Typed Linear Algebra for Weighted (Probabilistic) Automata

J.N. Oliveira

Ref. [Ol12] — 2012

J.N. Oliveira. Typed linear algebra for weighted (probabilistic) automata. In *CIAA*, volume 7381 of *LNCS*, pages 52–65, 2012. Invited paper.

Typed linear algebra for weighted (probabilistic) automata

(Extended abstract)

José N. Oliveira

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

Abstract. There is a need for a language able to reconcile the recent upsurge of interest in quantitative methods in the software sciences with logic and set theory that have been used for so many years in capturing the qualitative aspects of the same body of knowledge. Such a *lingua franca* should be typed, polymorphic, diagrammatic, calculational and easy to blend with traditional notation.
This paper puts forward *typed linear algebra* (LA) as a candidate notation for such a role. Typed LA emerges from regarding matrices as morphisms of suitable categories whereby traditional linear algebra is equipped with a type system.
In this paper we show typed LA at work in describing weighted (probabilistic) automata. Some attention is paid to the interface between the index-free language of matrix combinators and the corresponding index-wise notation, so as to blend with traditional set theoretic notation.

Keywords: Weighted automata, linear algebra, categories of matrices.

> *"Quantitative Formal Methods deals with systems whose behaviour of interest is more than the traditional Boolean "correct" or "incorrect" judgment. (...) The aim of the workshop was to create a new forum where current and novel theories and application areas of quantitative methods could be discussed, together with the verification techniques that might apply to them.*
>
> Andova et al. [2]

1 Introduction

There is a trend towards *quantitative methods* in computing. Further to predicting that something "may happen", going quantitative should allow one to anticipate "*how often or costly it will happen*". Or, looking from the negative side of things, if something bad can take place one wishes to know how likely is it to occur.

As happened with other sciences in the past (eg. physics), computer science is in some sense becoming *probabilistic*. However, traditional notation for probabilities is too descriptive and not meant for proving and calculating software as we understand this activity today. Quoting Hehner [14]:

> *Perhaps a thousand years ago the philosophers of the time [might give] reasons why their answer is right. Now we don't argue; we formalize, calculate, and unformalize.*

There has been work on tuning probabilistic notation and reasoning to software design. McIver and Morgan [22] develop a method for rigorous reasoning about probabilistic programs that includes a calculus which, in the Hoare style, operates at the level of the program text. At programming level, Erwig and Kollmansberger [11] give a collection of modules that make up a probabilistic functional programming library in Haskell based on the (finite) distribution monad. More recently, Gibbons and Hinze [13] have shown how to perform equational reasoning about programs that exploit both nondeterministic and probabilistic choice as part of a more ambitious plan to reason about effectful computations in general.

Sokolova [26] presents a coalgebraic analysis of probabilistic systems in a way that connects two main-stream research areas: coalgebraic reasoning and probabilistic modeling and verification. This work builds upon foundational work by Larsen and Skou [15] on probabilistic bisimulation. Broadening scope, recent work by Bonchi et al. [8] gives a coalgebraic perspective on so-called *linear weighted automata*, which generalize the probabilistic ones.

Weighted automata. Weighted automata [9, 10, 8] are a generalisation of finite state, non-deterministic automata where each state transition, in addition to some input, involves a quantity indicative of the *weight* (expressing eg. cost or probability) of its execution. The minimal structure for expressing weights is a *semiring* $(\mathbb{S}; +, \times, 0, 1)$ where $(\mathbb{S}; +, 0)$ is a commutative monoid, $(\mathbb{S}; \times, 1)$ is a monoid, multiplication distributes over addition and 0 annihilates multiplication ($0 \times s = s \times 0 = 0$).

Following [10], a weighted finite automaton $W = (A, Q; \lambda, \mu, \gamma)$ consists of an input alphabet A, a finite set of states Q and three functions: $\lambda, \gamma : Q \to \mathbb{S}$ are weight functions for entering and leaving a state, respectively, and $\mu : A \to \mathbb{S}^{Q \times Q}$ is such that $\mu(a)(p, q)$ indicates the cost of transition $p \xrightarrow{a} q$. Cost 0 means that there is no transition from p to q labelled a.

For $\mathbb{S}$ the Boolean algebra $\mathbb{B}$ of truth values, a weighted automaton becomes a (non-deterministic) labelled transition system (LTS), or non-deterministic finite-state automaton (FSA): $\mu(a) \in \mathbb{B}^{Q \times Q}$ is the state-transition relation associated to input a, λ is the set of initial states and γ the set of terminal states. For $\mathbb{S}$ the interval $[0, 1]$ of the real numbers ($\mathbb{R}$) W can be regarded as a *probabilistic automaton* under certain conditions [1]. Bonchi et al. [8] only consider μ and the output function γ. Their coalgebraic perspective twists the type of μ into

[1] For a comprehensive analysis and taxonomy of probabilistic systems see eg. [26].

$Q \to (\mathbb{S}^Q)^A$ and then amalgamates γ and μ into a coalgebra of functor $\mathsf{F}X = \mathbb{S} \times (\mathbb{S}^X)^A$.

State transition matrices. For each $a \in A$, $\mu(a) \in \mathbb{S}^{Q\times Q}$ can be regarded as a Q-indexed *matrix* expressing the cost of each state transition in which input a participates. In the same way, λ and γ can be regarded as Q-indexed vectors. It is therefore no wonder that the work on weighted automata often resorts to matrix terminology and operations such as matrix-matrix multiplication and matrix-vector multiplication. However, *linear algebra* (LA) is seldom assumed explicitly as the *central* notation and calculus — such reasoning takes place episodically, where convenient, conventional set theory doing the main job. This means that the main advantage of LA — the conciseness of blocked, index-free notation and its powerful algebra — is (partially) lost. There are, however, approaches in which LA is the main notational device, see eg. references [9, 28] which follow the tradition of Bloom et al. [7]. But such notation is *untyped* and therefore hard to combine with that of the relations, predicates and functions which are around.

Typed versus untyped mathematics. What does *(un)typed* mean in the previous sentence? It is a commonplace in mathematics to regard functions as special cases of relations (the deterministic, total ones) and relations as special cases of matrices (the Boolean ones, provided addition is trimmed to 1). Yet the three classes of object are treated in disparate ways, unrelatedly and with incompatible (if not contradictory) notation.

For instance, one writes $y = f(x)$ to define a function and $(x, y) \in \mathit{Graph}(f)$ — note how x and y swap position — to express the input/output pairs of the graph of function f, which is a relation. As far as typing is concerned, most people accept notation $f : A \to B$ for defining the signature of a function (as we have seen above) but only reluctantly will accept the same notation $R : A \to B$ to define the *type* of relation R, writing $R \subseteq A \times B$ instead. As far as matrices are concerned, writing $M : m \to n$ to declare the type of a matrix with m columns and n rows will look surprising — textbooks simply tell that M is of order $m \times n$ (or is it $n \times m$?), with loose typing rules. As for type checking, results are stated as *"valid only for matrices of the same order"* [1] and the like. Polymorphic functions are well-accepted. But telling that the identity matrix is as polymorphic as the identity function will sound odd to many people.

Relational mathematics [24] is a step forward towards conceptual unification between relations and matrices. But it is first and foremost *category theory* [20] which provides for successful unification, by regarding functions, relations and matrices as morphisms (arrows) of suitable categories. The category of functions is well known, that of relations less known and those of matrices by and large ignored.

In the sequel we will show how weighted automata can be described and reasoned about in the typed LA which emerges from regarding matrices as morphisms (rather than objects) of suitable categories, as pioneered by MacLane [20] and MacLane and Birkhoff [21]. This is part of a research line which started in [16] and whose aim is to provide evidence of the usefulness of changing notation (and reasoning style) and adopting *typed* LA as the lingua franca of quantitative methods in computer science.

2 Typed linear algebra

Computer scientists tend to regard matrices as rectangular shaped data structures implemented as bidimensional arrays, lists of lists and the like. Mathematicians tend to regard them as linear transforms, i.e. vector-to-vector operations. Yet matrices are abstract entities independent of either such views: they can be regarded as arrows of particular categories, whereby they become *typed*. This answers questions such as: what is the type of a matrix? What are their basic *constructors*? In what measure are these related to standard matrix operations and algebra?

By studying the categories of matrices of [20], the authors of [16] have identified typed, algebraically rich constructors aiming to repair the lack just mentioned. Backhouse [4] regards matrices as a way of compacting sets of equations into single equations which *is a tremendous improvement in concision that does not incur any loss of precision!* Reference [16] furthermore show how the very general concept of a *biproduct* [21] promotes individual values to blocks and value-level operations to block-level operations, in fact the great conceptual advantage offered by matrix notation.

Matrices as arrows. A matrix M with n rows and m columns is a function which tells the value occupying each cell (r, c), for $1 \leq r \leq n$, $1 \leq c \leq m$. The type of such cell-values varies, but the minimal algebraic structure of semirings is required for matrix operations to make sense. Standard linear algebra operates over the richer structure of a *field* (further offering additive and multiplicative inverses) and the field of real numbers ($\mathbb{R}$) is often taken by default.

Interestingly, what is meant by the *type* of a matrix in the sequel does not bear a direct relationship to such algebraic structures: it rather provides (as in programming) a way of interfacing matrices with each other. The type of a matrix M with m columns and n rows will be denoted by the arrow $m \longrightarrow n$ between the number of columns and the number of rows. By writing $m \xrightarrow{M} n$ (or the equivalent $n \xleftarrow{M} m$) one declares matrix M and its type.

The most interesting matrix combinator is *composition*, commonly referred to as *matrix multiplication.* Denoting the (r, c)-th cell of a given matrix M by rMc [2], the (r, c)-th cell of composite matrix $M \cdot N$ is given by

$$r(M \cdot N)c = \langle \sum x :: (rMx) \times (xNc) \rangle \tag{1}$$

where $\times$ is the cell-level semiring multiplicative operation and $\sum$ is the finite iteration of its additive operation.

[2] Rather than the more conventional $M(r, c)$ — we will explain later why we propose a different notation.

What is x in (1) and what is its range? This will be easy to answer by inspecting the types of both M and N:

$$\begin{tikzcd} n & m \arrow[l, "M"'] & k \arrow[l, "N"'] \arrow[ll, bend left, "M\cdot N"] \end{tikzcd} \tag{2}$$

Thus $1 \leq x \leq m$ and matrix multiplication can be abstracted by arrow composition.

For every n there is a matrix of type $n \longleftarrow n$ which is the unit of composition. This is nothing but the *identity matrix* of size n, indistinguishably denoted by $n \xleftarrow{id_n} n$ or $n \xleftarrow{1} n$. This is the diagonal of size n, that is [3], $r(id)c \triangleq r = c$ under the $\{0,1\}$ encoding of the Booleans:

$$id_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \qquad n \xleftarrow{id_n} n$$

Therefore,

$$id_n \cdot M \;=\; M \;=\; M \cdot id_m \qquad \begin{tikzcd} m \arrow[d, "M"'] & m \arrow[l, "id_m"'] \arrow[d, "M"] \arrow[dl, "M"] \\ n & n \arrow[l, "id_n"] \end{tikzcd} \tag{3}$$

where the subscripts m and n can be omitted wherever the underlying type diagrams are assumed.

Equipped with composition (2) and identity (3), matrices form a *category* whose *objects* are matrix dimensions and whose *morphisms* ($m \xleftarrow{M} n$ etc) are the matrices themselves [20, 21]. Strictly speaking, there is one such category per matrix cell-level algebra. Notation $Mat_{\mathbb{S}}$ will be used to denote such a category, parametric on semiring $\mathbb{S}$ or any other (richer) algebraic structure.

Vectors as arrows. Vectors are special cases of matrices in which one of the dimensions is 1, for instance

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} \qquad \text{and} \qquad w = \begin{pmatrix} w_1 \ldots w_n \end{pmatrix}$$

Column vector v is of type $m \longleftarrow 1$ (m rows, one column) and row vector w is of type $1 \longleftarrow n$ (one row, n columns). Our convention is that lowercase letters (eg. v, w) denote vectors and uppercase letters (eg. M, N) denote arbitrary matrices.

[3] Notation $x \triangleq y$ means $x = y$ by definition.

Converse of a matrix. One of the kernel operations of linear algebra is *transposition*, whereby a given matrix changes shape by turning its rows into columns and vice-versa. Given matrix $n \xleftarrow{M} m$, notation $m \xleftarrow{M^\circ} n$ denotes its transpose, or converse. The following idempotence and contravariance laws hold:

$$(M^\circ)^\circ = M \tag{4}$$
$$(M \cdot N)^\circ = N^\circ \cdot M^\circ \tag{5}$$

Bilinearity. Given two matrices of the same type $n \xleftarrow{M,N} m$ it makes sense to add them up index-wise, leading to matrix $M+N$ where symbol $+$ promotes the underlying semiring additive operator to matrix-level. Likewise, additive unit cell value 0 is promoted to matrix 0 wholly filled with 0s, the unit of matrix addition and zero of matrix composition:

$$M + 0 = M = 0 + M \tag{6}$$
$$M \cdot 0 = 0 = 0 \cdot M \tag{7}$$

Composition is bilinear relative to $+$:

$$M \cdot (N + P) = M \cdot N + M \cdot C \tag{8}$$
$$(N + P) \cdot M = N \cdot M + P \cdot M \tag{9}$$

In the same way $M + N$ denotes the promotion of addition of matrix cells to matrix addition, the same promotion can take place with respect to the whole semiring algebra. For instance, cell value multiplication leads to matrix multiplication, denoted $M \times N$ or simply MN (for M and N of the same type), also known as the *Hadamard product*, which is commutative, associative and distributive over addition (ie. bilinear). Clearly,

$$M \times \top = \top \times M = M \tag{10}$$

where matrix $\top$ is of the same type as M and is wholly filled with 1s.

Type generalization. Matrix types (the end points of arrows) can be generalized to arbitrary, denumerable sets since addition in $\mathbb{S}$ is commutative, that is, the summation of (1) can be evaluated in arbitrary order.

In fact, and as is standard in relational mathematics [24], objects in categories of matrices can be generalized from numeric dimensions (n, $m \in \mathbb{N}_0$) to arbitrary denumerable types (A, B), taking disjoint union $A + B$ for $m + n$, Cartesian product $A \times B$ for mn, unit type 1 for number 1, the empty set $\emptyset$ for 0, etc. Conversely, dimension n corresponds to the type made of the initial segment of the natural numbers up to n. Our convention is that lowercase letters (eg. n, m) denote the traditional dimension types (natural numbers), letting uppercase letters denote arbitrary other types.

3 Weighted automata as $\boldsymbol{Mat}_{\mathbb{S}}$ arrows

Following [8], we consider in the sequel a simpler notion of weighted automaton $W = (Q, A; \mu, \gamma)$ which deals without the input weight function λ. This facilitates the comparison between the coalgebraic approach of [8] and our own and helps in staying with the binary matrix block combinators of [16], to be presented shortly. For this purpose, we assign the type $Q \longrightarrow 1$ to output function γ, which is therefore regarded as a row vector in $Mat_{\mathbb{S}}$. Concerning μ, it can either be regarded as a matrix of type $Q \times A \longrightarrow Q$ or of type $Q \longrightarrow Q \times A$, as these types are isomorphic in $Mat_{\mathbb{S}}$ [4]. We prefer the second (coalgebraic) alternative and therefore regard the following diagram as representation of weighted automaton $W = (Q, A; \mu, \gamma)$:

$$Q \times A \xleftarrow{\ \mu\ } Q \xrightarrow{\ \gamma\ } 1 \tag{11}$$

Clearly, both μ and γ can be packaged into a single coalgebra (matrix) of type $(Q \times A) + 1 \xleftarrow{\ W\ } Q$ and made of two blocks

$$W = \left[\frac{\mu}{\gamma}\right] \tag{12}$$

provided we explain what the meaning of combinator $\left[\frac{\ }{\ }\right]$ is. This leads into matrix block notation and its algebra.

Block notation. Two basic binary combinators are available for building matrices out of other matrices, say M and N:

– $[M|N]$ — M and N side by side (read $[M|N]$ as "M *junc* N")
– $\left[\frac{M}{N}\right]$ — M on top of N (read $\left[\frac{M}{N}\right]$ as "M *split* N").

That is, matrices are stacked either vertically ($\left[\frac{M}{N}\right]$) or horizontally ($[M|N]$). Dimensions should agree, as shown in the diagram below, taken from [16], where m, n, p and t are types:

$$\begin{tikzcd}
 & m & \\
n \arrow[ur, "M"] \arrow[r, shift right, "i_1"'] & n+p \arrow[u, "{[M|N]}"] \arrow[l, shift right, "\pi_1"'] \arrow[r, shift left, "\pi_2"] & p \arrow[ul, "N"'] \arrow[l, shift left, "i_2"] \\
 & t \arrow[ul, "P"] \arrow[u, "{\left[\frac{P}{Q}\right]}"] \arrow[ur, "Q"'] &
\end{tikzcd}$$

$$[M|N] = M \cdot \pi_1 + N \cdot \pi_2 \tag{13}$$

$$\left[\frac{P}{Q}\right] = i_1 \cdot P + i_2 \cdot Q \tag{14}$$

[4] This follows from a self-adjunction in $Mat_{\mathbb{S}}$ which is studied in detail in [19]. The isomorphism reshapes matrices by reducing the number of columns by the same factor the number of rows increases, keeping the "rectangular area" and its information intact.

The special matrices i_1, i_2, π_1 and π_2 are fragments of the identity matrix as given by the so-called *reflexion laws*,

$$[i_1|i_2] = id$$
$$\left[\frac{\pi_1}{\pi_2}\right] = id$$

which play an important role in explaining the semantics of the two combinators. In brief, *junc* (13) and *split* (14) form a so-called *biproduct* [20]. The details of this, however, can be skipped for the purposes of this presentation, sufficing to be aware of the rich algebra of such combinators of which we single out two "fusion"-laws,

$$R \cdot [M|N] = [R \cdot M|R \cdot N] \tag{15}$$
$$\left[\frac{M}{N}\right] \cdot R = \left[\frac{M \cdot R}{N \cdot R}\right] \tag{16}$$

two structural equality laws,

$$[A|B] = [C|D] \equiv A = C \wedge B = D \tag{17}$$
$$\left[\frac{A}{B}\right] = \left[\frac{C}{D}\right] \equiv A = C \wedge B = D \tag{18}$$

and two absorption laws:

$$[A|B] \cdot (C \oplus D) = [A \cdot C|B \cdot D] \tag{19}$$
$$(C \oplus D) \cdot \left[\frac{A}{B}\right] = \left[\frac{C \cdot A}{D \cdot B}\right] \tag{20}$$

All these laws emerge as corollaries of the universal properties of biproducts. Mind the types: the laws are only valid for matrices which typecheck and types are obtained by unification, as explained in [16].

Weighted automata as matricial coalgebras. As suggested by (12) above, weighted automaton W can be regarded as a coalgebra for $Mat_{\mathbb{S}}$ endofunctor $\mathsf{F}X = (X \otimes id) \oplus id$, where $\oplus$ and $\otimes$ are the so-called *direct sum* and *Kronecker* bifunctors. The former,

$$M \oplus N = [i_1 \cdot M|i_2 \cdot N]$$

is of type

$$\begin{tikzcd} n \arrow[d, "M"'] & m \arrow[d, "N"'] & n+m \arrow[d, "M \oplus N"] \\ k & j & k+j \end{tikzcd}$$

and the latter is of type

$$\begin{tikzcd} n \arrow[d, "M"'] & m \arrow[d, "N"'] & n \times m \arrow[d, "M\otimes N"] \\ k & j & k \times j \end{tikzcd}$$

Fusion laws

$$[M|N] \otimes C = [M \otimes C|N \otimes C]$$
$$\begin{bmatrix} M \\ \hline N \end{bmatrix} \otimes C = \begin{bmatrix} M \otimes C \\ \hline N \otimes C \end{bmatrix}$$

capture the meaning of Kronecker product block-wise. Index-wise, one has:

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

4 Weighted automata homomorphisms

A homomorphism between two weighted automata W and W' is a function h making the following $Mat_{\mathbb{S}}$-diagram commute,

$$\begin{tikzcd} \mathsf{F}Q \arrow[d, "\mathsf{F}h"'] & Q \arrow[l, "W"'] \arrow[d, "h"] \\ \mathsf{F}Q' & Q' \arrow[l, "W'"] \end{tikzcd} \tag{21}$$

for $\mathsf{F}X = (X \otimes id) \oplus id$ (F-coalgebra homomorphism). The reader may wonder about how does h (a function) fit into a diagram of matrices. The explanation is easy: every function $A \xrightarrow{f} B$ can be represented in $Mat_{\mathbb{S}}$ by a matrix $[\![f]\!]$ of the same type defined by

$$b[\![f]\!]a \;\triangleq\; (b =_{\mathbb{S}} f\ a)$$

where, in general, $y =_{\mathbb{S}} x$ is the unit 1 of $\mathbb{S}$ if $y = x$ and 0 otherwise. Thus $[\![f]\!]$ is the matrix which represents the graph of f: there is a 1 in every entry of $[\![f]\!]$ addressed by $(f(a), a)$ and 0s everywhere else. As $\mathbb{S}$ is always implicit and all diagrams are drawn in $Mat_{\mathbb{S}}$ unless otherwise specified, subscript $\mathbb{S}$ in $=_{\mathbb{S}}$ and the parentheses in $[\![f]\!]$ can be safely dropped.

Below we show how diagram (21) unfolds into the usual definition of weighted automata homomorphism [8], which is termed *functional* simulation in [9]. For this we will rely on typed, blocked linear algebra:

$$\begin{array}{rl} & (\mathsf{F}h) \cdot W \;=\; W' \cdot h \\ \equiv & \quad \{ \text{ unfold } \mathsf{F}h \text{ ; } W \text{ and } W' \text{ are splits defined by (12) } \} \\ & ((h \otimes id) \oplus id) \cdot \begin{bmatrix} \mu \\ \hline \gamma \end{bmatrix} \;=\; \begin{bmatrix} \mu' \\ \hline \gamma' \end{bmatrix} \cdot h \end{array}$$

$$\equiv \qquad \{ \text{ absorption (20), identity (3) and fusion (16) } \}$$

$$\left[\frac{(h \otimes id) \cdot \mu}{\gamma}\right] = \left[\frac{\mu' \cdot h}{\gamma' \cdot h}\right]$$

$$\equiv \qquad \{ \text{ equality (18) } \}$$

$$\begin{cases} (h \otimes id) \cdot \mu = \mu' \cdot h \\ \gamma = \gamma' \cdot h \end{cases} \tag{22}$$

The reader wishing to convert the equalities of (22) into index-wise formulas for cross-checking with other sources is invited to do so based on the following rules interfacing index-free and index-wise matrix notation, where N is an arbitrary matrix and f, g are functional matrices:

$$y(f \cdot N)x = \langle \sum z : y = f(z) : zNx \rangle \tag{23}$$

$$y(g^\circ \cdot N \cdot f)x = (g(y))N(f(x)) \tag{24}$$

These rules are expressed in the style of the Eindhoven quantifier calculus [3]. Their calculation (deferred to the appendix) provides evidence of the safe mix among matrix, predicate and function notation in typed LA.

We start by unfolding the first equality of (22):

$$(h \otimes id) \cdot \mu = \mu' \cdot h$$

$$\equiv \qquad \{ \text{ index-wise equality on matrices of type } Q' \times A \longleftarrow Q \ \}$$

$$(q', a)((h \otimes id) \cdot \mu)q = (q', a)(\mu' \cdot h)q$$

$$\equiv \qquad \{ \text{ (24) on the right hand side, for } g, N, f := id, \mu', h \ \}$$

$$(q', a)((h \otimes id) \cdot \mu)q = (q', a)\mu'(h(q))$$

$$\equiv \qquad \{ \text{ (23) for } f, N := h \otimes id, \mu \ \}$$

$$\langle \sum (p, b) : (q', a) = (h \otimes id)(p, b) : (p, b)\mu q \rangle = (q', a)\mu'(h(q))$$

$$\equiv \qquad \{ \text{ since } (h \otimes id)(p, b) = (h(p), b)\text{; one-point rule [3] over } a = b \ \}$$

$$\langle \sum p : q' = h(p) : (p, a)\mu q \rangle = (q', a)\mu'(h(q))$$

$$\equiv \qquad \{ \text{ liberally writing } p \xleftarrow{a} q \text{ for the weight of the corresponding transition } \}$$

$$\langle \sum p : q' = h(p) : \ p \xleftarrow{a} q \ \rangle = \ q' \xleftarrow{a} h(q)$$

In words: the weight associated to transition $q' \xleftarrow{a} h(q)$ in the target automaton is the accumulation of the weights of all transitions $p \xleftarrow{a} q$ in the source automaton for all p which h maps to q'.

Unfolding the other matrix equality in (22) is simpler: as γ, γ' are row vectors, we get, for all $q \in Q$, $1\gamma q = 1(\gamma' \cdot h)q$, since there is only one row. By (24) this becomes $1\gamma q = 1\gamma'(h(q))$, that is $\gamma(q) = \gamma'(h(q))$ once γ, γ' are regarded back as functions.

Summing up, both calculations show that weighted automata homomorphisms defined in a category of matrices coincide with those defined by Bonchi et al. [8] in the category of sets. We regard this as just the beginning of a typed LA approach to weighted automata to be developed comprehensively in the near future.

5 Summary

This abstract addresses on-going work. Since the research presented in [16, 19], typed LA calculational techniques have been successfully applied to data mining [17] and probabilistic program calculation [23], the latter extending the algebra of programming of Bird and de Moor [6].

In the case of weighted automata, LA is a natural choice already identified by other researchers. Buchholz [9], for instance, praises matrix notation because it *allows an elegant and compact formulation of the theory.* Trčka [28] writes that *matrices (...) increase clarity and compactness, simplify proofs, make known results from linear algebra directly applicable* and also mentions their *didactic advantage.*

In broad terms, the approach put forward in this abstract proposes that LA be *typed* on the basis of a categorial approach in which index-free matrix terms form the main notation, diagrammatic representations and proofs included. That is to say, rather than accepting LA arguments embedded in ordinary set-theoretical reasoning, we propose that typed LA be regarded as a *lingua franca* for computing, the other approaches coming as suitable instantiations [5].

We should say we are not the first proposing this strategy. The acronym LAoP, for "linear algebra of programming" has been put forward already, albeit in a somewhat different setting, by Sernadas et al. [25], the key idea being *"to adopt linear algebra as the lingua franca of software verification"* [27]. Our contribution is the emphasis on LA polymorphic *types.* For this to work in practice, we believe the interfaces with standard logic, set theory and relation algebra should not be neglected. Schmidt [24] already relies on matrix notation for doing relation algebra. Our experiments eg. with the Eindhoven quantifier notation show that the interface between functions, relations, predicates and matrices is (at least pedagogically) relevant. The infix notation we adopt for matrix entries — yMx rather than $M(y, x)$ — intends to bridge with that commonly used for binary relations. For instance, $y \leq x$ is preferred to $\leq (y, x)$.

[5] Even so general a framework as that of an *allegory* [12] arises from matrices whose data values form *locales.*

6 Current and related work

One of our targets is the *linear algebra of components* which, anticipated in [18], promises a quantitative expansion of the coalgebraic approach of Barbosa [5] on software components.

The work by Bonchi et al. [8] on a coalgebraic perspective on weighted automata promises a similar outcome but their use of linear algebra is on a different plan: triggered by the need to extend the powerset functor quantitatively, they introduce a *vector space* which weights (*quantifies*) multi-way state evolution. (In a sense, powersets become "metric".) Because this is carried in the category of sets, their coalgebras involve functor $\mathsf{W} = \mathbb{K} \times (\mathbb{K}^{-}_{\omega})^{A}$ over a field $\mathbb{K}$, where $\mathbb{K}^{-}_{\omega}$ is the so-called field valuation (exponential) functor. Our approach flattens such exponentials by changing category: the category of sets and functions gives room to the category of matrices built on top of $\mathbb{K}$. Thus $(\_)^{A}$ within sets becomes $(\_) \times A$ within matrices. In this way, weights no longer need to be taken explicitly into account, as the underlying matrix algebra circumspectly takes care of them.

Much remains to be done, in particular calling for the unification with related work. For instance, we would like to relate our ideas with those of Trčka [28], who presents a matrix approach to the notions of strong, weak and branching bisimulation ranging from labeled transition systems to Markov reward chains. This already is the aim of Buchholz [9], who targets at a *universal definition of bisimulation which can be applied to a wide class of model types such that the different forms of bisimulation can all be seen as specific cases*, helping to unify system analysis.

We believe matrix types will improve the approaches of both [9] and [28] in a significant way. But, above all, in its use of matrix categories our strategy is close to the iteration theory $Mat_{\mathcal{L}(X^*)}$ of Bloom et al. [7] whose morphisms are matrices with entries in the semiring of languages. We intend to investigate the relationship between both approaches in a thorough way.

Acknowledgements

The author is indebted to Nelma Moreira for her comments on an earlier draft of this extended abstract. This research was carried out in the context of the QAIS (Quantitative analysis of interacting systems: foundations and algorithms) project funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT (Foundation for Science and Technology) contract PTDC/EIA-CCO/122240/2010.

Appendix

To calculate (23) we let $M := f$ in (1):

$$\begin{array}{ll} & y(f \cdot N)x \\ = & \quad \{ \text{ definition (1) } \} \end{array}$$

$$
\begin{array}{cl}
& \langle \sum z :: (y = f(z)) \times (zNx) \rangle \\
= & \quad \{ \text{ rule (25) below } \} \\
& \langle \sum z : y = f(z) : zNx \rangle
\end{array}
$$

The rule used above,

$$\langle \sum x : p(x) : e(x) \rangle = \langle \sum x :: (p(x)) \times (e(x)) \rangle \tag{25}$$

is illustrative of the interface between predicate logic and the semiring algebra underneath: on the left hand side, $p(x)$ is a predicate expressing the range of a summation; on the right hand side it is encoded into $\mathbb{S}$: 1 if $p(x)$ holds, 0 otherwise. Since $0 \times s = 0$, all terms such that $p(x)$ doesn't hold boil down to 0 and don't affect the summation [6].

Similarly, for $M := g^\circ$ in (1):

$$
\begin{array}{cl}
& y(g^\circ \cdot N)x \\
= & \quad \{ \text{ definition (1) ; } y(g^\circ)z \ = \ z =_{\mathbb{S}} g(y) \ \} \\
& \langle \sum z :: (z = g(y)) \times (zNx) \rangle \\
= & \quad \{ \text{ rule (25) } \} \\
& \langle \sum z : z = g(y) : zNx \rangle \\
= & \quad \{ \text{ one-point rule [3] } \} \\
& (g(y))Nx
\end{array}
$$

Thus $y(g^\circ \cdot N)x = (g(y))Nx$. The calculation of $y(N \cdot f)x = yN(f(x))$ follows the same steps. Rule (24) puts these two equalities together.

[6] For $\mathbb{S}$ the Boolean semiring, $\sum$ is existential quantification, $\times$ is conjunction and equality (25) becomes an instance of the *trading rule* of existential quantification [3].

References

[1] Abadir, K., Magnus, J.: Matrix algebra. Econometric exercises 1. Cambridge University Press (2005)

[2] Andova, S., McIver, A., D'Argenio, P.R., Cuijpers, P.J.L., Markovski, J., Morgan, C., Núñez, M. (eds.): Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications, EPTCS, vol. 13 (2009)

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

[4] Backhouse, R.: Mathematics of Program Construction. Univ. of Nottingham (2004), draft of book in preparation. 608 pages

[5] Barbosa, L.: Towards a Calculus of State-based Software Components. Journal of Universal Computer Science 9(8), 891–909 (August 2003)

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

[7] Bloom, S., Sabadini, N., Walters, R.: Matrices, machines and behaviors. Applied Categorical Structures 4(4), 343–360 (1996)

[8] Bonchi, F., Bonsangue, M., Boreale, M., Rutten, J., Silva, A.: A coalgebraic perspective on linear weighted automata. Information and Computation 211, 77–105 (2012)

[9] Buchholz, P.: Bisimulation relations for weighted automata. Theoretical Computer Science 393(1-3), 109–123 (2008)

[10] Droste, M., Gastin, P.: Weighted automata and weighted logics. In: Kuich, W., Vogler, H., Droste, M. (eds.) Handbook of Weighted Automata, chap. 5, pp. 175–211. EATCS Monographs in Theoretical Computer Science, Springer (2009)

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

[12] Freyd, P., Scedrov, A.: Categories, Allegories, Mathematical Library, vol. 39. North-Holland (1990)

[13] Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming. pp. 2–14. ICFP'11, ACM, New York, NY, USA (2011)

[14] Hehner, E.: A probability perspective. Formal Aspects of Computing 23, 391–419 (2011)

[15] Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94(1), 1–28 (1991)

[16] Macedo, H., Oliveira, J.: Matrices As Arrows! A Biproduct Approach to Typed Linear Algebra. In: MPC'10. LNCS, vol. 6120, pp. 271–287. Springer (2010)

[17] Macedo, H., Oliveira, J.: Do the middle letters of "OLAP" stand for linear algebra ("LA")? Technical Report TR-HASLab:04:2011, INESC TEC and University of Minho, Gualtar Campus, Braga (2011)

[18] Macedo, H., Oliveira, J.: Towards linear algebras of components. In: FACS 2010. LNCS, vol. 6921, pp. 300–303. Springer (2011)

[19] Macedo, H., Oliveira, J.: Typing linear algebra: A biproduct-oriented approach (2011). Accepted for publication in SCP

[20] MacLane, S.: Categories for the Working Mathematician. Springer-Verlag, New-York (1971)

[21] MacLane, S., Birkhoff, G.: Algebra. AMS Chelsea (1999)

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

[23] Oliveira, J.: Towards a linear algebra of programming. Accepted for publication in Formal Aspects of Computing (2012)

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

[25] Sernadas, A., Ramos, J., Mateus, P.: Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. Tech. rep., SQIG - IT and IST - TU Lisbon, 1049-001 Lisboa, Portugal (2008), short paper presented at LPAR 2008, Doha, Qatar. November 22-27

[26] Sokolova, A.: Coalgebraic Analysis of Probabilistic Systems. Ph.D. dissertation, Tech. Univ. Eindhoven, Eindhoven, The Netherlands (2005)

[27] SQIG-Group: LAP: Linear algebra of bounded resources programs (2011), `http://sqig.math.ist.utl.pt/work/LAP`, iT & Tech. Univ. Lisbon

[28] Trčka, N.: Strong, weak and branching bisimulation for transition systems and Markov reward chains: A unifying matrix approach. In: [2], pp. 55–65