Preparing Relational Algebra for "Just Good Enough" Hardware

José N. Oliveira

High Assurance Software Laboratory
INESC TEC and University of Minho
Braga, Portugal
(jno@di.uminho.pt)

Abstract. Device miniaturization is pointing towards tolerating imperfect hardware provided it is "good enough". Software design theories will have to face the impact of such a trend sooner or later.
A school of thought in software design is *relational*: it expresses specifications as relations and derives programs from specifications using relational algebra.
This paper proposes that *linear* algebra be adopted as an evolution of relational algebra able to cope with the quantification of the impact of imperfect hardware on (otherwise) reliable software.
The approach is illustrated by developing a monadic calculus for *component oriented* software construction with a probabilistic dimension quantifying (by linear algebra) the propagation of imperfect behaviour from lower to upper layers of software systems.

1 Introduction

In the trend towards miniaturization of automated systems the size of circuit transistors cannot be reduced endlessly, as these eventually become unreliable. There is, however, the idea that inexact hardware can be tolerated provided it is "good enough" [16].

Good enough has always been the way engineering works as a broad discipline: why invest in a "perfect" device if a less perfect (and less expensive) alternative fits the needs? Imperfect circuits will make a certain number of errors, but these will be tolerated if they nevertheless exhibit *almost* the same performance as perfect circuits. This is the principle behind *inexact circuit design* [16], where accuracy of the circuit is exchanged for cost savings (e.g. energy, delay, silicon) in a controlled way.

If unreliable hardware becomes widely accepted on the basis of fault tolerance guarantees, what will the impact of this be into the software layers which run on top of it in virtually any automated system? Running on less reliable hardware, functionally correct (e.g. proven) code becomes faulty and risky. Are we prepared to handle such risk at the software level in the same way it is tackled by hardware specialists? One needs to know how risk propagates across networks of software components so as to mitigate it.

The theory of software design by stepwise refinement already copes with some form of "approximation" in the sense that "vague" specifications are eventually realized by precise algorithms by taking design decisions which lead to (deterministic) code. However, there is a fundamental difference: all input-output pairs of a post-condition in a software specification are *equally* acceptable, giving room for the implementer to choose among them. In the case of imperfect design, one is coping with undesirable, possibly catastrophic outputs which one wishes to prove very unlikely.

In the area of safety critical systems, NASA has defined a *probabilistic risk assessment* (PRA) methodology [27] which characterizes risk in terms of three basic questions: *what can go wrong? how likely is it? and what are the consequences?* The PRA process answers these questions by systematically modeling and quantifying those scenarios that can lead to undesired consequences.

Altogether, it seems that (as happened with other sciences in the past) software design needs to become a *quantitative* or *probabilistic* science. Consider concepts such as e.g. *reliability.* From a qualitative perspective, a software system is *reliable* if it can *successfully carry out its own task as specified* [9]. But our italicized text is a inexact quotation of [9], the exact one being: *reliability [is] defined as a probabilistic measure of the system ability to successfully carry out its own task as specified.*

From a functional perspective, this means moving from specifications (input/output relations) and implementations (functions) to something which lives in between, for instance *probabilistic functions* expressing the propensity, or likelihood of multiple, possibly erroneous outputs. Typically, the classic non-deterministic choice between alternative behaviours,

$$\mathit{bad} \cup \mathit{good} \tag{1}$$

has to be replaced by probabilistic choice [19]

$$\mathit{bad} \;{}_{p}\!\diamond\; \mathit{good} \tag{2}$$

and the reasoning should be able to ensure that the probability p of bad behaviour is acceptably small.

Does the above entail abandoning relational reasoning in software design? Interestingly, the same style of reasoning will be preserved provided binary relations are generalized to (typed) matrices, the former being just a special case of the latter. This leads to a kind of *linear algebra of programming* [21]. Technically, in the same way relations can be *transposed* to set valued functions, which rely on the powerset monad to express non-determinism, so do probabilistic matrices, which transpose to distribution-valued functions that rely on the distribution monad to express probabilistic behaviour. It turns out that it is the converse of such transposition which helps, saving explicit set-theoretical constructions in one case and explicit distribution manipulation in the other via pointfree styled, algebraic reasoning.

Contribution. This paper proposes that, similarly to what has happened with the increasing role of *relational algebra* in computer science [7, 5, 25], *linear algebra* be adopted as its natural development where quantitative reasoning is required. Relation algebra and linear algebra share a lot in common once addressed from e.g. a categorial perspective [8, 17]. So it seems that there is room for evolution rather than radical change.

In this setting, this paper contributes (a) with a case study on such an evolution concerning a calculus of software components [2, 3] intended for quantitative analysis of software reliability; (b) with a strategy for reducing the impact of the “probabilistic move” based on re-interpreting software component semantics in linear algebra through a so-called “Kleisli-lifting’ which keeps as much of the original semantics definition as possible.

2 Context

Quantitative software reliability analysis is not so easy in practice because, as is well-known, software systems are nowadays built component-wise. Cortellessa and Grassi [9] quantify component-to-component error propagation in terms of a matrix whose entry (i, j) gives the probability of component i transferring control to component j — a kind of probabilistic call-graph. For our purposes, this abstracts too much from the semantics of component-oriented systems, which have been quite successfully formalized under the *components as coalgebras* motto (see e.g. [2]), building on extensive work on automata using coalgebra theory [24, 12].

Coalgebra theory can be regarded as a generic approach to transition systems, described by functions of type

$$f : S \to \mathbb{F}\ S \tag{3}$$

where S is a set of states and $\mathbb{F}\ S$ captures the future behaviour of the system according to evolution “pattern” $\mathbb{F}$ which (technically) is a functor. For $\mathbb{F}\ S = \mathbb{P}\ S$, the powerset functor, f is the *power-transpose* [5] of a binary relation on the state space S. Other instances of $\mathbb{F}$ lead to more sophisticated transition structures, for instance Mealy and Moore machines involving inputs and outputs. Barbosa [2] gives a software component calculus in which components are regarded as such machines, expressed as coalgebras.

In this paper we wish to investigate a (technically) cheap way of promoting the *components as coalgebras* approach from the qualitative, original formulation [2] to a quantitative, probabilistic extension able to cope with the impact of *inexact circuit design* into software. As the survey by Sokolova [26] shows, probabilistic systems have been in the software research agenda for quite some time. From the available literature we focus on a paper [12] which suits our needs: it studies trace semantics of state-based systems with different forms of branching such as e.g. the non-deterministic and the probabilistic, in a categorial setting. This fits with our previous work [22] on probabilistic automata as coalgebras in categories of matrices which shows that the cost of *going quantitative* amounts essentially to changing the underlying category where the reasoning takes place.

3 Motivation

This section presents a brief account of the component algebra [2] which is central to the current paper. For illustrative purposes, we have implemented this algebra of component combinators in Haskell, for the particular situation in which components are regarded as (monadic) Mealy machines. The original algebra has furthermore been extended probabilistically relying on the PFP library written by Erwig and Kollmansberger [10]. On purpose, the examples hide many technical details which are deferred to later sections.

Abstract Mealy machines. An $\mathbb{F}$-branching Mealy machine is a function of type

$$S \times I \to \mathbb{F}\ (S \times O) \tag{4}$$

where S is the machine's internal state space, I is the set of inputs and O the set of outputs. Our main principle is that of regarding a software system as a combination of Mealy machines, from elementary to more complex ones. For this to work, $\mathbb{F}$ in (4) will be regarded as a *monad* capturing effects which are propagated upwards, from component to composite machines.

Functions of type (4), for $\mathbb{F}$ a monad, will be referred to as monadic Mealy machines (MMM) in the sequel. This type (4) can be written in two other equivalent (isomorphic) ways, all useful in component algebra: the coalgebraic $S \to (\mathbb{F}\ (S \times O))^I$ — compare with (3) — and the state-monadic $I \to (\mathbb{F}\ (S \times O))^S$, depending on how currying is applied.

Methods = elementary Mealy machines. Let us see an example which shows how an aggregation of (possibly partial) functions sharing a data type already is a Mealy machine. In the example, a stack is modelled as a (partial) algebra of finite lists written in Haskell syntax as follows

$$\begin{array}{l} \mathit{push}\ (s, a) = a : s \\ \mathit{pop} = \mathit{tail} \\ \mathit{top} = \mathit{head} \\ \mathit{empty}\ s = (0 \equiv \mathit{length}\ s) \end{array} \quad \text{whose types are} \quad \begin{array}{l} \mathit{push} :: ([\,a\,], a) \to [\,a\,] \\ \mathit{pop} :: [\,a\,] \to [\,a\,] \\ \mathit{top} :: [\,a\,] \to a \\ \mathit{empty} :: [\,a\,] \to \mathbb{B} \end{array}$$

Below we show how to write each individual function as an elementary Mealy machine on the shared state space $S = [\,a\,]$ before aggregating them all into a single machine (component).

In the case of *push*, $I = a$. (Note that in Haskell syntax type variables are denoted by lower-case letters). What about O? We may regard it as an instance of the singleton type 1, whose unique inhabitant carries the information that the action indeed took place:

$$\begin{array}{l} \mathit{push}' :: ([\,a\,], a) \to ([\,a\,], 1) \\ \mathit{push}' = \mathit{push} \vartriangle ! \end{array}$$

This definition relies on the *pairing* operator, $(f \vartriangle g)\ x = (f\ x, g\ x)$ and on the uniquely defined (total and constant) function $! :: b \to 1$, often referred to as the ”bang” function.

Note how action (“method”) $push'$ is pure in the sense that it does not generate any effect. The same happens with

$$\begin{aligned}&\mathit{empty}' :: ([\,a\,],1) \to ([\,a\,],\mathbb{B})\\&\mathit{empty}' = (\mathit{id} \vartriangle \mathit{empty}) \cdot \pi_1\end{aligned}$$

where this time the singleton type is at the input side, meaning a “trigger” for the operation to take place. Functions id and π_1 are the identity function and the projection $\pi_1\ (x,y) = x$, respectively, the former ensuring that no state change takes place.

Concerning pop and top we have a new situation: as these are partial functions, some sort of totalization is required before promoting them to Mealy machines. The cheapest way of totalizing partial functions resorts to the “Maybe” monad $\mathbb{M}$, mapping into an error value $\bot$ the inputs for which the function is undefined and otherwise signaling a successful computation using the monad’s unit $\eta :: S \to \mathbb{M}\ S$:[1]

$$\begin{aligned}&\cdot \ \Leftarrow\ \cdot ::(a \to b) \to (a \to \mathbb{B}) \to a \to \mathbb{M}\ b\\&(f\ \Leftarrow\ p)\ a = \mathbf{if}\ p\ a\ \mathbf{then}\ (\eta \cdot f)\ a\ \mathbf{else}\ \bot\end{aligned}$$

Note how $f\ \Leftarrow\ p$ “fuses” f with a given pre-condition p, as in the following promotion of top to a $\mathbb{M}$-monadic Mealy machine

$$\begin{aligned}&\mathit{top}' :: ([\,a\,],1) \to \mathbb{M}\ ([\,a\,],a)\\&\mathit{top}' = (\mathit{id} \vartriangle \mathit{top}\ \Leftarrow\ (\neg \cdot \mathit{empty})) \cdot \pi_1\end{aligned}$$

which, as empty', does not change the state. Opting for the usual semantics of the pop method,

$$\begin{aligned}&\mathit{pop}' :: ([\,a\,],1) \to \mathbb{M}\ ([\,a\,],a)\\&\mathit{pop}' = (\mathit{pop} \vartriangle \mathit{top}\ \Leftarrow\ (\neg \cdot \mathit{empty})) \cdot \pi_1\end{aligned}$$

we finally go back to pure push' and empty' making them $\mathbb{M}$-compatible (ie. $\mathbb{M}$-resultric) through the *success* operator:

$$\begin{aligned}&\mathit{push}' :: ([\,a\,],a) \to \mathbb{M}\ ([\,a\,],1)\\&\mathit{push}' = \eta \cdot (\mathit{push} \vartriangle !)\\&\mathit{empty}' :: ([\,a\,],1) \to \mathbb{M}\ ([\,a\,],\mathbb{B})\\&\mathit{empty}' = \eta \cdot (\mathit{id} \vartriangle \mathit{empty}) \cdot \pi_1\end{aligned}$$

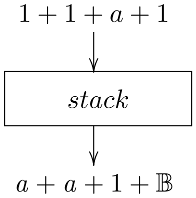
Components $= \sum$ *methods.* Now that we have the *methods* of a stack written as individual Mealy machines over the same monad and shared state space, we *add them up* to obtain the intended *stack* component [2]

$$\begin{aligned}&\mathit{stack} :: ([\,a\,],1+1+a+1) \to \mathbb{M}\ ([\,a\,],a+a+1+\mathbb{B})\\&\mathit{stack} = \mathit{pop}' \oplus \mathit{top}' \oplus \mathit{push}' \oplus \mathit{empty}'\end{aligned}$$

[1] Symbols $\bot$ and η pretty-print *Nothing* and *Just* of Haskell’s concrete syntax, respectively, cf. definition **data** $\mathbb{M}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$.

[2] Notation $x + y$ pretty-prints Haskell’s syntax for disjoint union, *Either* $x\ y$.

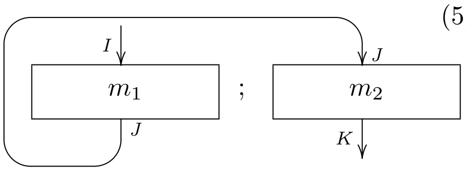
Before giving the details of the binary operator $\oplus$ which binds methods together, note that *stack* is also a (composite) Mealy machine (4), for $I = 1+1+a+1$ and $O = a+a+1+\mathbb{B}$. This I/O interfacing, pictured aside, captures the four alternatives which are available for interacting with a stack. Note how singleton types (1) at the input side mean *"do it!"* and at the output side mean *"done!"*.



Components such as *stack* arise as the *sum* of their methods, a MMM binary combinator whose definition in Haskell syntax is given aside. Isomorphism $\mathsf{dr}{::}(s, i{+}j) \to (s, i){+}(s, j)$ (resp. its converse dr°) distributes (resp. factorizes) the shared state across the sum of inputs (resp. outputs); $m_1 + m_2$ is the sum of m_1 and m_2 and "cozip" operator $\Delta :: \mathbb{F}\ a + \mathbb{F}\ b \to \mathbb{F}\ (a+b)$ promotes sums through functor $\mathbb{F}$.

$$\begin{array}{l}
\cdot \oplus \cdot :: (\mathit{Functor}\ \mathbb{F}) \Rightarrow \\
\quad \text{-- input machines} \\
\quad ((s,i) \to \mathbb{F}\ (s,o)) \to \\
\quad ((s,j) \to \mathbb{F}\ (s,p)) \to \\
\quad \text{-- output machine} \\
\quad (s, i+j) \to \mathbb{F}\ (s, o+p) \\
\quad \text{-- definition} \\
m_1 \oplus m_2 = (\mathbb{F}\ \mathsf{dr}^\circ) \cdot \Delta \cdot (m_1 + m_2) \cdot \mathsf{dr}
\end{array}$$

Systems = component compositions. Let us now consider the idea of building a system in which two stacks interact with each other, e.g. by popping from one and pushing the outcome onto the other.[3] For this another MMM combinator is needed taking two I/O compatible MMM m_1 and m_2 (with different internal states in general) and building a third one, $m_1 \,;\, m_2$, in which outputs of m_1 are sent to m_2 (5).



(5)

The type of this combinator as implemented in Haskell is given aside. It requires $\mathbb{F}$ to be a *strong* monad [15], a topic to be addressed later. Note how the output machine has a composite state pairing the states of the two input machines.

$$\begin{array}{l}
\cdot \,;\, \cdot :: (\mathit{Strong}\ \mathbb{F}, \mathit{Monad}\ \mathbb{F}) \Rightarrow \\
\quad \text{-- input machines} \\
\quad ((s,i) \to \mathbb{F}\ (s,j)) \to \\
\quad ((r,j) \to \mathbb{F}\ (r,k)) \to \\
\quad \text{-- output machine} \\
\quad ((s,r),i) \to \mathbb{F}\ ((s,r),k)
\end{array}$$

We defer to a later stage the analysis of the formal definition of this combinator, which is central to the principle of building components out of other components [2]. Instead, we build the composite machine already anticipated above,

$$m = \mathit{pop}' \,;\, \mathit{push}'$$

[3] This interaction will of course fail if the source stack is empty, but this is not our concern — monad $\mathbb{M}$ will take care of such effects.

which pops from a source stack ($m_1 = pop'$) and pushes onto a target stack ($m_2 = push'$). By running e.g. [4]

```
> m(([1],[2]),())
Just (([],[1,2]),())
```

we obtain the expected output and new state, while

```
> m(([],[2]),())
Nothing
```

fails, because the source stack is empty.

Faulty components. Let us finally consider the possibility of, due to hardware imperfection, pop' behaving in the source stack as expected, with probability p, and unexpectedly like top' with probability $1 - p$,

$$\begin{array}{l} pop'' :: \mathsf{P} \to ([\,a\,], 1) \to \mathbb{D}\ (\mathbb{M}\ ([\,a\,], a)) \\ pop''\ p = pop'\ {}_{p}\!\diamond\ top' \end{array}$$

recall (2). `P` pretty-prints the probability representation data type `ProbRep` of the PFP library and $\mathbb{D}$ denotes the (finite) *distribution* monad implemented in the same library. The choice operator $\cdot\ {}_{p}\!\diamond\ \cdot$ is the pointfree counterpart of a similar operator in PFP.

Concerning the target stack, the conjectured fault of $push'$ is that it does not push anything with probability $1 - q$:

$$\begin{array}{l} push'' :: \mathsf{P} \to ([\,a\,], a) \to \mathbb{D}\ (\mathbb{M}\ ([\,a\,], 1)) \\ push''\ q = push'\ {}_{q}\!\diamond\ !' \end{array}$$

where $!' = \eta \cdot (id \times !)$, of generic type $(s, a) \to \mathbb{M}\ (s, 1)$, is the promotion of the *bang* function ! to a MMM.

Note how pop'' and $push''$ have become “doubly” monadic in their cascading of the distribution ($\mathbb{D}$) and Maybe ($\mathbb{M}$) monads. To compose them as in $m = pop' \,;\, push'$ above we need a more sophisticated version of the semicolon combinator (aside) whose actual implementation is once again intentionally skipped. Thanks to this new combinator, we can build a faulty version of machine m above [5]

$$\begin{array}{l} \cdot \,;_D\, \cdot :: \\ \quad \text{-- input probabilistic MMMs} \\ \quad ((s, i) \to \mathbb{D}\ (\mathbb{M}\ (s, j))) \to \\ \quad ((r, j) \to \mathbb{D}\ (\mathbb{M}\ (r, k))) \to \\ \quad \text{-- output probabilistic MMM} \\ \quad ((s, r), i) \to \mathbb{D}\ (\mathbb{M}\ ((s, r), k)) \end{array}$$

$$m_2 = (pop''\ 0.95) \,;_D\, (push''\ 0.8)$$

and test it for the same composite state $([1], [2])$ as in the first experiment above, obtaining

[4] Recall that `()` is the Haskell notation for the unique inhabitant of type 1.

[5] The probabilities in these examples are chosen with no criterion apart from leading to distributions visible to the naked eye. By all means, 5% would be extremely high risk in realistic PRA [27], where only figures as small as 1.0E-7 become “acceptable”.

```
> m2(([1],[2]),())
 Just (([],[1,2]),())  76.0%
   Just (([],[2]),())  19.0%
Just (([1],[1,2]),())   4.0%
  Just (([1],[2]),())   1.0%
```

The simulation shows that the overall risk of faulty behaviour is 24% $(1-0.76)$, structured as 1%: both stacks misbehave; 4%: source stack misbehaves; 19%: target stack misbehaves. As expected, the second experiment

```
> m2(([],[2]),())
Nothing 100.0%
```

is *always catastrophic* (again popping from an empty stack).

Summing up: our animation in Haskell has been able to *simulate* fault propagation between two stack components with different fault patterns arising from conjectured hardware imperfections. In the sequel we will want to *reason* about such fault propagation rather than just simulate it.

4 Related work and research questions

Elsewhere [21, 20] we have shown that fault propagation can be reasoned about for functional programs of a particular kind — they are inductive extensions (termed *folds* or *catamorphisms*) of given *algebras*. In particular, the *linear (matrix) algebra of programming* mentioned in the introduction is used to decide which laws of programming [5] hold probabilistically or to find side-conditions for them to hold.

In the current paper we are faced with the dual situation: our programs are *coalgebras* and we want to *observe* and compare their behaviour expressed by *unfolds* (also known as *anamorphisms*) which tell how likely particular execution traces are. In particular, we want to be able to ascertain which (different) machines exhibit the same (probabilistic) traces for the same starting states (trace equivalence).

Looking at the types of $push''$, m_2 etc. above we realize that our MMMs have become probabilistic, leading to coalgebras of general shape

$$S \to (\mathbb{D}(\mathbb{F}(S \times O)))^I \tag{6}$$

This leads into our main research questions: *How tractable (mathematically) is this doubly-monadic framework? Can $\mathbb{F}$ be any monad?*

Relatives of shape (6) have been studied elsewhere [26], namely *reactive probabilistic automata*, $S \to (\mathbb{M}\ (\mathbb{D}\ S))^I$; *generative probabilistic automata*, $S \to \mathbb{M}\ (\mathbb{D}\ (O \times S))$; *bundle systems*, $S \to \mathbb{D}\ (\mathbb{P}\ (O \times S))$ and so on. In a coalgebraic approach to weighted automata, reference [6] studies coalgebras of functor $S \to \mathbb{K} \times (\mathbb{K}_\omega^S)^I$ for $\mathbb{K}$ a field. Such coalgebras rely on the so-called *field valuation* (exponential) functor $\mathbb{K}_\omega^-$ calling for *vector spaces.*

Inspired by this approach, a similar framework was studied directly in suitable *categories of matrices* [22]. We will follow a similar strategy in the current paper concerning probabilistic MMMs and their combinators.

5 Composition

The essence of the component algebra of [2] is a notion of component *composition* stated in a coalgebraic, categorial setting. Let us briefly review this framework, instantiated for generic $\mathbb{F}$-branching Mealy machines (4).

Let $X \xrightarrow{\eta} \mathbb{F}X \xleftarrow{\mu} \mathbb{F}^2 X$ be a monad and m_1, m_2 be two machines (functions) of types $S \times I \to \mathbb{F}\ (S \times J)$ and $Q \times J \to \mathbb{F}\ (Q \times K)$, respectively. Abstracting from their internal states as in picture (5) above, these machines can be represented by the arrows $I \xrightarrow{m_1} J$ and $J \xrightarrow{m_2} K$, respectively. Their *composition* by $I \xrightarrow{m_1 ; m_2} K$ is a machine with composite state $S \times Q$ built in the following way: first, m_1 is “wrapped” with the state Q of m_2,

$$\begin{tikzcd}
\mathbb{F}\ (S \times J) \times Q \arrow[d, "\tau_r"'] & (S \times I) \times Q \arrow[l, "m_1 \times id"'] & (S \times Q) \times I \arrow[l, "\mathsf{xr}"'] \arrow[dll, "g"] \\
\mathbb{F}\ ((S \times J) \times Q) & &
\end{tikzcd}$$

where xr is the obvious isomorphism and τ_r is the right *strength* of monad $\mathbb{F}$, $\tau_r : (\mathbb{F}\ A) \times B \to \mathbb{F}\ (A \times B)$, which therefore has to be a *strong* monad. The purpose of xr is to ensure the compound state and input I on the input side. In turn, m_2 is wrapped with the state of m_1,

$$\begin{tikzcd}
\mathbb{F}\ (S \times (Q \times K)) \arrow[d, "\mathbb{F}\ \mathsf{a}^\circ"'] & S \times \mathbb{F}\ (Q \times K) \arrow[l, "\tau_l"'] & S \times (Q \times J) \arrow[l, "id \times m_2"'] & (S \times J) \times Q \arrow[l, "\mathsf{xl}"'] \arrow[dlll, "f"] \\
\mathbb{F}\ ((S \times Q) \times K) & & &
\end{tikzcd}$$

where a° is the converse of isomorphism $\mathsf{a} : (A \times B) \times C \to A \times (B \times C)$ and xl is a variant of xr above. Finally, τ_l is the left strength of $\mathbb{F}$, $\tau_l : (B \times \mathbb{F}\ A) \to \mathbb{F}\ (B \times A)$.

Note how $\mathbb{F}\ \mathsf{a}^\circ$ ensures the compound state and type K on the output. In spite of the efforts of xl to approximate the input of contribution f to the output of the other (g), they do not match, as the latter is $\mathbb{F}$-*more complex* than the former.

$$\begin{tikzcd}
\mathbb{F}\ (\mathbb{F}\ C) \arrow[d, "\mu"'] & \mathbb{F}\ B \arrow[l, "\mathbb{F}\ f"'] \arrow[d, dash, dotted] & A \arrow[l, "g"'] \arrow[dll, bend left, "f \bullet g"] \\
\mathbb{F}\ C & B \arrow[l, "f"'] &
\end{tikzcd} \tag{7}$$

This suggests that f and g be composed using the *Kleisli composition* associated with monad $\mathbb{F}$, denoted by $f \bullet g$ and depicted in diagram (7). Thus we obtain the Haskell implementation of machine composition which was left unspecified in section 3:

$$m_1 \,;\, m_2 = ((\mathbb{F}\ \mathsf{a}^\circ) \cdot \tau_l \cdot (id \times m_2) \cdot \mathsf{xl}) \bullet (\tau_r \cdot (m_1 \times id) \cdot \mathsf{xr}) \tag{8}$$

An advantage of relying on Kleisli composition (7) is its rich algebra, forming a monoid with η

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{9}$$

$$f \bullet \eta = f = \eta \bullet f \tag{10}$$

and trading nicely with normal composition, cf. for instance

$$(\mathbb{F}\,f)\cdot(h\bullet k)=(\mathbb{F}\,f\cdot h)\bullet k \tag{11}$$
$$(f\cdot g)\bullet h=f\bullet(\mathbb{F}\,g\cdot h) \tag{12}$$

It turns out that Mealy machines too form a monoid whose binary operator is (8) and whose unit is the machine $J \xrightarrow{copy} J$ which faithfully passes its input to the output, never changing state:

$$\begin{array}{l} copy : 1\times J \to \mathbb{F}\,(1\times J) \\ copy = \eta \end{array}$$

However, such an algebraic structure holds up to behavioural equivalence only, denoted by symbol $\simeq$:

$$m\,;copy \;\simeq\; m \;\simeq\; copy\,;m \tag{13}$$
$$m\,;(n\,;p) \;\simeq\; (m\,;n)\,;p \tag{14}$$

$$\begin{tikzcd}
\mathbb{F}\,(S\times J) \arrow[d, "\mathbb{F}\,(h\times id)"'] & S\times I \arrow[l, "m_1"'] \arrow[d, "h\times id"] & S \arrow[d, "h"] \\
\mathbb{F}\,(Q\times J) & Q\times I \arrow[l, "m_2"] & Q
\end{tikzcd}$$

Behavioural equivalence can be established by defining *morphisms* between equivalent machines regarded as coalgebras. In general, given two $\mathbb{F}$-Mealy machines m_1 and m_2 (aside), a state transformation $h{:}S\to Q$ is a *morphism* between them if the diagram aside commutes.

For instance, to establish the first part of (13) it suffices to show that the natural isomorphism $\mathsf{lft}: A\times 1\to A$ is a morphism between $m\,;copy$ and m itself: $\mathbb{F}\,(\mathsf{lft}\times id)\cdot(m\,;copy)=m\cdot(\mathsf{lft}\times id)$. As example of pointfree calculation typical of $\simeq$ reasoning, the proof of this equality is given next, where $\mathsf{rgt}: 1\times A\to A$ is another natural isomorphism:

$$\begin{array}{cl}
& \mathbb{F}\,(\mathsf{lft}\times id)\cdot(m\,;copy)=m\cdot(\mathsf{lft}\times id) \\
\equiv & \quad\{\ \text{definition of composition (8)}\ \} \\
& \mathbb{F}\,(\mathsf{lft}\times id)\cdot(((\mathbb{F}\,\mathsf{a}^\circ)\cdot\tau_l\cdot(id\times\eta)\cdot\mathsf{xl})\bullet\tau_r\cdot(m\times id)\cdot\mathsf{xr})=m\cdot(\mathsf{lft}\times id) \\
\equiv & \quad\{\ \text{trading (11); } \tau_l\cdot(id\times\eta)=\eta\text{; } \mathsf{lft} \text{ commutes with } \mathsf{rgt} \text{ via } \mathsf{a}^\circ\ \} \\
& (\mathbb{F}\,(id\times\mathsf{rgt})\cdot\eta\cdot\mathsf{xl})\bullet(\tau_r\cdot(m\times id)\cdot\mathsf{xr})=m\cdot(\mathsf{lft}\times id) \\
\equiv & \quad\{\ \text{naturality of } \eta\text{; } (id\times\mathsf{rgt})\cdot\mathsf{xl}=\mathsf{lft}\ \} \\
& (\eta\cdot\mathsf{lft})\bullet(\tau_r\cdot(m\times id)\cdot\mathsf{xr})=m\cdot(\mathsf{lft}\times id) \\
\equiv & \quad\{\ \text{(12); unit } \eta \text{ (10)}\ \} \\
& (\mathbb{F}\,\mathsf{lft})\cdot\tau_r\cdot(m\times id)\cdot\mathsf{xr}=m\cdot(\mathsf{lft}\times id) \\
\equiv & \quad\{\ \mathbb{F}\,\mathsf{lft}\cdot\tau_r=\mathsf{lft}\ ;\ \mathsf{lft}\cdot\mathsf{xr}=\mathsf{lft}\times id\ \} \\
& \mathsf{lft}\cdot(m\times id)\cdot\mathsf{xr}=m\cdot\mathsf{lft}\cdot\mathsf{xr}
\end{array}$$

$\equiv \quad \{$ cancel xr; naturality of lft: $\mathsf{lft} \cdot (m \times id) = m \cdot \mathsf{lft}$ $\}$

$\quad true$

The component algebra of [2] contains several other combinators which are of interest. For economy of space we will restrict ourselves to composition, as this is enough for our main point in the paper: what is the impact in the component algebra of [2] of having *faulty* Mealy machines as models of components?

6 Composing non-deterministic components

Recall from the motivation how we have simulated faulty composition of $\mathbb{M}$-Mealy machines on top of the PFP Haskell library. In general, this means handling machines of type $I \rightarrow J$, that is, functions of type $Q \times I \rightarrow \mathbb{D}\ (\mathbb{F}\ (Q \times J))$ for some space state Q, where before we had $Q \times I \rightarrow \mathbb{F}\ (Q \times J)$. Thus, further to monad $\mathbb{F}$, another monad is around, the *distribution* monad $\mathbb{D}$.

Generalizing even further, we want to consider machines of type

$$Q \times I \rightarrow \mathbb{T}\ (\mathbb{F}\ (Q \times J)) \tag{15}$$

where monad $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}\ X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2 X$ caters for *transitional* effects (how the machine evolves) and monad $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{T}\ X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{T}^2 X$ specifies the *branching type* of the system [12]. A typical instance is $\mathbb{T} = \mathbb{P}$ (powerset) and $\mathbb{F} = \mathbb{M} = (1+)$ (‘maybe’), that is, we have machines

$$m : Q \times I \rightarrow \mathbb{P}\ (1 + Q \times J) \tag{16}$$

which are reactive, non-deterministic finite state automata with explicit termination.

Note, however, that m (16) could alternatively be specified as a *binary relation* R of type $Q \times I \rightarrow 1 + Q \times J$ of which m is the *power transpose* [5], following the equivalence

$$R = \lceil m \rceil \quad \equiv \quad \langle \forall b, a :: b\ R\ a \equiv b\ \in\ m\ a \rangle \tag{17}$$

which tells that a set-valued function m is uniquely represented by a binary relation $R = \lceil m \rceil$ and vice-versa. Moreover, $\lceil m \bullet n \rceil = \lceil m \rceil \cdot \lceil n \rceil$ holds, where $m \bullet n$ means the Kleisli composition of two *set-valued functions* and $\lceil m \rceil \cdot \lceil n \rceil$ is the *relational* composition of the corresponding *binary relations*:

$$b\ (R \cdot S)\ a \quad \equiv \quad \langle \exists c :: b\ R\ c \wedge c\ S\ a \rangle \tag{18}$$

In categorial-speak, this means that the category of binary relations coincides with the Kleisli category of the powerset monad $X \xrightarrow{sing} \mathbb{P}\ X \xleftarrow{dunion} \mathbb{P}^2 X$ where $sing\ a = \{a\}$ and $dunion\ S = \bigcup_{s \in S} s$.

Back to (16), the advantage of "thinking relationally" is that machine m can be "replaced" by the relation $\lceil m \rceil : Q \times I \rightarrow 1 + Q \times J$ from whose (relational) type the powerset has vanished. So, in a sense, it is as if we were back to the situation where only $\mathbb{M} = (1+)$ is present.

How do relational, $\mathbb{M}$-machines compose? Recall from (8) that machine composition relies on Kleisli composition (7) — in this case, of monad $\mathbb{M}\ X = 1 + X$, with structure $X \xrightarrow{i_2} 1 + X \xleftarrow{[i_1, id]} 1 + (1 + X)$, where i_1 and i_2 are the injections associated to binary sums. Thus

$$f \bullet g = [i_1, id] \cdot (id + f) \cdot g = [i_1, f] \cdot g \qquad (19)$$

where $[f, g]$ is the *junc* combinator satisfying $[f, g] \cdot i_1 = f$ and $[f, g] \cdot i_2 = g$. How about relations? Consider evaluating expression $[i_1, f] \cdot g$ for f replaced by some relation $1 + B \xleftarrow{R} C$,[6] g replaced by some other relation $1 + C \xleftarrow{S} A$ and functional composition replaced by relational composition:

$$R \bullet S = [i_1, R] \cdot S \qquad (20)$$

Unfolding $[i_1, R]$ to $i_1 \cdot i_1^\circ \cup R \cdot i_2^\circ$ — where R° denotes the converse of R — one obtains, abbreviating by $*$ the application of i_1 to the unique inhabitant of singleton type 1:

$$y\ (R \bullet S)\ a \quad \equiv \quad (y = *) \wedge (*\ S\ a) \vee \langle \exists c :: (y\ R\ c) \wedge ((i_2\ c)\ S\ a) \rangle$$

In words: composition $R \bullet S$ is doomed to fail wherever S fails; otherwise, it will fail where R fails. For the same input, $R \bullet S$ may *both* succeed or fail.

Summing up: we have encoded the Kleisli composition of a monad ($\mathbb{M}$) not in the category of sets and functions but in the Kleisli category of another monad (the powerset) which eventually we found familiar with — we met *relational algebra* there. Back to (8), can think of $\mathbb{M}$-monadic Mealy machines as *binary relations* which compose (as machines) according to definition

$$S\ ;R \quad = \quad [i_1, (id + \mathsf{a}^\circ) \cdot \tau_l \cdot (id \times R) \cdot \mathsf{xl}] \cdot \tau_r \cdot (S \times id) \cdot \mathsf{xr} \qquad (21)$$

where all dots mean relational composition (18).[7]

7 Composing probabilistic components

For the above constructions to help in reasoning about non-deterministic components we have to check if the properties of monad $\mathbb{M}$ remain intact once encoded

[6] Relations of this type *express the possibility that for some inputs, both termination and nontermination are possible* [that is] *relations from legal states to a "lifted" state set containing all legal states and in addition one "illegal state" standing for nontermination* [13].

[7] As usual, every function symbol f in (21) should be regarded as the homonym relation f such that $b\ f\ a$ holds iff $b = f\ a$.

relationally, extensive to the properties of the *strength* operators also present in (21).

Prior to this, however, let us not forget that our aim is to *prepare relational algebra for “just good enough” hardware* and its imprecision, calling for a probabilistic treatment of faults. In this direction, we should also check the scenario of the previous section once the *distribution* monad replaces the powerset,

$$m : Q \times I \to \mathbb{D}\,(1 + Q \times J) \tag{22}$$

whereupon non-deterministic branching becomes weighted with probabilities indicating the likelihood of state transitions, recall (2). So, m is now a distribution-valued function. We assume below that such distributions have countable support.[8]

It turns out that the strategy to cope with this situation is similar to that of the previous section: distribution-valued functions are adjoint to so-called *column stochastic* (CS) matrices, which represent the inhabitants of the Kleisli category associated with monad $\mathbb{D}$; and, for this monad, Kleisli composition corresponds to matrix *composition*, usually termed matrix *multiplication*:

$$b\ (M \cdot N)\ a \quad\equiv\quad \langle \textstyle\sum c :: (b\ M\ c) \times (c\ N\ a)\rangle \tag{23}$$

In this formula, both M and N are matrices. We prefer to denote the cell in (say) M addressed by row b and column a by the infix notation $b\ M\ a$, rather than the customary $M(b,a)$ or $M_{b,a}$. This stresses on the notational proximity with relations: matrices are just weighted relations.[9]

Summing up: in the same way the “Kleisli lifting” of section 6 makes the *powerset* monad implicit, leading into *relational algebra*, the same lifting now hides the *distribution* monad and leads to the *linear algebra* of CS matrices [21], under the universal correspondence

$$M = \lceil f \rceil \quad\equiv\quad \langle \forall b, a :: b\ M\ a = (f\ a)\ b\rangle \tag{24}$$

where $f : A \to \mathbb{D}\ B$ is a probabilistic function and $\mathbb{D}\ B$ is the set of all distributions on B with countable support; that is, for every $a \in A$, $f\ a = \mu$ where $\mu : B \to [0,1]$ is a function such that $\sum_{b\,\in\,B} m\ b = 1$.

Correspondence (24) establishes the isomorphism

$$A \to \mathbb{D}\ B \quad\cong\quad A \to B \tag{25}$$

where on the left hand side we have $\mathbb{D}$-valued functions and on the right hand side $A \to B$ denotes the set of all CS matrices with columns indexed by A, rows indexed by B and cells taking values from the interval $[0,1]$.[10] This *matrices as*

[8] This is reasonable in the sense that they arise from a finite number of applications of the choice (2) operator.

[9] Reference [22] argues in this direction by adapting rules typical of relational algebra to linear algebra.

[10] Each such column represents a distribution and therefore adds up to 1, as written above.

arrows approach [17] regards them as *morphisms* of suitable categories (of typed matrices). In the current paper we only consider matrices on the interval $[0, 1]$ subject to the *column stochasticity* constraint expressed above.

With no further detours let us adapt definition (20) of (relational) $\mathbb{M}$-Kleisli composition to the corresponding definition in linear algebra, where relation R gives place to matrix $1 + B \xleftarrow{M} C$, relation S to matrix $1 + C \xleftarrow{N} A$ and the little dot now denotes matrix composition (23):

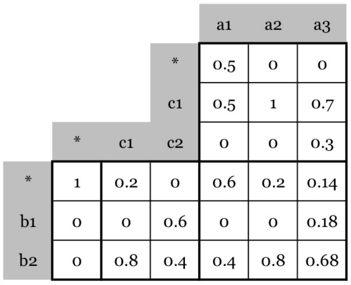
$$M \bullet N = [i_1|M] \cdot N \tag{26}$$

The reader may wonder about how does injection i_1 (a function) fit into a linear algebra expression (26). The explanation is the same as for functions in relational expressions: every function $f : A \to B$ is uniquely represented by the homonym matrix f defined by $b\ f\ a = 1$ if $b = f\ a$ and 0 otherwise.[11] Combinator $[M|N]$ occurring in (26) means the juxtaposition of matrices M and N, which therefore have to exhibit the same output type (and thus the same number of rows). Similarly to relations, it decomposes into $[M|N] = M \cdot i_1^\circ + N \cdot i_2^\circ$ where addition of matrices is the obvious cell-wise operation and the converse M° of a matrix M swaps its rows with columns (it is commonly known as the *transpose* of M). Because matrix multiplication is bilinear, we obtain $M \bullet N = i_1 \cdot i_1^\circ \cdot N + M \cdot i_2^\circ \cdot N$ and therefore the following pointwise version of (26)

$$y\ (M \bullet N)\ a \quad = \quad (y = *) \times (*\ N\ a) + \langle \sum c :: (y\ M\ c) \times ((i_2\ c)\ N\ a) \rangle$$

where $*$ is the same abbreviation used before and term $y = *$ evaluates to 1 if the equality holds and to 0 otherwise.[12]

The picture aside shows an example of probabilistic, $\mathbb{M}$-Kleisli composition of two matrices $N : \{a_1, a_2, a_3\} \to 1 + \{c_1, c_2\}$ and $M : \{c_1, c_2\} \to 1 + \{b_1, b_2\}$. Injection $i_1 : 1 \to 1 + \{b_1, b_2\}$ is the leftmost column vector. Note how, for input a_1, there is 60% probability of $M \bullet N$ failing, partly due (50%) to N failing or (50%) to passing output c_1 to M, which for such an input has 20% probability of failing again.



				a1	a2	a3
			*	0.5	0	0
			c1	0.5	1	0.7
	*	c1	c2	0	0	0.3
*	1	0.2	0	0.6	0.2	0.14
b1	0	0	0.6	0	0	0.18
b2	0	0.8	0.4	0.4	0.8	0.68

As before with relations, we can think of probabilistic $\mathbb{M}$-monadic Mealy machines as column stochastic matrices which compose (matricially) as follows

$$N \mathbin{\boldsymbol{;}} M \quad = \quad [i_1|(id \oplus \mathsf{a}^\circ) \cdot \tau_l \cdot (id \otimes M) \cdot \mathsf{xl}] \cdot \tau_r \cdot (N \otimes id) \cdot \mathsf{xr} \tag{27}$$

where relational product becomes matrix Kronecker product

$$(y, x)(M \otimes N)(b, a) = (yMb) \times (xNa) \tag{28}$$

[11] See section 8 for a technically more detailed explanation.

[12] See [22, 20] for a number of useful rules interfacing index-free and index-wise matrix notation. Such rules, expressed in the style of the Eindhoven quantifier calculus [1], provide evidence of the safe mix among matrix, predicate and function notation in typed linear algebra.

and relational sum gives place to matrix direct sum, $M \oplus N = \left[\begin{array}{c|c} M & 0 \\ \hline 0 & N \end{array}\right]$.

8 Monads in relational/linear algebra

The evolution from relational to (typed) linear algebra proposed in the previous sections corresponds to moving from non-deterministic choice (1) to probabilistic choice (2). The latter can now be defined matricially, for probabilistic f and g: $\lceil f\ {}_p\!\diamond g\rceil = p \otimes \lceil f\rceil + (1-p) \otimes \lceil g\rceil$.

A generic strategy can be identified: having a notion of composition (8) for machines of type $Q \times I \to \mathbb{F}\ (Q \times J)$ (4), where monad $\mathbb{F}$ captures their *transition* pattern, we want to *reuse* such a definition for more sophisticated machines of type $Q \times I \to \mathbb{T}\ (\mathbb{F}\ (Q \times J))$ (15) by porting it "as is" to the Kleisli category of the extra monad $\mathbb{T}$ which captures the *branching* structure.

For this to make sense we must be sure that the *lifting* of monad $\mathbb{F}$ by $\mathbb{T}$ still is a monad in the Kleisli category of $\mathbb{T}$. In general, let $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{T}X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{T}^2 X$ and $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2 X$ be two monads in a category C, and let $\mathsf{C}^\flat$ denote the Kleisli category induced by $\mathbb{T}$. Denote by $B \xleftarrow{f^\flat} A$ the morphism in $\mathsf{C}^\flat$ corresponding to $\mathbb{T}B \xleftarrow{f} A$ in C and define:

$$f^\flat \cdot g^\flat = (f \bullet g)^\flat = (\mu_{\mathbb{T}} \cdot \mathbb{T}\, f \cdot g)^\flat \tag{29}$$

For *any* morphism $B \xleftarrow{f} A$ in C define its lifting to $\mathsf{C}^\flat$ by $\overline{f} = (\eta_{\mathbb{T}} \cdot f)^\flat$. As in [12], assume a distributive law $\lambda : \mathbb{F}\mathbb{T} \to \mathbb{T}\mathbb{F}$ and define, for each endofunctor $\mathbb{F}$ in C, its lifting $\overline{\mathbb{F}}$ to $\mathsf{C}^\flat$ by

$$\overline{\mathbb{F}}(f^\flat) \;=\; (\lambda \cdot \mathbb{F}\ f)^\flat \quad \text{cf. diagram} \quad \mathbb{T}\mathbb{F}B \xleftarrow{\lambda} \mathbb{F}\mathbb{T}B \xleftarrow{\mathbb{F}\ f} \mathbb{F}\ A \tag{30}$$

for $\mathbb{T}\ B \xleftarrow{f} A$. For $\overline{\mathbb{F}}$ to be a functor in $\mathsf{C}^\flat$ two conditions must hold [12]:

$$\lambda \cdot \mathbb{F}\ \eta_{\mathbb{T}} = \eta_{\mathbb{T}} \tag{31}$$

$$\mu_{\mathbb{T}} \cdot \mathbb{T}\lambda \cdot \lambda = \lambda \cdot \mathbb{F}\ \mu_{\mathbb{T}} \tag{32}$$

We want to check under what conditions monad $\mathbb{F}$ lifts to a monad in the Kleisli of $\mathbb{T}$, that is, whether

$$X \xrightarrow{\overline{\eta_{\mathbb{F}}}=(\eta_{\mathbb{T}}\cdot\eta_{\mathbb{F}})^\flat} \overline{F}X \xleftarrow{\overline{\mu_{\mathbb{F}}}=(\eta_{\mathbb{T}}\cdot\mu_{\mathbb{F}})^\flat} \overline{F}^2 X \tag{33}$$

is a monad in $\mathsf{C}^\flat$. The standard monadic laws, e.g. $\overline{\mu_{\mathbb{F}}} \cdot \overline{\eta_{\mathbb{F}}} = id$, hold by construction.[13] It can be checked that the remaining natural laws, $\overline{\mathbb{F}}\ f^\flat \cdot \overline{\eta_{\mathbb{F}}} = \overline{\eta_{\mathbb{F}}} \cdot f^\flat$

[13] The general rule is that $\overline{f} = (\eta_{\mathbb{T}} \cdot f)^\flat$ embeds C in $\mathsf{C}^\flat$. Thus the lifting of e.g. an equality $f \cdot g = h$ in C, that is $\overline{f} \cdot \overline{g} = \overline{h}$ in $\mathsf{C}^\flat$, is $(\eta_{\mathbb{T}} \cdot f)^\flat \cdot (\eta_{\mathbb{T}} \cdot f)^\flat = (\eta_{\mathbb{T}} \cdot h)^\flat$

and $\overline{\mathbb{F}}\, f^\flat \cdot \overline{\mu_\mathbb{F}} = \overline{\mu_\mathbb{F}} \cdot (\overline{\mathbb{F}}^2\, f^\flat)$ are ensured by two "monad-monad" compatibility conditions:[14]

$$\lambda \cdot \eta_\mathbb{F} = \mathbb{T}\eta_\mathbb{F} \tag{34}$$

$$\mathbb{T}\mu_\mathbb{F} \cdot \lambda \cdot \mathbb{F}\lambda = \lambda \cdot \mu_\mathbb{F} \tag{35}$$

Recall that, in our component algebra illustration, $\mathbb{F}$ is the *maybe* monad $\mathbb{M}$ and $\mathbb{T}$ is one of either the powerset or distribution monads. From a result in [12] it can immediately be shown that the distributive law $\lambda : 1 + \mathbb{T}\ X \to \mathbb{T}\ (1 + X)$ between $\mathbb{M}$ and any other monad $\mathbb{T}$, $\lambda = [\eta_\mathbb{T} \cdot i_1, \mathbb{T}\ i_2]$, satisfies (31,32) in both cases.[15]

It is also easy to show that $\mathbb{M}$ satisfies (34,35) for any $\mathbb{T}$. [16] So nondeterministic (resp. probabilistic) composition of $\mathbb{M}$-monadic Mealy machines regarded as binary relations (resp. matrices) given by monadic definitions (21) (resp. (27)) is sound, where $\mathbb{M}$ can be *generalized* to any $\mathbb{T}$-liftable monad $\mathbb{F}$ satisfying (34,35).

In retrospect, recall from the motivation that we went as far as simulating probabilistic composition of $\mathbb{M}$-machines in Haskell using the operator $m_1 ;_D m_2$, the probabilistic evolution of $m_1 ; m_2$ (8). Although we have not seen its actual definition, we can say that fact $\lceil m_1 ;_D m_2 \rceil = \lceil m_1 \rceil ; \lceil m_2 \rceil$ holds, where composition $\lceil m_1 \rceil ; \lceil m_2 \rceil$ is given by (27).[17]

Instead of staying in the *original category* and elaborating the definition to the probabilistic case we have kept the *original definition* by changing category. The advantage is that all *probabilistic accounting* is silently carried out by (monadic) matrix composition and is not our concern.

9 Strong monads in relational/linear algebra

We are not yet done, however: definition (27) is *strongly* monadic and we need to know in what sense *strength* is preserved through Kleisli-lifting. The question is, which strong monads ($\mathbb{F}$) are still strong once lifted to the Kleisli category of another monad ($\mathbb{T}$)? Recall that the two strengths

$$\tau_l : B \times \mathbb{F}\ A \to \mathbb{F}\ (B \times A)$$
$$\tau_r : \mathbb{F}\ A \times B \to \mathbb{F}\ (A \times B)$$

distribute context (B) across $\mathbb{F}$-data structures. Their basic properties, $\mathbb{F}\ \mathsf{lft} \cdot \tau_r = \mathsf{lft}$ and $\mathbb{F}\ \mathsf{a}^\circ \cdot \tau_r = \tau_r \cdot (\tau_r \times id) \cdot \mathsf{a}^\circ$ (similarly for τ_l) are preserved by their liftings

which, by (29), reduces to the original $f \cdot g = h$. Within the image of the embedding, everything in $\mathsf{C}^\flat$ "works as if" in C. Our previous use of a function symbol f as denotation of the *corresponding* relation or matrix $\overline{f}$ is a very convenient abuse of notation.

[14] See e.g. [28], among the literature emerging from [4].

[15] This happens because the powerset and distribution monads are *commutative*.

[16] Concerning (34): $[\eta_\mathbb{T} \cdot i_1, \mathbb{T}\ i_2] \cdot i_2 = \mathbb{T}\ i_2$; concerning (35): $\mathbb{T}\ [i_1, id] \cdot [\eta_\mathbb{T} \cdot i_1, \mathbb{T}\ i_2 \cdot \lambda] = [\eta_\mathbb{T} \cdot [i_1, id] \cdot i_1, \lambda] = [\lambda \cdot i_1, \lambda] = \lambda \cdot [i_1, id]$.

[17] The actual implementation of $\cdot ;_D \cdot$ in the Haskell simulator follows *verbatim* pointfree formula (27) carefully using the encodings of section 8.

$\overline{\tau_r}$ and $\overline{\tau_l}$, recall footnote 13. So, what may fail is their *naturality*, e.g.

$$\overline{\tau_l} \cdot (N \otimes \overline{\mathbb{F}}\ M) = \overline{\mathbb{F}}\ (N \otimes M) \cdot \overline{\tau_l} \tag{36}$$

where M and N are arbitrary column-stochastic matrices. This is important because strength naturality is essential to many proofs of the component algebra of [2], for instance to that of (14). All proofs go in the style of that given for (13), with matrices in the place of functions.

Let us investigate (36) for $\mathbb{F} = \mathbb{M}$, in which case we have $\overline{\tau_l} = (\overline{!} \oplus id) \cdot \overline{\mathsf{dr}}$, of type $B \times (1 + A) \rightarrow 1 + B \times A$, where $\mathsf{dr} : A \times (C + B) \rightarrow A \times C + A \times B$ is the obvious isomorphism and $\overline{!} : A \rightarrow 1$ lifts the "bang" function to the row vector of its type wholly filled with 1s.

The naturality of τ_l (hereupon we drop the lifting bars, under the convention we have used before) arises from that of dr and of $! \oplus id$. The naturality of dr is easy to prove from that of its converse using relational/matrix biproducts [17]. Concerning $! \oplus id$:

$$\begin{array}{cl}
& (id \oplus N) \cdot (! \oplus id) = (! \oplus id) \cdot (M \oplus N) \\
\equiv & \quad \{ \text{ bifunctor } \cdot \oplus \cdot \ \} \\
& ! \oplus N = (! \cdot M) \oplus N \\
\equiv & \quad \{ \ ! \cdot M = ! \text{ because } M \text{ is assumed column stochastic [22] } \} \\
& \mathit{true}
\end{array}$$

The calculation for τ_r is similar. Thus $\overline{\mathbb{M}}$ is strong.

Note, however, that not every natural transformation remains natural once "Kleisli lifted". A very simple example is the diagonal function $\Delta : A \rightarrow A \times A$, $\Delta\ a = (a, a)$, shown aside for $A = \mathbb{B}$. Its natural property, $(M \otimes M) \cdot \Delta = \Delta \cdot M$ does not hold because $(y, z)\ ((M \otimes M) \cdot \Delta)\ x = (y\ M\ x) \times (z\ M\ x)$ on the left hand side, and $(y, z)\ (\Delta \cdot M)\ x = (\textbf{if}\ y = z\ \textbf{then}\ y\ M\ x\ \textbf{else}\ 0)$ on the right hand side. Thus the distributions captured by $(M \otimes M) \cdot \Delta$ have, in general, larger support. The same happens, of course, for relations: $(R \otimes R) \cdot \Delta \supseteq \Delta \cdot R$ holds but the converse inclusion does not.

$$\begin{array}{c|cc|}
 & F & T \\
(F,F) & 1 & 0 \\
(F,T) & 0 & 0 \\
(T,F) & 0 & 0 \\
(T,T) & 0 & 1
\end{array}$$

In the terminology of categorial physics, Δ fails to be a *uniform copying operation* [8]. This has to do with the fact that the *pairing* operator $(f \vartriangle g)\ a = (f\ \ a, g\ \ a)$ (note that $\Delta = id \vartriangle id$) does not form a categorial product once Kleisli-lifted [20]. The corresponding matrix operation is the so-called Khatri-Rao matrix product, defined by $(b, c)\ (M \vartriangle N)\ a = (b\ M\ c) \times (c\ N\ a)$. In relational algebra it is known as (strict) *fork* [11, 25]. [18]

[18] Both *Khatri-Rao* and *fork* can be regarded as the lifting of the pairing operator, $f^\flat \mathbin{\overline{\vartriangle}} g^\flat = (dstr \cdot (f \vartriangle g))^\flat$ where $dstr$ denotes the "double strength" of a commutative monad [12], a class of monads which includes both $\mathbb{D}$ and $\mathbb{P}$.

10 Conclusions and future work

Faced with the need to quantify software (un)reliability in presence of faults arising from (intentionally) inexact hardware, the semantics of software systems has to evolve towards weighted nondeterminism, for instance in a probabilistic way.

This paper proposes that such *semantics evolution* be obtained without sacrificing the simplicity of the original (qualitative) semantics definition. The idea is to keep quantification *implicit* rather than explicit, the trick being a change of category: instead of the category of sets where traditional (e.g. coalgebraic) semantics is expressed, we change to a suitable category (e.g. of matrices) tuned to the specific quantitative (e.g. probabilistic) effect.

Technically, this *"keep definition, change category"* approach consists of investing in the Kleisli category of the monad chosen to capture the new (e.g. quantitative) effect. The approach is useful because such a *Kleisli lifting* leads to rich algebraic theories: to *relational algebra* and *linear algebra*[19] in particular, both offering a useful *pointfree styled* calculus.

The approach is illustrated in the paper by enriching an existing software component calculus with fault propagation, by lifting it through a discrete distribution monad. As the original semantics are already monadic and coalgebraic, "keeping the definitions" entails monad-monad lifting.

Ideally, the proposed Kleisli-lifting should preserve theories, not only definitions (the theory of component behavioural equivalence of [2], in our case). But things are not so immediate in presence of *tupling* (cf. strong monads) as products become *weak* once lifted. Weak tupling calls for a wider perspective, interestingly bridging relational algebra to *categorial quantum physics* under the umbrella of *monoidal* categories. Thus the remarks by Coecke and Paquette, in their *Categories for the Practising Physicist* [8]:

> *Rel* [the category of relations] possesses more 'quantum features' than the category *Set* of sets and functions [...] The categories *FdHilb* [of finite dimensional Hilbert spaces] and *Rel* moreover admit a categorical matrix calculus.

Future work. This paper is part of a research line aiming at promoting linear algebra as the "natural" evolution of (pointfree) relational algebra towards quantitative reasoning in the software sciences. Work in this direction is still in its infancy [21, 17, 22, 20].

A full-fledged coalgebraic trace semantics for probabilistic, component oriented software systems will call for sub-distributions and, more generally, to measure theory [12, 23, 14]. The main result of [12] — that the (final) behaviour coalgebra in a Kleisli category is given by an initial algebra in sets — is central to the approach.

The connection to categorial quantum physics and monoidal categories [8] should be exploited, in particular concerning partial orders defined for quantum states which could be used to support a notion of refinement.

[19] This carries over to more sophisticated algebras and monads, for instance that of *stochastic relations*, the "Kleisli lifting" of the Giry monad [23].

On the applications side, it would be interesting to address case studies such as that of [18], the verification of a persistent memory manager (in IBM's 4765 secure coprocessor) in face of restarts and hardware failures, using probabilistic component algebra. As the authors of [18] write, the inclusion of hardware failures incurs a significant jump in system complexity.

Acknowledgments This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the *FCT - Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020537.

Feedback and exchange of ideas with Tarmo Uustalu, Alexandra Silva and Luís Barbosa are gratefully acknowledged.

References

[1] Backhouse, R., Michaelis, D.: Exercises in quantifier manipulation. In: Uustalu, T. (ed.) MPC'06, LNCS, vol. 4014, pp. 70–81. Springer (2006)

[2] Barbosa, L.: Towards a Calculus of State-based Software Components. JUCS 9(8), 891–909 (August 2003)

[3] Barbosa, L., Oliveira, J.: Transposing Partial Components — an Exercise on Coalgebraic Refinement. Theor. Comp. Sci. 365(1), 2–22 (2006)

[4] Beck, J.: Distributive laws. In: Eckmann, B. (ed.) Seminar on Triples and Categorical Homology Theory, Lecture Notes in Mathematics, vol. 80, pp. 119–140. Springer (1969)

[5] Bird, R., de Moor, O.: Algebra of Programming. Series in Computer Science, Prentice-Hall International (1997)

[6] Bonchi, F., Bonsangue, M., Boreale, M., Rutten, J., Silva, A.: A coalgebraic perspective on linear weighted automata. Inf. & Comp. 211, 77–105 (2012)

[7] Brink, C., Kahl, W., Schmidt, G. (eds.): Relational methods in computer science. Springer-Verlag New York, Inc., New York, NY, USA (1997)

[8] Coecke, B. (ed.): New Structures for Physics. No. 831 in Lecture Notes in Physics, Springer (2011)

[9] Cortellessa, V., Grassi, V.: A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In: Component-Based Software Engineering, LNCS, vol. 4608, pp. 140–156 (2007)

[10] Erwig, M., Kollmansberger, S.: Functional pearls: Probabilistic functional programming in Haskell. J. Funct. Program. 16, 21–34 (January 2006)

[11] Frias, M., Baum, G., Haeberer, A.: Fork algebras in algebra, logic and computer science. Fundam. Inform. pp. 1–25 (1997)

[12] Hasuo, I., Jacobs, B., Sokolova, A.: Generic trace semantics via coinduction. Logical Methods in Computer Science 3(4), 1–36 (2007)

[13] Kahl, W.: Refinement and development of programs from relational specifications. ENTCS 44(3), 4.1–4.43 (2003)

[14] Kerstan, H., König, B.: Coalgebraic trace semantics for probabilistic transition systems based on measure theory. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. pp. 410–424. LNCS, Springer (2012)

[15] Kock, A.: Strong functors and monoidal monads. Archiv der Mathematik 23(1), 113–120 (1972), `http://dx.doi.org/10.1007/BF01304852`

[16] Lingamneni, A., Enz, C., Palem, K., Piguet, C.: Synthesizing parsimonious inexact circuits through probabilistic design techniques. ACM Trans. Embed. Comput. Syst. 12(2s), 93:1–93:26 (May 2013)

[17] Macedo, H., Oliveira, J.: Typing linear algebra: A biproduct-oriented approach. Science of Computer Programming 78(11), 2160–2191 (2013)

[18] Marić, O., Sprenger, C.: Verification of a transactional memory manager under hardware failures and restarts (2013), conference paper (submitted)

[19] McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer-Verlag (2005)

[20] Murta, D., Oliveira, J.N.: Calculating risk in functional programming. CoRR abs/1311.3687 (2013)

[21] Oliveira, J.: Towards a linear algebra of programming. Formal Aspects of Computing 24(4-6), 433–458 (2012),

[22] Oliveira, J.: Weighted automata as coalgebras in categories of matrices. Int. Journal of Found. of Comp. Science 24(06), 709–728 (2013)

[23] Panangaden, P.: Labelled Markov Processes. Imperial College Press (2009)

[24] Rutten, J.: Universal coalgebra: A theory of systems. Theor. Comp. Sci. 249(1), 3–80 (2000), (Revised version of CWI Techn. Rep. CS-R9652, 1996)

[25] Schmidt, G.: Relational Mathematics. No. 132 in Encyclopedia of Mathematics and its Applications, Cambridge University Press (November 2010)

[26] Sokolova, A.: Probabilistic systems coalgebraically: A survey. Theor. Comput. Sci. 412(38), 5095–5110 (2011)

[27] Stamatelatos, M., Dezfuli, H.: Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners (2011), NASA/SP-2011-3421, 2nd edition, December 2011.

[28] Tanaka, M.: Pseudo-Distributive Laws and a Unified Framework for Variable Binding. Ph.D. thesis, School of Informatics, University of Edinburgh (2005)