— Software Architecture —

Software Components as Monadic Mealy machines (MMM)

J.N. Oliveira

MFES/Software Architecture Course 2016/17

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY
INESCTEC

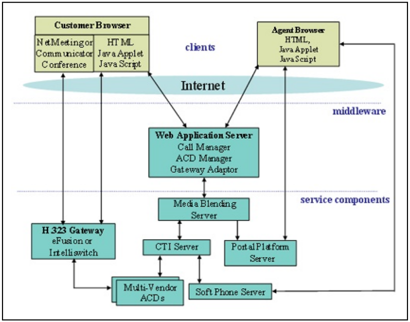
INESC TEC & University of Minho

Motivation

Software architecture

CBS — **component**-oriented software design.

Analogy and inspiration comes from **hardware** and general engineering practice.



Questions: What is a software **component**? How do we **connect** components together? Can we calculate the **behaviour** of CBS **systems** from that of their **components**?

Certification

Overall concern for **safety** and **certification**

Opportunities for Formal Methods in standard RTCA **DO 178C** for certifying airborne software.

Warning:

> *(...) the use of formal methods to be "at least as good as" a conventional approach that does not use formal methods.* (Joyce, 2011)

See also: Software Considerations in Airborne Systems and Equipment Certification by RTCA SC-205, EUROCAE WG-12 [1]

[1] RTCA = Radio Technical Commission for Aeronautics; EUROCAE = European Organisation for Civil Aviation Equipment

Dependable software systems

Quoting Daniel Jackson (2009):

> *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****.*

In formula

$$\textit{dependable system} = \textit{benefit} + \textit{risk}$$

one finds:

- **benefit** $=$ qualitative
- **risk** $=$ quantitative.

What about **evidence**?

Safety cases

MOD Defence Standard 00-56:

> **9.1** *The Contractor shall produce a* **Safety Case** *for the system [which] shall [provide] a* **compelling**, **comprehensible** *and* **valid** *case that a system is safe for a given application in a given environment.*

DS 00-56 (contd.):

> **10.5.4** *All assumptions, data, judgements and calculations underpinning the* **Risk Estimation** *shall be recorded in the* **Safety Case**, *such that the risk estimates can be reviewed and reconstructed.*

Risk estimation? Calculations? How, when and where is this performed in a **FM** life-cycle?

P(robabilistic)R(isk)A(nalysis)

NASA/SP-2011-3421 (Stamatelatos and Dezfuli, 2011):

> *1.2.2 A 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*

Moreover,

> *1.2.3 (...) The* ***total probability*** *from the set of scenarios modeled may also be non-negligible even though the probability of each scenario is small.*

Need for probabilism

As program semantics are usually **qualitative** — e.g. relational — how does one **quantify** risk?

PRA performed **a posteriori** — Hmmm... we've seen this mistake before, eg. in program correctness.

Need for a change:

> *Programming should incorporate* **risk** *as the rule rather than the exception (absence of risk =* **ideal** *case).*

Need for **combinators** expressing risk of failure, eg. **probabilistic choice** between **expected behaviour** and **misbehaviour** (McIver and Morgan, 2005).

Need for probabilism

Think of things that **can go wrong**:

$$\mathit{bad} \cup \mathit{good}$$

How **likely**? Try

$$\mathit{bad}\ {}_{p}\!\diamond\ \mathit{good} \tag{1}$$

where

$$\mathit{bad}\ {}_{p}\!\diamond\ \mathit{good} \ = \ p \times \mathit{bad} + (1 - p) \times \mathit{good}$$

for some **probability** p of *bad behaviour*, eg. the **imperfect** action

$$\mathit{top}\ {}_{(10^{-7})}\!\diamond\ \mathit{pop}$$

leaving a stack unchanged with 10^{-7} probability.

Probabilistic truth tables

Probabilistic **negation** $id \;_{(0.01)}\!\diamond\; neg$:

$$
\begin{aligned}
& id \;_{(0.01)}\!\diamond\; neg \\
= \;& 0.01 \times \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 1 & 0 \\ \textbf{True} & 0 & 1 \end{array} \right) + 0.99 \times \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0 & 1 \\ \textbf{True} & 1 & 0 \end{array} \right) \\
= \;& \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0.01 & 0 \\ \textbf{True} & 0 & 0.01 \end{array} \right) + \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0 & 0.99 \\ \textbf{True} & 0.99 & 0 \end{array} \right) \\
= \;& \begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0.01 & 0.99 \\ \textbf{True} & 0.99 & 0.01 \end{array}
\end{aligned}
$$

In this module

Our approach will be a **coalgebraic** semantics for software components modeled as **monadic Mealy machines** (Oliveira and Miraldo, 2016).

We shall be interested in reasoning about the risk of **faults propagating** in **component**-based software (**CBS**) systems.

This is central to software **certification** and to *safe* software **architectures**.

Traditional CBS **risk analysis** relies on *semantically weak* CBS models, e.g. component **call-graphs** (Cortellessa and Grassi, 2007).

Module overview

Foundations:

- **Functors**, algebras and coalgebras
- **Monads**
- Interaction with **relation algebra**
- **Kleisli composition**
- (Components as) coalgebras
- Distributive laws

Applications:

- Simulation and animation in **Haskell**

Background:

- Items in boldface have been studied in previous courses.

Background

Power transpose

Let $A \xrightarrow{R} B$ be a relation. Define the function

$$\Lambda R : A \to \mathbb{P}\, B$$
$$\Lambda R\ a = \{\, b \mid b\ R\ a \,\}$$

such that:

$$\Lambda R = f \quad \Leftrightarrow \quad \langle \forall\ b, a\ ::\ b\ R\ a \Leftrightarrow b \in f\ a \rangle$$

That is:

$$\begin{tikzcd} A \to \mathbb{P}\, B \arrow[rr, bend left, "(\in\cdot)"] \arrow[rr, phantom, "\cong"] & & A \to B \arrow[ll, bend left, "\Lambda"] \end{tikzcd} \qquad f = \Lambda R \ \Leftrightarrow\ \in \cdot f = R \qquad (2)$$

In words: **relations** can be represented by set-valued **functions**.

"Maybe" transpose

Let $A \xrightarrow{S} B$ be a **simple** relation. Define the function

$$\Gamma S : A \to B + 1$$

such that:

$$\Gamma S = f \quad \Leftrightarrow \quad \langle \forall\ b, a\ ::\ b\ S\ a \Leftrightarrow (i_1\ b) = f\ a \rangle$$

That is:

$$\begin{tikzcd} A \to B + 1 \arrow[rr, bend left, "(\in\cdot)"] \arrow[rr, phantom, "\cong"] & & A \to B \arrow[ll, bend left, "\Gamma"] \end{tikzcd} \qquad f = \Gamma S \ \Leftrightarrow\ S = i_1^{\circ} \cdot f \quad (3)$$

In words: simple **relations** can be represented by "pointer"-valued **functions**.

Distribution transpose

A **distribution** is a function $\mu : X \to [0,1]$ such that (a) the set $\{x \in X \mid \mu\, x > 0\}$ is finite; (b) $\langle \sum x \ :\ x \in X :\ \mu\, x \rangle = 1$.

$\mu\, x$ denotes the **probability** of event x taking place.

Denote by $\mathbb{D}\ B$ the set of all distributions on B.

Recall

$$\begin{tikzcd} A \times B \to C \arrow[r, bend left, "\overline{}"] \arrow[r, phantom, "\cong"] & A \to C^B \arrow[l, bend left, "\widehat{}"] \end{tikzcd} \qquad f = \overline{g} \Leftrightarrow \widehat{f} = g \tag{4}$$

Curry / uncurrying — a very important device.

Distribution transpose

Let a **matrix** be denoted by a function $m : A \times B \to \mathbb{R}$ such that $m\,(a, b)$ yields the contents of the **cell** addressed by **column** index a and **row** index b.

A matrix is said to be **column-stochastic** (CS) wherever $\overline{m}\ a$ is a distribution, for all $a \in A$.

Then:

$$\begin{tikzcd} A \to \mathbb{D}\,B \arrow[rr, bend left, "\widehat{\cdot}"] \arrow[rr, phantom, "\cong"] & & A \times B \to [0,1] \arrow[ll, bend left, "\overline{\cdot}"] \end{tikzcd} \qquad f = \overline{m} \Leftrightarrow \widehat{f} = m \qquad (5)$$

In words: CS **matrices** can be represented by “distribution”-valued **functions**.

Monads

Monads

All functors above — powerset $\mathbb{P}$, maybe $\mathbb{M}$ and distribution $\mathbb{D}$ are **monads**.

Recalling monads:

$$X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}\ X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2\ X$$

satisfying the equalities captured by the following commutative diagrams:

$$\begin{tikzcd}
\mathbb{F}\ X \arrow[r, "\eta_{\mathbb{F}}"] \arrow[d, "\mathbb{F}\ \eta_{\mathbb{F}}"'] \arrow[dr, "id"] & \mathbb{F}^2\ X \arrow[d, "\mu_{\mathbb{F}}"'] & \mathbb{F}^3\ X \arrow[l, "\mu_{\mathbb{F}}"'] \arrow[d, "\mathbb{F}\ \mu_{\mathbb{F}}"] \\
\mathbb{F}^2\ X \arrow[r, "\mu_{\mathbb{F}}"'] & \mathbb{F}\ X & \mathbb{F}^2\ X \arrow[l, "\mu_{\mathbb{F}}"']
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "\eta_{\mathbb{F}}"] \arrow[d, "f"] & \mathbb{F}\ X \arrow[d, "\mathbb{F}\ f"] & \mathbb{F}^2\ X \arrow[l, "\mu_{\mathbb{F}}"'] \arrow[d, "\mathbb{F}^2\ f"] \\
Y \arrow[r, "\eta_{\mathbb{F}}"] & \mathbb{F}\ Y & \mathbb{F}^2\ Y \arrow[l, "\mu_{\mathbb{T}}"']
\end{tikzcd}$$

Kleisli composition

Kleisli composition for monad $\mathbb{F}$:

$$
\begin{tikzcd}
\mathbb{F}\,(\mathbb{F}\,C) \arrow[d, "\mu_{\mathbb{F}}"'] & \mathbb{F}\,B \arrow[l, "\mathbb{F}\,f"'] \arrow[d, dash, dotted] & A \arrow[l, "g"'] \arrow[dll, bend left, "f \bullet g"] \\
\mathbb{F}\,C & B \arrow[l, "f"'] &
\end{tikzcd}
$$

Properties:

$$f \bullet (g \bullet h) = (f \bullet g) \bullet h$$
$$f \bullet \eta_{\mathbb{F}} = f = \eta_{\mathbb{F}} \bullet f$$

Conceptually, it is as if one (typewise) drops the $\mathbb{F}$'s from f and g in the diagram above.

Monads pointwise

In the same way pointwise composition is defined by

$$(f \cdot g)\ a = \textbf{let}\ b = g\ a\ \textbf{in}\ f\ b$$

there is a similar notation for pointwise Kleisli composition (the so-called **do**-notation):

$$(f \bullet g)\ a = \textbf{do}\ \{ b \leftarrow g\ a; f\ b \} \tag{6}$$

This extends to multiple bindings, as e.g. in monadic *pairing* (splits):

$$(f \triangledown g)\ x = \textbf{do}\ \{ a \leftarrow f\ x; b \leftarrow g\ x; \eta\ (a, b) \} \tag{7}$$

Example in $\mathbb{D}$

Probability of the sum of two dice:

$$\mathbf{do}\ \{x \leftarrow \mathit{uniform}\ [1\mathinner{..}6]; y \leftarrow \mathit{uniform}\ [1\mathinner{..}6]; \eta\ (x+y)\}$$

Using the Haskell PFP library by Erwig and Kollmannsberger (2006):

```
*Main> do { x <- uniform [1..6] ;
y <- uniform [1..6] ; return(x+y) }
 7  16.7%
 6  13.9%
 8  13.9%
 5  11.1%
 9  11.1%
 4   8.3%
10   8.3%
 3   5.6%
11   5.6%
 2   2.8%
12   2.8%
```

The most likely sum is 7, with 16.7% probability.

Genericity of the monad concept

The same code runs for different monads, e.g.

$$\textbf{do}\ \{x \leftarrow \mathit{Just}\ 3; y \leftarrow \mathit{Nothing}; \eta\ (x+y)\}$$

yielding `Nothing`, while

$$\textbf{do}\ \{x \leftarrow \{1,2\}; y \leftarrow \{-1,0\}; \eta\ (x+y)\}$$

yielding $0, 1, 2$ and so on.

Note how post-composition with η converts any function into a monadic function, e.g. $\eta \cdot \mathit{add}$ above, for $\mathit{add}\ (x,y) = x + y$.

Still the $\mathbb{D}$ monad

We can now explain the

$$bad\ {}_{p}\!\diamond\ good$$

notation given earlier (1).

Given $f : A \rightarrow \mathbb{D}\ B$ and $g : A \rightarrow \mathbb{D}\ B$, we define

$$f\ {}_{p}\!\diamond\ g = p * f + (1 - p) * g$$

where, in general,

$$(p * f)\ x = p * (f\ x)$$
$$(f + g)\ x = (f\ x) + (g\ x)$$

Recalling from above

$id\ {}_{(0.01)}\!\diamond\ neg$:

$$
\begin{aligned}
& id\ {}_{(0.01)}\!\diamond\ neg \\
= \quad & 0.01 \times \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 1 & 0 \\ \textbf{True} & 0 & 1 \end{array} \right) + 0.99 \times \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0 & 1 \\ \textbf{True} & 1 & 0 \end{array} \right) \\
= \quad & \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0.01 & 0 \\ \textbf{True} & 0 & 0.01 \end{array} \right) + \left(\begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0 & 0.99 \\ \textbf{True} & 0.99 & 0 \end{array} \right) \\
= \quad & \begin{array}{l|cc} & \textbf{False} & \textbf{True} \\ \hline \textbf{False} & 0.01 & 0.99 \\ \textbf{True} & 0.99 & 0.01 \end{array}
\end{aligned}
$$

In Haskell

PFP library by Erwig and Kollmansberger Erwig and Kollmannsberger (2006):

```
schoice (0.01) id not True
False  99.0%
 True   1.0%

schoice (0.01) id not False
 True  99.0%
False   1.0%
```

where `schoice p f g` is the **concrete syntax** for $f\ {}_p\!\diamond\ g$.

Exercises

Exercise 1: A way to **totalize** simple relations (partial functions) in Haskell is to use the $\mathbb{M}$ monad, by adding a pre-condition as another input: the totalized function will deliver Nothing wherever the pre-condition fails.
Write a pointwise and a pointfree definition of such a totalizer, with type

$$\cdot \Leftarrow \cdot :: (b \to a) \to (b \to \mathbb{B}) \to b \to \mathbb{M}\ a$$

and evaluate $\mathit{tail} \Leftarrow (\neg \cdot \mathit{empty})\ [\,]$ and $(/2) \Leftarrow ({>}0)\ 3$, where $\mathit{empty} = ({=}[\,])$.

The concrete syntax for $f \Leftarrow p$ should be `tot f p`. □

Exercises

Exercise 2: Specify a $\mathbb{D}$-monadic function $f\ n$ that yields $n+1$ with 99% probability and n with 1% probability. □

Exercise 3: Write the Haskell code for a monadic for-loop combinator of type

$$mfor :: (Integral\ n, Monad\ m) \Rightarrow (a \rightarrow m\ a) \rightarrow m\ a \rightarrow n \rightarrow m\ a$$

Then calculate the output distribution given by

$$mfor\ f\ (\eta\ 0)\ 10$$

where f was defined in the previous exercise. □

Mealy machines

State monad

Given any function of type $S \times I \rightarrow S \times O$, it can be converted into a function of type $I \rightarrow \mathbb{S}\ O$ where

$$\mathbb{S}\ X = (S \times X)^S \tag{8}$$

$\mathbb{S}$ is another monad, and a very important one — the **state monad**.

Elements of $\mathbb{S}\ X$ describe **actions** over a state S with outputs in X.

Note that any $h \in \mathbb{S}\ X$ is always of the form $h = \langle f, g \rangle$, where $f : S \rightarrow S$ updates the state and $g : S \rightarrow O$ yields an output.

Example: take $push\ (s, i) = (i : s, Ok)$; clearly,

$$push\ i = h \ \textbf{where}\ h = \langle (i{:}), \underline{Ok} \rangle$$

Some generic actions

Get the value of the state

$$get = \langle id, id \rangle$$

Modify the state:

$$modify\ f = \langle f, ! \rangle$$

Put a value in the state:

$$put\ s = modify\ \underline{s}$$

Query the state:

$$query\ f = \langle id, f \rangle$$

A simple transation:

$$trans\ g\ f = \mathbf{do}\ \{ modify\ g; query\ f \}$$

Mealy machines

A function $m : S \times I \to S \times O$ is called a (deterministic) **Mealy machine**.

Mealy machines, in practice, need to be more elaborate because of either partial behaviour (functions undefined for some inputs), non-determinism (vague operations) or probabilistic behaviour (e.g. probability of faults).

Thus a **monad** is required in the type definition:

$$m : S \times I \to \mathbb{D}\,(S \times O)$$

For $\mathbb{F} := \mathbb{P}$, $m = \Lambda R$ where R is a relational (non-deterministic) machine.

Below we shall see a concrete example involving $\mathbb{F} := \mathbb{M}$.

From functional models to objects (CBS)

The process of building CBS systems suggested in the sequel is made of several steps:

- Build a **relational** model of the problem in hands (this was covered in the course *Specification & Modelling*)
- Derive a **functional** model of the relational model — using any of the **transposes** studied above
- Promote a particular **type** S of the previous model to **state** of the component — i.e. **object** — to be built.
- At least one function of the model must have type $\mathbb{F}\ S \rightarrow \mathbb{G}\ S$, to express state transitions (otherwise the object would have no behaviour).

Then (next slide):

From functional models to objects (CBS)

- Convert each function into a **method** — an elementary Mealy machine.
- **Objectification** — build an **object** by “adding” all methods together.
- Transpose the final Mealy machine using the **state monad**.

Then develop combinators to **compose** the **objects** (software components) thus obtained.

The Mealy machines will be monadic (MMM) in general.

We shall see this process at work over a simple example — building a **stack object**.

Example — stack

Functional model of a **stack** (in Haskell):

$$push = flip\ (:)$$

$$pop = tail$$

$$top = head$$

$$empty = (0 =) \cdot length$$

Types:

$$push :: ([a], a) \to [a]$$
$$pop :: [a] \to [a]$$
$$top :: [a] \to a$$
$$empty :: [a] \to \mathbb{B}$$

Clearly, $S = [a]$ can act as **state** of a Mealy machine, with some extra I/O typing.

Methods = elementary Mealy machines

Example of a method

$$push' :: ([a], a) \rightarrow ([a], 1)$$
$$push' = \langle \widehat{push}, ! \rangle$$

which resorts

(a) to the **uncurry** operator,

$$\widehat{f}\ (a, b) = f\ a\ b$$

(b) to the **pairing** operator,

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

(c) and to uniquely defined (total) function $! :: b \rightarrow 1$ ('bang').

However: partiality, the rule rather than exception

Partiality, however, requires 'Maybe' ($\mathbb{M}$) Mealy machines, one per totalized (partial) function, eg.:

$$
\begin{array}{l}
pop' :: ([a], 1) \to \mathbb{M}\ ([a], a) \\
pop' = \langle pop, top \rangle \ \Leftarrow\ (\neg \cdot empty)\ \cdot \pi_1
\end{array}
$$

where $\cdot \ \Leftarrow\ \cdot$ totalizes a partial function by fusion with a **precondition**,

$$
\begin{array}{l}
\cdot\ \Leftarrow\ \cdot\ :: (a \to b) \to (a \to \mathbb{B}) \to a \to \mathbb{M}\ b \\
f\ \Leftarrow\ p\ = cond\ p\ (\eta \cdot f)\ \bot
\end{array}
$$

where unit η (of $\mathbb{M}$) means **success** and 'zero' element $\bot$ means **failure**.

Standard stack methods

$$empty' :: ([a], 1) \rightarrow \mathbb{M}\ ([a], \mathbb{B})$$
$$empty' = \eta \cdot \langle id, empty \rangle \cdot \pi_1$$

$$top' :: ([a], 1) \rightarrow \mathbb{M}\ ([a], a)$$
$$top' = (\langle id, top \rangle \ \Leftarrow\ (\neg \cdot empty)\ ) \cdot \pi_1$$

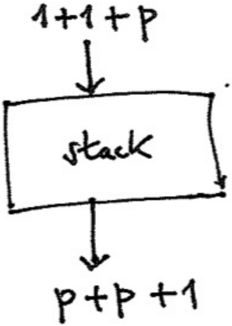
$$push' :: ([b], b) \rightarrow \mathbb{M}\ ([b], 1)$$
$$push' = \eta \cdot \langle \widehat{push}, ! \rangle$$

$$pop' :: ([a], 1) \rightarrow \mathbb{M}\ ([a], a)$$
$$pop' = (\langle pop, top \rangle \ \Leftarrow\ (\neg \cdot empty)\ ) \cdot \pi_1$$

Component $= \sum$ methods

The stack **component**

$$
\begin{aligned}
&\mathit{stack} :: ([p], (1+1)+p) \rightarrow \mathbb{M}\ ([p], (p+p)+1)\\
&\mathit{stack} = \mathit{pop}' \oplus \mathit{top}' \oplus \mathit{push}'
\end{aligned}
$$



is built thanks to the MMM **sum** combinator

$$
\begin{aligned}
&\cdot \oplus \cdot :: (\mathit{Functor}\ \mathbb{F}) \Rightarrow\\
&\quad \text{-- input machines}\\
&\quad ((s,i) \rightarrow \mathbb{F}\ (s,o)) \rightarrow\\
&\quad ((s,j) \rightarrow \mathbb{F}\ (s,p)) \rightarrow\\
&\quad \text{-- output machine}\\
&\quad (s, i+j) \rightarrow \mathbb{F}\ (s, o+p)\\
&\quad \text{-- definition}\\
&m_1 \oplus m_2 = (\mathbb{F}\ \mathit{undistr}) \cdot \Delta \cdot (m_1 + m_2) \cdot \mathit{distl}
\end{aligned}
$$

where (next slide)

Object $= \sum$ methods

- $m_1 + m_2$ is functional sum (coproduct);
- isomorphisms

$$
\begin{array}{l}
\mathit{distl} :: (b, c + a) \to (b, c) + (b, a) \\
\mathit{undistr} :: (a, b) + (a, c) \to (a, b + c)
\end{array}
$$

  handle the shared state across input and output sums;
- "Cozip" operator

$$
\begin{array}{l}
\Delta :: (\mathit{Functor}\ \mathbb{F}) \Rightarrow (\mathbb{F}\ a) + (\mathbb{F}\ b) \to \mathbb{F}\ (a + b) \\
\Delta = [\mathbb{F}\ i_1, \mathbb{F}\ i_2]
\end{array}
$$

  promotes coproducts through $\mathbb{F}$.

Exercises

Exercise 4: Evaluate expressions to express:

- pushing `"a"` into an empty stack
- poping from an empty stack
- getting the top of a stack with at least one element.

□

CBS Systems = comunicating objects

Consider the idea of building a system in which two stacks **interact** with each other, e.g. by popping from one and pushing the outcome onto the other.

For this another MMM combinator is needed taking two I/O-compatible MMM m_1 and m_2 (with different internal states in general) and building a third one, $m_1 \,;\, m_2$, in which outputs of m_1 are sent to m_2:

$$\xrightarrow{I} \boxed{m_1} \xrightarrow{J} (\,;\,) \xrightarrow{J} \boxed{m_2} \xrightarrow{K} \tag{9}$$

CBS Systems = comunicating objects

The type of this combinator as implemented in Haskell is

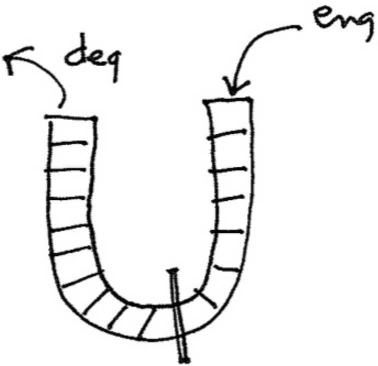
$$
\begin{array}{l}
(;) :: (\mathit{Strong}\ \mathbb{F}, \mathit{Monad}\ \mathbb{F}) \Rightarrow \\
\quad \text{-- input (sender) machine} \\
\quad ((s, i) \to \mathbb{F}\ (s, j)) \to \\
\quad \text{-- input (receiver) machine} \\
\quad ((r, j) \to \mathbb{F}\ (r, k)) \to \\
\quad \text{-- output (compound) machine} \\
\quad ((s, r), i) \to \mathbb{F}\ ((s, r), k)
\end{array}
$$

It requires $\mathbb{F}$ to be a **strong** monad [2], a topic to be addressed later. Note how the output **compound** machine has a **composite state** pairing the states of the two input machines.

[2] Details in (Oliveira and Miraldo, 2016).

Composing objects

Queue = two stacks:



out stack (left) interacting with **in** stack (right)

Object (MMM) sequential composition

Let m_1, m_2 be two machines of types $S \times I \to \mathbb{F}\,(S \times J)$ and $Q \times J \to \mathbb{F}\,(Q \times K)$, respectively.

We will represent these machines by arrows $I \xrightarrow[S]{m_1} J$ and $J \xrightarrow[Q]{m_2} K$, respectively.[3] Then their sequential *composition* $I \xrightarrow[S \times Q]{m_1;m_2} K$ is a machine with composite state $S \times Q$ built as is explained next.

[3]Wherever the state S of a machine $A \xrightarrow[S]{m} B$ is implicit from the context, simplified notation $A \xrightarrow{m} B$ will be used instead.

Sequential composition

First, we build $I \xrightarrow[S \times Q]{\textbf{extr}\ \ m_1} J$, the state-extension of m_1 with the state Q of m_2

$$\begin{tikzcd}
\mathbb{F}\,((S \times J) \times Q) & \mathbb{F}\,(S \times J) \times Q \arrow[l, "\tau_r"'] & (S \times I) \times Q \arrow[l, "m_1 \times id"'] \\
\mathbb{F}\,((S \times Q) \times J) \arrow[u, "\mathbb{F}\ \mathsf{xr}"] & & (S \times Q) \times I \arrow[ll, "\textbf{extr}\ \ m_1"] \arrow[u, "\mathsf{xr}"']
\end{tikzcd}$$

where

- $\mathsf{xr} : (A \times B) \times C \to (A \times C) \times B$ is the obvious isomorphism
- $\tau_r : (\mathbb{F}\ A) \times B \to \mathbb{F}\ (A \times B)$ is the right *strength* of monad $\mathbb{F}$.

Sequential composition

So $I \xrightarrow[S \times Q]{\mathbf{extr}\ m_1} J$ has the same interface as $I \xrightarrow[S]{m_1} J$, only the state differs. In turn, m_2 is extended in the same way,

$$
\begin{tikzcd}
\mathbb{F}\,(S \times (Q \times K)) & S \times \mathbb{F}\,(Q \times K) \arrow[l, "\tau_l"'] & S \times (Q \times J) \arrow[l, "id \times m_2"'] \\
\mathbb{F}\,((S \times Q) \times K) \arrow[u, "\mathbb{F}\ \mathsf{assocr}"] & & (S \times Q) \times J \arrow[ll, "\mathbf{extl}\ m_2"] \arrow[u, "\mathsf{assocr}"']
\end{tikzcd}
$$

adding the state of m_1 to the left, where
$\tau_l : (B \times \mathbb{F}\ A) \to \mathbb{F}\,(B \times A)$ is the left strength of $\mathbb{F}$, and assocr and *assocl* are well-known.

Therefore:

$$\mathbf{extl}\ m = \mathbb{F}\ assocl \cdot \tau_l \cdot (id \times m) \cdot \mathsf{assocr} \tag{10}$$

$$\mathbf{extr}\ m = \mathbb{F}\ \mathsf{xr} \cdot \tau_r \cdot (m \times id) \cdot \mathsf{xr} \tag{11}$$

Sequential composition

Putting both extensions together

$$\begin{tikzcd}
 & \mathbb{F}\,((S \times Q) \times J) \arrow[d, dotted, no head] & (S \times Q) \times I \arrow[l, "\mathbf{extr}\ m_1"'] \arrow[dll, bend left, "(ext\ m_2)\bullet(ext\ m_1)"] \\
\mathbb{F}\,((S \times Q) \times K) & (S \times Q) \times J \arrow[l, "\mathbf{extr}\ m_2"'] &
\end{tikzcd}$$

we get the meaning of object composition

$$m_1 \ ; m_2 \ = \ (\mathbf{extl}\ m_2) \bullet (\mathbf{extr}\ m_1) \tag{12}$$

which unfolds to

$$m_1 \ ; m_2 \ = \ ((\mathbb{F}\ assocl) \cdot \tau_l \cdot (id \times m_2) \cdot \mathrm{xl}) \bullet (\tau_r \cdot (m_1 \times id) \cdot \mathrm{xr})$$

Exercises

Exercise 5: Define

$$m = pop' \; ; \; push'$$

which pops from a source stack ($m_1 = pop'$) and pushes onto a target stack ($m_2 = push'$). Then run $m\ (([1], [2]), ())$ and $m\ (([], [2]), ())$ and observe the outcome. □

Exercise 6: What is the type of $stack \; ; \; stack$?

□

Exercise 7: Define an ($\mathbb{M}$-MMM) object $queue = enq \oplus deq$ which should behave as a *queue*, with two methods: enq — *enqueue*, add to the queue — and deq — *dequeue*, remove from the queue.

□

Object (MMM) interfacing

From above we see the need for some mechanism to control how methods talk to each other when composing two objects.

Such a mechanism is called **interface**-wrapping:

$$
\begin{array}{l}
\cdot_{\{\cdot \rightarrow \cdot\}} :: (\mathit{Functor}\ \mathbb{F}) \Rightarrow \\
\quad \text{-- input machine} \\
((a, e) \rightarrow \mathbb{F}\ (a, c)) \rightarrow \\
\quad \text{-- input wrapper} \\
(i \rightarrow e) \rightarrow \\
\quad \text{-- output wrapper} \\
(c \rightarrow d) \rightarrow \\
\quad \text{-- output machine} \\
(a, i) \rightarrow \mathbb{F}\ (a, d) \\
\quad \text{-- definition} \\
m_{\{f \rightarrow g\}} = \mathbb{F}\ (id \times g) \cdot m \cdot (id \times f)
\end{array}
$$

Object (MMM) interfacing

Note the types in:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rrr, bend right, "m_{\{f\to g\}}", "S"'] & I \arrow[r, "m", "S"'] & J \arrow[r, "g"] & B \end{tikzcd}$$

This diagram is explained in the following exercise:

Exercise 8: Using the laws of Kleisli composition, show that

$$m_{\{f\to g\}} = \ulcorner g \urcorner \bullet m \bullet \ulcorner f \urcorner \tag{13}$$

holds, where

$$\begin{aligned} \ulcorner \cdot \urcorner &: (I \to O) \to (S \times I \to \mathbb{F}\,(S \times O)) \\ \ulcorner f \urcorner &= \eta \cdot (id \times f) \end{aligned} \tag{14}$$

lifts functions to MMMs. $\square$

Universal property of MMM sums

Given two $\mathbb{F}$-monadic Mealy machines $I \xrightarrow[s]{p} O$ and $J \xrightarrow[s]{q} P$ (with the same state space) their sum is the machine

$I + J \xrightarrow[s]{p \oplus q} O + P$ defined by the following universal property:

$$k = p \oplus q \quad \Leftrightarrow \quad \left\{ \begin{array}{l} k_{\{i_1 \rightarrow id\}} = p_{\{id \rightarrow i_1\}} \\ k_{\{i_2 \rightarrow id\}} = q_{\{id \rightarrow i_2\}} \end{array} \right. \tag{15}$$

$\square$

Proof: see (Oliveira and Miraldo, 2016).

Therefore:

$$(p \oplus q)_{\{i_1 \rightarrow id\}} = p_{\{id \rightarrow i_1\}} \tag{16}$$

$$(p \oplus q)_{\{i_2 \rightarrow id\}} = q_{\{id \rightarrow i_2\}} \tag{17}$$

Exchange law

Universal property (15) is central to the calculation of the *exchange law* between machine sum and machine composition which follows:

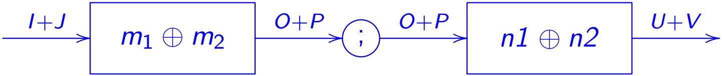
Let $I \xrightarrow[S]{m_1} O$, $J \xrightarrow[S]{m_2} P$ *and* $O \xrightarrow[Q]{n1} U$, $P \xrightarrow[Q]{n2} V$ *be pairs of* $\mathbb{F}$*-monadic Mealy machines (each pair sharing the same state space). Then the following exchange law holds, showing how sequential composition commutes with sums:*

$$(m_1 \oplus m_2) ; (n1 \oplus n2) = (m_1 ; n1) \oplus (m_2 ; n2) \quad (18)$$

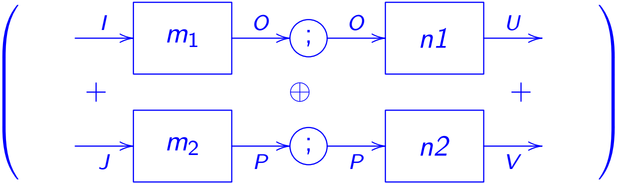
Proof: see (Oliveira and Miraldo, 2016).

Exchange law

In pictures, law (18) shows that the composition of machines



is the same machine as



Changing effect of composition

Sometimes, we need to reverse composite machines:

$$
\begin{aligned}
&\overleftarrow{\cdot} :: \mathit{Functor}\ \mathbb{F} \Rightarrow \\
&\quad \text{-- original MM} \\
&\quad (((b, a), i) \to \mathbb{F}\ ((b, a), o)) \to \\
&\quad \text{-- changed MM} \\
&\quad ((a, b), i) \to \mathbb{F}\ ((a, b), o) \\
&\overleftarrow{m} = \mathbb{F}\ (\mathit{swap} \times \mathit{id}) \cdot m \cdot (\mathit{swap} \times \mathit{id})
\end{aligned}
$$

where

$$\mathit{swap}\ (b, a) = (a, b)$$

thus changing which component is affected first in a composition.

Case study — folder

Let

$$folder = right \oplus left \oplus rd \oplus new$$

where

$$right = pop' \,;\, push'$$

$$left = \overleftarrow{right}$$

$$new = \mathbf{extl}\ push'$$

$$rd = \mathbf{extl}\ top'$$

or $new = nop \,;\, push'$, $rd = nop \,;\, top'$
where $nop = \eta$.

Summary

Component

(Monadic) Mealy machine (MMM), that is, an $\mathbb{F}$ *-branching transition structure of type:*

$$S \times I \to \mathbb{F}(S \times O) \tag{19}$$

where $\mathbb{F}$ *is a monad.*

Component-oriented design

Using MMM combinators as seen thus far.

Semantics

There are two alternative transpositions of $\mathbb{F}$ *-branching transition structure (19):*

State-monadic*:*

$$I \to (\mathbb{F}\,(S \times O))^S \tag{20}$$

Coalgebraic*:*

$$S \to (\mathbb{F}(S \times O))^I \tag{21}$$

Abstracting from the state

Recalling the **state monad** (8):

$$\mathbb{S}\ X = (S \times X)^S$$

we have the following diagram relating two base monads $\mathbb{S}$ and $\mathbb{F}$

$$\begin{tikzcd} & (\mathbb{F}\ (S \times X))^S & \\ \mathbb{F}\ X \arrow[ur, "i_{\mathbb{F}}"] & & \mathbb{S}\ X \arrow[ul, "i_{\mathbb{S}}"'] \end{tikzcd}$$

to the compound one — $\mathbb{T}\ X = (\mathbb{F}\ (S \times X))^S$ — where

$$i_{\mathbb{S}} = \eta^S$$
$$i_{\mathbb{F}} = \overline{(\mathbb{F}\ swap) \cdot \tau_r}$$

Abstracting from the state

Given a MMM $S \times I \xrightarrow{m} \mathbb{F}\,(S \times O)$, we denote by $m_{\mathbb{T}}$ the $\mathbb{T}$-transposed version of m, that is, $I \xrightarrow{m_{\mathbb{T}}} \mathbb{T}\ O$.

For instance, let

$$a = push'_{\mathbb{T}}\ 2$$

Then action a will be such that $[\![a]\!]\ [1] = Just\ ((), [2, 1])$.[4]

Moreover, we can **thread** $\mathbb{T}$-actions without caring about passing the state explicitly. For instance, let the thread

$$t = \mathbf{do}\ \{pop'_{\mathbb{T}}\ (); push'_{\mathbb{T}}\ 2\}$$

be defined.

[4]The meaning of $[\![a]\!]$ will be explained later on.

Abstracting from the state

Then run it over a starting state $s_0 = [0]$,

$$[\![t]\!] \; s_0 = \mathit{Just} \; ()$$

as expected. Now run the thread $\textbf{do} \, \{ \mathit{pop}'_{\mathbb{T}} \; (); t \}$:

$$[\![\, \textbf{do} \, \{ \mathit{popT} \; (); t \}]\!] \; s_0 = \mathit{Nothing}$$

This fails over the same starting state because the starting stack is empty.

Note how $\mathbb{M}$ effects and $\mathbb{S}$ effects are blended in an implicit way via the $\mathbb{T}$ compound monad.

NB: the pretty-printing above hides a number of details of the implementation. These can be found in module `SMT.hs`.

Faulty components

Why fauty components?

In the trend towards **miniaturization** of automated systems the size of circuit transistors cannot be reduced endlessly, as these eventually become **unreliable**.

There is, however, the idea that inexact hardware can be *tolerated* provided it is *"good enough"* (Lingamneni et al., 2013).

Good enough has always been the way **engineering** works as a broad discipline.

If unreliable hardware becomes widely accepted on the basis of fault tolerance guarantees, what will the **impact** of this be on the **software** layers which run on top of it in virtually any automated system?

Recall

Software **V&V**

compared with. . .

Motivation

MITnews

engineering science management architecture + planning humanities, arts, and social sciences campus

The surprising usefulness of sloppy arithmetic

A computer chip that performs imprecise calculations could process some types of data thousands of times more efficiently than existing chips.

Larry Hardesty, MIT News Office

Sloppy arithmetic useful?

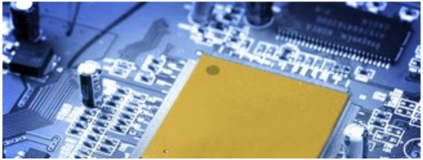
Horror!

But there is more. . .

“Just good enough” h/w

. . . coming from the land of the Swiss watch:

“We should stop designing perfect circuits”



02.10.13 - **Are integrated circuits "too good" for current technological applications? Christian Enz, the new Director of the Institute of Microengineering, backs the idea that perfection is overrated.**

Message:

> *Why* **perfection** *if (some)* **imperfection** *still meets the standards?*

S/w for "just good enough" h/w

What about **software** running over "just good enough" hardware?

Ready to **take the risk**?

Nonsense to run **safety critical** software on **defective** hardware?

Uups! — it seems "it already runs":

medical design *"IEC 60601-1 [brings]* **risk management** *into the very first stages of [product development]"*

Risk is everywhere — an inevitable (desired?) part of life.

Faulty components

Risk of pop' behaving like top' with **probability** $1 - p$

$$pop'' :: \mathbb{P} \to ([a], 1) \to \mathbb{D}\,(\mathbb{M}\,([a], a))$$
$$pop''\ p = pop'\ {}_p\!\diamond\ top'$$

and risk of $push'$ not pushing anything, with probability $1 - q$

$$push'' :: \mathbb{P} \to ([a], a) \to \mathbb{D}\,(\mathbb{M}\,([a], 1))$$
$$push''\ q = push'\ {}_q\!\diamond\ skip'$$

where $\mathbb{P} = [0, 1]$, $\mathbb{D}$ is the (finite) **distribution** monad and

$$\cdot\ {}_\cdot\!\diamond \cdot :: \mathbb{P} \to (t \to a) \to (t \to a) \to t \to \mathbb{D}\ a$$
$$(f\ {}_p\!\diamond\ g)\ x = choose\ p\ (f\ x)\ (g\ x)$$

choses between f and g .

Simulation

Example (no faults) — popping from one stack and pushing onto another,

$$m_1 = pop' \; ; \; push'$$

should produce the intended behaviour, eg.

```
 > curry m1 ([1],[2]) ()
Just (([],[1,2]),())
 > curry m1 ([],[2]) ()
Nothing
```

Example (faulty stacks) — now suppose the stacks are faulty,

$$m_2 = pop'' \; 0.95 \; ; \; push'' \; 0.8$$

over the same (global) state `([1],[2])`.

Simulation

Running the same simulation, now for machine m_2,

```
 > curry m2 ([1],[2]) ()
 Just (([],[1,2]),())  76.0%
   Just (([],[2]),())  19.0%
Just (([1],[1,2]),())   4.0%
  Just (([1],[2]),())   1.0%
```

the risk of faulty behaviour is 24% $(1 - 0.76)$, structured as: (a) 1% — both components misbehave; (b) 19% — left stack misbehaves; (c) 4% — right stack misbehaves.

As expected,

```
 > curry m2 ([],[2]) ()
Nothing 100.0%
```

is **catastrophic** (popping from an empty stack).

Faulty components

Simulation:

Using the **PFP library** *written by Erwig and Kollmannsberger (2006).*

Important:

Our MMMs have become probabilistic, leading to actions of general shape

$$I \to (\mathbb{D}(\mathbb{F}(S \times O)))^S$$

Challenge:

Need for the probabilistic extension of the MMM combinators above.

Question: do we need to start **all over again** for the **probabilistic** case?

Monads, again

No. :-)

Note that all our combinators are parametric on a "branching"-monad $\mathbb{F}$.

It all amounts to check whether the composition $\mathbb{D} \cdot \mathbb{M}$ forms a **monad** or not.

Monad composition is a well-studied subject. We just need, in our case, to check whether there is a function $\lambda : (\mathbb{M} \cdot \mathbb{D})\ X \to (\mathbb{D} \cdot \mathbb{M})\ X$ satisfying some laws (see below).

Such a function indeed exists:

$$\lambda\ \mathit{Nothing} = \eta\ \mathit{Nothing}$$
$$\lambda\ (\mathit{Just}\ a) = \mathbb{D}\ \mathit{Just}\ a$$

Composing monads

Here is how composite monad $\mathbb{H} = \mathbb{D} \cdot \mathbb{M}$ is built, assuming that $\mathbb{D}$ and $\mathbb{M}$ are so:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathbb{H}\ \textbf{where} \\
\quad \mathit{eta} = \eta_{\mathbb{D}} \cdot \eta_{\mathbb{M}} \\
\quad x \gg\!= f = (\mu \cdot \mathbb{D}\ (\mathbb{M}\ f))\ x\ \textbf{where}\ \mu = (\mathbb{D}\ \mu_{\mathbb{M}}) \cdot \mu_{\mathbb{D}} \cdot \mathbb{D}\ \lambda
\end{array}
$$

We can do everything as before for **probabilistic** objects (components), for instance addition over a run-time execution stack:

$$
\begin{array}{l}
t'' = \textbf{do}\ \{ \\
\quad x \leftarrow \mathit{pop}''\ 0.8_{\mathbb{T}}\ (); \\
\quad y \leftarrow \mathit{pop}''\ 0.9_{\mathbb{T}}\ (); \\
\quad \mathit{push}''\ 0.6_{\mathbb{T}}\ (x + y); \\
\quad \mathit{pop}''\ 0.7_{\mathbb{T}}\ ()\}
\end{array}
$$

Just 9	48.0%
Nothing	28.8%
Just 8	12.0%
Just 5	10.4%
Just 4	0.8%

such that $[\![t'']\!]\ [4, 5]$ will yield

Distributive laws

Let $X \xrightarrow{\eta_{\mathbb{T}}} \mathbb{T}X \xleftarrow{\mu_{\mathbb{T}}} \mathbb{T}^2X$ and $X \xrightarrow{\eta_{\mathbb{F}}} \mathbb{F}X \xleftarrow{\mu_{\mathbb{F}}} \mathbb{F}^2X$ be two monads.

A **distributive law** of $\mathbb{T}$ over $\mathbb{F}$ is a polymorphic function $\lambda : \mathbb{F}\,\mathbb{T} \to \mathbb{T}\,\mathbb{F}$ such that

$$\lambda \cdot \mathbb{F}\,\eta_{\mathbb{T}} = \eta_{\mathbb{T}} \tag{22}$$

$$\lambda \cdot \mathbb{F}\,\mu_{\mathbb{T}} = \mu_{\mathbb{T}} \cdot \mathbb{T}\,\lambda \cdot \lambda \tag{23}$$

and

$$\lambda \cdot \eta_{\mathbb{F}} = \mathbb{T}\eta_{\mathbb{F}} \tag{24}$$

$$\mathbb{T}\mu_{\mathbb{F}} \cdot \lambda \cdot \mathbb{F}\lambda = \lambda \cdot \mu_{\mathbb{F}} \tag{25}$$

hold.

In Haskell

From Cp.hs:

$$\textbf{class } \mathit{Functor}\ \mathbb{F} \Rightarrow \mathit{DistL}\ \mathbb{F}\ \textbf{where}$$
$$\quad \lambda :: \mathit{Monad}\ \mathbb{T} \Rightarrow \mathbb{F}\ (\mathbb{T}\ a) \rightarrow \mathbb{T}\ (\mathbb{F}\ a)$$

Listas:

$$\textbf{instance } \mathit{DistL}\ [\,]\ \textbf{where } \lambda = \mathit{sequence}$$

Maybe:

$$\textbf{instance } \mathit{DistL}\ \mathbb{M}\ \textbf{where}$$
$$\quad \lambda\ \mathit{Nothing} = \eta\ \mathit{Nothing}$$
$$\quad \lambda\ (\mathit{Just}\ a) = \mathbb{T}\ \mathit{Just}\ a$$

Exercises

Exercise 9: Let $m = (pop''\ 0.6) ; (push''\ 0.5)$. When running

$$[\![m_{\mathbb{T}}\ ()]\!]\ s_0$$

for any initial state s_0 you always get a Dirac distribution as output. Where is the probabilistic behaviour, then? □

Exercise 10: Build a probabilistic folder and exercise it in GHCi. □

More combinators

Parallel composition:

$$
\begin{aligned}
&\cdot \boxtimes \cdot :: (\mathit{Monad}\ \mathbb{F}, \mathit{Strong}\ \mathbb{F}) \Rightarrow \\
&\quad \text{-- input machines} \\
&\quad ((s, i) \to \mathbb{F}\ (s, o)) \to \\
&\quad ((t, j) \to \mathbb{F}\ (t, r)) \to \\
&\quad \text{-- output machine} \\
&\quad ((s, t), (i, j)) \to \mathbb{F}\ ((s, t), (o, r)) \\
&p \boxtimes q = (\mathbb{F}\ m) \cdot \delta \cdot (p \times q) \cdot m
\end{aligned}
$$

where $m = \langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle$ and

$$
\begin{aligned}
&\delta :: \mathit{Strong}\ \mathbb{F} \Rightarrow (\mathbb{F}\ a, \mathbb{F}\ b) \to \mathbb{F}\ (a, b) \\
&\delta = \tau_r \bullet \tau_l
\end{aligned}
$$

is the double strength operator.

Conditionals

Also useful is the MMM-level McCarthy **conditional** combinator,

$$
\begin{array}{l}
\cdot \rightarrow \cdot\ ,\ \cdot :: (\mathit{Monad}\ \mathbb{F}, \mathit{Functor}\ \mathbb{F}) \Rightarrow \\
\quad \text{-- condition} \\
\quad ((a, i) \rightarrow \mathbb{F}\ (a, \mathbb{B})) \rightarrow \\
\quad \text{-- 'then' branch} \\
\quad ((a, 1) \rightarrow \mathbb{F}\ (a, o)) \rightarrow \\
\quad \text{-- 'else' branch} \\
\quad ((a, 1) \rightarrow \mathbb{F}\ (a, o)) \rightarrow \\
\quad \text{-- output} \\
\quad (a, i) \rightarrow \mathbb{F}\ (a, o) \\
\quad \text{-- definition} \\
p \rightarrow m_1\ ,\ m_2 = [m_1, m_2] \bullet (\mathbb{F}\ \mathit{distl} \cdot p_{\{id \rightarrow outB\}})
\end{array}
$$

where $outB$ witnesses isomorphism $\mathbb{B} \cong 1 + 1$.

MMM behavioural equivalence

Two Mealy machines $I \xrightarrow[S]{m_1} J$ and $I \xrightarrow[Q]{m_2} J$ are **behaviourally equivalent** provided a function $h : S \to Q$ can be found such that

$$\begin{tikzcd}
\mathbb{F}(S \times J) \arrow[d, "\mathbb{F}(h \times id)"'] & S \times I \arrow[l, "m_1"'] \arrow[d, "h \times id"] & S \arrow[d, "h"] \\
\mathbb{F}(Q \times J) & Q \times I \arrow[l, "m_2"] & Q
\end{tikzcd}$$

holds:

$$\mathbb{F}(h \times id) \cdot m_1 = m_2 \cdot (h \times id) \tag{26}$$

$h : S \to Q$ is said to be a MM-**morphism** and we write $m_1 \simeq m_2$ to express the equivalence. **Behavioural** equivalence in what sense?

Behavioural equivalence

Let us reason about equality (26):

$$
\begin{array}{ll}
& \mathbb{F}\ (h \times id) \cdot m_1 = m_2 \cdot (h \times id) \\
\Leftrightarrow & \qquad \{\ \text{currying}\ \} \\
& \overline{\mathbb{F}\ (h \times id) \cdot m_1} = \overline{m_2 \cdot (h \times id)} \\
\Leftrightarrow & \qquad \{\ \text{absorption and fusion laws of exponentials}\ \} \\
& (\mathbb{F}\ (h \times id))^I \cdot \overline{m_1} = \overline{m_2} \cdot h \\
\Leftrightarrow & \qquad \{\ \text{define } \mathbb{H}\ X = (\mathbb{F}\ (X \times J))^I\ \} \\
& \mathbb{H}\ h \cdot \overline{m_1} = \overline{m_2} \cdot h
\end{array}
$$

$\overline{m_1} : S \to \mathbb{H}\ S$ and $\overline{m_2} : Q \to \mathbb{H}\ Q$ are said to be $\mathbb{H}$-**coalgebras**.

Behavioural equivalence

From

$$\mathbb{H}\ h \cdot \overline{m_1} = \overline{m_2} \cdot h$$

we infer

$$\overline{m_2} \cdot h \subseteq \mathbb{H}\ h \cdot \overline{m_1}$$

that is, m_1 and m_2 are **bisimilar**. In general, R is a bisimulation if

$$\overline{m_2} \cdot R \subseteq \mathbb{H}\ R \cdot \overline{m_1} \tag{27}$$

holds, that is (pointwise):

$$q'\ R\ q \Rightarrow (\overline{m_2}\ q')\ \mathbb{H}\ R\ (\overline{m_1}\ q)$$

Behaviour coalgebra

Recall

$$\begin{tikzcd}
(\mathbb{F}\ (S \times J))^I \arrow[d, "(\mathbb{F}\ (h \times id))^I"'] & S \arrow[l, "\overline{m_1}"'] \arrow[d, "h"] & S \arrow[d, "h"] \\
(\mathbb{F}\ (Q \times J))^I & Q \arrow[l, "\overline{m_2}"] & Q
\end{tikzcd}$$

In particular, $h = [\![(\overline{m_1})]\!]$ always exists, assigning to each state $s \in S$ the behaviour of m_1 taking s as **starting** state:

$$\begin{tikzcd}
(\mathbb{F}\ (S \times J))^I \arrow[d, "(\mathbb{F}\ (h \times id))^I"'] & S \arrow[l, "\overline{m_1}"'] \arrow[d, "h"] & S \arrow[d, "h"] \\
(\mathbb{F}\ (\Omega \times J))^I & \Omega \arrow[l, "\omega"] & \Omega
\end{tikzcd}$$

$\Omega = (\mathbb{F}\ J^\infty)^{I^\infty}$ — possibly infinite stream of inputs monadically mapped to similar stream of outputs.

Finite approximation

Recall $[\![m]\!] : S \rightarrow (I^{\infty} \rightarrow (\mathbb{F}\ J^{\infty}))$.

Finite approximation,

$$[\![m]\!] :: S \rightarrow ([I] \rightarrow \mathbb{F}\ [J])$$

assuming finite stream of inputs:

$$
\begin{array}{l}
[\![m]\!]\ s\ [\,] = \eta\ [\,] \\
[\![m]\!]\ s\ (i : is) = \mathbf{do}\ \{ \\
\quad (s', j) \leftarrow m\ s\ i; \quad \text{-- get next state, next output} \\
\quad js \leftarrow [\![m]\!]\ s'\ is; \quad \text{-- get behaviour for next state s'} \\
\quad \eta\ (j : js) \} \quad \text{-- append}
\end{array}
$$

Example (stack)

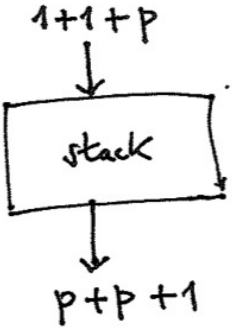
Recall:

$$
\begin{aligned}
&stack :: ([p], (1+1)+p) \\
&\quad \to \mathbb{M}\,([p], (p+p)+1) \\
&stack = (pop' \oplus top') \oplus push'
\end{aligned}
$$

Define, just for convenience, the following "methods":

$$
\begin{aligned}
mPOP &= i_1 \cdot i_1 \\
mTOP &= i_1 \cdot i_2 \\
mPUSH &= i_2
\end{aligned}
$$

and **coalgebra** $c = \overline{stack}$.



Example (stack)

Then experiment with the behaviour of m for finite streams of inputs:

$$Main\!> [\![c]\!]\ [\,]\ [mPUSH\ 2, mPOP\ (), mTOP\ ()]$$
$$Nothing$$
$$Main\!> [\![c]\!]\ [\,]\ [mPUSH\ 2, mTOP\ (), mTOP\ ()]$$
$$Just\ [i_2\ (), i_1\ (i_2\ 2), i_1\ (i_2\ 2)]$$

Exercises

Exercise 11: Define the machine $J \xrightarrow{copy} J$ that faithfully passes its input to the output, never changing state:

$$\begin{aligned} &copy : 1 \times J \to \mathbb{F}\,(1 \times J) \\ &copy = \eta \end{aligned}$$

Show that:

$$m \,;\, copy \;\simeq\; m \;\simeq\; copy \,;\, m \tag{28}$$

$$m \,;\, (n \,;\, p) \;\simeq\; (m \,;\, n) \,;\, p \tag{29}$$

$\square$

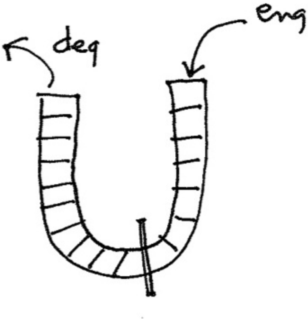
Case study (TP3 assignment)

Implement a queue by composing two stacks.

out stack (left) interacting with **in** stac
(right)

When **out** stack gets empty you should flush into it the data from the other sta

Build a faulty queue by injecting faults the stacks.



Test the behaviour of the (faulty) queue with a distribution of (finite) input streams. Can you measure the probability of the stack flushes?

References

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.

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

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

J. Joyce. Proposed Formal Methods Supplement for RTCA DO 178C, 2011. High Confidence Software and Systems, 11th Annual Conference, 3-6 May 2011, Annapolis.

A. Lingamneni, C. Enz, K. Palem, and C. Piguet. Synthesizing parsimonious inexact circuits through probabilistic design techniques. *ACM Trans. Embed. Comput. Syst.*, 12(2s): 93:1–93:26, May 2013. ISSN 1539-9087.

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

J.N. Oliveira and V.C. Miraldo. "Keep definition, change category" — a practical approach to state-based system calculi. *JLAMP*, 85(4):449–474, 2016.

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