KLEISLI DATABASE INSTANCES

DAVID I. SPIVAK

Abstract. We use monads to relax the atomicity requirement for data in a database. Depending on the choice of monad, the database fields may contain generalized values such as lists or sets of values, or they may contain exceptions such as various types of nulls. The return operation for monads ensures that any ordinary database instance will count as one of these generalized instances, and the bind operation ensures that generalized values behave well under joins of foreign key sequences. Different monads allow for vastly different types of information to be stored in the database. For example, we show that classical concepts like Markov chains, graphs, and finite state automata are each perfectly captured by a different monad on the same schema.

Contents

1. Introduction

Monads are category-theoretic constructs with wide-ranging applications in both mathematics and computer science. In [Mog], Moggi showed how to exploit their expressive capacity to incorporate fundamental programming concepts into purely functional languages, thus considerably extending the potency of the functional paradigm. Using monads, concepts that had been elusive to functional programming, such as state, input/output, and concurrency, were suddenly made available in that context.

In the present paper we describe a parallel use of monads in databases. This approach stems from a similarity between categories and database schemas, as presented in [Sp1]. The rough idea is as follows. A database schema can be modeled as a category $\mathcal{C}$, and an ordinary database instance is a functor $\delta\colon \mathcal{C} \to \mathbf{Set}$. Given a monad $T\colon \mathbf{Set} \to \mathbf{Set}$, a *Kleisli T-instance* is a functor

$$\delta\colon \mathcal{C} \to \mathbf{Kls}(T),$$

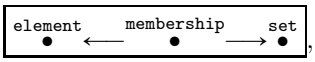
where $\mathbf{Kls}(T)$ is the Kleisli category of T, as will be explained in Section 2.2.

This project was supported by ONR grant N000141010841.

Values in a Kleisli T-instance are less restricted than ordinary values; we call these generalized values *T-values*. In particular, within a Kleisli instance we are permitted to relax the atomicity requirement for data (a requirement found in Codd's notion of *first normal form*, see [Cod]), while still maintaining referential integrity. For example, if T is the List monad then T-values are lists, so a single entry in a foreign key or data column could contain a list of entries of the target type. Similarly, T-values might include assurance information (i.e. a number between 0% and 100%), in which case each datum would come equipped with a probability of correctness. Importantly, the monadicity of T ensures that the extra information in T-values will naturally and predictably synthesize along any path obtained by joining a sequence of foreign keys. One can think of flattening lists of lists, of multiplying probabilities, or of propagating exceptions.

Kleisli instances offer additional functionality in a database, and such functionalities vary widely as the category of monads on **Set** is quite rich. Having a variety of available possibilities, the database architect can choose those that best fit the current needs. Moreover, a morphism between monads $T \to T'$ induces a functor from the category of T-instances to the category of T'-instances on the schema. In future work we will show that one can vary the choice of monad throughout the database schema, thus greatly increasing the expressive power of database schemas. By incorporating these features within the design specification of the database, as opposed to applying them from without, we reduce the barrier between database and program. Whereas normally such functionality is distributed throughout the technology stack, the monadic approach leads to a centralization of features, increasing our ability to manage the system with certainty.

The monad formalism also enables more economical schema design. For example, typically one encodes a set-membership relation with three tables, e.g.



and to encode that A has as an attribute a list of B's requires requires even more overhead. However, the same information can be captured with a single column when one employs the Multiset or List monad.

As an aside, the monad formalism also yields a surprising coincidence. We show that there is a database schema $\mathcal{L}oop$ such that, for different choices of monads T, the set of Kleisli T-instances on $\mathcal{L}oop$ can be interpreted in terms of classical mathematical subjects.

(1)

Classical mathematical subject	**monad T**	**Internal reference**
Discrete dynamical systems	Atomic	Example 2.1.3
Graphs	Multiset	Example 4.2.7
Markov chains	Dist	Example 4.2.9
Finite state automata	Inp^U	Example 4.3.8
Turing machines	$\mathrm{Tur}^{\{0,1\}}_{\{L,R,W_0,W_1\}}$	Example 4.3.13
"Jordan Canonical Form" (vector spaces with endomorphism)	Vect	Example 4.4.2
Multigraphs	Free rig	Example 4.4.4

Monads have been applied to databases in previous work (e.g. [BNT], [Gru], [LT], and [Maj])), but the sense in which they are applied is totally different than that which is presented in this paper. In each of these papers, monads were applied to make sense of queries and, in particular, aggregate functions on collections (e.g. counts and sums). The present paper, on the other hand, deals with the employment of monads within the database schema to provide additional expressivity in each field, e.g. allowing non-atomic data or annotating data with probability of correctness. While previous work may simplify aggregation in our context, it should be seen as orthogonal to the ideas presented here.

In this paper we assume the reader has encountered categories before, but it is not totally necessary. Readers with either very much or very little category theory may benefit by reading Section 2.1 and Example 2.2.2 and then skipping directly to Section 4. Readers with some background but who wish to review monads or their Kleisli categories will hopefully be satisfied with the brief overview in the intermediate sections. For a good reference on category theory, and monads in particular, one should consult [Awo] or [BW].

We begin this paper in Section 2 with a brief review of the categorical model of databases, as well as some background on monads and their Kleisli categories. We discuss a new application of monads to databases in Section 3. In section 4 we offer several examples that may be of interest, such as the List-instances. In Section 5 we discuss morphisms of monads, which for example allow one to transform ordinary atomic instances into List-instances. Finally in Section 6 we briefly discuss our plans for future work in this area.

1.1. **Acknowledgements.** I'd like to thank Steve Lack and Tom Leinster for their excellent answers to a question I posted on mathoverflow.net, and I'd like to thank Allen Brown, Peter Gates, and Ka Yu Tam for many useful conversations.

2. Background

In this section we recount a simple category-theoretic model of databases, and then review basic material on monads.

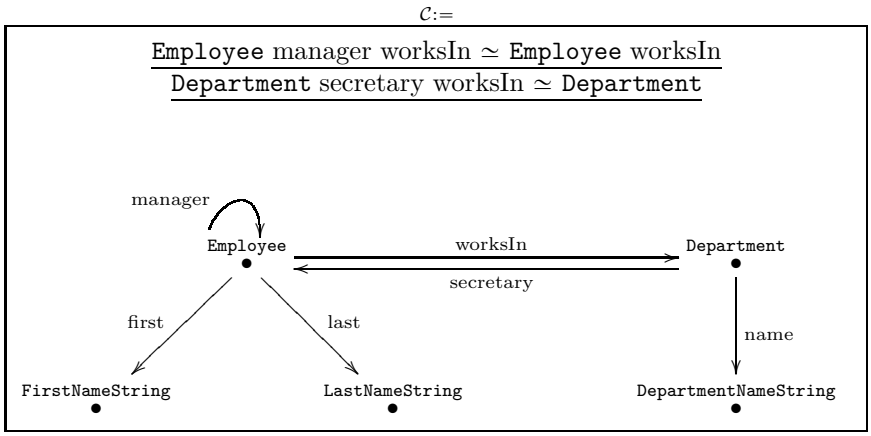
Notation 2.0.1. Let **Set** denote the category whose objects are sets and whose morphisms are functions. Throughout the paper we will be careful to reserve the word *function* to refer to mappings between sets. In a general category $\mathcal{K}$ we use words like *arrow* or *map*, but never function, to refer to morphisms in $\mathcal{K}$.

2.1. **Categorical databases.** We begin with some background on so-called categorical databases. Much more can be found in [Sp1].

Roughly, a database *schema* is a category presentation: it is given by a set of objects (which will be drawn as nodes), a set of generating arrows, and an equivalence relation on paths. We denote a path by writing its source object followed by a sequence of arrows. We denote an equivalence of paths using the $\simeq$-sign. For

example, consider the following schema:

(2)



Here we see a graph with five vertices and six arrows, and underlined at the top we see two path equivalence (PE) statements.[1] This information generates a category: the free category on the graph, modulo the path equivalence relation. In fact, in [Sp1, 3.4.1] the author defines a category **Sch**, whose objects are schemas (presented categories) as above, and proves that **Sch** is equivalent to **Cat**. From here on, we elide the difference between a schema (category presentation) and a category.

A schema $\mathcal{C}$ is supposed to describe the wiring of a database. We think of each object $c \in \mathrm{Ob}(\mathcal{C})$ as representing a table and each arrow $f\colon c \to c'$ emanating from c as representing a column of c that takes values in table c'. Roughly, an *instance* on $\mathcal{C}$ is the actual data: more precisely, an instance assigns to each table a set of rows of data that conform to the specifications given by $\mathcal{C}$. For example, the schema represented in Diagram (2) describes the wiring of the following database instance:

(3)

Employee				
ID	**first**	**last**	**manager**	**worksIn**
101	Alan	Turing	103	q10
102	Camille	Jordan	102	x02
103	Andrey	Markov	103	q10

Department		
ID	**name**	**secretary**
q10	Applied math	101
x02	Pure math	102

FirstNameString
ID
Alan
Alice
Andrey
Camille
David
⋮

LastNameString
ID
Arden
Hoover
Jordan
Markov
Turing
⋮

DepartmentNameString
ID
Applied math
Biology
Pure math
⋮

Every table has an ID column and perhaps other columns. Counting tables in (3) we find five, the number of nodes in (2); and counting the non-ID columns in (3) we find six, the number of arrows in (2).

[1] The first PE statement, "`Employee` manager worksIn ≃ `Employee` worksIn", identifies a path of length 2 with a path of length 1. The second PE statement, "`Department` secretary worksIn ≃ `Department`", identifies a path of length 2 with a path of length 0.

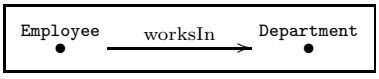
In fact, we can see that Diagram (3) constitutes an assignment of a set (of rows) to each node in $\mathcal{C}$ and a function to each arrow in $\mathcal{C}$. For example the node `Employee` is assigned the set $\{101, 102, 103\}$ and the arrow **manager**: `Employee` $\to$ `Employee` is assigned the function sending $101 \mapsto 103$ and $102 \mapsto 102$ and $103 \mapsto 103$. Thus an instance on schema $\mathcal{C}$ is precisely a functor $I\colon \mathcal{C} \to \mathbf{Set}$. The path equivalence relation on $\mathcal{C}$ ensures that the values behave in specified ways. For example, Alan Turing's manager is Andrey Markov, and these two men (are required to) work in the same department, Applied math. Similarly, the Secretary of Pure math is Camille Jordan, and he (by necessity) works in the Pure math department.

We summarize all this in a formal, if hasty, definition. A careful description is given in [Sp1].

Definition 2.1.1. A *schema* $\mathcal{C}$ is a small category presentation. An *instance on* $\mathcal{C}$ is a functor $\delta\colon \mathcal{C} \to \mathbf{Set}$.

Throughout this paper we will continually return to a couple examples.

Example 2.1.2. One of the most basic categories is the so-called free-arrow category. We simply add names to make it more reminiscent of databases.

$\mathcal{W} :=$ 

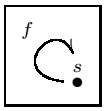
A instance $\delta\colon \mathcal{W} \to \mathbf{Set}$ consists of a set of employees, a set of departments, and a function mapping each employee to a department. For example

$\delta :=$

Employee	
ID	**WorksIn**
Alice	Math
Bob	EECS
Carl	Ling
Deb	EECS
Fred	Math
Jen	Bio

Department
ID
Bio
EECS
Ling
Math
Music

Example 2.1.3. The schema represented here

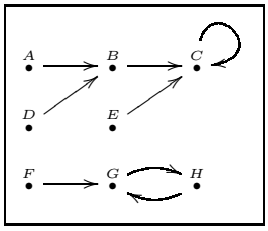
$\mathcal{L}oop :=$ 

has one object s and one generating arrow f, but a countably infinite set $\{f^n \mid n \in \mathbb{N}\}$ of paths. An instance $\delta\colon \mathcal{L}oop \to \mathbf{Set}$ is often called a *discrete dynamical system*. It consists of a set, which we might think of as the set of states of the system, and a function from that set to itself, which we might think of as the "next state" function. As with any database instance we can apply the Grothendieck construction (see [Sp2]) and get a nice picture of the system. For example one

might have

$\delta :=$

s	
ID	**f**
A	B
B	C
C	C
D	B
E	C
F	G
G	H
H	G

$\xmapsto{\text{pictured}}$ $\int \delta =$ 

Discrete dynamical systems are commonly used in modeling [San]. In fact, we will see throughout this paper that Kleisli instances on $\mathcal{L}oop$ are equivalent to structures of classical mathematical interest. A list of such examples is provided in the Introduction, Table (1).

2.2. **Monads and Kleisli categories.** In this section we define monads on the category **Set** and their Kleisli categories. Monads can be defined on any category, but the discussion will be a bit simpler if we are content with specializing to **Set**. One can replace **Set** with **Type**, the category of types for any typed λ-calculus, in what follows.

Definition 2.2.1. A *monad* $\top$ on **Set** consists of a triple $\top := (T, \eta, \mu)$, where $T\colon \mathbf{Set} \to \mathbf{Set}$ is a functor and $\eta\colon \mathrm{id}_{\mathbf{Set}} \to T$ and $\mu\colon T \circ T \to T$ are natural transformations, such that the following diagrams commute:

(4)
$$\begin{array}{ccc} T \circ \mathrm{id}_{\mathbf{Set}} & \xrightarrow{\mathrm{id}_T \circ \eta} & T \circ T \\ & \searrow & \downarrow \mu \\ & & T \end{array}$$

(5)
$$\begin{array}{ccc} \mathrm{id}_{\mathbf{Set}} \circ T & \xrightarrow{\eta \circ \mathrm{id}_T} & T \circ T \\ & \searrow & \downarrow \mu \\ & & T \end{array}$$

(6)
$$\begin{array}{ccc} T \circ T \circ T & \xrightarrow{\mu \circ \mathrm{id}_T} & T \circ T \\ \downarrow{\scriptstyle \mathrm{id}_T \circ \mu} & & \downarrow \mu \\ T \circ T & \xrightarrow[\mu]{} & T \end{array}$$

We call T the *functor part* of $\top$ and we refer to η and μ as the *unit map* and the *multiplication map* of $\top$, respectively. We sometimes abuse notation and refer to the functor part T as though it were the whole monad.

Example 2.2.2. We now go through Definition 2.2.1 using the List monad. The first step is to give a functor $\mathrm{List}\colon \mathbf{Set} \to \mathbf{Set}$. For every set X we must provide a set $\mathrm{List}(X)$ and for every function $f\colon X \to Y$ we must provide a function $\mathrm{List}(f)\colon \mathrm{List}(X) \to \mathrm{List}(Y)$. To clarify the situation, let us lay out two sets X, Y and a function between them.

$$X = \{p, q, r\} \qquad\qquad Y = \{1, 2, 3, 4\}$$

$$X \xrightarrow{\quad f \quad} Y$$

$$\begin{aligned} p &\longmapsto 1 \\ q &\longmapsto 2 \\ r &\longmapsto 3 \\ &\quad\ 4 \end{aligned}$$

Then $\mathrm{List}(X)$ is the set of all lists in elements of X. Thus the set $\mathrm{List}(X)$ includes the empty list $[]$, one element lists such as $[p]$, and all other lists in X (of finite length) such as $[p, q, r, r, p]$. Given our function f as above, we can apply it term-by-term to a list in X and return a list of the same length in Y.

$$\mathrm{List}(X) \xrightarrow{\quad \mathrm{List}(f) \quad} \mathrm{List}(Y)$$

$$[q, p, p, r, q, q, r, q] \longmapsto [4, 1, 1, 1, 4, 4, 1, 4]$$

Thus we have described the functor List. As a monad, it comes with two natural transformations, a unit map η and a multiplication map μ. Given a set X, the unit map $\eta_X \colon X \to \mathrm{List}(X)$ returns singleton lists as follows

$$X \xrightarrow{\quad \eta_X \quad} \mathrm{List}(X)$$

$$\begin{aligned} p &\longmapsto [p] \\ q &\longmapsto [q] \\ r &\longmapsto [r] \end{aligned}$$

Given a set X, the multiplication map $\mu_X \colon \mathrm{List}(\mathrm{List}(X)) \to \mathrm{List}(X)$ flattens lists of lists as follows.

$$\mathrm{List}(\mathrm{List}(X)) \xrightarrow{\quad \mu_X \quad} \mathrm{List}(X)$$

$$\big[[q, p, r], [], [q, r, p, r], [r]\big] \longmapsto [q, p, r, q, r, p, r, r]$$

The naturality of η and μ just mean that these maps work appropriately well under term-by-term replacement by a function $f \colon X \to Y$. Finally the three monad axioms (4), (5), and (6) can be exemplified as follows:

$$\begin{array}{ccc} [p, q, q] & \xrightarrow{\mathrm{id}_{\mathrm{List}} \circ \eta} & \big[[p], [q], [q]\big] \\ & \searrow & \downarrow \mu \\ & & [p, q, q] \end{array} \qquad \begin{array}{ccc} [p, q, q] & \xrightarrow{\eta \circ \mathrm{id}_{\mathrm{List}}} & \big[[p, q, q]\big] \\ & \searrow & \downarrow \mu \\ & & [p, q, q] \end{array}$$

$$\begin{array}{ccc} \Big[\big[[p,q],[r]\big],\big[[],[r,q,q]\big]\Big] & \xmapsto{\mu\circ \mathrm{id}_{\mathrm{List}}} & \big[[p,q],[r],[],[r,q,q]\big] \\ \Big\downarrow{\scriptstyle \mathrm{id}_{\mathrm{List}}\circ\mu} & & \Big\downarrow{\scriptstyle \mu} \\ \big[[p,q,r],[r,q,q]\big] & \xmapsto[\mu]{} & [p,q,r,r,q,q] \end{array}$$

The List monad is but one example of a huge variety of monads on **Set**. Many more examples will be given in Section 4. We now go on to define the Kleisli category associated to a monad. The definition may be a bit opaque. We give an example in 2.2.4, but the real motivation comes in Section 3. Readers who learn best by example might skip directly to Section 4.

Definition 2.2.3. Let $\top = (T, \eta, \mu)$ be a monad on **Set**. The *Kleisli category* associated to $\top$, denoted $\mathbf{Kls}(\top)$, is defined as follows. The objects are sets, i.e.

$$\mathrm{Ob}(\mathbf{Kls}(\top)) = \mathrm{Ob}(\mathbf{Set}).$$

For any sets $X, Y \in \mathrm{Ob}(\mathbf{Kls}(\top))$ we put

$$\mathrm{Hom}_{\mathbf{Kls}(\top)}(X,Y) = \mathrm{Hom}_{\mathbf{Set}}(X, T(Y)).$$

Given morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $\mathbf{Kls}(\top)$, we must define their composite $g \circ f$. Unwinding definitions, we are given functions

$$X \xrightarrow{f} T(Y) \tag{7}$$

$$Y \xrightarrow{g} T(Z) \tag{8}$$

in **Set**, and we need a function $X \to T(Z)$. Let $\overline{g}\colon T(Y) \to T(Z)$ denote the composite function

$$T(Y) \xrightarrow{T(g)} T(T(Z)) \xrightarrow{\mu} T(Z). \tag{9}$$

We define the map $g \circ f\colon X \to Z$ in $\mathbf{Kls}(\top)$ to be the composition of f and $\overline{g}$ in **Set**. Monad axiom (6) ensures that this composition law is associative, and axioms (4) and (5) ensure that the identity, which on $X \in \mathrm{Ob}(\mathbf{Kls}(\top))$ is given by $\eta_X\colon X \to TX$, is a left and right unit.

For any set X we refer to elements of $T(X)$ as *T-values in X*.

Example 2.2.4. We continue working with the List monad from Example 2.2.2. The objects of the Kleisli category $\mathbf{Kls}(\mathrm{List})$ are, as always, simply sets. Given sets X and Y (say $X = \{p,q,r\}$ and $Y = \{s,t\}$), a morphism $f\colon X \to Y$ in $\mathbf{Kls}(\mathrm{List})$ is a function $X \to \mathrm{List}(Y)$. In other words it consists of three lists in letters s, t. For example let us say

$$f(p) = [s,s] \qquad f(q) = [] \qquad f(r) = [t,s,t].$$

To explain the composition law, let us define a new set $Z = \{u,v\}$ and a function $g\colon Y \to \mathrm{List}(Z)$ given by

$$g(s) = [u,v,v] \qquad g(t) = [v,u].$$

Then the composition $g \circ f\colon X \to Z$ in $\mathbf{Kls}(\mathrm{List})$ corresponds to the obvious substitution:

$$g \circ f(p) = [u,v,v,u,v,v] \qquad g \circ f(q) = [] \qquad g \circ f(r) = [v,u,u,v,v,v,u].$$

Remark 2.2.5. Given a monad $\top$, its Kleisli category $\mathbf{Kls}(\top)$ is equivalent to the category of free $\top$-algebras ([BW]). However the database representation of maps that seems to be suggested by the Kleisli category is much more compact than that suggested by the category of free algebras. For example, consider the List monad. If X is a set with three elements and Y is any set, then a function $X \to \mathrm{List}(Y)$ can be represented by a table with three rows. On the other hand, one might imagine that a map $f\colon \mathrm{List}(X) \to \mathrm{List}(Y)$ should be represented by a table with infinitely many rows, one for each element of $\mathrm{List}(X)$. Our point is that the Kleisli representation is valuable because it is as succinct as possible.

3. Kleisli instances

Now that we have a categorical viewpoint of databases (Section 2.1) and an understanding of the Kleisli category, we can combine them.

Definition 3.1.6. Let $\mathcal{C}$ denote a schema and let $\top := (T, \eta, \mu)$ denote a monad on **Set** having Kleisli category $\mathbf{Kls}(\top)$. A *Kleisli* $\top$*-instance on* $\mathcal{C}$ (or simply a $\top$*-instance on* $\mathcal{C}$) is a functor $\delta\colon \mathcal{C} \to \mathbf{Kls}(\top)$.

3.2. **Representing Kleisli instances.** Let us examine Definition 3.1.6 in detail. For the remainder of Section 3.2, $\mathcal{C}$ will denote a schema, $\top = (T, \eta, \mu)$ will denote a monad on **Set**, and $\delta\colon \mathcal{C} \to \mathbf{Kls}(\top)$ will denote a Kleisli $\top$-instance on $\mathcal{C}$. We will first investigate what information is provided by our instance δ and then explain how to display it in an extension of the typical database format.

Our schema $\mathcal{C}$ consists of objects, arrows, and a path equivalence relation. For each object $c \in \mathrm{Ob}(\mathcal{C})$, our instance provides a set $\delta(c) \in \mathrm{Ob}(\mathbf{Kls}(\top)) = \mathrm{Ob}(\mathbf{Set})$. For each morphism $f\colon c \to c'$ in $\mathcal{C}$, our instance provides a morphism $\delta(f)\colon \delta(c) \to \delta(c')$ in $\mathbf{Kls}(\top)$; this is the same as a function

$$\delta(f)\colon \delta(c) \to T\delta(c').$$

A path $c_0 \xrightarrow{f_1} c_1 \xrightarrow{f_2} c_2 \xrightarrow{f_3} \cdots \xrightarrow{f_n} c_n$ in $\mathcal{C}$ is sent to a composition of functions

$$\delta(c_0) \xrightarrow{\delta(f_1)} T\delta(c_1) \xrightarrow{\delta(f_2)} T^2\delta(c_2) \xrightarrow{\delta(f_3)} \cdots \xrightarrow{\delta(f_n)} T^n\delta(c_n) \xrightarrow{\mu^{n-1}} T\delta(c_n),$$

and the path equivalence relation must be satisfied with respect to such compositions.

To represent an *atomic* database instance $\epsilon\colon \mathcal{C} \to \mathbf{Set}$, as in Section 2.1, we used (and will continue to use) a tabular format in which every object $c \in \mathrm{Ob}(\mathcal{C})$ was displayed as a table including one ID column and an additional column for every arrow emanating from c. In the ID column of table c were elements of the set $\epsilon(c)$ and in the column assigned to some arrow $f\colon c \to c'$ the cells were elements of the set $\epsilon(c')$.

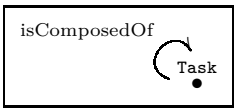
To represent a *Kleisli* database instance $\delta\colon \mathcal{C} \to \mathbf{Kls}(\top)$ is similar; we again use a tabular format in which every object $c \in \mathrm{Ob}(\mathcal{C})$ is displayed as a table including one ID column and an additional column for every arrow emanating from c. In the ID column of table c are again elements of the set $\delta(c)$; however in the column assigned to some arrow $f\colon c \to c'$ are not elements of $\delta(c')$ but T-values in $\delta(c')$, i.e. elements of $T\delta(c')$.

Example 3.2.1 (Lists). Let $\top = (\mathrm{List}, \eta, \mu)$ be the list monad as described in Examples 2.2.2 and 2.2.4. Here we show how a List-instance could be represented in

a tabular fashion. Our imagined scenario is as follows. We have a set K of tasks. Each task $k \in K$ is composed of an ordered sequence of other tasks in the set,

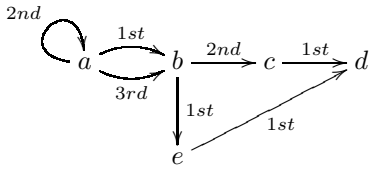
$$k \xmapsto{\text{is composed of}} [k_1, \ldots, k_n] \,.$$

Here a task k might be irreducible ($k \mapsto [k]$) or empty of requirements ($k \mapsto []$). Our situation is modeled by a List-instance on the schema

$\mathcal{L}oop \cong$ 

The following is an example of such:

Task	
ID	**IsComposedOf**
a	[b, a, b]
b	[e, c]
c	[d]
d	[]
e	[d]



A List-instance on $\mathcal{L}oop$ can be thought of as a directed graph such that every vertex has finitely many outgoing edges, which are linearly ordered.

3.3. **The categories $\mathcal{C}$–$\mathbf{Kls}(\top)$ and $\mathcal{C}$–$\widetilde{\mathbf{Kls}}(\top)$.** Kleisli instances are interesting objects in their own right, as we will see in Section 4; however, any category theorist will be interested in the morphisms between them. It seems that different notions of morphisms are appropriate in different circumstances. Below we define two categories for any schema $\mathcal{C}$ and monad $\top$; both have the same set of objects, namely the set of Kleisli $\top$-instances on $\mathcal{C}$, but one has more morphisms. In Remark 3.3.4, we will offer still another possibility. Perhaps the point is that there several viable notions of morphisms between Kleisli states and the choice of which to use should be dictated by ones purpose.

Definition 3.3.1. Let $\mathcal{C}$ be a schema, let $\top = (T, \eta, \mu)$ be a monad on **Set**, and let $\delta, \epsilon \colon \mathcal{C} \to \mathbf{Kls}(\top)$ be $\top$-instances on $\mathcal{C}$. A *general morphism of $\top$-instances on $\mathcal{C}$* is a natural transformation $a \colon \delta \to \epsilon$ of functors. We define the *category $\top$-instances on $\mathcal{C}$*, denoted $\mathcal{C}$–$\mathbf{Kls}(\top)$, to be the category having objects and general morphisms as above.

Let $\mathcal{C}$ be a category and $\top = (T, \eta, \mu)$ a monad on **Set**. Given two $\top$-instances $\delta, \epsilon \colon \mathcal{C} \to \mathbf{Kls}(\top)$, a general morphism $a \colon \delta \to \epsilon$ is simply a natural transformation of functors. Unpacking that definition, we have for every object $c \in \mathrm{Ob}(\mathcal{C})$ a component morphism $a_c \colon \delta(c) \to \epsilon(c)$ in $\mathbf{Kls}(\top)$, which is a function $a_c \colon \delta(c) \to T\epsilon(c)$. These components have to fit into naturality squares: given any $f \colon c \to c'$

in $\mathcal{C}$ we need the following diagram to commute:

$$\begin{array}{ccc} \delta(c) & \xrightarrow{\delta(f)} & T\delta(c') \\ {\scriptstyle a_c}\downarrow & & \downarrow{\scriptstyle \mu\circ Ta_{c'}} \\ T\epsilon(c) & \xrightarrow[\mu\circ T\epsilon(f)]{} & T\epsilon(c'). \end{array} \tag{10}$$

We will see in Example 4.4.2 that for the classical mathematical subject area of representation theory [EGH], these so-called general morphisms are precisely what one wants. In other words for the vector-space monad Vect, and a category G (generally either a group or a quiver), the category of general Vect-instances is the category of G-representations, G–$\mathbf{Kls}(\mathrm{Vect}) \cong \mathbf{Rep}(G)$.

However for classical computer science, these general morphisms seem to be too general. For example, we will show that there is a monad Tur_M^U for which the $\mathcal{L}oop$-instances are almost precisely the same thing as Turing machines.[2] In this setting, general morphisms seem strange and unmotivated whereas the basic morphisms (Definition 3.3.2) make much more sense. Indeed, given a basic morphism $p\colon \delta \to \delta'$ of unpointed Turing machines and any choice of start state $S \in \delta(s)$, the Turing machine (δ, S) computes the same function as $(\delta', p(S))$ computes.

Definition 3.3.2. Let $\mathcal{C}$ be a schema, let $\top = (T, \eta, \mu)$ be a monad on **Set**, and let $\delta, \epsilon\colon \mathcal{C} \to \mathbf{Kls}(\top)$ be $\top$-instances on $\mathcal{C}$. A *basic morphism* $b\colon \delta \to \epsilon$ consists of a component function $b_c\colon \delta(c) \to \epsilon(c)$ for each object $c \in \mathrm{Ob}(\mathcal{C})$, such that for each morphism $f\colon c \to c'$ in $\mathcal{C}$ the induced diagram of sets commutes:

$$\begin{array}{ccc} \delta(c) & \xrightarrow{\delta(f)} & T\delta(c') \\ {\scriptstyle b_c}\downarrow & & \downarrow{\scriptstyle Tb_{c'}} \\ \epsilon(c) & \xrightarrow[\epsilon(f)]{} & T\epsilon(c'). \end{array} \tag{11}$$

We denote by $\mathcal{C}$–$\widetilde{\mathbf{Kls}}(\top) \subseteq \mathcal{C}$–$\mathbf{Kls}(\top)$ the *basic subcategory of $\top$-instances on $\mathcal{C}$* which has as objects all $\top$-instances and which has as morphisms the basic morphisms, as defined above.

Remark 3.3.3. Given $\delta, \epsilon\colon \mathcal{C} \to \mathbf{Kls}(\top)$, there is another definition of basicness that is equivalent and perhaps more categorical. Namely a natural transformation $a\colon \delta \to \epsilon$ is basic if and only if, for each object $c \in \mathrm{Ob}(\mathcal{C})$ the component $a_c\colon \delta(c) \to T\epsilon(c)$ factors through the unit component $\eta_{\epsilon(c)}\colon \epsilon(c) \to T\epsilon(c)$.

For any monad $\top = (T, \eta, \mu)$ on **Set** and any category $\mathcal{C}$, there is a functor from the category of ordinary (atomic) database instances into the basic category of $\top$-instances,

$$E_\top^{\mathcal{C}}\colon \mathcal{C}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\widetilde{\mathbf{Kls}}(\top).$$

For an object $\delta\colon \mathcal{C} \to \mathbf{Set}$ we have $E_\top^{\mathcal{C}}(\delta) = \delta$, and for a morphism $a\colon \delta \to \epsilon$, we have $E_\top^{\mathcal{C}}(a) = \eta_\epsilon \circ a\colon \delta \to T\epsilon$. This functor is a faithful if and only if there exists a set X such that $\top(X)$ has cardinality at least 2.

[2] To be explicit, an object in $\mathcal{L}oop$–$\widetilde{\mathbf{Kls}}(\mathrm{Tur}_M^U)$ is equivalent to a Turing machine for which the start state has not been specified. We call objects in $\mathcal{L}oop$–$\widetilde{\mathbf{Kls}}(\mathrm{Tur}_M^U)$ *unpointed Turing Machines*.

Remark 3.3.4. Another notion of morphisms between $\top$-instances on $\mathcal{C}$ is often appropriate. If $\top$ is the extension of a monad $\top' = (T', \eta', \mu')$ on **Cat** via the adjunction $\mathbf{Set} \underset{\text{Ob}}{\overset{\text{Disc}}{\rightleftarrows}} \mathbf{Cat}$, then for any $X \in \text{Ob}(\mathbf{Set})$ one may say that "$T(X)$ naturally has the structure of a category" because $T'(\text{Disc}(X))$ is a category and $T(X) = \text{Ob}(T'(X))$. The monads $\mathcal{P}$, Multiset, List, and Exc_E (from Sections 4.2.6,4.2.5, and 4.3.1) are instances of this phenomenon. Consider, for example, the monad $\mathcal{P}$ where one recognizes that the power set of any set X does naturally come with a partial order. In this case, the monad $\mathcal{P}$ on **Set** is induced by a monad on **Cat** whose functor part is $\mathcal{X} \mapsto \text{Fun}(\mathcal{X}, [1])$, where $[1] = \boxed{\bullet \longrightarrow \bullet}$ is the walking arrow category.

When $\top$ extends to a monad on **Cat**, there seems to be another natural notion of morphisms between $\top$-instances on a schema $\mathcal{C}$. Namely, a *lax morphism* $a\colon \delta \to \epsilon$ between $\delta, \epsilon\colon \mathcal{C} \to \mathbf{Kls}(\top)$ consists of a component function $a_c\colon \delta(c) \to \epsilon(c)$ for each $c \in \text{Ob}(\mathcal{C})$ and, for each $f\colon c \to c'$ in $\mathcal{C}$, a natural transformation diagram

$$\begin{array}{ccc} \delta(c) & \xrightarrow{\delta(f)} & T(\delta(c')) \\ {\scriptstyle a_c}\downarrow & \swarrow & \downarrow{\scriptstyle a_{c'}} \\ \epsilon(c) & \xrightarrow[\epsilon(f)]{} & T(\epsilon(c')) \end{array}$$

in other words, a morphism $a_{c'} \circ \delta(f) \longrightarrow \epsilon(f) \circ a_c$.

When it is defined, this notion of morphism seems to have some advantages. For example when T =Multiset, we will see in Example 4.2.7 that the objects of $\mathcal{L}oop$–$\widetilde{\mathbf{Kls}}(\text{Multiset})$ are graphs, but the morphisms are more restrictive than graph morphisms. On the other hand, the category with the same objects and lax morphisms is equivalent to the category of graphs.

4. Examples

In this section we provide a survey of available monads that may be useful in databases. We divide them into five roughly sensible groups. In Section 4.1 we discuss two monads, one of which is initial in the category of monads (and gives rise to ordinary (atomic) database instances) and one of which is terminal in the category of monads (and gives rise to so-called unlinked instances). In Section 4.2 we give examples of monads that represent various kinds of collection such as subsets, multisets, lists, and probability distributions. In Section 4.3 we discuss monads which we describe as "tunable," because one can adjust the choice of monad in a controlled way; for example, for each choice of set E one obtains a different monad Exc_E of E-exceptions. In Section 4.4 we consider various classical algebraic monads, e.g. of vector spaces. Finally in Section 4.5 we consider process-oriented monads that include computations and experiments.

4.1. **Universal monads.**

4.1.1. *Atomic instances.* There is an identity monad on **Set**, which we denote

$$\text{Atomic} = (\text{id}_{\mathbf{Set}}, \text{id}, \text{id}).$$

Its instances are here called *atomic* (or *ordinary*, or *ordinary atomic*) instances. See Examples 2.1.2 and 2.1.3.

4.1.2. *Unlinked entities.* Consider the monad

$$\text{Unlinked} = (\{*\}^-, !, !),$$

where for any set $X \in \mathbf{Set}$, the set $\{*\}^X := \{*\}$ is the terminal object in $\mathbf{Set}$. The unit and multiplication maps are completely determined by their domain and codomain. An Unlinked-instance on $\mathcal{C}$ includes a set of records for each object $c \in \mathrm{Ob}(\mathcal{C})$, but the foreign keys offer no connection between them.

Example 4.1.3. Let $\mathcal{W}$ be as in Example 2.1.2. An instance $\delta\colon \mathcal{W} \to \mathbf{Kls}(\text{Unlinked})$ might look like

$\delta :=$ {

Employee	
ID	**WorksIn**
Alice	*
Bob	*
Carl	*
Deb	*
Fred	*
Jen	*

Department
ID
Bio
EECS
Math
Music

}

4.2. **Collection monads.**

4.2.1. *Subsets.* The monad

$$\mathcal{P} = (\mathbb{P}, \{-\}, \cup)$$

sends a set to its power set; the unit $\{-\}\colon X \to \mathbb{P}(X)$ is given by singleton subsets, $x \mapsto \{x\}$, and the multiplication is given by union $\cup\colon \mathbb{P}(\mathbb{P}(X)) \to \mathbb{P}(X)$. Note that there is an isomorphism of categories $\mathbf{Kls}(\mathcal{P}) \cong \mathbf{Rel}$, where $\mathbf{Rel}$ is the category of sets and binary relations [FS].

Example 4.2.2.

$\delta :=$ {

Employee	
ID	**WorksIn**
Alice	{Math, EECS}
Bob	{EECS}
Carl	{}
Deb	{EECS}
Fred	{Math, Bio}
Jen	{Bio}

Department
ID
Bio
EECS
Math
Music

}

Example 4.2.3 (Nonempty subsets). It is easy to see that the nonempty subsets functor $\mathbb{P}_+$ given by $\mathbb{P}_+(X) = \{Y \subseteq X \mid Y \neq \varnothing\}$ is the functor part of a monad. Note that $\mathbf{Kls}(\mathbb{P}_+)$ is the set of correspondences in the sense of theoretical economics, e.g. for "best response" strategies in game theory [Car, Section 2.1.5].

Example 4.2.4 (Turning a database inside out). Given a category $\mathcal{C}$ and an ordinary database instance $\delta\colon \mathcal{C} \to \mathbf{Set}$ we can, in a sense, invert δ by producing an instance on $\mathcal{C}^{\mathrm{op}}$:

$$\overleftarrow{\delta}\colon \mathcal{C}^{\mathrm{op}} \to \mathbf{Kls}(\mathcal{P}).$$

For $c \in \mathrm{Ob}(\mathcal{C})$ we have $\overleftarrow{\delta}(c) = \delta(c)$. For $f\colon c' \to c$ in $\mathcal{C}$ and $x \in \delta(c)$, we define $\overleftarrow{\delta}(f)\colon \delta(c) \to \mathbb{P}(\delta(c'))$ by

$$\overleftarrow{\delta}(f)(x) = \delta(f)^{-1}(x) \subseteq \delta(c').$$

4.2.5. *Lists.* This was the running example in Section 2.2 and Section 3; see in particular Example 3.2.1.

One can also define a *non-empty lists* monad List_+, similarly to the nonempty subsets monad $\mathbb{P}_+$ of Example 4.2.3.

4.2.6. *Finite multisets.* The monad Multiset $= (T, \eta, \mu)$ is given as follows. The functor part $T\colon \mathbf{Set} \to \mathbf{Set}$ is given by

$$T(X) := \coprod_{n\in\mathbb{N}} (X^n/\Sigma_n)$$

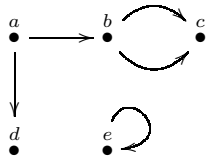
where Σ_n is the symmetric group on n letters, which acts on elements $x \in X^n$ by permuting the order of entries in x; the quotient of this action is the set of un-ordered n-tuples in X. The unit and multiplication in the Multiset monad are analogous to the unit and multiplication in the List monad.

The category $\mathbf{Kls}(\mathrm{Multiset})$ is equivalent to the category of sets and correspondences. Recall [Lur, Section 2.3.1] that for sets X and Y, a correspondence between X and Y is a diagram of the form $X \xleftarrow{f} C \xrightarrow{g} Y$, where C is a set and f, g are functions.

Example 4.2.7. Given a set X, a correspondence from X to itself is a graph with vertex set X. In other words, a graph is precisely a Multiset-instances on the category $\mathcal{L}oop$. We denote a multiset using usual set notation, but in which duplicate entries with the same name correspond to distinct elements of the multiset.

(12) $\delta :=$

s	
ID	**f**
a	{b,d}
b	{c,c}
c	{}
d	{}
e	{e}



The usual notion of graph morphism is captured by the lax notion given in Remark 3.3.4.

4.2.8. *Distributions.* Let $[0,1] \subseteq \mathbb{R}$ denote the set of real numbers between 0 and 1. Let X be a set and $p\colon X \to [0,1]$ a function. We say that p is a *finitary probability distribution on* X if there exists a finite subset $W \subseteq X$ such that

$$\sum_{w\in W} p(w) = 1, \tag{13}$$

and such that for all $x' \in X - W$ in the complement of W we have $p(x') = 0$. Note that W is unique if it exists; we call it *the support of* p and denote it $\mathbf{Supp}(p)$. Note also that if X is a finite set then every function p satisfying (13) is a finitary probability distribution on X.

For any set X, let $\mathrm{Dist}(X)$ denote the set of finitary probability distributions on X. It is easy to check that given a function $f\colon X \to Y$ one obtains a function $\mathrm{Dist}(f)\colon \mathrm{Dist}(X) \to \mathrm{Dist}(Y)$ by $\mathrm{Dist}(f)(y) = \sum_{f(x)=y} p(x)$. Thus we can consider $\mathrm{Dist}\colon \mathbf{Set} \to \mathbf{Set}$ as a functor, and in fact the functor part of a monad. Its unit $\eta\colon X \to \mathrm{Dist}(X)$ is given by the Kronecker delta function $x \mapsto \delta_x$ where $\delta_x(x) = 1$ and $\delta_x(x') = 0$ for $x' \neq x$. Its multiplication $\mu\colon \mathrm{Dist}(\mathrm{Dist}(X)) \to \mathrm{Dist}(X)$ is given

by weighted sum: given a finitary probability distribution $w\colon \mathrm{Dist}(X) \to [0,1]$ and $x \in X$, put $\mu(w)(x) = \sum_{p \in \mathbf{Supp}(w)} w(p)p(x)$.

Example 4.2.9 (Markov chains). Let $\mathcal{L}oop$ be as in Example 2.1.3. A Dist-instance on $\mathcal{L}oop$ is equivalent to a time-homogeneous Markov chain. To be explicit, a functor $\delta\colon \mathcal{L}oop \to \mathbf{Kls}(\mathrm{Dist})$ assigns to the unique object $s \in \mathrm{Ob}(\mathcal{L}oop)$ a set $S = \delta(s)$, which we call the state space, and to $f\colon s \to s$ a function $\delta(f)\colon S \to \mathrm{Dist}(S)$, which sends each element $x \in S$ to some probability distribution on elements of S. For example, the table δ on the left corresponds to the Markov matrix M on the right below:

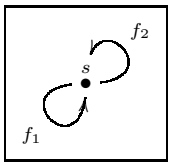
(14) $\delta :=$

s	
ID	**f**
1	.5(1)+.5(2)
2	1(2)
3	.7(1)+.3(3)
4	.4(1)+.3(2)+.3(4)

$$M := \begin{pmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.7 & 0 & 0.3 & 0 \\ 0.4 & 0.3 & 0 & 0.3 \end{pmatrix}$$

As one might hope, for any natural number $n \in \mathbb{N}$ the map $f^n\colon S \to \mathrm{Dist}(S)$ corresponds to the matrix M^n, which sends an element in S to its probable location after n iterations of the transition map.

One could also at least encode the information necessary to describe time-inhomogeneous Markov chains by using similar schemas, such as

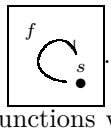
$\mathcal{L}oop \vee \mathcal{L}oop :=$ 

or $\mathcal{L}oop^{\vee\{1,2,\ldots\}} = \mathcal{L}oop \vee \mathcal{L}oop \vee \cdots$, and the same monad Dist.

4.3. **Tunable monads.** Some monads on **Set** come in families. To make this precise, we will say that a *tunable monad* is a pair $(\mathcal{I}, P)$ where $\mathcal{I}$ is a small category and $P\colon \mathcal{I} \to \mathbf{Monad}_{\mathbf{Set}}$ is a functor (see Definition 5.1.3). Of course then any monad can be trivially considered tunable by taking the indexing category to be $\mathcal{I} = \boxed{\bullet}$ (the discrete category on one object). It is clear that the degree to which a monad is tunable is measured by the complexity of $(\mathcal{I}, P)$. In the present section we will only mention three such monads; the first is indexed by **Fin**, the category of finite sets, the second is indexed by $\mathbf{Fin}^{\mathrm{op}}$, and the third is indexed by the category of monoids. See Section 5 for more on the value of tunable monads.

4.3.1. *Exceptions.* Let $E \in \mathbf{Fin}$ be a finite set. The monad $\mathrm{Exc}_E = (- \amalg E, \eta, \mu)$ is given as follows. The unit $\eta_X\colon X \to X \amalg E$ is given simply by the inclusion. The multiplication $\mu_X\colon X \amalg E \amalg E \to X \amalg E$ is given in the obvious way, by identity on each copy of X and E.

Example 4.3.2. If $E = \varnothing$ the exception monad reduces to the identity monad, i.e. $\mathrm{Exc}_\varnothing$-instances are ordinary atomic instances. If $E = \{*\}$ is the one-element set, then $\mathrm{Exc}_{\{*\}}$-instances correspond to databases in which a field can have null values; we call $\mathrm{Exc}_{\{*\}}$ the *Maybe monad*.

Example 4.3.3. Let $\mathcal{L}oop :=$ 

. Then for any set E, the Exc_E-instances on $\mathcal{L}oop$ can encode recursive functions with output values in E. For example, with $E = \mathbb{N}$ we obtain the factorial function $n \mapsto n!$ as an Exc_E-instance, $\delta\colon \mathcal{L}oop \to \mathbf{Kls}(\mathrm{Exc}_{\mathbb{N}})$. Namely, we put $\delta(s) := \mathbb{N} \times \mathbb{N}$ and we put $\delta(f)\colon \delta(s) \to \delta(s) \amalg E$ on $(m, n) \in \delta(s)$ by

$$\delta(m,n) := \begin{cases} (mn, n-1) \in \delta(s) & \text{if } n \geqslant 1 \\ m \in E & \text{if } n = 0. \end{cases}$$

Then for any $n \in \mathbb{N}$, the factorial of n is obtained by starting with $(1, n)$ and repeatedly applying $\delta(f)$ until an output (in $E = \mathbb{N}$) is returned.

Example 4.3.4 (Database schemas). In Section 2.1 we gave a definition of database schemas, but we did not mention data types. One model for typed database schemas can be found in [Sp1, Section 5.1], but here we present another model based on monads.

Let E be a set, the elements of which are names of datatypes, e.g. $E = \{\text{String}, \text{Int}, \text{Float}\}$. Let $\mathrm{List}_E = (T_E, \eta, \mu)$ be the monad with functor part $T_E(X) = \mathrm{List}(X \amalg E)$, sending a set X to the set of lists for which each entry is an element either of X or of E. Then we construe any instance $\delta \in \mathcal{L}oop$–$\mathbf{Kls}(\mathrm{List}_E)$ as a database schema in the following way. The set $\delta(s)$ serves as the set of tables, and for each table $x \in \delta(s)$ the list $\delta(f)(x)$ serves as the set of columns of x, each of which is either another table (indicating a foreign key) or a datatype.

4.3.5. *Inputs.* Let $U \in \mathbf{Fin}$ be a finite set. The monad $\mathrm{Inp}^U = (X \mapsto X^U, \eta, \mu)$ is given as follows. The unit $\eta_X\colon X \to X^U$ sends x to the constant function at x. If $\Delta_U\colon U \to U \times U$ is the diagonal map, then we can describe the multiplication $\mu_X\colon (X^U)^U \to X^U$ by

$$(X^U)^U \cong X^{U \times U} \xrightarrow{\Delta_U} X^U.$$

Example 4.3.6 (Tailored user experience). If U is a set of users then the database instance $\delta\colon \mathcal{C} \to \mathbf{Kls}(\mathrm{Inp}^U)$ would provide possibly different values for different $u \in U$.

Similarly, if U is the set of dates, then the values in a database instance δ could be made to depend on the date.

Example 4.3.7 (Each universal monad as a special case). With $U = \varnothing$ the input monad $\mathrm{Inp}^{\varnothing}$ reduces to the Unlinked monad of Section 4.1.2. If $U = \{*\}$, the input monad $\mathrm{Inp}^{\{*\}}$ reduces to the identity monad, whose Kleisli instances are ordinary atomic instances as in Section 4.1.1.

Example 4.3.8 (Finite state automata). A finite state automaton consists of a set S of states, a set T of transitions, and a function $T \times S \to S$. By currying, this can be rewritten as a function $S \to S^T$. The category of finite state automata with transitions T is precisely the category $\mathcal{L}oop$–$\widehat{\mathbf{Kls}}(\mathrm{Inp}^T)$ of T-Input instances on $\mathcal{L}oop$.

4.3.9. *Monoid annotation.* Let $(M, 1, \star)$ be a monoid. We define the monad

$$M\text{-annotated} := (X \mapsto M \times X, (1, -), (\star, -)).$$

One way to think about this is that M is a language of instructions, multiplication corresponding to carrying out a sequence of instructions and unit corresponding to doing nothing, and M-annotated instances keep track of such instructions as one follows foreign keys through the database. However, there are other ways to think about M-annotated instances as well, as we show in two examples.

Example 4.3.10 (Assurance). Consider the monoid $M = ([0,1], 1, *)$, where $[0,1] \subseteq \mathbb{R}$ is the unit interval, and $*\colon [0,1] \times [0,1] \to [0,1]$ is given by multiplication of real numbers. Think of a $[0,1]$-annotated values as assurances. In other words if a data entry clerk or a scientist is less than 100% sure that a certain datum is correct, they can annotate it with their assurance level. To keep things uncluttered we simply do not write our assurance value if it is unit (100%).

$\delta :=$

`Person`	
ID	**LivesAt**
Alice	15 Ashville Rd.
Bob	34 Vine St. (80%)
Carl	21 Post St. (90%)
Deb	110 W. 5th Ave.

The monad multiplication assures that probability values will propagate through the database (with an independence assumption) as we compose foreign keys.

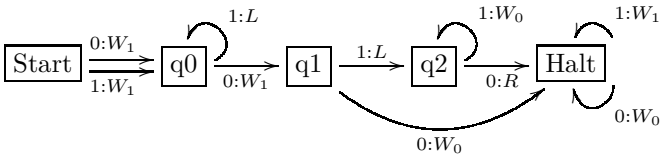
Example 4.3.11 (Time-delay). Consider the monoid $M = (\mathbb{R}_{\geqslant 0}, 0, +)$, where $\mathbb{R}_{\geqslant 0} \subseteq \mathbb{R}$ is the set of non-negative real numbers. Think of $\mathbb{R}_{\geqslant 0}$-annotated values as time-delays. In other words, each foreign key $f\colon c \to d$ in a database may correspond to a process that converts things of type c into things of type d, and the time delay monad allows us to also encode how long that process is expected to take. Monad multiplication assures that these values will be added together as we string together longer processes by composing foreign keys. Note that we could use $\mathbb{R}_{\geqslant 0} \cup \{\infty\}$ instead of $\mathbb{R}_{\geqslant 0}$ if we wanted to allow for never-ending processes.

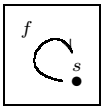
4.3.12. *Turing Machines.* Let M be a monoid and U a finite set. We have seen that on $\mathcal{L}oop$, the monad Inp^U encodes finite state automata (Example 4.3.8), the monad M-annotated encodes finite lists of instructions (Section 4.3.9), and the monad Exc_E encodes exceptions (Section 4.3.1). Let us set $E = \{\{\mathrm{Halt}\}\}$. We can combine these three monads into a new monad:

$$\mathrm{Tur}_M^U = \Big(X \mapsto \big(M \times (X \amalg \{\mathrm{Halt}\})\big)^U, \mu, \eta\Big)$$

We do not describe the unit and multiplication here, but they are easy enough to reconstruct in analogy with the descriptions in Sections 4.3.1, 4.3.5, and 4.3.9, assuming M acts trivially on $\{\mathrm{Halt}\}$.

Example 4.3.13. Consider the case where $U = \{0,1\}$ and where M is the free monoid on the set $\{L, R, W_0, W_1\}$, which we think of as the set of all sequences of instructions to move left, move right, write a 0, and write a 1. Then if X is thought of as the set of states of a Turing machine, a function $U \to M \times (X \amalg \{\mathrm{Halt}\})$ reads the input and produces an instruction and a new state (possibly the Halt state).



$\mathcal{L}oop :=$ 

$\delta :=$

s	
ID	**f**
Start	0: (W_1, q0), 1: (W_1, q0)
q0	0: (W_1, q1), 1: (L, q0)
q1	0: (W_0, Halt), 1: (L, q2)
q2	0: (R, Halt), 1: (W_0, q2)

A functor $\delta\colon \mathcal{L}oop \to \mathbf{Kls}(\mathrm{Tur}_M^U)$ consists of a set $X = \delta(s)$ of states and a function $\delta(f)\colon X \to (M \times (X \amalg \{\mathrm{Halt}\})^U$, which can be curried to $X \times U \to M \times (X \amalg \{\mathrm{Halt}\})$. After we choose a start state, we find ourselves with precisely the specification of Turing machines given in [BJ].

Tangentially, one may wonder how to evaluate such a Turing machine. Let *Tape* denote the set of positioned tapes, i.e. pairs (T, p) where $T\colon \mathbb{Z} \to \{0, 1\}$ is a function and $p \in \mathbb{Z}$. There is an evaluation function $e\colon \mathit{Tape} \to U$ given by $e(T, p) := T(p)$. By construction we have an action $\alpha\colon M \times \mathit{Tape} \to \mathit{Tape}$. We have a natural transformation $E\colon \mathrm{Tur}_M^U(-) \to (\mathit{Tape} \times (- \amalg \{\mathrm{Halt}\}))^{\mathit{Tape}}$, given on X by

$$\begin{aligned}(M \times (X \amalg \{\mathrm{Halt}\}))^U &\xrightarrow{e} (M \times (X \amalg \{\mathrm{Halt}\}))^{\mathit{Tape}} \\ &\xrightarrow{\mathrm{id}_{\mathit{Tape}}} (M \times \mathit{Tape} \times (X \amalg \{\mathrm{Halt}\}))^{\mathit{Tape}} \\ &\xrightarrow{\alpha} (\mathit{Tape} \times (X \amalg \{\mathrm{Halt}\}))^{\mathit{Tape}}.\end{aligned} \tag{15}$$

Choose a turing machine $\delta\colon \mathcal{L}oop \to \mathbf{Kls}(\mathrm{Tur}_M^U)$ with start state $S \in X$ and let $I \in \mathit{Tape}$ be the initialized tape. Then for each $n \in \mathbb{N}$ we have $\big(E \circ \delta(f^n)\big)(S)(I) \in \mathit{Tape} \times (X \amalg \{\mathrm{Halt}\})$, and we proceed with increasing values of n until the function returns the halt state, at which point we output the tape.

In Section 5.1 we will discuss morphisms of monads. We caution the reader that while Tur_M^U and $X \mapsto (\mathit{Tape} \times (X \amalg \{\mathrm{Halt}\}))^{\mathit{Tape}}$ are both monads, the mapping E in (15) is *not* a morphism of monads. It is a natural transformation of functors that preserves the unit but not the multiplication. This failure is somehow expected: if there were a morphism of monads from a Turing machine's specification to its implementation, the behavior of programs would be more easily analyzed than it turns out to be.

4.4. **Algebraic monads.**

4.4.1. *Vector spaces.* Let k be a field. There is a k-vector-space monad sending a set X to the free k-vector space with basis X. The unit map corresponds to the inclusion of basis vectors and the multiplication map corresponds to the ability to convert a linear combination of vectors into a single vector.

Example 4.4.2 (Representation theory). If G is a group (considered as a category with one object) then G–**Kls**(Vect_k) is equivalent to $\mathbf{Rep}_k(G)$, the category of G-representations. If Q is a free category then Q–**Kls**(Vect_k) is equivalent to $\mathbf{Rep}_k(Q)$, the category of quiver representations on Q (see [Kac]). In particular, Jordan Canonical Form is the classification of isomorphism classes in $\mathcal{L}oop$–**Kls**($\mathrm{Vect}_{\mathbb{C}}$).

4.4.3. *Others.* There are many algebraic theories—monoids, commutative monoids, groups, abelian groups, rings, commutative rings, etc., to name a few. In fact, some authors [Le2] *define* algebraic theories simply as monads on **Set**. Each monad on **Set** has an associated Kleisli category. In fact, two such monads have already been mentioned above under different names. The List-monad from Example 3.2.1 is another name for the monoid monad, and the Multiset monad from Section 4.2.6 is another name for the commutative monoid monad.

Example 4.4.4 (Multigraphs). A *multigraph* (see [HMP]) consists of a set of nodes and a set of *multi-arrows*, each of which points from one node to a finite list of nodes. A symmetric multigraph is almost the same except each multi-arrow points from one node to a finite set of nodes.

Let $\mathbb{N}[-]\colon \mathbf{Set} \to \mathbf{Set}$ denote the (functor part of the) free commutative rig monad [Gol] (respectively let $\mathbb{N}\langle -\rangle\colon \mathbf{Set} \to \mathbf{Set}$ denote the free rig monad). For example $\mathbb{N}[x, y]$ is the set of polynomials in x, y with natural number coefficients, containing elements like 3 and $xy^2 + 2x^3 + y$. (Similarly $\mathbb{N}\langle x, y\rangle$ would contain elements like $xyy + yxy$.) The set of **Kls**($\mathbb{N}[-]$)-instances (resp. the set of **Kls**($\mathbb{N}\langle -\rangle$)-instances) on $\mathcal{L}oop$ is precisely the set of symmetric multigraphs (resp. multigraphs). With morphisms as in Remark 3.3.4, the category of $\mathbb{N}\langle -\rangle$-instances on $\mathcal{L}oop$ is equivalent to the category of multigraphs.

4.5. **Process monads.** The examples in this section are a bit more far-flung, but still may be useful to give an idea of what is possible.

Example 4.5.1 (Computation). Fix a programming language L. For any set X, let $T(X)$ denote the set of programs that are written in L and such that, taking no input, will either halt and return a value in X or not halt. This is the functor part of a monad. By currying, a Kleisli map $X \to T(Y)$ is equivalent to a (possibly non-halting) computation taking input in X and returning values in Y.

Example 4.5.2 (Experiment). For any set X, let $T(X)$ denote the set of specifications for experiments that could be carried out from a fixed initial condition and that will result in a value in X. For example, if $X = \mathbb{Z}$ is the set of integers, then $T(X)$ might include as an element the phrase "survey 100 customers at the McDonalds on 32nd street, asking their favorite real number. Take their average as a real number and then apply the floor function to obtain an integer". Then T can be construed as the functor part of a monad. A Kleisli map $X \to TY$ is equivalent to the specification of an experiment that takes parameters in X and outputs a value in Y. The point is that the database does not hold the *results* of these experiments, but instead the *experiment specifications* which, if performed, will result in values later. Any value counts as a (trivial) experiment, so this generalizes ordinary databases.

5. Transformations

Monads, like everything in category theory, are not stand-alone objects but exist in a category, in which the morphisms are an integral part of the picture. In section 5.1 we will define morphisms of monads. These include operations like transforming a list into a multiset (by forgetting order) or transforming a probability distribution into a subset (by taking all elements that have nonzero probability). A morphism $f\colon \top \to \top'$ of monads results in a functor between the corresponding Kleisli categories. For any schema $\mathcal{C}$ we have a commutative square

$$\begin{array}{ccc} \mathcal{C}\text{–}\widetilde{\mathbf{Kls}}(\top) & \xrightarrow{f} & \mathcal{C}\text{–}\widetilde{\mathbf{Kls}}(\top') \\ \downarrow & & \downarrow \\ \mathcal{C}\text{–}\mathbf{Kls}(\top) & \xrightarrow[f]{} & \mathcal{C}\text{–}\mathbf{Kls}(\top') \end{array}$$

that converts $\top$-instances into $\top'$-instances in either sense given below (see Definitions 3.3.1 and 3.3.2). In Section 5.2 we will sketch some examples.

5.1. Morphisms of monads.

Definition 5.1.1. Let $\top = (T, \eta, \mu)$ and $\top' = (T', \eta', \mu')$ be monads on $\mathcal{S}$. A *morphism of monads from* $\top$ *to* $\top'$ is a natural transformation $\alpha\colon T \to U$ such that the following diagrams commute:

$$\begin{array}{ccc} \mathrm{id}_{\mathcal{S}} & \xrightarrow{\eta} & T \\ & \searrow^{\eta'} & \downarrow F \\ & & T' \end{array} \qquad \text{and} \qquad \begin{array}{ccc} T^2 & \xrightarrow{\mu} & T \\ {\scriptstyle F^2}\downarrow & & \downarrow{\scriptstyle F} \\ (T')^2 & \xrightarrow[\mu']{} & T'. \end{array}$$

Remark 5.1.2. An important upshot of Proposition 5.1.4 is the following. For any category $\mathcal{C}$ and morphism of monads $f\colon \top \to \top'$ we have a functor $\mathcal{C}\text{–}\mathbf{Kls}(f)\colon \mathcal{C}\text{–}\mathbf{Kls}(\top) \to \mathcal{C}\text{–}\mathbf{Kls}(\top')$. Thus any $\top$-instance on $\mathcal{C}$ can be transformed via f into a $\top'$-instance on $\mathcal{C}$.

Definition 5.1.3. A monad $\top = (T, \eta, \mu)$ is called *finitary* if the functor T is determined by its values on finite sets, in the following sense. Let $X \in \mathrm{Ob}(\mathbf{Set})$ be a set and let $\mathbf{Fin}/X$ denote the category whose objects are finite subsets of X and whose morphisms are functions over X. Note that for each object $f\colon Y \to X$ in $\mathbf{Fin}_{/X}$ there is an induced map $TY \xrightarrow{f} TX$, so we obtain a map

$$M_X\colon \operatorname*{colim}_{\substack{f\colon Y \to X \\ \in \mathbf{Fin}_{/X}}} (TY) \longrightarrow TX$$

Then $\top$ is finitary if the map denoted M_X is a bijection for every $X \in \mathrm{Ob}(\mathbf{Set})$.

The *category of finitary monads on* $\mathbf{Set}$, denoted $\mathbf{Monad}_{\mathbf{Set}}$, has finitary monads as objects and morphisms of monads (as in Definition 5.1.1) as morphisms.

Proposition 5.1.4. *A morphism of finitary monads induces a functor between their Kleisli categories. In other words there a functor* $\mathbf{Kls}(-)\colon \mathbf{Monad}_{\mathbf{Set}} \to \mathbf{Cat}$.

Proof. This is straightforward. □

5.2. **Examples of transformations.** In this section we write down several simple examples of morphisms of monads. In a few of these we are explicit, but we quickly move to a more colloquial style, assuming that any reader with sufficient interest and background can fill in the details for him or herself.

5.2.1. *Universals.*

Example 5.2.2. The initial object in $\mathbf{Monad}_{\mathbf{Set}}$ is $\mathrm{id}_{\mathbf{Set}}$, whose instances are ordinary atomic database instances. Given any monad $\top$, there is a unique morphism of monads $\mathrm{id}_{\mathbf{Set}} \to \top$. As in Remark 5.1.2, there is a unique formula to convert any atomic instance into a $\top$-instance.

Example 5.2.3. The terminal object in $\mathbf{Monad}_{\mathbf{Set}}$ is the Unlinked monad from Section 4.1.2. Given any monad $\top$, there is a unique morphism $\top \to \mathrm{Unlinked}$. As in Remark 5.1.2, given any database instance on $\mathcal{C}$, be it atomic or not, one can forget all the foreign key information and be left with an unlinked instance.

5.2.4. *Forgetting structure.*

Example 5.2.5 (Distributions to subsets). Recall the Subset monad $\mathcal{P}$ and the Distribution monad Dist from Sections 4.2.1 and 4.2.8, and let X be a set. Recall that the support of a distribution $p\colon X \to [0,1]$ is the subset $\mathbf{Supp}(p) = \{x \in X \mid p(x) \neq 0\} \subseteq X$. This notion of support induces a morphism of monads $\mathrm{Dist} \to \mathcal{P}$.

Example 5.2.6 (Multisets to subsets). Recall the Subset monad $\mathcal{P}$ and Multiset monad from Sections 4.2.1 and 4.2.6, and let X be a set. A multiset in X can be conceived as a function $Y \to X$, and its image is a subset of X. By this process one obtains a morphism of monads $\mathrm{Multiset} \to \mathcal{P}$.

Example 5.2.7 (Lists to multisets). Recall the List and Multiset monads from Sections 4.2.5 and 4.2.6, and let X be a set. For each natural number $n \in \mathbb{N}$ we have a function $X^n \to X^n/\sim$ that forgets the order of n-element lists. This induces a morphism of monads $\mathrm{List} \to \mathrm{Multiset}$.

Example 5.2.8. Recall the Atomic monad $\mathrm{id}_{\mathbf{Set}}$, the List monad, the non-empty list monad List_+, and the Maybe monad $\mathrm{Exc}_{\{*\}}$ from Sections 4.2.5 and 4.3.2. There is a morphism which could be called the "first element, if it exists" map $\mathrm{List} \to \mathrm{Exc}_{\{*\}}$. Similarly, there is a "first element" map $\mathrm{List}_+ \to \mathrm{id}_{\mathbf{Set}}$.

5.2.9. *Tunable monads.* As explained in Section 4.3, a tunable monad is a pair $(\mathcal{I}, P)$ where $P\colon \mathcal{I} \to \mathbf{Monad}_{\mathbf{Set}}$. For every object $i \in \mathrm{Ob}(\mathcal{I})$ we have a monad $P(i)$ and for every arrow in $\mathcal{I}$ we have a morphism of monads. By Proposition 5.1.4, we can compose with the functor $\mathbf{Kls}(-)\colon \mathbf{Monad}_{\mathbf{Set}} \to \mathbf{Cat}$. So for any morphism $f\colon i \to i'$ in $\mathcal{I}$ and any schema $\mathcal{C}$ we have a functor $\mathcal{C}$–$\mathbf{Kls}(P(i)) \to \mathcal{C}$–$\mathbf{Kls}(P(i'))$ that functorially converts $P(i)$-instances into $P(i')$-instances.

In Section 4.3 we discussed the monads Exc_E and Inp^U, for $E, U \in \mathbf{Fin}$, and M-annotated for monoids M. The value in these is found in the fact that as time goes on and the model evolves, the database architect may need to change parameters of the schema with minimal disturbance to users. For example, if at some point the architect wants to add a new sort of exception, or collapse two kinds of exception into one, he or she can do that by finding a function $E_{\mathrm{old}} \to E_{\mathrm{new}}$ from the old exception set to the new, and it will induce a functor that transforms databases instances with the old set of exceptions into instances with the new set.

5.2.10. *Others.*

Example 5.2.11 (Simulation). Recall the Computation monad and the Experiments monad from Examples 4.5.1 and 4.5.2. We could imagine that simulation is a morphism of monads from the latter to the former, converting a description of an experiment into a computation.

Example 5.2.12 (Programs take time). Recall the Time-delay monad and the Computation monad from Examples 4.3.11 and 4.5.1. Counting the number of clock cycles induces a morphism of monads from the latter to the former.

6. Future work

The above work can be made far more flexible if we allow the choice of monad to vary over the schema. This way, some columns can be nullable and others not, or we could allow for lists in some areas of the schema and not in others. We will tackle this in an upcoming paper. It would also be interesting to consider how these variable monads and their associated instances would behave under change of schema functors $F\colon \mathcal{C} \to \mathcal{D}$. We also plan to investigate whether our work here can be nicely integrated with the ideas of [BNT], [Gru], [LT], and [Maj], in which one uses monads to handle collections. Finally, it seems fruitful to explore how the coincidence of (1) relates to Leinster's definitions of $\top$-multi-categories and operads in [Le1, Chapter 4].

References

[Awo] Awodey, S. (2010) *Category Theory* 2nd edition. Oxford Logic Guides, 52. Oxford University Press.

[BJ] Boolos, G.; Jeffrey, R. (1989) *Computability and Logic* 3rd edition. Cambridge University Press.

[BNT] Buneman, P.; Naqvi, S.; Tannen, V.; Wong, L. (1995) "Principles of programming with complex objects and collection types". *Theoretical Computer Science* 149, pp. 3–48.

[BW] Barr, M.; Wells, C. (1990) *Category theory for computing science.* Prentice Hall International Series in Computer Science.

[Car] Carter, M. (2001) *Foundations of mathematical economics.* MIT Press.

[Cod] Codd, E.F. (1970) "A relational model of data for large shared data banks". *Communications of the ACM.*

[EGH] Edingof, P.; Golberg, O.; Hensel, S.; Liu, T.; Schwender, A.; Vaintrob, D.; Yuovina, E. (2011) *Introduction to Representation Theory.* Student Mathematical Library, 59, AMS.

[FS] Freyd, P.; Scedrov, A. (1990) *Categories, allegories.* North-Holland Mathematical library, 39.

[Gol] Golan, J.S. (1992) *Semirings and their applications. Updated and expanded version of The theory of semirings, with applications to mathematics and theoretical computer science.* Longman Sci. Tech.

[Gru] Grust, T. (2003) "Monad comprehensions: a versatile representation for queries", in *The Functional Approach to Data Management*, Eds: Gray, P.; Kerschberg, L.; King, P.; and Poulovassilis, A. pp. 288Ð311.

[HMP] Hermida, C.; Makka, M.; Power, J. (1998) "Higher dimensional multigraphs". *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science.*

[Kac] Kac, V.G. (1980) "Infinite root systems, representations of graphs and invariant theory". *Invent. Math* 56, no. 1. pp. 57–92.

[Le1] Leinster, T. (2004) *Higher operads, higher categories.* London Mathematical Society Lecture Note Series, 298. Cambridge University Press.

[Le2] Leinster, T. (2006) "Are operads algebraic theories?" Bull. London Math. Soc. 38, no. 2, pp. 233 – 238.

[LT] Lellahi, K.; Tannen, V. (1997) "A calculus for collections and aggregates." Proceeding CTCS '97 Proceedings of the 7th International Conference on Category Theory and Computer Science. Springer-Verlag.

[Lur] Lurie, J. (2009) *Higher topos theory*. Annals of Mathematics Studies, 170. Princeton University Press.

[Maj] Majkic, Z. (2011) "Data Base Mappings and Monads: (Co)Induction". ePrint available: http://arxiv.org/abs/1102.4769

[Mei] Meijer, E. (2012) "All your database are belong to us". *Communications of the ACM*. 55, no. 9, pp. 54–60.

[Mog] Moggi, E.. (1991) "Notions of computation and monads". *Information and Computation* 93, no. 1, pp. 55–92.

[San] Sandefur, J.T. (1993) *Discrete dynamical modeling*. Oxford University Press.

[Sp1] Spivak, D.I. (2012) "Functorial data migration." *Information and Computation* 217, pp. 31–51.

[Sp2] Spivak, D.I. (2012) "Database queries and constraints via lifting problems." ePrint available: http://arxiv.org/abs/1202.2591

Department of Mathematics, Massachusetts Institute of Technology, Cambridge MA 02139

E-mail address: dspivak@gmail.com