Why category theory matters: a functional programmer's perspective

Conference Paper · June 2015

CITATIONS
0

READS
670

1 author:

José Nuno Oliveira
INESC TEC and Univ. Minho, Braga
117 PUBLICATIONS **796** CITATIONS

SEE PROFILE

Why category theory matters: a functional programmer's perspective.

J.N. Oliveira

INESC TEC & UNIVERSITY OF MINHO

AMS/EMS/SPM 2015 Meeting
FCUP, Porto
Special Session 7, 13th June 2015, Room M107

Abstract

Since the early days of LISP, **functional programming** (FP) has evolved into a solid **paradigm** for producing software.

How did this happen? A look into the past shows how FP has been inspired by **category theory**.

The use of **monads** in main-stream FP surely is one of the "turning points", admitted even by category-theory *non aficionados*.

This tutorial talk addresses recent interest in **adjunctions** as a generic device for **structuring** and **reasoning** about FP code.

Algebras

Functional programming is a *data driven* style of programming — programs react to **input** data by delivering **output** data.

Clearly, one needs to know

- how to **generate** the **output** data.
- how to **inspect** the **input** data (i.e. how it **is built**).

Invariably, data are generated in an **algebraic** manner, i.e. using operators of some **algebra**:

$$A \xleftarrow{a} \mathbf{F}\ A$$

Example: the *Peano algebra*

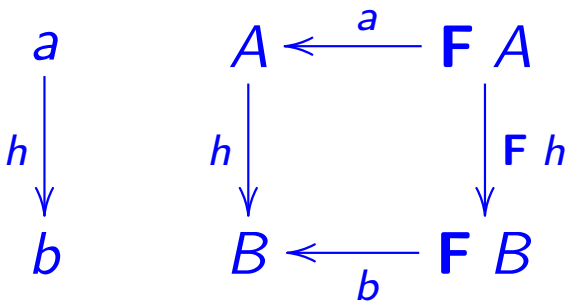
$$\mathbb{N}_0 \xleftarrow{[zero,succ]} 1 + \mathbb{N}_0$$

builds natural numbers, for $zero\ \_ = 0$ and $succ\ n = n + 1$.

Functors

F is an (endo)functor in some category, typically $\mathbb{S}$ (our name for the category of sets and functions between sets).

Quite often functional programs arise as **homomorphisms** between input and output **F-algebras**:



As is well known, such **F-homomorphisms** form a category $\mathbb{C}_{\mathbf{F}}$ whose objects are the algebras themselves.

Homomorphisms

Example: the function $(n\times)$ which multiplies a natural number by some given n is the following $(1+)$-homomorphism:

$$\begin{array}{c} [zero, succ] \\ \downarrow{\scriptstyle (n\times)} \\ [zero, (n+)] \end{array} \qquad \begin{array}{ccc} \mathbb{N}_0 & \xleftarrow{[zero,succ]} & 1+\mathbb{N}_0 \\ \downarrow{\scriptstyle (n\times)} & & \downarrow{\scriptstyle id+(n\times)} \\ B & \xleftarrow[[zero,(n+)]]{} & 1+B \end{array}$$

Multiplication happens because the **output** algebra is n-times faster than the **input** one — it runs $(n+)$ while the input runs $(1+)$:

$$n \times 0 = 0$$
$$n \times (1+m) = n + n \times m$$

Another way to write the same is the **for**-loop:

$$(n\times) \quad = \quad \textbf{for}\ (n+)\ 0$$

Initial algebras

For many $\mathbf{F}$ the category $\mathbb{C}_{\mathbf{F}}$ has initial objects $\mu_{\mathbf{F}} \xleftarrow{i} \mathbf{F}\ \mu_{\mathbf{F}}$.

Example: *Peano algebra* $\mathsf{in}_{\mathbb{N}_0} = [\mathit{zero}, \mathit{succ}]$ is initial for $\mathbf{F} = (1+)$ ($\mu_{\mathbf{F}} = \mathbb{N}_0$).

The unique $\mathbf{F}$-homomorphism from the initial $\mu_{\mathbf{F}} \xleftarrow{i} \mathbf{F}\ \mu_{\mathbf{F}}$ to any other algebra $A \xleftarrow{a} \mathbf{F}\ A$ is written $(\!|a|\!)$:

$$k = (\!|a|\!) \quad \begin{array}{c} i \\ \downarrow \\ a \end{array} \qquad \Leftrightarrow \qquad \begin{array}{ccc} \mu_{\mathbf{F}} & \xleftarrow{\ i\ } & \mathbf{F}\ \mu_{\mathbf{F}} \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle \mathbf{F}\ k} \\ A & \xleftarrow[\ a\ ]{} & \mathbf{F}\ A \end{array}$$

It is termed **catamorphism** of a or **fold** over a.

Catamorphisms

Another example:

$\mathbf{F}\ X = 1 + A \times X$ *for some fixed set* A
$\mu_{\mathbf{F}} = A^\star$ *(finite sequences of* A *s)*
$\mathsf{in} = [nil, cons]$

where

$nil\ \_ = [\,]$ *(empty sequence)*
$cons\ (a, x) = a : x$ *(sequence construction).*

Catamorphism $length = (\!|[zero, succ \cdot \pi_2]|\!)$ computes the length of a sequence, for $\pi_2\ (a, x) = x$.

Algebra $[zero, succ \cdot \pi_2]$ is the composition of the Peano algebra $[zero, succ]$ with natural transformation $\alpha = id + \pi_2$ between the two functors.

Not yet there

However, **many** programs fall outside these schemas, for instance:

$$add\ (0, y) = y$$
$$add\ (1 + x, y) = 1 + add\ (x, y)$$

— not a catamorphism because it has two inputs;

$$sq\ 0 = 0$$
$$sq\ (1 + x) = 2\ x + 1 + sq\ x$$

— not a catamorphism because $[zero, (2\ x + 1+)]$ is not a $(1+)$-algebra (it depends on x);

$$msq\ 0 = return\ 0$$
$$msq\ (1 + x) = \mathbf{do}\ \{y \leftarrow msq\ x; print\ x; return\ (2\ x + 1 + y)\}$$

— **what** ???

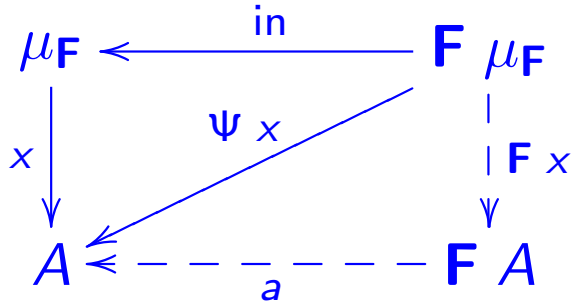
Not yet there

The following problems can be identified:

- In $\mathbb{N}_0 \xleftarrow{add} \mathbb{N}_0 \times \mathbb{N}_0$ there is extra **context** information in the input, that is, instead of $A \xleftarrow{f} \mu_{\mathbf{F}}$ one has $A \xleftarrow{f} \mathbf{L}\ (\mu_{\mathbf{F}})$ for some functor $\mathbf{L}$.
- In $\mathbb{N}_0 \xleftarrow{sq} \mathbb{N}_0$ the output algebra needs to access the input x (catamorphisms **lose the input** too quickly!)
- In $\mathbf{IO}\ (\mathbb{N}_0) \xleftarrow{msq} \mathbb{N}_0$ there is a monadic effect on the output (printing + calculating the output), that is, instead of $A \xleftarrow{f} \mu_{\mathbf{F}}$ one has $\mathbf{M}\ A \xleftarrow{f} \mu_{\mathbf{F}}$ for some **monad** $\mathbf{M}$.

Mendler style

The second observation suggests the so-called **Mendler**-style for catamorphisms:



$$x \cdot \mathsf{in} = \Psi\ x$$

under the requirement that Ψ is a **natural transformation**. In general, given category $\mathbb{C}$ and endofunctor $\mathbb{C} \xleftarrow{\mathbf{F}} \mathbb{C}$, we shall assume some $\mathbb{C}\ (\mathbf{F}\ X, A) \xleftarrow{\Psi} \mathbb{C}\ (X, A)$ such that

$$(\Psi\ f) \cdot \mathbf{F}\ h = \Psi\ (f \cdot h)$$

holds (naturality of Ψ on X).

Mendler style

Can we be sure that equation

$$x \cdot \mathsf{in} = \Psi\ x \tag{1}$$

for any Ψ as above has a unique solution?

The answer is yes, noting that there is an isomorphism between algebras $A \xleftarrow{a} \mathbf{F}\ A$ and natural transformations $\mathbb{C}\ (\mathbf{F}\ X, A) \xleftarrow{\Psi} \mathbb{C}\ (X, A)$: from $A \xleftarrow{a} \mathbf{F}\ A$ we derive

$$\Psi_a\ x = a \cdot \mathbf{F}\ x$$

and from some given Ψ derive $\mathbf{F}$-algebra $(\Psi\ id)$ — recall

$$(\Psi\ f) \cdot \mathbf{F}\ h = \Psi\ (f \cdot h)$$

Mendler style

Let us denote such a solution to $x \cdot \mathsf{in} = \Psi\ x$ by $(\!|\Psi|\!)$ and get re-assured of its uniqueness:

$$
\begin{array}{ll}
 & x \cdot \mathsf{in} = \Psi\ x \\
\Leftrightarrow & \quad \{\ \text{naturality: } \Psi\ x = (\Psi\ id) \cdot \mathbf{F}\ x\ \} \\
 & x \cdot \mathsf{in} = (\Psi\ id) \cdot \mathbf{F}\ x \\
\Leftrightarrow & \quad \{\ \text{catamorphism}\ \} \\
 & x = (\!|\Psi\ id|\!) \\
\Leftrightarrow & \quad \{\ \text{choice of notation above}\ \} \\
 & x = (\!|\Psi|\!) \\
\Box &
\end{array}
$$

Example: **for** $b\ i = (\!|\lambda f \rightarrow [\underline{i}, b \cdot f]|\!)$ where $\underline{i}\ x = i$ for every x.

Calculational properties

FP has a long tradition of relying on calculational rules. From

$$x = (\!|\Psi|\!) \;\Leftrightarrow\; x \cdot \mathsf{in} = \Psi\ x \tag{2}$$

we infer rules such as the **cancellation** rule,

$$(\!|\Psi|\!) \cdot \mathsf{in} = \Psi\ (\!|\Psi|\!)$$

the **reflexion** rule,

$$id = (\!|\Psi|\!) \;\Leftrightarrow\; \Psi\ id = \mathsf{in}$$

the **fusion** rule,

$$h \cdot (\!|\Phi|\!) = (\!|\Psi|\!) \;\Leftarrow\; h \cdot (\Phi\ f) = \Psi\ (h \cdot f)$$

etc., all useful in FP **transformation**, **optimization** and so on.

Mendler style with context

Recall "counter-example"

$$add\ (0, y) = y$$
$$add\ (1 + x, y) = 1 + add\ (x, y)$$

We can write the same as

$$add \cdot [zero \times id, succ \times id] = [\pi_2, succ \cdot add]$$

that is

$$add \cdot (\mathsf{in}_{\mathbb{N}_0} \times id) = [\pi_2, succ \cdot add] \cdot distl$$

where $\mathsf{in}_{\mathbb{N}_0} = [zero, succ]$ and $A \times C + B \times C \xleftarrow{distl} (A + B) \times C$ is the obvious isomorphism.

Mendler style with context

Clearly:

$$add \cdot \underbrace{\mathsf{in}_{\mathbb{N}_0} \times id}_{\mathbf{L}\ \mathsf{in}_{\mathbb{N}_0}} = \underbrace{[\pi_2, succ \cdot add] \cdot distl}_{\Phi\ add}$$

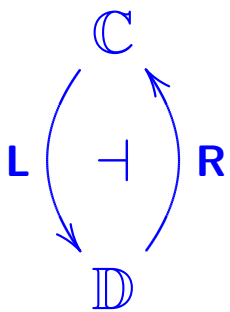
In general, let functor $\mathbf{L}\ X = X \times C$ be given, where object C captures the **context** which surrounds the input in

$$\begin{array}{ccc} \mathbf{L}\ \mu_{\mathbf{F}} & & \mathbf{L}\ \mu_{\mathbf{F}} \xleftarrow{\mathbf{L}\ \mathsf{in}} \mathbf{L}\ (\mathbf{F}\ \mu_{\mathbf{F}}) \\ {\scriptstyle x}\downarrow & & {\scriptstyle x}\downarrow \quad \searrow^{\Phi\ x} \\ B & & B \end{array}$$

Does $x \cdot (\mathbf{L}\ \mathsf{in}) = \Phi\ x$ have a unique solution?

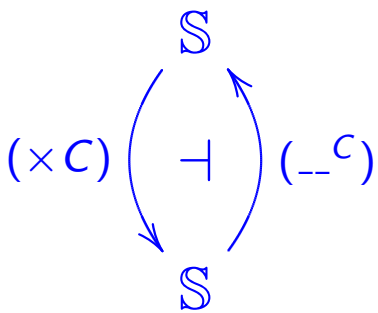
L in for left adjoint

Adjunctions:



$$\mathbb{D}\,(\mathbf{L}\,A, B) \underset{\lambda}{\overset{\lambda^\circ}{\cong}} \mathbb{C}\,(A, \mathbf{R}\,B)$$

In the example:



$$\mathbb{S}\,(A \times C, B) \underset{\mathit{curry}}{\overset{\mathit{uncurry}}{\cong}} \mathbb{S}(A, B^C)$$

Mendler style + adjunction

$$B \xleftarrow{x\cdot(\mathbf{L}\ \mathsf{in})} \mathbf{L}\ (\mathbf{F}\ \mu_{\mathbf{F}}) \;=\; B \xleftarrow{\Phi\ x} \mathbf{L}\ (\mathbf{F}\ \mu_{\mathbf{F}})$$

$\Leftrightarrow$ $\{$ isomorphism λ $\}$

$$\mathbf{R}\ B \xleftarrow{\lambda\ (x\cdot(\mathbf{L}\ \mathsf{in}))} \mathbf{F}\ \mu_{\mathbf{F}} \;=\; \mathbf{R}\ B \xleftarrow{\lambda\ (\Phi\ x)} \mathbf{F}\ \mu_{\mathbf{F}}$$

$\Leftrightarrow$ $\{$ $\mathbf{L} \dashv \mathbf{R}$ — λ naturality $\}$

$$\mathbf{R}\ B \xleftarrow{(\lambda\ x)\cdot\mathsf{in}} \mathbf{F}\ \mu_{\mathbf{F}} \;=\; \mathbf{R}\ B \xleftarrow{(\lambda\cdot\Phi)\ x} \mathbf{F}\ \mu_{\mathbf{F}}$$

$\Leftrightarrow$ $\{$ λ is injective $(\lambda^{\circ} \cdot \lambda = id)$ $\}$

$$\mathbf{R}\ B \xleftarrow{(\lambda\ x)\cdot\mathsf{in}} \mathbf{F}\ \mu_{\mathbf{F}} \;=\; \mathbf{R}\ B \xleftarrow{(\lambda\cdot\Phi\cdot\lambda^{\circ})\ (\lambda\ x)} \mathbf{F}\ \mu_{\mathbf{F}}$$

$\longrightarrow$ overleaf

Mendler style + adjunction

$$\mathbf{R}\ B \xleftarrow{(\lambda\ x)\cdot \mathsf{in}} \mathbf{F}\ \mu_{\mathbf{F}} \;=\; \mathbf{R}\ B \xleftarrow{(\lambda\cdot\Phi\cdot\lambda^\circ)\ (\lambda\ x)} \mathbf{F}\ \mu_{\mathbf{F}}$$

$$\Leftrightarrow \qquad \{\ (2)\ \text{for}\ x := \lambda\ x\ ,\ \Psi = \lambda\cdot\Phi\cdot\lambda^\circ\ \}$$

$$\mathbf{R}\ B \xleftarrow{\lambda\ x} \mu_{\mathbf{F}} \;=\; \mathbf{R}\ B \xleftarrow{(\!|\lambda\cdot\Phi\cdot\lambda^\circ|\!)} \mu_{\mathbf{F}}$$

$$\Leftrightarrow \qquad \{\ \text{isomorphism}\ \lambda^\circ\ \}$$

$$B \xleftarrow{x} \mathbf{L}\ \mu_{\mathbf{F}} \;=\; B \xleftarrow{\lambda^\circ\ (\!|\lambda\cdot\Phi\cdot\lambda^\circ|\!)} \mathbf{L}\ \mu_{\mathbf{F}}$$

Altogether:

$$x\cdot(\mathbf{L}\ \mathsf{in}) = \Phi\ x \quad\Leftrightarrow\quad x = (\!|\Phi|\!)_\lambda \quad\Leftrightarrow\quad \lambda\ x = (\!|\lambda\cdot\Phi\cdot\lambda^\circ|\!) \qquad (3)$$

where $(\!|\Phi|\!)_\lambda$ abbreviates $\lambda^\circ\ (\!|\lambda\cdot\Phi\cdot\lambda^\circ|\!)$.

Calculation properties (extended)

From

$$x \cdot (\mathbf{L}\ \mathsf{in}) = \Phi\ x \ \Leftrightarrow\ x = (\!|\Phi|\!)_\lambda \tag{4}$$

we infer **extended** rules such as the **cancellation** rule,

$$(\!|\Phi|\!)_\lambda \cdot (\mathbf{L}\ \mathsf{in}) = \Phi\ (\!|\Phi|\!)_\lambda$$

the **reflexion** rule,

$$\mathbf{L}\ \mathsf{in} = \Phi\ id \ \Leftrightarrow\ id = (\!|\Phi|\!)_\lambda$$

the **fusion** rule,

$$h \cdot (\!|\Phi|\!)_\lambda = (\!|\Psi|\!)_\lambda \ \Leftarrow\ h \cdot (\Phi\ f) = \Psi\ (h \cdot f)$$

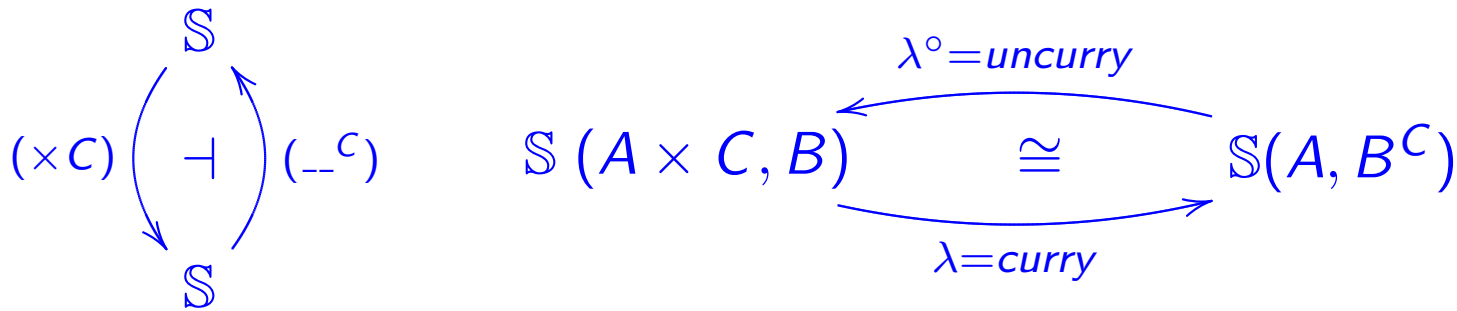
and others, which generalize what we had before (cf. $\mathbf{Id} \dashv \mathbf{Id}$, $\lambda = id$.)

Two examples

From

$$add \cdot \underbrace{\mathsf{in}_{\mathbb{N}_0} \times id}_{\mathbf{L}\ \mathsf{in}_{\mathbb{N}_0}} = \underbrace{[\pi_2, succ \cdot add] \cdot distl}_{\Phi\ add}$$

adjunction



grants unique solution add such that $\lambda\ add$ is function[1]

$$\begin{cases} \lambda\ add \cdot zero = \underline{id} \\ \lambda\ add \cdot succ = (succ \cdot) \cdot (\lambda\ add) \end{cases}$$

[1] Details in the annex.

More interesting example

By primitive recursion,

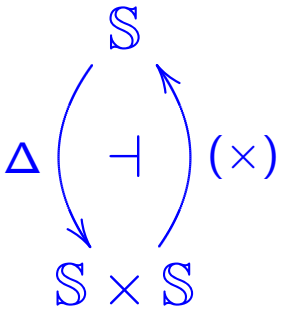
$$sq\ 0 = 0$$
$$sq\ (1 + x) = 2\ x + 1 + sq\ x$$

rewrites to

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

leading to **mutual recursion**. Can we calculate unique solutions to **mutually recursive** systems of equations?

Adjunction $\Delta \dashv (\times)$



$$(\mathbb{S} \times \mathbb{S})\ (\Delta\ A, (B, C)) \underset{\lambda\ (f,g)=\langle f,g\rangle}{\overset{\lambda^\circ\ f=(\pi_1 \cdot f, \pi_2 \cdot f)}{\cong}} \mathbb{S}\ (A, B \times C)$$

where $\Delta\ A = (A, A)$ and $\Delta\ f = (f, f)$ enables us to pair the two equations,

$$(sq, odd) \cdot (\Delta\ \mathsf{in}_{\mathbb{N}_0}) = \underbrace{([zero, add \cdot \langle sq, odd\rangle], [one, (2+) \cdot odd])}_{\Phi\ (sq,odd)}$$

and therefore, naming $sqodd = \lambda\ (sq, odd)$

$$sqodd \cdot \mathsf{in}_{\mathbb{N}_0} = \langle\!|\underbrace{\lambda \cdot \Phi \cdot \lambda^\circ}_{\Psi}|\!\rangle$$

Then (next slide):

Adjunction $\Delta \dashv (\times)$

$$sqodd \cdot \mathsf{in}_{\mathbb{N}_0} = (\Psi\ id) \cdot (id + \langle sq, odd \rangle)$$

We just have to calculate $\Psi\ id$:

$$\begin{aligned}
& \Psi\ id \\
= & \quad \{\ \} \\
& \lambda\ (\Phi\ (\lambda^\circ\ id)) \\
= & \quad \{\ \lambda^\circ\ f = (\pi_1 \cdot f, \pi_2 \cdot f)\ \} \\
& \lambda\ (\Phi\ (\pi_1, \pi_2)) \\
= & \quad \{\ \text{definition of } \Phi\ \} \\
& \lambda\ ([zero, add \cdot \langle \pi_1, \pi_2 \rangle], [one, (2+) \cdot \pi_2]) \\
= & \quad \{\ \lambda\ (f, g) = \langle f, g \rangle\ \} \\
& \langle [zero, add], [one, (2+) \cdot \pi_2] \rangle
\end{aligned}$$

Adjunction $\Delta \dashv (\times)$

This leads to the **adjoint solution**

$$sqodd\ 0 = (0, 1)$$
$$sqodd\ (1 + n) = (s + o, 2 + o)\ \textbf{where}\ (s, o) = \langle sq, odd \rangle\ n$$

which can also be written (functionally) as

$$sqodd = \textbf{for}\ loop\ (0, 1)\ \textbf{where}\ loop\ (s, o) = (s + o, 2 + o)$$

interestingly very close to the same program written in C:

```
int sqodd (int a) {
int s = 0; int o = 1; int j;
for (j = 1; j < a + 1; j++) { s += o; o += 2; }
return s;
};
```

More adjunctions

For a comprehensive account and many more examples see the main reference in the field:

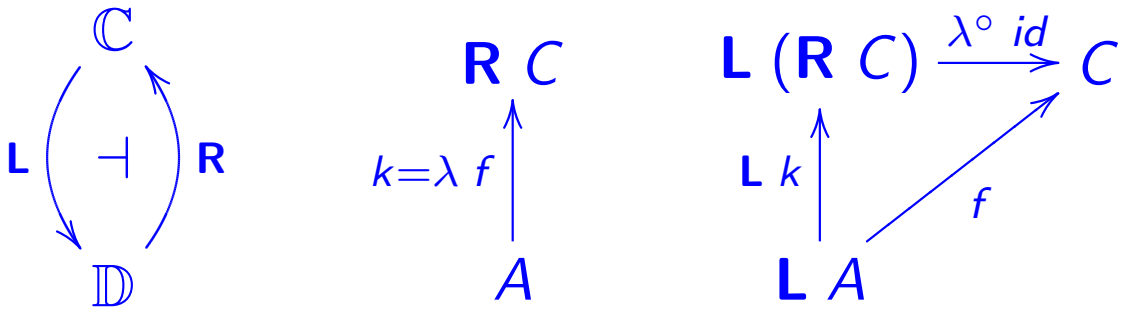
> *Ralf Hinze. Adjoint folds and unfolds-an extended study. Science of Computer Programming, 78(11):2108–2159, 2013. ISSN 0167-6423. URL* `http://www.sciencedirect.com/science/article/pii/S0167642312001396`.

This also covers the dual case — **final algebras** and corresponding morphisms ('unfolds').

Monads

Note the **FP**-pragmatic view of an **adjunction**: a too complex **input** gets simpler by "complicating" the **output**.

Also recall that adjunctions



define **monads** in a natural way — $\mathbf{M} = \mathbf{R} \cdot \mathbf{L}$ is a monad:

$$\mathbf{M}\ (\mathbf{M}\ X) \xrightarrow{\mu = \mathbf{R}\ (\lambda^\circ\ id)} \mathbf{M}\ X \xleftarrow{\eta = \lambda\ id} X$$

Monads

Monads are **very** important in functional programming — they help in handling **too complex** outputs, e.g. $\mathbf{M}\ A \xleftarrow{f} \mu_{\mathbf{F}}$ for some **monad** $\mathbf{M}$ — as we have identified above.

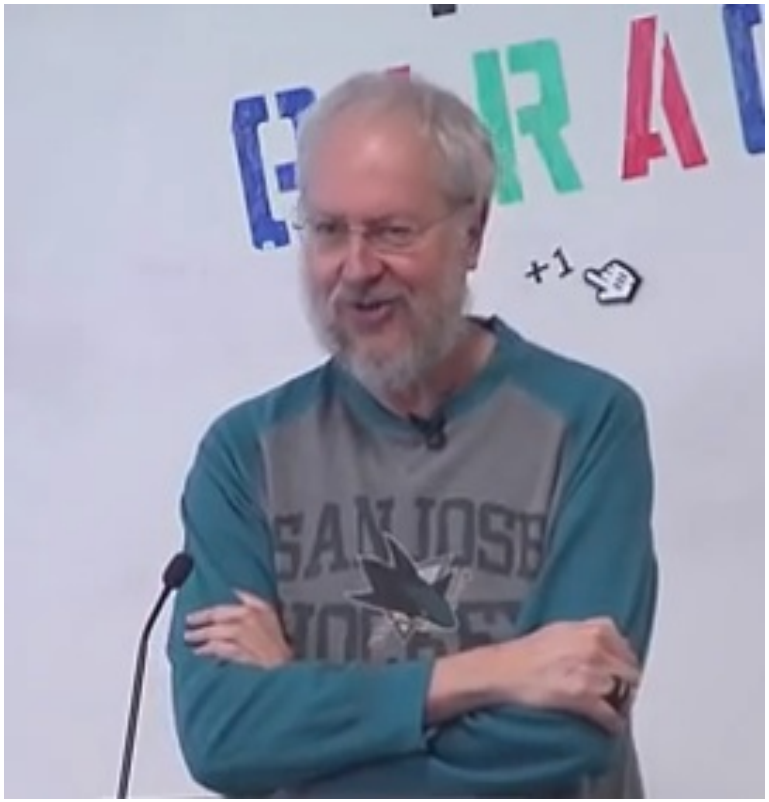
Examples are $\mathbf{M}\ X = \mathcal{P}\ X$ (non-deterministic programs), $\mathbf{M}\ X = \mathbf{D}\ X$ (distributions with finite support in probabilistic programs), $\mathbf{M}\ X = (X \times C)^C$ (state monad arising from $(\times C) \dashv (\_\_^C)$, for programs with internal state), etc etc

However, the concept still bewilders the programming community (next slide).

The monadic "curse" :-)

"Monads [...] come with a curse. The monadic curse is that once someone learns what monads are and how to use them, they lose the ability to explain it to other people"

(Douglas Crockford: Google Tech Talk on how to express monads in JavaScript, 2013)



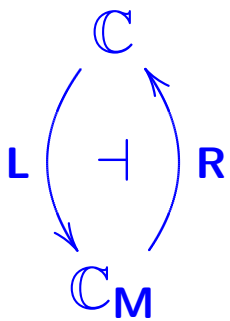
Douglas Crockford (2013)

Kleisli categories

It has become practical to reason about **monadic** programs not in the original category $\mathbb{C}$ but rather in the associated **Kleisli** category $\mathbb{C}_{\mathbf{M}}$ arising from adjunction

$$\left\{\begin{array}{l}\mathbf{L}\ X = X\\ \mathbf{L}\ f = \eta \cdot f\end{array}\right. \dashv \left\{\begin{array}{l}\mathbf{R}\ X = \mathbf{M}\ X\\ \mathbf{R}\ f = \mu \cdot \mathbf{M}\ f\end{array}\right.$$

where $\mathbf{M}\ (\mathbf{M}\ X) \xrightarrow{\mu} \mathbf{M}\ X \xleftarrow{\eta} X$ is the monad of interest:



$$\mathbb{C}_{\mathbf{M}}\ (A, B) \underset{\lambda}{\overset{\lambda^\circ}{\cong}} \mathbb{C}\ (A, \mathbf{M}\ B)$$

By the way —"folk" **FP** notation for $\mathbf{R}\ f$ is (in Haskell syntax):

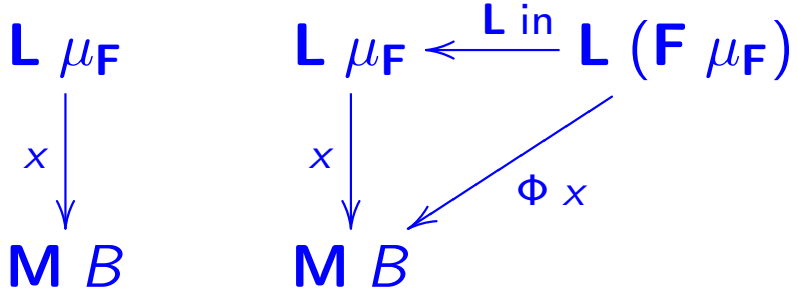
$$\mathbf{R}\ f\ x \quad = \quad \mathbf{do}\ \{a \leftarrow x; f\ a\}$$

Kleisli categories

(Enriched) Kleisli categories offer powerful frameworks for reasoning about **FP**s, namely:

- Category of **binary relations** — cf. **powerset** monad — homsets = Boolean algebras
- Category of **stochastic matrices** — cf. **distribution** monad — cf. (typed) linear algebra.

Currently studying calculational properties offered by Kleisli categories concerning our starting point, but now extended with a monad on the output:



Epilogue

FP relies on a few ingredients which put the paradigm at the forefront of program development:

- **higher order** functions (i.e. exponentials)
- **polymorphic** functions (i.e. natural transformations)
- **parametric** types, that is, functors
- **effect-full** types, that is, monads.

Its categorial basis makes possible — and this is rare in the software sciences — an **Algebra of Programming**.

Epilogue

Galois connections first and **adjunctions** now are improving our understanding of the theory behind **FP**.

What looked different in the past is being unified in a beautiful piece of **engineering mathematics**.

Can these mathematics be scaled up to large-scale software production?

A long way to go, still...

Annex

Running example

Solving

$$add \cdot \underbrace{\mathsf{in}_{\mathbb{N}_0} \times id}_{\mathbf{L}\ \mathsf{in}_{\mathbb{N}_0}} = \underbrace{[\pi_2, succ \cdot add] \cdot distl}_{\Phi\ add}$$

for add:

$$\begin{array}{ll} & (curry\ add) \cdot \mathsf{in}_{\mathbb{N}_0} = curry\ ([\pi_2, succ \cdot add] \cdot distl) \\ \Leftrightarrow & \quad \{\ \text{exponencials: } ex\ f\ g = f \cdot g\ \} \\ & (curry\ add) \cdot \mathsf{in}_{\mathbb{N}_0} = ex\ [\pi_2, succ \cdot add] \cdot (curry\ distl) \\ \Leftrightarrow & \quad \{\ curry\ distl = [curry\ i_1, curry\ i_2]\ \} \\ & \begin{cases} curry\ add \cdot zero = ex\ [\pi_2, succ \cdot add] \cdot (curry\ i_1) \\ curry\ add \cdot succ = ex\ [\pi_2, succ \cdot add] \cdot (curry\ i_2) \end{cases} \end{array}$$

Running example

$$
\begin{array}{ll}
\Leftrightarrow & \{ \text{ coproducts } \} \\
& \begin{cases} curry\ add \cdot zero = curry\ \pi_2 \\ curry\ add \cdot succ = curry\ (succ \cdot add) \end{cases} \\
\Leftrightarrow & \{ \text{ exponentials } \} \\
& \begin{cases} curry\ add \cdot zero = \underline{id} \\ curry\ add \cdot succ = (succ \cdot) \cdot (curry\ add) \end{cases} \\
\Leftrightarrow & \{ \ \} \\
& curry\ add \cdot \mathsf{in}_{\mathbb{N}_0} = [\underline{id}, (succ \cdot) \cdot (curry\ add)] \\
\Leftrightarrow & \{ \ \} \\
& curry\ add = \langle\!| [\underline{id}, (succ \cdot) \cdot (\_)] |\!\rangle
\end{array}
$$

(Higher order solution.)

References

Ralf Hinze. Adjoint folds and unfolds-an extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2012.07.011. URL `http://www.sciencedirect.com/science/article/pii/S0167642312001396`.

J.N. Oliveira. Towards a linear algebra of programming. *Formal Aspects of Computing*, 24(4-6):433–458, 2012. doi: 10.1007/s00165-012-0240-9.