A study of risk-aware program transformation

Daniel Murta[a,1], José Nuno Oliveira[a,*]

[a]HASLAB - High Assurance Software Laboratory
INESC TEC / Univ. Minho, Braga, Portugal

Abstract

In the trend towards tolerating hardware unreliability, *accuracy* is exchanged for *cost savings*. Running on less reliable machines, functionally correct code becomes risky and one needs to know how risk propagates so as to mitigate it. Risk estimation, however, seems to live outside the average programmer's technical competence and core practice.

In this paper we propose that program design by source-to-source transformation be *risk-aware* in the sense of making probabilistic faults visible and supporting equational reasoning on the probabilistic behaviour of programs caused by faults. This reasoning is carried out in a linear algebra extension to the standard, *à la* Bird-Moor algebra of programming.

This paper studies, in particular, the propagation of faults across standard program transformation techniques known as *tupling* and *fusion*, enabling the *fault of the whole* to be expressed in terms of the *faults of its parts*.

1. Introduction

With software as invasive in everyday life as it is today, one need not be on the staff of a space agency to ask the question: *what risks do we run day-to-day by relying on so much software?* Jackson (2009) writes:

> *(...) a dependable system is one (..) in which you can place your reliance or trust. A rational person or organization only does this with evidence that the system's benefits outweigh its risks.*

Over the years, NASA has defined a *probabilistic risk assessment* (PRA) methodology to enhance the safety decision process. Quoting (Stamatelatos and Dezfuli, 2011):

> *PRA characterizes risk in terms of three basic questions: (1) What can* go wrong*? (2) How* likely *is it? and (3) What are the* consequences*? The PRA process answers these questions by systematically (...) identifying,*

*Corresponding author.

[1]Partially supported by *Fundação para a Ciência e a Tecnologia*, Portugal, under grant number BI1-2012_PTDC/EIA-CCO/122240/2010_UMINHO.

> *modeling, and* quantifying *scenarios that can lead to undesired consequences.*

This may leave one with the feeling that PRA takes place *a posteriori*, that is, once the system is built. Even if this is not so in general, limitations of current programming practice are apparent concerning timely assessment of the risks involved in the future use of computer programs. *Things that can go wrong* can be guessed; but, how is the *likelihood* of such bad behaviour expressed? and how does one quantify its *consequences* (fault propagation)?

This paper addresses these questions and issues in the context of *functional programming* (FP) over *unreliable* hardware. Note that such unreliability can be intentional, as is the case in *inexact circuit design* (Lingamneni et al., 2013), where accuracy of the circuit is exchanged for cost savings (e.g. energy, delay, silicon).

We will show that FP is well prepared for smoothly incorporating risk analysis in the design of programs. This is because the standard *qualitative* semantics of FPs can evolve towards a *quantitative* one simply by upgrading its underlying *relational* algebra of programs *à la* Bird-Moor (1997) into a *linear* algebra of programming (Oliveira, 2012).

The need for quantitative rather than qualitative semantics is nicely explained in the following excerpt of the preface of (Andova et al., 2009):

> *Quantitative Formal Methods deal with systems whose behaviour of interest is more than the traditional Boolean "correct" or "incorrect" judgment. (...) Today there are many quantitative aspects of system design: they include timing (whether discrete, continuous or hybrid); probabilistic aspects of success or failure including cost and reward; and quantified information flow.*

The basic idea of the current paper is simple: suppose one writes function $good$ for the intended behaviour of a program and there is evidence that, with probability p, such behaviour can turn into a bad function. Using the *probabilistic choice* combinator $(\cdot\ {}_{\cdot}\!\diamond\ \cdot)$ of (McIver and Morgan, 2005; Oliveira, 2012), one may write term

$$bad\ {}_{p}\!\diamond\ good \tag{1}$$

to express the complete (ie. with risk incorporated) behaviour of what one is programming.

What is needed, then, is a method for evaluating the propagation of risk, for instance across recursion schemes. This is what the *linear* algebra of programming (LAoP) is intended for. This paper investigates, in particular, the quantitative extension of the so-called *mutual recursion* and *banana-split* laws (Bird and de Moor, 1997) which underpin the refinement of primitive recursive functions into linear implementations and checks under what conditions such implementations are as good as their original definitions with respect to fault propagation.

The approach will be illustrated in two ways: either by running programs as probabilistic (monadic) functions written in Haskell (Jones, 2003) using the PFP library of

Erwig and Kollmansberger (2006), or by running finite approximations of them directly as matrices in MATLAB [2].

Contribution. In the trend towards tolerating hardware unreliability, software is doomed to misbehave in some degree. Are the laws of program transformation still valid in this setting?

- This paper shows how the standard *algebra of programming* (AoP) dear to the so-called *program transformation* school of software design extends and incorporates risk simply by switching from standard (*"sharp"*) functions to *probabilistic* functions handled as matrices in *linear algebra*.[3]
- The laws of such a linear algebra of programming (LAoP) are shown to capture the notion of probabilistic indistinguishability, essential to decide whether *program transformation* rules can be safely applied or not.
- The approach is shown to be readily applicable to *recursive programs* which handle possibly interfering threads of computation.
- In particular, mutually recursive computations are addressed showing under what conditions mutual recursion slicing holds in the probabilistic setting.
- Finally, the paper shows that a well-known *tupling* technique known as the *"banana-split"* program transformation is still valid in presence of faults.

Paper outline. The following section presents two motivating programs which will be subject to fault-injection as an illustration of risk simulation and calculation. Section 3 addresses the derivation of such programs via mutual-recursion transformation, an exercise which is extended in section 4 to the probabilistic setting.[4] A basis for the probabilistic setting is given in section 5, where the LAoP is put in context, leading to the study of probabilistic mutual recursion given in section 6. This in turn leads to an asymmetry (section 7) which explains the different fault propagation patterns found in the two motivating examples (section 8). The topic of fault propagation in functional programming is further analysed in section 9 by moving to more elaborate data types and showing how the *risk of the whole* can be calculated combining the *risk of the parts*. The two last sections conclude, review related work and give prospects for future work. Proofs of auxiliary results are deferred to Appendix A.

2. Motivation

Let us start from two programs written in C. One supposedly computes the square of a non-negative integer n by adding up the n-first odd numbers:

---

[2] MATLAB ™ is a trademark of The MathWorks ®.

[3] This extends to deterministic imperative programs via probabilistic functional semantics denotation.

[4] Readers more interested in the calculational (probabilistic) theory presented in this paper and not so much in the Haskell/C examples given as illustration may wish to skip these sections and go straight to section 5.

```
int sq(int n) {
  int s=0; int o=1;
  int i;
  for (i=1;i<n+1;i++) {s+=o; o+=2;}
  return s;
};
```

The other supposedly computes the n-th entry in the Fibonacci series, for n positive:

```
int fib(int n) {
  int x=0; int y=1; int i;
  for (i=1;i<=n;i++) {int a=y; y=y+x; x=a;}
  return x;
};
```

Both programs are for-loops whose bodies rely on the same operation: addition of natural numbers. Suppose one knows that, in the machine where such programs will run, there is the risk of addition misbehaving in some known way: with probability p, $x + y$ may evaluate to y, in which case $(x+) = id$, the identity function. Or one might know that, in some unfriendly environment, the processor's arithmetic-logic unit may reset addition output to 0, with probability q.

The question is: what is the impact of such faults in the overall behaviour of each for-loop? Can we *measure* such an impact? Can we *predict* it? Are there versions of the same programs which mitigate such faults better than the ones given?

The standard approach to these questions relies on simulation: one performs a large number of experiments in which the programs run with the given *faults injected* according to the given probabilities and then performs statistic analysis of the outcome of such simulations. Software *fault injection* (Voas and McGraw, 1997) is a more and more widespread technique for quality assurance which measures the propagation of faults through paths that might otherwise rarely be followed in testing. The G-SWFIT technique, for instance, emulates the software fault classes most frequently observed in the field through a library of fault emulation operators, and injects such faults directly in the target executable code (Durães and Madeira, 2006).

In this paper we adopt a different strategy: instead of simulating risky behaviour *a posteriori*, this is taken into account *a priori* by moving from imperative to functional code whereby faulty behaviour is encoded in terms of probabilistic functions (Erwig and Kollmansberger, 2006). Take the two versions of faulty addition given above as examples: the first can be expressed by turning $(+)$ into the probabilistic function

$$fadd_p\ x = id\ {}_p\!\diamond\ (x+) \tag{2}$$

($fadd$ for "faulty addition") which misbehaves as the identity function id with probability p and exhibits the correct behaviour with probability $1 - p$; similarly, the second version is expressed by probabilistic choice

$$fadd_q\ x = \underline{0}\ {}_q\!\diamond\ (x+)$$

where $\underline{0}\ \_ = 0$ is the everywhere-0 constant function. Of course, we might think of more elaborate fault patterns, for instance

$$fadd_{p,q}\ x = (\underline{0}\ _q\diamond\ id)\ _p\diamond\ (x+)$$

in which the probability of $fadd$ resetting to 0 is qp and $(1-q)\ p$ is that of degenerating into the identity; or even thinking of normal distributions centered upon the expected output $x + y$, and so on.

Probabilistic functions are distribution-valued functions which can be written in the monadic style over the *distribution monad*. This is termed $Dist$ in the PFP library written by Erwig and Kollmansberger (2006), which we shall be using in the sequel.[5] Moreover, probabilistic functions can be reasoned about using the laws of monads, explicitly as advocated by Gibbons and Hinze (2011) or implicitly as in the probabilistic notation proposed by Morgan (2012) as extension to the standard Eindhoven quantifier calculus (Backhouse and Michaelis, 2006).

There is yet another alternative: every probabilistic function $f : A \to Dist\ B$ is in one-to-one correspondence with a matrix whose columns are indexed by A, whose rows are indexed by B and whose multiplication corresponds to composition in the Kleisli category induced by $Dist$ (Oliveira, 2012, 2013). This offers the possibility of using the rich field of *linear algebra* to calculate with probabilistic functions, in the same way relation algebra is advocated by Bird and de Moor (1997) for reasoning about standard (*sharp*) functions.

One of the advantages of such a *linear* algebra of programming (LAoP) is the way recursive probabilistic functions are handled: simply by using the same combinators (e.g. maps, folds) of the standard algebra of programming (Bird and de Moor, 1997). The shift from a qualitative to a quantitative semantics is therefore rather smooth — the game is the same, the move ensured just by change of underlying category. Following this approach, Oliveira (2012) already gives an example of what might be referred to as *fault-fusion*: the risk of the whole misbehaving can be expressed in terms of the risk of the parts misbehaving wherever a particular fusion law is applicable.

Note, however, that not every law of the algebra of programming extends quantitatively. In this paper we address the linear algebra extension of one such law which is particularly relevant to program calculation: the *mutual recursion* law enabling systems of mutually recursive functions to be merged into a single, more efficient function (Bird and de Moor, 1997). Both C programs given above can be derived from their specifications using such a law. Below we show how they can be turned into probabilistic functions expressing safe and risky behaviour in a natural and calculational way.

3. Mutual recursion

Let us write the standard definition of the Fibonacci function in Haskell syntax:

$$\begin{array}{l} fib\ 0 = 0 \\ fib\ 1 = 1 \\ fib\ (n+2) = fib\ n + fib\ (n+1) \end{array}$$

[5] All distributions in our approach are generated by finite application of the $choice$ operator (1) and therefore have finite support.

The linear version encoded in the C program given above is obtained by pairing *fib* with its *derivative*, $f\ n = \mathit{fib}\ (n+1)$: [6]

$$\begin{aligned}
&\mathit{fib}\ 0 = 0\\
&\mathit{fib}\ (n+1) = f\ n\\
&f\ 0 = 1\\
&f\ (n+1) = \mathit{fib}\ n + f\ n
\end{aligned}$$

The pairing of the two functions,

$$(\mathit{fib} \vartriangle f)\ n = (\mathit{fib}\ n, f\ n)$$

can be expressed primitive-recursively by

$$\begin{aligned}
&(\mathit{fib} \vartriangle f)\ 0 = (\mathit{fib}\ 0, f\ 0) = (0,1)\\
&(\mathit{fib} \vartriangle f)\ (n+1) = (f\ n, \mathit{fib}\ n + f\ n)
\end{aligned}$$

or by the equivalent

$$\begin{aligned}
&(\mathit{fib} \vartriangle f)\ 0 = (0,1)\\
&(\mathit{fib} \vartriangle f)\ (n+1) = (y, x+y)\ \mathbf{where}\ (x,y) = (\mathit{fib} \vartriangle f)\ n
\end{aligned}$$

itself the same as

$$\begin{aligned}
&(\mathit{fib} \vartriangle f) = \mathsf{for}\ \mathit{loop}\ (0,1)\\
&\quad \mathbf{where}\ \mathit{loop}\ (x,y) = (y, x+y)
\end{aligned}$$

by introduction of the $\mathsf{for}\ \mathit{loop}$ combinator,

$$\begin{aligned}
&\mathsf{for}\ b\ i\ 0 = i\\
&\mathsf{for}\ b\ i\ (n+1) = b\ (\mathsf{for}\ b\ i\ n)
\end{aligned}$$

where b is the loop body and i provides for initialization. This is the natural-number equivalent to combinator foldr over finite lists in Haskell, ie. the *catamorphism* (Bird and de Moor, 1997) *of the natural numbers*. Therefore, we can define

$$\begin{aligned}
&\mathit{fibl}\ n =\\
&\quad \mathbf{let}\ (x,y) = \mathsf{for}\ \mathit{loop}\ (0,1)\ n\\
&\qquad \mathit{loop}\ (x,y) = (y, x+y)\\
&\quad \mathbf{in}\ x
\end{aligned}$$

as the linear version of *fib* obtained by pairing *fib* with its derivative — compare with the C program given above.

The other program computing squares can be derived in the same way from the specification $sq\ n = n^2$: the two mutually recursive functions

[6] Since $f\ 0 = \mathit{fib}\ 1 = 1$ and $f\ (n+1) = \mathit{fib}\ (n+2) = \mathit{fib}\ n + \mathit{fib}\ (n+1) = \mathit{fib}\ n + f\ n$.

$$sq\ 0 = 0$$
$$sq\ (n+1) = sq\ n + odd\ n$$
$$odd\ 0 = 1$$
$$odd\ (n+1) = 2 + odd\ n$$

arise from the binomial $(n+1)^2 = n^2 + 2n + 1$ and introduction of function $odd\ n = 2\ n + 1$, thus named because $2\ n + 1$ is the n-th odd number. (That is, the square of a natural number always is a sum of consecutive odd numbers.) Pairing them up into $(sq \vartriangle odd)\ x = (sq\ x, odd\ x)$ and proceeding in the same way as above we obtain $(sq \vartriangle odd) = \mathsf{for}\ loop\ (0,1)$ where $loop\ (s,o) = (s+o, o+2)$ and thereupon the following functional version of the given C program: [7]

$$\begin{array}{l} sql\ n = \\ \quad \mathbf{let}\ (s,o) = \mathsf{for}\ loop\ (0,1)\ n \\ \qquad loop\ (s,o) = (s+o, o+2) \\ \quad \mathbf{in}\ s \end{array}$$

Clearly, each recursive function above and its linear version are, extensionally, the same function. Let us now see what happens once we start injecting risky (faulty) behaviour in each of them.

4. Going probabilistic

Probabilistic extensions of any of the functions above can be obtained by writing them monadically and then instantiating them with the distribution monad (Erwig and Kollmansberger, 2006). (Readers less conversant with monadic programming may find the short note on program "monadification" given in appendix Appendix B useful at this point.) Take the recursive version of *fib* given in the beginning of section 3 and "monadify it" into:

$$\begin{array}{l} mfib\ 0 = return\ 0 \\ mfib\ 1 = return\ 1 \\ mfib\ (n+2) = \\ \quad \mathbf{do}\ \{x \leftarrow mfib\ n; y \leftarrow mfib\ (n+1); return\ (x+y)\} \end{array}$$

By running $mfib\ n$ inside the $Dist$ monad one gets $fib\ n$ with 100% probability, since $return$ yields the *one-point*, Dirac distribution of its argument.

Now let us inject one of the faults mentioned in section 2, say $fadd_p\ x = id_p \diamond (x+)$ with $p = 0.1$, for instance. For this we just replace $return\ (x+y)$ (perfect addition) by $fadd_{0.1}\ x\ y$ and run test cases, e.g. [8]

[7] Notice how the syntax `s+=o; o+=2;` in C nicely tallies with $(s+o, o+2)$ in Haskell.

[8] The probabilities in this example and others to follow are chosen with no criterion at all apart from leading to distributions visible to the naked eye. By all means, 0.1 would be extremely high risk in realistic PRA (Stamatelatos and Dezfuli, 2011), where only figures as small as 1.0E-7 are "acceptable" risks.

```
Main> mfib 4
3  81.0%
2  18.0%
1   1.0%
```

We see that the correct behaviour (100% chance of getting $\mathit{fib}\ 4 = 3$) is no longer ensured — with chance 18% one may get 2 as result and even 1 is a possible output, with probability 1%.

Similar experiments can be carried out with the linear version by defining its monadic evolution

$$\begin{array}{l} \mathit{mfibl}\ n = \\ \quad \mathbf{do}\ \{(x, y) \leftarrow \mathsf{mfor}\ \mathit{loop}\ (0, 1)\ n; \mathit{return}\ x\} \\ \quad \mathbf{where}\ \mathit{loop}\ (x, y) = \mathit{return}\ (y, x + y) \end{array}$$

relying on the monadic extension of the for combinator:

$$\begin{array}{l} \mathsf{mfor}\ b\ i\ 0 = \mathit{return}\ i \\ \mathsf{mfor}\ b\ i\ (n + 1) = \mathbf{do}\ \{x \leftarrow \mathsf{mfor}\ b\ i\ n; b\ x\} \end{array}$$

To inject into *mfibl* the same fault injected before into *mfib* amounts to replacing, in the loop body, *good* addition $(x + y)$ by the *bad* one $(\mathit{fadd}_{0.1}\ x\ y)$:

$$\mathit{loop}\ (x, y) = \mathbf{do}\ \{z \leftarrow \mathit{fadd}_{0.1}\ x\ y; \mathit{return}\ (y, z)\}$$

Running the same experiment as above we still get $\mathit{mfibl}\ 4 = \mathit{mfib}\ 4$. However, behavioural equality between the two (one recursive, the other linear) fault-injected versions of *fib* is no longer true for arguments $n > 4$, see for instance:

n	$\mathit{mfib}\ n$		$\mathit{mfibl}\ n$	
5	5	65.6%	5	72.9%
	4	21.9%	3	16.2%
	3	10.5%	4	8.1%
	2	1.9%	2	2.7%
	1	0.1%	1	0.1%
6	8	47.8%	8	65.6%
	7	26.6%	6	14.6%
	6	11.8%	5	14.6%
	5	9.8%	3	2.4%
	4	2.7%	4	2.4%
	3	1.1%	2	0.4%
	2	0.2%	1	0.0%
	1	0.0%		

Note how the linear version performs better than the recursive one in the sense of hitting the correct answer with higher probability.[9]

[9]Intuitively, this is to be expected, since the linear version performs the faulty operation less often.

Finally, let us now carry out similar experiments concerning the injection of the same fault (in the addition function) in suitably extended (monadic) versions of the square function, the recursive one

$$\begin{array}{l} \mathit{msq}\ 0 = \mathit{return}\ 0 \\ \mathit{msq}\ (n+1) = \mathbf{do}\ \{ m \leftarrow \mathit{msq}\ n; \mathit{fadd}_{0.1}\ m\ (\mathit{odd}\ n) \} \end{array}$$

and the linear one:

$$\begin{array}{l} \mathit{msql}\ n = \mathbf{do}\ \{ (s,o) \leftarrow \mathsf{mfor}\ \mathit{loop}\ (0,1)\ n; \mathit{return}\ s \}\ \mathbf{where} \\ \quad \mathit{loop}\ (s,o) = \mathbf{do}\ \{ z \leftarrow \mathit{fadd}_{0.1}\ s\ o; \mathit{return}\ (z, o+2) \} \end{array}$$

In this case — as much as we can test — both versions exhibit the same behaviour, that is, they are probabilistically indistinguishable, see for instance:

n	$\mathit{msq}\ n$	$\mathit{msql}\ n$
0	`0 100.0%`	`0 100.0%`
1	`1 100.0%`	`1 100.0%`
2	`4 90.0%`	`4 90.0%`
	`3 10.0%`	`3 10.0%`
3	`9 81.0%`	`9 81.0%`
	`5 10.0%`	`5 10.0%`
	`8 9.0%`	`8 9.0%`
$\vdots$	$\vdots$	$\vdots$
6	`36 59.0%`	`36 59.0%`
	`11 10.0%`	`11 10.0%`
	`20 9.0%`	`20 9.0%`
	`27 8.1%`	`27 8.1%`
	`32 7.3%`	`32 7.3%`
	`35 6.6%`	`35 6.6%`
$\vdots$	$\vdots$	$\vdots$

Summing up, we are in presence of two examples in which the risk of bad behaviour propagates differently across the mutual recursion (tupling) program transformation.

In the remainder of this paper we will resort to linear algebra to explain this discrepancy. We will show that, even if the transformation does not hold in general for probabilistic functions, there are side conditions sufficient for it to hold. It turns out that the square example will meet one such side-condition while Fibonacci will not. This will explain the different behaviour witnessed in the examples above.

5. Probabilistic for-loops in the LAoP

Consider the probabilistic Boolean function $f = \underline{\mathit{False}}\ {}_{0.05}\diamond\ (\neg)$ which is such that $f\ \mathit{True} = \mathit{False}$ (100%) and $f\ \mathit{False}$ is either True (95%) or False (5%) — an instance of *faulty negation*. It is easy to represent f in the form of a matrix M,

$$M = \begin{array}{r} \\ \mathit{False} \\ \mathit{True} \end{array} \begin{array}{c} \begin{array}{cc} \mathit{False} & \mathit{True} \end{array} \\ \begin{pmatrix} 0.05 & 1.00 \\ 0.95 & 0.00 \end{pmatrix} \end{array} \tag{3}$$

where the inputs spread across columns and the outputs across rows. Because columns represent distributions, all figures in the same column should sum up to 1.

Matrices with this property will be referred to as *column-stochastic* (CS). The multiplication of two CS-matrices is a CS-matrix, as is the identity matrix id (square, diagonal matrix with 1s in the diagonal) which is the unit of such multiplication: $M.id = M = id \cdot M$, where matrix multiplication is denoted by an infix dot $(\cdot)$.

Note that CS-matrices are *total* in the sense that no column adds to less than 1. Relaxing such a constraint would lead to so called *sub-stochastic* matrices. In the current paper, all matrices are CS, that is, they are total in the above sense.

We will write $M : n \to m$, or draw the arrow $n \xrightarrow{M} m$, to indicate the *type* of a CS-matrix M, meaning that it has n columns and m rows. This view enables us to regard all CS-matrices as morphisms of a category whose objects are matrix dimensions, each dimension having its identity morphism id. If one extends such objects to arbitrary types (with Cartesian product and disjoint union for addition and multiplication of matrix dimensions), this category of matrices turns out to represent the Kleisli category induced by the (finite) distribution monad. In the example above, $f : Bool \to Dist\ Bool$ is represented by a matrix M of type $Bool \to Bool$ (3) on the Kleisli-category side.

Let notation $[\![f]\!]$ mean the matrix which represents probabilistic function f in such a CS-matrix category. For f of type $A \to Dist\ B$, $[\![f]\!]$ will be a matrix of type $A \to B$, that is, cell $b\ [\![f]\!]\ a$ in the matrix [10] records the probability of b in distribution $\delta = f\ a$. Then probabilistic function (monadic) composition,

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

becomes matrix multiplication,

$$[\![f \bullet g]\!] = [\![f]\!] \cdot [\![g]\!] \tag{4}$$

and probabilistic function choice is given by

$$[\![f\ {}_p\!\diamond\ g]\!] \quad = \quad p[\![f]\!] + (1-p)[\![g]\!] \tag{5}$$

where $+$ denotes addition of two matrices of the same type and $p\ M$ denotes the multiplication of every cell in M by probability p.

Clearly, $[\![return]\!] = id$. Any conventional function $f : A \to B$ can be turned into a "sharp" probabilistic one through the composition $return \cdot f$ which, represented as a CS-matrix, is the matrix $M = [\![return \cdot f]\!]$ such that $b\ M\ a = 1$ if $b = f\ a$ and is 0 otherwise. A probabilistic function $f : A \to Dist\ B$ is said to be *sharp* if, for all $a \in A$, $f\ a$ is a Dirac distribution. (Recall that a Dirac distribution is one whose support is a singleton set, the unique element of which is offered with 100% probability.) We will write $[\![f]\!]$ as shorthand for $[\![return \cdot f]\!]$ and therefore rely on the fact that $(f\ a)\ [\![f]\!]\ a = 1$, all other cells being 0.

[10] Following the infix notation usually adopted for relations (which are Boolean matrices), for instance $y \leqslant x$, we write $y\ M\ x$ to denote the contents of the cell in matrix M addressed by row y and column x. This and other notational conventions of the linear algebra of programming are explained in detail in (Oliveira, 2013).

The fact that sharp functions are representable by matrices and that function composition corresponds to chaining the corresponding matrix arrows makes it easy to picture probabilistic functional programs in the form of diagrams in the matrix (Kleisli) category. Take, for instance, the for-loop combinator given above,

$$\begin{array}{l}\mathsf{for}\ b\ i\ 0 = i\\ \mathsf{for}\ b\ i\ (n+1) = b\ (\mathsf{for}\ b\ i\ n)\end{array}$$

and re-write it as follows,

$$\begin{array}{l}(\mathsf{for}\ b\ i)\cdot \underline{0} = \underline{i}\\ ((\mathsf{for}\ b\ i)\cdot \mathsf{succ})\ n = (b \cdot (\mathsf{for}\ b\ i))\ n\end{array}$$

where $\mathsf{succ}\ n = n+1$ and (recall) the under-bar notation denotes constant functions. This is the same as writing two matrix equalities:

$$\begin{array}{l}[\![\mathsf{for}\ b\ i]\!]\cdot [\![\underline{0}]\!] = [\![\underline{i}]\!]\\ [\![\mathsf{for}\ b\ i]\!]\cdot [\![\mathsf{succ}]\!] = [\![b]\!]\cdot [\![\mathsf{for}\ b\ i]\!]\end{array}$$

These can be reduced to a single equality

$$[\![\mathsf{for}\ b\ i]\!]\cdot [[\![\underline{0}]\!] \,|\, [\![\mathsf{succ}]\!]] = [[\![\underline{i}]\!] \,|\, [\![b]\!]\cdot [\![\mathsf{for}\ b\ i]\!]] \tag{6}$$

by resorting to the $[M \,|\, N]$ combinator which glues two matrices $M : A \to C$ and $N : B \to C$ side-by-side, yielding $[M \,|\, N] : A + B \to C$. As explained by Macedo and Oliveira (2013), this combinator — which corresponds to the relational "junc" operator of Bird and de Moor (1997) — is a universal construction in any category of matrices, therefore satisfying (among others) the fusion law

$$P \cdot [M \,|\, N] \quad = \quad [P \cdot M \,|\, P \cdot N] \tag{7}$$

and (for suitably typed matrices) the equality law,

$$[M \,|\, N] = [P \,|\, Q] \quad \equiv \quad M = P \wedge N = Q \tag{8}$$

both silently used in the derivation of (6) above.

Our matrix semantics for the for-loop combinator can still be simplified in two ways: first, the $[\![\cdot]\!]$ parentheses in (6) can be dropped, since we may assume they are implicitly surrounding functions everywhere:

$$(\mathsf{for}\ b\ i)\cdot [\underline{0} \,|\, \mathsf{succ}] = [\underline{i} \,|\, b \cdot (\mathsf{for}\ b\ i)]$$

Second, $[\underline{i} \,|\, b \cdot (\mathsf{for}\ b\ i)]$ can be factored into the composition $[\underline{i} \,|\, b] \cdot (id \oplus (\mathsf{for}\ b\ i))$, since absorption law

$$[M \,|\, N] \cdot (P \oplus Q) \quad = \quad [M \cdot P \,|\, N \cdot Q] \tag{9}$$

holds, where $\cdot \oplus \cdot$ is the matrix direct sum (block) operation: $M \oplus N = \left[\begin{array}{c|c} M & 0 \\ \hline 0 & N \end{array}\right]$. Altogether, we get an equality of two matrix compositions,

$$(\mathsf{for}\ b\ i) \cdot [\underline{0} \,|\, \mathsf{succ}] = [\underline{i} \,|\, b] \cdot (id \oplus (\mathsf{for}\ b\ i))$$

which corresponds to the typed matrix diagram which follows:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[rr, bend left, "\mathsf{out}=\mathsf{in}^\circ"] \arrow[d, "\mathsf{for}\ b\ i"'] & \cong & 1+\mathbb{N}_0 \arrow[ll, bend left, "\mathsf{in}=[\underline{0}\,|\,\mathsf{succ}]"] \arrow[d, "id\oplus(\mathsf{for}\ b\ i)"] \\
B & & 1+B \arrow[ll, bend left, "{[\underline{i}\,|\,b]}"]
\end{tikzcd} \tag{10}$$

Symbol $\cong$ indicates that function $\mathsf{in} = [\underline{0} \,|\, \mathsf{succ}]$ is a bijection, and therefore its converse (or inverse) in° is also a function.[11] As is costumary, we denote by out the converse of in, as in (10). Note that bijections are the olny CS matrices which are invertible.

Why does diagram (10) matter? First, it can be recognized as an instance of a *catamorphism* diagram (Bird and de Moor, 1997), here interpreted in the category of CS-matrices rather than in that of total functions or binary relations — the *qualitative* to *quantitative* shift promised in the introduction of this paper.

In fact, because composition is closed for CS-matrices and these include sharp functions, b and $\underline{i}$ can vary inside the CS-matrix space and the diagram will still make sense. For instance, the base case, which is represented by constant function $\underline{i} : 1 \to \mathbb{N}_0$ — a column vector — corresponds to the Dirac distribution on i, which can be changed to any other distribution. Moreover, the diagram tells that $\mathsf{for}\ b\ i$ is a solution to the equation $k \cdot \mathsf{in} = [\underline{i} \,|\, b] \cdot (id \oplus k)$. Because in is a bijection, this yields the *unique* solution[12] characterized by universal property:

$$k = \mathsf{for}\ b\ i \quad \equiv \quad k \cdot \mathsf{in} = [\underline{i} \,|\, b] \cdot (id \oplus k) \tag{11}$$

This unique solution can be computed as the fixpoint in k of equation

$$k \quad = \quad [\underline{i} \,|\, b \cdot k] \cdot \mathsf{out} \tag{12}$$

which is obtained from (11) by absorption (9) and shunting $\mathsf{in} = [\underline{0} \,|\, \mathsf{succ}]$ to the right-hand side, since it is a bijection.

Equation (12) also serves to emulate the construction of the least fixpoint using matrix algebra packages such as, for instance, MATLAB. In this case, we build finite approximations of the fixpoint by restricting to (say) n inputs (from 0 to $n-1$) and m

[11] In general, by the converse M° of a matrix M we mean its transpose, that is, $x\ M^\circ\ y = y\ M\ x$ holds: the effect is that of swapping rows with columns.

[12] The argument is the same as in (Bird and de Moor, 1997) just by replacing the powerset monad by the distribution monad. More generally, it is standard that an initial algebra of base functor F lifts to the corresponding initial algebra in the Kleisli category of a monad which distributes over F. The so-called *polynomial* (or *shapely*) functors distribute over the probabilistic monad (Hasuo et al., 2007).

outputs (from 0 to $m-1$):[13]

$$\begin{tikzcd} n \arrow[r, "\mathsf{out}"] \arrow[d, "k"'] & 1+n \arrow[d, "id \oplus k"] \\ m & 1+m \arrow[l, "{[\underline{i} \mid b]}"] \end{tikzcd}$$

Let us see an example: suppose we want to emulate a fault in the odd function, $odd = (1+) \cdot (2*)$, in which $(2*) = \mathsf{for}\ (2+)\ 0$ is disturbed by the propagation of the same fault of addition operator we have seen before:

$$ftwice_p = \mathsf{mfor}\ fadd_p\ 2\ 0 = \mathsf{mfor}\ (id\ {}_p\!\diamond (x+))\ 0$$

For instance, $ftwice_{0.1}\ 4$ is the distribution

```
8  65.6%
6  29.2%
4   4.9%
2   0.4%
0   0.0%
```

In MATLAB, we will first draw the corresponding diagram,

$$\begin{tikzcd} n \arrow[r, "\mathsf{out}={[\underline{0} \mid \mathsf{succ}]}^{\circ}"] \arrow[d, "ftwice_p"'] & 1+n \arrow[d, "id \oplus ftwice_p"] \\ m & 1+m \arrow[l, "{[\underline{0} \mid id_p \diamond (2+)]}"] \end{tikzcd}$$

parametric on probability p and the n and m dimensions, which nevertheless have to be passed explicitly when encoding each arrow of the diagram as a MATLAB matrix. The probabilistic choice occurring in the corresponding instance of (12),

$$k \quad = \quad [\underline{0} \,|\, (id\ {}_p\!\diamond (2+)) \cdot k] \cdot [\underline{0} \,|\, \mathsf{succ}]^{\circ} \tag{13}$$

is encoded in MATLAB as function:

```
function C = choice(p,M,N)
    if size(M) ~= size(N)
        error('Dimensions must agree');
    else
        C = p*M+(1-p)*N
   end
end
```

[13]Compared to (10), this diagram corresponds to restricting $\mathbb{N}_0$ to the first n natural numbers (finite approximation), similarly for m. As MATLAB is not typed, tracing matrix dimensions without the help of diagrams of this kind would be a nightmare.

— recall (5) and note the need for explicit type error checking. This is used in the MATLAB encoding of $fadd_p$ (2)

```
function R = fadd(p,x,m)
%  fadd : P -> x -> m -> m
   R = choice(p,eye(m),addk(k,m,m));
end
```

where dimension m is again passed as parameter, eye is the MATLAB constructor of identity matrices and $addk$ is a suitable function encoding $(k+)$ using matrices. Using standard linear algebra, the right-hand side of equation (13) unfolds into the following MATLAB code, parametric on p:

```
function R = twiceF(p,K)
   [m n] = size(K);
   R = zero(m)*zero(n)' + fadd(p,2,m)*K*succ(n,n)';
end
```

For $n, m = 5, 8$ and $p = 0.1$, the least fixpoint of equation (13) — i.e. $K = twiceF\ (p, K)$ in MATLAB— is the matrix

```
1       0.1     0.01    0.001   0.0001
0       0       0       0       0
0       0.9     0.18    0.027   0.0036
0       0       0       0       0
0       0       0.81    0.243   0.0486
0       0       0       0       0
0       0       0       0.729   0.2916
0       0       0       0       0
0       0       0       0       0.6561
```

whose leftmost column (resp. top row) corresponds to input (resp. output) 0. The five columns of the matrix correspond to the distributions output by the monadic $ftwice_{0.1}\ n$, for $n = 0 \mathinner{..} 4$.

So much for an illustration of the correspondence between monadic probabilistic programming (in Haskell) and column stochastic matrix construction (in MATLAB). In the following section we will return to analytical methods relying solely on universal property (11) and its corollaries.

6. Probabilistic mutual recursion in the LAoP

As we have seen above, mutual recursion arises from the *pairing* — *tupling*, in general (Hu et al., 1997) — of two (sharp) functions f and g, defined by

$$(f \mathbin{\vartriangle} g)\ x = (f\ x, g\ x)$$

where $f \mathbin{\vartriangle} g : A \rightarrow B \times C$ for $f : A \rightarrow B$ and $g : A \rightarrow C$. This tupling operator is known as *split* in the functional setting (Bird and de Moor, 1997) or as *fork* in the relational one (Frias et al., 1997; Schmidt, 2010). Macedo (2012) shows that these operators generalize to the so-called Khatri-Rao product $M \vartriangle N$ of two arbitrary matrices M and N, defined index-wise by

$$(b, c)\ (M \vartriangle N)\ a \quad = \quad (b\ M\ a) \times (c\ N\ a) \tag{14}$$

Thus the Khatri-Rao product is a "column-wise" version of the well-known Kronecker product $M \otimes N$ defined by:

$$(y, x)\ (M \otimes N)\ (b, a) \quad = \quad (y\ M\ b) \times (x\ N\ a) \tag{15}$$

Both products are intimately related by the *absorption* law

$$(M \otimes N) \cdot (P \vartriangle Q) = (M \cdot P) \vartriangle (N \cdot Q) \tag{16}$$

valid for any (suitably typed) matrices M, N, P, Q (Macedo, 2012).

Khatri-Rao coincides with Kronecker for column vectors $u : 1 \to B$, $v : 1 \to C$,

$$u \vartriangle v = u \otimes v \tag{17}$$

and commutes with matrix junc'ing via the *exchange law* (Macedo, 2012):

$$[M \mid N] \vartriangle [P \mid Q] = [M \vartriangle P \mid N \vartriangle Q] \tag{18}$$

for suitably typed matrices M, N, P and Q.

For *sharp* functions f and g, pairing is an universal construct ensuring that any function k producing pairs is uniquely factored to the left and to the right,

$$k = f \vartriangle g \quad \equiv \quad \mathit{fst} \cdot k = f \wedge \mathit{snd} \cdot k = g \tag{19}$$

where $\mathit{fst}\ (b, c) = b$ and $\mathit{snd}\ (b, c) = c$. (Note how liberally we keep omitting the $[\![\cdot]\!]$ parentheses around the occurrence of functions inside matrix expressions.)

From (19) a number of useful corollaries arise, namely (keep in mind that f and g should be sharp functions for the time being) *fusion*,

$$(f \vartriangle g) \cdot h = (f \cdot h) \vartriangle (g \cdot h) \tag{20}$$

reconstruction,[14]

$$k = (\mathit{fst} \cdot k) \vartriangle (\mathit{snd} \cdot k) \tag{21}$$

reflection

$$\mathit{fst} \vartriangle \mathit{snd} \quad = \quad \mathit{id} \tag{22}$$

and pairwise *equality*:

$$k \vartriangle h = f \vartriangle g \quad \equiv \quad k = f \wedge h = g \tag{23}$$

[14]Cf. *loss-less decomposition* (Oliveira, 2011).

This makes it easy to prove the mutual recursion law, below instantiated to for-loops, where $\mathsf{F}\ f$ abbreviates $id \oplus f$: [15]

$$
\begin{array}{ll}
& f \vartriangle g = \mathsf{for}\ (h \vartriangle k)\ (i, j) \\
\equiv & \qquad \{\ \text{universal property (11)}\ \} \\
& (f \vartriangle g) \cdot \mathsf{in} = [\underline{i, j} \mid h \vartriangle k] \cdot \mathsf{F}\ (f \vartriangle g) \\
\equiv & \qquad \{\ \text{fusion (20) ; constant functions}\ \} \\
& (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = [\underline{i} \vartriangle \underline{j} \mid h \vartriangle k] \cdot \mathsf{F}\ (f \vartriangle g) \\
\equiv & \qquad \{\ \text{exchange law (18)}\ \} \\
& (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = ([\underline{i} \mid h] \vartriangle [\underline{j} \mid k]) \cdot \mathsf{F}\ (f \vartriangle g) \\
\equiv & \qquad \{\ \text{fusion (20) again}\ \} \\
& (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = ([\underline{i} \mid h] \cdot \mathsf{F}\ (f \vartriangle g)) \vartriangle ([\underline{j} \mid k] \cdot \mathsf{F}\ (f \vartriangle g)) \\
\equiv & \qquad \{\ \text{equality (23)}\ \} \\
& \left\{ \begin{array}{l} f \cdot \mathsf{in} = [\underline{i} \mid h] \cdot \mathsf{F}\ (f \vartriangle g) \\ g \cdot \mathsf{in} = [\underline{j} \mid k] \cdot \mathsf{F}\ (f \vartriangle g) \end{array} \right. \\
\Box &
\end{array}
$$

Read in reverse direction, this reasoning explains how two recursive, mutually dependent functions f and g (regarded as matrices) combine with each other into one single function $f \vartriangle g$, from which one can extract both f and g by projecting according to the *cancellation* rule,

$$
fst \cdot (f \vartriangle g) = f \ \wedge\ snd \cdot (f \vartriangle g) = g \tag{24}
$$

yet another corollary of (19).

The law just derived can be identified as the underpinning of the (pointwise) derivations of *fibl* (resp. sql) from *fib* (resp. sq) back to section 2. But note that f and g have been regarded as *sharp* functions thus far, and therefore what we have written is just a rephrasing of what can be found already in the literature of *tupling*, see e.g. references (Bird and de Moor, 1997; Hu et al., 1997) among others.

We are now interested in checking the probabilistic generalization of (19). Let two probabilistic functions f and g and their product $f \vartriangle g$ be depicted as the CS-matrices

[15] As is well-known, for sharp functions this law extends to other inductive types, e.g. lists, trees etc (Bird and de Moor, 1997; Hu et al., 1997).

of the following diagram:

$$
\begin{tikzcd}
2 & 2 \times 3 \arrow[l, "fst=\begin{bmatrix}1&1&1&0&0&0\\0&0&0&1&1&1\end{bmatrix}"'] \arrow[r, "snd=\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&0\\0&0&1&0&0&1\end{bmatrix}"] & 3 \\
& & \\
& 4 \arrow[uul, "f=\begin{bmatrix}0.5&0.3&0&0.75\\0.5&0.7&1&0.25\end{bmatrix}"] \arrow[uu, "f \vartriangle g = \begin{bmatrix}0.15&0.12&0&0\\0.35&0.06&0&0.75\\0&0.12&0&0\\0.15&0.28&0.1&0\\0.35&0.14&0.2&0.25\\0&0.28&0.7&0\end{bmatrix}"] \arrow[uur, "g=\begin{bmatrix}0.3&0.4&0.1&0\\0.7&0.2&0.2&1\\0&0.4&0.7&0\end{bmatrix}"'] &
\end{tikzcd}
$$

Note that, in general, projections fst and snd regarded as matrices are succinctly defined by:

$$fst = id \otimes ! \ , \quad snd = ! \otimes id \tag{25}$$

Here, ! denotes the unit of the Khatri-Rao product

$$! \vartriangle M \ = \ M \ = \ M \vartriangle ! \tag{26}$$

which is the unique row vector of its type wholly filled with 1s (Macedo, 2012). We can handle this product and its projections in Haskell by running the following monadic functions

$$
\begin{aligned}
&(f \vartriangle g)\ a = \mathbf{do}\ \{ b \leftarrow f\ a; c \leftarrow g\ a; return\ (b, c) \} \\
&mfst\ d = \mathbf{do}\ \{ (b, c) \leftarrow d; return\ b \} \\
&msnd\ d = \mathbf{do}\ \{ (b, c) \leftarrow d; return\ c \}
\end{aligned}
$$

inside the distribution monad $Dist$, thereby implementing the Khatri-Rao product and its projections. For instance, $(f \vartriangle g)\ 2$ above will yield

```
(2,1)  28.0%
(2,3)  28.0%
(2,2)  14.0%
(1,1)  12.0%
(1,3)  12.0%
(1,2)   6.0%
```

as in the second column of the corresponding matrix given above. Moreover, both in Haskell and MATLAB we can observe the cancellations $fst \cdot (f \vartriangle g) = f$ and $snd \cdot (f \vartriangle g) = g$.

However, *reconstruction* (21) does not hold for an arbitrary probabilistic function k. This is because not every CS-matrix $k : A \rightarrow B \times C$ outputting pairs is the Khatri-

Rao product of two CS-matrices, as the following counter-example shows: matrix

$$k : 3 \to 2 \times 3$$
$$k = \begin{bmatrix} 0 & 0.4 & 0.2 \\ 0.2 & 0 & 0.17 \\ 0.2 & 0.1 & 0.13 \\ 0.6 & 0.4 & 0.2 \\ 0 & 0 & 0.17 \\ 0 & 0.1 & 0.13 \end{bmatrix}$$

cannot be recovered from its projections, cf. the first column in:

$$(fst \cdot k) \vartriangle (snd \cdot k) = \begin{bmatrix} 0.24 & 0.4 & 0.2 \\ 0.08 & 0 & 0.17 \\ 0.08 & 0.1 & 0.13 \\ 0.36 & 0.4 & 0.2 \\ 0.12 & 0 & 0.17 \\ 0.12 & 0.1 & 0.13 \end{bmatrix}$$

This happens because probabilistic Khatri-Rao is a *weak* product in the category of CS-matrices[16] — the expected equivalence (19) is only an implication,

$$k = f \vartriangle g \quad \Rightarrow \quad fst \cdot k = f \wedge snd \cdot k = g \tag{27}$$

ensuring existence but not uniqueness. The proof of (27), which is equivalent to cancellation (24) — substitute k and simplify — can be found in Appendix A. This proof relies on properties (16) and (26) of the Khatri-Rao product.

Weak product (27) also grants pairwise equality (23) — substitute k by $k \vartriangle h$ and simplify — but the converse substitution of f and g, in the $\Leftarrow$ direction, leading to *reconstruction* (21) is invalid. In turn, this invalidates fusion (20) for arbitrary probabilistic functions f, g and h, although the property will still hold in case h is sharp [17], as the straightforward proof of (A.1) in Appendix A shows.

Altogether, the mutual recursion law will not hold in general for probabilistic functions, as its calculation (above) relies on fusion (20). This is consistent with what we have observed in section 4 concerning the two versions of Fibonacci, mfib before the application of mutual recursion and mfibl after, which differ substantially for inputs larger than 4. However, the corresponding pair of probabilistic functions of the other example — msq and $msql$ — seemed to be the same (ie. probabilistically indistinguishable), as much as could be tested.

In the following section we explain the difference observed in the two experiments by investigating sufficient conditions for the mutual recursion law to hold for probabilistic functions (CS-matrices).

[16] Recall that CS-matrices represent *total* probabilistic functions.

[17] The same happens with *forks* in relation algebra (Bird and de Moor, 1997).

7. Asymmetric Khatri-Rao product

In order to convert (27) into equivalence (19) generalized to probabilistic f, g and k, we have to find conditions for the converse implication

$$k = f \vartriangle g \quad \Leftarrow \quad \mathit{fst} \cdot k = f \wedge \mathit{snd} \cdot k = g$$

to hold, which is equivalent to (21) under the substitution or introduction of variables f and g. For this we may seek inspiration in relation algebra, where one knows that if one of the projections of a binary relation R outputting pairs is functional (ie., deterministic), then $(b, c)\ R\ a \equiv b\ (\mathit{fst} \cdot R)\ a\ \wedge\ c\ (\mathit{snd} \cdot R)\ a$ holds. That is, by forking $\mathit{fst} \cdot R$ and $\mathit{snd} \cdot R$ one rebuilds R.

Back to probabilistic functions (ie. CS-matrices), this suggests the conjecture:

If either $\mathit{fst} \cdot k$ or $\mathit{snd} \cdot k$ are sharp functions then (21) holds. (28)

Some remarks first, before checking this conjecture. Let $k : A \to B \times C$ be a CS-matrix (example aside, for two element data types A, B and C). The fact that $f = \mathit{fst} \cdot k : A \to B$ is sharp, e.g.

$$k \quad = \quad \begin{array}{c c} & \begin{array}{cc} a_1 & a_2 \end{array} \\ \begin{array}{c} (b_1, c_1) \\ (b_1, c_2) \\ (b_2, c_1) \\ (b_2, c_2) \end{array} & \begin{bmatrix} 0.3 & 0 \\ 0.7 & 0 \\ 0 & 0.4 \\ 0 & 0.6 \end{bmatrix} \end{array}$$

$$f \quad = \quad \begin{array}{c c} & \begin{array}{cc} a_1 & a_2 \end{array} \\ \begin{array}{c} b_1 \\ b_2 \end{array} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{array}$$

in the example — means that, for $b = f\ a$, the corresponding C-block in matrix k adds up to 1 and all the other entries in the a-column of k are 0. Projection $\mathit{snd} \cdot k : A \to C$ yields such blocks (aside); $\langle \mathit{fst} \cdot k, \mathit{snd} \cdot k \rangle$ puts these back in place, rebuilding k.

$$\mathit{snd} \cdot k \quad = \quad \begin{array}{c c} & \begin{array}{cc} a_1 & a_2 \end{array} \\ \begin{array}{c} c_1 \\ c_2 \end{array} & \begin{bmatrix} 0.3 & 0.4 \\ 0.7 & 0.6 \end{bmatrix} \end{array}$$

To prove (28) we will resort to the definition of (typed) matrix composition, for $M : B \to C$ and $N : A \to B$:

$$c(M \cdot N)a \quad = \quad \langle \textstyle\sum\ b\ ::\ (c\ M\ b) \times (b\ N\ a) \rangle \tag{29}$$

We also need two rules which interface index-free and index-wise matrix notation,

$$y(f \cdot N)x \quad = \quad \langle \textstyle\sum\ z\ :\ y = f\ z\ :\ zNx \rangle \tag{30}$$

$$y(g^\circ \cdot N \cdot f)x \quad = \quad (g\ y)\ N\ (f\ x) \tag{31}$$

where N is an arbitrary matrix and f, g are functional (ie. sharp) matrices.[18]

[18] These rules are derived by Oliveira (2013) adopting the Eindhoven notation (Backhouse and Michaelis, 2006; Morgan, 2012) for summations, e.g. $\langle \sum\ x\ :\ R :\ S \rangle$ where R is the range (a predicate) which binds the dummy x and S is the summand. $\langle \sum\ x\ ::\ S \rangle$ corresponds to R true for all x, the convention being to omit R in this case.

Let us suppose $\mathit{fst} \cdot k$ in (21) is sharp, denoting by $f : A \to B$ such a sharp function: $f = \mathit{fst} \cdot k$. Regarded as a matrix, f is such that $b\ f\ a = 1$ if $b = f\ a$, otherwise $b\ f\ a = 0$. It is easy to check that facts

$$\langle \sum\ c\ ::\ (f\ a, c)\ k\ a\rangle = 1 \tag{32}$$

$$\langle \sum\ (b, c)\ :\ (b \neq f\ a) :\ ((b, c)\ k\ a)\rangle = 0 \tag{33}$$

hold — see below. Define $m = \langle \mathit{fst} \cdot k, \mathit{snd} \cdot k\rangle$, that is,

$$(b, c)\ m\ a = (b\ (\mathit{fst} \cdot k)\ a) \times (c\ (\mathit{snd} \cdot k)\ a)$$

the same as

$$(b, c)\ m\ a = (b\ f\ a) \times \langle \sum\ b'\ ::\ (b', c)\ k\ a\rangle \tag{34}$$

since $f = \mathit{fst} \cdot k$ an snd is sharp (30). Our aim is to prove that $m = k$.

Case $b \neq f\ a$:. In this case $b\ f\ a = 0$ and (34) yields $(b, c)\ m\ a = 0$, for all a, b and c. From (33) we also get $(b, c)\ k\ a = 0$ and so $m = k$ for this case.

Case $b = f\ a$:. we have

$$\begin{aligned}
& (f\ a, c)\ m\ a \\
= & \quad \{\ (34)\ ;\ (b\ f\ a) = 1 \text{ for } b = f\ a\ \} \\
& \langle \sum\ b'\ ::\ (b', c)\ k\ a\rangle \\
= & \quad \{\ b' = f\ a\ \vee\ b' \neq f\ a\ \} \\
& \langle \sum\ b'\ :\ b' = f\ a\ \vee\ b' \neq f\ a :\ (b', c)\ k\ a\rangle \\
= & \quad \{\ \text{split summation ; one-point over } b' = f\ a\ \} \\
& ((f\ a, c)\ k\ a) + \langle \sum\ b'\ :\ b' \neq f\ a :\ (b', c)\ k\ a\rangle \\
= & \quad \{\ (33)\ \} \\
& (f\ a, c)\ k\ a
\end{aligned}$$

Thus m and k are extensionally the same for all cells addressed by $(f\ a, c)$, completing the proof.
□

The proof assuming $\mathit{snd} \cdot k$ sharp instead of $\mathit{fst} \cdot k$ being so is essentially the same. The remaining assumptions (32) and (33) are easily proved in the appendix.

8. Probabilistic mutual recursion resumed

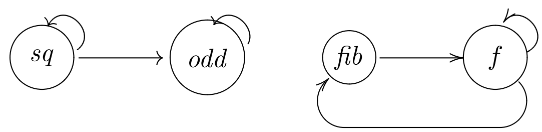
Back to the case studies of section 4, we now capitalize on the result of the previous section granting that, if one of the projections of a probabilistic pair-valued function k is a sharp function, then property (19) holds and all its corollaries. [19] This means that,

[19] This includes, of course, the standard case in which both f and g are sharp functions.

under the same assumption, the mutual recursion law will hold too.

Put in other words, the probabilistic behaviour of a pair-valued recursive function k, for instance a for-loop $k = \mathsf{for}\ b\ i$, will be the same as the product $f \vartriangle g$ of its mutually recursive projections f and g, provided either f is sharp or g is sharp.

This enables us to spot a difference between the two examples of section 4 just by looking at the corresponding call graphs:



We see that sq depends on itself and on odd but odd only depends on itself. Probabilistic msq was obtained from sq by injecting a fault in the addition operation but this did not interfere with odd, which remained a sharp function. Thus $msql$ and msq exhibit the same probabilistic behaviour.

Comparatively, $mfib$ was obtained from fib by injecting a similar fault but this time the fault propagates to its derivative f and then back to $mfib$. Thus both $mfib$ and f are genuinely probabilistic and the derived linear version $mfibl$ is not granted to exhibit the same behaviour.

This can be confirmed by further querying our experiments in two ways. First, we check that the odd projection of $msql$ remains sharp in spite of the probabilistic process it runs inside of: we define $msqlo$ as the same as $msql$ but returning o instead of s,

$$
\begin{array}{l}
msqlo\ n = \mathbf{do}\ \{(s,o) \leftarrow \mathsf{mfor}\ loop\ (0,1)\ n; return\ o\}\ \mathbf{where} \\
\quad loop\ (s,o) = \mathbf{do}\ \{z \leftarrow fadd_{0.1}\ s\ o; return\ (z, o+2)\}
\end{array}
$$

and run for instance

```
Main> msqlo 5
11 100.0%
```

to observe that it yields the Dirac distribution on 11, the fifth odd number; while its companion projection yields:

```
Main> msql 5
25  65.6%
 9  10.0%
16   9.0%
21   8.1%
24   7.3%
```

Second, we disturb this situation by injecting another fault, this time in the odd function itself:

$$
\begin{array}{l}
odd'\ 0 = return\ 1 \\
odd'\ (n+1) = \mathbf{do}\ \{x \leftarrow odd'\ n; fadd_{0.1}\ 2\ x\}
\end{array}
$$

Then we check that suitably adapted msq, mutually dependent on odd',

$$msq'\ 0 = return\ 0$$
$$msq'\ (n+1) = \mathbf{do}\ \{m \leftarrow msq'\ n; x \leftarrow odd'\ n; fadd_{0.1}\ m\ x\}$$

and its linear version,

$$msql'\ n = \mathbf{do}\ \{(s,o) \leftarrow \mathsf{mfor}\ loop\ (0,1)\ n; return\ s\}\ \mathbf{where}$$
$$loop\ (s,o) = \mathbf{do}\ \{z \leftarrow fadd_{0.1}\ s\ o; x \leftarrow fadd_{0.1}\ 2\ o; return\ (z,x)\}$$

now exhibit different probabilistic behaviours, for instance:

n	$msq'\ n$		$msql'\ n$	
	9	59.0%	9	65.6%
	7	19.7%	5	15.4%
	5	10.3%	7	7.3%
	8	6.6%	8	7.3%
3	6	2.2%	3	2.6%
	3	1.9%	4	0.8%
	4	0.2%	6	0.8%
	1	0.1%	1	0.1%
	2	0.0%	2	0.1%

As in the Fibonacci example, we observe that linear scores better than mutually recursive.

9. Generalizing to other fault propagation patterns

Besides mutual recursion, other fault propagation patterns in functional programs arise from calculations in the LAoP. These extend to other datatypes, as for-loops generalize to folds over lists, and more generally to catamorphisms over other inductive data types (Bird and de Moor, 1997). Below we give examples of this generalization.

Base-case fault distribution. The first example, still dealing with for-loops, shows that faults in the base case propagate linearly through the choice operator — the law of *base-case fault distribution*:

$$\mathsf{for}\ f\ (a\ {}_p\!\diamond\ b) \quad = \quad (\mathsf{for}\ f\ a)\ {}_p\!\diamond\ (\mathsf{for}\ f\ b) \tag{35}$$

The need for a generalization can be seen already in writing "$a\ {}_p\!\diamond\ b$", an abuse of notation since the choice operator chooses between functions, not arbitrary values. Thus construct $\mathsf{for}\ f\ \ i$ has to give room to $(\![h\,|\,f]\!)$, where standard *catamorphism* notation (Bird and de Moor, 1997) is adopted to give freedom to the base case to be any probabilistic function h of its type. Thus (11) becomes, for $\mathsf{F}\ f = id \oplus f$,

$$k = (\![h\,|\,f]\!) \quad \equiv \quad k \cdot \mathsf{in} = [h\,|\,f] \cdot (\mathsf{F}\ k) \tag{36}$$

Clearly,

$$\mathsf{for}\ f\ a \quad = \quad (\![\underline{a}\,|\,f]\!) \tag{37}$$

holds. In (35), abbreviation for $f\ (a\ {}_p\diamond\ b)$ replacing $(\![\underline{a}\ {}_p\diamond\ \underline{b}\,|\,f]\!)$ is welcome as it enhances readability.

The proof of (35) is given in Appendix A. It relies on properties of probabilistic choice already given by Oliveira (2012), namely *choice-fusion*

$$(f\ {}_p\diamond\ g) \cdot h \quad = \quad (f \cdot h)\ {}_p\diamond\ (g \cdot h) \tag{38}$$

$$h \cdot (f\ {}_p\diamond\ g) \quad = \quad (h \cdot f)\ {}_p\diamond\ (h \cdot f) \tag{39}$$

and the *exchange law*:

$$[f \mid g]\ {}_p\diamond\ [h \mid k] \quad = \quad [f\ {}_p\diamond\ h \mid g\ {}_p\diamond\ k] \tag{40}$$

Pipelining. Other interesting patterns of fault propagation arise in *pipelining*, that is, compositions of probabilistic functions $k = f \cdot g$ whereby one is able to obtain the *fault of the whole* (probabilistic k) expressed in terms of the *faults of its parts* (probabilistic f and g) by "fault fusion".

The example of fault fusion given below involves *sequences* rather than natural numbers, which means evolving from the for combinator to the corresponding combinator at sequence processing level, here given in Haskell (monadic) notation: [20]

$$\begin{array}{l} \mathsf{fold}\ f\ d\ [\,] = d \\ \mathsf{fold}\ f\ d\ (h : t) = \mathbf{do}\ \{x \leftarrow \mathsf{fold}\ f\ d\ t; f\ (h, x)\} \end{array}$$

The semantics of this combinator are captured in linear algebra by the universal property

$$k = \mathsf{fold}\ f\ d \quad \equiv \quad k \cdot \mathsf{in} = [d \,|\, f] \cdot (\mathsf{F}\ k) \tag{41}$$

where $\mathsf{F}\ k = id \oplus id \otimes k$ and $\mathsf{in} = [\mathsf{nil} \,|\, \mathsf{cons}]$ is the initial algebra of sequences, for $\mathsf{nil}\ \_ = [\,]$ and $\mathsf{cons}\ (h, t) = h : t$. Recursive pattern $\mathsf{F}\ k = id \oplus id \otimes k$ involves, besides direct sum $(id \oplus \cdot)$ splitting base from recursive case (as in for), the Kronecker product $id \otimes k$ which delivers to f the head of the input sequence and the outcome of the recursive call over its tail — the pair (h, x) in the code above. The base case is captured in (41) by vector d, a distribution. By substituting k by $\mathsf{fold}\ f\ d$ and in-lining the definition of $\mathsf{F}\ k$ in (41) we get the cancellation property

$$\mathsf{fold}\ f\ d \cdot \mathsf{in} = [d \,|\, f \cdot (id \otimes (\mathsf{fold}\ f\ d))] \tag{42}$$

from which the Haskell code above is derived by monadic conversion.

As examples, consider $count = \mathsf{fold}\ (\mathsf{succ} \cdot snd)\ \underline{0}$, the function that counts how many items can be found in the input sequence, and $cat = \mathsf{fold}\ \mathsf{cons}\ \mathsf{nil}$, that which copies the input sequence to the output (thus $cat = id$). Suppose there is some risk that cat might fail passing items from input to output, with probability p, as captured by

$$fcat_p = \mathsf{fold}\ (lose\ {}_p\diamond\ send)\ \mathsf{nil}$$

[20] As already mentioned, both are instances of the generic probabilistic *catamorphism* construct, see (45) in section 10.

where $lose = snd$ and $send = \mathsf{cons}$. For instance, for $p = 0.1$, distribution $fcat_{0.1}$ `"abc"` will range from perfect copy (72.9%) to complete loss (0.1%):

```
"abc"  72.9%
 "ab"   8.1%
 "ac"   8.1%
 "bc"   8.1%
  "a"   0.9%
  "b"   0.9%
  "c"   0.9%
   ""   0.1%
```

Now suppose that $count$ too may be faulty in the sense of skipping elements with probability q:

$$fcount_q = \mathsf{fold}\ ((id\ {}_q{\diamond}\ \mathsf{succ}) \cdot snd)\ \underline{0}$$

For instance, for $q = 0.15$, distribution $fcount_{0.15}$ `"abc"` will be:

```
3  61.4%
2  32.5%
1   5.7%
0   0.3%
```

What can we tell about the risk of faults in the pipeline $fcount_q \cdot fcat_p$? We could try specific runs, e.g. $(fcount_{0.15} \cdot fcat_{0.1})$ `"abc"` yielding distribution

```
3  44.8%
2  41.3%
1  12.7%
0   1.3%
```

whose figures combine, *in some way*, those given earlier for the individual runs.

What we would like to know is the *general* formula which combines such figures and expresses the overall risk of failure. For this we resort to the *fusion law* which emerges from (41) in the standard way (Bird and de Moor, 1997) and also in the probabilistic setting:

$$k \cdot (\mathsf{fold}\ g\ e) = \mathsf{fold}\ f\ d \quad \Leftarrow \quad k \cdot [e \mid g] = [d \mid f] \cdot (\mathsf{F}\ k) \tag{43}$$

In our case, this enables us to solve the equation $fcount_q \cdot fcat_p = \mathsf{fold}\ x\ y$ for unknowns x and y:

$$\begin{array}{ll}
& fcount_q \cdot fcat_p = \mathsf{fold}\ x\ y \\
\Leftarrow & \quad \{\ \text{fold fusion (43) ; definition of } fcat_p\ \} \\
& fcount_q \cdot [\mathsf{nil} \mid lose\ {}_p{\diamond}\ send] = [x \mid y] \cdot (\mathsf{F}\ fcount_q) \\
\equiv & \quad \{\ \text{(7) ; definition of F; (9) ; (8)}\ \} \\
& \left\{ \begin{array}{l} fcount_q \cdot \mathsf{nil} = x \\ fcount_q \cdot (lose\ {}_p{\diamond}\ send) = y \cdot (id \otimes fcount_q) \end{array} \right.
\end{array}$$

$$\equiv \quad \{\ fcount_q \cdot \mathsf{nil} = \underline{0}\ ;\ lose = snd;\ send = \mathsf{cons}\ \}$$

$$\begin{cases} x = \underline{0} \\ fcount_q \cdot (snd\ {}_{p}\!\diamond\ \mathsf{cons}) = y \cdot (id \otimes fcount_q) \end{cases}$$

Second, we solve the second equality just above for y:

$$\begin{array}{ll}
& fcount_q \cdot (snd\ {}_{p}\!\diamond\ \mathsf{cons}) = y \cdot (id \otimes fcount_q) \\
\equiv & \{\ \text{choice fusion (39)}\ \} \\
& (fcount_q \cdot snd)\ {}_{p}\!\diamond\ (fcount_q \cdot \mathsf{cons}) = y \cdot (id \otimes fcount_q) \\
\equiv & \{\ \text{unfolding } fcount_q \cdot \mathsf{cons}\ \} \\
& (fcount_q \cdot snd)\ {}_{p}\!\diamond\ ((id\ {}_{q}\!\diamond\ \mathsf{succ}) \cdot snd \cdot (id \otimes fcount_q)) \\
& = y \cdot (id \otimes fcount_q) \\
\equiv & \{\ \text{free theorem of } snd\ \} \\
& (fcount_q \cdot snd)\ {}_{p}\!\diamond\ ((id\ {}_{q}\!\diamond\ \mathsf{succ}) \cdot fcount_q \cdot snd) \\
& = y \cdot (id \otimes fcount_q) \\
\equiv & \{\ \text{choice fusion (38) factoring } fcount_q \cdot snd\ \} \\
& (id\ {}_{p}\!\diamond\ (id\ {}_{q}\!\diamond\ \mathsf{succ})) \cdot fcount_q \cdot snd = y \cdot (id \otimes fcount_q) \\
\equiv & \{\ \text{free theorem of } snd \text{ again}\ \} \\
& (id\ {}_{p}\!\diamond\ (id\ {}_{q}\!\diamond\ \mathsf{succ})) \cdot snd \cdot (id \otimes fcount_q) = y \cdot (id \otimes fcount_q) \\
\Leftarrow & \{\ \text{Leibniz } (id \otimes fcount_{.} \text{ cancelled from both sides})\ \} \\
& y = (id\ {}_{p}\!\diamond\ (id\ {}_{q}\!\diamond\ \mathsf{succ})) \cdot snd
\end{array}$$

$\square$

Summing up, we have been able to consolidate the risk of the pipeline $fcount_q \cdot fcat_p$, obtaining the overall behavior

$$\begin{array}{l}
fcount_q \cdot fcat_p = \mathsf{fold}\ y\ \underline{0}\ \mathbf{where} \\
\quad y = ((p + q - pq)\ id + (1-p)\ (1-q)\ \mathsf{succ}) \cdot snd
\end{array}$$

in which the probabilistic definition of y combines the choices according to (5). It can be checked that this behaviour (which corresponds to that of an even more risky counter reading from a perfect channel) matches up with the distributions obtained for the specific runs given earlier.

Consolidating risk by fusion offers new opportunities for reasoning about faulty pipelines. For instance, from the expression given by y above we can infer that different pipelines may have the same behaviour, e.g.

$$fcount_0 \cdot fcat_p \quad = \quad fcount_p \cdot fcat_0$$

since terms

$$(0 + p - 0\ p)\ id + (1 - 0)\ (1 - p)\ \mathsf{succ}$$
$$(p + 0 - p0)\ id + (1 - p)\ (1 - 0)\ \mathsf{succ}$$

are the same. In words: for the same probabilities, a *perfect* counter reading from a *faulty* channel is indistinguishable from a *faulty* counter reading from a *perfect* channel.

10. Probabilistic "banana-split"

Our final result has to do with a program transformation technique known as *banana-split* (Bird and de Moor, 1997). Suppose you want to compute the average of a non-empty list of integers:

$$avg\ l \quad = \quad \frac{sum\ l}{count\ l} \tag{44}$$

Clearly, you need to visit the input list l twice, one for computing the sum of all integers and the other for knowing how many there are. *Banana-split* is known as a corollary of the mutual recursion law which enables one to merge *both* visits into a single one by keeping both values (current sum and current count) in a pair.

From the results of section 8 one cannot take *banana-split* for granted in presence of faults, as mutual-recursion does not hold in general. Let us start with an example: we inject faults in (44) by defining

$$favg_{p,q} = fsum_p \vartriangle fcount_q$$

for $fcount_q$ as before and

$$fsum_p = \mathsf{fold}\ (uncurry\ fadd_p)\ \underline{0}$$

a (faulty) list sum function.[21] For instance, the outcome

```
Main> favg 0.15 0.1 [2,3]
(5,2)  58.5%
(5,1)  13.0%
(2,2)  10.3%
(3,2)  10.3%
(2,1)   2.3%
(3,1)   2.3%
(0,2)   1.8%
(5,0)   0.7%
(0,1)   0.4%
(2,0)   0.1%
(3,0)   0.1%
(0,0)   0.0%
```

[21] We focus on computing the pair of values of (44), leaving aside the final division and the problem of the divisions by zero which arise from faulty counting, to be handled by raising exceptions as in e.g. (Oliveira, 2014).

will lead to the correct average $2.5 = \frac{5}{2}$ with 58.5% probability, the wrong average of 5 with 13.0% probability and so on and so forth.

By application of *banana split* (details below) we transform $favg_{p,q}$ into a single fold on total/count pairs (t, c),

$$
\begin{array}{l}
favgbs_{p,q} = \mathsf{fold}\ body\ \underline{(0,0)}\ \mathbf{where} \\
\quad body\ (a,(t,c)) = \mathbf{do}\ \{ t' \leftarrow fadd_p\ a\ t; c' \leftarrow (id\ {}_q\!\diamond \mathsf{succ})\ c; return\ (t',c') \}
\end{array}
$$

which happens to yield the same output for the same arguments.

Perhaps the run above is not a good choice after all for showing some possible discrepancy between the two versions of the code, before and after *banana split* — one would say. It turns out that further experiments won't succeed in finding a run discriminating both solutions, as these will remain probabilistically indistinguishable.

We show below that this happens because the *banana split* program transformation law *does* hold probabilistically, independently of mutual recursion. To give a single proof covering for-loops and folds on lists as special cases, we generalize both (11) and (41) to

$$k = \llparenthesis f \rrparenthesis \quad \equiv \quad k \cdot \mathsf{in} = f \cdot (\mathsf{F}\ k) \tag{45}$$

where f is a suitably typed probabilistic function and the customary *banana* brackets $\llparenthesis \_ \rrparenthesis$ are used to denote such a generic fold, or *catamorphism*. Functor F is allowed to range over so called *polynomial* or *shapely* functors involving finite products and sums (Hasuo et al., 2007). Instances $\mathsf{F}\ X = id \oplus X$ and $\mathsf{F}\ X = id \oplus id \otimes X$ give us back for-loops and list folds, respectively. Cancellation

$$\llparenthesis f \rrparenthesis \cdot \mathsf{in} = f \cdot \mathsf{F}\ \llparenthesis f \rrparenthesis \tag{46}$$

follows trivially from (45).

Theorem 1 (Probabilistic 'banana-split'). *Transformation*

$$\llparenthesis f \rrparenthesis \vartriangle \llparenthesis g \rrparenthesis \quad = \quad \llparenthesis (f \otimes g) \cdot \mathsf{unzip}_{\mathsf{F}} \rrparenthesis \tag{47}$$

where

$$\mathsf{unzip}_{\mathsf{F}} = \mathsf{F}\ fst \vartriangle \mathsf{F}\ snd \tag{48}$$

holds for f and g probabilistic and for all functors F over which $\mathsf{unzip}_{\mathsf{F}}$ is natural:

$$(\mathsf{F}\ f \otimes \mathsf{F}\ g) \cdot \mathsf{unzip}_{\mathsf{F}} \quad = \quad \mathsf{unzip}_{\mathsf{F}} \cdot \mathsf{F}\ (f \otimes g) \tag{49}$$

Proof: *Relying on* absorption law *(16) we proceed by cata-universality, by solving for f the right hand side equation of (45), once k is instantiated to $k = \llparenthesis f \rrparenthesis \vartriangle \llparenthesis g \rrparenthesis$:*

$$
\begin{array}{cl}
 & (\llparenthesis f \rrparenthesis \vartriangle \llparenthesis g \rrparenthesis) \cdot \mathsf{in} \\
= & \quad \{ \text{ as } \mathsf{in} \text{ is a sharp function, pair-fusion holds (A.1) } \} \\
 & (\llparenthesis f \rrparenthesis \cdot \mathsf{in}) \vartriangle (\llparenthesis g \rrparenthesis \cdot \mathsf{in}) \\
= & \quad \{ \text{ two cancellations (46) } \}
\end{array}
$$

$$
\begin{array}{ll}
& (f \cdot \mathsf{F}\ \llparenthesis f \rrparenthesis) \vartriangle (g \cdot \mathsf{F}\ \llparenthesis g \rrparenthesis) \\
= & \qquad \{\ \textit{pairing-absorption (16)}\ \} \\
& (f \otimes g) \cdot (\mathsf{F}\ \llparenthesis f \rrparenthesis \vartriangle \mathsf{F}\ \llparenthesis g \rrparenthesis) \\
= & \qquad \{\ \textit{(50) below}\ \} \\
& (f \otimes g) \cdot \mathsf{unzip}_{\mathsf{F}} \cdot (\mathsf{F}\ (\llparenthesis f \rrparenthesis \vartriangle \llparenthesis g \rrparenthesis))
\end{array}
$$

□

Thus (47) holds, by (45). As shown in the appendix, fact

$$\mathsf{unzip}_{\mathsf{F}} \cdot \mathsf{F}\ (f \vartriangle g) \quad = \quad \mathsf{F}\ f \vartriangle \mathsf{F}\ g \tag{50}$$

used in the proof is an immediate corollary of the naturality (49) of $\mathsf{unzip}_{\mathsf{F}}$*. The following diagram of (50) may help in understanding its meaning:*

$$
\begin{tikzcd}
\mathsf{F}\ A & (\mathsf{F}\ A) \otimes (\mathsf{F}\ B) \arrow[l, "fst"'] \arrow[r, "snd"] & \mathsf{F}\ B \\
& \mathsf{F}\ (A \otimes B) \arrow[ul, "\mathsf{F}\ fst"] \arrow[u, "\mathsf{unzip}_{\mathsf{F}}"] \arrow[ur, "\mathsf{F}\ snd"'] & \\
& \mathsf{F}\ C \arrow[uul, "\mathsf{F}\ f"] \arrow[u, "\mathsf{F}\ f \vartriangle g"] \arrow[uur, "\mathsf{F}\ g"'] &
\end{tikzcd}
$$

□

In the appendix we show that *shapely* functors (including those which support folds and for-loops) are such that (49) holds, thus granting "banana-split" (47) for a wide range of programming schemes.

In retrospect, note how law (47) was proved not as a corollary of mutual recursion but as an *independent* result. Also note the major role of function $\mathsf{unzip}_{\mathsf{F}}$ (48) in each inductive step: it separates that part of the output which is to be fed to f from that to be fed to g. It is this separation which grants *non-interference* between both computations, as happened in the *square* example but not in *Fibonacci* example, as we have seen.

For completeness, we state the (conditioned) mutual recursion law in a similar generic setting:

Theorem 2 (Probabilistic mutual-recursion). *Transformation*

$$\left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\ (f \vartriangle g) \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\ (f \vartriangle g) \end{array} \right. \quad \equiv \quad f \vartriangle g = \llparenthesis h \vartriangle k \rrparenthesis \tag{51}$$

*holds **provided** one of probabilistic f or g is sharp.*
Proof: *generalize the rationale of section 6 from* for*-loops to* F*-catamorphisms. Typically, for one such function, say f, to be sharp, it has to be independent of the other (say g), assumed truly probabilistic. This means that $h \cdot \mathsf{F}\ (f \vartriangle g) = h' \cdot (\mathsf{G}\ f)$, for some h' and* G.
□

11. Conclusions

The production of *safety critical* software is bound to a number of certification standards in which estimating the *risk of failure* plays a central role. NASA's procedures guide for *probabilistic risk assessment* (PRA) reviews the historical background of risk analysis, evolving from a qualitative to a quantitative perspective of risk (Stamatelatos and Dezfuli, 2011). The UK MoD Defence Standard 00-56 (MoD, 2007) enforces that *all (...) calculations underpinning the risk estimation* be recorded in so-called *safety cases* (documents supporting the claim that some given software is safe) *such that the risk estimates can be reviewed and reconstructed.*

Risk estimation seems to live outside programmers' core practice: either the software system once completed is subject (by others) to intensive simulation over faults injected into safety-critical parts, or the estimation proceeds by analysis of worst case scenarios on a large-scale view of the system's operation.

Software development and risk analysis are performed separately because programming language semantics are (in general) *qualitative* and risk estimation calls for *quantitative* semantic models such as those already prominent in security (McIver and Morgan, 2005). Quantitative methods face another problem, diagnosed by Morgan (2012): probability theory is too descriptive and not fit enough for calculation as this is understood in today's research in program correctness.

In this paper we propose that risk calculation be constructively handled in the programming process since the early stages, rather than being an *a posteriori* concern. This means that risk is taken into account as the "normal" situation, absence of risk being an ideal case. In particular, operations are modelled as probabilistic choice between expected behaviour and faulty behaviour.

Functional programming appears to be particularly apt for this purpose because of its strong mathematical basis. The obstacles mentioned above are circumvented by adopting a linear algebra approach to probability calculation (Oliveira, 2012), a strategy which fits into the calculational style of functional program development based on its algebra of programming (Bird and de Moor, 1997).

This puts functional programming in the forefront of risk estimation simply by exploring the adjunction between distribution-valued functions and matrices of probabilities. One side of the adjunction is "good for programming": the *monadic* one, as we have shown by our experiments in Haskell; the other side (linear algebra) is "good for calculation".

This does not prevent one from actually running case studies in a matrix-speaking language such as e.g. MATLAB. Interestingly, we have observed that, although using MATLAB for the purposes of this paper may seem a "tour de force" (since it is poorly typed and not polymorphic, calling for explicit type error checking in the old style), MATLAB tends to perform faster than Haskell when the probabilistic monadic calculations involve distributions of wider support.[22]

[22] All experiments reported in the current paper can be reproduced by downloading the Haskell and MATLAB sources available from `http://wiki.di.uminho.pt/twiki/bin/view/Research/QAIS/WorkBench`. The PFP library is credited to Erwig and Kollmansberger (2006).

The core of this paper shows how to calculate the propagation of faults across standard program transformation techniques known as *tupling* (Hu et al., 1997) and *fusion* (Harper, 2011). This enables one to find conditions for the *fault of the whole* to be expressed in terms of the *faults of its parts* — a *compositional* approach to risk calculation.

12. Related and future work

Quantitative program analysis. Program analysis techniques based on languages such as e.g. Rely (Carbin et al., 2013) evaluate quantitative reliability of computations running on unreliable hardware, e.g. unreliable arithmetic/logical operations (as in the current paper) or unreliable physical memories. Rely's analysis generates *reliability pre-conditions* which are handled by *reliability transformers*, bridging to current work on probabilistic Hoare logic (Barthe et al., 2012).

The work by Pierro et al. (2010) is closer to ours in its adoption of (untyped) linear algebra in the compositional construction of a so-called *linear operator semantics*, leading to probabilistic program analysis inspired by classical *abstract interpretation*. As in our setting, the key element in the construction is the use of tensor products to capture different aspects of a program.

Link to categorial physics. On the foundations side, probabilistic *weak* tupling has been addressed in the more wide setting of *monoidal* categories adopted in e.g. categorial quantum physics (Coecke, 2011). These include not only $FdHilb$, the category of finite dimensional Hilbert spaces, but also Rel, the category of binary relations. We hope to exploit this connection in the future, in particular concerning partial orders defined for quantum states which could be used to support a notion of refinement.

Linear algebra of programming. Both (Oliveira, 2012) and the current paper are concerned with probabilistic catamorphisms. In this respect, the main novelty of this paper compared to (Oliveira, 2012) is the study of *probabilistic mutual-recursion*.

We would like to find side-conditions for Theorem 2 weaker than that imposing one function to be sharp. Interestingly, this seems to link to work by Wong and Butz (2000) on another topic: Bayesian embedded multivalued dependencies as necessary and sufficient conditions for lossless decomposition of probabilistic relations. For this we also hope to be able to generalize some previous work in this field (Oliveira, 2011).

Future work should extend the current results to probabilistic algorithmic control, including non-termination. This corresponds to studying probabilistic *hylomorphisms*, the most generic pattern of recursion, requiring *sub-distributions* as in (Hasuo et al., 2007).

Refinement. Our experiments in probabilistic mutual recursion show that linear versions consistently score better than the recursive. This conforms to intuition, as program optimization leads to less computations and therefore to lesser propagation of faults. We would like to *quantify* such a difference in probabilistic behaviour. In general, one may think of ordering fault-injected functions with respect to some expected, sharp function. Let $f : A \to B$ be such a function and $g, h : A \to B$ be probabilistic approximations to it, all represented as CS-matrices. Then g and h can be compared against f as follows,

$$g \leqslant_f h \quad \textit{iff} \quad g \times f \leqslant h \times f$$

where $M \times N$ denotes the Hadamard (entry-wise) product of matrices M and N. That is, for each a, we compare the probability which g and h offer for the correct value $f\ a$. Of course, $g \leqslant_f f$ always holds, that is, f is the best approximation to itself. The question is — how effective is it to calculate with this preorder? Is the difference $h \times f - g \times f$ a metric suitable for quantifying fault propagation across correctness-preserving program transformations?

Monadic probabilism. In a real setting, software designers might be more concerned with correct/incorrect results and not so much with the probabilities of specific, incorrect results. This can be approached by functions of type $A \to Dist\ (1 + B)$, where 1 means "incorrect". Type $1 + B$ is monadic, in the sense that incorrect results cannot grant correct results anymore. Functions of the above type are addressed in linear algebra in (Oliveira, 2014).

Note that our linear algebra semantics for functional programs assume *strict* evaluation. Whether the results presented are still valid for *lazy* evaluation needs investigation. This could be done *stacking* a suitable monad as in (Oliveira, 2014) and investigating its *lifting* through $Dist$, relating to another follow-up of the strategy put forward in this paper: its application to fault-propagation in component-oriented software systems. Cortellessa and Grassi (2007) quantify component-to-component error propagation in terms of a matrix which emulates a probabilistic *call-graph*. We are currently working on a formal alternative to this approach in which components represented by *coalgebras* (Barbosa, 2003) extended probabilistically, by adding to the coalgebraic matrices of (Oliveira, 2013) a *behaviour* monad inside the *distribution* one.

Applications. On the applications side, we plan to address case studies such as that of (Marić and Sprenger, 2014) — the verification of a persistent memory manager (in IBM's 4765 secure coprocessor) in face of restarts and hardware failures — using probabilistic linear algebra. The work will consist in modelling the device functionally and carrying out proofs using matrix algebra where Marić and Sprenger (2014) use explicit monad transformers in Isabelle. As the authors of this paper write, the inclusion of hardware failures incurs a significant jump in system complexity.

Altogether, we hope to show that the linear algebra of programming is a wide-range formalism suitable to generically support quantitative methods in the software sciences.

Acknowledgements

This work is financed by the *FCT – Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013.

Part of the research was carried out in the QAIS (Quantitative analysis of interacting systems) 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.

José Oliveira wishes to thank CSW Critical Software SA for their invitation to the final workshop of FP7 project CriticalStep (http://www.critical-step.eu) — WS on Dependability and Certification — where the central idea of this paper was briefly presented.

While doing this work Daniel Murta was supported by a grant[23] awarded by FCT.

References

S. Andova, A. McIver, P. R. D'Argenio, P. J. L. Cuijpers, J. Markovski, C. Morgan, and M. Núñez, editors. *Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications*, volume 13 of *EPTCS*, 2009.

R.C. Backhouse and D. Michaelis. Exercises in quantifier manipulation. In T. Uustalu, editor, *MPC'06*, volume 4014 of *LNCS*, pages 70–81. Springer, 2006.

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

G. Barthe, B. Grégoire, and S.Z. Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *MPC'12*, pages 1–6, 2012.

R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.

M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware, 2013. 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA/SPLASH 2013), Indianapolis, IN, USA, October 2013.

B. Coecke, editor. *New Structures for Physics*. Number 831 in Lecture Notes in Physics. Springer, 2011. doi: 10.1007/978-3-642-12821-9.

V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *Component-Based Software Engineering*, volume 4608 of *LNCS*, pages 140–156. 2007.

J.A. Durães and H.S. Madeira. Emulation of software faults: a field data study and a practical approach, 2006. IEEE Transactions on Software Engineering.

M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

M.F. Frias, G. Baum, and A.M. Haeberer. Fork algebras in algebra, logic and computer science. *Fundam. Inform.*, pages 1–25, 1997.

[23]Reference: `BI1-2012_PTDC/EIA-CCO/122240/2010_UMINHO`

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

T. Harper. A library writer's guide to shortcut fusion. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 47–58, New York, NY, USA, 2011. ACM.

I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4):1–36, 2007. doi: 10.2168/LMCS-3(4:11)2007.

Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *In ACM SIGPLAN International Conference on Functional Programming*, pages 164–175. ACM Press, 1997.

D. Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, 2009.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003. ISBN 9780521826143. doi: DOI:10.2277/0521826144.

A. Lingamneni, C. Enz, K. Palem, and C. Piguet. Synthesizing parsimonious inexact circuits through probabilistic design techniques. *ACM Trans. Embed. Comput. Syst.*, 12(2s):93:1–93:26, May 2013. ISSN 1539-9087. doi: 10.1145/2465787.2465795.

H. Macedo. *Matrices as Arrows — Why Categories of Matrices Matter*. PhD thesis, University of Minho, October 2012. MAPi PhD programme.

H.D. Macedo and J.N. Oliveira. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming*, 78(11):2160–2191, 2013. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2012.07.012.

O. Marić and C. Sprenger. Verification of a transactional memory manager under hardware failures and restarts. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *LNCS*, pages 449–464. Springer, 2014. ISBN 978-3-319-06409-3.

S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013. ISBN 1449335942.

A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer-Verlag, 2005. ISBN 0387401156.

UK MoD. Safety management requirements for defence systems: Part 1 requirements, 2007. UK MoD Defence Standard 00-56. `http://www.dstan.mod.uk/standards/defstans/00/056/01000400.pdf`.

C. Morgan. Elementary probability theory in the Eindhoven style. In *MPC*, LNCS, pages 48–73, 2012.

J.N. Oliveira. Pointfree foundations for (generic) lossless decomposition. Technical Report TR-HASLab:3:2011, HASLab/INESC TEC & U. Minho, 2011. URL `https://repositorium.sdum.uminho.pt/handle/1822/24648`.

J.N. Oliveira. Towards a linear algebra of programming. *Formal Aspects of Computing*, 24(4-6):433–458, 2012. doi: 10.1007/s00165-012-0240-9.

J.N. Oliveira. Weighted automata as coalgebras in categories of matrices. *Int. Journal of Found. of Comp. Science*, 24(06):709–728, 2013. doi: 10.1142/S0129054113400145.

J.N. Oliveira. Relational algebra for "just good enough" hardware. In *RAMiCS*, volume 8428 of *LNCS*, pages 119–138. Springer Berlin, Heidelberg, 2014.

Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic semantics and program analysis. In *SFM*, pages 1–42, 2010.

G. Schmidt. *Relational Mathematics*. Number 132 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, November 2010. ISBN 9780521762687.

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

J. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley & Sons, 1997. ISBN ISBN 0-471-18381-4. 416 pages.

S.K.M. Wong and C.J. Butz. The implication of probabilistic conditional independence and embedded multivalued dependency. In *IPMU00*, pages 876–881, 2000. 8th Conf. on Inf. Processing and Management of Uncertainty in K.-B. Systems.

Appendix A. Proofs of auxiliary results

Proof of cancellation (24).

$$
\begin{array}{ll}
 & fst \cdot (f \vartriangle g) = f \ \wedge \ snd \cdot (f \vartriangle g) = g \\
\equiv & \quad \{ \ fst = id \otimes ! \text{ and } snd = ! \otimes id \ (25) \ \} \\
 & (id \otimes !) \cdot (f \vartriangle g) = f \ \wedge \ (! \otimes id) \cdot (f \vartriangle g) = g \\
\equiv & \quad \{ \ \text{absorption (16) twice} \ \} \\
 & f \vartriangle (! \cdot g) = f \ \wedge \ (! \cdot f) \vartriangle g = g \\
\equiv & \quad \{ \ ! \cdot k = ! \text{ for any probabilistic function } k \text{ (Oliveira, 2012)} \ \} \\
 & f \vartriangle ! = f \ \wedge \ ! \vartriangle g = g \\
\equiv & \quad \{ \ ! \text{ is the unit of Khatri-Rao (26)} \ \} \\
 & f = f \ \wedge \ g = g
\end{array}
$$

□

Proof of (32). This equality arises from rule (30):

$$
\begin{aligned}
& \langle \textstyle\sum\ c\ ::\ (f\ a, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{ \text{ one-point rule } \} \\
& \langle \textstyle\sum\ b, c\ :\ f\ a = b :\ (b, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{\ b = \mathit{fst}\ (b, c)\ ;\ (30)\ \} \\
& (f\ a)\ (\mathit{fst} \cdot k)\ a = 1 \\
\equiv \quad & \quad \{\ f = \mathit{fst} \cdot k\ \} \\
& (f\ a)\ f\ a = 1 \\
\equiv \quad & \quad \{\ f \text{ is sharp } \} \\
& \mathit{true}
\end{aligned}
$$

$\square$

Proof of (33). This equality arises from k being probabilistic:

$$
\begin{aligned}
& \langle \textstyle\sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 0 \\
\equiv \quad & \quad \{\ 1 + 0 = 1\ \} \\
& 1 + \langle \textstyle\sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{\ (32)\ \} \\
& \langle \textstyle\sum\ c\ ::\ (f\ a, c)\ k\ a \rangle + \langle \textstyle\sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{ \text{ one-point rule } \} \\
& \langle \textstyle\sum\ b, c\ :\ b = f\ a :\ (b, c)\ k\ a \rangle + \langle \textstyle\sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{ \text{ merge quantifiers } \} \\
& \langle \textstyle\sum\ (b, c)\ ::\ (b, c)\ k\ a \rangle = 1 \\
\equiv \quad & \quad \{\ k \text{ is probabilistic } \} \\
& \mathit{true}
\end{aligned}
$$

$\square$

Proof of base-case fault propagation (35). Clearly, by (37) and universal property (36), our goal (35) re-writes to the equality

$$
\begin{aligned}
& ((\mathsf{for}\ f\ a)\ {}_p\!\diamond (\mathsf{for}\ f\ b)) \cdot \mathsf{in} = \\
& \qquad\qquad [\underline{a}\ {}_p\!\diamond\ \underline{b} \,|\, f] \cdot (\mathsf{F}\ ((\mathsf{for}\ f\ a)\ {}_p\!\diamond (\mathsf{for}\ f\ b)))
\end{aligned}
$$

which holds by transforming the left-hand side into the right-hand side:

$$((\mathsf{for}\ f\ a)\ {}_{p}{\diamond}\ (\mathsf{for}\ f\ b)) \cdot \mathsf{in}$$

$$= \quad \{\ \text{choice-fusion (38)}\ \}$$

$$(\mathsf{for}\ f\ a \cdot \mathsf{in})\ {}_{p}{\diamond}\ (\mathsf{for}\ f\ b \cdot \mathsf{in})$$

$$= \quad \{\ \text{(37) and (36), twice}\ \}$$

$$([\underline{a} \,|\, f] \cdot \mathsf{F}\ (\mathsf{for}\ f\ a))\ {}_{p}{\diamond}\ ([\underline{b} \,|\, f] \cdot \mathsf{F}\ (\mathsf{for}\ f\ b)))$$

$$= \quad \{\ \mathsf{F}\, f = id \oplus f\ \text{; absorption: } [M \,|\, N] \cdot (P \oplus Q) = [M \cdot P \,|\, N \cdot Q]\ \}$$

$$[\underline{a} \,|\, f \cdot (\mathsf{for}\ f\ a)]\ {}_{p}{\diamond}\ [\underline{b} \,|\, f \cdot (\mathsf{for}\ f\ b)]$$

$$= \quad \{\ \text{exchange law (40)}\ \}$$

$$[\underline{a}\ {}_{p}{\diamond}\ \underline{b} \,|\, (f \cdot \mathsf{for}\ f\ a)\ {}_{p}{\diamond}\ (f \cdot \mathsf{for}\ f\ b)]$$

$$= \quad \{\ \text{choice-fusion (39)}\ \}$$

$$[\underline{a}\ {}_{p}{\diamond}\ \underline{b} \,|\, f \cdot ((\mathsf{for}\ f\ a)\ {}_{p}{\diamond}\ (\mathsf{for}\ f\ b))]$$

$$= \quad \{\ \text{again absorption: } [M \,|\, N] \cdot (P \oplus Q) = [M \cdot P \,|\, N \cdot Q]\ \}$$

$$[\underline{a}\ {}_{p}{\diamond}\ \underline{b} \,|\, f] \cdot (id \oplus ((\mathsf{for}\ f\ a)\ {}_{p}{\diamond}\ (\mathsf{for}\ f\ b)))$$

$$= \quad \{\ \mathsf{F}\, f = id \oplus f\ \}$$

$$[\underline{a}\ {}_{p}{\diamond}\ \underline{b} \,|\, f] \cdot (\mathsf{F}\ ((\mathsf{for}\ f\ a)\ {}_{p}{\diamond}\ (\mathsf{for}\ f\ b)))$$

$\square$

Proof of Khatri-Rao (conditional) fusion. We want to prove

$$(M \vartriangle N) \cdot h = (M \cdot h) \vartriangle (N \cdot h) \quad \Leftarrow \quad h \text{ is sharp} \tag{A.1}$$

for arbitrary (suitably typed) matrices M and N:

$$(b, c)\ ((M \vartriangle N) \cdot h)\ a$$

$$= \quad \{\ \text{(31) for } h \text{ a standard function}\ \}$$

$$(b, c)\ (M \vartriangle N)\ (h\ a)$$

$$= \quad \{\ \text{pointwise Khatri-Rao (14)}\ \}$$

$$(b\ M\ (h\ a)) \times (c\ N\ (h\ a))$$

$$= \quad \{\ \text{(31) for } h \text{ a standard function}\ \}$$

$$b\ (M \cdot h)\ a \times c\ (N \cdot h)\ a$$

$$= \quad \{\ \text{pointwise Khatri Rao (14) — twice}\ \}$$

$$(b, c)\ (M \cdot h \vartriangle N \cdot h)\ a$$

$\square$

Proofs concerning naturality of $\mathsf{unzip}_\mathsf{F}$ *(49).* This section shows that $\mathsf{unzip}_\mathsf{F}$ is natural for so-called *polynomial* or *shapely* functors. This is proved by induction on the structure of such functors.

The property holds trivially for the identity functor $\mathsf{F}\ X = X$, where $\mathsf{unzip}_\mathsf{F} = id$, and for any constant functor $\mathsf{F}\ X = K$, in which case $\mathsf{unzip}_\mathsf{F} = id \vartriangle id$. Next, we show that the property is structurally preserved by functor composition, say $\mathsf{F} = \mathsf{G}\ \mathsf{H}$, whereby

$$\mathsf{unzip}_{\mathsf{GH}} \quad = \quad \mathsf{unzip}_\mathsf{G} \cdot (\mathsf{G}\ \mathsf{unzip}_\mathsf{H}) \tag{A.2}$$

holds by pair-fusion (A.1), cf. the sharp right term. In this and the remaining calculations we generalize probabilistic functions f and g in (49) to arbitrary matrices M, N over a semiring. We have:

$$\begin{array}{cl}
& \mathsf{unzip}_{\mathsf{GH}} \cdot \mathsf{G}\ \mathsf{H}\ (M \otimes N) \\
= & \qquad \{\ \text{(A.2)}\ \} \\
& \mathsf{unzip}_\mathsf{G} \cdot (\mathsf{G}\ \mathsf{unzip}_\mathsf{H}) \cdot \mathsf{G}\ (\mathsf{H}\ (M \otimes N)) \\
= & \qquad \{\ \text{functor } \mathsf{G} \text{ (composition)}\ \} \\
& \mathsf{unzip}_\mathsf{G} \cdot \mathsf{G}\ (\mathsf{unzip}_\mathsf{H} \cdot \mathsf{H}\ (M \otimes N)) \\
= & \qquad \{\ \text{induction hypothesis: assume (49) for } \mathsf{F} = \mathsf{H}; \mathsf{G} \text{ again}\ \} \\
& \mathsf{unzip}_\mathsf{G} \cdot \mathsf{G}\ ((\mathsf{H}\ M) \otimes (\mathsf{H}\ N)) \cdot (\mathsf{G}\ \mathsf{unzip}_\mathsf{H}) \\
= & \qquad \{\ \text{induction hypothesis: assume (49) for } \mathsf{F} = \mathsf{G}\ \} \\
& ((\mathsf{G}\ \mathsf{H}\ M) \otimes (\mathsf{G}\ \mathsf{H}\ N)) \cdot \mathsf{unzip}_\mathsf{G} \cdot \mathsf{G}\ (\mathsf{unzip}_\mathsf{H}) \\
= & \qquad \{\ \text{(A.2)}\ \} \\
& ((\mathsf{G}\ \mathsf{H}\ M) \otimes (\mathsf{G}\ \mathsf{H}\ N)) \cdot \mathsf{unzip}_{\mathsf{GH}}
\end{array}$$

□

Next, we do the same for sums, say $\mathsf{F} = \mathsf{G} \oplus \mathsf{H}$. In this case we have:

$$\mathsf{unzip}_\mathsf{F} \quad = \quad (\mathsf{G}\ fst \oplus \mathsf{H}\ fst) \vartriangle (\mathsf{G}\ snd \oplus \mathsf{H}\ snd) \tag{A.3}$$

Facts

$$\mathsf{unzip}_\mathsf{F} \cdot i_1 \quad = \quad (i_1 \otimes i_1) \cdot \mathsf{unzip}_\mathsf{G} \tag{A.4}$$

$$\mathsf{unzip}_\mathsf{F} \cdot i_2 \quad = \quad (i_2 \otimes i_2) \cdot \mathsf{unzip}_\mathsf{H} \tag{A.5}$$

are easy to prove via exchange law (18), where i_1 and i_2 are the injections of the direct sum, that is $[i_1 \,|\, i_2] = id$. The same law also grants equality

$$\begin{array}{cl}
& [(i_1 \otimes i_1) \cdot (M \vartriangle N) \,|\, (i_2 \otimes i_2) \cdot (P \vartriangle Q)] \\
= & (M \oplus P) \vartriangle (N \oplus Q)
\end{array} \tag{A.6}$$

which is valid for all suitably typed matrices M, N, P and Q, and will help in the proof that (49) holds for sums of functors which (inductively) satisfy the same property:

$$
\begin{array}{ll}
& \mathsf{unzip_F} \cdot \mathsf{F}\ (M \otimes N) \\
\equiv & \quad \{\ \mathsf{F} = \mathsf{G} \oplus \mathsf{H}\ \} \\
& \mathsf{unzip_F} \cdot ((\mathsf{G}\ (M \otimes N)) \oplus (\mathsf{H}\ (M \otimes N))) \\
= & \quad \{\ M \oplus N = [i_1 \cdot M \,|\, i_2 \cdot N]\ ;\ \text{fusion (7)}\ \} \\
& [\mathsf{unzip_F} \cdot i_1 \cdot (\mathsf{G}\ (M \otimes N)) \,|\, \mathsf{unzip_F} \cdot i_2 \cdot (\mathsf{H}\ (M \otimes N))] \\
= & \quad \{\ \text{(A.4,A.5)}\ \} \\
& [(i_1 \otimes i_1) \cdot \mathsf{unzip_G} \cdot (\mathsf{G}\ (M \otimes N)) \,|\, (i_2 \otimes i_2) \cdot \mathsf{unzip_H} \cdot (\mathsf{H}\ (M \otimes N))] \\
= & \quad \{\ \text{induction hypothesis: assume (49) for } \mathsf{F} = \mathsf{G} \text{ and } \mathsf{F} = \mathsf{H}\ \} \\
& [(i_1 \otimes i_1) \cdot (\mathsf{G}\ M \otimes \mathsf{G}\ N) \cdot \mathsf{unzip_G} \,|\, (i_2 \otimes i_2) \cdot (\mathsf{H}\ M \otimes \mathsf{H}\ N) \cdot \mathsf{unzip_H}] \\
\equiv & \quad \{\ \text{definitions of } \mathsf{unzip_G} \text{ and } \mathsf{unzip_H}\ ;\ \text{absorptions}\ \} \\
& [(i_1 \otimes i_1) \cdot \mathsf{G}\ (M \cdot \mathit{fst}) \vartriangle \mathsf{G}\ (N \cdot \mathit{snd}) \,|\, (i_2 \otimes i_2) \cdot \mathsf{H}\ (M \cdot \mathit{fst}) \otimes \mathsf{H}\ (N \cdot \mathit{snd})] \\
= & \quad \{\ \text{(A.6)}\ \} \\
& (\mathsf{G}\ (M \cdot \mathit{fst})) \oplus (\mathsf{H}\ (M \cdot \mathit{fst})) \vartriangle (\mathsf{G}\ (N \cdot \mathit{snd})) \oplus (\mathsf{H}\ (N \cdot \mathit{snd})) \\
\equiv & \quad \{\ \mathsf{F} = \mathsf{G} \oplus \mathsf{H}\ \} \\
& \mathsf{F}\ (M \cdot \mathit{fst}) \vartriangle \mathsf{F}\ (N \cdot \mathit{snd}) \\
\equiv & \quad \{\ \text{functor } \mathsf{F}\ ;\ \text{reverse absorption}\ \} \\
& (\mathsf{F}\ M \otimes \mathsf{F}\ N) \cdot (\mathsf{F}\ \mathit{fst} \vartriangle \mathsf{F}\ \mathit{snd}) \\
\equiv & \quad \{\ \text{definition of } \mathsf{unzip_F}\ \} \\
& (\mathsf{F}\ M \otimes \mathsf{F}\ N) \cdot \mathsf{unzip_F} \\
\Box &
\end{array}
$$

Finally, we do the proof for products, say $\mathsf{F} = \mathsf{G} \otimes \mathsf{H}$ where

$$(\mathsf{G} \otimes \mathsf{H})\ M = \mathsf{G}\ M \otimes \mathsf{H}\ M \tag{A.7}$$

In this case we have

$$\mathsf{unzip_{G \otimes H}} \quad = \quad \mathsf{tr} \cdot (\mathsf{unzip_G} \otimes \mathsf{unzip_H}) \tag{A.8}$$

where tr is the natural isomorphism

$$
\begin{array}{l}
\mathsf{tr} : ((B \times C) \times (D \times A)) \to ((B \times D) \times (C \times A)) \\
\mathsf{tr} = (\mathit{fst} \otimes \mathit{fst}) \vartriangle (\mathit{snd} \otimes \mathit{snd})
\end{array} \tag{A.9}
$$

that is,

$$\mathsf{tr} \cdot ((N \otimes P) \otimes (Q \otimes M)) = ((N \otimes Q) \otimes (P \otimes M)) \cdot \mathsf{tr} \tag{A.10}$$

holds. The proof of the naturality of $\mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}}$ follows:

$$
\begin{array}{ll}
& \mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}} \cdot (\mathsf{G}\otimes\mathsf{H})\ (M\otimes N) \\
= & \quad \{\ \text{definition of product functor (A.7)}\ \} \\
& \mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}} \cdot (\mathsf{G}\ (M\otimes N)\otimes \mathsf{H}\ (M\otimes N)) \\
= & \quad \{\ \text{inline definition of } \mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}} \text{ (A.9) ; Kronecker bifunctor}\ \} \\
& \mathsf{tr}\cdot(\mathsf{unzip}_{\mathsf{G}}\cdot \mathsf{G}\ (M\otimes N)\otimes(\mathsf{unzip}_{\mathsf{H}}\cdot\mathsf{H}\ (M\otimes N))) \\
= & \quad \{\ \text{naturality of } \mathsf{unzip}_{\mathsf{G}} \text{ and } \mathsf{unzip}_{\mathsf{H}} \text{ assumed (inductive step)}\ \} \\
& \mathsf{tr}\cdot((\mathsf{G}\ M\otimes\mathsf{G}\ N)\cdot\mathsf{unzip}_{\mathsf{G}}\otimes(\mathsf{H}\ M\otimes\mathsf{H}\ N)\cdot\mathsf{unzip}_{\mathsf{H}}) \\
= & \quad \{\ \text{Kronecker bifunctor}\ \} \\
& \mathsf{tr}\cdot((\mathsf{G}\ M\otimes\mathsf{G}\ N)\otimes(\mathsf{H}\ M\otimes\mathsf{H}\ N))\cdot(\mathsf{unzip}_{\mathsf{G}}\otimes\mathsf{unzip}_{\mathsf{H}}) \\
= & \quad \{\ \text{(A.10)}\ \} \\
& ((\mathsf{G}\ M\otimes\mathsf{H}\ M)\otimes(\mathsf{G}\ N\otimes\mathsf{H}\ N))\cdot\mathsf{tr}\cdot(\mathsf{unzip}_{\mathsf{G}}\otimes\mathsf{unzip}_{\mathsf{H}}) \\
= & \quad \{\ \text{fold over definitions of } \mathsf{G}\otimes\mathsf{H} \text{ and } \mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}}\ \} \\
& ((\mathsf{G}\otimes\mathsf{H})\ M\otimes(\mathsf{G}\otimes\mathsf{H})\ N)\cdot\mathsf{unzip}_{\mathsf{G}\otimes\mathsf{H}}
\end{array}
$$

$\square$

Proof of fact (50) assuming (49).

$$
\begin{array}{ll}
& \mathsf{unzip}_{\mathsf{F}}\cdot\mathsf{F}\ (f \vartriangle g) \\
= & \quad \{\ \text{reverse pairing-absorption (16)}\ \} \\
& \mathsf{unzip}_{\mathsf{F}}\cdot\mathsf{F}\ (f\otimes g)\cdot\mathsf{F}\ (id \vartriangle id) \\
= & \quad \{\ \text{naturality (49) assumed}\ \} \\
& (\mathsf{F}\ f\otimes\mathsf{F}\ g)\cdot\mathsf{unzip}_{\mathsf{F}}\cdot\mathsf{F}\ (id \vartriangle id) \\
= & \quad \{\ \text{functor } \mathsf{F}\text{; } \mathsf{unzip}_{\mathsf{F}} \text{ (48) ; pairing-fusion (A.1), as } id \vartriangle id \text{ is sharp}\ \} \\
& (\mathsf{F}\ f\otimes\mathsf{F}\ g)\cdot\mathsf{F}\ (fst\cdot id \vartriangle id) \vartriangle \mathsf{F}\ (fst\cdot id \vartriangle id) \\
= & \quad \{\ \text{standard pairing-cancellation (24)}\ \} \\
& (\mathsf{F}\ f\otimes\mathsf{F}\ g)\cdot(\mathsf{F}\ id \vartriangle \mathsf{F}\ id) \\
= & \quad \{\ \text{functor } \mathsf{F}\text{; pairing-absorption (16)}\ \} \\
& \mathsf{F}\ f \vartriangle \mathsf{F}\ g
\end{array}
$$

$\square$

Appendix B. On program "monadification"

Wherever one writes "non-monadic" functional programs, for instance the map function in Haskell syntax,

$$
\begin{aligned}
&map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
&map\ f\ [\,] = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

one is actually writing a monadic program for a very special monad: the *identity* one. In this identity monad, $return\ x$ is x — that is, $return = id$ — and monadic composition of functions f and g is nothing but normal composition:

$$\mathbf{do}\ \{b \leftarrow g\ a; f\ b\}\ =\ \mathbf{let}\ b = g\ a\ \mathbf{in}\ f\ b\ =\ f\ (g\ a)\ =\ (f \cdot g)\ a$$

This gives a hint for converting regular programs into monadic ones, the idea being: (a) to make the *identity* monad apparent in the first place, and then (b) to generalize from *identity* to any monad.

Taking the example of map above, the first step is the "refactoring" that introduces explicit identity functions id on the "exit" points of map and makes so-called program evaluation "thunks" (Marlow, 2013) explicit using $\mathbf{let}$ notation:

$$
\begin{aligned}
&map\ f\ [\,] = id\ [\,] \\
&map\ f\ (x : xs) = \mathbf{let} \\
&\quad x' = f\ x \\
&\quad xs' = map\ f\ xs \\
&\quad \mathbf{in}\ id\ (x' : xs')
\end{aligned}
$$

The "monadification" step consists in generalizing from the identity monad to any monad — id generalizes to $return$ and $\mathbf{let}$ generalizes to $\mathbf{do}$:

$$
\begin{aligned}
&mmap :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b] \\
&mmap\ f\ [\,] = return\ [\,] \\
&mmap\ f\ (x : xs) = \mathbf{do}\ \{ \\
&\quad x' \leftarrow f\ x; \\
&\quad xs' \leftarrow mmap\ f\ xs; \\
&\quad return\ (x' : xs') \\
&\quad \}
\end{aligned}
$$

Note the new name $mmap$ standing for "monadic map" and the richer type of $mmap$, parametric on monad m: instantiating this to the identity monad we go back to the type of map wherefrom we have started.