Categories for the Working Programmer

Jeremy Gibbons, University of Oxford
Build Stuff, November 2015
1. Motivation

“A monad is just a monoid in the category of endofunctors; what’s the problem?”
1. Motivation

“A monad is just a monoid in the category of endofunctors; what’s the problem?”

I’ll try to show how

category theory inspires better code.

But you don’t really need the category theory: it all makes sense as functional programs* too.
1. Motivation

“A monad is just a monoid in the category of endofunctors; what’s the problem?”

I’ll try to show how

*category theory*

*inspires better code.*

But you don’t really need the category theory: it all makes sense as functional programs* too.

____________________

*Haskell*
2. Functions that consume lists

Two equations, indirectly defining \( \text{sum} \):

\[
\text{sum} :: [\text{Integer}] \rightarrow \text{Integer}
\]

\[
\text{sum} [ ] = 0
\]

\[
\text{sum} (x:xs) = x + \text{sum} \ xs
\]
2. Functions that consume lists

Two equations, indirectly defining \( \text{sum} \):

\[
\text{sum} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum} [ ] = 0 \\
\text{sum} (x : xs) = x + \text{sum} \ xs
\]

Not just \( + \). For \textit{any} given \( f \) and \( e \), these equations uniquely determine \( h \):

\[
h [ ] = e \\
h (x : xs) = f \ x \ (h \ xs)
\]
2. Functions that consume lists

Two equations, indirectly defining \( \text{sum} \):

\[
\text{sum} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum} \; [\;] = 0 \\
\text{sum} \; (x : xs) = x + \text{sum} \; xs
\]

Not just \(+\). For \( \text{any} \) given \( f \) and \( e \), these equations uniquely determine \( h \):

\[
h \; [\;] = e \\
h \; (x : xs) = f \; x \; (h \; xs)
\]

The unique solution is called \( \text{foldr} \; f \; e \) in the Haskell libraries:

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr} \; f \; e \; [\;] = e \\
\text{foldr} \; f \; e \; (x : xs) = f \; x \; (\text{foldr} \; f \; e \; xs)
\]
3. Some applications of foldr

\[\text{sum} \; = \; \text{foldr} \; (+) \; 0\]
\[\text{and} \; = \; \text{foldr} \; (\wedge) \; \text{True}\]
\[\text{decimal} \; = \; \text{foldr} \; (\lambda x \rightarrow (\text{fromInteger} \; d \; + \; x) \; / \; 10) \; 0\]
\[\text{id} \; = \; \text{foldr} \; (:) \; []\]
\[\text{length} \; = \; \text{foldr} \; (\lambda n \rightarrow 1 \; + \; n) \; 0\]
\[\text{map} \; f \; = \; \text{foldr} \; ((:) \; \circ \; f) \; []\]
\[\text{filter} \; p \; = \; \text{foldr} \; (\lambda xs \rightarrow \text{if} \; p \; x \; \text{then} \; x : \; xs \; \text{else} \; xs) \; []\]
\[\text{concat} \; = \; \text{foldr} \; (+) \; []\]
\[\text{reverse} \; = \; \text{foldr} \; \text{snoc} \; []\; \text{where} \; \text{snoc} \; x \; xs \; = \; xs \; + \; [x] \quad \text{-- quadratic}\]
\[\text{xs} \; + \; \text{ys} \; = \; \text{foldr} \; (:) \; \text{ys} \; \text{xs}\]
\[\text{inits} \; = \; \text{foldr} \; (\lambda xss \rightarrow [ ] : \; \text{map} \; (:) \; xss) \; []\; []\]
\[\text{tails} \; = \; \text{foldr} \; (\lambda xss \rightarrow (x : \; \text{head} \; xss) : \; xss) \; []\; []\]

etc etc
4. What’s special about lists?

…only the special syntax. We might have defined lists ourselves:

```haskell
data List a = Nil | Cons a (List a)
```

Then we could have

```haskell
foldList :: (a → b → b) → b → List a → b
foldList f e Nil = e
foldList f e (Cons x xs) = f x (foldList f e xs)
```
4. What’s special about lists?

...only the special syntax. We might have defined lists ourselves:

```haskell
data List a = Nil | Cons a (List a)
```

Then we could have

```haskell
foldList :: (a → b → b) → b → List a → b
foldList f e Nil = e
foldList f e (Cons x xs) = f x (foldList f e xs)
```

Similarly,

```haskell
data Tree a = Tip a | Bin (Tree a) (Tree a)
foldTree :: (a → b) → (b → b → b) → Tree a → b
foldTree f g (Tip x) = f x
foldTree f g (Bin xs ys) = g (foldTree f g xs) (foldTree f g ys)
```
5. It’s not always so obvious

Rose trees (eg for games, or XML):

\[
\textbf{data} \ Rose \ a = \ Node \ a \ [ \ Rose \ a ]
\]
5. It’s not always so obvious

Rose trees (eg for games, or XML):

```
data Rose a = Node a [ Rose a ]
foldRose1 :: (a → c → b) → (b → c → c) → c → Rose a → b
foldRose1 f g e (Node x ts) = f x (foldr g e (map (foldRose1 f g e) ts))
```
5. It’s not always so obvious

Rose trees (eg for games, or XML):

\[
\textbf{data} \ Rose \ a = \text{Node} \ a \ [ \ Rose \ a ]
\]

\[
foldRose_1 :: (a \to c \to b) \to (b \to c \to c) \to c \to Rose \ a \to b
\]

\[
foldRose_1 \ f \ g \ e \ (\text{Node} \ x \ ts) = f \ x \ (\text{foldr} \ g \ e \ (\text{map} \ (foldRose_1 \ f \ g \ e) \ ts))
\]

\[
foldRose_2 :: (a \to b \to b) \to ([b] \to b) \to Rose \ a \to b
\]

\[
foldRose_2 \ f \ g \ (\text{Node} \ x \ ts) = f \ x \ (g \ (\text{map} \ (foldRose_2 \ f \ g) \ ts))
\]
5. It’s not always so obvious

Rose trees (eg for games, or XML):

**data** Rose a = Node a [ Rose a ]

foldRose₁ :: (a → c → b) → (b → c → c) → c → Rose a → b
foldRose₁ f g e (Node x ts) = f x (foldr g e (map (foldRose₁ f g e) ts))

foldRose₂ :: (a → b → b) → ([b] → b) → Rose a → b
foldRose₂ f g (Node x ts) = f x (g (map (foldRose₂ f g) ts))

foldRose₃ :: (a → [b] → b) → Rose a → b
foldRose₃ f (Node x ts) = f x (map (foldRose₃ f) ts)

Which should we choose?
5. It’s not always so obvious

Rose trees (eg for games, or XML):

```haskell
data Rose a = Node a [ Rose a ]

foldRose1 :: (a → c → b) → (b → c → c) → c → Rose a → b
foldRose1 f g e (Node x ts) = f x (foldr g e (map (foldRose1 f g e) ts))

foldRose2 :: (a → b → b) → ([ b ] → b) → Rose a → b
foldRose2 f g (Node x ts) = f x (g (map (foldRose2 f g) ts))

foldRose3 :: (a → [ b ] → b) → Rose a → b
foldRose3 f (Node x ts) = f x (map (foldRose3 f) ts)
```

Which should we choose?

Haskell libraries get folds for non-empty lists ‘wrong’!

```haskell
foldr1, foldl1 :: (a → a → a) → [ a ] → a
```
6. Preparing for genericity

Separate out list-specific ‘shape’ from type recursion:

```haskell
data ListS a b = NilS | ConsS a b
data Fix s a = In (s a (Fix s a))
type List a = Fix ListS a
```

For example, list \([1, 2, 3]\) is represented by

```haskell
In (ConsS 1 (In (ConsS 2 (In (ConsS 3 (In NilS))))))
```

For convenience, define inverse `out` to `In`:

```haskell
out :: Fix s a \to s a \,(Fix s a)
out (In \,x) = x
```
6. Preparing for genericity

Separate out list-specific ‘shape’ from type recursion:

```haskell
data ListS a b = NilS | ConsS a b

data Fix s a = In { out :: s a (Fix s a) }  -- In and out together

type List a = Fix ListS a
```

Shape is mostly opaque; just need to ‘locate’ the as and bs:

```haskell
bimap :: (a → a') → (b → b') → ListS a b → ListS a' b'

bimap f g NilS = NilS

bimap f g (ConsS a b) = ConsS (f a) (g b)
```
6. Preparing for genericity

Separate out list-specific ‘shape’ from type recursion:

```
data ListS a b = NilS | ConsS a b

data Fix s a = In { out :: s a (Fix s a) }  -- In and out together

type List a = Fix ListS a

bimap :: (a → a') → (b → b') → ListS a b → ListS a' b'
```

Now we can define a more cleanly separated version of `foldr` on `List`:

```
foldList :: (ListS a b → b) → List a → b
foldList f = f ∘ bimap id (foldList f) ∘ out
```

eg `foldList add :: List Integer → Integer`, where

```
add :: ListS Integer Integer → Integer
add NilS = 0
add (ConsS m n) = m + n
```
7. Going datatype-generic

Now we can properly abstract away the list-specific details. To be suitable, a shape must support \textit{bimap}:

\begin{verbatim}
class Bifunctor s where
  bimap :: (a → a') → (b → b') → s a b → s a' b'
\end{verbatim}

Then \textit{fold} works for any suitable shape:

\begin{verbatim}
fold :: Bifunctor s ⇒ (s a b → b) → Fix s a → b
fold f = f ◦ bimap id (fold f) ◦ out
\end{verbatim}

Of course, \textit{ListS} is a suitable shape…

\begin{verbatim}
instance Bifunctor ListS where
  bimap f g NilS = NilS
  bimap f g (ConsS a b) = ConsS (f a) (g b)
\end{verbatim}
7. Going datatype-generic

Now we can properly abstract away the list-specific details. To be suitable, a shape must support \textit{bimap}:

\begin{verbatim}
    class Bifunctor s where
        bimap :: (a -> a') -> (b -> b') -> s a b -> s a' b'
\end{verbatim}

Then \textit{fold} works for any suitable shape:

\begin{verbatim}
    fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
    fold f = f \circ \text{bimap id (fold f) \circ out}
\end{verbatim}

...but binary trees are also suitable:

\begin{verbatim}
    data TreeS a b = TipS a | BinS b b

    instance Bifunctor TreeS where
        bimap f g (TipS a) = TipS (f a)
        bimap f g (BinS b1 b2) = BinS (g b1) (g b2)
\end{verbatim}
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately.
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately. An *algebra* for functor $S A$ is a pair $(B, f)$ where $f :: S A B \to B$. 
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately.

An algebra for functor $S A$ is a pair $(B, f)$ where $f :: S A B \to B$.
A homomorphism between $(B, f)$ and $(C, g)$ is a function $h :: B \to C$ such that

$$h \circ f = g \circ \text{bimap id } h$$
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately.

An \textit{algebra} for functor $S \ A$ is a pair $(B, f)$ where $f :: S \ A \ B \to B$.

A \textit{homomorphism} between $(B, f)$ and $(C, g)$ is a function $h :: B \to C$ such that

$$h \circ f = g \circ \text{bimap} \ \text{id} \ h$$

Algebra $(B, f)$ is \textit{initial} if there is a unique homomorphism to each $(C, g)$. 
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately.

An **algebra** for functor $S A$ is a pair $(B, f)$ where $f :: S A B \to B$.

A **homomorphism** between $(B, f)$ and $(C, g)$ is a function $h :: B \to C$ such that

$$h \circ f = g \circ \text{bimap id } h$$

Algebra $(B, f)$ is **initial** if there is a unique homomorphism to each $(C, g)$.

Eg $(\text{List Integer, In})$ and $(\text{Integer, add})$ are both algebras for $\text{ListS Integer}$:

$$\text{In} :: \text{ListS Integer } (\text{List Integer}) \to \text{List Integer}$$

$$\text{add} :: \text{ListS Integer Integer} \to \text{Integer}$$

and $\text{sum} :: \text{List Integer} \to \text{Integer}$ is a homomorphism. The initial algebra is $(\text{List Integer, In})$, and the unique homomorphism to $(C, g)$ is $\text{fold } g$. 
8. The categorical view, in a nutshell

Think of a bifunctor, $S$. It is also a functor in each argument separately.

An *algebra* for functor $S A$ is a pair $(B, f)$ where $f :: S A B \to B$.

A *homomorphism* between $(B, f)$ and $(C, g)$ is a function $h :: B \to C$ such that

$$h \circ f = g \circ \text{bimap id } h$$

Algebra $(B, f)$ is *initial* if there is a unique homomorphism to each $(C, g)$.

Eg $(\text{List Integer}, \text{In})$ and $(\text{Integer}, \text{add})$ are both algebras for $\text{ListS Integer}$:

$$\text{In} :: \text{ListS Integer} (\text{List Integer}) \to \text{List Integer}$$

$$\text{add} :: \text{ListS Integer Integer} \to \text{Integer}$$

and $\text{sum} :: \text{List Integer} \to \text{Integer}$ is a homomorphism. The initial algebra is $(\text{List Integer}, \text{In})$, and the unique homomorphism to $(C, g)$ is $\text{fold } g$.

**Theorem:** for all sensible shape functors $S$, initial algebras exist.
9. Duality

Recall

\[ \text{fold} :: \text{Bifunctor } s \Rightarrow (s \ a \ b \rightarrow b) \rightarrow (\text{Fix } s \ a \rightarrow b) \]
\[ \text{fold } f = f \circ \text{bimap } \text{id} \ (\text{fold } f) \circ \text{out} \]
9. Duality

Recall

\[\text{fold} :: \text{Bifunctor } s \Rightarrow (s \ a \ b \to b) \to (\text{Fix } s \ a \to b)\]

\[\text{fold } f = f \circ \text{bimap id (fold } f) \circ \text{out}\]

Reverse certain arrows:

\[\text{unfold} :: \text{Bifunctor } s \Rightarrow (b \to s \ a \ b) \to (b \to \text{Fix } s \ a)\]

\[\text{unfold } f = \text{In} \circ \text{bimap id (unfold } f) \circ f\]
9. Duality

Recall

\[
\text{fold} :: \text{Bifunctor } s \Rightarrow (s \ a \ b \to b) \to (\text{Fix } s \ a \to b)
\]
\[
\text{fold } f = f \circ \text{bimap id } (\text{fold } f) \circ \text{out}
\]

Reverse certain arrows:

\[
\text{unfold} :: \text{Bifunctor } s \Rightarrow (b \to s \ a \ b) \to (b \to \text{Fix } s \ a)
\]
\[
\text{unfold } f = \text{In} \circ \text{bimap id } (\text{unfold } f) \circ f
\]

The datatype-generic presentation makes the duality very clear—unlike with

\[
\text{unfoldr} :: (b \to \text{Maybe } (a, b)) \to b \to [ a]
\]
\[
\text{unfoldr } f \ b = \text{case } f \ b \text{ of }
\]
\[
\text{Nothing } \to [ ]
\]
\[
\text{Just } (a, b') \to a : \text{unfoldr } f \ b'
\]
9. Duality

Recall

\[
\text{fold} :: \text{Bifunctor } s \Rightarrow (s \ a \ b \to b) \to (\text{Fix } s \ a \to b)
\]

\[
\text{fold } f = f \circ \text{bimap id (fold } f \circ \text{out)
}\]

Reverse certain arrows:

\[
\text{unfold} :: \text{Bifunctor } s \Rightarrow (b \to s \ a \ b) \to (b \to \text{Fix } s \ a)
\]

\[
\text{unfold } f = \text{In } \circ \text{bimap id (unfold } f \circ f\)
\]

The datatype-generic presentation makes the duality very clear—unlike with

\[
\text{unfoldr} :: (b \to \text{Maybe (a, b)}) \to b \to [ a]
\]

\[
\text{unfoldr } f \ b = \text{case } f \ b \text{ of}
\]

\[
\text{Nothing} \to [ ]
\]

\[
\text{Just } (a, b') \to a : \text{unfoldr } f \ b'
\]

Categorically, \textit{coalgebras} \((B, f)\) with \(f :: B \to S \ A \ B\), \textit{finality}.
10. Conclusions

- category theory as an organisational tool, not for intimidation
- helping you to write better code, with *less mess*
- the mathematics is really quite pretty
- …but the Haskell makes sense on its own too
10. Conclusions

- category theory as an organisational tool, not for intimidation
- helping you to write better code, with *less mess*
- the mathematics is really quite pretty
- …but the Haskell makes sense on its own too

http://patternsinfp.wordpress.com/
http://www.cs.ox.ac.uk/jeremy.gibbons/
11. BTW, Software Engineering Programme

MSc in Software Engineering
(part-time)

flexible, professional education

MSc in Software and Systems Security

flexible, part-time, professional education
Appendix: category theory
12. ‘Category’

A category consists of

- a collection of *objects*
- for each pair $A, B$ of objects, a collection $A \rightarrow B$ of *arrows*
- an *identity* arrow $id_A : A \rightarrow A$ for each object $A$
- *composition* $f \circ g : A \rightarrow C$ of compatible arrows $f : B \rightarrow C$ and $g : A \rightarrow B$
- composition is *associative*, and identities are *neutral* elements

(think of *paths* in labelled directed graphs)
A category consists of

- a collection of *objects* (sets)
- for each pair $A, B$ of objects, a collection $A \to B$ of *arrows* (functions)
- an *identity* arrow $id_A : A \to A$ for each object $A$
- *composition* $f \circ g : A \to C$ of compatible arrows $f : B \to C$ and $g : A \to B$
- composition is *associative*, and identities are *neutral* elements
13. ‘Functor’

A functor $F$ is simultaneously

- an operation on objects
- an operation on arrows

such that

- $F f : F A \to F B$ when $f : A \to B$
- $F id = id$
- $F (f \circ g) = F f \circ F g$
13. ‘Functor’

Functor \textit{List} is simultaneously

- an operation on objects \((List \ A = [A])\)
- an operation on arrows \((List \ f = \text{map } f)\)

such that

- \(List \ f : List \ A \to List \ B\) when \(f : A \to B\)
- \(List \ id = id\)
- \(List \ (f \circ g) = List \ f \circ List \ g\)
13. ‘Functor’

Functor $ListS A$ is simultaneously

- an operation on objects ($(ListS A) B = ListS A B$)
- an operation on arrows ($(ListS A) f = bimap id f$)

such that

- $(ListS A) f : ListS A B \to ListS A B'$ when $f : B \to B'$
- $(ListS A) id = id$
- $(ListS A) (f \circ g) = (ListS A) f \circ (ListS A) g$
14. ‘Algebra’

An algebra for functor $F$ is a pair $(A, f)$ with $f : F A \to A$.

For example, $(\text{Integer}, \text{sum})$ is a $\text{List}$-algebra.

More pertinently, $(\text{Integer}, \text{add})$ is a $(\text{ListS Integer})$-algebra.

\[ \text{add} :: \text{ListS Integer Integer} \to \text{Integer} \]

So is $(\text{List Integer}, \text{In})$:

\[ \text{In} :: \text{ListS Integer (List Integer)} \to \text{List Integer} \]
15. ‘Homomorphism’

For functor $F$, a *homomorphism* $h$ between $F$-algebras $(A, f)$ and $(B, g)$ is an arrow $h: A \to B$ such that

$$h \circ f = g \circ F h$$
15. ‘Homomorphism’

For functor $F$, a homomorphism $h$ between $F$-algebras $(A, f)$ and $(B, g)$ is an arrow $h: A \to B$ such that

$$h \circ f = g \circ F h$$
15. ‘Homomorphism’

For functor $F$, a homomorphism $h$ between $F$-algebras $(A, f)$ and $(B, g)$ is an arrow $h : A \to B$ such that

$$h \circ f = g \circ F h$$

For example, $\text{sum} : \text{List Integer} \to \text{Integer}$ is a homomorphism from $(\text{List Integer}, \text{In})$ to $(\text{Integer}, \text{add})$: $\text{sum} \circ \text{In} = \text{add} \circ \text{bimap id sum}$
15. ‘Homomorphism’

For functor $F$, a *homomorphism* $h$ between $F$-algebras $(A, f)$ and $(B, g)$ is an arrow $h : A \rightarrow B$ such that

\[ h \circ f = g \circ F h \]

For example, $\text{sum} : \text{List Integer} \rightarrow \text{Integer}$ is a homomorphism from $(\text{List Integer}, \text{In})$ to $(\text{Integer}, \text{add})$:

\[ \text{sum} \circ \text{In} = \text{add} \circ \text{bimap} \text{id} \text{sum} \]

(Identity function is a homomorphism, and homomorphisms compose. So $F$-algebras and their homomorphisms also form a category.)
16. ‘Initial’

An $F$-algebra $(A, f)$ is *initial* if, for each other $F$-algebra $(B, g)$, there is a unique homomorphism from $(A, f)$ to $(B, g)$.
16. ‘Initial’

An $F$-algebra $(A, f)$ is *initial* if, for each other $F$-algebra $(B, g)$, there is a unique homomorphism from $(A, f)$ to $(B, g)$.

**Theorem:** $(\text{List Integer}, \text{In})$ is the initial $(\text{ListS Integer})$-algebra.

The homomorphisms are precisely the folds, and uniqueness is the *universal property*. 
16. ‘Initial’

An $F$-algebra $(A, f)$ is initial if, for each other $F$-algebra $(B, g)$, there is a unique homomorphism from $(A, f)$ to $(B, g)$.

**Theorem:** $(\text{List Integer}, \text{In})$ is the initial $(\text{ListS Integer})$-algebra.

The homomorphisms are precisely the folds, and uniqueness is the universal property.

**Theorem:** For any polynomial* shape functor $F$, there is an initial $F$-algebra.

Datatype-generically, too.

*(polynomial*:
constructed from sums and products,
like simple algebraic datatypes)
16. ‘Initial’

An $F$-algebra $(A, f)$ is *initial* if, for each other $F$-algebra $(B, g)$, there is a unique homomorphism from $(A, f)$ to $(B, g)$.

**Theorem:** $(\text{List Integer, In})$ is the initial $(\text{ListS Integer})$-algebra.

The homomorphisms are precisely the folds, and uniqueness is the *universal property*.

**Theorem:** For any polynomial* shape functor $F$, there is an initial $F$-algebra.

Datatype-generically, too.

(More generally, an *initial object* in a category is one with a unique arrow to every other object. In *SET*, the initial object is $\emptyset$, and ‘initial $F$-algebra’ is short for ‘initial object in the category of $F$-algebras’.)
17. Morally correct

- those two theorems hold in $SET$, but not some other settings
- not quite true for realistic Haskell
  - *undefined* values, *infinite* data structures, *strictness*...
- defining equations do not always uniquely define $foldr$—consider

\[
\begin{align*}
h ([ ] ) &= 3 \\
h (x : xs) &= const (const 3) \times (h \; xs)
\end{align*}
\]
17. Morally correct

- those two theorems hold in $SET$, but not some other settings
- not quite true for realistic Haskell
  
  *undefined* values, *infinite* data structures, *strictness* . . .

- defining equations do not always uniquely define $foldr$—consider

  \[
  h[\ ] = 3 \\
  h(x:xs) = \text{const}(\text{const }3) \times (h\ xs)
  \]

- (in $CPO$, some strictness side-conditions needed)
- (all works fine in *strong functional programming*, eg Agda)