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

March 2013

Abstract

This file is a Haskell executable. It contains the experimental part of the homonym paper (submitted). Please unzip `website.zip` first and then move to the just created directory `website`. For the Haskell part run

```
ghci -XNPlusKPatterns paper_haskell_matlab
```

and follow the examples in part I below. For the MATLAB part, open MATLAB in the `website/matlab` directory, where all the executables can be found and run the scripts of part II.

Both parts follow the structure of the homonym paper.

Part I
Haskell

1 Introduction

2 Motivation

3 Mutual recursion

Fibonacci:

$$fib\ 0 = 0$$
$$fib\ 1 = 1$$
$$fib\ (n+2) = fib\ n + fib\ (n+1)$$

Mutually-recursive equivalent:

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

For-loop combinator:

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

Linear Fibonacci:

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

Linear square:

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

4 Going probabilistic

Start with:

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

Define:

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

Re-define:

```
$mfib\ 0 = return\ 0$
$mfib\ 1 = return\ 1$
$mfib\ (n+2) = \mathbf{do}\ \{x \leftarrow mfib\ n; y \leftarrow mfib\ (n+1); fadd\ 0.1\ x\ y\}$
```

Run *mfib* 4:

```
3  81.0%
2  18.0%
1   1.0%
```

Define

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

where

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

Run *mfib* 5 and *mfibl* 5 and then *mfib* 6 and *mfibl*′ 6, to obtain the table in the paper — table 1 below.

Then define

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

and

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

Run *msq n* and *msql n* for $n = 1\,..$ to check their probabilistic equality, as in the table (table 4 below).

5 Probabilistic for-loops in the LAoP

Define

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

Run *ftwice* 4:

```
*Main> ftwice 4
8  65.6%
6  29.2%
4   4.9%
2   0.4%
0   0.0%
```

Define

```
$mfibl'\ n =$
  $\textbf{do}\ \{(x, y) \leftarrow mfor\ loop\ (0,1)\ n; return\ x\}$
```

Run *mfibl′* 5:

```
*Main> mfibl' 5
5  72.9%
3  16.2%
4   8.1%
2   2.7%
1   0.1%
```

6 Probabilistic mutual recursion in the LAoP

Function f:

$f\ 1 = uniform\ [1\,..\,2]$
$f\ 2 = D\ [(1, 0.3), (2, 0.7)]$
$f\ 3 = return\ 2$
$f\ 4 = D\ [(1, 0.75), (2, 0.25)]$

Function g:

$g\ 1 = D\ [(1, 0.3), (2, 0.7)]$
$g\ 2 = D\ [(1, 0.4), (2, 0.2), (3, 0.4)]$
$g\ 3 = D\ [(1, 0.1), (2, 0.2), (3, 0.7)]$
$g\ 4 = return\ 2$

Function k:

$$k\ 1 = D\ ([((1,1),0.24),((1,2),0.08),((1,3),0.08)] +\!\!+ [((2,1),0.36),((2,2),0.12),((2,3),0.12)])$$

Checking column 1:

$$k1 = k\ 1 \equiv ((\mathit{mfst} \cdot k) \vartriangle (\mathit{msnd} \cdot k))\ 1$$

```
(2,1)  36.0%
(1,1)  24.0%
(2,2)  12.0%
(2,3)  12.0%
(1,2)   8.0%
(1,3)   8.0%
```

7 Asymmetric Khatri-Rao product

8 Probabilistic mutual recursion resumed

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

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

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

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

9 Generalizing to other fault propagation patterns

Define

$$\mathit{fcat} = \mathit{mfold}\ (\mathit{snd}\ {}_{0.1}\!\diamond\ \mathsf{cons})\ (\mathit{return}\ [\,])$$

where

```
$mfold :: Monad\ m \Rightarrow ((a,b) \rightarrow m\ b) \rightarrow m\ b \rightarrow [a] \rightarrow m\ b$
$mfold\ f\ d\ [\,] = d$
$mfold\ f\ d\ (h:t) = \mathbf{do}\ \{x \leftarrow mfold\ f\ d\ t; f\ (h,x)\}$
```

Run *fcat* `"abc"` to obtain:

```
*Main> fcat "abc"
"abc"  72.9%
 "ab"   8.1%
 "ac"   8.1%
 "bc"   8.1%
  "a"   0.9%
  "b"   0.9%
  "c"   0.9%
   ""   0.1%
```

Define

$$fcount = mfold\ ((id\ {}_{0.15}\!\diamond\ \mathsf{succ}) \cdot snd)\ (return\ 0)$$

Run *fcount* `"abc"`:

```
*Main> fcount "abc"
3  61.4%
2  32.5%
1   5.7%
0   0.3%
```

Define

$$pipe = fcount \bullet fcat$$

Run *pipe* `"abc"`:

```
*Main> pipe "abc"
3  44.8%
2  41.3%
1  12.7%
0   1.3%
```

Fusion — define

$$\begin{aligned} &ffcount = mfold\ (mix\ 0.1\ 0.15)\ (return\ 0)\\ &\quad \mathbf{where}\ mix\ p\ q = (choice\ p\ return\ (id\ {}_{q}\!\diamond\ \mathsf{succ})) \cdot snd \end{aligned}$$

and run:

```
*Main> ffcount "abc"
3  44.8%
2  41.3%
1  12.7%
0   1.3%
```

Auxiliary

$$\begin{aligned} &\mathsf{cons}\ (h, t) = h : t\\ &\mathsf{nil}\ \_ = [\,]\\ &add\ (x, y) = x + y\\ &zero = \underline{0}\\ &one = \underline{1} \end{aligned}$$

Part II
MATLAB

1 Introduction

2 Motivation

3 Mutual recursion

To run the recursive version of *fib*, without injecting any faulty behaviour, run for instance:

```
  >> execFibr(@nfAdd,5,10,4)

 ans =

      1     0     0     0     0
      0     1     1     0     0
      0     0     0     1     0
      0     0     0     0     1
      0     0     0     0     0
      0     0     0     0     0
      0     0     0     0     0
      0     0     0     0     0
      0     0     0     0     0
      0     0     0     0     0
```

The first parameter of the command is, basically, the *add* function the command will use to calculate its result. In this case we are passing a *non-faulty* add (nfAdd), since we want a sharp Fibonacci. The next two parameters are the numbers of columns and rows, respectively, of the result matrix. In order to calculate (to see) *fib 4*, for example, one has to force, at least, 5 columns, because the first one corresponds to zero (*fib 0*). The last parameter corresponds to the actual input number we want to pass to the Fibonacci function. If $matrix(3,4) = 1$ (*matrix(line,column) with indexes beginning at 1*) then $fib(4-1) = (3-1)$ or $fib3 = 2$. Thus, observing the matrix, we can see that *fib 0 = 0*, *fib 1 = 1...* as it was supposed.

To run the linear version of *fib*, without injecting any faulty behaviour, run for instance:

```
  >> execFibl(@nfAdd,6,10,5)

 ans =

      1     0     0     0     0     0
      0     1     1     0     0     0
      0     0     0     1     0     0
      0     0     0     0     1     0
      0     0     0     0     0     0
      0     0     0     0     0     1
      0     0     0     0     0     0
      0     0     0     0     0     0
```

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

Table 1: Faulty Fibonacci (recursive and linear)

```
     0     0     0     0     0     0
     0     0     0     0     0     0
```

Please note that, in this case, we increased the number of rows to 6 so that we could see the result of *fib* for 5.

To run the linear version of *sq*, without injecting any faulty behaviour, run for instance:

```
 >> execSql(@nfAdd,4,10,3)

ans =

     1     0     0     0
     0     1     0     0
     0     0     0     0
     0     0     0     0
     0     0     1     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     1
```

Please note that, in this case, we used 10 for the number of columns of the result matrix, simply, because we know that *sq 3 = 9*, so we must have at least 10 rows to visualize 9 as result.

4 Going probabilistic

In this section we intend to inject some faults in the *sq* and *fib* functions. To do this, instead of using a non-faulty add function as the first parameter of the commands in the previous section, we can use faulty ones.

To obtain the results displayed in table 1 presented in section 4 of the article run:

```
 >> mfib(7,12)
```

```
ans =

    1.0000         0         0         0         0         0         0
         0    1.0000    1.0000    0.1000    0.0100    0.0010    0.0001
         0         0         0    0.9000    0.1800    0.0189    0.0019
         0         0         0         0    0.8100    0.1053    0.0109
         0         0         0         0         0    0.2187    0.0266
         0         0         0         0         0    0.6561    0.0984
         0         0         0         0         0         0    0.1181
         0         0         0         0         0         0    0.2657
         0         0         0         0         0         0    0.4783
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
```

and

```
>> mfibl(7,14)

ans =

    1.0000         0         0         0         0         0         0
         0    1.0000    1.0000    0.1000    0.0100    0.0010    0.0001
         0         0         0    0.9000    0.1800    0.0270    0.0036
         0         0         0         0    0.8100    0.1620    0.0243
         0         0         0         0         0    0.0810    0.0243
         0         0         0         0         0    0.7290    0.1458
         0         0         0         0         0         0    0.1458
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.6561
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
```

The last two columns of each result matrix present the results displayed in table 1.

To obtain the results displayed in table 4 also present in section 4 of the article run:

```
>> msq(7,40)

ans =

    1.0000         0         0         0         0         0         0
         0    1.0000         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0    0.1000         0         0         0         0
         0         0    0.9000         0         0         0         0
         0         0         0    0.1000         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0    0.1000         0         0
         0         0         0    0.0900         0         0         0
```

n	*msq n*	*msql n*
0	`0 100.0%`	`0 100.0%`
1	`1 100.0%`	`1 100.0%`
2	`4  90.0%`	`4  90.0%`
	`3  10.0%`	`3  10.0%`
3	`9  81.0%`	`9  81.0%`
	`5  10.0%`	`5  10.0%`
	`8   9.0%`	`8   9.0%`
⋮	⋮	⋮
6	`36  59.0%`	`36  59.0%`
	`11  10.0%`	`11  10.0%`
	`20   9.0%`	`20   9.0%`
	`27   8.1%`	`27   8.1%`
	`32   7.3%`	`32   7.3%`
	`35   6.6%`	`35   6.6%`
⋮	⋮	⋮

Table 2: Faulty square (recursive and linear)

```
     0         0         0    0.8100         0    0.1000         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0    0.1000
     0         0         0         0    0.0900         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0    0.0810         0         0
     0         0         0         0    0.7290    0.0900         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0    0.0900
     0         0         0         0         0    0.0810         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0    0.0729         0
     0         0         0         0         0    0.6561         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0    0.0810
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0    0.0729
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0    0.0656
     0         0         0         0         0         0    0.5905
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
     0         0         0         0         0         0         0
```

and

```
 >> msql(7,40)

ans =

    1.0000         0         0         0         0         0         0
         0    1.0000         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0    0.1000         0         0         0         0
         0         0    0.9000         0         0         0         0
         0         0         0    0.1000         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0    0.1000         0         0
         0         0         0    0.0900         0         0         0
         0         0         0    0.8100         0    0.1000         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.1000
         0         0         0         0    0.0900         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0    0.0810         0         0
         0         0         0         0    0.7290    0.0900         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.0900
         0         0         0         0         0    0.0810         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0    0.0729         0
         0         0         0         0         0    0.6561         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.0810
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.0729
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0    0.0656
         0         0         0         0         0         0    0.5905
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
```

5 Probabilistic for-loops in the LAoP

6 Probabilistic mutual recursion in the LAoP

To obtain the results displayed in first diagram of section 6 in the paper run:

```
 >> f = [0.5 0.3 0 0.75;0.5 0.7 1 0.25];
```

```
>> g = [0.3 0.4 0.1 0;0.7 0.2 0.2 1;0 0.4 0.7 0];
>> kr(f,g)

ans =

    0.1500    0.1200         0         0
    0.3500    0.0600         0    0.7500
         0    0.1200         0         0
    0.1500    0.2800    0.1000         0
    0.3500    0.1400    0.2000    0.2500
         0    0.2800    0.7000         0

>> fst(2,3)

ans =

     1     1     1     0     0     0
     0     0     0     1     1     1

>> snd(2,3)

ans =

     1     0     0     1     0     0
     0     1     0     0     1     0
     0     0     1     0     0     1
```

7 Asymmetric Khatri-Rao product

8 Probabilistic mutual recursion resumed

In this section we injected two faults to the *square* functions. They were called *sq'* and *sql'* in the paper. To obtain the results displayed in table 3 present in section 8 of the paper, run the following commands:

```
>> msq2(4,12)

ans =

    1.0000         0         0         0
         0    1.0000    0.0100    0.0010
         0         0    0.0900    0.0001
         0         0    0.0900    0.0188
         0         0    0.8100    0.0024
         0         0         0    0.1029
         0         0         0    0.0219
         0         0         0    0.1968
         0         0         0    0.0656
         0         0         0    0.5905
         0         0         0         0
         0         0         0         0
```

and

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

Table 3: Double faulty *square* functions

```
>> msql2(4,12)

ans =

    1.0000         0         0         0
         0    1.0000    0.0100    0.0010
         0         0    0.0900    0.0009
         0         0    0.0900    0.0261
         0         0    0.8100    0.0081
         0         0         0    0.1539
         0         0         0    0.0081
         0         0         0    0.0729
         0         0         0    0.0729
         0         0         0    0.6561
         0         0         0         0
         0         0         0         0
```

9 Generalizing to other fault propagation patterns

This section was conceived so that we could extend faulty behaviour to functions with types other than the Naturals. In this case we opted for the Sequences type, and to do that we presented two particular functions over sequences which are the *count* and *cat*. The *cat* function is the identity of sequences and the *count* counts the number of elements of a sequence. These functions were also implemented with Matlab, however their output is bit more difficult to understand because, by the time we start dealing with sequences, the cardinality of the types grows very much.

To obtain the results of $fcat"abc"$ run the following command:

```
>> faultyCat(3,3,3)
```

Prompting the command, you realize that the output given is somewhat extensive - a matrix 40×40. The reason for this is very simple. $faultyCat$ receives:

1. The number of different elements that can constitute the sequence. If you choose 3, like in this case, you can imagine a sequence only with 1, 2 and 3 as possible elements;
2. The maximum length of the sequences. If you choose 3, like in this case, you can imagine sequences with 0, 1, 2 or 3 as length;

3. The number of iterations the function is supposed to run.

When we pass to $faultyCat$ those parameters we've just passed, it generates all the sequences possible respecting the parameters: 3 different elements and 3 as maximum length. There are 3 different lists with one element only, 9 with two elements and 27 with three elements and, lastly, the empty list. Doing the math, we have $1 + 3 + 9 + 27 = 40$ different sequences, and that's were the dimension of the result matrix comes from.

In order to easily understand the result matrix, scrolling up the screen is possible to visualize that a matrix called "columns" was calculated - this matrix indicates the order of the result matrix. So, to see, for instance what is the result given by $faultyCat$ for a the sequence *"abc"*, like in the paper, firstly you need to count the position of the sequence in the *columns* matrix, and then you need to look up, in the result matrix, the column with the number you got first. Thus, for the sequence *"abc"*, looking it up in the *columns* matrix, we realize it stands in the 35^{th} line, which is the sequence [1230]. Then, the 35^{th} column in the result matrix is the following:

```
    0.0010
    0.0090
    0.0090
    0.0090
         0
         0
         0
    0.0810
         0
         0
    0.0810
    0.0810
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
         0
    0.7290
         0
         0
```

```
          0
          0
          0
```

This column is the result of $faultyCat$"abc", just like in the paper. To interpret this column, once again we have to rely on the *columns* matrix. The first row of $faultyCat$"abc" is 0.0010 and the first row the *columns* matrix is 0000, which corresponds to the empty sequence ([]). This means that $faultyCat$"abc" $=$ [] with 0.1 per cent of probability. The same 'line of though" is applied to the remaining lines, which allow us to obtain the same result displayed in the paper:

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

To execute $fcount$"abc" prompt the following command in matlab:

```
 >> faultyCount(3,3,3)
```

This function has the same parameters $faultyCat$ has, however, in this case, we are calculating the length of sequences, which means that the output type is Naturals, not Sequences as it was before. This means that the result matrix does not have to be as large as it was with $faultyCat$. The 35^{th} column of the result displays $faultyCount$"abc":

```
    0.0034
    0.0574
    0.3251
    0.6141
```

To execute *fcount . fcat "abc"*:

```
 >> count = faultyCount(3,3,3);
 >> cat = faultyCat(3,3,3);
 >> result = count*cat;
 >> result(1:4,35)

ans =

    0.0130
    0.1267
    0.4126
    0.4477
```

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.

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