

Calculating risk in functional programming

Daniel Murta^{a,1}, José Nuno Oliveira^{a,*}

^aHASLAB - 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 risk be constructively handled in *functional programming* by (a) writing programs which may choose between *expected* and *faulty* behaviour, and by (b) reasoning about them in a linear algebra extension to standard, *à la* Bird-Moor algebra of programming.

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

1. Introduction

With software so invasive in every-day’s life as it is today, you don’t need to be staff of a space agency to place 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, modeling, and quantifying scenarios that can lead to undesired consequences.

*Corresponding author.

¹Partially supported by *Fundação para a Ciência e a Tecnologia*, Portugal, under grant number B11-2012-PTDC/EIA-CCO/122240/2010-UMINHO.

This may leave one with the feeling that PRA takes place *a posteriori*, that is, once the system is built. Even if a wrong understanding of PRA, 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 (eg. 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, 2012a).

The basic idea 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 $(\cdot \diamond \cdot)$ of McIver and Morgan (2005); Oliveira (2012a), one may write term

$$bad \ _p \diamond \ good$$

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 are such implementations 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 using the PFP library of Erwig and Kollmansberger (2006), or by running finite approximations of them directly as matrices in MATLAB².

Contribution. 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. In this context, this paper presents the following contributions:

- It shows how the standard *algebra of functional programs* dear to the so-called *program transformation* school of software design extends and incorporates risk

²MATLABTM is a trademark of The MathWorks ®.

simply by switching from “*sharp*” functions to *probabilistic* functions handled as matrices in linear algebra.³

- The laws of such a *linear* algebra of programming 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*” functional 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. A basis for this is given in section 5, where the LAoP is put in context, leading to the approach to 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 delved 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 Appendix A.

2. Motivation

Let us start from two programs written in C, one which supposedly computes the square of a non-negative integer n ,

```
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;
};
```

and the other

```
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;
};
```

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

which supposedly computes the n -th entry in the Fibonacci series, for n positive.

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

$$fadd_x = id \text{ }_p\Diamond (x+)$$

(*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

$$fadd_x = \underline{0} \text{ }_q\Diamond (x+)$$

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

$$fadd_x = (\underline{0} \text{ }_q\Diamond id) \text{ }_p\Diamond (x+)$$

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 of Erwig and Kollmansberger (2006), which we shall be using in the sequel.⁴ Moreover,

⁴All distributions in our approach are generated by finite application of the *choice* operator and therefore have finite support.

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 $f : a \rightarrow \text{Dist } b$ 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, 2012a,b). 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 (eg. 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 (2012a) 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 take the standard definition of the Fibonacci function, written in Haskell syntax:

$$\begin{aligned} \text{fib } 0 &= 0 \\ \text{fib } 1 &= 1 \\ \text{fib } (n + 2) &= \text{fib } n + \text{fib } (n + 1) \end{aligned}$$

The linear version encoded in the C program given above is obtained by pairing *fib* with its *derivative*, $f \ n = \text{fib } (n + 1)$:⁵

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

⁵Since $f \ 0 = \text{fib } 1 = 1$ and $f \ (n + 1) = \text{fib } (n + 2) = \text{fib } n + \text{fib } (n + 1) = \text{fib } n + f \ n$.

The pairing of the two functions,

$$(fib, f) n = (fib n, f n)$$

can be expressed primitive-recursively by

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

or by the equivalent

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

itself the same as

$$\begin{aligned}(fib, f) &= \textbf{for loop } (0, 1) \\ &\textbf{ where loop } (x, y) = (y, x + y)\end{aligned}$$

by introduction of the for *loop* combinator,

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

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

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

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

$$\begin{aligned}sq 0 &= 0 \\ sq (n + 1) &= sq n + odd n \\ odd 0 &= 1 \\ odd (n + 1) &= 2 + odd n\end{aligned}$$

arise from the binomial $(n + 1)^2 = n^2 + 2n + 1$ and introduction of function $odd n = 2n + 1$, thus named because $2n + 1$ is the *n*-th odd number. (That is, the square of a natural number always is a sum of odd numbers.) Pairing them up into $(sq, odd) x = (sq x, odd x)$ and proceeding in the same way as above we obtain $(sq, odd) =$

for $loop (0, 1)$ where $loop (s, o) = (s + o, o + 2)$ and thereupon the following functional version of the given C program: ⁶

```

sql n =
  let (s, o) = for loop (0, 1) n
      loop (s, o) = (s + o, o + 2)
  in s

```

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). Take the recursive version of fib given in the beginning of section 3 and “monadify it” into:

```

mfib 0 = return 0
mfib 1 = return 1
mfib (n + 2) =
  do { x ← mfib n; y ← mfib (n + 1); return (x + y) }

```

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, eg. ⁷

```

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

```

We see that the correct behaviour (100% chance of getting $fib 4 = 3$) 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

```

mfibl n =
  do { (x, y) ← mfor loop (0, 1) n; return x }
  where loop (x, y) = return (y, x + y)

```

⁶Notice how the syntax $s+=o; o+=2;$ in C nicely tallies with $(s + o, o + 2)$ in Haskell.

⁷The 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.

relying on the monadic extension of the for combinator:

$$\begin{aligned} mfor\ b\ i\ 0 &= return\ i \\ mfor\ b\ i\ (n + 1) &= \mathbf{do}\ \{x \leftarrow mfor\ b\ i\ n; b\ x\} \end{aligned}$$

To inject into *mfibl* the same fault injected before into *mfib* amounts to replacing, in the loop body, *good* addition by the *bad* one:

$$loop\ (x, y) = \mathbf{do}\ \{z \leftarrow fadd_{0,1}\ x\ y; return\ (y, z)\}$$

Running the same experiment as above we still get *mfibl* 4 = *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

<i>n</i>	<i>mfib</i> <i>n</i>	<i>mfibl</i> <i>n</i>
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%	

the linear version performing better than the recursive one in the sense of hitting the correct answer with higher probability.

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

$$\begin{aligned} msq\ 0 &= return\ 0 \\ msq\ (n + 1) &= \mathbf{do}\ \{m \leftarrow msq\ n; fadd_{0,1}\ m\ (2 * n + 1)\} \end{aligned}$$

and the linear one:

$$\begin{aligned} msq_l\ n &= \\ &\mathbf{do}\ \{(s, o) \leftarrow mfor\ loop\ (0, 1)\ n; return\ s\} \\ &\mathbf{where}\ loop\ (s, o) = \\ &\mathbf{do}\ \{z \leftarrow fadd_{0,1}\ s\ o; return\ (z, o + 2)\} \end{aligned}$$

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	$msq\ n$	$msql\ n$
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%
⋮	⋮	⋮
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%
⋮	⋮	⋮

Summing up, we are in presence of two examples in which the risk of bad behaviour propagates differently across the mutual recursion functional 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, explaining the different behaviour witnessed in the examples above.

5. Probabilistic for-loops in the LAoP

Consider the probabilistic Boolean function $f = \underline{False}_{0.05} \diamond (\neg)$ which is such that $f\ True = False$ (100%) and $f\ False$ 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 ,

$$M = \begin{matrix} & \begin{matrix} False & True \end{matrix} \\ \begin{matrix} False \\ True \end{matrix} & \begin{pmatrix} 0.05 & 1.00 \\ 0.95 & 0.00 \end{pmatrix} \end{matrix} \quad (1)$$

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: $M \cdot id = M = id \cdot M$, where matrix multiplication is denoted by an infix dot (\cdot).

We will write $M : n \rightarrow m$, 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, $f : Bool \rightarrow Dist\ Bool$ is represented by a matrix of type $M : Bool \rightarrow Bool$ (1) on the Kleisli-category side.

Let notation $\llbracket f \rrbracket$ mean the matrix which represents probabilistic function f in such a CS-matrix category. For f of type $A \rightarrow Dist\ B$, $\llbracket f \rrbracket$ will be a matrix of type $A \rightarrow B$, that is, cell $b \llbracket f \rrbracket a$ in the matrix⁸ 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,

$$\llbracket f \bullet g \rrbracket = \llbracket f \rrbracket \cdot \llbracket g \rrbracket \quad (2)$$

and probabilistic function choice is given by

$$\llbracket f\ p \diamond g \rrbracket = p \llbracket f \rrbracket + (1 - p) \llbracket g \rrbracket \quad (3)$$

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, $\llbracket return \rrbracket = id$. Any conventional function $f : A \rightarrow B$ can be turned into a “sharp” probabilistic one through the composition $return \cdot f$ which, represented as a CS-matrix, is the matrix $M = \llbracket return \cdot f \rrbracket$ such that $b\ M\ a = 1$ if $b = f\ a$ and is 0 otherwise.⁹ We will write $\llbracket f \rrbracket$ as shorthand for $\llbracket return \cdot f \rrbracket$ and therefore will rely on fact $(f\ a)\ \llbracket f \rrbracket\ a = 1$, all other cells being 0.

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,

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

and re-write it as follows,

$$\begin{aligned} (\text{for } b\ i) \cdot \underline{0} &= \underline{i} \\ ((\text{for } b\ i) \cdot \text{succ})\ n &= (b \cdot (\text{for } b\ i))\ n \end{aligned}$$

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

⁸Following the infix notation usually adopted for relations (which are Boolean matrices), for instance $y \leq x$, 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, 2012b).

⁹A probabilistic function $f : A \rightarrow Dist\ B$ is said to be *sharp* if, for all $a \in A$, $f\ a$ is a Dirac distribution. A Dirac distribution is one whose support is a singleton set, the unique element of which is offered with 100% probability.

$$\begin{aligned} \llbracket \text{for } b \ i \rrbracket \cdot \llbracket 0 \rrbracket &= \llbracket \dot{z} \rrbracket \\ \llbracket \text{for } b \ i \rrbracket \cdot \llbracket \text{succ} \rrbracket &= \llbracket b \rrbracket \cdot \llbracket \text{for } b \ i \rrbracket \end{aligned}$$

which can be reduced to a single equality,

$$\llbracket \text{for } b \ i \rrbracket \cdot \llbracket \llbracket 0 \rrbracket \mid \llbracket \text{succ} \rrbracket \rrbracket = \llbracket \llbracket \dot{z} \rrbracket \mid (\llbracket b \rrbracket \cdot \llbracket \text{for } b \ i \rrbracket) \rrbracket \quad (4)$$

by resorting to the $[M|N]$ combinator which glues two matrices $M : A \rightarrow C$ and $N : B \rightarrow C$ side-by-side, yielding $[M|N] : A + B \rightarrow C$. 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

$$P \cdot [M|N] = [P \cdot M | P \cdot N] \quad (5)$$

and (for suitably typed matrices) the equality law,

$$[M|N] = [P|Q] \equiv M = P \wedge N = Q \quad (6)$$

both silently used in the derivation above.

Our matrix semantics for the for-loop combinator can still be simplified in two ways: first, the $\llbracket \cdot \rrbracket$ parentheses in (4) can be dropped, since we may assume they are implicitly surrounding functions everywhere:

$$(\text{for } b \ i) \cdot [0|\text{succ}] = [\dot{z}|(b \cdot (\text{for } b \ i))]$$

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

$$[M|N] \cdot (P \oplus Q) = [M \cdot P | N \cdot Q] \quad (7)$$

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 matrix compositions,

$$(\text{for } b \ i) \cdot [0|\text{succ}] = [\dot{z}|b] \cdot (id \oplus (\text{for } b \ i))$$

which corresponds to the typed matrix diagram which follows,

$$\begin{array}{ccc} & \xrightarrow{in^\circ = \left[\begin{array}{c} 0^\circ \\ \text{succ}^\circ \end{array} \right]} & \\ \mathbb{N}_0 & \xrightarrow{\cong} & 1 + \mathbb{N}_0 \\ \text{for } b \ i \downarrow & \xrightarrow{in = [0|\text{succ}]} & \downarrow id \oplus (\text{for } b \ i) \\ B & \xrightarrow{[\dot{z}|b]} & 1 + B \end{array}$$

where symbol \cong indicates that function $in = [0|\text{succ}]$ is a bijection, and therefore its converse in° is also a function. 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. The diagram also uses the *split* combinator $[\cdot]$ which is the converse dual of $[\cdot]$:

$$[M|N]^\circ = \left[\frac{M^\circ}{N^\circ} \right] \quad (8)$$

Why does this diagram 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 the 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 $\underline{i} : 1 \rightarrow \mathbb{N}_0$ — a column vector — corresponds to the Dirac distribution on i , which can be changed to any other distribution.

Moreover, because in is a bijection, not only the diagram tells that for b i is a solution to the equation

$$k \cdot \text{in} = [\underline{i}|b] \cdot (\text{id} \oplus k)$$

but it turns out that this is the unique solution:¹⁰

$$k = \text{for } b \ i \equiv k \cdot \text{in} = [\underline{i}|b] \cdot (\text{id} \oplus k) \quad (9)$$

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

$$k = \underline{i} \cdot \underline{0}^\circ + b \cdot k \cdot \text{succ}^\circ \quad (10)$$

which is obtained from (9) above by use of the so-called ‘divide-and-conquer’ law:

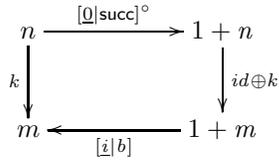
$$[M|N] \cdot \left[\frac{P}{Q} \right] = M \cdot P + N \cdot Q \quad (11)$$

Equation (10) tells how the matrix $k = \text{for } b \ i$ is recursively filled up: first the outer-product $\underline{i} \cdot \underline{0}^\circ$, that is, the everywhere-0 matrix apart from the 1 in cell $(i, 0)$, which is added to $b \cdot \underline{i} \cdot \underline{0}^\circ \cdot \text{succ}^\circ$, and so on. For sharp b , this is $(b \ i) \cdot 1^\circ$, the n -th entry being $(b^n \ i) \cdot n^\circ$. Note that each contribution of the fixpoint ascending chain is a matrix which “fills an empty column”, thus ensuring that no column ever adds up to more than 1.

Equation (10) also serves to emulate the construction of the fixpoint using matrix algebra packages such as, for instance, MATLAB. In this case we build finite approximations of the fixpoint helped by the corresponding diagram approximation, for inputs

¹⁰The argument is the same as in (Bird and de Moor, 1997) just by replacing the powerset monad by the distribution monad.

at most n and at most m possible outputs:



As MATLAB is not typed, tracing matrix dimensions without the help of diagrams of this kind would be a nightmare.

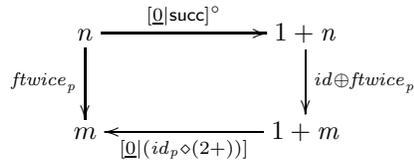
Let us see an example: suppose we want to emulate a fault in the *odd* function, $odd = (1+) \cdot (2*)$, in which $(2*) = \text{for } (2+) \text{ } 0$ is disturbed by the propagation of the same fault of addition we have seen before:

$$ftwice_p = m \text{ for } fadd_p \ 2 \ 0$$

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,



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 in the corresponding instance of (10),

$$k = \underline{0} \cdot \underline{0}^\circ + (id_p \diamond (2+)) \cdot k \cdot \text{succ}^\circ \quad (12)$$

is captured by MATLAB function

```
function C = faddk(p,k,n,m)
    M = eye(m,n);
    N = addk(k,n,m);
    C = choice(p,M,N);
end
```

(note the types, ie. dimensions n and m , passed as parameters) where

```
function C = choice(p,M,N)
    if size(M) ~= size(N)
```

```

        error('Dimensions must agree');
    else
        C = p*M+(1-p)*N
    end
end
end

```

(note the need for explicit type error checking). The right-hand side of the equation (12) is captured by

```

function Y = twiceF(p,n,m,X)
    if size(X) ~= [m n]
        error('Dimensions must agree');
    else
        Y = zero(m)*zero(n)' +
            faddk(p,2,m,m)*X*succ(n,n)'
    end
end

```

For $n, m = 5, 8$ and $p = 0.1$, the fixpoint of equation (12) 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 $n = 0 \dots 4$.

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 go back to analytical methods relying solely on universal property (9) 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

$$(f, g) x = (f x, g x)$$

where $(f, g): A \rightarrow B \times C$ 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 $M \triangle N$ of two arbitrary matrices M and N , defined index-wise by

$$(b, c) (M \triangle N) a = (b M a) \times (c N a) \quad (13)$$

Thus the Khatri-Rao product is a “column-wise” version of the well-known Kronecker product $\cdot \otimes \cdot$, defined by

$$(y, x) (M \otimes N) (b, a) = (y M b) \times (x N a) \quad (14)$$

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

$$u \triangle v = u \otimes v \quad (15)$$

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

$$[M|N] \triangle [P|Q] = [(M \triangle P)|(N \triangle Q)] \quad (16)$$

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,

$$k = f \triangle g \equiv fst \cdot k = f \wedge snd \cdot k = g \quad (17)$$

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

From (17) a number of useful corollaries arise, namely (keep in mind that f and g should be sharp functions for the time being) *fusion*,

$$(f \triangle g) \cdot h = (f \cdot h) \triangle (g \cdot h) \quad (18)$$

reconstruction,¹¹

$$k = (fst \cdot k) \triangle (snd \cdot k) \quad (19)$$

and pairwise *equality*:

$$k \triangle h = f \triangle g \equiv k = f \wedge h = g \quad (20)$$

This makes it easy to prove the mutual recursion law, below instantiated to for-loops, where $F f$ abbreviates $id \oplus f$:¹²

$$\begin{aligned} f \triangle g &= \text{for } (h \triangle k) (i, j) \\ &\equiv \{ \text{universal property (10)} \} \\ (f \triangle g) \cdot \text{in} &= [(\underline{i}, \underline{j})|(h \triangle k)] \cdot F (f \triangle g) \\ &\equiv \{ \text{fusion (18); constant functions} \} \\ (f \cdot \text{in}) \triangle (g \cdot \text{in}) &= [(\underline{i} \triangle \underline{j})|(h \triangle k)] \cdot F (f \triangle g) \end{aligned}$$

¹¹Cf. *loss-less decomposition* (Oliveira, 2011).

¹²As is well-known, for sharp functions this law extends to other inductive types, eg. lists, trees etc (Bird and de Moor, 1997; Hu et al., 1997).

$$\begin{aligned}
&\equiv \{ \text{exchange law (16)} \} \\
&(f \cdot \text{in}) \triangle (g \cdot \text{in}) = ([\underline{i}|h] \triangle [\underline{j}|k]) \cdot F(f \triangle g) \\
&\equiv \{ \text{fusion (18) again} \} \\
&(f \cdot \text{in}) \triangle (g \cdot \text{in}) = ([\underline{i}|h] \cdot F(f \triangle g)) \triangle ([\underline{j}|k] \cdot F(f \triangle g)) \\
&\equiv \{ \text{equality (20)} \} \\
&\left\{ \begin{array}{l} f \cdot \text{in} = [\underline{i}|h] \cdot F(f \triangle g) \\ g \cdot \text{in} = [\underline{j}|k] \cdot F(f \triangle g) \end{array} \right.
\end{aligned}$$

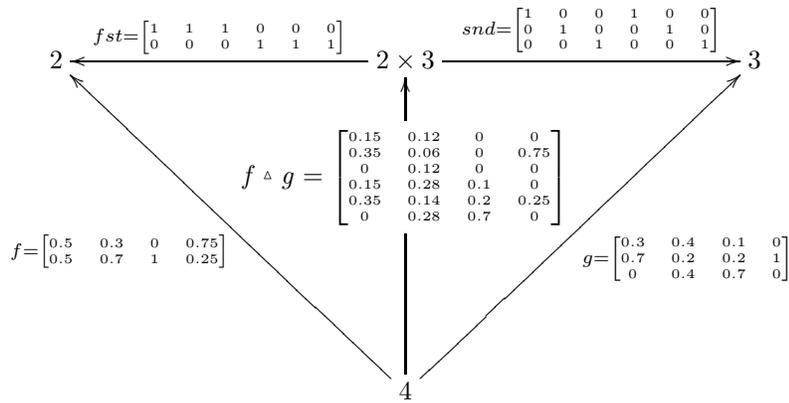
Read in reverse direction, this reasoning explains how two recursive, mutually dependent functions f and g (regarded as matrices) combine with each other into one single function $f \triangle g$, from which one can extract both f and g by projecting according to the *cancellation rule*,

$$fst \cdot (f \triangle g) = f \wedge snd \cdot (f \triangle g) = g \quad (21)$$

yet another corollary of (17).

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 eg. references (Bird and de Moor, 1997; Hu et al., 1997) among several others.

We are now interested in checking the probabilistic extension of (17). Let two probabilistic functions f and g and their product $f \triangle g$ be depicted as the CS-matrices of the following diagram:



We can handle this in Haskell by running the following monadic functions

$$(f \triangle g) a = \mathbf{do} \{ b \leftarrow f a; c \leftarrow g a; \text{return } (b, c) \}$$

```

mfst d = do { (b, c) ← d; return b }
msnd d = do { (b, c) ← d; return c }

```

inside the distribution monad $Dist$, thereby implementing the Khatri-Rao product and its projections. For instance, $(f \triangle g) \cdot 2$ 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 $fst \cdot (f \triangle g) = f$ and $snd \cdot (f \triangle g) = g$.

However, *reconstruction* (19) does not extend probabilistically. 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

$$k : 3 \rightarrow 2 \times 3$$

$$k = \begin{bmatrix} 0 & 0.4 & 0.2 \\ 0.2 & 0 & 0.17 \\ 0.2 & 0.1 & 0.13 \\ 0.6 & 0.4 & 0.2 \\ 0 & 0 & 0.17 \\ 0 & 0.1 & 0.13 \end{bmatrix}$$

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

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

This happens because probabilistic Khatri-Rao is a *weak* product — the expected equivalence (17) is only an implication,

$$k = f \triangle g \Rightarrow fst \cdot k = f \wedge snd \cdot k = g \quad (22)$$

ensuring existence but not uniqueness. The proof of (22), which is equivalent to cancellation (21) — substitute k and simplify — can be found in appendix Appendix A. This proof relies on properties (15) and (16) of the Khatri-Rao product.

Weak product (22) also grants pairwise equality (20) — substitute k by $k \triangle h$ and simplify — but the converse substitution of f and g , in the \Leftarrow direction, leading to *reconstruction* (19) is of course invalid. In turn, this invalidates fusion (18) for arbitrary

probabilistic functions f , g and h , although the property will still hold in case h is sharp¹³, as the straightforward proof in appendix Appendix A shows.

Altogether, the mutual recursion law will not hold in general for probabilistic functions, as its calculation (above) relies on fusion (18). 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 $mfiobl$ 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).

7. Asymmetric Khatri-Rao product

To re-establish the equivalence in (17) given (22) we just have to find conditions for the converse implication

$$k = f \triangle g \iff fst \cdot k = f \wedge snd \cdot k = g$$

to hold, which is equivalent to (19) 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 $(b, c) R a \equiv b (fst \cdot R) a \wedge c (snd \cdot R) a$ holds. That is, by forking $fst \cdot R$ and $snd \cdot R$ one rebuilds R .

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

If either $fst \cdot k$ or $snd \cdot k$ are sharp functions then (19) holds.

Some intuitions first, before checking this. Let $k : A \rightarrow B \times C$ be a CS-matrix. The fact that $f = fst \cdot k : A \rightarrow B$ is sharp 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 $snd \cdot k : A \rightarrow C$ yields such a block; $\langle fst \cdot k, snd \cdot k \rangle$ puts it back in place.

The proof of this conjecture, whereby (19) grants for free the *reflection law*

$$\langle fst, snd \rangle = id \tag{23}$$

(take $f, g, k := fst, snd, id$ and note that all functions involved are sharp), will resort to the definition of (typed) matrix composition, for $M : B \rightarrow C$ and $N : A \rightarrow B$,

$$c(M \cdot N)a = \langle \sum b :: (c M b) \times (b N a) \rangle \tag{24}$$

¹³The same happens with *forks* in relation algebra (Bird and de Moor, 1997).

and to two rules which interface index-free and index-wise matrix notation, where N is an arbitrary matrix and f, g are functional (ie. sharp) matrices: ¹⁴

$$y(f \cdot N)x = \langle \sum z : y = f z : zNx \rangle \quad (25)$$

$$y(g^\circ \cdot N \cdot f)x = (g y) N (f x) \quad (26)$$

Let us suppose $fst \cdot k$ in (19) is sharp. We denote by $f : A \rightarrow B$ the proper function which $fst \cdot k$ is, by hypothesis. Thus $f = fst \cdot k$. 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

$$\langle \sum c :: (f a, c) k a \rangle = 1 \quad (27)$$

$$\langle \sum (b, c) : (b \neq f a) : ((b, c) k a) \rangle = 0 \quad (28)$$

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

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

the same as

$$(b, c) m a = (b f a) \times \langle \sum b' :: (b', c) k a \rangle \quad (29)$$

since $f = fst \cdot k$ an snd is sharp (25). Our aim is to prove that $m = k$.

Case $b \neq f a$. In this case $b f a = 0$ and (29) yields $(b, c) m a = 0$. From (28) we also get $(b, c) k a = 0$ and so $m = k$ for this case.

Case $b = f a$. we have

$$\begin{aligned} & (f a, c) m a \\ = & \{ (29) ; (b f a) = 1 \text{ for } b = f a \} \\ & \langle \sum b' :: (b', c) k a \rangle \\ = & \{ b' = f a \vee b' \neq f a \} \\ & \langle \sum b' : b' = f a \vee b' \neq f a : (b', c) k a \rangle \\ = & \{ \text{split summation ; one-point over } b' = f a \} \\ & ((f a, c) k a) + \langle \sum b' : b' \neq f a : (b', c) k a \rangle \\ = & \{ (28) \} \\ & (f a, c) k a \end{aligned}$$

¹⁴These rules are derived by Oliveira (2012b) adopting the Eindhoven notation (Backhouse and Michaelis, 2006; Morgan, 2012) for summations, eg. $\langle \sum x : R : S \rangle$ 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 omit R in this case.

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

□

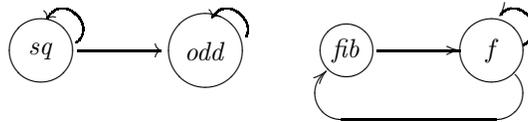
The proof assuming $snd \cdot k$ sharp instead of $fst \cdot k$ being so will be essentially the same. The remaining assumptions (27) and (28) are easily proved in the appendix.

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 (17) holds and all its corollaries.¹⁵ This means that, under the same assumption, the mutual recursion law will hold too.

Put in other words, the probabilistic behaviour of a pair-valued recursive function, for instance a for-loop $k = \text{for } b \ i$, will be the same as the product $f \ \Delta \ g$ 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 $mfiobl$ 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 ,

```
msqlo n =
  do { (s, o) ← mfor loop (0, 1) n; return o }
  where loop (s, o) =
    do { z ← fadd0,1 s o; return (z, o + 2) }
```

and run eg. `msqlo 5`, for instance

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

¹⁵This includes, of course, the standard case in which both f and g are sharp functions.

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

```
Main> msq1 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,

```
odd' 0 = return 1
odd' (n + 1) = do { x ← odd' n; fadd0,1 2 x }
```

and check that suitably adapted *msq*, mutually dependent on *odd'*,

```
msq' 0 = return 0
msq' (n + 1) = do { m ← msq' n; x ← odd' n; fadd0,1 m x }
```

and its linear version,

```
msql' n =
  do { (s, o) ← mfor loop (0, 1) n; return s } where
  loop (s, o) = do {
    z ← fadd0,1 s o; x ← fadd0,1 2 o;
    return (z, x) }
```

now exhibit different probabilistic behaviours, for instance,

<i>n</i>	<i>msq' n</i>	<i>msql' n</i>
3	9 59.0%	9 65.6%
	7 19.7%	5 15.4%
	5 10.3%	7 7.3%
	8 6.6%	8 7.3%
	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%

where linear scores better than mutually recursive, still.

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. 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*:

$$\text{for } f (a \text{ } _p \diamond b) = (\text{for } f a) \text{ } _p \diamond (\text{for } f b) \quad (30)$$

The need for a generalization can be seen already in writing “ $a \text{ } _p \diamond b$ ”, an abuse of notation since the choice operator chooses between functions, not arbitrary values. Thus construct $\text{for } f i$ 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 (9) becomes, for $F f = id \oplus f$,

$$k = ([h|f]) \equiv k \cdot \text{in} = [h|f] \cdot (F k) \quad (31)$$

Clearly,

$$\text{for } f a = ([\underline{a}|f]) \quad (32)$$

holds. In (30), abbreviation for $\text{for } f (a \text{ } _p \diamond b)$ replacing $([[\underline{a} \text{ } _p \diamond \underline{b}]|f])$ is welcome as it enhances readability.

The proof of (30) is given in appendix Appendix A. It relies on properties of probabilistic choice already given by Oliveira (2012a), namely *choice-fusion*

$$(f \text{ } _p \diamond g) \cdot h = (f \cdot h) \text{ } _p \diamond (g \cdot h) \quad (33)$$

$$h \cdot (f \text{ } _p \diamond g) = (h \cdot f) \text{ } _p \diamond (h \cdot g) \quad (34)$$

and the *exchange law*:

$$[f|g] \text{ } _p \diamond [h|k] = [(f \text{ } _p \diamond h)|(g \text{ } _p \diamond k)] \quad (35)$$

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 level¹⁶,

$$k = \text{fold } f d \equiv k \cdot \text{in} = [d|f] \cdot (F k) \quad (36)$$

where $F k = id \oplus (id \otimes k)$ and $\text{in} = [\text{nil}|\text{cons}]$ is the initial algebra of sequences, for (in Haskell notation) $\text{nil } _ = []$ and $\text{cons } (h, t) = h : t$. Besides the direct sum $(id \oplus \cdot)$ splitting base from recursive case, as with for, recursive pattern $F k$ involves the Kronecker product $id \otimes k$ which delivers to f the head of the current sequence and the outcome of the recursive call k . The base case is captured by vector d , a

¹⁶Both are instances of the generic *catamorphism* construct, as already mentioned.

distribution. For sharp functions, $\text{fold } f \ \underline{u}$ means the same as $\text{foldr } (\text{curry } f) \ u$ in standard Haskell. (This difference is not a very significant one, as we shall see in the examples below.) Substitution of k will yield a closed formula for probabilistic fold (cancellation property):

$$\begin{aligned} \text{fold } f \ d &= [d]f \cdot (id \otimes (\text{fold } f \ d)) \cdot \left[\frac{\text{nil}^\circ}{\text{cons}^\circ} \right] \\ &\equiv \{ \text{divide-and-conquer (11)} \} \\ &= d \cdot \text{nil}^\circ + f \cdot (id \otimes (\text{fold } f \ d)) \cdot \text{cons}^\circ \end{aligned} \quad (37)$$

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

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

where $\text{lose} = \text{snd}$ and $\text{send} = \text{cons}$. 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 :

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

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, eg. $(fcount_q \cdot fcat_p)$ "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 (36) in the standard way (Bird and de Moor, 1997) and also in the probabilistic setting:

$$k \cdot (\text{fold } g \ e) = \text{fold } f \ d \iff k \cdot [e|g] = [d|f] \cdot (\mathbf{F} \ k) \quad (38)$$

In our case, this enables us to solve the equation $\text{fcount}_q \cdot \text{fcats}_p = \text{fold } x \ y$ for unknowns x and y :

$$\begin{aligned} & \text{fcount}_q \cdot \text{fcats}_p = \text{fold } x \ y \\ \iff & \quad \{ \text{fold fusion (38)} ; \text{definition of } \text{fcats}_p \} \\ & \text{fcount}_q \cdot [\text{nil} | (\text{lose}_p \diamond \text{send})] = [x | y] \cdot (\mathbf{F} \ \text{fcount}_q) \\ \equiv & \quad \{ (5) ; \text{definition of } \mathbf{F} ; (7) ; (6) \} \\ & \begin{cases} \text{fcount}_q \cdot \text{nil} = x \\ \text{fcount}_q \cdot (\text{lose}_p \diamond \text{send}) = y \cdot (\text{id} \otimes \text{fcount}_q) \end{cases} \\ \equiv & \quad \{ \text{fcount}_q \cdot \text{nil} = \underline{0} \} \\ & \begin{cases} x = \underline{0} \\ \text{fcount}_q \cdot (\text{snd}_p \diamond \text{cons}) = y \cdot (\text{id} \otimes \text{fcount}_q) \end{cases} \end{aligned}$$

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

$$\begin{aligned} & \text{fcount}_q \cdot (\text{snd}_p \diamond \text{cons}) = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \equiv & \quad \{ \text{choice fusion (34)} \} \\ & (\text{fcount}_q \cdot \text{snd})_p \diamond (\text{fcount}_q \cdot \text{cons}) = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \equiv & \quad \{ \text{unfolding } \text{fcount}_q \cdot \text{cons} \} \\ & (\text{fcount}_q \cdot \text{snd})_p \diamond ((\text{id}_q \diamond \text{succ}) \cdot \text{snd} \cdot (\text{id} \otimes \text{fcount}_q)) \\ & = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \equiv & \quad \{ \text{free theorem of } \text{snd} \} \\ & (\text{fcount}_q \cdot \text{snd})_p \diamond ((\text{id}_q \diamond \text{succ}) \cdot \text{fcount}_q \cdot \text{snd}) \\ & = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \equiv & \quad \{ \text{choice fusion (33)} \} \\ & (\text{id}_p \diamond (\text{id}_q \diamond \text{succ})) \cdot \text{fcount}_q \cdot \text{snd} = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \equiv & \quad \{ \text{free theorem of } \text{snd} \text{ again} \} \\ & (\text{id}_p \diamond (\text{id}_q \diamond \text{succ})) \cdot \text{snd} \cdot (\text{id} \otimes \text{fcount}_q) = y \cdot (\text{id} \otimes \text{fcount}_q) \\ \Leftarrow & \quad \{ \text{Leibniz } (\text{id} \otimes \text{fcount}_q \text{ cancelled from both sides}) \} \\ & y = (\text{id}_p \diamond (\text{id}_q \diamond \text{succ})) \cdot \text{snd} \end{aligned}$$

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

$$\begin{aligned}
 fcount_q \cdot fcat_p = \\
 \text{fold } y \ \underline{0} \ \text{where} \\
 y = ((p + q - pq) \text{ id} + (1 - p) (1 - q) \text{ succ}) \cdot \text{snd}
 \end{aligned}$$

in which the probabilistic definition of y combines the choices according to (3). It can be checked that this behaviour (which corresponds to that of a even more risky $fcount$, reading from a perfect cat) matches up with the distributions obtained for the specific runs given earlier.

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:

$$\text{avg } l = \frac{\text{sum } l}{\text{count } l} \tag{39}$$

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 (39) by defining

$$\text{favg}_{p,q} = \text{fsum}_p \ \Delta \ \text{fcount}_q$$

for $fcount_q$ as before and

$$\text{fsum}_p = \text{fold } (\text{uncurry } \text{fadd}_p) \ \underline{0}$$

a (faulty) list sum function.¹⁷ For instance, we have 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%

```

¹⁷We focus on computing the pair of values of (39), leaving aside the final division and the problem of the divisions by zero which arise from faulty counting (to be handled by raising exceptions).

(5, 0)	0.7%
(0, 1)	0.4%
(2, 0)	0.1%
(3, 0)	0.1%
(0, 0)	0.0%

which 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 $favgs_{p,q}$ into a single fold on total/count pairs (t, c) ,

```

favgsp,q = fold body (0, 0) where
  body (a, (t, c)) = do {
    t' ← faddp a t;
    c' ← (idq ◊ succ) c;
    return (t', c')}

```

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 both for-loops and folds on lists, we generalize both (9) and (36) to

$$k = \langle f \rangle \equiv k \cdot \text{in} = f \cdot (\mathbf{F} k) \quad (40)$$

where f is a suitably typed probabilistic function covering both the inductive and the base cases of (9,36), and the customary *banana* brackets $\langle _ \rangle$ are used to denote such a generic fold, or *catamorphism*.¹⁸ Cancellation

$$\langle f \rangle \cdot \text{in} = f \cdot \mathbf{F} \langle f \rangle \quad (41)$$

follows trivially from (40).

Theorem 1 (Probabilistic ‘banana-split’). *Transformation*

$$\langle f \rangle \triangle \langle g \rangle = \langle (f \otimes g) \cdot \text{unzip}_{\mathbf{F}} \rangle \quad (42)$$

where

$$\text{unzip}_{\mathbf{F}} = (\mathbf{F} \text{fst}) \triangle (\mathbf{F} \text{snd}) \quad (43)$$

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

$$(\mathbf{F} f \otimes \mathbf{F} g) \cdot \text{unzip}_{\mathbf{F}} = \text{unzip}_{\mathbf{F}} \cdot \mathbf{F} (f \otimes g) \quad (44)$$

¹⁸Functors $\mathbf{F} X = id \oplus X$ and $\mathbf{F} X = id \oplus (id \otimes X)$ give us back for-loops and list folds, respectively.

Proof: Relying on absorption law

$$(M \cdot N) \triangle (P \cdot Q) = (M \otimes P) \cdot (N \triangle Q) \quad (45)$$

valid for any (suitably typed) matrices M, N, P, Q (Macedo, 2012), we proceed by cata-universality, by solving for f the right hand side equation of (40), once k is instantiated to $k = (\!|f\!) \triangle (\!|g\!)$:

$$\begin{aligned} & ((\!|f\!) \triangle (\!|g\!)) \cdot \text{in} \\ = & \quad \{ \text{as in is a proper function, pair-fusion holds (A.1)} \} \\ & ((\!|f\!) \cdot \text{in}) \triangle ((\!|g\!) \cdot \text{in}) \\ = & \quad \{ \text{two cancellations (41)} \} \\ & (f \cdot F (\!|f\!)) \triangle (g \cdot F (\!|g\!)) \\ = & \quad \{ \text{pairing-absorption (45)} \} \\ & (f \otimes g) \cdot ((F (\!|f\!)) \triangle (F (\!|g\!))) \\ = & \quad \{ (46) \text{ below} \} \\ & (f \otimes g) \cdot \text{unzip}_F \cdot F ((\!|f\!) \triangle (\!|g\!)) \end{aligned}$$

□

Thus (42) holds, by (40). As shown in the appendix, fact

$$\text{unzip}_F \cdot F (f \triangle g) = (F f) \triangle (F g) \quad (46)$$

used in the proof is an immediate corollary of the naturality (44) of unzip_F .

□

In the appendix we show that functors which support folds and for-loops are such that (44) holds, thus granting “banana-split” (42) for such programming schemes. Moreover, this property is structurally preserved by functor composition, sum etc.

In retrospect, note how law (42) was proved not as a corollary of mutual recursion but as an *independent* result. Also note the major role of function unzip_F (43) 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*

$$\left\{ \begin{array}{l} f \cdot \text{in} = h \cdot F (f \triangle g) \\ g \cdot \text{in} = k \cdot F (f \triangle g) \end{array} \right. \equiv f \triangle g = (\!|h \triangle k\!) \quad (47)$$

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 $h \cdot F(f \triangle g) = h' \cdot (G f)$, 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 worse 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, 2012a), 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 eg. 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.

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

Program analysis techniques based on languages such as eg. Rely (Carbin et al., 2013) evaluate quantitative reliability of computations running on unreliable hardware, eg. 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.

On the foundations side, probabilistic *weak* tupling has been addressed in the more wide setting of *monoidal* categories adopted in eg. categorical quantum physics (Coecke, 2011). These include not only *FdHilb*, the category of finite dimensional Hilbert spaces, but also *Rel*, the category of binary relations. Thus the remarks by Coecke and Paquette, in their *Categories for the Practising Physicist* (Coecke, 2011):

Rel [the category of relations] possesses more ‘quantum features’ than the category *Set* of sets and functions [...] The categories *FdHilb* and *Rel* moreover admit a categorical matrix calculus.

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.

On a more practical register, we would like to find side-conditions for probabilistic mutual-recursion (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 hope to be able to generalize previous work in this field (Oliveira, 2011).

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 $f : A \rightarrow B$ be such a function and $g, h : A \rightarrow B$ be probabilistic approximations to it, all represented as CS-matrices. Then g and h can be compared against f as follows,

$$g \leq_f h \quad \text{iff} \quad g \times f \leq h \times f$$

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, $g \leq_f f$ 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?

Scaling-up, another follow-up of the strategy put forward in this paper is 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 (Barbosa et al., 2013) in which components represented by *coalgebras* (Barbosa, 2003) extended probabilistically, by adding to the coalgebraic matrices of (Oliveira, 2012b) a *behaviour* monad inside the *distribution* one.

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 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.

Daniel Murta holds grant BI1-2012_PTDC/EIA-CCO/122240/2010_UMINHO awarded by FCT (Portuguese Foundation for Science and Technology).

References

- R. 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.
- L.S. Barbosa, D.R. Murta, and J.N. Oliveira. Introducing fault propagation in a software component calculus, 2013. In preparation.
- G. Barthe, B. Grégoire, and S.Z. Béguélin. 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.
- 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. doi: 10.1145/2034773.2034777.
- T. Harper. A library writer’s guide to shortcut fusion. In *Haskell Symposium 2011*, September 2011.
- 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.
- A. Lingamneni, C.ENZ, K. Palem, and C. Piguët. 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. URL <http://doi.acm.org/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>.
- 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, University of Minho and INESC TEC, 2011. available from <http://wiki.di.uminho.pt/twiki/bin/view/DI/FMHAS/TechnicalReports>.
- J.N. Oliveira. Towards a linear algebra of programming. *Formal Asp. Comput.*, 24 (4-6):433–458, 2012a.
- J.N. Oliveira. Typed linear algebra for weighted (probabilistic) automata. In *CIAA*, volume 7381 of *LNCS*, pages 52–65, 2012b.
- 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, Dec.
- J. Voas and G. McGraw. *Software Fault Injection: Inoculating 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 in appendix

Proof of cancellation (21). Base case (f and g are column vectors)¹⁹:

$$\begin{aligned}
&fst \cdot (f \triangle g) = f \wedge snd \cdot (f \triangle g) = g \\
\equiv &\quad \{ fst = id \otimes ! \text{ and } snd = ! \otimes id \} \\
&(id \otimes !) \cdot (f \triangle g) = f \wedge (! \otimes id) \cdot (f \triangle g) = g \\
\equiv &\quad \{ \text{for vectors, } f \triangle g = f \otimes g \text{ (15)} \} \\
&(id \otimes !) \cdot (f \otimes g) = f \wedge (! \otimes id) \cdot (f \otimes g) = g \\
\equiv &\quad \{ \text{functor-} \cdot \otimes \cdot ; \text{natural-id} \} \\
&f \otimes (! \cdot g) = f \wedge (! \cdot f) \otimes g = g \\
\equiv &\quad \{ g \text{ is probabilistic, therefore } ! \cdot f = ! \cdot g = ! \text{ (Oliveira, 2012a)} \} \\
&f \otimes ! = f \wedge ! \otimes g = g \\
\equiv &\quad \{ 1 \xleftarrow{!} 1 = 1 \text{ and } M \otimes 1 = M \} \\
&f = f \wedge g = g
\end{aligned}$$

□

Inductive step: $f = [f_1|f_2]$ and $g = [g_1|g_2]$. Calculating $fst \cdot (f \triangle g) = f$ first:

$$\begin{aligned}
&fst \cdot (f \triangle g) = f \\
\equiv &\quad \{ f = [f_1|f_2] \text{ and } g = [g_1|g_2] \} \\
&fst \cdot ([f_1|f_2] \triangle [g_1|g_2]) = [f_1|f_2] \\
\equiv &\quad \{ \text{exchange law (16)} \} \\
&fst \cdot [(f_1 \triangle g_1)|(f_2 \triangle g_2)] = [f_1|f_2] \\
\equiv &\quad \{ \text{fusion (5)} \} \\
&[(fst \cdot (f_1 \triangle g_1))|(fst \cdot (f_2 \triangle g_2))] = [f_1|f_2] \\
\equiv &\quad \left\{ \begin{array}{l} \text{induction hypothesis: } fst \cdot (f \triangle g) = f \\ \text{holds for } f, g := f_i, g_i \text{ (} i = 1, 2 \text{)} \end{array} \right\} \\
&[f_1|f_2] = [f_1|f_2]
\end{aligned}$$

□

Branch $snd \cdot (f \triangle g) = g$ is calculated in a similar way.

¹⁹Row vector $!:A \rightarrow 1$ corresponds to the sharp, constant function which maps every input to the singleton datatype.

Proof of (27). This equality arises from rule (25):

$$\begin{aligned}
& \langle \sum c :: (f a, c) k a \rangle = 1 \\
\equiv & \quad \{ \text{one-point rule} \} \\
& \langle \sum b, c : f a = b : (b, c) k a \rangle = 1 \\
\equiv & \quad \{ b = fst (b, c) ; (25) \} \\
& (f a) (fst \cdot k) a = 1 \\
\equiv & \quad \{ f = fst \cdot k \} \\
& (f a) f a = 1 \\
\equiv & \quad \{ f \text{ is sharp} \} \\
& \text{true}
\end{aligned}$$

□

Proof of (28). This equality arises from k being probabilistic:

$$\begin{aligned}
& \langle \sum b, c : b \neq f a : (b, c) k a \rangle = 0 \\
\equiv & \quad \{ 1 + 0 = 1 \} \\
& 1 + \langle \sum b, c : b \neq f a : (b, c) k a \rangle = 1 \\
\equiv & \quad \{ (27) \} \\
& \langle \sum c :: (f a, c) k a \rangle + \\
& \quad \langle \sum b, c : b \neq f a : (b, c) k a \rangle = 1 \\
\equiv & \quad \{ \text{merge quantifiers} \} \\
& \langle \sum (b, c) :: (b, c) k a \rangle = 1 \\
\equiv & \quad \{ k \text{ is probabilistic} \} \\
& \text{true}
\end{aligned}$$

□

Proof of base-case fault propagation (30). Clearly, by (32) and universal property (31), our target (30) re-writes to the equality

$$\begin{aligned}
& ((\text{for } f a)_{p \diamond} (\text{for } f b)) \cdot \text{in} = \\
& \quad [(\underline{a}_{p \diamond} \underline{b})|f] \cdot (\text{F } ((\text{for } f a)_{p \diamond} (\text{for } f b)))
\end{aligned}$$

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

$$\begin{aligned}
& ((\text{for } f \ a) \text{ }_p \diamond (\text{for } f \ b)) \cdot \text{in} \\
= & \quad \{ \text{choice-fusion (33)} \} \\
& (\text{for } f \ a \cdot \text{in}) \text{ }_p \diamond (\text{for } f \ b \cdot \text{in}) \\
= & \quad \{ (32) \text{ and } (31), \text{ twice} \} \\
& ([\underline{a}|f] \cdot \mathbf{F} (\text{for } f \ a)) \text{ }_p \diamond ([\underline{b}|f] \cdot \mathbf{F} (\text{for } f \ b)) \\
= & \quad \{ \mathbf{F} f = id \oplus f ; [M|N] \cdot (P \oplus Q) = [M \cdot P|N \cdot Q] \} \\
& [\underline{a}|(f \cdot (\text{for } f \ a))] \text{ }_p \diamond [\underline{b}|(f \cdot (\text{for } f \ b))] \\
= & \quad \{ \text{exchange law (35)} \} \\
& [(\underline{a} \text{ }_p \diamond \underline{b})|(f \cdot (\text{for } f \ a) \text{ }_p \diamond (f \cdot \text{for } f \ b))] \\
= & \quad \{ \text{choice-fusion (34)} \} \\
& [(\underline{a} \text{ }_p \diamond \underline{b})|(f \cdot ((\text{for } f \ a) \text{ }_p \diamond (\text{for } f \ b)))] \\
= & \quad \{ [M|N] \cdot (P \oplus Q) = [M \cdot P|N \cdot Q] \} \\
& [(\underline{a} \text{ }_p \diamond \underline{b})|f] \cdot (id \oplus ((\text{for } f \ a) \text{ }_p \diamond (\text{for } f \ b))) \\
= & \quad \{ \mathbf{F} f = id \oplus f \} \\
& [(\underline{a} \text{ }_p \diamond \underline{b})|f] \cdot (\mathbf{F} ((\text{for } f \ a) \text{ }_p \diamond (\text{for } f \ b)))
\end{aligned}$$

□

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

$$(M \triangle N) \cdot h = (M \cdot h) \triangle (N \cdot h) \iff h \text{ is sharp} \quad (\text{A.1})$$

where probabilistic functions f and g are generalized to arbitrary matrices M and N :

$$\begin{aligned}
& (b, c) ((M \triangle N) \cdot h) \ a \\
= & \quad \{ (26) \text{ for } h \text{ a standard function} \} \\
& (b, c) (M \triangle N) (h \ a) \\
= & \quad \{ \text{pointwise Khatri-Rao (13)} \} \\
& (b \ M \ (h \ a)) \times (c \ N \ (h \ a)) \\
= & \quad \{ (26) \text{ for } h \text{ a standard function} \} \\
& b \ (M \cdot h) \ a \times c \ (N \cdot h) \ a \\
= & \quad \{ \text{pointwise Khatri Rao (13) — twice} \} \\
& (b, c) ((M \cdot h) \triangle (N \cdot h)) \ a
\end{aligned}$$

□

Proofs concerning naturality of unzip_F (44). This property holds trivially for the identity functor $F X = X$, where $\text{unzip}_F = id$, and for any constant functor $F X = K$, in which case $\text{unzip}_F = id \triangle id$.

We show next that the property is structurally preserved by functor composition, say $F = G H$, whereby

$$\text{unzip}_{GH} = \text{unzip}_G \cdot (G \text{unzip}_H) \quad (\text{A.2})$$

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 (44) to arbitrary matrices M, N over a semiring. We have:

$$\begin{aligned} & \text{unzip}_{GH} \cdot G H (M \otimes N) \\ = & \quad \{ (\text{A.2}) \} \\ & \text{unzip}_G \cdot (G \text{unzip}_H) \cdot G (H (M \otimes N)) \\ = & \quad \{ \text{functor } G \text{ (composition)} \} \\ & \text{unzip}_G \cdot G (\text{unzip}_H \cdot H (M \otimes N)) \\ = & \quad \{ \text{induction hypothesis: assume (44) for } F = H; G \text{ again} \} \\ & \text{unzip}_G \cdot G (((H M) \otimes (H N))) \cdot (G \text{unzip}_H) \\ = & \quad \{ \text{induction hypothesis: assume (44) for } F = G \} \\ & ((G (H M)) \otimes (G (H N))) \cdot \text{unzip}_G \cdot G (\text{unzip}_H) \\ = & \quad \{ (\text{A.2}) \} \\ & ((G H M) \otimes (G H N)) \cdot \text{unzip}_{GH} \end{aligned}$$

□

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

$$\text{unzip}_F = ((G \text{fst}) \oplus (H \text{fst})) \triangle ((G \text{snd}) \oplus (H \text{snd})) \quad (\text{A.3})$$

Facts

$$\text{unzip}_F \cdot i_1 = (i_1 \otimes i_1) \cdot \text{unzip}_G \quad (\text{A.4})$$

$$\text{unzip}_F \cdot i_2 = (i_2 \otimes i_2) \cdot \text{unzip}_H \quad (\text{A.5})$$

are easy to prove via exchange law (16), 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

$$\begin{aligned} & [((i_1 \otimes i_1) \cdot (M \triangle N)) | ((i_2 \otimes i_2) \cdot (P \triangle Q))] \\ = & (M \oplus P) \triangle (N \oplus Q) \end{aligned} \quad (\text{A.6})$$

which is valid for all suitably typed matrices M, N, P and Q , and will help in the proof

that (44) holds for sums of functors which (inductively) satisfy the same property:

$$\begin{aligned}
& \text{unzip}_F \cdot F (M \otimes N) \\
\equiv & \quad \{ F = G \oplus H \} \\
& \text{unzip}_F \cdot ((G (M \otimes N)) \oplus (H (M \otimes N))) \\
= & \quad \{ M \otimes N = [(i_1 \cdot M)|(i_2 \cdot N)] \} \\
& [(\text{unzip}_F \cdot i_1 \cdot (G (M \otimes N)))(\text{unzip}_F \cdot i_2 \cdot (H (M \otimes N)))] \\
= & \quad \{ \text{(A.4,A.5)} \} \\
& [((i_1 \otimes i_1) \cdot \text{unzip}_G \cdot (G (M \otimes N)))(i_2 \otimes i_2 \cdot \text{unzip}_H \cdot (H (M \otimes N)))] \\
= & \quad \{ \text{induction hypothesis: assume (44) for } F = G \text{ and } F = H \} \\
& [((i_1 \otimes i_1) \cdot ((G M) \otimes (G N)) \cdot \text{unzip}_G)(i_2 \otimes i_2 \cdot ((H M) \otimes (H N)) \cdot \text{unzip}_H)] \\
\equiv & \quad \{ \text{definitions of } \text{unzip}_G \text{ and } \text{unzip}_H ; \text{ absorptions} \} \\
& [((i_1 \otimes i_1) \cdot (G (M \cdot fst)) \triangle (G (N \cdot snd)))(i_2 \otimes i_2 \cdot ((H (M \cdot fst)) \otimes (H (N \cdot snd)))))] \\
= & \quad \{ \text{(A.6)} \} \\
& ((G (M \cdot fst)) \oplus (H (M \cdot fst))) \triangle ((G (N \cdot snd)) \oplus (H (N \cdot snd))) \\
\equiv & \quad \{ F = G \oplus H \} \\
& (F (M \cdot fst)) \triangle (F (N \cdot snd)) \\
\equiv & \quad \{ \text{functor } F ; \text{ reverse absorption} \} \\
& ((F M) \otimes (F N)) \cdot ((F \cdot fst) \triangle (F \cdot snd)) \\
\equiv & \quad \{ \text{definition of } \text{unzip}_F \} \\
& ((F M) \otimes (F N)) \cdot \text{unzip}_F
\end{aligned}$$

□

Finally, we address functor $F X = id \otimes X$ which participates in the recursion schema of folds. Note that we can replace id by $id \oplus id$ since $\cdot \oplus \cdot$ is a bifunctor in any category of matrices (Macedo, 2012). Then

$$F X = (id \oplus id) \otimes X = (id \otimes X) \oplus (id \otimes X) = (F X) \oplus (F X)$$

This reduces this case to the previous one, for $G X = F X$ and $H X = F X$, where the identities in these functors are of smaller size. Thus, in a sense, induction proceeds on the size of the identity matrix which participates in functor definition $F X = id \otimes X$.

Proof of fact (46).

$$\begin{aligned}
& \text{unzip}_F \cdot F (f \triangle g) \\
= & \quad \{ \text{reverse pairing-absorption (45)} \} \\
& \text{unzip}_F \cdot F (f \otimes g) \cdot F (id \triangle id) \\
= & \quad \{ \text{naturality (44)} \} \\
& (F f \otimes F g) \cdot \text{unzip}_F \cdot F (id \triangle id) \\
= & \quad \{ \text{functor } F; \text{unzip}_F \text{ (43)}; \text{pairing-fusion (A.1), as } id \triangle id \text{ is sharp} \} \\
& (F f \otimes F g) \cdot (F (fst \cdot (id \triangle id)) \triangle F (fst \cdot (id \triangle id))) \\
= & \quad \{ \text{standard pairing-cancellation (21)} \} \\
& (F f \otimes F g) \cdot (F id \triangle F id) \\
= & \quad \{ \text{functor } F; \text{pairing-absorption (45)} \} \\
& (F f) \triangle (F g)
\end{aligned}$$

□