Calculating fault propagation in functional programs

Daniel R. Murta , José N. Oliveira
pg23205@alunos.uminho.pt, jno@di.uminho.pt

Techn. Report TR-HASLab:01:2013
May. 2013

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:01:2013
Calculating fault propagation in functional programs
by Daniel R. Murta , José N. Oliveira

Abstract

The production of safety critical software is bound to a number of safety and certification standards in which estimating the *risk of failure* plays a central role. Yet risk estimation seems to live outside most programmers' core practice, involving simulation techniques and worst case analysis performed *a posteriori*.

In this paper we propose that risk be constructively handled in functional programming by writing programs which choose between *expected* and *faulty* behaviour and by reasoning about them in a linear algebra extension to the standard algebra of programming.

In particular, the paper calculates 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*.

Calculating fault propagation in functional programs

Daniel R. Murta , José N. Oliveira
pg23205@alunos.uminho.pt, jno@di.uminho.pt

May. 2013

Abstract

The production of safety critical software is bound to a number of safety and certification standards in which estimating the *risk of failure* plays a central role. Yet risk estimation seems to live outside most programmers' core practice, involving simulation techniques and worst case analysis performed *a posteriori*.

In this paper we propose that risk be constructively handled in functional programming by writing programs which choose between *expected* and *faulty* behaviour and by reasoning about them in a linear algebra extension to the standard algebra of programming.

In particular, the paper calculates 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*.

Tech.Report version: May 3, 2013

Calculating fault propagation in functional programs

Daniel R. Murta José N. Oliveira

HASLab — High Assurance Software Laboratory,
INESC TEC and University of Minho,
4700 Braga, Portugal
pg23205@alunos.uminho.pt jno@di.uminho.pt

Abstract

The production of safety critical software is bound to a number of safety and certification standards in which estimating the *risk of failure* plays a central role. Yet risk estimation seems to live outside most programmers' core practice, involving simulation techniques and worst case analysis performed *a posteriori*.

In this paper we propose that risk be constructively handled in functional programming by writing programs which choose between *expected* and *faulty* behaviour and by reasoning about them in a linear algebra extension to the standard algebra of programming.

In particular, the paper calculates 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*.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance

General Terms Functional programming, fault propagation

Keywords Linear algebra of programming, program transformation, fault propagation, probabilistic risk assessment

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 [12] 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 NASA's procedure guide for PRA [21]:

> *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.*

Reading this quotation one is left with the feeling that PRA takes place *a posteriori*, that is, once the system is built. Even if the wrong interpretation of the pragmatics of PRA, limitations of current programming practice are apparent concerning timely assessment of the risks involved in the future use of the program one is writing at the moment. *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). We will show that FP is well prepared for incorporating risk analysis smoothly 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 [4] into a linear algebra of programming (LAoP) [18].

The basic idea is simple: suppose one writes function $good$ for the intended behaviour of a program and that there is evidence that, with probability p, such behaviour can turn into a bad function. Using the *probabilistic choice* combinator $(\cdot \ _p\diamond \ \cdot)$ of [15, 18], 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 LAoP [18] is intended for. This paper investigates, in particular, a quantitative extension of the so-called *mutual recursion law* [4] which underpins 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.

We will illustrate our results in two ways: either by running our programs as probabilistic (monadic) functions written in Haskell using the PFP library of [7], or running finite approximations of them directly as matrices in MATLAB [1].

Paper outline The following section presents two motivating examples of 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 then (section 4) extended to the probabilistic setting in which risk is to be evaluated. A basis for this

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

is given in section 5, where LAoP [18] is introduced 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 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 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;
};
```

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 the 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* [22] is a more and more widespread technique for quality software 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 [6].

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 [7]. Take the two versions of faulty addition given above as examples: the first can be expressed by turning $(+)$ into the probabilistic function

$$fadd\ x = id\ {}_p\diamond\ (x+)$$

($fadd$ 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 is expressed by probabilistic choice

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

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

$$fadd\ x = (\underline{0}\ {}_q\diamond\ id)\ {}_p\diamond\ (x+)$$

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*, which is termed $Dist$ in the PFP library [7] we shall be using in the sequel. Moreover, probabilistic functions can be reasoned about using the laws of monads, explicitly as in [9] or implicitly as in the probabilistic notation proposed by Morgan [17] as extension to the standard Eindhoven quantifier calculus [1].

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 $Dist$ [18, 19]. This offers the possibility of using the rich field of *linear algebra* to calculate with probabilistic functions, in the same way relation algebra has been used [4] to reason 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 [4]. The shift from a qualitative to a quantitative semantics is therefore rather smooth — the game is the same, the move ensured by the change of underlying category [2]. Reference [18] includes an example of what might be refereed to as *fault-fusion*: the risk of the whole misbehaving can be expressed in terms of the risk of the parts misbehaving wherever such 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* (or Fokkinga) law enabling systems of mutually recursive functions to be merged into a single, more efficient function [4]. 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:

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

The linear version encoded in the C program given above is obtained by pairing fib with its *derivative*, $f\ n = fib\ (n+1)$: [3]

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

The pairing of the two functions,

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

[2] This can also be observed in other areas, for instance weighted versus standard finite-state automata [19].

[3] Since $f\ 0 = fib\ 1 = 1$ and $f\ (n+1) = fib\ (n+2) = fib\ n + fib\ (n+1) = fib\ n + f\ n$.

can be expressed primitive recursively by

$$(fib, f)\ 0 = (fib\ 0, f\ 0) = (0, 1)$$
$$(fib, f)\ (n+1) = (f\ n, fib\ n + f\ n)$$

or by the equivalent

$$(fib, f)\ 0 = (0, 1)$$
$$(fib, f)\ (n+1) = (y, x+y)\ \mathbf{where}\ (x, y) = (fib, f)\ n$$

itself the same as

$$(fib, f) = \mathsf{for}\ loop\ (0, 1)$$
$$\quad \mathbf{where}\ loop\ (x, y) = (y, x+y)$$

by introduction of the for *loop* combinator,

$$\mathsf{for}\ b\ i\ 0 = i$$
$$\mathsf{for}\ b\ i\ (n+1) = b\ (\mathsf{for}\ b\ i\ n)$$

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* [4] of natural numbers). Therefore, we can define

$$fibl\ n =$$
$$\quad \mathbf{let}\ (x, y) = \mathsf{for}\ loop\ (0, 1)\ n$$
$$\quad\quad loop\ (x, y) = (y, x+y)$$
$$\quad \mathbf{in}\ x$$

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

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

arise from $(n+1)^2 = n^2 + 2n + 1$ and introduction of function $odd\ n = 2\ n + 1$, thus named because $2\ n + 1$ is the n-th odd number.[4] 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: [5]

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

Clearly, each recursive function above and its linear version are, extensionally, the same function. Let us now see what happens when 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 [7]. Take the recursive version of *fib* given in the beginning of section 3 and "monadify it" into:

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

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. We just replace $return\ (x+y)$ (perfect addition) by $fadd\ 0.1\ x\ y$ and run test cases, eg. [6]

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

We see that the correct behaviour (100% chances 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

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

relying on the monadic extension of the for combinator:

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

To inject into $mfibl$ the same fault injected before into $mfib$ amounts to replacing 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

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

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

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

$$msq\ 0 = return\ 0$$
$$msq\ (n+1) = \mathbf{do}\ \{m \leftarrow msq\ n; fadd\ 0.1\ m\ (2 * n + 1)\}$$

and the linear one:

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

[4] That is, the square of a natural number always is a sum of odd numbers.

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

[6] 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 a realistic PRA [21], where figures such as small as 1.0E-7 are "acceptable" risks.

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% 3 10.0%	4 90.0% 3 10.0%
3	9 81.0% 5 10.0% 8 9.0%	9 81.0% 5 10.0% 8 9.0%
⋮	⋮	⋮
6	36 59.0% 11 10.0% 20 9.0% 27 8.1% 32 7.3% 35 6.6%	36 59.0% 11 10.0% 20 9.0% 27 8.1% 32 7.3% 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,

$$M = \begin{array}{c} \\ True \\ \\ False \end{array} \begin{array}{c} \begin{array}{cc} False & True \end{array} \\ \begin{pmatrix} 0.95 & 0 \\ \\ 0.05 & 1.00 \end{pmatrix} \end{array} \tag{1}$$

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

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 [14]. 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 be the Kleisli category induced by the distribution monad. In the example above, $f : Bool \to Dist\ Bool$ is represented by a matrix of type $M : Bool \to Bool$ (1) on the Kleisli-category side.

Let notation $[\![f]\!]$ mean the matrix which represents probabilistic function f in such a 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 [7] 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,

$$[\![f \bullet g]\!] = [\![f]\!] \cdot [\![g]\!] \tag{2}$$

and probabilistic function choice is given by

$$[\![f\ {}_p\diamond\ g]\!] \quad = \quad p[\![f]\!] + (1-p)[\![g]\!] \tag{3}$$

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, $[\![return]\!] = id$. 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.[8] We will write $[\![f]\!]$ as shorthand for $[\![return \cdot f]\!]$ and therefore will rely on fact $(f\ a)\ [\![f]\!]\ a = 1$, 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,

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

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 the matrix equalities,

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

which can be reduced to a single equality,

$$[\![\mathsf{for}\ b\ i]\!] \cdot [[\![\underline{0}]\!]|[\![\mathsf{succ}]\!]] = [[\![\underline{i}]\!]|([\![b]\!] \cdot [\![\mathsf{for}\ b\ i]\!])] \tag{4}$$

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 in [14], this combinator (which corresponds to the "junc" operator in [4]) is a universal construction in any category of matrices, therefore satisfying (among others) the fusion law

$$P \cdot [M|N] \quad = \quad [P \cdot M|P \cdot N] \tag{5}$$

and (for suitably typed matrices) the equality law,

$$[M|N] = [P|Q] \quad \equiv \quad M = P \wedge N = Q \tag{6}$$

both silently used in the derivation above.

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

[7] 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 [19].

[8] 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. A Dirac distribution is one whose support is a singleton set, the unique element of which is offered with 100% probability.

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

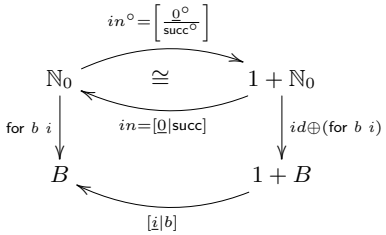
Second, $[\underline{i}|(b \cdot (\mathsf{for}\ b\ i))]$ can be factored into composition $[\underline{i}|b] \cdot (id \oplus (\mathsf{for}\ b\ i))$, since absorption law

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

holds [14], 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,

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

which corresponds to the typed matrix diagram which follows,



where symbol $\cong$ indicates that function $\mathsf{in} = [\underline{0}|\mathsf{succ}]$ 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 $\left[\frac{\cdot}{\cdot}\right]$ which is the converse dual of $[\cdot|\cdot]$:

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

Why does this diagram matter? First, it can be recognized as an instance of a *catamorphism* diagram [4], 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 $\underline{i} : 1 \to \mathbb{N}_0$ — 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 $\mathsf{for}\ b\ i$ is a solution to the equation

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

but it turns out that this is the unique solution:[9]

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

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

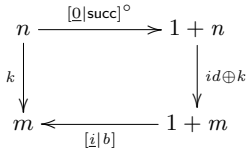
$$k = \underline{i} \cdot \underline{0}^\circ + b \cdot k \cdot \mathsf{succ}^\circ \tag{10}$$

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

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

Equation (10) tells how the matrix $k = \mathsf{for}\ b\ i$ 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 $b \cdot \underline{i} \cdot \underline{0}^\circ \cdot \mathsf{succ}^\circ$, and so on. For sharp b, this is $\underline{(b\ i)} \cdot 1^\circ$, the n-th entry being $\underline{(b^n\ i)} \cdot n^\circ$. 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 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.

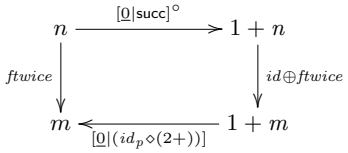
Let us see an example: suppose we want to emulate a fault in the *odd* function, $odd = (1+) \cdot (2*)$, in which $(2*) = \mathsf{for}\ (2+)\ 0$ is disturbed by the propagation of the same fault of addition we have seen before:

$$ftwice = mfor\ (fadd\ 0.1\ 2)\ 0$$

For instance, $ftwice\ 4$ 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 building each arrow of the diagram in MATLAB. The probabilistic choice in the corresponding instance of (10),

$$k \quad = \quad \underline{0} \cdot \underline{0}^\circ + (id\ _p\diamond\ (2+)) \cdot k \cdot \mathsf{succ}^\circ \tag{12}$$

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
```

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

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

[9] The argument is the same as in [4] just by replacing the powerset monad by the distribution monad.

```
            faddk(0.1,2,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* n, for $n = 0 \,..\, 4$.

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 [11], in general) of two (sharp) functions f and g, defined by

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

where $(f, g) : A \to B \times C$ for $f : A \to B$ and $g : A \to C$. This tupling operator is known as *split* [4] in the functional setting or as *fork* in the relational one [8, 20]. Macedo [13] 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

$$(b, c)\ (M \vartriangle N)\ a \quad = \quad (b\ M\ a) \times (c\ N\ a) \tag{13}$$

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

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

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

$$u \vartriangle v = u \otimes v \tag{15}$$

and commutes with matrix junc'ing via the *exchange law* [13]:

$$[M|N] \vartriangle [P|Q] = [(M \vartriangle P)|(N \vartriangle Q)] \tag{16}$$

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:

$$k = f \vartriangle g \quad \equiv \quad fst \cdot k = f \wedge snd \cdot k = g \tag{17}$$

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*,

$$(f \vartriangle g) \cdot h = (f \cdot h) \vartriangle (g \cdot h) \tag{18}$$

reconstruction,

$$k = (fst \cdot k) \vartriangle (snd \cdot k) \tag{19}$$

and pairwise *equality*:

$$k \vartriangle h = f \vartriangle g \quad \equiv \quad k = f \wedge h = g \tag{20}$$

This makes it easy to prove the mutual recursion law, below instantiated to for-loops but (as is well-known) valid for any inductive type (eg. lists, trees etc) [4, 11]; $\mathsf{F}\ f$ abbreviates $id \oplus f$:

$$\begin{array}{ll} & f \vartriangle g = \mathsf{for}\ (h \vartriangle k)\ (i, j) \\ \equiv & \quad \{\ \text{universal property (10)}\ \} \\ & (f \vartriangle g) \cdot \mathsf{in} = [\underline{(i,j)}|(h \vartriangle k)] \cdot \mathsf{F}\ (f \vartriangle g) \\ \equiv & \quad \{\ \text{fusion (18) ; constant functions}\ \} \\ & (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = [(\underline{i} \vartriangle \underline{j})|(h \vartriangle k)] \cdot \mathsf{F}\ (f \vartriangle g) \\ \equiv & \quad \{\ \text{exchange law (16)}\ \} \\ & (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = ([\underline{i}|h] \vartriangle [\underline{j}|k]) \cdot \mathsf{F}\ (f \vartriangle g) \\ \equiv & \quad \{\ \text{fusion (18) again}\ \} \\ & (f \cdot \mathsf{in}) \vartriangle (g \cdot \mathsf{in}) = ([\underline{i}|h] \cdot \mathsf{F}\ (f \vartriangle g)) \vartriangle ([\underline{j}|k] \cdot \mathsf{F}\ (f \vartriangle g)) \\ \equiv & \quad \{\ \text{equality (20)}\ \} \\ & \left\{ \begin{array}{l} f \cdot \mathsf{in} = [\underline{i}|h] \cdot \mathsf{F}\ (f \vartriangle g) \\ g \cdot \mathsf{in} = [\underline{j}|k] \cdot \mathsf{F}\ (f \vartriangle g) \end{array} \right. \end{array}$$

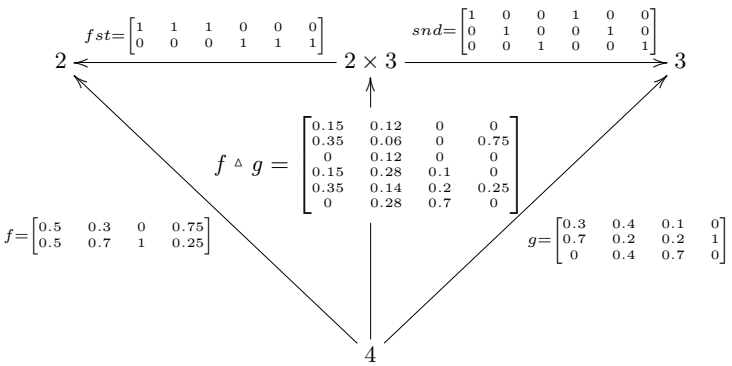
Read in reverse direction, this reasoning explains how two recursive, mutually dependent functions f and g (regarded as matrices) combine with each other into one single function $f \vartriangle g$, from which one can extract both f and g by projecting according to the *cancellation* rule,

$$fst \cdot (f \vartriangle g) = f\ \wedge\ snd \cdot (f \vartriangle g) = g \tag{21}$$

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 [4, 11] among several others.

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



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

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

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

However, *reconstruction* (19) will not survive the probabilistic extension. This is because not every CS-matrix $k : A \to B \times C$ 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 — the expected equivalence (17) is only an implication,

$$k = f \vartriangle g \quad \Rightarrow \quad fst \cdot k = f \wedge snd \cdot k = g \tag{22}$$

ensuring existence but not uniqueness. The proof of (22), which is equivalent to cancellation (21) — substitute k and simplify — can be found in appendix 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 $k \vartriangle h$ 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 sharp [10], as the straightforward proof in 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 $mfibl$ 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).

[10] The same happens with *forks* in relation algebra [4].

7. Asymmetric Khatri-Rao product

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

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

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 [4], 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 leads into the following conjecture:

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

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

In the proof of this conjecture we will resort to the definition of (typed) matrix composition, for $M : B \to C$ and $N : A \to B$,

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

and to two rules given in [19] which interface index-free and index-wise matrix notation, where N is an arbitrary matrix and f, g are functional (ie. sharp) matrices: [11]

$$y(f \cdot N)x \quad = \quad \langle \sum\ z\ :\ y = f\ z :\ zNx \rangle \tag{24}$$
$$y(g^{\circ} \cdot N \cdot f)x \quad = \quad (g\ y)\ N\ (f\ x) \tag{25}$$

Let us suppose $fst \cdot k$ in (19) is sharp. We introduce $f : A \to B$ as denotation of 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

$$\langle \sum\ c\ ::\ (f\ a, c)\ k\ a \rangle = 1 \tag{26}$$
$$\langle \sum (b, c)\ :\ (b \neq f\ a) :\ ((b, c)\ k\ a) \rangle = 0 \tag{27}$$

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

$$(b, c)\ m\ a = (b\ f\ a) \times \langle \sum\ b'\ ::\ (b', c)\ k\ a \rangle \tag{28}$$

since $f = fst \cdot k$ an snd is sharp (24). Our aim is to prove that $m = k$.

Case $b \neq f\ a$**:** In this case $b\ f\ a = 0$ and (28) yields $(b, c)\ m\ a = 0$. From (27) we also get $(b, c)\ k\ a = 0$ and so $m = k$ for this case.

[11] These rules, expressed in the style of the Eindhoven quantifier calculus [1], are derived in [19]. We adopt the Eindhoven notation [1, 17] 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 being true for all x. (The convention is to omit R in this case.)

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

$$
\begin{array}{ll}
& (f\ a, c)\ m\ a \\
= & \quad \{\ (28)\ ;\ (b\ f\ a) = 1 \text{ for } b = f\ a\ \} \\
& \langle \sum\ b'\ ::\ (b', c)\ k\ a \rangle \\
= & \quad \{\ b' = f\ a\ \vee\ b' \neq f\ a\ \} \\
& \langle \sum\ b'\ :\ b' = f\ a\ \vee\ b' \neq f\ a :\ (b', c)\ k\ a \rangle \\
= & \quad \{\ \text{split summation ; one-point over } b' = f\ a\ \} \\
& ((f\ a, c)\ k\ a) + \langle \sum\ b'\ :\ b' \neq f\ a :\ (b', c)\ k\ a \rangle \\
= & \quad \{\ (27)\ \} \\
& (f\ a, c)\ k\ a
\end{array}
$$

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. It remains to prove assumptions (26) and (27):

Proof of (26) This equality arises from rule (24):

$$
\begin{array}{ll}
& \langle \sum\ c\ ::\ (f\ a, c)\ k\ a \rangle = 1 \\
\equiv & \quad \{\ \text{one-point rule}\ \} \\
& \langle \sum\ b, c\ :\ f\ a = b :\ (b, c)\ k\ a \rangle = 1 \\
\equiv & \quad \{\ b = \mathit{fst}\ (b, c)\ ;(24)\ \} \\
& (f\ a)\ (\mathit{fst} \cdot k)\ a = 1 \\
\equiv & \quad \{\ f = \mathit{fst} \cdot k\ \} \\
& (f\ a)\ f\ a = 1 \\
\equiv & \quad \{\ f \text{ is sharp}\ \} \\
& \mathit{true}
\end{array}
$$

□

Proof of (27) This equality arises from k being probabilistic:

$$
\begin{array}{ll}
& \langle \sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 0 \\
\equiv & \quad \{\ 1 + 0 = 1\ \} \\
& 1 + \langle \sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 1 \\
\equiv & \quad \{\ (26)\ \} \\
& \langle \sum\ c\ ::\ (f\ a, c)\ k\ a \rangle + \\
& \quad \langle \sum\ b, c\ :\ b \neq f\ a :\ (b, c)\ k\ a \rangle = 1 \\
\equiv & \quad \{\ \text{merge quantifiers}\ \} \\
& \langle \sum\ (b, c)\ ::\ (b, c)\ k\ a \rangle = 1 \\
\equiv & \quad \{\ k \text{ is probabilistic}\ \} \\
& \mathit{true}
\end{array}
$$

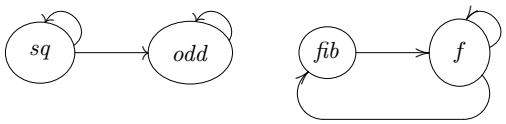
□

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. [12] 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 $k = \mathsf{for}\ b\ i$, 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,

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

and run eg. $msqlo\ 5$, 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,

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

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

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

and its linear version,

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

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

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%

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 [4].

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*:

$$\mathsf{for}\ f\ (a\ {}_p\diamond\ b) \quad = \quad (\mathsf{for}\ f\ a)\ {}_p\diamond\ (\mathsf{for}\ f\ b) \tag{29}$$

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 [4] is adopted to give freedom to the base case to be any probabilistic function h of its type. Thus (9) becomes, for $\mathsf{F}\ f = id \oplus f$,

$$k = (\!|[h|f]|\!) \quad \equiv \quad k \cdot \mathsf{in} = [h|f] \cdot (\mathsf{F}\ k) \tag{30}$$

Clearly,

$$\mathsf{for}\ f\ a \quad = \quad (\!|[\underline{a}|f]|\!) \tag{31}$$

holds. In (29), abbreviation $\mathsf{for}\ f\ (a\ {}_p\diamond\ b)$ replacing $(\!|[(\underline{a}\ {}_p\diamond\ \underline{b})|f]|\!)$ is welcome as it enhances readability.

The proof of (29) is given in appendix A. It relies on properties of probabilistic choice already given in [18], namely *choice-fusion*

$$(f\ {}_p\diamond\ g) \cdot h \quad = \quad (f \cdot h)\ {}_p\diamond\ (g \cdot h) \tag{32}$$

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

and the *exchange law*:

$$[f|g]\ {}_p\diamond\ [h|k] \quad = \quad [(f\ {}_p\diamond\ h)|(g\ {}_p\diamond\ k)] \tag{34}$$

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 [13],

$$k = \mathsf{fold}\ f\ d \quad \equiv \quad k \cdot \mathsf{in} = [d|f] \cdot (\mathsf{F}\ k) \tag{35}$$

where $\mathsf{F}\ k = id \oplus (id \otimes k)$ and $\mathsf{in} = [\mathsf{nil}|\mathsf{cons}]$ is the initial algebra of sequences, for (in Haskell notation) $\mathsf{nil}\ \_ = [\,]$ and $\mathsf{cons}\ (h, t) = h : t$. Besides the direct sum $(id \oplus \cdot)$ splitting base from recursive case, as with for, recursive pattern $\mathsf{F}\ k$ 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 distribution. For sharp functions, $\mathsf{fold}\ f\ \underline{u}$ means the same as $\mathsf{foldr}\ (curry\ f)\ u$ 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):

$$\begin{aligned} \mathsf{fold}\ f\ d \quad &= \quad [d|f \cdot (id \otimes (\mathsf{fold}\ f\ d))] \cdot \left[\frac{\mathsf{nil}^\circ}{\mathsf{cons}^\circ}\right] \\ &\equiv \qquad \{\ \text{divide-and-conquer (11)}\ \} \\ &\qquad d \cdot \mathsf{nil}^\circ + f \cdot (id \otimes (\mathsf{fold}\ f\ d)) \cdot \mathsf{cons}^\circ \end{aligned} \tag{36}$$

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

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

where $lose = snd$ and $send = \mathsf{cons}$. For instance, for $p = 0.1$, distribution $fcat$ `"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:

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

For instance, for $q = 0.15$, distribution $fcount$ `"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 $fcount \cdot fcat$? We could try specific runs, eg. $(fcount \cdot fcat)$ `"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 (35) in the standard way [4] and also in the probabilistic setting:

$$k \cdot (\mathsf{fold}\ g\ e) = \mathsf{fold}\ f\ d \quad \Leftarrow \quad k \cdot [e|g] = [d|f] \cdot (\mathsf{F}\ k) \tag{37}$$

[13] Both are instances of the generic *catamorphism* construct, as already mentioned.

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

$$
\begin{array}{ll}
 & fcount \cdot fcat = \mathsf{fold}\ x\ y \\
\Leftarrow & \quad \{\ \text{fold fusion (37) ; definition of } fcat\ \} \\
 & fcount \cdot [\mathsf{nil}|(lose\ {}_p\diamond\ send)] = [x|y] \cdot (\mathsf{F}\ fcount) \\
\equiv & \quad \{\ (5)\ ;\ \text{definition of } \mathsf{F};\ (7)\ ;\ (6)\ \} \\
 & \left\{ \begin{array}{l} fcount \cdot \mathsf{nil} = x \\ fcount \cdot (lose\ {}_p\diamond\ send) = y \cdot (id \otimes fcount) \end{array} \right. \\
\equiv & \quad \{\ fcount \cdot \mathsf{nil} = \underline{0}\ \} \\
 & \left\{ \begin{array}{l} x = \underline{0} \\ fcount \cdot (snd\ {}_p\diamond\ \mathsf{cons}) = y \cdot (id \otimes fcount) \end{array} \right.
\end{array}
$$

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

$$
\begin{array}{ll}
 & fcount \cdot (snd\ {}_p\diamond\ \mathsf{cons}) = y \cdot (id \otimes fcount) \\
\equiv & \quad \{\ \text{choice fusion (33)}\ \} \\
 & (fcount \cdot snd)\ {}_p\diamond\ (fcount \cdot \mathsf{cons}) = y \cdot (id \otimes fcount) \\
\equiv & \quad \{\ \text{unfolding } fcount \cdot \mathsf{cons}\ \} \\
 & (fcount \cdot snd)\ {}_p\diamond\ ((id\ {}_q\diamond\ \mathsf{succ}) \cdot snd \cdot (id \otimes fcount)) \\
 & = y \cdot (id \otimes fcount) \\
\equiv & \quad \{\ \text{free theorem of } snd\ \} \\
 & (fcount \cdot snd)\ {}_p\diamond\ ((id\ {}_q\diamond\ \mathsf{succ}) \cdot fcount \cdot snd) \\
 & = y \cdot (id \otimes fcount) \\
\equiv & \quad \{\ \text{choice fusion (32)}\ \} \\
 & (id\ {}_p\diamond\ (id\ {}_q\diamond\ \mathsf{succ})) \cdot fcount \cdot snd = y \cdot (id \otimes fcount) \\
\equiv & \quad \{\ \text{free theorem of } snd \text{ again}\ \} \\
 & (id\ {}_p\diamond\ (id\ {}_q\diamond\ \mathsf{succ})) \cdot snd \cdot (id \otimes fcount) = y \cdot (id \otimes fcount) \\
\Leftarrow & \quad \{\ \text{Leibniz } (id \otimes fcount \text{ cancelled from both sides})\ \} \\
 & y = (id\ {}_p\diamond\ (id\ {}_q\diamond\ \mathsf{succ})) \cdot snd
\end{array}
$$

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

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

in which the probabilistic definition of y combines the choices according to (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. Conclusions

The production of *safety critical* software is bound to a number of safety and 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 [21]. The UK MoD Defence Standard 00-56 [16] 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 [15]. Quantitative methods face another problem, diagnosed in [17]: 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 [18], a strategy which fits into the calculational style of functional program development based on its algebra of programming [4].

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. [14]

The core of the paper has shown how to calculate the propagation of faults across standard program transformation techniques known as *tupling* [11] and *fusion* [10]. This enables one to find conditions for the *fault of the whole* to be expressed in terms of the *faults of its parts*.

11. Future work

In our experiments with probabilistic mutual recursion transformation, 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 be able 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

[14] 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 should be credited to M. Erwig and S. Kollmansberger [7].

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?

Another follow-up of the strategy put forward in this paper is its application to fault-propagation in component-oriented software systems. Reference [5] quantifies 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 [3] in which components represented by *coalgebras* [2] extended probabilistically, adding to the coalgebraic matrices of [19] a *behaviour* monad inside the *distribution* one. We hope to show that the linear algebra of programming is a wide-range formalism suitable to support quantitative methods in the software sciences, in general.

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 ideas of this paper were briefly presented.

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

References

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

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

[3] L.S. Barbosa, D.R. Murta, and J.N. Oliveira. Introducing fault propagation in a software component calculus, 2013. In preparation.

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

[5] 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.

[6] 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.

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

[8] M.F. Frias. Fork algebras in algebra, logic and computer science, 2002. Logic and Computer Science. World Scientific Publishing Co.

[9] 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.

[10] T. Harper. A library writer's guide to shortcut fusion. In *Haskell Symposium 2011*, September 2011.

[11] 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.

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

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

[14] H.D. Macedo and J.N. Oliveira. Matrices As Arrows! A Biproduct Approach to Typed Linear Algebra. In *MPC'10*, volume 6120 of *LNCS*, pages 271–287. Springer, 2010.

[15] A. McIver and C. Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems*. Monographs in Computer Science. Springer-Verlag, 2005.

[16] 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`.

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

[18] J.N. Oliveira. Towards a linear algebra of programming. *Formal Asp. Comput.*, 24(4-6):433–458, 2012.

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

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

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

[22] J. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley & Sons, 1997. 416 pages.

A. Proofs in appendix

Proof of cancellation (21) Base case (f and g are column vectors) [15]:

$$
\begin{array}{ll}
 & fst \cdot (f \vartriangle g) = f \wedge snd \cdot (f \vartriangle g) = g \\
\equiv & \quad \{\ fst = id \otimes ! \text{ and } snd = ! \otimes id\ \} \\
 & (id \otimes !) \cdot (f \vartriangle g) = f \wedge (! \otimes id) \cdot (f \vartriangle g) = g \\
\equiv & \quad \{\ \text{for vectors, } f \vartriangle g = f \otimes g \text{ (15)}\ \} \\
 & (id \otimes !) \cdot (f \otimes g) = f \wedge (! \otimes id) \cdot (f \otimes g) = g \\
\equiv & \quad \{\ \text{functor-}\cdot \otimes \cdot\text{; natural-}id\ \} \\
 & f \otimes (! \cdot g) = f \wedge (! \cdot f) \otimes g = g \\
\equiv & \quad \{\ g \text{ is probabilistic, therefore } ! \cdot f = ! \cdot g = ! \text{ [18]}\ \} \\
 & f \otimes ! = f \wedge ! \otimes g = g \\
\equiv & \quad \{\ 1 \xleftarrow{!} 1\ = 1 \text{ and } M \otimes 1 = M\ \} \\
 & f = f \wedge g = g
\end{array}
$$

□

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

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

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

□

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

Proof of base-case fault propagation (29) Clearly, by (31) and universal property (30), our target (29) re-writes to the equality

$$
\begin{array}{l}
((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b)) \cdot \mathsf{in} = \\
\qquad\qquad [(\underline{a}\ {}_p\Diamond\ \underline{b})|f] \cdot (\mathsf{F}\ ((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b)))
\end{array}
$$

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

$$
\begin{array}{ll}
& ((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b)) \cdot \mathsf{in} \\
= & \quad \{\ \text{choice-fusion (32)}\ \} \\
& (\mathsf{for}\ f\ a \cdot \mathsf{in})\ {}_p\Diamond\ (\mathsf{for}\ f\ b \cdot \mathsf{in}) \\
= & \quad \{\ \text{(31) and (30), twice}\ \} \\
& ([\underline{a}|f] \cdot \mathsf{F}\ (\mathsf{for}\ f\ a))\ {}_p\Diamond\ ([\underline{b}|f] \cdot \mathsf{F}\ (\mathsf{for}\ f\ b))) \\
= & \quad \{\ \mathsf{F}\ f = id \oplus f\ ;\ [M|N] \cdot (P \oplus Q) = [M \cdot P|N \cdot Q]\ \} \\
& [\underline{a}|(f \cdot (\mathsf{for}\ f\ a))]\ {}_p\Diamond\ [\underline{b}|(f \cdot (\mathsf{for}\ f\ b))] \\
= & \quad \{\ \text{exchange law (34)}\ \} \\
& [(\underline{a}\ {}_p\Diamond\ \underline{b})|((f \cdot \mathsf{for}\ f\ a)\ {}_p\Diamond\ (f \cdot \mathsf{for}\ f\ b))] \\
= & \quad \{\ \text{choice-fusion (33)}\ \} \\
& [(\underline{a}\ {}_p\Diamond\ \underline{b})|(f \cdot ((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b)))] \\
= & \quad \{\ [M|N] \cdot (P \oplus Q) = [M \cdot P|N \cdot Q]\ \} \\
& [(\underline{a}\ {}_p\Diamond\ \underline{b})|f] \cdot (id \oplus ((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b))) \\
= & \quad \{\ \mathsf{F}\ f = id \oplus f\ \} \\
& [(\underline{a}\ {}_p\Diamond\ \underline{b})|f] \cdot (\mathsf{F}\ ((\mathsf{for}\ f\ a)\ {}_p\Diamond\ (\mathsf{for}\ f\ b)))
\end{array}
$$

□

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

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

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

$$
\begin{array}{ll}
& (b, c)\ ((M \vartriangle N) \cdot h)\ a \\
= & \quad \{\ \text{(25) for } h \text{ a standard function}\ \} \\
& (b, c)\ (M \vartriangle N)\ (h\ a) \\
= & \quad \{\ \text{pointwise Khatri-Rao (13)}\ \} \\
& (b\ M\ (h\ a)) \times (c\ N\ (h\ a)) \\
= & \quad \{\ \text{(25) for } h \text{ a standard function}\ \} \\
& b\ (M \cdot h)\ a \times c\ (N \cdot h)\ a \\
= & \quad \{\ \text{pointwise Khatri Rao (13) — twice}\ \} \\
& (b, c)\ ((M \cdot h) \vartriangle (N \cdot h))\ a
\end{array}
$$

□