which either is empty or consists of an uncertain head value and a corresponding tail sequence dependent on this value. The identity monad yields the list type, the deterministic sequence.

There are some important sequences with a specific kind of restricted uncertainty. For instance, in one of the examples we considered a list of monadic values where the uncertainty of succeeding values are independent. To enable more specialized versions of uncertain sequences we introduce a sequence type class which represents our type theoretic definition.

4.2.1 The Sequence Type Class

The simple adoption of the type theoretic definition may look the following way:

```
class Sequence s where
    unwrap :: s m a → m (a, s m a)
    empty :: s m a → Bool
```

This definition has a weak point. There are sequence instances which do not work with all monads and so cannot be parametrized by the underlying monad. A simple example is the list type. It has one parameter, namely the type of its values, and can only be described as an uncertain sequence using the identity or the option monad. A even more serious problem arises if one tries to implement a specialized "meta-sequence". For instance, consider the case of a sequence which adds the possibility of skipping values to another sequence. This sequence can be implemented by taking an uncertain sequence and forming a new sequence which combines the indeterminism monad modeling all possible gaps with the monad of the underlying sequence.

Using functional dependencies we can give a more suitable definition of an uncertain sequence:

```
class (UncertaintyMonad m) ⇒ Sequence m s | s → m where
    unwrap :: s a → m (a, s a)
    empty :: s a → Bool
```

Now a sequence instance has two parameter, the uncertainty modeled by the sequence type and the sequence type. To avoid arising ambiguity problems (a sequence might be instanciated several times using different monads) the sequence type should determine the underlying monad. This dependency is natural as a sequence implementation cannot genuinely model two different kinds of uncertainty at the same time. We will see that both classic lists and sequences with gaps are instances of this class definition.

We require the monad to be an uncertainty monad. This enables us to apply our theory of uncertainty which we presented in the preceding sections. The classic list accordingly uses the option monad, the trivial uncertainty monad, instead of the identity monad. Furthermore, we only require that a sequence can be unwrapped if it is not empty.

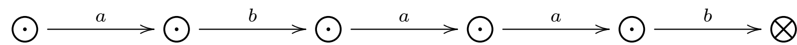
4.2.2 Sequence Instances

List

List are deterministic sequences using the option monad.

```
instance Sequence Maybe [] where
    unwrap (x:xs) = unit (x, xs)
    unwrap [] = error "Sequence is empty!"
    empty = null
```

We already saw an example of a classic list:

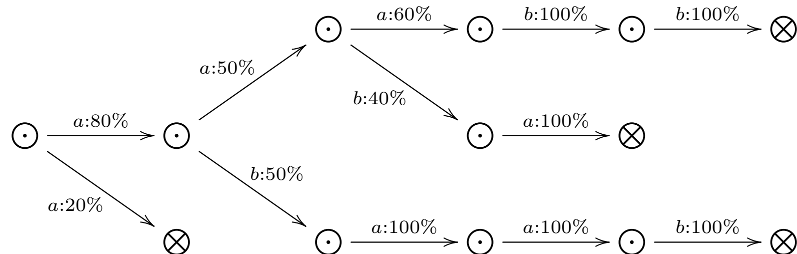


Tree

A tree with monadic branches is the direct adoption of the type theoretic definition.

```
data Tree m a = Leaf | Tree { branches :: m (a, Tree m a) }
instance (UncertaintyMonad m) ⇒ Sequence m (Tree m) where
    unwrap (Tree br) = br
    unwrap Leaf = error "Sequence is empty!"
    empty Leaf = True
    empty _ = False
```

The following example of a tree exposes some features an arbitrary uncertain sequence might exhibit. There can be several successor sequences for a single current value, and not every successor sequence has to have the same length. We will come back to this example

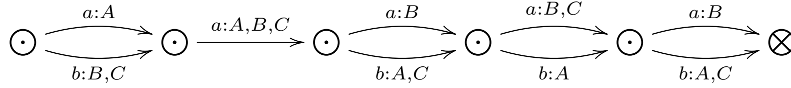


Monadic List

A list of monadic values is a special kind of uncertain sequence where the uncertainties of the items are pairwise independent.

```
newtype MonadicList m a = MList { mlist :: [m a] }
instance (UncertaintyMonad m) ⇒ Sequence m (MonadicList m) where
    unwrap (MList (mx:mxs)) = fmap ((flip (,)) (MList mxs)) mx
    unwrap (MList []) = error "Sequence is empty!"
    empty = null . mlist
```

The following example of a monadic list uses the environment monad. Its domain consists of three values, option A, B, C, and some of the values of the list depend on the choice of this option.

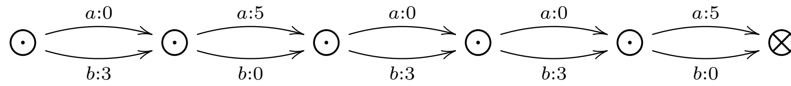


Perturbed List

A list together with a perturbation function gives a special kind of uncertain sequence where the certain items of the list are made uncertain using the perturbation function. Consequently, the uncertainty of two values is equal given two equal values in the underlying list. In particular, the values are pairwise independent. This sequence instance hence is a special case of a monadic list.

```
data PerturbedList m a = PList { plist :: [a], pert :: a → m a }
instance (UncertaintyMonad m) ⇒ Sequence m (PerturbedList m) where
    unwrap (PList (x:xs) f) = fmap ((flip (,)) (PList xs f)) (f x)
    unwrap (PList [] _) = error "Sequence is empty!"
    empty = null . plist
```

The example represents the classic list *"abaab"* but allows mismatches. The mismatch score is 3 when taking a *"b"* where an *"a"* was read, and the other way round it is 5.



Gapped Sequence

Our last sequence instance takes a given uncertain sequence and additionally allows gaps. An arbitrary gap can be seen as an indeterministic jump to one of the suffixes of the sequence. Hence the monad of the gapped sequence is Indet ∘ m. The definition of the function suffixes :: s a → (Indet ∘ m) (s a) is discussed in the next section in 5.2.

```
data GappedSeq s m a = (UncertaintyMonad m, Sequence m s) ⇒ GSeq { gseq :: s a }
instance (AdditiveUncertaintyMonad m, DistributiveLaw Indet m, Sequence m s) ⇒
        Sequence (Indet ∘ m) (GappedSeq s m) where
    unwrap (GSeq seq) =
        (suffixes seq) ≫= (Compose . unit . (fmap (\(x, s) → (x, GSeq s))) . unwrap)
    empty = empty . gseq
```

We do not present an example for this instance as allowing arbitrary gaps results in a large tree even for small sequences. To consider sequences with not entirely arbitrary gaps it is usually better to construct/implement a more customized representation of the sequence. However, this implementation shows that working with gapped sequences basically is possible.

4.3 Working with Uncertain Sequences

Next we want to introduce basic functions and properties that facilitate working with sequences. It shall be shown that with the preparatory work of the preceding sections it is not significantly more different to reason about uncertain sequences than about classic lists. However, we note that our sequences are a static concept, i.e. we do not introduce any means of sequence manipulation beyond the construction of a new sequence via the corresponding data constructors.

4.3.1 Predicates & Functions

Since the idea of uncertain sequences is a generalization of the idea of a list, we expect similar fundamental predicates and functions for sequences as we have for lists. Indeed there are essentially one predicate and three functions. We can check whether a sequence is empty via the null[3]-counterpart empty :: s a → Bool, we can get the current uncertain value of a sequence via the head-counterpart current :: s a → m a, and we can get an uncertain value of successor sequences via the tail-counterpart next :: s a → m (s a). Additionally, we define a new function step :: a → s a → m (s a) which gives the uncertain value of successor sequences to a given current value. For classic lists such a function did not exist as there was only a single value with the corresponding definite successor list.

While the predicate empty is part of the class definition, we can give the implementation of the three functions in Haskell:

```
current :: s a → m a
current = (fmap fst) . unwrap

next :: s a → m (s a)
next = (fmap snd) . unwrap

step :: (Eq a) ⇒ a → s a → m (s a)
step x = (lift (\(y, s) → if x == y then unit s else mnull s)) . unwrap
```

Note that if the sequence is empty, in general all of the functions are undefined, as we do not require that unwrap has any result in that case. For performance reasons we will add these functions to the class definition of an uncertain sequence and give a default implementation. For instance, this allows to give an improved implementation of the function next for monadic lists taking the independency of current value and successor sequence into account.

4.3.2 Properties

Now we present some basic properties over sequences. We start by giving the corresponding classic definition and afterwards show how properties as introduced in section 3.3 allow to easily transfer this definition to a compact and yet expressive generalization in the context of uncertainty. In the following we use the term "string" for classic lists since it is more common in this context.

[3] null, head, tail are the corresponding names of the functions for lists in Haskell.

Equality

If we think of the equality of two strings, one can give the following recursive definition:

> *Two strings are equal if*
> 1. *either both are empty*
> 2. *or they have equal head values and their tail strings are equal.*

With a slight shift of the point of view one can alternatively give a recursive definition of equality to a reference string:

> *A string is equal to a reference if*
> 1. *the string is empty given that the reference is empty,*
> 2. *or the string is not empty and the successor string to the head value of the reference is equal to the tail of the reference.*

The successor to a value in this case is meant to give the tail string if the head value equals the given value, and fail otherwise. This definition of equality to a reference string can be directly transferred to the more general concept of properties. More precisely, $\textsc{Equal}(L)$ is a property over uncertain sequences which gives the uncertainty of L being equal to the respective uncertain reference sequence.

$$\textsc{Equal}([]) \ := \ \mathsf{empty} \ ? \ \textsc{Certain} : \textsc{Impossible}$$
$$\textsc{Equal}(x\!:\!L) \ := \ \mathsf{empty} \ ? \ \textsc{Impossible} : \big[\textsc{Equal}(L) \ \mid \ \textsc{Step}(x)\big].$$

In the following definitions we proceed equally, i.e. given a string we construct the corresponding property over uncertain sequences which act as reference. The situation vice versa with the classic strings as references can be defined similarly.

Prefix

The classic definition:

> *A reference is a prefix of a string if*
> 1. *the reference is empty,*
> 2. *or the string is not empty and the tail of the reference is a prefix of the successor string to the head value of the reference.*

Transferred to a property:

$$\textsc{Prefix}([]) \ := \ \textsc{Certain}$$
$$\textsc{Prefix}(x\!:\!L) \ := \ \mathsf{empty} \ ? \ \textsc{Impossible} : \big[\textsc{Prefix}(L) \ \mid \ \textsc{Step}(x)\big].$$

Suffix

The classic definition:

> *A reference is a suffix of a string if*
> 1. *the reference is empty,*
> 2. *or the string is not empty and*
>   (a) *either the string is equal to the reference*
>   (b) *or the reference is a suffix of the tail of the string.*

Transferred to a property:

$$\textsc{Suffix}([]) \ := \ \textsc{Certain}$$
$$\textsc{Suffix}(x\!:\!L) \ := \ \mathsf{empty} \ ? \ \textsc{Impossible} : \big[\textsc{Equal}(x\!:\!L) \ \vee \ \textsc{Suffix}(L) \ \mid \ \textsc{Step}(x)\big].$$

Substring

The classic definition:

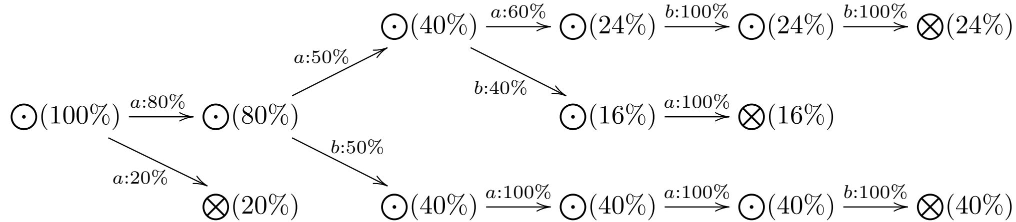
A reference is a substring of a string if
1. *the reference is empty,*
2. *or the string is not empty and*
   (a) *either the reference is a prefix of the string*
   (b) *or the reference is substring of the tail of the string.*

Transferred to a property:

$\textsc{Substring}([]) := \textsc{Certain}$
$\textsc{Substring}(x\!:\!L) := \textsf{empty} \;?\; \textsc{Impossible} : \big[\textsc{Prefix}(x\!:\!L) \vee \textsc{Substring}(L) \mid \textsc{Step}(x)\big].$

4.3.3 An Example

We conclude this section by exemplarily showing the usage of these properties by our example of the tree, but of course any of the other examples apply as well. For easier verification that indeed the results we get are correct we annotate to each item the overall probability of getting to this item from the start of the sequence:



We can now apply our properties to do some queries:

```
> evaluate (equal "aaba") example
[(16.00%,())]
> evaluate (equal "a") example
[(20.00%,())]
> evaluate (prefix "ab") example
[(40.00%,())]
> evaluate (prefix "abb") example
[]
> evaluate (suffix "ab") example
[(40.00%,())]
> evaluate (suffix "a") example
[(36.00%,())]
> evaluate (suffix "b") example
[(64.00%,())]
> evaluate (substring "ab") example
[(120.00%,())]
```

With the overall probabilities from above it can easily be seen that these results are correct. By calculating the occurence of substring we basically did a first alignment. In the next section a more efficient technique is presented.

5 Suffix Trees to Uncertain Sequences

5.1 About Suffix Trees

We have seen in the last section that with the help of the SUBSTRING property we can already compute alignments to an uncertain sequence. The underlying computations follow the naive technique of checking for a substring, i.e. checking every position in the sequence whether the substring can be found starting in that position. This method is fine for onetime alignments, but in practice it is often needed to compute multiple alignments to the same reference sequence. In such a case the naive method is very slow and there are more efficient ways. These methods usually first do precomputations processing the reference to be able to do faster alignments afterwards. One of them is an alignment algorithm using suffix trees which shall be presented in the following.

The basic idea of a suffix tree is the fact that every substring of a reference is a prefix of some (at least one) suffixes of the reference. So if one can check all suffixes for a certain prefix, one actually computes an alignment. The suffix tree to a reference is a compact representation of all its suffixes in a way that allows a simultaneous check on prefixes. The alignment algorithm using suffix trees is divided into two phases, the precomputation constructing the suffix tree to a given reference followed by the actual alignments exploiting the optimized structure of the tree, i.e. calculating only prefix-alignments. Ckecking for a prefix can be done in $\mathcal{O}(m)$ time while naively checking for a substring usually needs $\mathcal{O}(n \cdot m)$ time, where n is the length of the reference and m the length of the searched substring and usually it holds $m \ll n$. The great advantage of this method is the fact that the suffix tree has to be constructed only once and arbitrary many of the way faster alignments can be calculated afterwards.

While the actual alignment cannot be substantially improved any further, the next step is to improve the construction time of the suffix tree and also the space-efficiency of this new structure. The popular construction algorithms of Weiner [20], McCreight [13], and Ukkonen [16] achieve the theoretically optimal worst case time of $\mathcal{O}(n)$, where again n is the length of the reference. The downside of these algorithms is their intricate implementation. Another disadvantage of many suffix tree algorithms is that they are very greedy for space.

Our approach is based on a method of Giegerich, Kurtz and Stoye [6]. They explain a top-down construction of suffix trees suitable for functional programming languages. Although their algorithm is not even optimal in the average case, due to lazy evaluation it de facto is competitive regarding both time- and space-efficiency.

5.2 The Suffix Tree Construction Algorithm

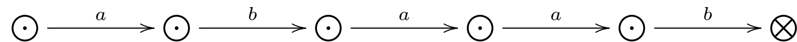
Our algorithm for the construction of a suffix tree to a given uncertain sequence is divided into two major parts. In the first part we calculate the suffixes for the sequence and in the second part we use these suffixes to construct the actual suffix tree.

5.2.1 Determining Suffixes

First, we need a way to get all suffixes of an uncertain sequence. In the last section we defined what it means for an uncertain sequence to have a specific *string* as suffix. But now we want to get the *true* suffixes, i.e. the full sequence that starts in a particular position. To gain clarity about how we can calculate them we take a closer look at an example of a classic list. Consider the string *"abaab"*, then the suffixes are given by *"abaab"*, *"baab"*, *"aab"*, *"ab"* and *"b"*. We can get all suffixes by going through the list and in every position taking the remaining list starting at this position. In Haskell the function can be given as follows:

```
classicsuffixes :: [a] → [[a]]
classicsuffixes [] = []
classicsuffixes xs = xs : (classicsuffixes (tail xs))
```

Now we look at this string given as a sequence:



When going through the list, stepping to the next position was by just taking the tail list without caring about the head value. The proper sequence function for stepping is NEXT (and not STEP as the current value is irrelevant). For an arbitrary sequence we do not get a definite successor sequence value but an uncertain value of possible successors. Consequently, a suffix starting at a specific position is an uncertain value of possible suffix sequences, and the first position can consistently be seen as a certain value. We expect the value of all suffixes to be of the type [m (s a)] for a sequence Sequence m s. A first attempt of implementing a suffixes function looks the following way assuming that the input is unit seq for an uncertain sequence seq:

```
suffixes' :: (Sequence m s) ⇒ m (s a) → [m (s a)]
suffixes' mseq = mseq : (suffixes' (mseq ≫= next))
```

Although this implementation seems to be in perfect analogy to the classic case, there is one obvious problem. As the argument is an uncertain value of sequences, we do not know when the inner sequences are empty and hence the function will usually result in an exception as soon as the first empty sequence is tried to be unwrapped.

We need to examine the case of a classic list a little more detailed. We have understood the suffixes function to be of the form m (s a) → [m (s a)] so far. But actually it can also be interpreted as of the form s a → [m (s a)] resp. s a → (Indet ∘ m) (s a). Similar to next :: s a → m (s a) giving an uncertain value of successor sequences suffixes :: s a → (Indet ∘ m) (s a) gives us an indeterministic uncertain value of suffix sequences. It seems more adequate that the calculation of suffixes results in the additional uncertainty of indeterminism. And the new point of view allows us to solve the problem of the first attempt similar to the way it is handled in the classic case. If our sequence is empty, there is definitely no suffix left and the result hence is

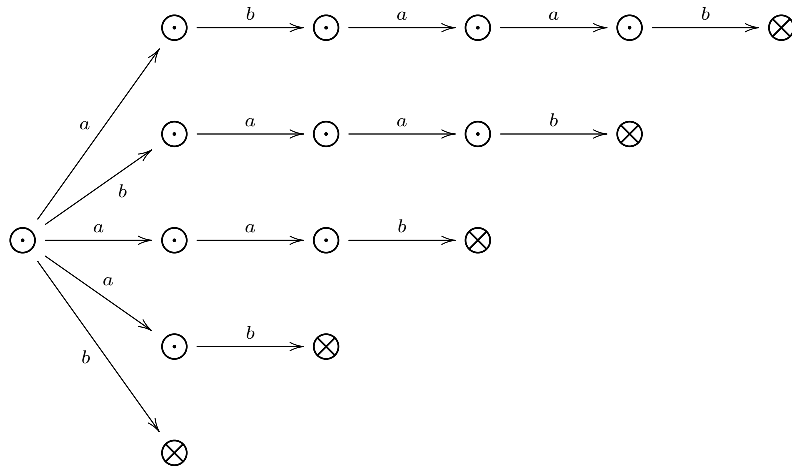
an impossible value. Otherwise we can apply the suffixes function recursively. Consequently, a correct suffixes function is:

```
suffixes :: (DistributiveLaw Indet m, Sequence m s) ⇒ s a → (Indet ∘ m) (s a)
suffixes seq = if empty seq then mnull seq
                    else plus (unit seq) ((Compose (unit (next seq))) ≫= suffixes)
```

Adding the current sequence to the suffixes of the remaining sequence requires us to use the addition of the fully additive composition monad Indet ∘ m. We found a way to calculate all suffixes of an uncertain sequence. But it comes to the prize that we have to require a distributive law between the indeterminism monad and the monad of the sequence.

5.2.2 An Intermediate Version of a Suffix Tree

With the calculation of suffixes the first step towards a suffix tree construction algorithm is done. We now have an indeterministic uncertain value containing all the possible suffix sequences. For a classic list the situation now looks as follows:



In the last section we saw that trees are sequences and hence we can use the suffixes of an uncertain sequence to form a new sequence. But now we consider a sequence with uncertainty Indet ∘ m instead of just m. Actually, only the first item really makes use of the expressiveness of the new uncertainty to hold all suffixes while the other items are the items of the former sequence trivially embedded in indeterministic values. It reminds of a gapped sequence where the underlying monad was of the same form. Actually, suffixes (and a suffix tree) to an uncertain sequence are a special form of gapped sequence. At this, the first position allows arbitrary gaps, i.e. immediate jumps to all suffixes, and the other positions represent the respective suffix.

We can implement this pre-version of a suffix tree similar to gapped sequences presented in the last section, but here only the root item allows gaps while the other items behave like the underlying sequences:

```
data SuffixesSeq s m a = (UncertaintyMonad m, Sequence m s) ⇒
        SSRoot { sseq :: s a } | SSBranch { sseq :: s a }

instance (AdditiveUncertaintyMonad m, DistributiveLaw Indet m, Sequence m s) ⇒
        Sequence (Indet ∘ m) (SuffixesSeq s m) where
    unwrap (SSRoot seq) =
        (suffixes seq) ≫= (Compose . unit . (fmap (\(x, s) → (x, SSBranch s))) . unwrap)
    unwrap (SSBranch seq) =
        Compose (unit ((fmap (\(x, s) → (x, SSBranch s))) (unwrap seq)))
    empty (SSRoot seq) = empty seq
    empty (SSBranch seq) = empty seq
```

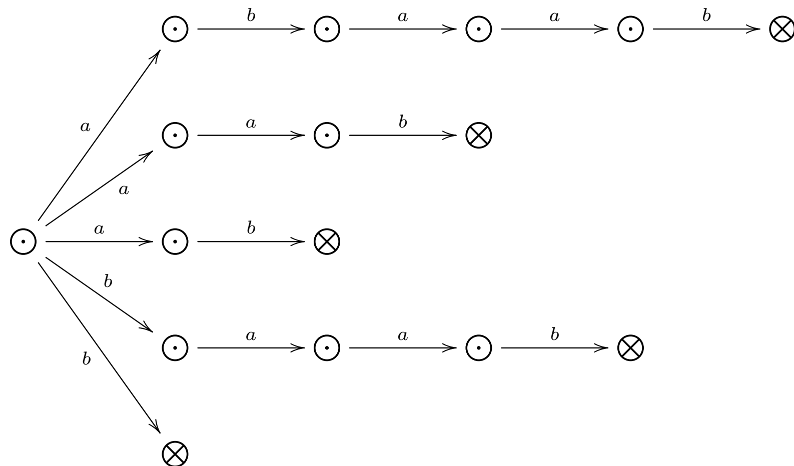
By now we did not really achieve anything regarding a more efficient alignment. As we still have to go through all suffixes, i.e. branches of the root, to find an alignment, we basically do the same as we do in the naive alignment method. And even worse we now use a data structure which potentially requires far more space than the original sequence. Improving both of these aspects will be the task of the second part of the suffix tree algorithm.

5.2.3 The Final Suffix Tree

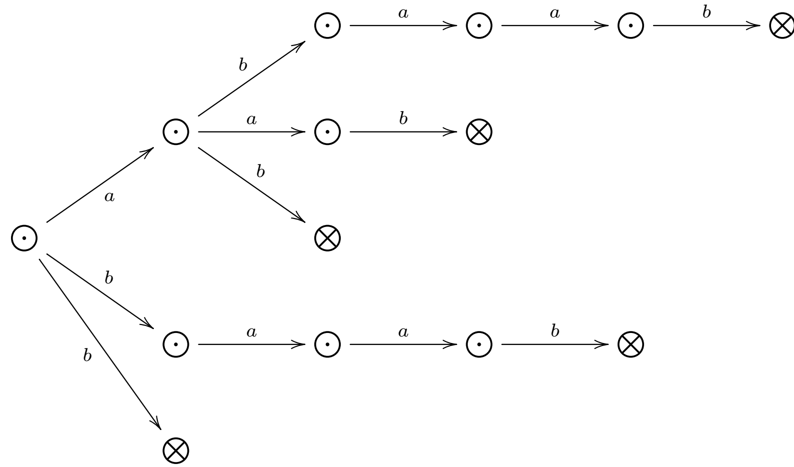
Now we want to use the suffixes sequence presented above as a basis and try to improve the structure by removing redundant information. The final suffix tree will still be a sequence with uncertainty Indet ∘ m and we will use the type Tree (Indet ∘ m) a to represent it.

The way to improve this intermediate version is obvious to see, at least for the example of a classic string presented above. The first item has multiple branches with the same label *"a"* resp. *"b"*. The idea is to first group and then unify these equal branches to a single branch and concurrently unify the corresponding successor items to a single successor item. For our example the procedure looks as follows:

1. After grouping:



2. After unifying:



Observe that we do not unify the two branches with label *"b"* as one of them points to an empty sequence. If we unified them, we would lose the information that there is a one-letter suffix *"b"*. For some applications this might not matter, but we nevertheless keep the information of ending branches. We unify all branches with non-empty successor and all branches with empty successor, and thus in general have two branches labeled with the same value.

In our very simple example we are done after applying this procedure once. However, generally we would now have the same situation of multiple branches with same labels for the new non-empty items. Hence we need to recursively apply this procedure of grouping and then unifying to the non-empty successors. Of course for finite sequences eventually there are no non-empty successors left and the procedure terminates.

In the following we want to transfer the method to sequences of arbitrary uncertainty. This once more involves steps which need further clarification. So we divide the second part, the actual construction of a suffix tree, in several substeps. The whole procedure of grouping and unifying starts with the indeterministic uncertain value of suffix sequences and at the end is recursively applied to the remaining suffixes in each branch.

1. Grouping of equally labeled successors

The first step already reveals a problem. Contrary to the situation of the visualized example, we generally cannot see which current values an uncertain sequence holds. So we do not know how to group the successors. This can be solved by requiring the entire alphabet, i.e. the list of all values occuring in the sequence, to be given. If we do so, the grouping step is quite simple. For every value x of the alphabet we take the list of suffixes and apply step x to the inner sequences. The alphabet is given by xs :: [a].

```
group :: (Indet ∘ m) (s a) → (Indet ∘ m) (a, Tree (Indet ∘ m) a)
group sufs = (Compose (Indet (fmap unit xs))) ≫=
                  (\x → fmap ((,) x) (unify (sufs ≫= (Compose . unit . (step x)))))
```

2. Divide empty and non-empty successors

Before we unify the successor sequences to a certain value we first partition empty and non-empty sequences. Afterwards we unify each of the two parts to a single successor leaf resp. tree and, using the full additivity of our composition Indet ∘ m, aggregate them again. The partition uses the function mfilter which filters an uncertain value given a predicate, i.e. all values not satisfying the predicate are made impossible.

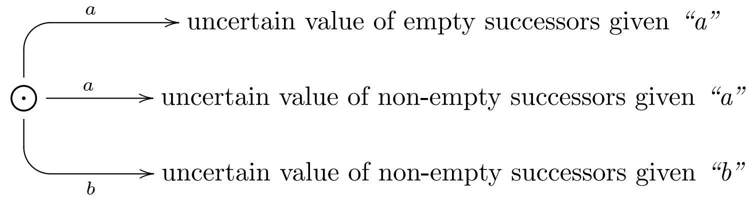
```
mfilter :: (MonadWithNull m) ⇒ (a → Bool) → m a → m a
mfilter pred = lift (\x → if pred x then unit x else mnull x)

unify :: (Indet ∘ m) (s a) → (Indet ∘ m) (Tree (Indet ∘ m) a)
unify sufs = plus (unifytoleaf (mfilter empty sufs)) (unifytotree (mfilter (not . empty) sufs))
```

3. Unification

Now the empty sequences have to be replaced by a single leaf and the non-empty sequences are used to construct the single successor suffix tree via recursively applying the construction function. Looking at the current situation it might be surprising that we can do this without getting uncertain successor items, although these represent the unification of all successors to the respective value and these successors are uncertain.

The reason for this situation is that we assume the alphabet to be given in the grouping step. This lets us calculate the successors for a value by "filtering from the outside" without ever really having to use the "inner values" of an uncertain value. The situation might be as follows:



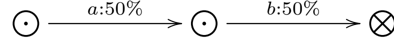
What we actually want to have is that the uncertainty is attached to the corresponding branch. So, roughly spoken, we need to retrospectively take the uncertainty of the successors and give it to the current item instead. It is done using the function weightbranch given below. This function calculates the entire uncertainty of the corresponding successors and attaches it to the corresponding item using the fact that the uncertainties operate on the uncertain values presented in section 3.1.

For empty successors we now get a branch leading to a leaf with the correct uncertainty. But for non-empty successors we on the one hand attach the uncertainty to the current branch and on the other hand use the uncertain value of successors in subsequent calculations, so we have the uncertainty twice. We can fix this issue by requiring the uncertainties to be invertible. Then the uncertainties in the situation behave correctly, as can be seen when sketching the uncertainty's point of view:

$$\omega(\text{succs}) = \omega(\text{succs}) \cdot \omega(\text{succs})^{-1} \cdot \omega(\text{succs}) = \underbrace{\omega(\text{succs})}_{(*)} \cdot \omega\big(\underbrace{\omega(\text{succs})^{-1} \cdot \text{succs}}_{(**)}\big).$$

The uncertainty (∗) is the uncertainty we attach to our current branch and the uncertain value (∗∗) is the normalized uncertain value of successors. The value is normalized in the sense that its uncertainty is certainty. The normalization is achieved by requiring invertible uncertainties[1].

The normalization makes perfectly sense if we think of the case of probabilities. The succeeding suffix tree is the suffix tree that is used *under the condition* that the corresponding prior value has been taken. Such conditional probabilities always are connected with some kind of normalization. Consider the following simple example:



We have the probabilities $P(\text{“}a\text{”}) = 50\%$ and $P(\text{“}ab\text{”}) = 25\%$. Applying the formula for conditional probabilities results in:

$$P(\text{“}ab\text{”} \mid \text{“}a\text{”}) = P(\text{“}a\text{” and “}ab\text{”}) \cdot P(\text{“}a\text{”})^{-1} = P(\text{“}ab\text{”}) \cdot P(\text{“}a\text{”})^{-1} = 25\% \cdot 2 = 50\%.$$

Indeed the *“b”*-branch is annotated with 50%. We do a similar calculation in the situation of arbitrary uncertainty. Hence we normalize using the inverse of the uncertainty for taking the prior value. In the classic case this observation was of no importance since we only have trivial inverses for the option monad.

After all we can present the correct implementation of the unification step:

```
normalize :: (UncertaintyMonad m, Invertible m) ⇒ m a → m a
normalize mx = (inverse (measure mx)) ∗∗∗ mx

unifytoleaf :: (Indet ∘ m) (s a) → (Indet ∘ m) (Tree (Indet ∘ m) a)
unifytoleaf sufs = weightbranch sufs Leaf

unifytotree :: (Indet ∘ m) (s a) → (Indet ∘ m) (Tree (Indet ∘ m) a)
unifytotree sufs = weightbranch sufs (stree (normalize sufs))

weightbranch :: (Indet ∘ m) (s a) → Tree (Indet ∘ m) a
                                  → (Indet ∘ m) (Tree (Indet ∘ m) a)
weightbranch sufs t = (uncertainty sufs) ∗∗∗ (unit t)
```

The algorithm

All in all we get the following top-down approach of a suffix tree construction algorithm for uncertain sequences:

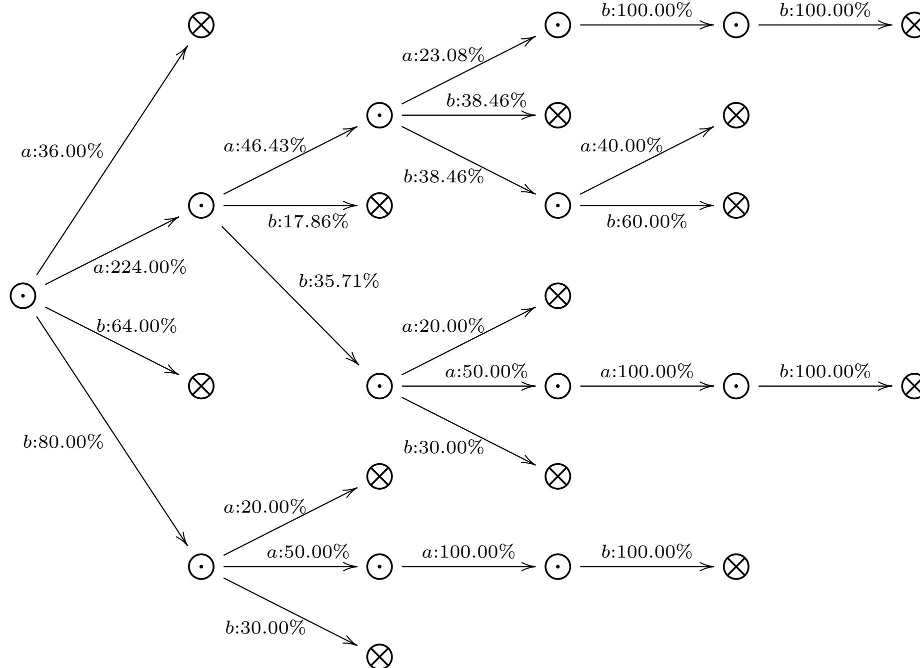
```
suffixtree :: forall a m s. (Eq a, AdditiveUncertaintyMonad m, DistributiveLaw Indet m,
        Invertible m, Sequence m s) ⇒ [a] → (Indet ∘ m) (s a) → Tree (Indet ∘ m) a
suffixtree xs =
    let group, unify, unifytoleaf, unifytotree, weightbranch ...
        stree :: (Indet ∘ m) (s a) → Tree (Indet ∘ m) a
        stree = Tree . group
    in stree . (mfilter (not . empty))
```

[1]In fact it would be sufficient to require only a normalize function. But we do not see an easy way of inheriting this property in the context of composition. Moreover, every “normalizable” monad of our example monads also has inverses.

We need to restrict ourselves to sequences using an additive uncertainty monad m. This monad is required to have invertible uncertainties and the composition with the indeterminism monad has to yield a monad again, i.e. we need a suitable distributive law. Unfortunately, we do not have such a distributive law for the environment monad. Consequently, we cannot apply our suffix tree algorithm to sequences using this monad. But the less efficient alignment method using the substring property still works.

5.2.4 An Example, revisited

We can use our algorithm to construct the suffix tree to the example of the last section in 4.3 (some of the probabilities are rounded):



Every item except the first has outgoing probabilities that sum up to 1. This is the result of normalizing the succeeding suffixes. It does not hold for the root item as it represents all suffixes and is not conditioned by any prior value. In fact, the probabilities sum up to 4.04, which is the expected length of the sequence. Furthermore, we see that our previous alignment of *"ab"* via the substring property now can easily be calculated, $224.00\% \cdot (17.86\% + 35.71\%) \approx 120\%$ (actually we get 119.9968% because of the rounded probabilities). Or we apply the prefix property:

```
> evaluate (prefix "ab") (suffixtree "ab" (suffixes example))
[[(120.00%,())]]
```

5.3 Further Remarks

5.3.1 Correctness via QuickCheck

With the help of properties we can use QuickCheck to easily examine the correctness of our algorithm. QuickCheck is a library written in Haskell that automatically generates various test cases for certain predicates. To be able to write proper predicates we require the uncertainties of a monad to be equatable as we did in section 3.1, i.e. we have a function (===) :: m () → m () → Bool. For equalities like [] === [[]] in the case of composed monads we also need a function isnull :: m () → Bool which we already discussed in section 4.1[2]. In our implementation we use the following type classes of equatable functors similar to the other kinds of special functors:

```
class (Functor m) ⇒ Equatable m where
    (===) :: m () → m () → Bool
    isnull :: m () → Bool

class (Equatable m) ⇒ StronglyEquatable m where
    ( ≡ ) :: (Equatable n, Measurable n, Additive n) ⇒ m (n ()) → m (n ()) → Bool
    sisnull :: (Equatable n, Measurable n, Additive n) ⇒ m (n ()) → Bool
```

We can now define the predicate stating the correctness of a suffix tree, i.e. a string is a substring of a sequence with the same uncertainty as it is a prefix of the suffix tree of this sequence. Since the uncertainty types are not the same, we need to embed the uncertainty of type m () resulting of the substring query in the type (Indet ∘ m) () via Compose . unit.

```
checksuffixtree :: (QCValues a, Eq a, AdditiveUncertaintyMonad m, Invertible m,
                        Equatable m, DistributiveLaw Indet m, Sequence m s)
        ⇒ s a → [QCValue a] → Bool
checksuffixtree seq xs =
    let tree = suffixtree values (suffixes seq)
        str = fmap qcvalue xs
    in (Compose (unit (evaluate (substring str) seq))) === (evaluate (prefix str) tree)
```

The type QCValue and the class QCValues provide a finite list of values which is used to generate test cases.

```
newtype QCValue a = QCValue { qcvalue :: a } deriving (Eq, Show)

class QCValues a where
    values :: [a]

instance (QCValues a) ⇒ Arbitrary (QCValue a) where
    arbitrary = oneof (fmap (return . QCValue) values)
```

For our example we restrict ourselves to the four characters *"a"*, *"b"*, *"c"* and *"d"*.

[2]We use the isnull function only for QuickCheck, but do not require it generally. Hence the grave consequences discussed before do not matter.

```
instance QCValues Char where
    values = "abcd"
```

Finally, we can apply QuickCheck to exemplarily test our algorithm for our tree example. We can similarly check the correctness for any sequence which uses a monad with equatable uncertainties.

```
> quickCheck (checksuffixtree example)
+++ OK, passed 100 tests.
```

5.3.2 Space Consumption

By using a suffix tree to calculate an alignment one gains a much better time performance if aligning multiple relatively short sequences. But this comes at the price of an expensive suffix tree construction before the first alignment. For a frequently needed reference sequence this can be done once and the suffix tree can be stored for later use. Hence the time performance of the construction algorithm is not primarily important. The space consumption of the tree on the other hand is more interesting and we want to make some notes regarding this now.

For the subsequent considerations we use two test strings over the alphabet *"a"*, *"b"*, *"c"*:

```
string1 :: [Char]
string1 = repeat 'a'
        ≡ "aaaaaaaa..."

string2 :: [Char]
string2 = getlist 1 where getlist n = ("abc" >>= (replicate n)) ++ (getlist (n + 1))
        ≡ "abcaabbccaaabbbccc..."
```

One way to turn strings into uncertain sequences is to annotate each value with a probability. The actual probabilities do not matter now, so we just annotate with probability 1.0:

```
annotate :: [Char] → MonadicList ((,) Prob) Char
annotate = MList . (fmap unit)
```

The more interesting monads in the context of space consumption are the ones allowing a sequence to fork, i.e. having multiple successor sequences. The following function takes a string and turns it into an uncertain sequence with periodic entirely uncertain positions where all letters of the alphabet are possible:

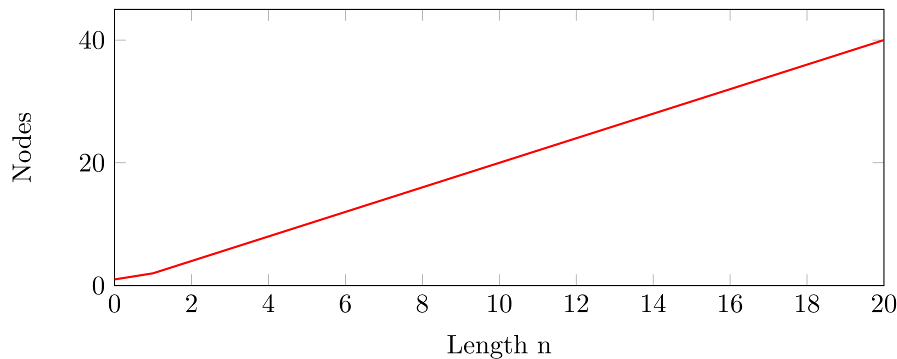
```
disturb :: Int → [Char] → MonadicList Indet Char
disturb n = MList . (getdisturbed n) where
    getdisturbed _ [] = []
    getdisturbed 0 (x:xs) = (Indet "abc") : (getdisturbed n xs)
    getdisturbed i (x:xs) = (unit x) : (getdisturbed (i−1) xs)
```

A proper asymptotic measure of space consumption for suffix trees is the number of inner nodes and leaves. For the two special cases of suffix trees to these monadic list we can implement this measure by deconstructing the tree and the monadic values:

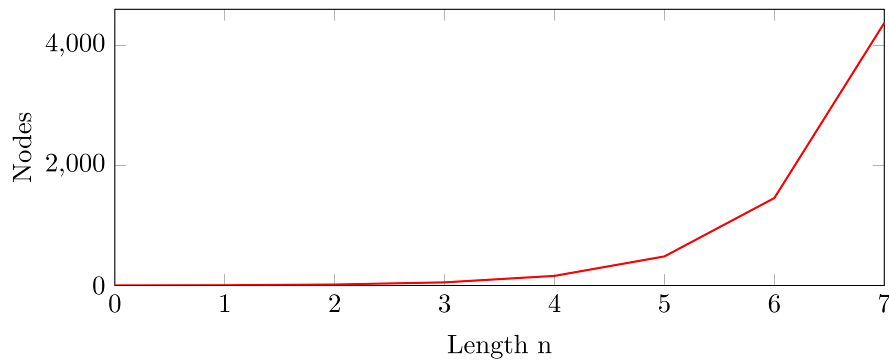
```
treesize1 :: Tree (Indet ∘ ((,) Prob)) a → Int
treesize1 (Tree br) = 1 +
                      sum (indetvalue (fmap snd (decompose (fmap (treesize1 . snd) br))))
treesize1 Leaf = 1

treesize2 :: Tree (Indet ∘ Indet) a → Int
treesize2 (Tree br) = 1 + sum (indetvalue (join (decompose (fmap (treesize2 . snd) br))))
treesize2 Leaf = 1
```

First we calculate the size of the suffix tree for substrings of various length of our first test string without uncertainty via disturb (-1) (take n string1):



This perfectly matches our expectations as the suffix tree for the test string in every inner position either ends with an *"a"* or continues with an *"a"*. So we get a linearly growing size with slope 2. Now we do the same but annotate the values (via annotate (take n string1)):

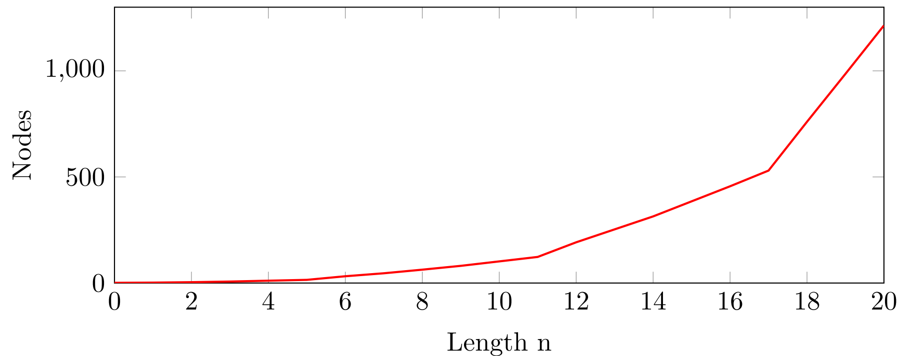


One might expect that the annotation does not cause a significant change, but obviously it does. The reason is that the impossible values are not "empty" as they are in the case of e.g. the option monad or the indeterminism monad. So the algorithm even calculates successor trees of

impossible branches as long as there is an impossible non-empty suffix left. The result is an entire tree of height n with three leaves and three successor trees for every inner node (one for each of the three letters of the alphabet).

A way to solve this problem is to require a way to treat impossible values differently from other uncertain values. The function isnull :: m () → Bool definitely enables the distinction, but we already mentioned in section 4.1 that we want to avoid such a function. A weaker form might be possible, such as a conditional bind condbind :: m a → (a → m b) → (a → m b) → m b applying the first Kleisli morphism if the value is a impossible value and the second else.

Next we look at the more interesting second test string with every sixth value turned to an indeterministic value (via disturb 5 (take n string2)):



We see that the slope increases every sixth step of length. This is of course due to the indeterministic value every sixth item in the sequence. The problem is that it increases exponentially and so a sequence with only a few ambiguous successors results in a vast space consumption. A suffix tree usually is highly branched on the upper levels near the root but in the larger part of lower levels there are long isolated branches representing only one suffix. So while branches and consequently ambiguities often are aggregated in the upper part, they do not vanish in the lower part. This causes the exponential growth of nodes.

In the general case we cannot improve the situation as indeterminism yields an exponentially increasing amount of possibilities. But, for instance, if we have monadic lists, we know about the independency of the items. In the case of an isolated branch holding only a single suffix in the lower part of the tree we can use this fact. We need a function issingleton :: m a → Bool[3] which tells us when there is only one suffix left in the monadic value of suffixes. From that point on we do not have to construct a tree anymore but can just use the underlying single sequence as a branch. This preserves the more efficient representation of a monadic list.

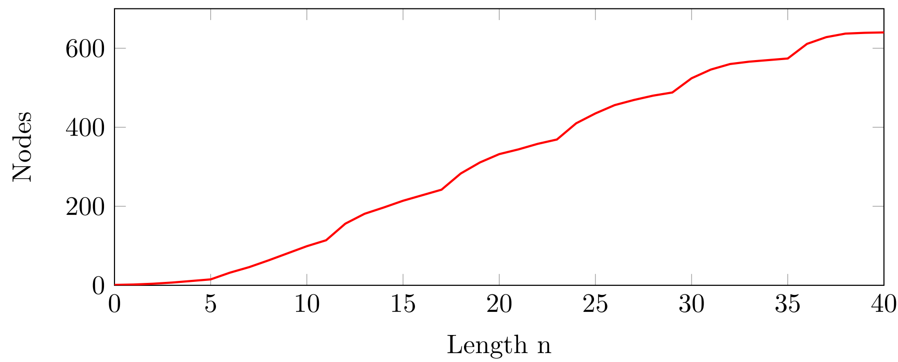
But beside of this idea of optimization the exponential growth might not matter very much in practice. An advantage of using a functional language like Haskell is that we can make use of lazy evaluation. This improves the current situation as large parts of the lower levels of the suffix tree might never actually have to be constructed, particularly if we want to align only short strings. To simulate a potentially more relevant behavior we use the following new measure which counts the nodes only up to a certain depth:

[3] Again a weaker form might be sufficient, e.g. a conditional bind as we had above for the isnull function.

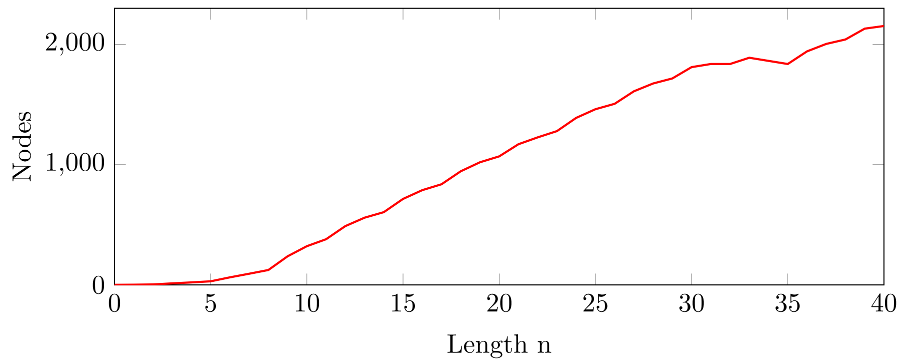
```
treesize3 :: Int → Tree (Indet ∘ Indet) a → Int
treesize3 0 _ = 0
treesize3 n (Tree br) = 1 +
                    sum (indetvalue (join (decompose (fmap ((treesize3 (n−1)) . snd) br))))
treesize3 n Leaf = 1
```

The following diagrams show that the situation changes significantly. In particular, there is an upper bound if the tree is fully branched up to the considered depth. But of course this bound itself increases exponentially. (f: distance between indeterministic values, d: measure depth)

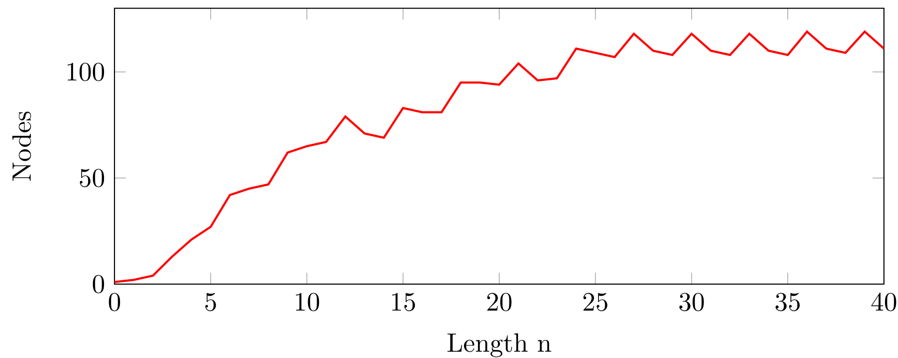
Parameters: $f = 6, d = 10$



Parameters: $f = 3, d = 10$



Parameters: $f = 3, d = 5$



6 Related Work

Some of the monads we presented are frequently used to theoretically describe or practically implement frameworks to describe uncertainty. In the following we exemplarily depict two approaches and afterwards point out the difference to our work.

Erwig and Kollmansberger [4] develop a domain-specific embedded language for probabilistic programming in Haskell. Their language is based on the probability distribution monad. Additionaly, they combine it with the IO monad to allow sampling in cases where the full calculation of the modeled distributions is too expensive. It is an example of a language extension using some specific monads as foundation to provide a simple interface to describe a complex probabilistic system. In collaboration with biologists they used this language to successfully develop a model for genome evolution.

Gibbons [5] presents the topic of embedded languages based on monads from a more theoretic point of view. He shows how algebraic theories of computational effects can be embedded to functional languages like Haskell using monads rather than having to define a language extension. By defining suitable type classes he gives the examples of indeterministic and probabilistic choice and failing computations. He also discusses the matter of combining the theories of indeterminism and probability distribution.

The main difference to our work is that we do not aim to describe systems with inherent uncertainty. In our attempt the entire uncertainty is given in the input data. Instead of giving ways to effectively "create" and "manipulate" uncertainty we focus on means to properly work with given uncertainty. Monads enable us to work with such uncertain data in a uniform manner, i.e. without actually caring about any specific uncertainty, particularly as the case of no uncertainty is always a special case.

The more static treatment of uncertainty makes it possible to cover a wider range of uncertainties. Many of the approaches similar to the two we mentioned above either use explicitly one single monad, e.g. the probability distribution monad, or they give more general type classes which are nevertheless often motivated by one specific monad instance, e.g. the class of indeterministic choice with its prototype, the list monad. So if the creation and manipulation of uncertainty is not necessarily required, our approach offers a more general alternative description.

6.1 The Haskell Classes

Haskell provides an implementation of functors, monads, and a definition similar to our monad with null and fully additive monads in the module Prelude resp. Control.Monad. We want to briefly state similarities and differences.

Functor

The class Control.Monad.Functor obviously is equivalent to our functor class. The functor laws (F1), (F2) are the same for Control.Monad.Functor. For our implementation we actually use the predefined functor class.

Monad

The class Control.Monad.Monad strictly speaking is a superclass of our monad class. But every instance of Control.Monad.Monad can be turned into an instance of our monad class:

```
instance (Control.Monad.Monad m) ⇒ Functor m where
    fmap f mx = (Control.Monad.≫=) mx (Control.Monad.return . f)

instance (Control.Monad.Monad m) ⇒ Monad m where
    unit = Control.Monad.return
    lift = flip (Control.Monad.≫=)
```

The monad laws are all satisfied for this instance.

MonadPlus

The class Control.Monad.MonadPlus is a mixture of several of our classes. It can be turned into a monad with null satisfying all conditions via:

```
instance (Control.Monad.MonadPlus m) ⇒ MonadWithNull m where
    mnull = const (Control.Monad.mzero)
```

The Control.Monad.mzero mainly acts as the neutral element to Control.Monad.mplus. In particular, it is unique and hence independent from values of the underlying type. We understand the null operator in the way that it wraps up a value into an impossible uncertain value. So in general it is not independent from values of the underlying type. Our interpretation of the null operator is not related to any additive structure in the first place.

We can also make the class Control.Monad.MonadPlus an instance of our fully additive functor or any of our weaker forms of additive functors (after turning the monad to a functor as described above):

```
instance (Control.Monad.MonadPlus m) ⇒ FullyAdditive m where
    plus = (Control.Monad.mplus)
```

7 Conclusion

In this work we have presented an approach to model uncertain data based on monads, and we showed how this approach can be applied to the problem of sequence alignment. The work was motivated by various situations in the context of sequencing in bioinformatics which exhibit different kinds of uncertainty. Consequently, we aimed to not be focused on a specific kind of uncertainty and sequence structure. The result is a general framework for reasoning about uncertainty and in particular uncertain sequences. It is based on the concept of monads. The considerations in this work have shown that, given some minor additional structure, monads provide a uniform description of different kinds of uncertainty. We also tried to confirm that the additional requirements we demand are minimal and include many of the well-known monad examples.

Our sequence class gives a way to implement different kinds of uncertain sequences by both varying the underlying monad, i.e. uncertainty, and the sequence-specific structure. We mainly presented simple implementations but we also saw that the suffix tree of a sequence can be described as a sequence again. It remains to be seen whether this interface has the expressiveness to describe all desired kinds of complex uncertain sequences. But with a little effort it is certainly possible to implement sequences that cover most of the examples presented at the beginning of the work. The examples of sequences we gave in this work support this assumption since, although they are based on simple sequence implementations, they are yet able to cover interesting situations of uncertainty in sequences.

We always put emphasis on the fact that the classic case of definite data itself is a special kind of uncertainty, namely the absence of any vagueness. This is the reason why the general case tends to be very similar to the classic case. Solving a problem with uncertainty hence does not require to completely start from the beginning but instead rather requires to carefully think about the basic conceptual structure behind an algorithm. In the example of a suffix tree algorithm we have seen how this led to some interesting insights:

- The suffix tree algorithm generally can be described as taking a sequence and returning again a sequence based on the former. More precisely, the algorithm transforms an uncertain sequence to a sequence representing all suffixes concurrently. This implies the modification of the uncertainty in the way that the uncertainty of indeterminism is added. In particular, a suffix tree can only be constructed if the uncertainty allows a proper combination with indeterminism.
- The construction algorithm for a suffix tree revealed a normalization step which was not at all obvious when looking at the classic algorithm. This unexpected fact shows that the attempt to generalize an algorithm to the setting of uncertainty leads to a deeper understanding of its inner mechanics and might expose surprising insights. So even without the actual intention to model uncertain data it can be worth thinking about uncertainty when examining data structures or algorithms.

However, both facts also underline the result that the transformation to the context of uncertainty is not a simple and quickly done procedure. It requires a careful and extensive analysis of the problem and the algorithm, and it possibly reveals steps requiring additional structure which are not foreseen. For the suffix tree we expected the need of composability with the indeterminism monad to describe the tree as a sequence again, but we did not expect the necessity of the normalization property. Yet we think that the new insights and the resulting more general algorithm compensate for the potentially elaborate analysis.

A great benefit of using monads and monadic sequences is their modularity. Our suffix tree algorithm does not need to be rewritten to apply to more complex examples or unexpected future situations. Instead, one needs to think about the specific uncertainty to be model and about its integration in a suitable sequence instance. If the new constructs fulfill the requirements of our algorithm this both indicates that it is possible to construct a suffix tree for this kind of uncertain sequence and simultaneously one actually can apply the algorithm without any additional work. So work can be done in a modularized fashion, i.e. one does not have to care about the big picture of the sequence alignment process.

In conclusion we can say that our modeling approach proved to be a successful attempt to handle uncertainties, at least in theory. It remains to be shown whether it will also prove a proper concept to be applied to our motivating examples in practice. It may well be the case that it is necessary to introduce additional functionality to achieve a better performance. For instance, simplifications for the list modeling the set monad might be possible, or ways to abort computations on branches which have become very unlikely, e.g. very low probability in the case of the distribution monad. But these optimization steps should be possible to integrate without significantly changing the foundations of our modeling approach.

8 Appendix

In the following we will proof the relations between the properties of the composing functors/-monads and of the composition which we stated in section 2.3.

Endofunctor

The composition $\mathrm{M}\circ\mathrm{N}$ of two endofunctors M and N over a category $\mathcal{C}$ is given by $A \mapsto \mathrm{MN}A$ and $(f\colon A \to B) \mapsto (\mathrm{M}(\mathrm{N}f)\colon \mathrm{MN}A \to \mathrm{MN}B$. The functor laws hold for any $f\colon A \to B, g\colon B \to C$:

$$\text{(F1)} \qquad (\mathrm{M}\circ\mathrm{N})(\mathrm{id}_A) \overset{\text{(Def)}}{=} \mathrm{M}(\mathrm{N}\,\mathrm{id}_A) \overset{\text{(F1)}}{=} \mathrm{M}\,\mathrm{id}_{\mathrm{N}A} \overset{\text{(F1)}}{=} \mathrm{id}_{\mathrm{MN}A} \overset{\text{(Def)}}{=} \mathrm{id}_{(\mathrm{M}\circ\mathrm{N})A}$$

$$\text{(F2)} \qquad (\mathrm{M}\circ\mathrm{N})(g\circ f) \overset{\text{(Def)}}{=} \mathrm{M}(\mathrm{N}(g\circ f)) \overset{\text{(F2)}}{=} \mathrm{M}(\mathrm{N}g\circ\mathrm{N}f) \overset{\text{(F2)}}{=} \mathrm{M}(\mathrm{N}g)\circ\mathrm{M}(\mathrm{N}f) \overset{\text{(Def)}}{=} (\mathrm{M}\circ\mathrm{N})(g)\circ(\mathrm{M}\circ\mathrm{N})(f)$$

Monad

Let M and N be two monads over a category $\mathcal{C}$. If we have a distributive law $\lambda\colon \mathrm{NM} \to \mathrm{MN}$, then composition $\mathrm{M}\circ\mathrm{N}$ with $\eta^{\mathrm{M}\circ\mathrm{N}}_A := \eta^{\mathrm{M}}_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_A$ and $\mu^{\mathrm{M}\circ\mathrm{N}}_A := \mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A}$ is again a monad. The monad laws hold for any $f\colon A\to B$:

$$\text{(M1)} \qquad (\mathrm{M}\circ\mathrm{N})(f)\circ\eta^{\mathrm{M}\circ\mathrm{N}}_A \overset{\text{(Def)}}{=} \mathrm{M}(\mathrm{N}f)\circ\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_A \overset{\text{(M1)}}{=} \eta^{\mathrm{M}}_{\mathrm{N}B}\circ\mathrm{N}f\circ\eta^{\mathrm{N}}_A \overset{\text{(M1)}}{=} \eta^{\mathrm{M}}_{\mathrm{N}B}\circ\eta^{\mathrm{N}}_B\circ f \overset{\text{(Def)}}{=} \eta^{\mathrm{M}\circ\mathrm{N}}_B\circ f$$

$$\begin{aligned}
\text{(M2)} \qquad & (\mathrm{M}\circ\mathrm{N})(f)\circ\mu^{\mathrm{M}\circ\mathrm{N}}_A \overset{\text{(Def)}}{=} \mathrm{M}(\mathrm{N}f)\circ\mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A} \overset{\text{(F2)}}{=} \mathrm{M}(\mathrm{N}f\circ\mu^{\mathrm{N}}_A)\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A} \\
& \overset{\text{(M2)}}{=} \mathrm{M}(\mu^{\mathrm{N}}_B\circ\mathrm{N}(\mathrm{N}f))\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A} \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_B\circ\mathrm{M}(\mathrm{N}(\mathrm{N}f))\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A} \\
& \overset{\text{(M2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_B\circ\mu^{\mathrm{M}}_{\mathrm{NN}B}\circ\mathrm{M}(\mathrm{M}(\mathrm{N}(\mathrm{N}f)))\circ\mathrm{M}\lambda_{\mathrm{N}A} \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_B\circ\mu^{\mathrm{M}}_{\mathrm{NN}B}\circ\mathrm{M}(\mathrm{M}(\mathrm{N}(\mathrm{N}f))\circ\lambda_{\mathrm{N}A}) \\
& \overset{\text{(D1)}}{=} \mathrm{M}\mu^{\mathrm{N}}_B\circ\mu^{\mathrm{M}}_{\mathrm{NN}B}\circ\mathrm{M}(\lambda_{\mathrm{N}B}\circ\mathrm{N}(\mathrm{M}(\mathrm{N}f))) \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_B\circ\mu^{\mathrm{M}}_{\mathrm{NN}B}\circ\mathrm{M}\lambda_{\mathrm{N}B}\circ\mathrm{M}(\mathrm{N}(\mathrm{M}(\mathrm{N}f))) \\
& \overset{\text{(Def)}}{=} \mu^{\mathrm{M}\circ\mathrm{N}}_B\circ(\mathrm{M}\circ\mathrm{N})((\mathrm{M}\circ\mathrm{N})(f))
\end{aligned}$$

$$\begin{aligned}
\text{(M3)} \qquad & \mu^{\mathrm{M}\circ\mathrm{N}}_A\circ\eta^{\mathrm{M}\circ\mathrm{N}}_{(\mathrm{M}\circ\mathrm{N})A} \overset{\text{(Def)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A}\circ\eta^{\mathrm{M}}_{\mathrm{NMN}A}\circ\eta^{\mathrm{N}}_{\mathrm{MN}A} \\
& \overset{\text{(M1)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\eta^{\mathrm{M}}_{\mathrm{MNN}A}\circ\lambda_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_{\mathrm{MN}A} \overset{\text{(M3)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mathrm{id}_{\mathrm{MNN}A}\circ\lambda_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_{\mathrm{MN}A} \overset{\text{(D2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mathrm{M}\eta^{\mathrm{N}}_{\mathrm{N}A} \\
& \overset{\text{(F2)}}{=} \mathrm{M}(\mu^{\mathrm{N}}_A\circ\eta^{\mathrm{N}}_{\mathrm{N}A}) \overset{\text{(M3)}}{=} \mathrm{M}\,\mathrm{id}_{\mathrm{N}A} \overset{\text{(F1)}}{=} \mathrm{id}_{\mathrm{MN}A} \overset{\text{(Def)}}{=} \mathrm{id}_{(\mathrm{M}\circ\mathrm{N})A}
\end{aligned}$$

$$\begin{aligned}
\text{(M4)} \qquad & \mu^{\mathrm{M}\circ\mathrm{N}}_A\circ(\mathrm{M}\circ\mathrm{N})(\eta^{\mathrm{M}\circ\mathrm{N}}_A) \overset{\text{(Def)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A}\circ\mathrm{M}(\mathrm{N}(\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_A)) \\
& \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}(\lambda_{\mathrm{N}A}\circ\mathrm{N}(\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\eta^{\mathrm{N}}_A)) \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}(\lambda_{\mathrm{N}A}\circ\mathrm{N}\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\mathrm{N}\eta^{\mathrm{N}}_A) \\
& \overset{\text{(D3)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}(\eta^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{N}\eta^{\mathrm{N}}_A) \overset{\text{(M2)}}{=} \mu^{\mathrm{M}}_{\mathrm{N}A}\circ\mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_A)\circ\mathrm{M}(\eta^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{N}\eta^{\mathrm{N}}_A) \\
& \overset{\text{(F2)}}{=} \mu^{\mathrm{M}}_{\mathrm{N}A}\circ\mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_A\circ\eta^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{N}\eta^{\mathrm{N}}_A) \overset{\text{(M1)}}{=} \mu^{\mathrm{M}}_{\mathrm{N}A}\circ\mathrm{M}(\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\mu^{\mathrm{N}}_A\circ\mathrm{N}\eta^{\mathrm{N}}_A) \overset{\text{(M4)}}{=} \mu^{\mathrm{M}}\circ\mathrm{M}(\eta^{\mathrm{M}}_{\mathrm{N}A}\circ\mathrm{id}_{\mathrm{N}A}) \\
& \overset{\text{(M4)}}{=} \mathrm{id}_{\mathrm{MN}A} \overset{\text{(Def)}}{=} \mathrm{id}_{(\mathrm{M}\circ\mathrm{N})A}
\end{aligned}$$

$$\begin{aligned}
\text{(M5)} \qquad & \mu^{\mathrm{M}\circ\mathrm{N}}_A\circ(\mathrm{M}\circ\mathrm{N})(\mu^{\mathrm{M}\circ\mathrm{N}}_A) \overset{\text{(Def)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A}\circ\mathrm{M}(\mathrm{N}(\mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}\lambda_{\mathrm{N}A})) \\
& \overset{\text{(F2)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}(\lambda_{\mathrm{N}A}\circ\mathrm{N}(\mathrm{M}\mu^{\mathrm{N}}_A)\circ\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A})) \\
& \overset{\text{(D1)}}{=} \mathrm{M}\mu^{\mathrm{N}}_A\circ\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{M}(\mathrm{M}(\mathrm{N}\mu^{\mathrm{N}}_A)\circ\lambda_{\mathrm{NN}A}\circ\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}\circ\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))
\end{aligned}$$

$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}(\mathrm{N}\mu^{\mathrm{N}}_{A})) \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(M2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mathrm{MN}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(F2)}{=} \mathrm{M}(\mu^{\mathrm{N}}_{A} \circ \mathrm{N}\mu^{\mathrm{N}}_{A}) \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(M5)}{=} \mathrm{M}(\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{N}}_{\mathrm{N}A}) \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A} \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(M2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A}) \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A}) \circ \mathrm{M}(\mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A} \circ \lambda_{\mathrm{NN}A} \circ \mathrm{N}\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(D4)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A} \circ \mu^{\mathrm{M}}_{\mathrm{NNN}A} \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \lambda_{\mathrm{MNN}A} \circ \mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(M2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A}) \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \lambda_{\mathrm{MNN}A} \circ \mathrm{N}(\mathrm{M}\lambda_{\mathrm{N}A}))$$
$$\overset{(D1)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A}) \circ \mathrm{M}\lambda_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{N}\lambda_{\mathrm{N}A}) \circ \lambda_{\mathrm{NMN}A})$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{\mathrm{N}A} \circ \lambda_{\mathrm{NN}A} \circ \mathrm{N}\lambda_{\mathrm{N}A}) \circ \lambda_{\mathrm{NMN}A})$$
$$\overset{(D5)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\lambda_{\mathrm{N}A} \circ \mu^{\mathrm{N}}_{\mathrm{MN}A}) \circ \lambda_{\mathrm{NMN}A})$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}\mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}(\lambda_{\mathrm{N}A} \circ \mu^{\mathrm{N}}_{\mathrm{MN}A}) \circ \mathrm{M}\lambda_{\mathrm{NMN}A}$$
$$\overset{(M5)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mu^{\mathrm{M}}_{\mathrm{MNN}A} \circ \mathrm{M}(\mathrm{M}(\lambda_{\mathrm{N}A} \circ \mu^{\mathrm{N}}_{\mathrm{MN}A}) \circ \mathrm{M}\lambda_{\mathrm{NMN}A}$$
$$\overset{(M2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\lambda_{\mathrm{N}A} \circ \mu^{\mathrm{N}}_{\mathrm{MN}A}) \circ \mu^{\mathrm{M}}_{\mathrm{MNNN}A} \circ \mathrm{M}\lambda_{\mathrm{NMN}A}$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}\lambda_{\mathrm{N}A} \circ \mathrm{M}\mu^{\mathrm{N}}_{\mathrm{MN}A} \circ \mu^{\mathrm{M}}_{\mathrm{MNNN}A} \circ \mathrm{M}\lambda_{\mathrm{NMN}A} \overset{(Def)}{=} \mu^{\mathrm{M}\circ\mathrm{N}}_{A} \circ \mu^{\mathrm{M}\circ\mathrm{N}}_{(\mathrm{M}\circ\mathrm{N})A}$$

Monad with Null

Given that two monads M and N over a category $\mathcal{C}$ can be composed to a monad $\mathrm{M} \circ \mathrm{N}$ using a distributive law $\lambda\colon \mathrm{NM} \to \mathrm{MN}$, this monad is a monad with null if the outer monad M is a monad with null by $\nu^{\mathrm{M}\circ\mathrm{N}} := \nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{A}$. The laws for a monad with null hold for any $f\colon A \to B, g\colon A \to \mathrm{MN}A$:

(N1) $$(\mathrm{M} \circ \mathrm{N})(f) \circ \nu^{\mathrm{M}\circ\mathrm{N}}_{A} \overset{(Def)}{=} \mathrm{M}(\mathrm{N}f) \circ \nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{A} \overset{(N1)}{=} \nu^{\mathrm{M}}_{\mathrm{N}B} \circ \mathrm{N}f \circ \eta^{\mathrm{N}}_{A} \overset{(M1)}{=} \nu^{\mathrm{M}}_{\mathrm{N}B} \circ \eta^{\mathrm{N}}_{B} \circ f \overset{(Def)}{=} \nu^{\mathrm{M}\circ\mathrm{N}}_{B} \circ f$$

(N2) $$\mu^{\mathrm{M}\circ\mathrm{N}}_{A} \circ \nu^{\mathrm{M}\circ\mathrm{N}}_{(\mathrm{M}\circ\mathrm{N})A} \circ g \overset{(Def)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}\lambda_{\mathrm{N}A} \circ \nu^{\mathrm{M}}_{\mathrm{NMN}A} \circ \eta^{\mathrm{N}}_{\mathrm{MN}A} \circ g$$
$$\overset{(N1)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \nu^{\mathrm{M}}_{\mathrm{MNN}A} \circ \lambda_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{\mathrm{MN}A} \circ g \overset{(M2)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}(\mathrm{M}\mu^{\mathrm{N}}_{A}) \circ \nu^{\mathrm{M}}_{\mathrm{MNN}A} \circ \lambda_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{\mathrm{MN}A} \circ g$$
$$\overset{(N1)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \nu^{\mathrm{M}}_{\mathrm{MN}A} \circ \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \lambda_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{\mathrm{MN}A} \circ g \overset{(M1)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \nu^{\mathrm{M}}_{\mathrm{MN}A} \circ [\mathrm{M}\mu^{\mathrm{N}}_{A} \circ \lambda_{\mathrm{N}A} \circ \mathrm{N}g] \circ \eta^{\mathrm{N}}_{A}$$
$$\overset{(N2)}{=} \nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{A} \overset{(Def)}{=} \nu^{\mathrm{M}\circ\mathrm{N}}_{A}$$

(N3) $$\mu^{\mathrm{M}\circ\mathrm{N}}_{A} \circ (\mathrm{M} \circ \mathrm{N})(\nu^{\mathrm{M}\circ\mathrm{N}}_{A}) \circ g \overset{(Def)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}\lambda_{\mathrm{N}A} \circ \mathrm{M}(\mathrm{N}(\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{A})) \circ g$$
$$\overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\lambda_{\mathrm{N}A} \circ \mathrm{N}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{N}\eta^{\mathrm{N}}_{A}) \circ g \overset{(D6)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\nu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{N}\eta^{\mathrm{N}}_{A}) \circ g$$
$$\overset{(N1)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}(\mathrm{N}\eta^{\mathrm{N}}_{A}) \circ \nu^{\mathrm{M}}_{\mathrm{N}A}) \circ g \overset{(F2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mu^{\mathrm{M}}_{\mathrm{NN}A} \circ \mathrm{M}(\mathrm{M}(\mathrm{N}\eta^{\mathrm{N}}_{A})) \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ g$$
$$\overset{(M2)}{=} \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mathrm{M}(\mathrm{N}\eta^{\mathrm{N}}_{A}) \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ g \overset{(F2)}{=} \mathrm{M}(\mu^{\mathrm{N}}_{A} \circ \mathrm{N}\eta^{\mathrm{N}}_{A}) \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ g$$
$$\overset{(M4)}{=} \mathrm{M}\,\mathrm{id}_{\mathrm{N}A} \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ g \overset{(F1)}{=} \mathrm{id}_{\mathrm{MN}A} \circ \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ g \overset{(Id)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{id}_{\mathrm{MN}A} \circ g$$
$$\overset{(F1)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\,\mathrm{id}_{\mathrm{N}A} \circ g \overset{(M3)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}(\mu^{\mathrm{N}}_{A} \circ \eta^{\mathrm{N}}_{\mathrm{N}A}) \circ g$$
$$\overset{(F2)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \mathrm{M}\eta^{\mathrm{N}}_{\mathrm{N}A} \circ g \overset{(D2)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\mu^{\mathrm{N}}_{A} \circ \lambda_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{\mathrm{MN}A} \circ g$$
$$\overset{(M1)}{=} \mu^{\mathrm{M}}_{\mathrm{N}A} \circ \mathrm{M}\nu^{\mathrm{M}}_{\mathrm{N}A} \circ [\mathrm{M}\mu^{\mathrm{N}}_{A} \circ \lambda_{\mathrm{N}A} \circ \mathrm{N}g] \circ \eta^{\mathrm{N}}_{A} \overset{(N3)}{=} \nu^{\mathrm{M}}_{\mathrm{N}A} \circ \eta^{\mathrm{N}}_{A} \overset{(Def)}{=} \nu^{\mathrm{M}\circ\mathrm{N}}_{A}$$

Additive Functor

The composition of a strongly additive endofunctor M and an additive endofunctor N over a category $\mathcal{C}$ is an additive endofunctor $\mathrm{M} \circ \mathrm{N}$ by $+^{\mathrm{M} \circ \mathrm{N}} := +^{\mathrm{M}}$, which is well-defined and associative by definition. $\mathrm{M} \circ \mathrm{N}$ is a monad with null if M is a monad with null, so again by definition the null values $\nu_{\mathbb{1}}^{\mathrm{M} \circ \mathrm{N}}(x) = \nu_{\mathrm{N}\mathbb{1}}^{\mathrm{M}}(\eta_{\mathbb{1}}^{\mathrm{N}}(x))$ are neutral elements of $+^{\mathrm{M} \circ \mathrm{N}} = +^{\mathrm{M}}$ for $x \in \mathbb{1}$.

If N too is strongly additive, $\mathrm{M} \circ \mathrm{N}$ is strongly additive. If M is fully additive and N an arbitrary endofunctor, $\mathrm{M} \circ \mathrm{N}$ is fully additive. The argument is exactly the same, in the former case using that N together with the inner monad forms an additive endofunctor.

Measurable Functor

If we have a strongly measurable endofunctor M and a measurable and additive endofunctor N over a category $\mathcal{C}$, the composition $\mathrm{M} \circ \mathrm{N}$ is a measurable endofunctor via $\omega_A^{\mathrm{M} \circ \mathrm{N}} := \omega_{\mathrm{N};A}^{\mathrm{M}}$. The measurable functor laws hold:

$$\text{(U1)} \qquad \omega_A^{\mathrm{M} \circ \mathrm{N}} \circ \eta_A^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \omega_{\mathrm{N};A}^{\mathrm{M}} \circ \eta_{\mathrm{N}A}^{\mathrm{M}} \circ \eta_A^{\mathrm{N}} \overset{\text{(SU1)}}{=} \eta_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \circ \omega_A^{\mathrm{N}} \circ \eta_A^{\mathrm{N}} \overset{\text{(U1)}}{=} \eta_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \circ \eta_{\mathbb{1}}^{\mathrm{N}} \circ \tau_A \overset{\text{(Def)}}{=} \eta_A^{\mathrm{M} \circ \mathrm{N}} \circ \tau_A$$

$$\text{(U2)} \qquad \omega_A^{\mathrm{M} \circ \mathrm{N}} \circ \nu_A^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \omega_{\mathrm{N};A}^{\mathrm{M}} \circ \nu_{\mathrm{N}A}^{\mathrm{M}} \circ \eta_A^{\mathrm{N}} \overset{\text{(SU2)}}{=} \nu_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \circ \omega_A^{\mathrm{N}} \circ \eta_A^{\mathrm{N}} \overset{\text{(U1)}}{=} \nu_{\mathrm{N}\mathbb{1}}^{\mathrm{M}} \circ \eta_{\mathbb{1}}^{\mathrm{N}} \circ \tau_A \overset{\text{(Def)}}{=} \nu_A^{\mathrm{M} \circ \mathrm{N}} \circ \tau_A$$

If N is strongly measurable and strongly additive, $\mathrm{M} \circ \mathrm{N}$ is strongly measurable by $\omega_{\mathrm{K};A}^{\mathrm{M} \circ \mathrm{N}} := \omega_{\mathrm{N} \circ \mathrm{K};A}^{\mathrm{M}}$. The strongly measurable functor laws hold (using the definition of $\omega_A^{\mathrm{N} \circ \mathrm{K}}$):

$$\begin{aligned} \text{(SU1)} \qquad & \omega_{\mathrm{K};A}^{\mathrm{M} \circ \mathrm{N}} \circ \eta_{\mathrm{K}A}^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \omega_{\mathrm{N} \circ \mathrm{K};A}^{\mathrm{M}} \circ \eta_{\mathrm{NK}A}^{\mathrm{M}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(SU1)}}{=} \eta_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \omega_A^{\mathrm{N} \circ \mathrm{K}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(Def)}}{=} \eta_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \omega_{\mathrm{K};A}^{\mathrm{N}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \\ & \overset{\text{(SU1)}}{=} \eta_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \eta_{\mathrm{K}\mathbb{1}}^{\mathrm{N}} \circ \omega_A^{\mathrm{K}} \overset{\text{(Def)}}{=} \eta_{\mathrm{K}\mathbb{1}}^{\mathrm{M} \circ \mathrm{N}} \circ \omega_A^{\mathrm{K}} \end{aligned}$$

$$\begin{aligned} \text{(SU1)} \qquad & \omega_{\mathrm{K};A}^{\mathrm{M} \circ \mathrm{N}} \circ \nu_{\mathrm{K}A}^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \omega_{\mathrm{N} \circ \mathrm{K};A}^{\mathrm{M}} \circ \nu_{\mathrm{NK}A}^{\mathrm{M}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(SU2)}}{=} \nu_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \omega_A^{\mathrm{N} \circ \mathrm{K}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(Def)}}{=} \nu_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \omega_{\mathrm{K};A}^{\mathrm{N}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \\ & \overset{\text{(SU1)}}{=} \nu_{\mathrm{NK}\mathbb{1}}^{\mathrm{M}} \circ \eta_{\mathrm{K}\mathbb{1}}^{\mathrm{N}} \circ \omega_A^{\mathrm{K}} \overset{\text{(Def)}}{=} \nu_{\mathrm{K}\mathbb{1}}^{\mathrm{M} \circ \mathrm{N}} \circ \omega_A^{\mathrm{K}} \end{aligned}$$

Distributive Law

Let two monads M and N over a category $\mathcal{C}$ be given which compose to a monad $\mathrm{M} \circ \mathrm{N}$ using a distributive law $\lambda \colon \mathrm{NM} \to \mathrm{MN}$. Furthermore, let a third monad K over $\mathcal{C}$ and another two distributive laws $\lambda^{\mathrm{M}} \colon \mathrm{MK} \to \mathrm{KM}, \lambda^{\mathrm{N}} \colon \mathrm{NK} \to \mathrm{KN}$ be given. If these three distributive laws satisfy the Yang-Baxter equation (presented in section 3.2), then K and $\mathrm{M} \circ \mathrm{N}$ can be composed using the distributive law $\lambda_A^{\mathrm{M} \circ \mathrm{N}} := \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}}$. We prove (D1)-(D4) and (D6) but drop the proof of (D5). It can be found in [3].

$$\begin{aligned} \text{(D1)} \qquad & \mathrm{K}((\mathrm{M} \circ \mathrm{N})(f)) \circ \lambda_A^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \mathrm{K}(\mathrm{M}(\mathrm{N}f)) \circ \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}} \overset{\text{(D1)}}{=} \lambda_{\mathrm{N}B}^{\mathrm{M}} \circ \mathrm{M}(\mathrm{K}(\mathrm{N}f)) \circ \mathrm{M}\lambda_A^{\mathrm{N}} \\ & \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}B}^{\mathrm{M}} \circ \mathrm{M}(\mathrm{K}(\mathrm{N}f) \circ \lambda_A^{\mathrm{N}}) \overset{\text{(D1)}}{=} \lambda_{\mathrm{N}B}^{\mathrm{M}} \circ \mathrm{M}(\lambda_B^{\mathrm{N}} \circ \mathrm{N}(\mathrm{K}f)) \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}B}^{\mathrm{M}} \circ \mathrm{M}\lambda_B^{\mathrm{N}} \circ \mathrm{M}(\mathrm{N}(\mathrm{K}f)) \\ & \overset{\text{(Def)}}{=} \lambda_B^{\mathrm{M} \circ \mathrm{N}} \circ (\mathrm{M} \circ \mathrm{N})(\mathrm{K}f) \end{aligned}$$

$$\begin{aligned} \text{(D2)} \qquad & \lambda_A^{\mathrm{M} \circ \mathrm{N}} \circ \eta_{\mathrm{K}A}^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}} \circ \eta_{\mathrm{MK}A}^{\mathrm{M}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(M1)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \eta_{\mathrm{KN}A}^{\mathrm{M}} \circ \lambda_A^{\mathrm{N}} \circ \eta_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(D2)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \eta_{\mathrm{KN}A}^{\mathrm{M}} \circ \mathrm{K}\eta_A^{\mathrm{N}} \\ & \overset{\text{(D2)}}{=} \mathrm{K}\eta_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{K}\eta_A^{\mathrm{N}} \overset{\text{(F2)}}{=} \mathrm{K}(\eta_{\mathrm{N}A}^{\mathrm{M}} \circ \eta_A^{\mathrm{N}}) \overset{\text{(Def)}}{=} \mathrm{K}\eta_A^{\mathrm{M} \circ \mathrm{N}} \end{aligned}$$

$$\text{(D3)} \qquad \lambda_A^{\mathrm{M} \circ \mathrm{N}} \circ (\mathrm{M} \circ \mathrm{N})(\eta_A^{\mathrm{K}}) \overset{\text{(Def)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}} \circ \mathrm{M}(\mathrm{N}\eta_A^{\mathrm{K}}) \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}(\lambda_A^{\mathrm{N}} \circ \mathrm{N}\eta_A^{\mathrm{K}}) \overset{\text{(D3)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\eta_{\mathrm{N}A}^{\mathrm{K}} \overset{\text{(D3)}}{=} \eta_{\mathrm{MN}A}^{\mathrm{K}}$$

$$\begin{aligned} \text{(D4)} \qquad & \mu_{\mathrm{MN}A}^{\mathrm{K}} \circ \mathrm{K}\lambda_A^{\mathrm{M} \circ \mathrm{N}} \circ \lambda_{\mathrm{K}A}^{\mathrm{M} \circ \mathrm{N}} \overset{\text{(Def)}}{=} \mu_{\mathrm{MN}A}^{\mathrm{K}} \circ \mathrm{K}(\lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}}) \circ \lambda_{\mathrm{MK}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_{\mathrm{K}A}^{\mathrm{N}} \\ & \overset{\text{(F2)}}{=} \mu_{\mathrm{MN}A}^{\mathrm{K}} \circ \mathrm{K}\lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{K}(\mathrm{M}\lambda_A^{\mathrm{N}}) \circ \lambda_{\mathrm{MK}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(D1)}}{=} \mu_{\mathrm{MN}A}^{\mathrm{K}} \circ \mathrm{K}\lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \lambda_{\mathrm{KN}A}^{\mathrm{M}} \circ \mathrm{M}(\mathrm{K}\lambda_A^{\mathrm{N}}) \circ \mathrm{M}\lambda_{\mathrm{K}A}^{\mathrm{N}} \\ & \overset{\text{(D4)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\mu_{\mathrm{N}A}^{\mathrm{K}} \circ \mathrm{M}(\mathrm{K}\lambda_A^{\mathrm{N}}) \circ \mathrm{M}\lambda_{\mathrm{K}A}^{\mathrm{N}} \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}(\mu_{\mathrm{N}A}^{\mathrm{K}} \circ \mathrm{K}\lambda_A^{\mathrm{N}} \circ \lambda_{\mathrm{K}A}^{\mathrm{N}}) \overset{\text{(D4)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}(\lambda_A^{\mathrm{N}} \circ \mathrm{N}\mu_A^{\mathrm{K}}) \\ & \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}} \circ \mathrm{M}(\mathrm{N}\mu_A^{\mathrm{K}}) \overset{\text{(Def)}}{=} \lambda_A^{\mathrm{M} \circ \mathrm{N}} \circ (\mathrm{M} \circ \mathrm{N})(\mu_A^{\mathrm{K}}) \end{aligned}$$

$$\text{(D6)} \qquad \lambda_A^{\mathrm{M} \circ \mathrm{N}} \circ (\mathrm{M} \circ \mathrm{N})(\nu_A^{\mathrm{K}}) \overset{\text{(Def)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\lambda_A^{\mathrm{N}} \circ \mathrm{M}(\mathrm{N}\nu_A^{\mathrm{K}}) \overset{\text{(F2)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}(\lambda_A^{\mathrm{N}} \circ \mathrm{N}\nu_A^{\mathrm{K}}) \overset{\text{(D6)}}{=} \lambda_{\mathrm{N}A}^{\mathrm{M}} \circ \mathrm{M}\nu_{\mathrm{N}A}^{\mathrm{K}} \overset{\text{(D6)}}{=} \nu_{\mathrm{MN}A}^{\mathrm{K}}$$

Bibliography

[1] Michael Barr and Charles Wells. Toposes, triples and theories, 2005.

[2] Jon Beck. Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer-Verlag, 1969.

[3] Eugenia Cheng. Iterated distributive laws. Technical report, 2007.

[4] Martin Erwig and Steve Kollmansberger. Modeling genome evolution with a dsel for probabilistic programming. In *Proceedings of the 8th international conference on Practical Aspects of Declarative Languages*, PADL'06, pages 134–149. Springer-Verlag, 2006.

[5] Jeremy Gibbons. Unifying theories of programming with monads. In *UTP Symposium*, 2012.

[6] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, WAE '99, pages 30–42. Springer-Verlag, 1999.

[7] Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin Heidelberg, 1982.

[8] Claire Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 186–195. IEEE Press, 1989.

[9] Mark P. Jones and Luc Duponcheel. Composing monads. Technical report, 1993.

[10] David J. King and Philip Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143. Springer-Verlag, 1993.

[11] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 192–203. ACM, 2005.

[12] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343. ACM, 1995.

[13] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[14] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[15] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 472–492. ACM, 1994.

[16] Esko Ukkonen. On-line construction of suffix trees. In *Algorithmica*, volume 14, pages 249–260, 1995.

[17] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113, 2006.

[18] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78. ACM, 1990.

[19] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52. Springer-Verlag, 1995.

[20] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, SWAT '73, pages 1–11. IEEE Computer Society, 1973.