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
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 $\mathbb{N}_0$):

$$(a\ast) = \text{for } (a+)\ 0$$

that is,

$$a \ast 0 = 0$$

$$a \ast (n + 1) = a + a \ast n$$

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

$$a \ast 0 = .95\ 0$$

$$a \ast 0 = .05\ a$$

$$a \ast (n + 1) =_1 a + a \ast n$$

in “extended” functional notation.
Example: fault propagation

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

\[ f \ n = \ fsucc(fmul \ a \ n) \]

where \( fsucc \) is a faulty **successor** function,

\[
\begin{align*}
fsucc \ n &= q \ n + 1 \\
fsucc \ n &= 1 - q \ n
\end{align*}
\]

and \( fmul \) is the even more seriously faulty multiplication,

\[
\begin{align*}
fmul \ a \ 0 &= 0 \\
fmul \ a \ (n + 1) &= p \ fmul \ a \ n \\
fmul \ a \ (n + 1) &= 1 - p \ a + fmul \ a \ n
\end{align*}
\]

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:

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

For instance:

Standard monadic function *return* $a$ is the **Dirac distribution** $\mu$ 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 $p$

\[
fadd p a x = \text{choose} p 0 (a+x)
\]

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

\[
fmul p a 0 = \text{return} 0
\]
\[
fmul p a n = \text{do} \{ x \leftarrow fmul p a (n-1) ;
\quad fadd p a x
\}
\]

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

\[
fsucc q = \text{schoice} q \text{id succ}
\]

Functions **choose** and **schoice** are suitable library functions.
Running (Haskell) composition $\textit{fsucc }q \bullet \textit{fmul }p$, 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

$$f = \Lambda R \iff \langle \forall b, a :: b R a \iff b \in f a \rangle$$  \hspace{1cm} (2)

that is,

$$\begin{array}{c}
A \rightarrow \mathcal{P}B \\
\cong \\
\Lambda
\end{array} \cong A \rightarrow B$$  \hspace{1cm} (3)

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

An adjunction, offering two ways for reasoning — one relational ($\text{Rel}$)

\[
\begin{array}{ccc}
\mathcal{P}A & \xrightarrow{\mathcal{E}} & A \\
\downarrow f & & \downarrow \mathcal{E} \\
B & & B
\end{array}
\]

the other monadic ($\text{Set}$):

\[
\begin{array}{ccc}
A & \xleftarrow{\text{return}} & \mathcal{P}A \\
\downarrow R & & \downarrow \mathcal{E}_R \\
B & & \mathcal{P}B
\end{array}
\]

\[f = \mathcal{E}_R \cdot \text{return}\]

where \((\mathcal{E}_R)s = \{ b | a \leftarrow s; bRa \}\)

The same duality in “going probabilistic” (next slide).
Probabilistic functions

Outputs become distributions,

\[ A \rightarrow \mathcal{DB} \cong A \rightarrow B \]  

(4)

where \( \mathcal{DB} \) is the \( B \)-distribution data type

\[ \mathcal{DB} = \{ \mu \in [0,1]^B \mid \sum_{b \in B} \mu b = 1 \} \]  

(5)

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

However, what does \( A \rightarrow B \) on the right hand side of (4) mean?
Probabilistic functions

One has:

\[ A \rightarrow [0, 1]^B \]
⇔ \{ uncurrying \}
\[ A \times B \rightarrow [0, 1] \]
⇔ \{ swapping \}
\[ B \times A \rightarrow [0, 1] \]

where \( B \times A \rightarrow [0, 1] \) 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:

\[
A \rightarrow DB \cong A \rightarrow_{CS} B
\]  

(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 \( f :: A \rightarrow DB \) and \( CS \)-matrix \( M \):

\[
M = [f] \iff \langle \forall b, a :: b M a = (f a)b \rangle
\]  

(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.
Compositionality — matrices compose with each other:

\[ \begin{array}{ccc}
B & \xleftarrow{M} & A \\
\downarrow & & \downarrow \\
C & \xleftarrow{N} & \\
\end{array} \]

\[ M \cdot N \]

where

\[ b(M \cdot N)c = \left\langle \sum a :: (bMa) \times (aNc) \right\rangle \quad (8) \]

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

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

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

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 $A \leftarrow A$.*

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

Given matrix $n \leftarrow^M m$, notation $m \leftarrow^{M^\circ} n$ 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")
- \( X = \begin{bmatrix} P \\ Q \end{bmatrix} \) — \( P \) on top of \( Q \) ("split").

Mind the (polymorphic) types:

(A so-called biproduct)
Rich set of laws, for instance

- **Divide-and-conquer:**
  \[
  [A|B] \cdot \begin{bmatrix} C \\ D \end{bmatrix} = A \cdot C + B \cdot D \tag{11}
  \]

- **“Fusion”-laws:**
  \[
  C \cdot [A|B] = [C \cdot A|C \cdot B] \tag{12}
  \]
  \[
  \begin{bmatrix} A \\ B \end{bmatrix} \cdot C = \begin{bmatrix} A \cdot C \\ B \cdot C \end{bmatrix} \tag{13}
  \]
Special matrices

The following (Boolean) matrices are relevant:

- The **bottom** matrix \( B \leftarrow A \) — wholly filled with 0s
- The **top** matrix \( B \leftarrow A \uparrow \) — wholly filled with 1s
- The **identity** matrix \( B \leftarrow B \id \) — diagonal of 1s
- The **bang** (row) vector \( 1 \leftarrow ! A \) — wholly filled with 1s

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

\[
1 \leftarrow A \uparrow = !
\]

Also note that, on type \( 1 \leftarrow 1 \):

\[
\uparrow = ! = \text{id}
\]
Useful for matrix index manipulation

Two useful rules of thumb,

\[
y(f \cdot N)x = \langle \sum z : y = f z : zN x \rangle
\]  
(14)

\[
y(g^\circ \cdot N \cdot f)x = (gy)N(fx)
\]  
(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

\[
b[f]a \triangleq (b =_S f a)
\]

where \( y =_S x \) is 1 if \( y = x \) and 0 otherwise. Thus matrix \([f]\) represents the graph of \( f \).
Useful for matrix index manipulation

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

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

$$\langle \sum x : p \ x : e \ x \rangle = \langle \sum x :: (p \ x) \times (e \ x) \rangle$$  \hspace{1cm} (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 $A \xrightarrow{f} DB$, its matrix transform $A \xrightarrow{[f]} B$ is such that

$$! \cdot [f] = !$$

(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:

<table>
<thead>
<tr>
<th></th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>True</strong></td>
<td>0.1</td>
<td>0.8</td>
</tr>
<tr>
<td><strong>False</strong></td>
<td>0.9</td>
<td>0.2</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
[f \circ g] &= [f] \cdot [g] \quad (18)
\end{align*}
\]

In monad-speak:

\[
\begin{align*}
[\lambda a. \text{do } \{ b \leftarrow g a; f b \}] &= [f] \cdot [g]
\end{align*}
\]

(It is easy to show that (18) preserves probabilistic functions.)
Probabilistic “junc”

Probabilistic $A + B \overset{[f,g]}{\rightarrow} DC$ — run either $f$ or $g$ — transposes into

$$[[f,g]] = [[f]][[g]]$$  \hspace{1cm} (19)

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

Checking the 100% constraint (17):

$$! \cdot [[f]][[g]]$$

$\iff$ \hspace{0.5cm} \{ fusion-+ (12) \}

$$[! \cdot [f]! \cdot [g]]$$

$\iff$ \hspace{0.5cm} \{ $f$ and $g$ probabilistic (17) ; $[!][] = !$ \}

$$!$$
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

$$\left[f \; p \diamond \; g\right] = p[f] + (1 - p)[g]$$  \hspace{1cm} (20)
Probabilistic choice

Probabilistic choice “is probabilistic”:

\[ ! \cdot \[ f \ p \Diamond g \] \]

\[ = \{ \text{definition (20); bilinearity} \} \]

\[ ! \cdot (p[f]) + ! \cdot ((1 - p)[g]) \]

\[ = \{ p \text{ is a scalar} \} \]

\[ p(! \cdot [f]) + (1 - p)(! \cdot [g]) \]

\[ = \{ f \text{ and } g \text{ are probabilistic} \} \]

\[ p! + (1 - p)! \]

\[ = \{ \text{bilinearity} \} \]

\[ (p + 1 - p)! \]

\[ = \{ \text{cancellation} \} \]

!
Properties

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

\[ f \circledast p f = f \]  \hspace{1cm} (21)  
\[ f \circledast 0 g = g \]  \hspace{1cm} (22)  
\[ f \circledast p g = g \circledast (1-p) f \]  \hspace{1cm} (23)  

fusion-laws

\[ (f \circledast p g) \bullet h = (f \bullet h) \circledast p (g \bullet h) \]  \hspace{1cm} (24)  
\[ h \bullet (f \circledast p g) = (h \bullet f) \circledast p (h \bullet g) \]  \hspace{1cm} (25)  

and the exchange law:

\[ [f, g] \circledast p [h, k] = [f \circledast p h, g \circledast p k] \]  \hspace{1cm} (26)
The **direct sum** of two matrices,

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

which has type

\[
\begin{array}{c c c c}
A & B & A + B \\
M \downarrow & N \downarrow & A + B \downarrow M \oplus N \\
C & D & C + D
\end{array}
\]

(a **bifunctor**) enables us to sum probabilistic functions:

\[
[f \oplus g] = [f] \oplus [g]
\]

**Distribution** over choice

\[
h \oplus (f \ _p \diamond g) = (h \oplus f) \ _p \diamond (h \oplus g)
\]  

is central to probabilistic function calculation.
Probabilistic recursion

Recall

\[
\text{fmul } p \ a \ 0 = \text{return } 0 \\
\text{fmul } p \ a \ (n+1) = \text{do } \{ x <- \text{fmul } p \ a \ n ; \text{fadd } p \ a \ x \}
\]

Converting this to its **matrix-transpose** we get \( \text{fmul} \) as the unique solution to LAoP equation

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

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

\[
\text{fmul} = 0 \cdot 0^\circ + \text{fadd} \cdot \text{fmul} \cdot \text{succ}^\circ
\]

How do we reason about this equation?
Probabilistic recursion

We might introduce indices, cf.:

\[ \text{fmul} = 0 \cdot 0^\circ + \text{fadd} \cdot \text{fmul} \cdot \text{succ}^\circ \]

\[ \Leftrightarrow \quad \{ \text{linearity and composition} \} \]

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

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

\[ y \text{ fmul } x = \langle \sum z, k : \ z + 1 = x : \ y(\text{fadd})k \times k(\text{fmul})z \rangle \]

where

\[ y(\text{fadd})k = y(0 \ p \diamond (a+))k = p(y0k) + (1 - p)(y(a+)k) \]
\[ = p(y = 0) + (1 - p)(y = a + k) \]

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.

\[
\begin{align*}
\text{in}^o &= \left[ \frac{0^o}{\text{succ}^o} \right] \\
\mathcal{N}_0 \xrightarrow{\cong} 1 + \mathcal{N}_0 \xleftarrow{\cong} 1 + \mathcal{N}_0 \\
\text{in} &= [0 \mid \text{succ}] \\
\text{id} \oplus \text{fmul}
\end{align*}
\]

Following the usual notation for the **unique** solution of diagrams of this kind, we write \( fmul = ([0 \mid 0 \odot (a+)]]) \).

Catamorphisms have several useful properties which are rather advantageous in calculations.
For instance, the \textbf{cata}-fusion law:

\[
|h| = f \cdot |g| \iff f \cdot g = h \cdot (id \oplus f)
\]  

(29)

\textbf{Application}: suppose \(f\) and \(|g|\) are probabilistic functions denoting faulty programs.

Then their \textbf{fusion} \(|h|\) will record how their faults \textbf{combine} with each other and \textbf{propagate} to outer evaluation levels.

\textbf{Example} in the following slides: (static) \textbf{prediction} (pen & paper calculation) of how the faults of \textit{fsucc} and \textit{fmul} “fuse” with each other.
Altogether, this is the exercise of calculating catamorphism $f_{prog}$ such that

$$f_{prog} = f_{succ} \cdot f_{mul}$$  \hspace{1cm} (30)$$

in the LAoP (Oliveira, 2012), given faulty

$$f_{succ} = \text{id}_{q} \triangleright succ$$

and faulty

$$f_{mul} = (\begin{bmatrix} |0|0 \triangleright (a+) \end{bmatrix})$$

The exercise clearly fits with cata-fusion (29).
In fact, by (29) the outcome will be

\[ fprog \rightarrow ([stop|step]) \]

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

\[ \text{fsucc} \cdot [0|0 \ p \diamond (a+)] = [stop|step] \cdot (id \oplus \text{fsucc}) \]

that is, \( \text{fsucc} \cdot 0 = \text{stop} \) and \( \text{fsucc} \cdot (0 \ p \diamond (a+)) = \text{step} \cdot (id \oplus \text{fsucc}) \).
The first equality yields \( \text{stop} \) almost immediately:

\[
fsucc \cdot 0 = \text{stop} \cdot \text{id}
\]

\[
\Leftrightarrow \quad \{ \text{definition of } fsucc \} \quad \text{stop} = (\text{id}_q \diamond \text{succ}) \cdot 0
\]

\[
\Leftrightarrow \quad \{ \text{choice-fusion (24) ; succ } 0 = 1 \} \quad \text{stop} = 0_q \diamond 1
\]

The calculation of \( \text{step} \) follows from the other equality in the diagram:

\[
fsucc \cdot (0_p \diamond (a^+)) = \text{step} \cdot fsucc
\]

(next slide)
Probabilistic cata-fusion

\[ fsucc \cdot (0 \diamond (a+)) = \text{step} \cdot fsucc \]
\[ \iff \{ \text{choice-fusion (25)} ; fsucc \cdot 0 = \text{stop} \} \]
\[ \text{stop} \diamond (fsucc \cdot (a+)) = \text{step} \cdot fsucc \]
\[ \iff \{ \text{fsucc} \text{ commutes with} (a+) \text{ since} succ \text{ commutes with} (a+) \} \]
\[ \text{stop} \diamond ((a+) \cdot fsucc) = \text{step} \cdot fsucc \]
\[ \iff \{ \text{stop} \text{ is (probabil.) constant, thus} \text{stop} \cdot f = \text{stop} , \forall f ; (24) \} \]
\[ (\text{stop} \diamond (a+)) \cdot fsucc = \text{step} \cdot fsucc \]
\[ \iff \{ \text{Leibniz} \} \]
\[ \text{step} = \text{stop} \diamond (a+) \]

In summary:

\[ fprog = fsucc \cdot fmul = ([stop \mid stop \diamond (a+)]) \] , for \( stop = 0 \diamond 1 \)

expresses the combined impact of the faults of the two functions.
Back to programming

Once we map our calculated solution into its monadic equivalent,

\[
\text{fprog'} \; p \; q \; a \; 0 = \text{stop} \; q \; 0 \\
\text{fprog'} \; p \; q \; a \; b = \text{do} \; \{ \; x \leftarrow \text{fprog'} \; p \; q \; a \; (b-1); \quad \text{step} \; p \; q \; a \; x \; \}
\]

where

\[
\text{stop} \; q = \text{schoice} \; q \; (\text{const} \; 0) \; (\text{const} \; 1)
\]

\[
\text{step} \; p \; q \; a = \text{choice} \; p \; (\text{stop} \; q) \; (\text{return} \; (a+))
\]

and experiment with it, we confirm that the two programs — before and after fusion — are \textbf{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

```c
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).
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,

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

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

\[(sq, odd)0 = (sq 0, odd 0) = (0, 1)\]
\[(sq, odd)(a + 1) = (sq(a + 1), odd(a + 1)) = (sq a + odd a, 2 + odd a)\]

whose second clause can be re-written into

\[(sq, odd)(a + 1) = (q + i, 2 + i) \text{ where } (q, i) = (sq, odd)a\]
Thus, the pair \((sq, odd)\) is the \textbf{for}-loop

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

which we may incorporate into

\[
sq \ n = s
\]

\[
\text{where } (s, o) = \textbf{for loop} (0, 1) \ n
\]

\[
\text{where } loop(s, o) = (s + o, o + 2)
\]

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

```c
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 $C \overset{f}{\to} DA$ and $C \overset{g}{\to} DB$ is captured by the **Khatri-Rao** product of the corresponding matrices (parentheses again omitted):

$$k = f \triangle g \quad \Rightarrow \quad \begin{Bmatrix} 
\text{fst} \cdot k = f \\
\text{snd} \cdot k = g 
\end{Bmatrix}$$  \hspace{1cm} (31)

cf. diagram

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

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

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

(32)

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

$$u \triangle v = u \otimes v$$

(33)

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

$$[M_1|M_2] \triangle [N_1|N_2] = [M_1 \triangle N_1|M_2 \triangle N_2]$$

(34)

Projections:

$$\text{fst} = \text{id} \otimes !$$

$$\text{snd} = ! \otimes \text{id}$$
Pairing

Example:

\[ \text{fst} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \]

\[ \text{snd} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \]

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

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

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

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

\[
(f \text{‘kr’} g) \ a = \text{do} \ \{ \ b <- f \ a ; \\
\quad c <- g \ a ; \\
\quad \text{return} \ (b,c) \}
\]

mfst \ d = \text{do} \ \{ \ (b,c) <- d ; \\
\quad \text{return} \ b \}

msnd \ d = \text{do} \ \{ \ (b,c) <- d ; \\
\quad \text{return} \ c \}

Probabilistic mutual recursion

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

\[
\begin{align*}
  f \cdot \text{in} &= h \cdot F (f \triangle g) \\
  g \cdot \text{in} &= k \cdot F (f \triangle g)
\end{align*}
\implies f \triangle g = (h \triangle k)
\]  

(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

\[
sq 0 = 0 \\
\text{sq} (n + 1) = \text{sq} n + 2n + 1
\]

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

The matrix transpose of the pair \((sq, odd)\)

\[(sq, odd) = \text{for loop } (0, 1) \text{ where } \text{loop}(q, i) = (q + i, 2 + i)\]

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

\[(sq \triangle odd) \cdot \text{in} = \left[(1, 0)|(+ \triangle (2+) \cdot \text{snd}\right] \cdot (id \oplus (sq \triangle odd))\]
	hanks 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:

\[
\begin{align*}
\text{fsq 0} &= \text{return 0} \\
\text{fsq}(n+1) &= \{ x \leftarrow \text{fsq n} ; x \text{ 'fadd' } (2*n+1) \}
\end{align*}
\]

Linear version:

\[
\begin{align*}
\text{fsql n} &= \{ s, i \leftarrow \text{floop n} ; \text{return } s \}
\text{where } \\
\text{floop 0} &= \text{return } (0,1) \\
\text{floop (n+1)} &= \{ s, i \leftarrow \text{floop n} ; \\
&s' \leftarrow s \text{ 'fadd' } i ; \\
&\text{return } (s',2+i) \}
\end{align*}
\]

Both over the same faulty addition, eg.:

\[
\begin{align*}
x +. y &= \text{D } [(y,0.1),(x+y,0.9)] \\
x .+ y &= \text{D } [(x,0.1),(x+y,0.9)] \\
x .+. y &= \text{mynormal (x+y)}
\end{align*}
\]
Probabilistic mutual recursion

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

\[
\begin{align*}
  \text{fib} \ 0 & \quad = \quad 1 \\
  \text{fib} \ 1 & \quad = \quad 1 \\
  \text{fib}(n + 2) & \quad = \quad \text{fib}(n + 1) + \text{fib} \ n
  \end{align*}
\]

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

\[
\begin{align*}
  f \cdot [0|\text{suc}] & \quad = \quad [1|\text{add} \cdot (f \triangle \text{fib})] \\
  \text{fib} \cdot [0|\text{suc}] & \quad = \quad [1|f]
  \end{align*}
\]
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:

```c
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

\[
\begin{align*}
\text{ffib 0} &= \text{return 1} \\
\text{ffib 1} &= \text{return 1} \\
\text{ffib n} &= \text{do } a \leftarrow \text{ffib(n-1)} \\
&\quad \text{; b } \leftarrow \text{ffib(n-2)}; \\
&\quad (a \text{ `fadd` b)}
\end{align*}
\]

and its linear version

\[
\begin{align*}
\text{ffibl n} &= \text{do } (a,b) \leftarrow \text{auxm } n; \text{ return } b \\
\text{where auxm 0} &= \text{return (1,1)} \\
\text{auxm n} &= \text{do } (a,b) \leftarrow \text{auxm(n-1)}; \\
&\quad s \leftarrow a \text{ `fadd` b}; \\
&\quad \text{return (s,a)}
\end{align*}
\]

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):

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

where

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

    % Defining output
    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:
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).
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.


