Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

"Keep definition, change category" — A practical approach to state-based system calculi

José Nuno Oliveira [a,*], Victor Cacciari Miraldo [b,1]

[a] High Assurance Software Laboratory, INESC TEC and University of Minho, Braga, Portugal
[b] Dep. of Information and Computing Sciences, Universiteit Utrecht, Utrecht, Netherlands

ARTICLE INFO

Article history:
Received 14 October 2014
Received in revised form 24 November 2015
Accepted 26 November 2015
Available online xxxx

Keywords:
Mealy automata
Kleisli category
Relation algebra
Linear algebra
Software calculi

ABSTRACT

Faced with the need to quantify software (un)reliability in the presence of faults, the semantics of state-based systems is urged to evolve towards quantified (e.g. probabilistic) nondeterminism. When one is approaching such semantics from a categorical perspective, this inevitably calls for some technical elaboration, in a monadic setting.

This paper proposes that such an evolution be undertaken without sacrificing the simplicity of the original (qualitative) definitions, by keeping quantification implicit rather than explicit. The approach is a monad lifting strategy whereby, under some conditions, definitions can be preserved provided the semantics *moves to another category*.

The technique is illustrated by showing how to introduce probabilism in an existing software component calculus, by moving to a suitable category of matrices and using linear algebra in the reasoning.

The paper also addresses the problem of preserving monadic *strength* in the move from original to target (Kleisli) categories, a topic which bears relationship to recent studies in categorial physics.

1. Introduction

The calculus of state-based systems has been a long quest in the theory of computing. The study of deterministic, non-deterministic, probabilistic and weighted automata are steps towards increasingly sophisticated mathematical models intended to express the complexity of the real-life situations they wish to control or mimic.

In the coalgebraic approach [40] a state-based system is regarded as a function of type $S \to \mathbb{F}\, S$ which expresses its *behavior pattern*, that is, how it evolves from the current state (S) to future states ($\mathbb{F}\, S$). In this setting, this evolution is mirrored in the more and more complex *functor* $\mathbb{F}$ which is required to capture the overall behavior:

- $\mathbb{F}\, S = S$ — the system is *deterministic* and total
- $\mathbb{F}\, S = 1 + S$ — the system is *deterministic* but partial (alternative 1 means *failure*)
- $\mathbb{F}\, S = \mathbb{P}\, S$ — the system is *non-deterministic* (where $\mathbb{P}\, S$ is the set of all finite subsets of S)
- $\mathbb{F}\, S = \mathbb{D}\, S$ — the system is *probabilistic* (where $\mathbb{D}\, S$ is the set of all *distributions* of S with finite support).

* Corresponding author.
E-mail addresses: jno@di.uminho.pt (J.N. Oliveira), v.cacciarimiraldo@uu.nl (V.C. Miraldo).
URLs: http://haslab.uminho.pt (J.N. Oliveira), http://www.cs.uu.nl (V.C. Miraldo).

[1] Partially supported by grant number UMINHO/BI/9/2014 in the context of FCT Project QAIS (Quantitative Analysis of Information Systems).

http://dx.doi.org/10.1016/j.jlamp.2015.11.007
2352-2208/

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

The behavior pattern of a complex coalgebraic system may combine two or more functors $\mathbb{F}$ above. A survey by Sokolova [43] identifies no less than thirteen such combinations in the literature, most of them concerned with discrete probabilism. As one would expect, the more $\mathbb{F}$s are involved in the functor expressing the *next state pattern*, the more intricate the corresponding formalization is.

This paper exploits a technique to tame such complexity based on the fact that all functors $\mathbb{F}$ above are *monads* and therefore associated to particular *Kleisli categories* [30]. However, as is well known, not every combination of monads forms a compound monad. Category theory is a generic mathematical framework based on a typed notion of *composition*. The slogan *"keep definition, change category"* emphasizes the stepwise compositionality of the proposed approach: once another monad is combined with $\mathbb{F}$ to capture another aspect of the behavior of the system, one keeps the definition of the previous step and *re-interprets* it inside the Kleisli category of the new monad.

The approach is attractive for two reasons: first, the structure of the "original" definition (written for the simplest case) is preserved and does not get convoluted; second, such Kleisli categories often have good potential for reasoning, as is the case of *relation algebra* (associated to the Kleisli category of the powerset monad) and *linear algebra* (similarly associated to the distribution monad and other weighted semantic models). The cost (if understood as such) is that of expressing the semantics of state-based systems in the *pointfree* language of category theory, putting emphasis on composition and other standard constructs.

Paper structure. This paper is an extension of a previous publication [38] which found its motivation in the need for state-based system calculi to adapt to new trends in circuit design that point towards tolerating some *imperfection* of hardware devices. This calls for semantic models able to cope with faulty behavior in a *quantitative* way, e.g. probabilistic. Sect. 2 and Sect. 3 recall such a motivation. The case study of [38] — the enrichment of a calculus of software components [3] towards fault propagation — is given in Sect. 4 and extended with more results in Sect. 5. The required monad–monad lifting strategy is also enriched in Sect. 8 concerning free monads. More insight into the problem of lifting strong monads is given in Sect. 9. For easy reference, Appendix A gives a minimal set of standard definitions required in the main text. Appendix B gives proofs of auxiliary or lengthy to prove results.

Contribution. This paper proposes that, similarly to what has happened with the increasing role of *relation algebra* in computer science [6,8,42], *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 [9,28]. So there is room for evolution rather than radical change.

The contributions of this paper include (a) a case study on such an evolution concerning a calculus of software components [3,4] intended for quantitative analysis of software reliability (sections 4 and 5); (b) a (generic) strategy for reducing the impact of the "probabilistic move" based on re-interpreting state-based system semantics in linear algebra through monadic "Kleisli-lifting", keeping as much of the original semantics definition as possible (sections 6 to 8); (c) a discussion on the loss (across Kleisli-lifting) of the naturality of operations involving pairing, a technical aspect of systems' modeling which needs attention (Sect. 9).

2. Motivation

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" [26].

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 suffices? 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* [26], 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 on 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 [44] 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 (as happened with other sciences in the past) that 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*

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

if it can *successfully carry out its own task as specified* [11]. But our italicized text is an inexact quotation of [11], 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 behaviors,

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

has to be replaced by probabilistic choice [31]

$$bad\ {}_p\diamond\ good \tag{2}$$

and the reasoning should be able to ensure that the probability p of bad behavior 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* [36]. 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 behavior. 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.

3. Coalgebraic approach

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 [11] 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. [3]), building on extensive work on automata using coalgebra theory [17,40].

As already mentioned, coalgebra theory offers 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 behavior of the system expressed by functor $\mathbb{F}$. For $\mathbb{F} = \mathbb{P}$, the powerset functor, f is the *power-transpose* [6] 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 [3] 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 [3] to a quantitative, probabilistic extension able to cope with the impact of *inexact circuit design* on software. As the survey by Sokolova [43] shows, probabilistic systems have been in the software research agenda for quite some time. From the available literature we rely on a paper [17] 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 [37] 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.

4. Case study

This section introduces our case study, the enrichment of the component algebra of [3] towards probabilistic software components. 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 Kollmannsberger [12]. 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.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

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 modeled as a (partial) algebra of finite lists written in Haskell syntax as follows

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

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? As in [3] we regard it as the singleton type 1 by pairing *push* with the uniquely defined (total and constant) function $! :: b \to 1$:

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

This definition relies on the *pairing* operator, $(f \vartriangle g)\ x = (f\ x, g\ x)$.

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

$$\begin{array}{l} empty' :: ([a], 1) \to ([a], \mathsf{B}) \\ empty' = (id \vartriangle empty) \cdot \pi_1 \end{array}$$

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 while checking for emptiness.

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 $*$ the inputs for which the function is undefined and otherwise signaling a successful computation using the monad's unit $\eta :: S \to \mathbb{M}\ S$[2]:

$$\begin{array}{l} (\Leftarrow) :: (a \to b) \to (a \to \mathsf{B}) \to a \to \mathbb{M}\ b \\ (f \Leftarrow p)\ a = \mathbf{if}\ p\ a\ \mathbf{then}\ (\eta \cdot f)\ a\ \mathbf{else}\ * \end{array}$$

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

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

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

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

we finally go back to pure $push'$ and $empty'$ making them $\mathbb{M}$-compatible (i.e. $\mathbb{M}$-resultric) through the success operator, i.e. the unit η of monad $\mathbb{M}$:

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

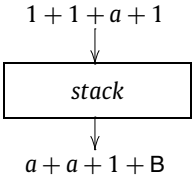
Components = $\sum$ methods. Now that we have the *methods* of a stack written as individual Mealy machines over the same monad and shared state space, we *add them up* to obtain the intended *stack* component.[3]

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

[3] Coproduct notation $x + y$ in the type declaration pretty-prints Haskell's syntax for disjoint union, *Either x y*.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

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



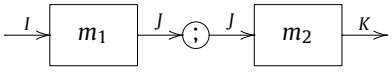
Before giving the details of the binary operator $\oplus$ which binds methods together, note that *stack* is also a (composite) Mealy machine (4), for $I = 1+1+a+1$ and $O = a+a+1+\mathsf{B}$. This I/O interfacing, pictured above, captures the four alternatives which are available for interacting with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

Components such as *stack* arise as the *sum* of their methods, a MMM binary combinator whose definition in Haskell syntax is[4]

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

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 (coproduct) of m_1 and m_2 and "cozip" operator $\mathsf{cozip} :: \mathbb{F}\,a + \mathbb{F}\,b \to \mathbb{F}\,(a+b)$ promotes sums through functor $\mathbb{F}$.[5]

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.[6] 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)

The type of this combinator as implemented in Haskell is

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

It requires $\mathbb{F}$ to be a *strong* monad [25], a topic to be addressed later. Note how the output machine has a composite state pairing the states of the two input machines.

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 [3]. As an example, we build the composite machine already anticipated above,

$$m = pop' ; push'$$

which pops from a source stack ($m_1 = pop'$) and pushes onto a target stack ($m_2 = push'$). By running e.g.[7]

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

[4] Components of this kind are identified as *separable* in [2], where they are regarded as *a collection of actions over a shared state space, each of them with a specific interface*. This means that one may also "extract" from the overall component any of its methods (i.e. sub-components).

[5] See Appendix B for the definitions of dr and cozip.

[6] 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.

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

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

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, due to hardware imperfection, of pop' behaving in the source stack of the previous example as expected, with probability p, and unexpectedly like top' with probability $1-p$,

$$pop'' :: \mathsf{P} \to ([a],1) \to \mathbb{D}\,(\mathbb{M}\,([a],a))$$
$$pop''\ p = pop'\ {}_p\diamond\ top'$$

expressed in the notation of (2). P is the probability interval $[0\,..\,1]$ and $\mathbb{D}$ denotes the (finite) *distribution* monad. The choice operator $({}_p\diamond)$ is defined by

$$(f\ {}_p\diamond\ g)\ x = \{(f\ x, p), (g\ x, 1-p)\}$$

where a distribution $\{(x_i, p_i)\}_{i\in S}$ is denoted by pairing each element of its support S with the corresponding probability.

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

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

where $!' = \eta \cdot (id \times !)$, of generic type $(s,a) \to \mathbb{M}\,(s,1)$, is the promotion of 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[8]:

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

Thanks to this new combinator, we can build a faulty version of machine m above [9]

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

Once we test it for the same composite state $([1],[2])$ as in the first experiment above, we obtain:

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

This simulation shows that the overall risk of faulty behavior is 24% ($1-0.76$), structured as: both stacks misbehave (1%); source stack misbehaves (4%); target stack misbehaves (19%). 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. Note that such animations may give false negatives in the sense of two faults canceling each other out and the result ending up "correct" for the wrong reason.

In the sequel we will want to *reason* about such fault propagation rather than just *simulate* it.

[8] Its actual implementation is once again intentionally skipped.

[9] 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 [44], where only figures as small as 1.0E–7 become "acceptable".

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

5. Component algebra

The essence of the component algebra of [2,3] is a notion of component *composition* stated in a coalgebraic, categorial setting. Below we briefly review this framework, instantiated for generic $\mathbb{F}$-branching Mealy machines (4).

Let $X \xrightarrow{\eta} \mathbb{F}X \xleftarrow{\mu} \mathbb{F}^2X$ 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. We will represent these machines by arrows $I \xrightarrow[S]{m_1} J$ and $J \xrightarrow[Q]{m_2} K$, respectively.[10] Then their *composition* $I \xrightarrow[S \times Q]{m_1;m_2} K$ is a machine with composite state $S \times Q$ built in the following way: first, we build $I \xrightarrow[S \times Q]{\text{extr } m_1} J$, the state-extension of m_1 with the state Q of m_2

$$\begin{array}{ccccc} \mathbb{F}((S \times J) \times Q) & \xleftarrow{\tau_r} & \mathbb{F}(S \times J) \times Q & \xleftarrow{m_1 \times id} & (S \times I) \times Q \\ {\scriptstyle \mathbb{F}\,\mathsf{xr}}\uparrow & & & & \uparrow{\scriptstyle \mathsf{xr}} \\ \mathbb{F}((S \times Q) \times J) & & \xleftarrow[\text{extr } m_1]{} & & (S \times Q) \times I \end{array}$$

where $\mathsf{xr} : (A \times B) \times C \to (A \times C) \times B$ 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.[11] The purpose of xr is to bring together the compound state and input I, on the input side. Note that $I \xrightarrow[S \times Q]{\text{extr } m_1} J$ is of the same I/O type as m_1 but its state is of type $S \times Q$, not S.

In turn, m_2 is extended in the same way, adding the state of m_1 to the left,

$$\begin{array}{ccccc} \mathbb{F}(S \times (Q \times K)) & \xleftarrow{\tau_l} & S \times \mathbb{F}(Q \times K) & \xleftarrow{id \times m_2} & S \times (Q \times J) \\ {\scriptstyle \mathbb{F}\,\mathsf{a}}\uparrow & & & & \uparrow{\scriptstyle \mathsf{a}} \\ \mathbb{F}((S \times Q) \times K) & & \xleftarrow[\text{extl } m_2]{} & & (S \times Q) \times J \end{array}$$

where $\mathsf{a} : (A \times B) \times C \to A \times (B \times C)$ is the well-known isomorphism. Finally, τ_l is the left strength of $\mathbb{F}$, $\tau_l : (B \times \mathbb{F}\,A) \to \mathbb{F}(B \times A)$. Thus we can define

$$\mathsf{extl}\ m = \mathbb{F}\,\mathsf{a}^\circ \cdot \tau_l \cdot (id \times m) \cdot \mathsf{a} \tag{6}$$

where a° denotes the converse of isomorphism a, and

$$\mathsf{extr}\ m = \mathbb{F}\,\mathsf{xr} \cdot \tau_r \cdot (m \times id) \cdot \mathsf{xr} \tag{7}$$

since $\mathsf{xr}^\circ = \mathsf{xr}$. Note how $\mathbb{F}\,\mathsf{a}^\circ$ brings together the compound state and type K, on the output. In spite of the efforts of xr to reconcile the input of extension $\mathsf{extr}\ m_1$ with the output of $\mathsf{extr}\ m_2$, they do not match, as the latter is $\mathbb{F}$-*more complex* than the former:

$$\begin{array}{ccccc} & & \mathbb{F}((S \times Q) \times J) & \xleftarrow{\text{extr } m_1} & (S \times Q) \times I \\ & & \vdots & & \\ \mathbb{F}((S \times Q) \times K) & \xleftarrow{\text{extr } m_2} & (S \times Q) \times J & & \\ \mathbb{F}(\mathbb{F}\,C) & \xleftarrow{\mathbb{F}\,f} & \mathbb{F}\,B & \xleftarrow{g} & A \\ {\scriptstyle \mu}\downarrow & & & & \\ \mathbb{F}\,C & \xleftarrow{f} & B & & \\ & {\scriptstyle f \bullet g}\nwarrow & & & \end{array} \tag{8}$$

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 (8). Thus we obtain a generic definition of monadic Mealy machine composition which was left implicit in Sect. 4.

Definition 1. Given two $\mathbb{F}$-monadic Mealy machines (MMM) $I \xrightarrow[S]{m_1} J$ and $J \xrightarrow[Q]{m_2} K$ with states S and Q, respectively, their *composition* is the $\mathbb{F}$-monadic Mealy machine $I \xrightarrow[S \times Q]{m_1;m_2} K$ defined by:

[10] Wherever the state S of a machine $A \xrightarrow[S]{m} B$ is implicit from the context, simplified notation $A \xrightarrow{m} B$ will be used instead.

[11] See Definition 14 in Appendix A.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

$$m_1 ; m_2 = (\mathsf{extl}\, m_2) \bullet (\mathsf{extr}\, m_1) \tag{9}$$

which unfolds to

$$m_1 ; m_2 = ((\mathbb{F}\, \mathsf{a}^\circ) \cdot \tau_l \cdot (id \times m_2) \cdot \mathsf{xl}) \bullet (\tau_r \cdot (m_1 \times id) \cdot \mathsf{xr}) \qquad \square \tag{10}$$

Note the economy of this definition compared to the one relying directly on the multiplication μ and unit η of $\mathbb{F}$. An advantage of relying on Kleisli composition (8) is its rich algebra, forming a monoid with η

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h \tag{11}$$

$$f \bullet \eta = f = \eta \bullet f \tag{12}$$

and trading nicely with normal composition, cf. for instance

$$(\mathbb{F}\, f) \cdot (h \bullet k) = (\mathbb{F}\, f \cdot h) \bullet k \tag{13}$$

$$(f \cdot g) \bullet h = f \bullet (\mathbb{F}\, g \cdot h) \tag{14}$$

Further to composition and sum, our component algebra includes the operator which lifts any function to a trivial MMM by ignoring its internal state:

$$\ulcorner \cdot \urcorner : (I \to O) \to (S \times I \to \mathbb{F}\, (S \times O))$$

$$\ulcorner f \urcorner = \eta \cdot (id \times f) \tag{15}$$

Functional lifting is convenient for expressing interface-level operations, that is, "state-less" transformations of the input and/or output types of a machine:

$$m_{\{f \to g\}} = \ulcorner g \urcorner \bullet m \bullet \ulcorner f \urcorner \tag{16}$$

for $f : J \to I$, $g : O \to P$ and $m : I \to O$, yielding $m_{\{f \to g\}} : J \to P$. Unfolding (16) one obtains

$$m_{\{f \to g\}} = \mathbb{F}\, (id \times g) \cdot m \cdot (id \times f) \tag{17}$$

using the laws of Kleisli composition. This combinator is termed *wiring* in [2] where it is used at length. For our purposes, wiring will be particularly useful for defining machine sum as a universal construction.

Definition 2. Given two $\mathbb{F}$-monadic Mealy machines $I \xrightarrow[S]{p} O$ and $J \xrightarrow[S]{q} P$ (with the same state space) their sum is the machine $I + J \xrightarrow[S]{p \oplus q} O + P$ defined by the following universal property:

$$k = p \oplus q \equiv \begin{cases} k_{\{i_1 \to id\}} = p_{\{id \to i_1\}} \\ k_{\{i_2 \to id\}} = q_{\{id \to i_2\}} \end{cases} \qquad \square \tag{18}$$

Property (18) is convenient for decomposing component expressions where $\oplus$ is the outermost combinator. Before showing this, we prove the consistency of this definition with respect to the closed formula given earlier for sums:

$$\begin{aligned}
& k = p \oplus q \\
\equiv \quad & \{ \text{ closed formula for } \oplus \ \} \\
& k = \mathbb{F}\, \mathsf{dr}^\circ \cdot \mathsf{cozip} \cdot (p + q) \cdot \mathsf{dr} \\
\equiv \quad & \{ \ \mathsf{dr} \text{ is an isomorphism ; property (B.5)} \ \} \\
& k \cdot \mathsf{dr}^\circ = [\mathbb{F}\, (id \times i_1), \mathbb{F}\, (id \times i_2)] \cdot (p + q) \\
\equiv \quad & \{ \text{ definition of } \mathsf{dr}^\circ \text{ (B.1) ; coproduct fusion and absorption [6] } \} \\
& [k \cdot (id \times i_1), k \cdot (id \times i_2)] = [\mathbb{F}\, (id \times i_1) \cdot p, \mathbb{F}\, (id \times i_2) \cdot q] \\
\equiv \quad & \{ \text{ universal property } \} \\
& k \cdot (id \times i_1) = \mathbb{F}\, (id \times i_1) \cdot p \wedge k \cdot (id \times i_2) = \mathbb{F}\, (id \times i_2) \cdot q \\
\equiv \quad & \{ \text{ (17) twice } \} \\
& k_{\{i_1 \to id\}} = p_{\{id \to i_1\}} \wedge k_{\{i_2 \to id\}} = p_{\{id \to i_2\}} \qquad \square
\end{aligned}$$

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

Lemma 3. *The following properties of the machine sum operator hold:*

$$\mathsf{extr}\ (p \oplus q) = \mathsf{extr}\ p \oplus \mathsf{extr}\ q \tag{19}$$

$$\mathsf{extl}\ (p \oplus q) = \mathsf{extl}\ p \oplus \mathsf{extl}\ q \tag{20}$$

$$(p \oplus q)_{\{i_1 \to id\}} = p_{\{id \to i_1\}} \tag{21}$$

$$(p \oplus q)_{\{i_2 \to id\}} = q_{\{id \to i_2\}} \tag{22}$$

Proof. Equalities (19) and (20) unfold into

$$(\mathsf{extr}\ (p \oplus q))_{\{i_1 \to id\}} = (\mathsf{extr}\ p)_{\{id \to i_1\}} \tag{23}$$

$$(\mathsf{extr}\ (p \oplus q))_{\{i_2 \to id\}} = (\mathsf{extr}\ q)_{\{id \to i_2\}} \tag{24}$$

and

$$(\mathsf{extl}\ (p \oplus q))_{\{i_1 \to id\}} = (\mathsf{extl}\ p)_{\{id \to i_1\}} \tag{25}$$

$$(\mathsf{extl}\ (p \oplus q))_{\{i_2 \to id\}} = (\mathsf{extl}\ q)_{\{id \to i_2\}} \tag{26}$$

respectively, which are proved in Appendix B. Equalities (21) and (22) follow straightforwardly from (18) by cancellation, i.e. via substitution $k := p \oplus q$. □

Universal property (18) also provides a strategy for calculating the *exchange law* between machine sum and machine composition which follows.

Theorem 4 *(Exchange law). Let* $I \xrightarrow[S]{m_1} O$, $J \xrightarrow[S]{m_2} P$ *and* $O \xrightarrow[Q]{n_1} U$, $P \xrightarrow[Q]{n_2} V$ *be pairs of* $\mathbb{F}$*-monadic Mealy machines (each pair sharing the same state space). Then the following* exchange law *holds, showing how sequential composition commutes with sums*[12]*:*

$$(m_1 \oplus m_2)\ ;\ (n_1 \oplus n_2) = (m_1\ ;\ n_1) \oplus (m_2\ ;\ n_2) \tag{27}$$

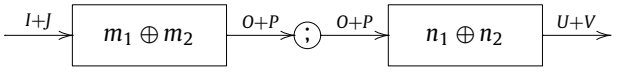
Proof. To prove (27) we observe that, by (18), the equality unfolds into

$$\begin{cases} (m_1 \oplus m_2\ ;\ n_1 \oplus n_2)_{\{i_1 \to id\}} = (m_1\ ;\ n_1)_{\{id \to i_1\}} \\ (m_1 \oplus m_2\ ;\ n_1 \oplus n_2)_{\{i_2 \to id\}} = (m_2\ ;\ n_2)_{\{id \to i_2\}} \end{cases}$$

Below we prove the first conjunct, the proof of the other being similar.

$$\begin{aligned}
& (m_1 \oplus m_2\ ;\ n_1 \oplus n_2)_{\{i_1 \to id\}} \\
= & \quad \{\ \text{definitions of wiring (16) and (;) (9)}\ \} \\
& (\mathsf{extl}\ (n_1 \oplus n_2) \bullet \mathsf{extr}\ (m_1 \oplus m_2)) \bullet \ulcorner i_1 \urcorner \\
= & \quad \{\ \text{definition of } \ulcorner \cdot \urcorner \ (15)\ ;\ \bullet \text{ associativity}\ ;\ (16)\ \} \\
& \mathsf{extl}\ (n_1 \oplus n_2) \bullet (\mathsf{extr}\ (m_1 \oplus m_2)_{\{i_1 \to id\}}) \\
= & \quad \{\ \text{property (23) etc.}\ \} \\
& \mathsf{extl}\ (n_1 \oplus n_2) \bullet \mathbb{F}\ (id \times i_1) \cdot \mathsf{extr}\ m_1 \\
= & \quad \{\ \bullet / \cdot \text{ assoc; property (25)}\ \} \\
& \mathbb{F}\ (id \times i_1) \cdot (\mathsf{extl}\ n_1 \bullet \mathsf{extr}\ m_1) \\
= & \quad \{\ \text{definitions of (;) (9) and wiring (16)}\ \} \\
& (m_1\ ;\ n_1)_{\{id \to i_1\}} \qquad \square
\end{aligned}$$

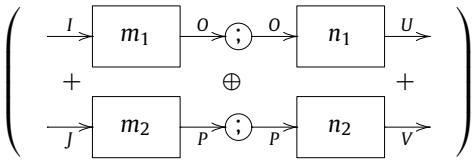
In pictures, law (27) shows that the composition of machines



(28)

[12] For similar exchange laws in process calculi see e.g. [18].

is the same machine as



(29)

Thinking of (28) as the pipeline of two components $p = m_1 \oplus m_2$ and $q = n_1 \oplus n_2$ with two-methods each, (29) gives us a composite (also two-method) machine whose methods are obtained by synchronizing the methods of p and q.

In general, component communication is not so immediate, for it usually requires extra *wirings* (16) at interface-level to route the data across sub-components, as the examples in technical report [32] amply show. This report also describes how the above algebraic kernel for state-based system composition can be used to assign a MMM-semantics to a proof-of-concept object-oriented language and prove semantic properties of the language.[13]

6. Composing non-deterministic components

Having defined an algebra of combinators of software components understood as $\mathbb{F}$-branching Mealy machines, let us come back to the central point of this paper: is such an algebra applicable to *faulty* (e.g. probabilistic) Mealy machines?

Recall from the motivation how we have simulated faulty composition of (probabilistic) $\mathbb{M}$-Mealy machines in Haskell. In general, this means handling machines of type $Q \times I \to \mathbb{D}\,(\mathbb{F}\,(Q \times J))$ for some space state Q, where before we had $Q \times I \to \mathbb{F}\,(Q \times J)$. Thus another monad — the *distribution* monad $\mathbb{D}$ — is combined with $\mathbb{F}$. Generalizing even further, we want to consider machines of type

$$Q \times I \to \mathbb{T}\,(\mathbb{F}\,(Q \times J)) \tag{30}$$

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, adopting the terminology of [17]. A typical instance is $\mathbb{T} = \mathbb{P}$ (powerset) and $\mathbb{F} = \mathbb{M} = (1+)$ ('maybe'), that is, machines

$$m : Q \times I \to \mathbb{P}\,(1 + Q \times J) \tag{31}$$

which are reactive, possibly terminating non-deterministic finite state automata.

Note, however, that m (31) could alternatively be specified as a *binary relation* R of type $Q \times I \to 1 + Q \times J$ of which m is the *power transpose* [6], following the equivalence

$$R = \lceil m \rceil \;\equiv\; \langle \forall b, a :: b\ R\ a \equiv b \in m\ a \rangle \tag{32}$$

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*:

$$b\ (R \cdot S)\ a \;\equiv\; \langle \exists c :: b\ R\ c \wedge c\ S\ a \rangle \tag{33}$$

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 (31), the advantage of "thinking relationally" is that machine m can be "replaced" by the relation $\lceil m \rceil : Q \times I \to 1 + Q \times J$ from whose (relational) type the powerset has vanished.[14] 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 (9), (10) that machine composition relies on Kleisli composition (8) — 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:

$$f \bullet g = [i_1, id] \cdot (id + f) \cdot g = [i_1, f] \cdot g \tag{34}$$

[13] This is supported by the MMM Haskell library available from https://github.com/VictorCMiraldo/mmm.

[14] "Vanished" is perhaps too strong a word but it serves our purpose: the powerset becomes *implicit* rather than *explicit* (as before) and so it does not get in the way of the calculations.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

How about relations? Consider evaluating expression $[i_1, f] \cdot g$ for f replaced by some relation $1 + B \xleftarrow{R} C$,[15] g by some other relation $1 + C \xleftarrow{S} A$ and function composition replaced by relation composition:

$$R \bullet S = [i_1, R] \cdot S \tag{35}$$

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

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

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 and 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 monad. This is the category of *binary relations*. We can thus think of $\mathbb{M}$-monadic Mealy machines as *binary relations* which compose (as machines) according to definition[16]

$$S ; R \ = \ [i_1, (id + \mathsf{a}^\circ) \cdot \tau_l \cdot (id \times R) \cdot \mathsf{x}] \cdot \tau_r \cdot (S \times id) \cdot \mathsf{x} \tag{36}$$

where all dots mean relation composition (33).[17]

Note, however, that 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 relationally, extensive to the properties of the *strength* operators also present in (36). Prior to doing this, let us replay the scenario above for the *distribution* in place of the *powerset* monad.

7. Composing probabilistic components

The probabilistic treatment of imprecision (faults) in $\mathbb{M}$-Mealy machines calls for $\mathbb{T} = \mathbb{D}$ and $\mathbb{F} = \mathbb{M}$ in type (30), leading to

$$m : Q \times I \to \mathbb{D}\ (1 + Q \times J) \tag{37}$$

whereupon non-deterministic branching becomes weighted with probabilities indicating the likelihood of state transitions, recall (2). So, m is now a distribution-valued function.

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*:

$$b\ (M \cdot N)\ a \ \equiv \ \langle \sum c :: (b\ M\ c) \times (c\ N\ a) \rangle \tag{38}$$

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.[18]

Summing up: in the same way the "Kleisli lifting" of Sect. 6 makes the *powerset* monad implicit, leading into *relation algebra*, the same lifting now hides the *distribution* monad and leads to the *linear algebra* of CS matrices [36], under the universal correspondence

$$M = \lceil f \rceil \ \equiv \ \langle \forall b, a :: b\ M\ a = (f\ a)\ b \rangle \tag{39}$$

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 $a \in A$, $f\ a$ is some function $\mu : B \to [0, 1]$ to the unit interval such that $\sum_{b \in B} \mu\ b = 1$.

Correspondence (39) establishes the isomorphism

$$A \to \mathbb{D}\ B \ \cong \ A \to_{\mathrm{CS}} B \tag{40}$$

where on the left-hand side we have $\mathbb{D}$-valued functions and on the right-hand side $A \to_{\mathrm{CS}} B$ denotes the set of all CS matrices with columns indexed by A, rows indexed by B and cells taking values from the interval $[0, 1]$.[19] This *matrices as*

[15] 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* [22].

[16] Note that, following precedents in the literature of the algebra of programming [6] and components as coalgebras [3], we write "." for (backward) composition of relations and ";" for (forward) composition of components.

[17] As usual, every function symbol f in (36) should be regarded as the homonym relation f such that $b\ f\ a$ holds iff $b = f\ a$.

[18] Ref. [37] argues in this direction by adapting well-known rules of relation algebra to linear algebra.

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

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

arrows approach [28] regards them as *morphisms* of suitable categories (of typed matrices). In the current paper we focus on matrices on the interval $[0,1]$ subject to the *column stochasticity* constraint assumed in (39).

With no further detours let us adapt definition (35) 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 (38):

$$M \bullet N = [i_1 | M] \cdot N \tag{41}$$

The reader may wonder about how does injection i_1 (a function) fit into a linear algebra expression (41). The explanation is the same as for functions in relational expressions: every function $f : A \to B$ is uniquely represented by a Dirac distribution, and hence by the homonym matrix f defined by $b\, f\, a = 1$ if $b = f\, a$ and 0 otherwise.[20] Combinator $[M|N]$ occurring in (41) means the juxtaposition of matrices M and N side-by-side, 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. (This 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 (41)

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

where $*$ is the same abbreviation used before and term $y = *$ evaluates to 1 if the equality holds and to 0 otherwise.[21]

				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

The picture above 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.

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

$$N\ ;\ M = [i_1 | (id \oplus \mathsf{a}^\circ) \cdot \tau_l \cdot (id \otimes M) \cdot \mathsf{xl}] \cdot \tau_r \cdot (N \otimes id) \cdot \mathsf{xr} \tag{42}$$

where relational product becomes matrix Kronecker product

$$(y, x)(M \otimes N)(b, a) = (yMb) \times (xNa) \tag{43}$$

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 relation/linear algebra

The evolution from relation 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: $\lceil f\ {}_p\diamond\ g \rceil = p \otimes \lceil f \rceil + (1-p) \otimes \lceil g \rceil$.

[20] See Sect. 8 for a technically more detailed explanation. We should also say that matrices in typed linear algebra [36] are not restricted to the usual view of matrices as "rectangles filled with numbers", but are rather regarded as typed and possibly infinite dimensional, provided all columns have countable support. In this way, matrix composition is well-defined, as summations range over the non-zero elements only; also note that $A + B$ (resp. $A \times B$) in the type of a matrix means the disjoint union (resp. Cartesian product) of A and B. More details in e.g. [34,36].

[21] See [34,37] 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.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

A generic strategy can be identified: having a notion of composition (10) 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 $Q \times I \to \mathbb{T}\,(\mathbb{F}\,(Q \times J))$ (30) by porting it "as is" to the Kleisli category of the extra monad $\mathbb{T}$ which captures the *branching* structure [17].

For the above 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 $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{T}X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{T}^2X$ and $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2X$ be two monads in a category C,[22] and let $\mathsf{C}_{\mathbb{T}}$ denote the Kleisli category induced by $\mathbb{T}$. Denote by $B \xleftarrow{f^\flat} A$ the morphism in $\mathsf{C}_{\mathbb{T}}$ corresponding to $\mathbb{T}B \xleftarrow{f} A$ in C and define:

$$f^\flat \cdot g^\flat = (f \bullet g)^\flat = (\mu_{\mathbb{T}} \cdot \mathbb{T}f \cdot g)^\flat \tag{44}$$

For *any* morphism $B \xleftarrow{f} A$ in C define its lifting to $\mathsf{C}_{\mathbb{T}}$ by $\overline{f} = (\eta_{\mathbb{T}} \cdot f)^\flat$. The following equalities

$$\overline{f} \cdot g^\flat = (\mathbb{T}f \cdot g)^\flat \tag{45}$$

$$f^\flat \cdot \overline{g} = (f \cdot g)^\flat \tag{46}$$

prove useful, leading to

$$\overline{f} \cdot \overline{g} = \overline{f \cdot g} \tag{47}$$

$$\overline{\mathbb{F}}\,\overline{f} = \overline{\mathbb{F}f} \tag{48}$$

where the definition of functor $\overline{\mathbb{F}}$ (to be given shortly) requires the notion of a *distributive law*.

Definition 5. (See Beck [5].) Let $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{T}X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{T}^2X$ and $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2X$ be two monads. A **distributive law** of $\mathbb{T}$ over $\mathbb{F}$ is a natural transformation $\lambda : \mathbb{F}\,\mathbb{T} \to \mathbb{T}\,\mathbb{F}$ such that

$$\lambda \cdot \mathbb{F}\,\eta_{\mathbb{T}} = \eta_{\mathbb{T}} \tag{49}$$

$$\lambda \cdot \mathbb{F}\,\mu_{\mathbb{T}} = \mu_{\mathbb{T}} \cdot \mathbb{T}\lambda \cdot \lambda \tag{50}$$

and

$$\lambda \cdot \eta_{\mathbb{F}} = \mathbb{T}\eta_{\mathbb{F}} \tag{51}$$

$$\mathbb{T}\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda = \lambda \cdot \mu_{\mathbb{F}} \tag{52}$$

hold. □

Theorem 6. *(See Mulry [33].) 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 $\mathsf{C}_{\mathbb{T}}$ by*

$$\overline{\mathbb{F}}(f^\flat) \;=\; (\lambda \cdot \mathbb{F}f)^\flat \quad \textit{cf. diagram} \quad \mathbb{T}\mathbb{F}B \xleftarrow{\lambda} \mathbb{F}\mathbb{T}B \xleftarrow{\mathbb{F}f} \mathbb{F}A \tag{53}$$

for $\mathbb{T}\,B \xleftarrow{f} A$. For $\overline{\mathbb{F}}$ to be a functor in $\mathsf{C}_{\mathbb{T}}$ the two conditions (49) and (50) are enough.

Proof. See [33]. □

Theorem 7 *(Monad lifting). Let monads $\mathbb{F}$, $\mathbb{T}$ and distributive law λ be such as in Definition 5 and Theorem 6. Then*

$$X \xrightarrow{\overline{\eta_{\mathbb{F}}} = (\eta_{\mathbb{T}} \cdot \eta_{\mathbb{F}})^\flat} \overline{\mathbb{F}}X \xleftarrow{\overline{\mu_{\mathbb{F}}} = (\eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}})^\flat} \overline{\mathbb{F}}^2X \tag{54}$$

*is a **monad** in $\mathsf{C}_{\mathbb{T}}$ thanks to conditions (51) and (52).*

Proof. To the best of the authors knowledge, this result has been implicit in the literature which emerged from [5] but hardly stated explicitly. The closest they have found is a theorem in [45] but even there it is given as a summary of two chapters of a PhD thesis. Therefore, a (compact) proof is given in Appendix B. □

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 [17] 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 (49), (50) in both cases. It is also easy to show that

[22] In this paper we adopt as base category C the category of sets and total functions on sets.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

$\mathbb{M}$ satisfies (51), (52) for any $\mathbb{T}$.[23] So nondeterministic (resp. probabilistic) composition of $\mathbb{M}$-monadic Mealy machines regarded as binary relations (resp. matrices) given by monadic definitions (36) (resp. (42)) is sound.

The scope of Theorem 7 is wider than it may seem at first sight. Suppose our software components are still possibly failing machines ($\mathbb{F} = \mathbb{M}$) which, further to their internal state, share a global state S, i.e. $\mathbb{T}$ is the *state monad* $\mathbb{S}\,X = (X \times S)^S$. Then $\mathbb{M}$ lifts to the Kleisli category of $\mathbb{S}$ which is, not surprisingly, that of deterministic Mealy machines with state S.

On the other hand, $\mathbb{M}$ can be *generalized* to any $\mathbb{T}$-liftable monad $\mathbb{F}$ satisfying (51), (52). Theorem 8 below shows that this also includes the *free monad* $\mathbb{F}_{\mathbb{G}}\,X \cong X + \mathbb{G}\,(\mathbb{F}_{\mathbb{G}}\,X)$ over a functor $\mathbb{G}$, of which $\mathbb{M}$ is an instance, for the constant functor $\mathbb{G} = 1$. Thus a number of well-known monads are liftable, e.g. binary trees (for $\mathbb{G}\,X = X^2$), lists (for $\mathbb{G}\,X = A \times X$, for some A) and so on.

Theorem 8 *(Free-monad lifting). Let $\mathbb{G}$ be a functor such that the initial algebra $\mathbb{F}_{\mathbb{G}}\,X \xleftarrow{\mathsf{in}} X + \mathbb{G}\,(\mathbb{F}_{\mathbb{G}}\,X)$ exists, that is, $\mathbb{F}_{\mathbb{G}}$ is the* free monad *on $\mathbb{G}$. We denote by $(\!|h|\!)$ the unique morphism[24] from $\mathbb{F}_{\mathbb{G}}\,X$ to a given algebra $Y \xleftarrow{h} X + \mathbb{G}\,Y$ of carrier Y, cf. diagram:*

$$\begin{array}{ccc} \mathbb{F}_{\mathbb{G}}\,X & \xleftarrow{\mathsf{in}} & X + \mathbb{G}\,(\mathbb{F}_{\mathbb{G}}\,X) \\ {\scriptstyle (\!|h|\!)}\downarrow & & \downarrow{\scriptstyle id + \mathbb{G}\,(\!|h|\!)} \\ Y & \xleftarrow{h} & X + \mathbb{G}\,Y \end{array}$$

Let $\mathbb{G}$ distribute over a given monad $\mathbb{T}$ through law $\mathbb{T}\,\mathbb{G} \xleftarrow{\lambda_{\mathbb{G}}} \mathbb{G}\,\mathbb{T}$. Then the $\mathbb{F}_{\mathbb{G}}$ monad defined by

$$\eta = \mathsf{in} \cdot i_1 = \mathsf{in}_1 \tag{55}$$

$$\mu = (\!|[id, \mathsf{in} \cdot i_2]|\!) = (\!|[id, \mathsf{in}_2]|\!) \tag{56}$$

is liftable to the Kleisli category of $\mathbb{T}$, that is, that (51), (52) hold, for distributive law

$$\lambda = (\!|[\mathbb{T}\,\mathsf{in}_1, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) \tag{57}$$

of type $\mathbb{T}\,\mathbb{F}_{\mathbb{G}} \xleftarrow{\lambda} \mathbb{F}_{\mathbb{G}}\,\mathbb{T}$.

Proof. See Appendix B. □

9. Strong monads in relation/linear algebra

Recall from the motivation that we went as far as simulating probabilistic composition of $\mathbb{M}$-machines in Haskell using the operator $m_1 \mathbin{;_D} m_2$, the probabilistic evolution of $m_1 \,;\, m_2$ (10). Although we have not seen its actual definition, we can say that fact $\lceil m_1 \mathbin{;_D} m_2 \rceil = \lceil m_1 \rceil \,;\, \lceil m_2 \rceil$ holds, where composition $\lceil m_1 \rceil \,;\, \lceil m_2 \rceil$ is given by (42) in the context of (39).[25]

We are not yet done, however: definition (42) 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 $\mathbb{F}$-information structures. Their basic properties (A.1), (A.2) are preserved by their liftings $\overline{\tau_r}$ and $\overline{\tau_l}$.[26] So, what may fail is their *naturality* (A.3) once lifted, that is,

$$\overline{\tau_l} \cdot (N \otimes \overline{\mathbb{F}}\,M) = \overline{\mathbb{F}}\,(N \otimes M) \cdot \overline{\tau_l} \tag{58}$$

$$\overline{\tau_r} \cdot (\overline{\mathbb{F}}\,M \otimes N) = \overline{\mathbb{F}}\,(M \otimes N) \cdot \overline{\tau_r} \tag{59}$$

where M and N are arbitrary arrows in the Kleisli category of monad $\mathbb{T}$, denoted $\mathsf{C}_{\mathbb{T}}$ in the previous section, and $\otimes$ denotes the tensor in $\mathsf{C}_{\mathbb{T}}$ which results from lifting products in C

$$f^\flat \otimes g^\flat = (\delta \cdot (f \times g))^\flat \tag{60}$$

[23] Concerning (51): $[\eta_{\mathbb{T}} \cdot i_1, \mathbb{T}\,i_2] \cdot i_2 = \mathbb{T}\,i_2$; concerning (52): $\mathbb{T}\,[i_1, id] \cdot [\eta_{\mathbb{T}} \cdot i_1, \mathbb{T}\,i_2 \cdot \lambda] = [\eta_{\mathbb{T}} \cdot [i_1, id] \cdot i_1, \lambda] = [\lambda \cdot i_1, \lambda] = \lambda \cdot [i_1, id]$.

[24] Usually termed *fold* or *catamorphism* [6].

[25] The actual implementation of $(;_D)$ in the Haskell simulator follows *verbatim* pointfree formula (42) carefully using the encodings of Sect. 8.

[26] The general rule is that $\overline{f} = (\eta_{\mathbb{T}} \cdot f)^\flat$ embeds C in $\mathsf{C}_{\mathbb{T}}$. Thus the lifting of e.g. an equality $f \cdot g = h$ in C, that is $\overline{f} \cdot \overline{g} = \overline{h}$ in $\mathsf{C}_{\mathbb{T}}$ reduces to the original $f \cdot g = h$, cf. (48), (47). Within the image of the embedding, everything in $\mathsf{C}_{\mathbb{T}}$ "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, widely adopted abuse of notation [6].

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

where δ denotes the "double strength" of a commutative monad

$$\delta = \tau_r \bullet \tau_l = \tau_l \bullet \tau_r$$

(see Definition 15), a class of monads which includes both $\mathbb{D}$ and $\mathbb{P}$.

Henceforth we shall assume that $\mathbb{T}$ is commutative, heading to the following theorem which gives sufficient conditions for a strong monad $\mathbb{F}$ to remain strong once lifted by a commutative $\mathbb{T}$.[27]

Theorem 9 *(Strong monad lifting). Let $\mathbb{F}$, $\mathbb{T}$, λ be as in Theorem 6. Further assume that $\mathbb{F}$ is strong and that $\mathbb{T}$ is commutative.[28] Then $\overline{\mathbb{F}}$ is strong in $\mathsf{C}_{\mathbb{T}}$ provided the following condition holds in C,*

$$\lambda \cdot (\mathbb{F}\,\delta) \cdot \tau_l = \mathbb{T}\,\tau_l \cdot \delta \cdot (id \times \lambda) \tag{61}$$

or the equivalent

$$\lambda \cdot (\mathbb{F}\,\delta) \cdot \tau_r = \mathbb{T}\,\tau_r \cdot \delta \cdot (\lambda \times id) \tag{62}$$

cf. diagram

$$\begin{array}{ccc}
\mathbb{F}\,(\mathbb{T}\,X \times \mathbb{T}\,Y) & \xleftarrow{\tau_l} & \mathbb{T}\,X \times \mathbb{F}\,\mathbb{T}\,Y \\
\downarrow{\scriptstyle \mathbb{F}\,\delta} & & \downarrow{\scriptstyle id \times \lambda} \\
\mathbb{F}\,\mathbb{T}\,(X \times Y) & & \mathbb{T}\,X \times \mathbb{T}\,\mathbb{F}\,Y \\
\downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle \delta} \\
\mathbb{T}\,\mathbb{F}\,(X \times Y) & \xleftarrow{\mathbb{T}\,\tau_l} & \mathbb{T}\,(X \times \mathbb{F}\,Y)
\end{array}$$

for (61), that for (62) being similar.

Proof. See Appendix B. □

Let us fix $\mathbb{F} = \mathbb{M} = (1+)$ and give two examples of commutative $\mathbb{T}$, one which satisfies e.g. (61) and another which does not. For $\mathbb{M}$ we have $\tau_l = (! + id) \cdot \mathsf{dr}$, of type $B \times (1 + A) \to 1 + B \times A$, where $\mathsf{dr} : A \times (C + B) \to A \times C + A \times B$ is the obvious isomorphism and $! : A \to 1$ is the unique function of its type, recall Sect. 4.

Take $\mathbb{T} = \mathbb{P}$, the powerset monad, which is commutative and such that $\delta(s, r)$ is the Cartesian product of s and r. Let $(x, y) = (\{\}, *)$, where $*$ denotes the sole inhabitant of 1. Then running the right-hand side of (61), $(\mathbb{P}\,\tau_l)\,(\delta\,(x, \lambda\,y))$ will yield $\{\}$ while the left-hand side $\lambda\,(\mathbb{M}\,\delta\,(\tau_l\,(x, y)))$ will yield $\{*\}$. (Below we will come back to this anomaly, rephrased in relation algebra.)

Now take $\mathbb{T} = \mathbb{D}$, the distribution monad. We prove below that (61) holds for this lifting of $\mathbb{M}$:

$$\begin{array}{ll}
& \lambda \cdot (\mathbb{M}\,\delta) \cdot \tau_l = \mathbb{D}\,\tau_l \cdot \delta \cdot (id \times \lambda) \\
\equiv & \quad \{\ \text{distributive law } \lambda = [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2];\ \text{functor } \mathbb{M}f = id + f\ \} \\
& [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2 \cdot \delta] \cdot \tau_l = \mathbb{D}\,\tau_l \cdot \delta \cdot (id \times [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2]) \\
\equiv & \quad \{\ \text{strength } \tau_l = (! + id) \cdot \mathsf{dr}\ \} \\
& [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2 \cdot \delta] \cdot (! + id) \cdot \mathsf{dr} = \mathbb{D}\,\tau_l \cdot \delta \cdot (id \times [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2]) \\
\equiv & \quad \{\ \text{coproducts};\ \mathsf{dr}^{\circ} = [id \times i_1, id \times i_2]\ \} \\
& [\eta_{\mathbb{D}} \cdot i_1 \cdot !, \mathbb{D}\,i_2 \cdot \delta] = \mathbb{D}\,\tau_l \cdot \delta \cdot (id \times [\eta_{\mathbb{D}} \cdot i_1, \mathbb{D}\,i_2]) \cdot [id \times i_1, id \times i_2] \\
\equiv & \quad \{\ \text{coproducts};\ \delta \cdot (id \times \eta_{\mathbb{D}}) = \tau_r\ \} \\
& \begin{cases} \eta_{\mathbb{D}} \cdot i_1 \cdot ! = \mathbb{D}\,\tau_l \cdot \tau_r \cdot (id \times i_1) \\ \mathbb{D}\,i_2 \cdot \delta = \mathbb{D}\,\tau_l \cdot \delta \cdot (id \times \mathbb{D}\,i_2) \end{cases} \\
\equiv & \quad \{\ \tau_r \text{ is natural (A.3)};\ \mathbb{D}\,id = id\ \}
\end{array}$$

[27] Note that requiring $\mathbb{T}$ to be commutative rules out some potentially useful examples, such as the state monad, the IO monad, the list monad and others. Nevertheless, monads $\mathbb{M}$, $\mathbb{D}$ and $\mathbb{P}$ are commutative.

[28] Therefore also strong, cf. Definition 15 in Appendix A.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

$$
\begin{array}{ll}
& \left\{ \begin{array}{l} \eta_{\mathbb{D}} \cdot i_1 \cdot ! = \mathbb{D}\ \tau_l \cdot \mathbb{D}\ (id \times i_1) \cdot \tau_r \\ \mathbb{D}\ i_2 \cdot \delta = \mathbb{D}\ \tau_l \cdot \mathbb{D}\ (id \times i_2) \cdot \delta \end{array} \right. \\
\equiv & \quad \{\ (63), (64)\ \text{below}\ ;\ \text{functor}\ \mathbb{D}\ \} \\
& \left\{ \begin{array}{l} \eta_{\mathbb{D}} \cdot i_1 \cdot ! = \mathbb{D}\ i_1 \cdot \mathbb{D}\ ! \cdot \tau_r \\ \mathbb{D}\ i_2 \cdot \delta = \mathbb{D}\ i_2 \cdot \delta \end{array} \right. \\
\equiv & \quad \{\ \mathbb{D}\ ! = \eta_{\mathbb{D}} \cdot !\ \} \\
& \eta_{\mathbb{D}} \cdot i_1 \cdot ! = \mathbb{D}\ i_1 \cdot \eta_{\mathbb{D}} \cdot ! \\
\equiv & \quad \{\ \text{natural}\ \eta_{\mathbb{D}}\ \} \\
& \eta_{\mathbb{D}} \cdot i_1 \cdot ! = \eta_{\mathbb{D}} \cdot i_1 \cdot ! \\
& \square
\end{array}
$$

NB: One of the steps of the proof above relies on facts

$$\tau_l \cdot (id \times i_1) = i_1 \cdot ! \tag{63}$$

$$\tau_l \cdot (id \times i_2) = i_2 \tag{64}$$

which stem directly from definition $\tau_l = (! + id) \cdot \mathsf{dr}$ via (B.1) and coproduct algebra.

If one tries to rework the proof above for $\mathbb{P}$ instead of $\mathbb{D}$ all steps are valid but the second-to-last: $\mathbb{P}\ ! \neq \eta \cdot !$ since $(\mathbb{P}\ !)\ \{\} = \{\} \neq \{*\} = \eta\ (!\ \{\})$, confirming the counter-example given above. What in fact holds is the inclusion $\mathbb{P}\ ! \ \dot{\subseteq}\ \eta \cdot !$ where $f\ \dot{\subseteq}\ g \equiv \langle \forall a :: f\ a \subseteq g\ b \rangle$ is the lifting of the powerset ordering to functions of type $A \to \mathbb{P}\ B$.

In general, wherever $\sqsubseteq$ is a partial order definable on monad $\mathbb{T}$ in category C, it makes sense to order the morphisms in $\mathsf{C}_{\mathbb{T}}$ as follows:

$$f^{\flat} \sqsubseteq g^{\flat} \equiv f\ \dot{\sqsubseteq}\ g \tag{65}$$

This makes $\mathsf{C}_{\mathbb{T}}$ an *order-enriched* category [17], e.g. the category of relations (for $\mathbb{T} = \mathbb{P}$), of partial maps (for $\mathbb{T} = \mathbb{M}$), of subdistributions (for $\mathbb{T}$ the subdistribution monad[29]) etc.

It turns out that reasoning in $\mathsf{C}_{\mathbb{T}}$ ordered by $\sqsubseteq$ (65) is actually simpler than in C (which is the category of sets in our case) relying on order $\dot{\sqsubseteq}$, as argued elsewhere [6,36,42]. We follow this strategy below in checking the validity of (58), (59) directly in $\mathsf{C}_{\mathbb{T}}$.

First, we show how the argument is rephrased and how it gets simpler should one work directly in the category of stochastic matrices which represents the Kleisli category of $\mathbb{D}$ where, as we have seen, the tensor product is known as *Kronecker* product (43).

We want to show that (58) holds, where M and N are (column) stochastic matrices. The naturality of τ_l (NB: 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 relation/matrix biproducts [28]. Concerning $! \oplus id : B + A \to 1 + A$:

$$
\begin{array}{ll}
& (! \oplus id) \cdot (M \oplus N) \\
= & \quad \{\ \text{bifunctor}\ \oplus\ \} \\
& (! \cdot M) \oplus N \\
= & \quad \{\ ! \cdot M = !\ \text{because}\ M\ \text{is assumed column stochastic [37]}\ \} \\
& ! \oplus N \\
= & \quad \{\ \text{bifunctor}\ \oplus\ \} \\
& (id \oplus N) \cdot (! \oplus id) \\
& \square
\end{array}
$$

(The calculation for τ_r will be similar.) Immediately we get that $\overline{\mathbb{M}}$ is strong in the Kleisli category of $\mathbb{D}$.

By contrast, take $\mathbb{T} = \mathbb{P}$ in which case the corresponding Kleisli category is that of binary relations and the tensor product is given by $(y, x)(M \otimes N)(b, a) \ = \ (yMb) \wedge (xNa)$. (The notation $\times$ we used before for this tensor product is also common in the literature.) In this case, proof step $! \cdot M = !$ for M a binary relation does not hold in general: it fails wherever M is not *entire*, i.e. not totally defined, $! \cdot M \subseteq !$ holding instead, expressed in the enriched structure of this Kleisli category — an *allegory* in the sense of [13].

[29] This monad corresponds to $\mathbb{D}$ relaxing the condition of all probabilities in distributions summing up to 1 to summing at most 1. $\mathbb{D}$ itself is trivially ordered by the identity.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

That (58) is an inclusion rather than an equality for relations could have been anticipated immediately from the *free theorem* [46] of $! + id$:

$$(! + id) \cdot (S + R) \subseteq (id + R) \cdot (! + id)$$

Similarly, it can be anticipated that (58) will also fail for the *sub*-distribution monad [17], that is, for matrices M which are column *sub*-stochastic i.e. such that constraint $! \cdot M = !$ is relaxed to $! \cdot M \leq !$.

Pairing. The fact that a (strong) *transition* monad lifts to the Kleisli category of a (commutative) *branching* monad in coalgebraic modeling of software components does not mean that everything goes smoothly forever in such a Kleisli setting.

Pairing provides a simple and relevant counter-example. In the category of sets, $(f \vartriangle g)\ a = (f\ a, g\ a)$ is a right adjoint, forming a categorial product. However, this is not a categorial product once Kleisli lifted [9,34]. The corresponding matrix operation is the so-called *Khatri–Rao* matrix product [27] defined by $(b, c)\ (M \vartriangle N)\ a = (b\ M\ a) \times (c\ N\ a)$. In relation algebra, $(b, c)\ (R \vartriangle S)\ a = (b\ R\ a) \wedge (c\ S\ a)$ is known as (strict) *fork* [14,42]. Both can be regarded as liftings of the pairing operator, which can be written generically as

$$M \vartriangle N = (M \otimes N) \cdot \overline{\Delta} \tag{66}$$

where $\overline{\Delta}$ is the lifting of the diagonal function $\Delta : A \to A \times A$, $\Delta\ a = (a, a)$, shown below as a matrix, for A the Booleans:

$$\begin{array}{c} \\ (F,F) \\ (F,T) \\ (T,F) \\ (T,T) \end{array} \begin{array}{c} \begin{array}{cc} F & T \end{array} \\ \left[\begin{array}{cc} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{array}\right] \end{array}$$

Although Δ is a natural transformation for functions, $\Delta \cdot f = (f \times f) \cdot \Delta$, this naturality carries further neither to relations nor to stochastic matrices. In the latter case, the natural property of Δ (removing the overbar as before, for simplicity), $(M \otimes M) \cdot \Delta = \Delta \cdot M$ 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.

Again, the weaker $\Delta \cdot R \subseteq (R \times R) \cdot \Delta$ holds for relations but, in this case, even this weaker form fails (in general) for *sub*-stochastic matrices, as the example shows:

$$\begin{array}{ll} & \Delta \cdot \begin{bmatrix} 0.8 & 0 \\ 0.1 & 1 \end{bmatrix} \leq \left(\begin{bmatrix} 0.8 & 0 \\ 0.1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0.8 & 0 \\ 0.1 & 1 \end{bmatrix} \right) \cdot \Delta \\ \equiv & \quad \{\ \text{expand } \Delta \text{ and the Kronecker product, then compose}\ \} \\ & \begin{bmatrix} 0.8 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0.1 & 1 \end{bmatrix} \leq \begin{bmatrix} 0.64 & 0 \\ 0.08 & 0 \\ 0.08 & 0 \\ 0.01 & 1 \end{bmatrix} \\ \equiv & \quad \{\ \text{matrix pointwise} \leq\ \} \\ & \mathit{false} \end{array}$$

□

To the best of the authors' knowledge, loss of naturality across Kleisli *lifting* has not been openly addressed in the literature. It clearly is a topic for future research essential to the success of the *"keep definition, change category"* strategy — see Sect. 12. Fortunately, in the context of the current paper its impact is limited, as shown next.

Impact on component algebra. In the terminology of categorial physics, Δ fails to be a *uniform copying operation* [9]. In general, this failure can be characterized by the fact that the equality $\delta \cdot (f \vartriangle f) = \mathbb{T}\ (id \vartriangle id) \cdot f$ does **not** hold in most commutative monads $\mathbb{T}$ of interest. As mentioned above, this is related to the fact that the lifting of $(\vartriangle)$ no longer participates in the adjunction which defines categorial products.

Back to our context, that of MMM regarded as software components, the question is: what is the impact of τ_l and τ_r losing naturality in the lifting of the combinators defined in Sect. 5? It turns out that it has *no impact* and that the calculus remains the same, as our final theorem shows.

Theorem 10 *(Component algebra lifting). Let $\mathbb{F}$, $\mathbb{T}$, λ be as in Theorem 7 and $\mathbb{T}$ be commutative so as to ensure tensor (60).*

Recall the algebra of $\mathbb{F}$-monadic Mealy machine combinators $p\ ;\ q$, $p \oplus q$ and $p_{\{f \to g\}}$ given in Sect. 5. Then the same algebra holds once the lifting $\overline{\mathbb{F}}$ (of $\mathbb{F}$ by $\mathbb{T}$) replaces $\mathbb{F}$ in the definitions of such combinators; strengths τ_l, τ_r and isomorphisms a, xr *etc. are lifted by $\mathbb{T}$ to $\overline{\tau_l}$, $\overline{\tau_r}$,* $\overline{\mathsf{a}}$, $\overline{\mathsf{xr}}$ *etc.;* cozip *(B.2) is replaced by $[\overline{\mathbb{F}}\ \overline{i_1} | \overline{\mathbb{F}}\ \overline{i_2}]$, where $[f^\flat | g^\flat] = [f, g]^\flat$; product is replaced by tensor (60); and coproduct is replaced by direct sum $\oplus$.*

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

Proof. Our use of strength naturality (A.3) in the proofs of (23) and (25) in Appendix B — and similarly in those of (24) and (26) — is confined to contexts in which f and g are specific functions (e.g. the identity and the coproduct injections) for which such properties trivially lift.[30] □

Pipelining, wiring and sum are kernel MMM combinators which have a wide range of application, see e.g. [32]. The impact of the pairing anomaly discussed above on lifting other combinators of [2,3] not included in such a kernel is a subject for future work.

On $\mathsf{C}_{\mathbb{T}}$ *order-enrichment.* We have seen above that enriching $\mathsf{C}_{\mathbb{T}}$ with a partial order induced by monad $\mathbb{T}$ is convenient for reasoning, wherever this is possible. It turns out that the order-enrichment of $\mathsf{C}_{\mathbb{T}}$ is actually essential to defining suitable coalgebraic *trace semantics* for our software components. Hasuo et al. [17] show that such semantics arises naturally wherever the ordering on $\mathsf{C}_{\mathbb{T}}$ is a CPPO,[31] which is the case for $\mathbb{T} = \mathbb{P}$, $\mathbb{M}$ and $\mathbb{D}$ extended to sub-distributions. The interested reader is referred to [17] for the technical details.

10. Related work

Mealy machines are addressed coalgebraically by Rutten [41] for the special case of $\mathbb{T}$ and $\mathbb{F}$ in our generic formulation

$$S \to (\mathbb{T}(\mathbb{F}(S \times O)))^I \tag{67}$$

being the identity monad, cf. $S \to (S \times O)^I$. This type is isomorphic to $I \to (S \times O)^S$ and so, in this case, Mealy machines $I \to O$ form the Kleisli category of the state monad, parametric on S.

Type (67) also extends that of Mealy machines in the work by Barbosa [3], who considers $\mathbb{F}$ as a (commutative) monad but keeps $\mathbb{T}$ as the identity. This is also the case in Ref. [16], which gives a new foundation for Barbosa's Mealy machines in terms of many-sorted Lawvere (FP) theories, as alternative to the bicategorial framework of [3].

There is a tight relationship between [16] and the current paper, in the sense that both are interested in the calculus of *components as coalgebras* of type $S \to (\mathbb{F}\,(S \times O))^I$ regarded as "interface morphisms" $I \to O$ in suitable frameworks. That chosen in [16] is FP-theory *ArrTh* which encodes Hughes' *arrows* [19], which are in close relationship with Kleisli categories. Our direct use of Mealy machines as arrows of the Kleisli category of the state monad (see above) correspond to this level of the formalization given in [16]. The presence of monads in the overall functor is addressed in both approaches by shifting to the corresponding Kleisli category. Another theory *MArrTh* is given when commutativity holds and a third one, *PLTh* where the arrow-theoretic operator *first* is dropped.

Altogether, the level of genericity in [16] is much higher than ours; the theory is elegant and generic, but the categorial machinery involved is an order of magnitude more complex. In the words of [16], the complexity of calculations would be overwhelming without the help of a suitable Kleisli category. This conforms to the idea expressed in the current paper that it is far more practical to lift monad $\mathbb{F}$ through $\mathbb{T}$ (67) and take advantage of the (enriched) Kleisli setting than working in the *composite monad* $\mathbb{T}\,\mathbb{F}$ in sets.

There is, however, one major difference between [16] and the current paper: in (67) there are two monads and not a single one, leading to other topic of the paper, that of *combining effects* [20]. Variants of (67) have been studied elsewhere [43], 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, Ref. [7] studies coalgebras of functor $S \to \mathbb{K} \times (\mathbb{K}^S_\omega)^I$ 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 coalgebraic framework for weighted automata was studied by one of the authors directly in suitable *categories of matrices* [37].

The current paper draws much from the work on lifting coalgebras to Kleisli categories by Hasuo et al. [15,17]. They study the lifting of $\mathbb{F}$-coalgebras in the category of sets and (total) functions to the Kleisli category of a (commutative) monad $\mathbb{T}$. Our work specializes this framework by further requiring $\mathbb{F}$ to be a (strong) monad too, recall sections 8 and 9.

Jacobs et al. [21] also address the combination of monads in coalgebraic models of state-transition systems by providing a framework where the two main approaches to coalgebraic trace semantics — via either Kleisli categories or Eilenberg-Moore categories — are thoroughly studied and related. The references of [21] also mention work on trace semantics for probabilistic systems which are related to the main topic of the current paper.

11. Conclusions

Faced with the need to quantify software (un)reliability in presence of faults arising from (intentionally) inexact hardware, the semantics of state-based systems is urged to evolve towards weighted nondeterminism, typically in a probabilistic way.

[30] In general, given a natural transformation $\alpha : \mathbb{F} \to \mathbb{G}$ in C, $\alpha \cdot \mathbb{F} f = \mathbb{G} f \cdot \alpha$ is preserved in $\mathsf{C}_{\mathbb{T}}$ in the sense that $\overline{\alpha} \cdot \overline{\mathbb{F}}\,\overline{f} = \overline{\mathbb{G}}\,\overline{f} \cdot \overline{\alpha}$ immediately holds, as it is straightforward to prove.

[31] A CPPO-enriched category is such that each homset forms a ω-complete partial order.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

This paper proposes that *semantic evolutions* of this kind be obtained without sacrificing the simplicity of the original (qualitative) definitions. 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 are expressed, we change to a suitable category (e.g. of matrices) tuned to the specific quantitative (e.g. probabilistic) effect.

In our case study, we move from the original category of sets where a component calculus is defined [3] and, rather than elaborating its definitions and proofs to the nondeterministic/probabilistic cases, we keep the original definitions by changing to suitable categories of relations/(stochastic) matrices. In the latter case, the main advantage is that all *probabilistic accounting* is silently carried out by (monadic) matrix composition and is not our concern.

As the original semantics are already monadic and coalgebraic, "keeping the definitions" entails monad lifting, as studied in Sect. 8. The approach works wherever state-based semantic models are structured around a pair $(\mathbb{F}, \mathbb{T})$ of monads subject to the (cumulative) set of conditions summarized in the following table:

$\mathbb{T}$	$\mathbb{F}$	Conditions	$\overline{\mathbb{F}}$
Monad	Functor	(49) + (50)	Functor
	Monad	+ (51) + (52)	Monad
Commutative monad	Strong monad	+ (61) or (62)	Strong monad

Jacobs et al. [21] refer to properties (49), (50) as the *Kleisli*-laws and to (51), (52) as the *Eilenberg–Moore*-laws, since the latter ensure functor lifting in such categories. Our Theorem 7 says that the latter also suffice for the lifted *functor* to be also a *monad* in $\mathsf{C}_{\mathbb{T}}$ and Theorem 9 finds a condition (61) (equivalent to (62)) for such a lifted monad to be *strong*.

In general, this *"keep definition, change category"* approach consists of investing in the Kleisli category $\mathsf{C}_{\mathbb{T}}$ of the monad $\mathbb{T}$ chosen to capture the new (e.g. quantitative) effect — the so-called *branching* monad in [17], where the lifted monad $\mathbb{F}$ is referred to as the *transition* monad. Note that a lot more is demanded from $\mathbb{T}$ compared to $\mathbb{F}$, namely it is required to be commutative (therefore strong) and order-enriched [17].

$\mathbb{T}$	Kleisli
$\mathbb{P}$	Relation algebra
Vec	Matrix algebra
$\mathbb{D}$	Stochastic matrices
Giry	Stochastic relations

Altogether, the proposed strategy is useful because such a lifting leads to rich algebraic theories and to *relation algebra*/*linear algebra*[32] in particular, both offering useful *pointfree styled* calculi, as shown in the table above.

Ideally, the lifting should preserve theories, not only definitions. (E.g. the theory of component behavior equivalence of [3], in our case study.) 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 relation algebra to *categorial quantum physics* under the umbrella of *monoidal* categories. Thus the remarks by Coecke and Paquette [10]:

> *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.

This topic is regarded as central to the work to be carried out in the future, as suggested below.

12. Future work

This paper is part of a line of research aiming at promoting linear algebra as the "natural" evolution of (pointfree) relation algebra towards quantitative reasoning in the software sciences. Other applications of linear algebra in the software field [28,29,34,36,37] can be regarded as instances of the generic strategy put forward in this paper. Thus far, the main difficulty encountered in this technique is the fact that *strong monads* do not lift in general because the *pairing* operator lifts differently depending on the "branching" monad.

For the distribution monad, this issue is addressed in e.g. [34] where it is shown that, if one of the arguments of probabilistic pairing is pure (i.e., of the form $\overline{f} = (\eta_{\mathbb{T}} \cdot f)^{\flat}$, recall Sect. 8) then the universal property of pairing holds. This helps in some situations but is far from providing a general answer, calling for further research possibly in connection to recent advances in categorial quantum physics and monoidal categories [9].

A fully fledged, coalgebraic trace semantics for probabilistic, component oriented software systems will call for sub-distributions and, more generally, measure theory [23,39] in particular if applied to hybrid systems [35].

Not every combination of monads is suitable for immediate lifting, as is the case of some of the patterns addressed in [21]. The alternative approach proposed in this paper should be carefully evaluated and studied, in particular in what concerns its application to software component calculi in the general setting of [2,3].

[32] This carries over to more sophisticated algebras and monads, for instance that of *stochastic relations*, the "Kleisli lifting" of the Giry monad [39].

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

Acknowledgements

This work is funded by ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) within project FCOMP-01-0124-FEDER-020537 and by National Funds through the *FCT – Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) grant number UMINHO/BI/9/2014.

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

Appendix A. Strong and commutative monads in brief

For easy reference and self-containment, this appendix gathers together standard definitions that lead to the notion of a *commutative monad* which is at focus in the current paper. We use diagrams in order to make types clear and more explicit. For further reference see e.g. [17,24].

Definition 11 *(Monad).* A *monad* is an endofunctor $\mathbb{F} : \mathsf{C} \to \mathsf{C}$ on a category C equipped with two natural transformations $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{F}\,X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{F}^2\,X$ satisfying the equalities captured by the following commutative diagrams:

$$\begin{array}{ccccc}
\mathbb{F}\,X & \xrightarrow{\eta_{\mathbb{T}}} & \mathbb{F}^2\,X & \xleftarrow{\mu_{\mathbb{T}}} & \mathbb{F}^3\,X \\
{\scriptstyle \mathbb{F}\,\eta_{\mathbb{T}}}\downarrow & \searrow^{id} & {\scriptstyle \mu_{\mathbb{T}}}\downarrow & & \downarrow{\scriptstyle \mathbb{F}\,\mu_{\mathbb{T}}} \\
\mathbb{F}^2\,X & \xrightarrow[\mu_{\mathbb{T}}]{} & \mathbb{F}\,X & \xleftarrow{\mu_{\mathbb{T}}} & \mathbb{F}^2\,X
\end{array}$$

The naturality conditions of $\eta_{\mathbb{F}}$ and $\mu_{\mathbb{F}}$ are captured by the commutative diagrams, for all $f : X \to Y$:

$$\begin{array}{ccccc}
X & \xrightarrow{\eta_{\mathbb{F}}} & \mathbb{F}\,X & \xleftarrow{\mu_{\mathbb{F}}} & \mathbb{F}^2\,X \\
{\scriptstyle f}\downarrow & & {\scriptstyle \mathbb{F}\,f}\downarrow & & \downarrow{\scriptstyle \mathbb{F}^2\,f} \\
Y & \xrightarrow{\eta_{\mathbb{F}}} & \mathbb{F}\,Y & \xleftarrow{\mu_{\mathbb{T}}} & \mathbb{F}^2\,Y
\end{array}$$

□

Definition 12 *(Strong functor).* A functor $\mathbb{F}$ is said to be *strong* wherever it comes equipped with two natural transformations

$$X \times (\mathbb{F}\,Y) \xrightarrow{\tau_l} \mathbb{F}\,(X \times Y) \xleftarrow{\tau_r} (\mathbb{F}\,X) \times Y$$

such that

$$\begin{array}{ccccc}
\mathbb{F}\,((X \times Y) \times Z) & \xleftarrow{\tau_r} & \mathbb{F}\,(X \times Y) \times Z & \xleftarrow{\tau_r \times id} & (\mathbb{F}\,X \times Y) \times Z \\
{\scriptstyle \mathsf{a}}\downarrow & & & & \downarrow{\scriptstyle \mathsf{a}} \\
\mathbb{F}\,(X \times (Y \times Z)) & & \xleftarrow[\tau_r]{} & & \mathbb{F}\,X \times (Y \times Z)
\end{array} \tag{A.1}$$

and

$$\begin{array}{ccc}
\mathbb{F}\,(X \times 1) & \xleftarrow{\tau_r} & (\mathbb{F}\,X) \times 1 \\
{\scriptstyle \mathbb{F}\,\mathsf{lft}}\downarrow & \swarrow_{\mathsf{lft}} & \\
\mathbb{F}\,X & &
\end{array} \tag{A.2}$$

hold, where lft and a are well-known isomorphisms. (Similarly for τ_l and $\mathsf{rgt} : 1 \times X \to X$.) The naturality conditions of τ_l and τ_r are captured by the commutative diagrams

$$\begin{array}{ccccc}
X \times (\mathbb{F}\,Y) & \xrightarrow{\tau_l} & \mathbb{F}\,(X \times Y) & \xleftarrow{\tau_r} & (\mathbb{F}\,X) \times Y \\
{\scriptstyle f \times (\mathbb{F}\,g)}\downarrow & & {\scriptstyle \mathbb{F}\,(f \times g)}\downarrow & & \downarrow{\scriptstyle (\mathbb{F}\,f) \times g} \\
A \times (\mathbb{F}\,B) & \xrightarrow{\tau_l} & \mathbb{F}\,(A \times B) & \xleftarrow{\tau_r} & (\mathbb{F}\,A) \times B
\end{array} \tag{A.3}$$

for all $f : X \to A$ and $g : Y \to B$. □

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

Definition 13 *(Strong natural transformation).* Given strong functors $\mathbb{F}$ and $\mathbb{G}$, natural transformation $\alpha : \mathbb{F}\,X \to \mathbb{G}\,X$ is said to be *strong* wherever it commutes with $(\times Y)$ in the following sense (similarly for $(Y\times)$ and τ_l):

$$\begin{array}{ccc} \mathbb{F}\,(X \times Y) & \xleftarrow{\tau_r} & (\mathbb{F}\,X) \times Y \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\alpha\times id} \\ \mathbb{G}\,(X \times Y) & \xleftarrow[\tau_r]{} & (\mathbb{G}\,X) \times Y \end{array} \qquad \square \tag{A.4}$$

Definition 14 *(Strong monad).* A monad $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2 X$ is said to be *strong* wherever: (a) $\mathbb{F}$ is a strong functor; (b) $\mu_{\mathbb{F}}$ and $\eta_{\mathbb{F}}$ are strong natural transformations. For $\alpha = \eta_{\mathbb{F}}$ (A.4) instantiates to $\eta_{\mathbb{F}} = \tau_r \cdot (\eta_{\mathbb{F}} \times id)$. For $\alpha = \mu_{\mathbb{F}}$, (A.4) will instantiate to $\tau_r \bullet \tau_r = \tau_r \cdot (\mu_{\mathbb{F}} \times id)$ since

$$\begin{aligned} &\tau_r : \mathbb{F}^2\,X \times Y \to \mathbb{F}^2\,(X \times Y) \\ &\tau_r = (\mathbb{F}\,\tau_r) \cdot \tau_r \end{aligned}$$

(Similarly for τ_l.) $\square$

Definition 15 *(Commutative monad).* A strong monad $\mathbb{F}$ said to be *commutative* wherever the equality holds:

$$\tau_r \bullet \tau_l = \tau_l \bullet \tau_r \tag{A.5}$$

In this case it makes sense to define the "double strength" operator $\delta = \tau_r \bullet \tau_l$. $\square$

Appendix B. Auxiliary definitions and proofs

Basics. The following operators are required in the main body of the paper:

$$\mathsf{dr}^{\circ} = [id \times i_1, id \times i_2] \tag{B.1}$$

$$\mathsf{cozip} = [\mathbb{F}\,i_1, \mathbb{F}\,i_2] \tag{B.2}$$

where $[f, g]$ is the *junc* combinator satisfying $[f, g] \cdot i_1 = f$ and $[f, g] \cdot i_2 = g$. From the type

$$\mathsf{cozip} : \mathbb{F}\,A + \mathbb{F}\,B \to \mathbb{F}\,(A + B) \tag{B.3}$$

the corresponding free theorem is recorded

$$\mathbb{F}\,(f + g) \cdot \mathsf{cozip} = \mathsf{cozip} \cdot (\mathbb{F}\,f + \mathbb{F}\,g) \tag{B.4}$$

for further reference. The operator furthermore satisfies property

$$[\mathbb{F}\,(id \times i_1), \mathbb{F}\,(id \times i_2)] = (\mathbb{F}\,\mathsf{dr}^{\circ}) \cdot \mathsf{cozip} \tag{B.5}$$

Proofs related to Section 5. Proof of fact (23), that of (24) being similar:

$$\begin{aligned} & (\mathsf{extr}\,(p \oplus q))_{\{i_1 \to id\}} \\ = & \quad \{\ \text{wiring (16)}\ ;\ \text{functor}\ \mathbb{F}\ \} \\ & \mathsf{extr}\,(p \oplus q) \cdot (id \times i_1) \\ = & \quad \{\ \text{definition of extr}\ m\ (7)\ \} \\ & \mathbb{F}\,\mathsf{xr} \cdot \tau_r \cdot ((p \oplus q) \times id) \cdot \mathsf{xr} \cdot (id \times i_1) \\ = & \quad \{\ \mathsf{xr}\ \text{natural property};\ \times\ \text{functor}\ \} \\ & \mathbb{F}\,\mathsf{xr} \cdot \tau_r \cdot (((p \oplus q) \cdot (id \times i_1)) \times id) \cdot \mathsf{xr} \\ = & \quad \{\ (p \oplus q) \cdot (id \times i_1) = (p \oplus q)_{\{i_1 \to id\}} = p_{\{id \to i_1\}}\ (17), (18)\ \} \\ & \mathbb{F}\,\mathsf{xr} \cdot \tau_r \cdot ((\mathbb{F}\,(id \times i_1) \cdot p) \times id) \cdot \mathsf{xr} \\ = & \quad \{\ \times\ \text{functor}\ \} \\ & \mathbb{F}\,\mathsf{xr} \cdot \tau_r \cdot (\mathbb{F}\,(id \times i_1) \times id) \cdot (p \times id) \cdot \mathsf{xr} \end{aligned}$$

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

$$
\begin{array}{ll}
= & \{ \text{ natural properties of } \tau_r \text{ and } \mathsf{xr} \ \} \\
& \mathbb{F}\ (id \times i_1) \cdot \mathbb{F}\ \mathsf{xr} \cdot \tau_r \cdot (p \times id) \cdot \mathsf{xr} \\
= & \{ \ (7) \text{ and } (17) \text{ again } \ \} \\
& (\mathsf{extr}\ p)_{\{id \to i_1\}} \qquad \square
\end{array}
$$

Proof of (25), that of (26) being similar:

$$
\begin{array}{ll}
& (\mathsf{extl}\ (p \oplus q))_{\{i_1 \to id\}} \\
= & \{ \text{ wiring } (17) \ ; \bullet \,/\, \cdot \text{ associativity } \} \\
& \mathsf{extl}\ (p \oplus q) \cdot (id \times i_1) \\
= & \{ \text{ definition of } \mathsf{extl}\ (6) \ \} \\
& \mathbb{F}\ \mathsf{a}^\circ \cdot \tau_l \cdot (id \times (p \oplus q)) \cdot \mathsf{a} \cdot (id \times i_1) \\
= & \{ \mathsf{a} \text{ natural } \} \\
& \mathbb{F}\ \mathsf{a}^\circ \cdot \tau_l \cdot (id \times (p \oplus q)) \cdot (id \times (id \times i_1)) \cdot \mathsf{a} \\
= & \{ \times \text{ functor; } (21) \text{ and } (17) \ \} \\
& \mathbb{F}\ \mathsf{a}^\circ \cdot \tau_l \cdot (id \times \mathbb{F}\ (id \times i_1)) \cdot (id \times p) \cdot \mathsf{a} \\
= & \{ \tau_l \text{ natural; } \mathbb{F} \text{ functor; } \mathsf{a}^\circ \text{ natural } \} \\
& \mathbb{F}\ (id \times i_1) \cdot \mathbb{F}\ \mathsf{a}^\circ \cdot \tau_l \cdot (id \times p) \cdot \mathsf{a} \\
= & \{ \text{ definitions of } \mathsf{extl}\ (6) \text{ and wiring } (17) \ \} \\
& (\mathsf{extl}\ p)_{\{id \to i_1\}} \qquad \square
\end{array}
$$

Proof of Theorem 7. The standard monadic laws, e.g. $\overline{\mu_{\mathbb{F}}} \cdot \overline{\eta_{\mathbb{F}}} = \overline{id}$, hold by construction (recall footnote 26). The proofs of the remaining natural laws,

$$\overline{\mathbb{F}}\, f^\flat \cdot \overline{\eta_{\mathbb{F}}} = \overline{\eta_{\mathbb{F}}} \cdot f^\flat \tag{B.6}$$

$$\overline{\mathbb{F}}\, f^\flat \cdot \overline{\mu_{\mathbb{F}}} = \overline{\mu_{\mathbb{F}}} \cdot (\overline{\mathbb{F}}^2\, f^\flat) \tag{B.7}$$

are given next. The naturality of $\overline{\eta_{\mathbb{F}}}$ (B.6) relies on property (51) of the definition of a distributive law:

$$
\begin{array}{ll}
& \overline{\mathbb{F}}(f^\flat) \cdot \overline{\eta_{\mathbb{F}}} = \overline{\eta_{\mathbb{F}}} \cdot f^\flat \\
\equiv & \{ \text{ definition } (53) \ ; (45) \text{ and } (46) \ \} \\
& (\lambda \cdot \mathbb{F} f \cdot \eta_{\mathbb{F}})^\flat = (\mathbb{T}\ \eta_{\mathbb{F}} \cdot f)^\flat \\
\equiv & \{ \text{ drop } (\_)^\flat \ ; \text{ natural } \eta_{\mathbb{F}} \ \} \\
& \lambda \cdot \eta_{\mathbb{F}} \cdot f = \mathbb{T} \eta_{\mathbb{F}} \cdot f \\
\Leftarrow & \{ \text{ Leibniz } \} \\
& \lambda \cdot \eta_{\mathbb{F}} = \mathbb{T} \eta_{\mathbb{F}} \\
\equiv & \{ \text{ assume } (51) \ \} \\
& \mathit{true} \qquad \square
\end{array}
$$

The naturality of $\overline{\mu_{\mathbb{F}}}$ (B.7) relies on property (52) of the definition of a distributive law:

$$
\begin{array}{ll}
& \overline{\mathbb{F}}(f^\flat) \cdot \overline{\mu_{\mathbb{F}}} = \overline{\mu_{\mathbb{F}}} \cdot \overline{\mathbb{F}}(\overline{\mathbb{F}} f^\flat) \\
\equiv & \{ \text{ definitions } (53) \text{ and } (54) \ \} \\
& (\lambda \cdot \mathbb{F} f)^\flat \cdot (\eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}})^\flat = (\eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}})^\flat \cdot (\lambda \cdot \mathbb{F}(\lambda \cdot \mathbb{F} f))^\flat \\
\equiv & \{ \ (44) \ ; \text{ drop } (\_)^\flat \ \} \\
& \mu_{\mathbb{T}} \cdot \mathbb{T}(\lambda \cdot \mathbb{F} f) \cdot (\eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}}) = \mu_{\mathbb{T}} \cdot \mathbb{T}(\eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}}) \cdot (\lambda \cdot \mathbb{F}(\lambda \cdot \mathbb{F} f))
\end{array}
$$

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

$$
\begin{array}{ll}
\equiv & \{\ \text{functors}\ \} \\
& \mu_{\mathbb{T}} \cdot \mathbb{T}\lambda \cdot \mathbb{T}(\mathbb{F} f) \cdot \eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}} = \mu_{\mathbb{T}} \cdot \mathbb{T}\eta_{\mathbb{T}} \cdot \mathbb{T}\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda \cdot \mathbb{F}^2 f \\
\equiv & \{\ \text{natural } \eta_{\mathbb{T}} \text{ followed by natural } \mu_{\mathbb{F}}\ \} \\
& \mu_{\mathbb{T}} \cdot \mathbb{T}\lambda \cdot \eta_{\mathbb{T}} \cdot \mu_{\mathbb{F}} \cdot \mathbb{F}^2 f = \mu_{\mathbb{T}} \cdot \mathbb{T}\eta_{\mathbb{T}} \cdot \mathbb{T}\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda \cdot \mathbb{F}^2 f \\
\equiv & \{\ \text{natural } \eta_{\mathbb{T}} \text{ again}\ \} \\
& \mu_{\mathbb{T}} \cdot \eta_{\mathbb{T}} \cdot \lambda \cdot \mu_{\mathbb{F}} \cdot \mathbb{F}^2 f = \mu_{\mathbb{T}} \cdot \mathbb{T}\,\eta_{\mathbb{T}} \cdot \mathbb{T}\,\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda \cdot \mathbb{F}^2 f \\
\equiv & \{\ \text{monad } \mathbb{T}\ \} \\
& \lambda \cdot \mu_{\mathbb{F}} \cdot \mathbb{F}^2 f = \mathbb{T}\,\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda \cdot \mathbb{F}^2 f\lambda \cdot \mu_{\mathbb{F}} = \mathbb{T}\,\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda \\
\equiv & \{\ \text{assume (52)}\ \} \\
& \mathit{true} \qquad \square
\end{array}
$$

Proof of Theorem 8 (Free-monad lifting). Proof of (51) for $\mathbb{F} = \mathbb{F}_{\mathbb{G}}$:

$$
\begin{array}{ll}
& \lambda \cdot \eta \\
= & \{\ \text{definitions of } \lambda \text{ (57) and } \eta \text{ (55)}\ \} \\
& (\!|[\mathbb{T}\,\mathsf{in}_1, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) \cdot \mathsf{in} \cdot i_1 \\
= & \{\ (\!|\cdot|\!) \text{ cancellation [6]}\ \} \\
& [\mathbb{T}\,\mathsf{in}_1, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}] \cdot (id + \mathbb{F}\,\lambda) \cdot i_1 \\
= & \{\ \text{coproducts (absorption, cancellation, etc.) [6]}\ \} \\
& \mathbb{T}\,\mathsf{in}_1 \\
= & \{\ (55)\ \} \\
& \mathbb{T}\,\eta \qquad \square
\end{array}
$$

Concerning the proof of (52), let us annotate the instance of this equality for $\mathbb{F} = \mathbb{F}_{\mathbb{G}}$ with a sketch of the proof strategy:

$$
\overbrace{\underbrace{\lambda \cdot \mu}_{(\!|\alpha|\!)} = (\mathbb{T}\,\mu) \cdot \underbrace{\lambda \cdot \mathbb{F}_{\mathbb{G}}\,\lambda}_{(\!|\beta|\!)}}^{(\!|\cdot|\!)\text{-fusion}} \tag{B.8}
$$

In words: we first need to calculate α and β and then resort to standard $(\!|\cdot|\!)$-fusion [6] to complete the proof. It turns out that α is calculated using the same (standard) procedure:

$$
\begin{array}{ll}
& \lambda \cdot \mu = (\!|\alpha|\!) \\
\equiv & \{\ \text{definition of } \mu \text{ (56)}\ \} \\
& \lambda \cdot (\!|[id, \mathsf{in}_2]|\!) = (\!|\alpha|\!) \\
\Leftarrow & \{\ (\!|\cdot|\!) \text{ fusion; unfold } \alpha = [\alpha_1, \alpha_2]\ \} \\
& \lambda \cdot [id, \mathsf{in}_2] = [\alpha_1, \alpha_2] \cdot (id + \mathbb{F}\,\lambda) \\
\equiv & \{\ \text{coproducts}\ \} \\
& \begin{cases} \lambda = \alpha_1 \\ \lambda \cdot \mathsf{in}_2 = \alpha_2 \cdot \mathbb{F}\,\lambda \end{cases}
\end{array}
$$

To obtain α_2, we unfold the second equality just above:

$$
\begin{array}{ll}
& \lambda \cdot \mathsf{in}_2 = \alpha_2 \cdot \mathbb{F}\,\lambda \\
\equiv & \{\ \text{definition of } \lambda \text{ (57); } (\!|\cdot|\!)\text{-cancellation again}\ \} \\
& [\mathbb{T}\,\mathsf{in}_1, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}] \cdot (id + \mathbb{F}\,\lambda) \cdot i_2 = \alpha_2 \cdot \mathbb{F}\,\lambda \\
\equiv & \{\ \text{coproducts (as before)}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} \cdot \mathbb{F}\,\lambda = \alpha_2 \cdot \mathbb{F}\,\lambda \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
& \alpha_2 = \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}
\end{array}
$$

Altogether, we obtain:

$$\lambda \cdot \mu = (\!|[\lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) \tag{B.9}$$

— recall (B.8). Next, we want to calculate β:

$$
\begin{array}{ll}
& \lambda \cdot \mathbb{F}_{\mathbb{G}}\,\lambda = (\!|\beta|\!) \\
\equiv & \quad \{\ \lambda\ (57);\ (\!|\cdot|\!)\text{-absorption [6]}\ \} \\
& \beta = [\mathbb{T}\,\mathsf{in}_1 \cdot \lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]
\end{array}
$$

Thus:

$$\lambda \cdot \mathbb{F}_{\mathbb{G}}\,\lambda = (\!|[\mathbb{T}\,\mathsf{in}_1 \cdot \lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) \tag{B.10}$$

We can now address (52):

$$
\begin{array}{ll}
& \mathbb{T}\,\mu \cdot (\lambda \cdot \mathbb{F}_{\mathbb{G}}\,\lambda) = \lambda \cdot \mu \\
\equiv & \quad \{\ \text{(B.9) and (B.10) above}\ \} \\
& \mathbb{T}\,\mu \cdot (\!|[\mathbb{T}\,\mathsf{in}_1 \cdot \lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) = (\!|[\lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}]|\!) \\
\Leftarrow & \quad \{\ \text{standard } (\!|\cdot|\!)\text{-fusion over functor } id + \mathbb{G}f\ \} \\
& \mathbb{T}\,\mu \cdot [\mathbb{T}\,\mathsf{in}_1 \cdot \lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}] = [\lambda, \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}}] \cdot (id + \mathbb{G}\,(\mathbb{T}\,\mu)) \\
\equiv & \quad \{\ \text{coproducts}\ \} \\
& \begin{cases} \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_1 \cdot \lambda = \lambda \\ \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} = \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} \cdot \mathbb{G}\,(\mathbb{T}\,\mu) \end{cases}
\end{array}
$$

Concerning the first branch:

$$
\begin{array}{ll}
& \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_1 \cdot \lambda \\
= & \quad \{\ \text{functor } \mathbb{T}\ ;\ \mathsf{in}_1 = \eta\ \} \\
& \mathbb{T}\,(\mu \cdot \eta) \cdot \lambda \\
= & \quad \{\ \mu \cdot \eta = id;\ \mathbb{T}\,id = id\ \text{(monads, functors)}\ \} \\
& \lambda \qquad \square
\end{array}
$$

And finally, the second branch:

$$
\begin{array}{ll}
& \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} = \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} \cdot \mathbb{G}\,(\mathbb{T}\,\mu) \\
\equiv & \quad \{\ \lambda_{\mathbb{G}}\ \text{natural}\ \} \\
& \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_2 \cdot \lambda_{\mathbb{G}} = \mathbb{T}\,\mathsf{in}_2 \cdot \mathbb{T}\,(\mathbb{G}\,\mu) \cdot \lambda_{\mathbb{G}} \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
& \mathbb{T}\,\mu \cdot \mathbb{T}\,\mathsf{in}_2 = \mathbb{T}\,\mathsf{in}_2 \cdot \mathbb{T}\,(\mathbb{G}\,\mu) \\
\Leftarrow & \quad \{\ \text{functor } \mathbb{T}\ \} \\
& \mu \cdot \mathsf{in}_2 = \mathsf{in}_2 \cdot \mathbb{G}\,\mu \\
\equiv & \quad \{\ \text{catamorphism } \mu\ (56)\ \} \\
& \mathit{true} \qquad \square
\end{array}
$$

This completes the proof.[33]

[33] Note that there is a sketch of this construction in [20].

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

Proof of Theorem 9 (Strong monad lifting). Assuming that, by Theorems 6 and 7, $\overline{\mathbb{F}}$ is a monad in $\mathsf{C}_{\mathbb{T}}$, it remains to check under what conditions it is *strong* (Definition 14).

As before (recall footnote 26) we are not concerned with the properties stated in the diagrams of Definitions 12 and 13, which hold by construction of $\overline{\mathbb{F}}$, $\overline{\tau_l}$ and $\overline{\tau_r}$. It is rather the naturality of $\overline{\tau_l}$ (58) and $\overline{\tau_r}$ (59) which needs checking. Concerning (58):

$$
\begin{array}{ll}
& \overline{\tau_l} \cdot (g^\flat \otimes \overline{\mathbb{F}}\, f^\flat) = \overline{\mathbb{F}}\, (g^\flat \otimes f^\flat) \cdot \overline{\tau_l} \\
\equiv & \quad \{\ (53)\ ;\ (60)\ \} \\
& \overline{\tau_l} \cdot (\delta \cdot (g \times (\lambda \cdot \mathbb{F}\, f)))^\flat = (\lambda \cdot \mathbb{F}\, (\delta \cdot (g \times f)))^\flat \cdot \overline{\tau_l} \\
\equiv & \quad \{\ (45)\ ;\ (46)\ \} \\
& (\mathbb{T}\, \tau_l \cdot \delta \cdot (g \times (\lambda \cdot \mathbb{F}\, f)))^\flat = (\lambda \cdot \mathbb{F}\, (\delta \cdot (g \times f)) \cdot \tau_l)^\flat \\
\equiv & \quad \{\ \text{drop } (\_)^\flat\ \} \\
& \mathbb{T}\, \tau_l \cdot \delta \cdot (g \times (\lambda \cdot \mathbb{F}\, f)) = \lambda \cdot \mathbb{F}\, (\delta \cdot (g \times f)) \cdot \tau_l \\
\equiv & \quad \{\ \text{products ; functor } \mathbb{F}\ \} \\
& \mathbb{T}\, \tau_l \cdot \delta \cdot (id \times \lambda) \cdot (g \times \mathbb{F}\, f) = \lambda \cdot \mathbb{F}\, \delta \cdot \mathbb{F}(g \times f)) \cdot \tau_l \\
\equiv & \quad \{\ \text{naturality of } \tau_l\ \} \\
& \mathbb{T}\, \tau_l \cdot \delta \cdot (id \times \lambda) \cdot (g \times \mathbb{F}\, f) = \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l \cdot (g \times \mathbb{F}\, f) \\
\\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
& \mathbb{T}\, \tau_l \cdot \delta \cdot (id \times \lambda) = \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l \\
\equiv & \quad \{\ \text{assume (61)}\ \} \\
& true \qquad \square
\end{array}
$$

So (61) is required.

The proof that (59) requires (62) is similar. However, this is not an extra condition, as — since $\tau_r = \mathbb{F}\, \mathsf{swap} \cdot \tau_l \cdot \mathsf{swap}$, where swap is isomorphism $\mathsf{swap} = \pi_2 \vartriangle \pi_1$ — (62) is equivalent to (61), as shown next:

$$
\begin{array}{ll}
& \lambda \cdot \mathbb{F}\, \delta \cdot \tau_r = \mathbb{T}\, \tau_r \cdot \delta \cdot (\lambda \times id) \\
\equiv & \quad \{\ \tau_r = \mathbb{F}\, \mathsf{swap} \cdot \tau_l \cdot \mathsf{swap}\ \} \\
& \lambda \cdot \mathbb{F}\, \delta \cdot \mathbb{F}\, \mathsf{swap} \cdot \tau_l \cdot \mathsf{swap} = \mathbb{T}\, \tau_r \cdot \delta \cdot (\lambda \times id) \\
\equiv & \quad \{\ \mathbb{T}\, \mathsf{swap} \cdot \delta = \delta \cdot \mathsf{swap}\ ;\ \mathsf{swap} \text{ isomorphism}\ \} \\
& \lambda \cdot \mathbb{F}\, (\mathbb{T}\, \mathsf{swap} \cdot \delta) \cdot \tau_l = \mathbb{T}\, \tau_r \cdot \delta \cdot (\lambda \times id) \cdot \mathsf{swap} \\
\equiv & \quad \{\ \text{naturality } (\lambda \text{ and swap})\ ;\ \text{functor } \mathbb{F}\ \} \\
& \mathbb{T}\, \mathbb{F}\, \mathsf{swap} \cdot \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l = \mathbb{T}\, \tau_r \cdot \delta \cdot \mathsf{swap} \cdot (id \times \lambda) \\
\equiv & \quad \{\ \mathbb{T}\, \mathsf{swap} \cdot \delta = \delta \cdot \mathsf{swap}\ \} \\
& \mathbb{T}\, \mathbb{F}\, \mathsf{swap} \cdot \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l = \mathbb{T}\, \tau_r \cdot \mathbb{T}\, \mathsf{swap} \cdot \delta \cdot (id \times \lambda) \\
\equiv & \quad \{\ \tau_r \cdot \mathsf{swap} = \mathbb{F}\, \mathsf{swap} \cdot \tau_l\ \} \\
& \mathbb{T}\, \mathbb{F}\, \mathsf{swap} \cdot \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l = \mathbb{T}\, (\mathbb{F}\, \mathsf{swap} \cdot \tau_l) \cdot \delta \cdot (id \times \lambda) \\
\equiv & \quad \{\ \text{drop isomorphism } \mathbb{T}\, \mathbb{F}\, \mathsf{swap}\ \} \\
& \lambda \cdot \mathbb{F}\, \delta \cdot \tau_l = \mathbb{T}\, (\tau_l) \cdot \delta \cdot (id \times \lambda) \qquad \square
\end{array}
$$

References

[1] R. Backhouse, D. Michaelis, Exercises in quantifier manipulation, in: T. Uustalu (Ed.), MPC'06, in: LNCS, vol. 4014, Springer, 2006, pp. 70–81.
[2] L. Barbosa, Components as coalgebras, Ph.D. thesis, University of Minho, December 2001.
[3] L. Barbosa, Towards a calculus of state-based software components, J. Univers. Comput. Sci. 9 (8) (August 2003) 891–909.
[4] L. Barbosa, J. Oliveira, Transposing partial components — an exercise on coalgebraic refinement, Theor. Comput. Sci. 365 (1) (2006) 2–22.
[5] J. Beck, Distributive laws, in: B. Eckmann (Ed.), Seminar on Triples and Categorical Homology Theory, in: Lecture Notes in Mathematics, vol. 80, Springer, 1969, pp. 119–140.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007

[6] R. Bird, O. de Moor, Algebra of Programming, Ser. Comput. Sci., Prentice–Hall International, 1997.
[7] F. Bonchi, M. Bonsangue, M. Boreale, J. Rutten, A. Silva, A coalgebraic perspective on linear weighted automata, Inf. Comput. 211 (2012) 77–105.
[8] C. Brink, W. Kahl, G. Schmidt (Eds.), Relational Methods in Computer Science, Springer-Verlag, New York, Inc., New York, NY, USA, 1997.
[9] B. Coecke (Ed.), New Structures for Physics, Lecture Notes in Physics, vol. 831, Springer, 2011.
[10] B. Coecke, É. Paquette, Categories for the practising physicist, in: New Structures for Physics, Springer, 2011, pp. 173–283, Chapter 3 of [9].
[11] V. Cortellessa, V. Grassi, A modeling approach to analyze the impact of error propagation on reliability of component-based systems, in: Component-Based Software Engineering, in: LNCS, vol. 4608, 2007, pp. 140–156.
[12] M. Erwig, S. Kollmannsberger, Functional pearls: probabilistic functional programming in Haskell, J. Funct. Program. 16 (January 2006) 21–34.
[13] P. Freyd, A. Scedrov, Categories, Allegories, Mathematical Library, vol. 39, North-Holland, 1990.
[14] M. Frias, G. Baum, A. Haeberer, Fork algebras in algebra, logic and computer science, Fundam. Inform. (1997) 1–25.
[15] I. Hasuo, Generic forward and backward simulations, in: C. Baier, H. Hermanns (Eds.), CONCUR 2006, in: LNCS, vol. 4137, Springer, Berlin, Heidelberg, 2006, pp. 406–420.
[16] I. Hasuo, C. Heunen, B. Jacobs, A. Sokolova, Coalgebraic components in a many-sorted microcosm, in: Proc. CALCO 2009, Udine, Italy, 7–10, Sep. 2009, 2009, pp. 64–80.
[17] I. Hasuo, B. Jacobs, A. Sokolova, Generic trace semantics via coinduction, Log. Methods Comput. Sci. 3 (4) (2007) 1–36.
[18] C. Hoare, S. van Staden, B. Möller, G. Struth, J. Villard, H. Zhu, P. O'Hearn, Developments in concurrent Kleene algebra, in: RAMiCS, in: LNCS, vol. 8428, Springer, 2014, pp. 1–18.
[19] J. Hughes, Generalising monads to arrows, Sci. Comput. Program. 37 (May 2000) 67–111.
[20] M. Hyland, G. Plotkin, J. Power, Combining effects: sum and tensor, Theor. Comput. Sci. 357 (1–3) (2006) 70–99.
[21] B. Jacobs, A. Silva, A. Sokolova, Trace semantics via determinization, J. Comput. Syst. Sci. 81 (5) (2015) 859–879.
[22] W. Kahl, Refinement and development of programs from relational specifications, Electron. Notes Theor. Comput. Sci. 44 (3) (2003) 4.1–4.43.
[23] H. Kerstan, B. König, Coalgebraic trace semantics for probabilistic transition systems based on measure theory, in: CONCUR 2012, in: LNCS, Springer, 2012, pp. 410–424.
[24] A. Kock, Monads on symmetric monoidal closed categories, Arch. Math. 21 (1970) 1–10.
[25] A. Kock, Strong functors and monoidal monads, Arch. Math. 23 (1) (1972) 113–120, http://dx.doi.org/10.1007/BF01304852.
[26] A. Lingamneni, C. Enz, K. Palem, C. Piguet, Synthesizing parsimonious inexact circuits through probabilistic design techniques, ACM Trans. Embed. Comput. Syst. 12(2s) (May 2013) 93:1–93:26.
[27] S. Liu, G. Trenkler, Hadamard, Khatri–Rao, Kronecker and other matrix products, Int. J. Inf. Syst. Sci. 4 (1) (2008) 160–177.
[28] H. Macedo, J. Oliveira, Typing linear algebra: a biproduct-oriented approach, Sci. Comput. Program. 78 (11) (2013) 2160–2191.
[29] H. Macedo, J. Oliveira, A linear algebra approach to OLAP, Form. Asp. Comput. 27 (2) (2015) 283–307.
[30] S. MacLane, Categories for the Working Mathematician, Springer, 1971.
[31] A. McIver, C. Morgan, Abstraction, Refinement and Proof for Probabilistic Systems, Monographs in Computer Science, Springer-Verlag, 2005.
[32] V. Miraldo, Object oriented programming with monadic Mealy machines, Technical Report TR-HASLab:04:2014, INESC TEC & U. Minho, Gualtar Campus, Braga, 2014, http://wiki.di.uminho.pt/twiki/bin/view/DI/FMHAS/TechnicalReports.
[33] P. Mulry, Lifting theorems for Kleisli categories, in: Mathematical Foundations of Programming Semantics, in: LNCS, vol. 802, Springer, Berlin, Heidelberg, 1994, pp. 304–319.
[34] D. Murta, J. Oliveira, A study of risk-aware program transformation, Sci. Comput. Program. 110 (2015) 51–77.
[35] R. Neves, L. Barbosa, D. Hofmann, M. Martins, Continuity as a computational effect, CoRR, http://arxiv.org/abs/1507.03219, 2015.
[36] J. Oliveira, Towards a linear algebra of programming, Form. Asp. Comput. 24 (4–6) (2012) 433–458.
[37] J. Oliveira, Weighted automata as coalgebras in categories of matrices, Int. J. Found. Comput. Sci. 24 (06) (2013) 709–728.
[38] J. Oliveira, Relational algebra for "just good enough" hardware, in: RAMiCS, in: LNCS, vol. 8428, Springer, Berlin, Heidelberg, 2014, pp. 119–138.
[39] P. Panangaden, Labelled Markov Processes, Imperial College Press, 2009.
[40] J. Rutten, Universal coalgebra: a theory of systems, Theor. Comput. Sci. 249 (1) (2000) 3–80 (Revised version of CWI Techn. Rep. CS-R9652, 1996).
[41] J. Rutten, Algebraic specification and coalgebraic synthesis of Mealy automata, Electron. Notes Theor. Comput. Sci. 160 (2006) 305–319.
[42] G. Schmidt, Relational Mathematics, Encyclopedia of Mathematics and Its Applications, vol. 132, Cambridge University Press, November 2010.
[43] A. Sokolova, Probabilistic systems coalgebraically: a survey, Theor. Comput. Sci. 412 (38) (2011) 5095–5110.
[44] M. Stamatelatos, H. Dezfuli, Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners, NASA/SP-2011-3421, 2nd edition, December 2011.
[45] M. Tanaka, Pseudo-distributive laws and a unified framework for variable binding, Ph.D. thesis, School of Informatics, University of Edinburgh, 2005.
[46] P. Wadler, Theorems for free!, in: 4th International Symposium on Functional Programming Languages and Computer Architecture, ACM, London, Sep. 1989, pp. 347–359.

Please cite this article in press as: J.N. Oliveira, V.C. Miraldo, "Keep definition, change category" — A practical approach to state-based system calculi, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.11.007