Calculating Fault Propagation in Functional Programs using the LAoP

J.N. Oliveira

High Assurance Software Laboratory
INESC TEC and University of Minho, Portugal

Joint work with D. Murta
QAIS Project - Grant PTDC/EIA-CCO/122240/2010

Motivation

Research questions:

- How do software **faults** propagate in computer programs?
- Can faulty behavior be predicted in some way, eg. by **calculation**?
- Are there versions of the same program or system which are "better" than others concerning **fault propagation**?

In this talk:

- Faulty behavior can be mimicked **probabilistically**
- Faults can be **injected** and simulated using **monadic** programming
- Better: Instead repeated simulation, programs can be converted into (inductive) matrices and **reasoned** about in **LAoP**, an extension of the AoP towards **quantitative** reasoning.

Trustworthiness in software design

Two dual approaches to software trustworthiness:

1. **"Angelic"** — prevent bad things from happening — **weakest pre-conditions** (Dijkstra): the least one should impose for a program not blow up.
2. **"Demonic"** — force bad things to happen — **strongest post-conditions**: evaluate worst blow-up scenario arising from fault.

Fault injection: expensive techniques and tools based on extensive simulation of faults (eg. CSW **Xception**+**Xtract**).

Can't fault propagation be **calculated** as a pen & paper exercise?

Example: fault-injected multiplication

Safe multiplication (over $\mathbb{N}_0$):

$$(a*) = \mathbf{for}\ (a+)\ 0$$

that is,

$$a * 0 = 0$$
$$a * (n+1) = a + a * n$$

Bad multiplication, **fault-injected** — 5% probability of a wrong base case

$$a * 0 =_{.95} 0$$
$$a * 0 =_{.05} a$$
$$a * (n+1) =_{1} a + a * n$$

in "extended" functional notation.

Example: fault propagation

What is the **fault pattern** in the *pipeline*

$$f\ n \quad = \quad fsucc(fmul\ a\ n)$$

where $fsucc$ is a faulty **successor** function,

$$\begin{aligned} fsucc\ n &=_q \quad n+1 \\ fsucc\ n &=_{1-q} \quad n \end{aligned}$$

and $fmul$ is the even more seriously faulty multiplication,

$$\begin{aligned} & fmul\ a\ 0 = 0 \\ & fmul\ a\ (n+1) \ =_p \ fmul\ a\ n \\ & fmul\ a\ (n+1) \ =_{1-p} \ a + fmul\ a\ n \end{aligned}$$

for $0 \leq p, q \leq 1$ in general?

Implementing the "extended notation"

How do we implement our **probability annotated** (Haskell) programs?

- We propose to use **distribution**-valued functions.

Do such functions **compose**?

- Yes, provide you program this.

Do you need *heavy machinery* to program in such a way?

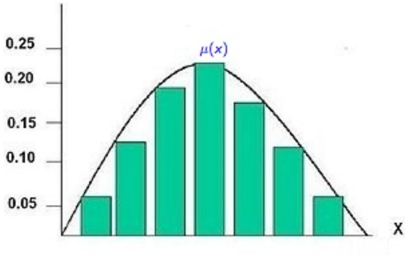
- Not in **Haskell** — distributions form a **monad** and therefore handling distributions is as easy as handling lists, for instance.
- **PFP library** by Erwig and Kollmansberger (2006) offers the distribution monad and a wide range of utility functions on probabilities.

About the distribution monad

The datatype of **distributions** on X which supports the monad:

$$\mathcal{D}X \quad = \quad \{\mu : X \to [0,1] \mid \sum_{x \in X} \mu(x) = 1\} \qquad (1)$$

For instance:



Standard monadic function *return a* is the **Dirac distribution** μ such that $\mu\ a = 1$ and $\mu\ x = 0$ for $x \neq a$.

Using the PFP library

The monad

```
newtype Dist a = D {unD :: [(a,ProbRep)]}

instance Monad Dist where
  return x = D [(x,1)]
  d >>= f  = D [(y,q*p) | (x,p) <- unD d, (y,q) <- unD (f x)]
  fail _   = D []
```

is available from `Probability.hs`.

Example: base-case fault-injected multiplication

```
a * 0 = D [(0,0.95),(a,0.05)]
a * (n+1) = do x <- a * n
               return (a + x)
```

Other (generic) examples in PFP

Faulty **add** : yields 0 with probability *p*

```
fadd p a x = choose p 0 (a+x)
```

Faulty **multiplication**: **propagates** *fadd* faults

```
fmul p a 0 = return 0
fmul p a n = do { x <- fmul p a (n-1) ;
                  fadd p a x
                }
```

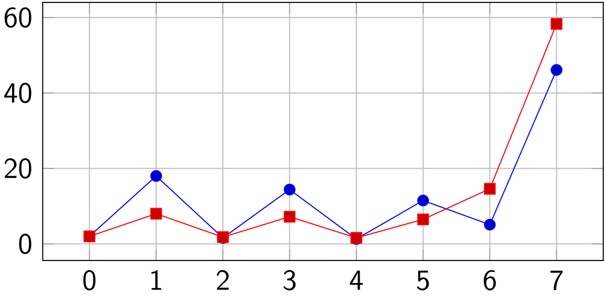
Faulty **succ**: does nothing with probability *q*

```
fsucc q = schoice q id succ
```

Functions *choose* and *schoice* are suitable library functions.

Experiments

Running (Haskell) composition $fsucc\ q \bullet fmul\ p$, yields for $a = 2$ and input 3 ($1 + 2 \times 3 = 7$),



for p = 20%, q = 10% (in blue) and for p = 10%, q = 20% (in red).

However

Problem:

- Complete probabilistic model but...
- Combinatorial **explosion** of recursive probability layers limits experiments
- Would need **Monte Carlo** simulation and the like...

Alternative:

Reason about the monadic code (Gibbons & Hinze).

Our approach:

(Pointwise) monads are better for **programming** *than for* **calculating**. *Fortunately, they "never come alone"...*

Winding back: ND functions

Nondeterministic outputs — set-valued functions are relations

$$f = \Lambda R \quad \Leftrightarrow \quad \langle \forall\, b, a \; :: \; b \; R \; a \Leftrightarrow b \in f \; a \rangle \tag{2}$$

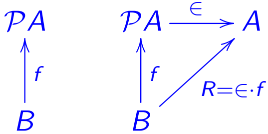
that is,

$$A \to \mathcal{P}B \;\; \underset{\Lambda}{\overset{(\in\cdot)}{\cong}} \;\; A \to B \tag{3}$$

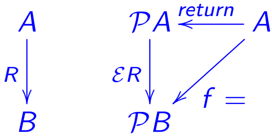
where $A \to B$ on the right hand side is the **relational type** $A \to B$ of all relations $R \subseteq B \times A$.

Nondeterministic functions

An adjunction, offering two ways for reasoning — one relational (**Rel**)



the other monadic (**Set**):



$$f = \mathcal{E}R \cdot return$$
$$where\ (\mathcal{E}R)s = \{b | a \leftarrow s; bRa\}$$

The same duality in "going probabilistic" (next slide).

Probabilistic functions

Outputs become **distributions**,

$$A \to \mathcal{D}B \cong A \to B \tag{4}$$

where $\mathcal{D}B$ is the B-distribution data type

$$\mathcal{D}B = \{\mu \in [0,1]^B \mid \sum_{b \in B} \mu\ b = 1\} \tag{5}$$

and where $[0,1]$ is the interval of all non-negative reals at most 1.

However, what does $A \to B$ on the right hand side of (4) mean?

Probabilistic functions

One has:

$$
\begin{array}{cl}
& A \to [0,1]^B \\
\Leftrightarrow & \{ \text{ uncurrying } \} \\
& A \times B \to [0,1] \\
\Leftrightarrow & \{ \text{ swapping } \} \\
& B \times A \to [0,1]
\end{array}
$$

where $B \times A \to [0,1]$ can be identified with the set of all **matrices** taking elements from $[0,1]$ with as many **columns** (resp. **rows**) as elements in A (resp. B).

Column stochastic matrices

In fact:

$$A \to \mathcal{D}B \quad \cong \quad A \to_{CS} B \tag{6}$$

where CS denotes the **category** of **column-stochastic** matrices (columns in such matrices add up to 1).

Such a **matrix**-transform is captured by the universal property, for all $f :: A \to \mathcal{D}B$ and CS-matrix M:

$$M = [\![f]\!] \quad \Leftrightarrow \quad \langle \forall\ b, a \ :: \ b\ M\ a = (f\ \ a) b \rangle \tag{7}$$

Research question:

> *Is CS "as useful" to probabilistic reasoning as Rel is to non-deterministic reasoning in the AoP (Bird and de Moor, 1997) ?*

Towards a LAoP

My answer:

I believe so — in general and in fault-propagation, in particular

Still, several things to be explained:

- **categories of matrices** — what's this?
- category of **CS matrices** — what's this?
- the **AoP** is pointfree — universal property (7) above is pointwise...

Answering these questions will generalize the **AoP** into something one may identify as a **Linear** Algebra of Programming (**LAoP**) — details in (Oliveira, 2012)

Arrow notation for matrices

In a category of matrices, these are typed: arrow $A \xrightarrow{M} B$ denotes a matrix M from A (source) to B (target).

A, B are types. Writing $B \xleftarrow{M} A$ means the same as $A \xrightarrow{M} B$. We represent source types column-wise and target types rows-wise.

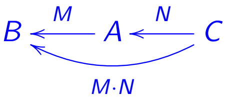
For instance, coefficient matrix aside is of type $3 \leftarrow \{x, y, z\}$.

Matrices of types $A \leftarrow 1$ (resp. $1 \leftarrow A$) are known as column (resp. row) **vectors**.

	x	y	z
1	0	2	-3
2	5	1	3
3	-1	0	2

Arrow notation for matrices

Compositionality — matrices compose with each other:



where

$$b(M \cdot N)c \;=\; \langle \sum a \;::\; (bMa) \times (aNc) \rangle \qquad (8)$$

Matrix **composition** normally referred to as *multiplication*. The minimal algebraic structure for (8) to make sense is that of a **semiring** $(\mathbb{S}; +, \times, 0, 1)$.

Typed linear algebra

For matrices M and N **of the same type** $B \longleftarrow A$, we can extend cell level algebra to matrix level, eg. by **adding** or **multiplying** matrices,

$$M + N \quad , \quad M \times N$$

the latter known as the **Hadamard** product.

Expressions such as eg. $M + N$, $M \times N$ for M and N of different types **won't typecheck**.

The underlying type system is **polymorphic** *and type inference proceeds by* **unification***. For instance, the* **identity matrix** *is of polymorphic type* $A \longleftarrow A$.

$$id = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Converse

Given matrix $n \xleftarrow{M} m$, notation $m \xleftarrow{M^\circ} n$ denotes its transpose, or converse.

Thus M changes shape by turning its rows into columns and vice-versa.

The following idempotence and contravariance laws hold:

$$(M^\circ)^\circ = M \tag{9}$$

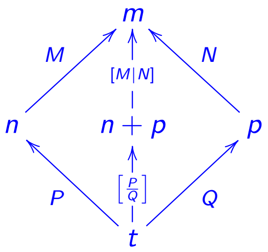
$$(M \cdot N)^\circ = N^\circ \cdot M^\circ \tag{10}$$

Polymorphic (block) combinators

Two ways of putting matrices together to build larger ones:

- $X = [M|N]$ — M and N side by side ("‘junc")
- $X = \left[\frac{P}{Q}\right]$ — P on top of Q ("‘split").

Mind the (polymorphic) types:



(A so-called biproduct)

Blocked linear algebra

Rich set of laws, for instance

- **Divide-and-conquer:**

$$[A|B] \cdot \left[\frac{C}{D}\right] = A \cdot C + B \cdot D \tag{11}$$

- **"Fusion"-laws:**

$$C \cdot [A|B] = [C \cdot A|C \cdot B] \tag{12}$$

$$\left[\frac{A}{B}\right] \cdot C = \left[\frac{A \cdot C}{B \cdot C}\right] \tag{13}$$

Special matrices

The following (Boolean) matrices are relevant:

- The **bottom** matrix $B \xleftarrow{\perp} A$ — wholly filled with 0s
- The **top** matrix $B \xleftarrow{\top} A$ — wholly filled with 1s
- The **identity** matrix $B \xleftarrow{id} B$ — diagonal of 1s
- The **bang** (row) vector $1 \xleftarrow{!} A$ — wholly filled with 1s

Thus, (typewise) **bang** matrices are special cases of **top** matrices:

$$1 \xleftarrow{\top} A \quad = \quad !$$

Also note that, on type $1 \longleftarrow 1$:

$$\top \;=\; ! \;=\; id$$

Useful for matrix index manipulation

Two useful **rules of thumb**,

$$y(f \cdot N)x = \langle \sum z \;:\; y = f\,z : \; zNx \rangle \quad (14)$$

$$y(g^\circ \cdot N \cdot f)x = (g\,y)N(f\,x) \quad (15)$$

(adapted from relation algebra) where N is an arbitrary matrix and f, g are functions.

Wondering about how do *functions* f, g fit into matrix expressions? Easy: every $A \xrightarrow{f} B$ can be represented by a matrix $[\![f]\!]$ of the same type defined by

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

where $y =_{\mathbb{S}} x$ is 1 if $y = x$ and 0 otherwise. Thus matrix $[\![f]\!]$ represents the graph of f.

Useful for matrix index manipulation

Example: $[\![succ]\!]$, *where* $succ\ n = n + 1$, *is the matrix aside. We normally drop the parentheses for improved readability.*

	0	1	2	3	4	5	...
0	0	0	0	0	0	0	
1	1	0	0	0	0	0	
2	0	1	0	0	0	0	
3	0	0	1	0	0	0	
4	0	0	0	1	0	0	
5	0	0	0	0	1	0	
6	0	0	0	0	0	1	
⋮	0	0	0	0	0	0	

In general, the **Eindhoven**-styled **trading**-rule

$$\langle \sum\ x\ :\ p\, x :\ e\, x \rangle \ =\ \langle \sum\ x\ ::\ (p\ x) \times (e\ x) \rangle \qquad (16)$$

holds for Boolean term $p\ x$ which, on the right is such that $p\ x = 1$ if $p\ x$ holds, 0 otherwise.

Matrix transformed probabilistic functions

Given probabilistic function $A \xrightarrow{f} \mathcal{D}B$, its matrix **transform** $A \xrightarrow{[\![f]\!]} B$ is such that

$$! \cdot [\![f]\!] \;=\; ! \qquad (17)$$

that is, all **columns** of $[\![f]\!]$ add up to one.

For $A = B$, probabilistic function f can be regarded as a **Markov chain**.

Example — probabilistic **negation**:

	True	**False**
True	0.1	0.8
False	0.9	0.2

Linear algebra of probabilistic functions

Every *sharp* function is probabilistic — it offers a **Dirac distribution** for every input. This includes the identity function id represented by the identity matrix $[\![id]\!]$.

Compositionality: probabilistic functions compose, under monad-flavoured definition

$$[\![f \bullet g]\!] \quad = \quad [\![f]\!] \cdot [\![g]\!] \tag{18}$$

In monad-speak:

$$[\![\lambda \mathrm{a}.\ \textbf{do}\ \{\mathrm{b} \leftarrow \ \mathrm{g\ a}; \mathrm{f\ \ b}\}\ ]\!] \quad = \quad [\![f]\!] \cdot [\![g]\!]$$

(It is easy to show that (18) preserves probabilistic functions.)

Probabilistic “junc”

Probabilistic $A + B \xrightarrow{[f,g]} \mathcal{D}C$ — run either f or g — transposes into

$$[\![[f,g]]\!] \quad = \quad [[\![f]\!] | [\![g]\!]] \tag{19}$$

where (recall) $[M|N]$ denotes M and N put **side by side**.

Checking the 100% constraint (17):

$$
\begin{array}{ll}
 & ! \cdot [[\![f]\!] | [\![g]\!]] \\
\Leftrightarrow & \qquad \{ \text{ fusion-+ (12) } \} \\
 & [! \cdot [\![f]\!] | ! \cdot [\![g]\!]] \\
\Leftrightarrow & \qquad \{ \ f \text{ and } g \text{ probabilistic (17) ; } [!|!] = ! \ \} \\
 & !
\end{array}
$$

Probabilistic choice

In their programming language *pGCL*, McIver and Morgan (2005) introduce notation

$$prog\ {}_{p}\!\diamond\ prog'$$

as a form of **probabilistic choice** between two branches of a program $prog$, chosen with probability p, and $prog'$ chosen with probability $1 - p$.

This corresponds to the choice between two probabilistic functions f and g **of the same type** defined by

$$[\![f\ {}_{p}\!\diamond\ g]\!] \quad = \quad p[\![f]\!] + (1 - p)[\![g]\!] \tag{20}$$

Probabilistic choice

Probabilistic choice "is probabilistic":

$$
\begin{array}{cl}
& ! \cdot [\![f \ {}_p\!\diamond\, g]\!] \\
= & \qquad \{ \text{ definition (20) ; bilinearity } \} \\
& ! \cdot (p[\![f]\!]) + ! \cdot ((1-p)[\![g]\!]) \\
= & \qquad \{ \ p \text{ is a scalar } \} \\
& p(! \cdot [\![f]\!]) + (1-p)(! \cdot [\![g]\!]) \\
= & \qquad \{ \ f \text{ and } g \text{ are probabilistic } \} \\
& p! + (1-p)! \\
= & \qquad \{ \text{ bilinearity } \} \\
& (p + 1 - p)! \\
= & \qquad \{ \text{ cancellation } \} \\
& !
\end{array}
$$

Properties

Probabilistic choice enjoys many properties easy to derive from the definition, eg. basic

$$f\ {}_p\!\diamond\ f = f \tag{21}$$

$$f\ {}_0\!\diamond\ g = g \tag{22}$$

$$f\ {}_p\!\diamond\ g = g\ {}_{1-p}\!\diamond\ f \tag{23}$$

fusion-laws

$$(f\ {}_p\!\diamond\ g) \bullet h = (f \bullet h)\ {}_p\!\diamond\ (g \bullet h) \tag{24}$$

$$h \bullet (f\ {}_p\!\diamond\ g) = (h \bullet f)\ {}_p\!\diamond\ (h \bullet g) \tag{25}$$

and the **exchange** law:

$$[f, g]\ {}_p\!\diamond\ [h, k] = [f\ {}_p\!\diamond\ h, g\ {}_p\!\diamond\ k] \tag{26}$$

Probabilistic sums

The **direct sum** of two matrices,

$$M \oplus N \;=\; [i_1 \cdot M | i_2 \cdot N] \;=\; \left[\begin{array}{c} M \cdot \pi_1 \\ \hline N \cdot \pi_2 \end{array}\right] \;=\; \left[\begin{array}{c|c} M & 0 \\ \hline 0 & N \end{array}\right] \qquad (27)$$

which has type $A \xrightarrow{M} C$, $B \xrightarrow{N} D$, $A + B \xrightarrow{M \oplus N} C + D$ (a **bifunctor**) enables us to sum probabilistic functions:

$$[\![f \oplus g]\!] \;=\; [\![f]\!] \oplus [\![g]\!]$$

Distribution over choice

$$h \oplus (f \;{}_p\!\diamond\; g) \;=\; (h \oplus f) \;{}_p\!\diamond\; (h \oplus g) \qquad (28)$$

is central to probabilistic function calculation.

Probabilistic recursion

Recall

```
fmul p a 0 = return 0
fmul p a (n+1) = do { x <- fmul p a n ; fadd p a x }
```

Converting this to its **matrix-transpose** we get *fmul* as the unique solution to LAoP equation

$$X = [\underline{0} | (\underline{0}\ {}_p\diamond\ (a+)) \cdot X] \cdot [0 | succ]^\circ$$

where matrix $\underline{0}\ {}_p\diamond\ (a+)$ represents *fadd*. Thus, using divide-and-conquer (11):

$$fmul = \underline{0} \cdot \underline{0}^\circ + fadd \cdot fmul \cdot succ^\circ$$

How do we reason about this equation?

Probabilistic recursion

We might introduce indices, cf.:

$$fmul = \underline{0} \cdot \underline{0}^\circ + fadd \cdot fmul \cdot succ^\circ$$

$$\Leftrightarrow \quad \{ \text{ linearity and composition } \}$$

$$y\ fmul\ x = y(\underline{0} \cdot \underline{0}^\circ)x + \langle \sum\ z\ ::\ y(fadd \cdot fmul)z\ \times\ (z\ succ^\circ\ x)\rangle$$

Term $y(\underline{0} \cdot \underline{0}^\circ)x = 1$ iff both $y = x = 0$, otherwise it equals 0, in which case

$$y\ fmul\ x = \langle \sum\ z, k\ :\ z + 1 = x :\ y(fadd)k\ \times\ k(fmul)z\rangle$$

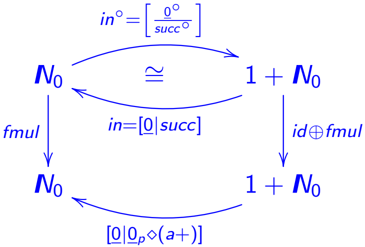
where

$$\begin{aligned} y(fadd)k &= y(\underline{0}\ {}_p\!\diamond\, (a+))k = p(y\underline{0}k) + (1-p)(y(a+)k) \\ &= p(y = 0) + (1-p)(y = a + k) \end{aligned}$$

Hmmmm...

Probabilistic recursion

Far better: inspired by the AoP (Bird and de Moor, 1997), we regard *fmul* as a **catamorphism** in its category of matrices, cf.



Following the usual notation for the **unique** solution of diagrams of this kind, we write $fmul = (\![\underline{0}|\underline{0}\ {}_p\diamond\ (a+)]\!)$.

Catamorphisms have several useful properties which are rather advantageous in calculations.

Probabilistic cata-fusion

For instance, the **cata**-fusion law:

$$(\!|h|\!) = f \cdot (\!|g|\!) \quad \Leftarrow \quad f \cdot g = h \cdot (id \oplus f) \tag{29}$$

Application: suppose f and $(\!|g|\!)$ are probabilistic functions denoting faulty programs.

Then their **fusion** $(\!|h|\!)$ will record how their faults **combine** with each other and **propagate** to outer evaluation levels.

Example in the following slides : (static) **prediction** (pen & paper calculation) of how the faults of *fsucc* and *fmul* "fuse" with each other.

Probabilistic cata-fusion

Altogether, this is the exercise of calculating catamorphism *fprog* such that

$$fprog \;=\; fsucc \cdot fmul \tag{30}$$

in the LAoP (Oliveira, 2012), given faulty

$$fsucc \;=\; id \;{}_{q}\!\diamond\, succ$$

and faulty

$$fmul \;=\; (\!|[\underline{0}|\underline{0} \;{}_{p}\!\diamond\, (a+)]|\!)$$

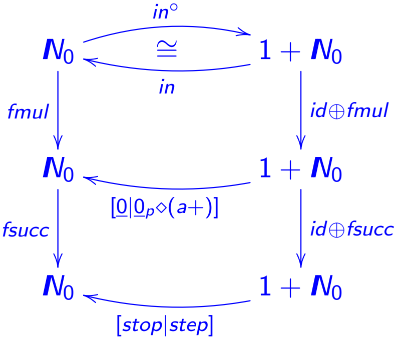
The exercise clearly fits with cata-fusion (29).

Probabilistic cata-fusion

In fact, by (29) the outcome will be

$$fprog = (\!|[stop|step]|\!)$$

provided the lower rectangle aside commutes; thus we just have to solve the equation below for *stop* and *step*:



$$fsucc \cdot [\underline{0}|\underline{0}\ {}_p\!\diamond (a+)] = [stop|step] \cdot (id \oplus fsucc)$$

that is, $fsucc \cdot \underline{0} = stop$ and $fsucc \cdot (\underline{0}\ {}_p\!\diamond (a+)) = step \cdot (id \oplus fsucc)$.

Probabilistic cata-fusion

The first equality yields *stop* almost immediately:

$$
\begin{aligned}
& fsucc \cdot \underline{0} = stop \cdot id \\
\Leftrightarrow \quad & \qquad \{ \text{ definition of } fsucc \ \} \\
& stop = (id \ {}_{q}\!\diamond\, succ) \cdot \underline{0} \\
\Leftrightarrow \quad & \qquad \{ \text{ choice-fusion (24) ; } succ\ 0 = 1 \ \} \\
& stop \ = \ \underline{0} \ {}_{q}\!\diamond\, \underline{1}
\end{aligned}
$$

The calculation of *step* follows from the other equality in the diagram:

$$fsucc \cdot (\underline{0} \ {}_{p}\!\diamond\, (a+)) \ = \ step \cdot fsucc$$

(next slide)

Probabilistic cata-fusion

$$
\begin{array}{ll}
 & fsucc \cdot (\underline{0}\ {}_p\!\diamond\ (a+)) \;=\; step \cdot fsucc \\
\Leftrightarrow & \quad \{ \ \text{choice-fusion (25)} \ ; \ fsucc \cdot \underline{0} = stop \ \} \\
 & stop\ {}_p\!\diamond\ (fsucc \cdot (a+)) \;=\; step \cdot fsucc \\
\Leftrightarrow & \quad \{ \ fsucc \text{ commutes with } (a+) \text{ since } succ \text{ commutes with } (a+) \ \} \\
 & stop\ {}_p\!\diamond\ ((a+) \cdot fsucc) \;=\; step \cdot fsucc \\
\Leftrightarrow & \quad \{ \ stop \text{ is (probabil.) constant, thus } stop \cdot f = stop, \forall f \ ; \ (24) \ \} \\
 & (stop\ {}_p\!\diamond\ (a+)) \cdot fsucc \;=\; step \cdot fsucc \\
\Leftarrow & \quad \{ \ \text{Leibniz} \ \} \\
 & step = stop\ {}_p\!\diamond\ (a+)
\end{array}
$$

In summary:

$$
fprog \;=\; fsucc \cdot fmul \;=\; (\!|[stop|stop\ {}_p\!\diamond\ (a+)]|\!) \quad , \quad \text{for } stop = \underline{0}\ {}_q\!\diamond\ \underline{1}
$$

expresses the combined impact of the faults of the two functions.

Back to programming

Once we map our calculated solution into its monadic equivalent,

```
fprog' p q a 0 = stop q 0
fprog' p q a b = do { x <- fprog' p q a (b-1);
                      step p q a x
                    }
```

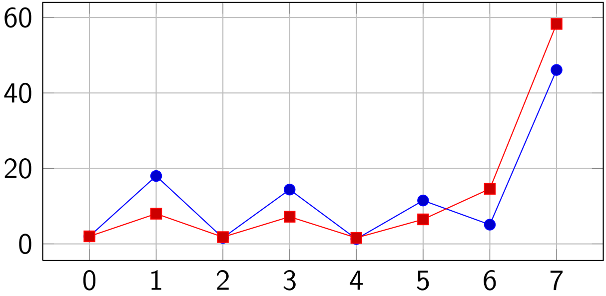
where

```
stop q = schoice q (const 0) (const 1)

step p q a = choice p (stop q) (return.(a+))
```

and experiment with it, we confirm that the two programs — before and after fusion — are **probabilistically indistinguishable**.

Recall experiments

Both programs (before and after "fault-fusion") have the same behaviour, eg. for $a = 2$ and input 3 ($1 + 2 \times 3 = 7$),



for p = 20%, q = 10% (in blue) and for p = 10%, q = 20% (in red).

Last but not least: mutual recursion

The programs we have handled thus far are relatively uninteresting: **for**-loops with one variable only.

We would like to reason about faults in programs such as eg. the following C program

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

computing the **square** of a natural number (two variables s and o).

Program genetics

First of all, we investigate the genetics of this program: how can we be **sure** this program computes $sq\ n = n^2$?

Easy: using standard AoP we get, from $sq\ n = n^2$, two mutually recursive functions,

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

since $(n+1)^2 = n^2 + 2n + 1$, and $odd\ n = 2n + 1$ is the n-the odd number, etc.

Program genetics

Now, tally (pair up) the two functions

$$(sq, odd)x = (sq\ x, odd\ x)$$

and derive

$$\begin{array}{rcl} (sq, odd)0 & = & (sq\ 0, odd\ 0) = (0, 1) \\ (sq, odd)(a+1) & = & (sq(a+1), odd(a+1)) \\ & = & (sq\ a + odd\ a, 2 + odd\ a) \end{array}$$

whose second clause can be re-written into

$$(sq, odd)(a+1) = (q+i, 2+i)\ \textbf{where}\ (q, i) = (sq, odd)a$$

Program genetics

Thus, the pair (sq, odd) is the **for**-loop

$$(sq, odd) = \textbf{for } loop\ (0,1) \quad \textbf{where } loop(q,i) = (q+i, 2+i)$$

which we may incorporate into

$$\begin{array}{l} sq\ n = s \\ \quad where\ (s,o) = \textbf{for } loop\ (0,1)\ \ n \\ \qquad where\ loop(s,o) = (s+o, o+2) \end{array}$$

matching with the C encoding we've started from (aside).

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

(Look how "wise" the syntax of C is compared to what we've just calculated...)

Pairing faulty programs

The lesson learnt from the previous calculation is that, to handle multi-variable faulty **for**-loops we need to investigate about **pairing** in the **CS**-matrix category.

The general result is known as the **mutual recursion theorem** in the AoP: multi-variable programs arise by calculation from systems of mutually recursive functions by pairing.

For this to work for probabilistic functions, pairing has to be a **product** in the CS category.

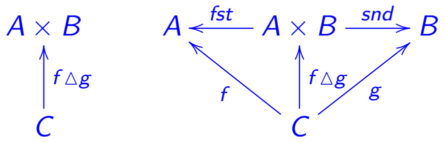
The following slides investigate **probabilistic** pairing, eventually enabling calculation about faults injected in programs such as sq above.

Pairing

Pairing the outputs of probabilistic functions $C \xrightarrow{f} \mathcal{D}A$ and $C \xrightarrow{g} \mathcal{D}B$ is captured by the **Khatri-Rao** product of the corresponding matrices (parentheses again omitted):

$$k = f \vartriangle g \quad \Rightarrow \quad \left\{ \begin{array}{r} fst \cdot k = f \\ snd \cdot k = g \end{array} \right. \tag{31}$$

cf. diagram



(Warning: mind $\Rightarrow$, thus a **weak** categorial product in CS — cf. “forks” in **Rel**.)

Pairing

Khatri-Rao easily captured in terms of the well-known **Kronecker** product $M \otimes N$ of two arbitrary matrices:

$$(y, x)(M \otimes N)(b, a) \quad = \quad (yMb) \times (xNa) \tag{32}$$

Khatri-Rao coincides with Kronecker for column vectors u and v,

$$u \vartriangle v \quad = \quad u \otimes v \tag{33}$$

and expands column-wise as shown by the *exchange law*

$$[M_1|M_2] \vartriangle [N_1|N_2] \quad = \quad [M_1 \vartriangle N_1|M_2 \vartriangle N_2] \tag{34}$$

Projections:

$$\begin{aligned} fst &= id \otimes\, ! \\ snd &= \,! \otimes id \end{aligned}$$

Pairing

Example:

$$2 \xleftarrow{fst} 2 \times 3 \xrightarrow{snd} 3, \qquad 2 \xleftarrow{f} 4 \xrightarrow{g} 3, \qquad 4 \xrightarrow{f \triangle g} 2 \times 3$$

$$fst = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

$$snd = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$f \triangle g = \begin{bmatrix} 0.15 & 0.12 & 0 & 0 \\ 0.35 & 0.06 & 0 & 0.75 \\ 0 & 0.12 & 0 & 0 \\ 0.15 & 0.28 & 0.1 & 0 \\ 0.35 & 0.14 & 0.2 & 0.25 \\ 0 & 0.28 & 0.7 & 0 \end{bmatrix}$$

$$f = \begin{bmatrix} 0.5 & 0.3 & 0 & 0.75 \\ 0.5 & 0.7 & 1 & 0.25 \end{bmatrix}$$

$$g = \begin{bmatrix} 0.3 & 0.4 & 0.1 & 0 \\ 0.7 & 0.2 & 0.2 & 1 \\ 0 & 0.4 & 0.7 & 0 \end{bmatrix}$$

Pairing

The monadic equivalent to Khatri-Rao (probabilistic pairing) is quite intuitive:

```
(f `kr` g) a = do { b <- f a ;
                    c <- g a ;
                    return (b,c)
                  }
mfst d = do { (b,c) <- d ;
              return b
            }
msnd d = do { (b,c) <- d ;
             return c
            }
```

Matrix-wise, much more about Khatri-Rao product etc in the PhD thesis by Hugo Macedo (2012).

Probabilistic mutual recursion

The AoP mutual recursion law, also known as Fokkinga law,

$$\left\{ \begin{array}{l} f \cdot in = h \cdot \mathsf{F}\,(f \vartriangle g) \\ g \cdot in = k \cdot \mathsf{F}\,(f \vartriangle g) \end{array} \right. \quad \Leftrightarrow \quad f \vartriangle g = (\!| h \vartriangle k |\!) \qquad (35)$$

(for polynomial F) extends to the LAoP under some conditions, related to **pairing** (Khatri-Rao) being a weak **product** in category CS.

The square of a natural number

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

is not a **for**-loop (cata over $\mathbb{N}_0$) for $\mathsf{F}\,X = id \oplus X$, but it becomes so thanks to (35) — as we did before in a pointwise manner.

Probabilistic mutual recursion

The matrix transpose of the pair (sq, odd)

$$(sq, odd) = \textbf{for } loop\ (0,1) \quad \textbf{where } loop(q,i) = (q+i, 2+i)$$

we've calculated before is, using the Khatri-Rao combinator,

$$(sq \vartriangle odd) \cdot in = \left[\underline{(1,0)} | (+) \vartriangle (2+) \cdot snd\right] \cdot (id \oplus (sq \vartriangle odd))$$

thanks to the (probabilistic) mutual-recursion law (35).

This calculation leads to the following probabilistically indistinguishable versions of sq (next slide).

Probabilistic mutual recursion

Recursive version:

```
fsq 0 = return 0
fsq(n+1) = do { x <- fsq n ; x ‘fadd‘ (2*n+1) }
```

Linear version:

```
fsql n = do (s,i) <- floop n ; return s
         where floop 0 = return (0,1)
               floop (n+1) = do (s,i) <- floop n ;
                                s’ <- s ‘fadd‘ i ;
                                return (s’,2+i)
```

Both over the same faulty addition, eg.:

```
x +. y = D [(y,0.1),(x+y,0.9)]
x .+ y = D [(x,0.1),(x+y,0.9)]
x .+. y = mynormal (x+y)
```

Probabilistic mutual recursion

Another example of application of mutual recursion is the calculation of **Fibonacci** numbers, as the doubly recursive mathematical definition,

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

converts — by introducing $f\ n = fib(n+1)$ — into a mutual-recursive pair (*"mutumorphism"*)

$$\begin{array}{rcl} f \cdot [\underline{0}|suc] & = & [\underline{1}|add \cdot (f \vartriangle fib)] \\ fib \cdot [\underline{0}|suc] & = & [\underline{1}|f] \end{array}$$

Probabilistic mutual recursion

The same reasoning we did before concerning the *sq* function will yield the following linear version from the given system of mutually recursive functions:

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

Does this transformation extend to the probabilistic (faulty) setting?

Probabilistic mutual recursion

In this case, experiments in Haskell show that the doubly recursive

```
ffib 0 = return 1
ffib 1 = return 1
ffib n  = do a <- ffib(n-1) ;
             b <- ffib(n-2);
             (a `fadd` b)
```

and its linear version

```
ffibl n = do (a,b) <- auxm n ; return b
          where auxm 0 = return (1,1)
                auxm n = do (a,b) <- auxm(n-1);
                            s <- a `fadd` b;
                            return (s,a)
```

perform differently — probabilistic behavior of linear version performs better. Why?

Probabilistic mutual recursion

We've developed a Matlab library for checking (finite approximations to) faulty recursive functions encoded as matrices, cf. eg (Fibonacci):

```
function R = execFibl10(fAdd,n,m,N)
    R = snd(n,n)*aux(fAdd,n,m,N);
end
```

where

```
function R = aux (fAdd,n,m,N)
    if (N==0)
        R = fibl10(fAdd,zeros(n*n,m));
    else
        R = fibl10(fAdd,aux(fAdd,n,m,N-1));
    end
end
```

computes the N first iterations of the fixpoint (Kleene theorem) of linear Fibonacci — see the next slide.

Probabilistic mutual recursion

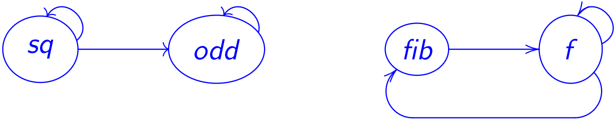
```
function R = fibl10(fAdd,Rec)
    [rRec cRec] = size(Rec);
    m = sqrt(rRec);

    %Defining out
    coref1 = [1 zeros(1,cRec-1);zeros(cRec-1,cRec)]; %Equal to zero coref
    coref2 = [zeros(1,cRec);zeros(cRec-1,1) eye(cRec-1)]; %Not equal to zero coref
    pred = zeros(cRec,cRec);
    for k=0:(cRec-1)
        if (k>0)
            pred(k,k+1) = 1;
        end
    end
    out = juncMat(inj1Mat(1,1+cRec)*bang(cRec),inj2Mat(cRec,1+cRec)*pred)*splitMat(coref1,coref2);

    %Defining recursive call
    FRec = sumMat(idMat(1),Rec);

    %Defining algebra
    one = zeros(m,1);
    one(1+1,1) = 1;
    zero = zeros(m,1);
    zero(1+0,1) = 1;
    a = juncMat(kr(one,zero),kr(fAdd(rRec,m),fst(div(rRec,m),m)));

    R = a*FRec*out;
end
```

Probabilistic mutual recursion

Thanks to this library we have found sufficient conditions for the mutual recursion law (35) to hold probabilistically.

For instance, if the first projection of a probabilistic function is a sharp function, then Khatri-Rao is a (**strong**) product — $\Rightarrow$ in (31) becomes $\Leftrightarrow$ — and probabilistic mutual recursion holds.

This explains the difference in faulty behaviour between the linear versions of *sq* and *fib* — *odd* is a sharp function (no faults), compare the dependency graphs:



Closing

The research question which motivated this talk splits in two other questions, in fact two sides of the same coin:

(a) Can **the** AoP be extended quantitatively in some useful way?
(b) What happens to the discipline once we generalize from relations to matrices?

The answer leads us into **linear algebra**, which eventually provides a surprisingly simple framework for calculating with **set-theory**, **probabilities**, functions and relations, provided it is **typed** — as advocated by Macedo (2012).

Closing

The comment by Sir Arthur Eddington in his *Relativity Theory of Electrons and Protons*

> *"I cannot believe that anything so* ***ugly*** *as multiplication of matrices is an essential part of the scheme of nature"*

can be understood as a call for better laid out **linear algebra** — perhaps **typed** :-)? And — is this kind of **foundation** that sought in 1967, in the Garmisch NATO workshop:

> *In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being* ***provocative****, in implying the need for software manufacture to be based on the types of* ***theoretical foundations*** *and practical disciplines, that are traditional in the established branches of engineering. (Naur and Randell, 1969)*

? Only **time** and **experience** will tell.

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

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

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

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

P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*, 1969. Scientific Affairs Division, NATO. URL `http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/`.

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