Categories for the Working Haskeller

Jeremy Gibbons, University of Oxford

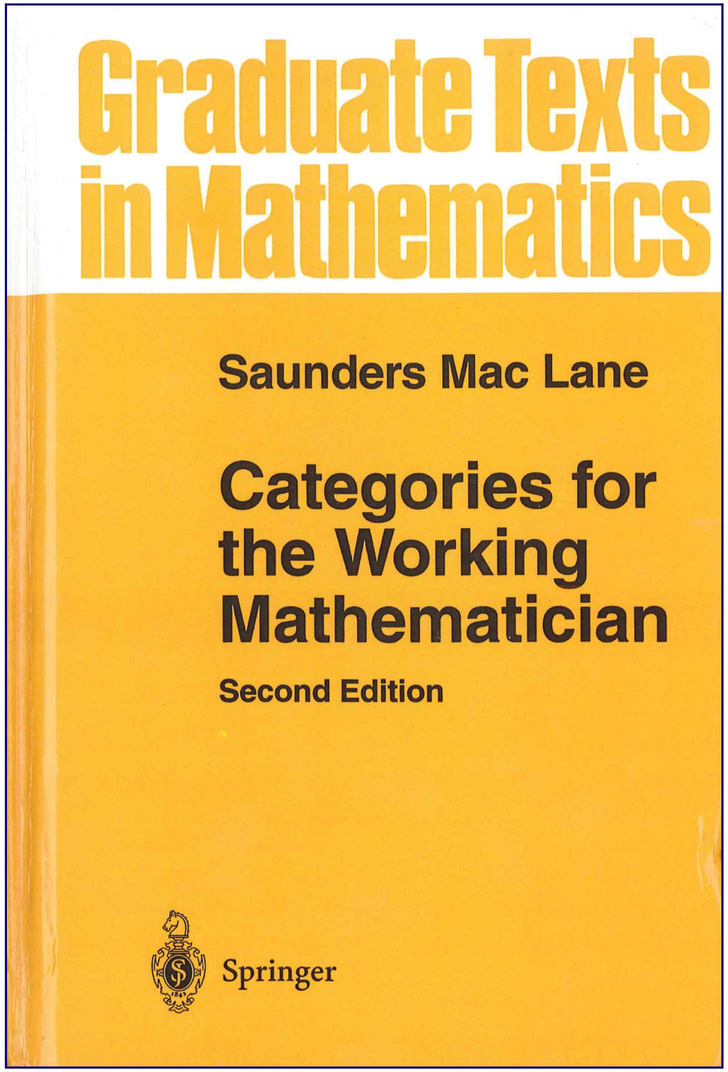
Haskell eXchange, October 2014

1. Motivation

"What part of

> *monads are just monoids in*
> *the category of endofunctors*

don't you understand?"



1. Motivation

"What part of

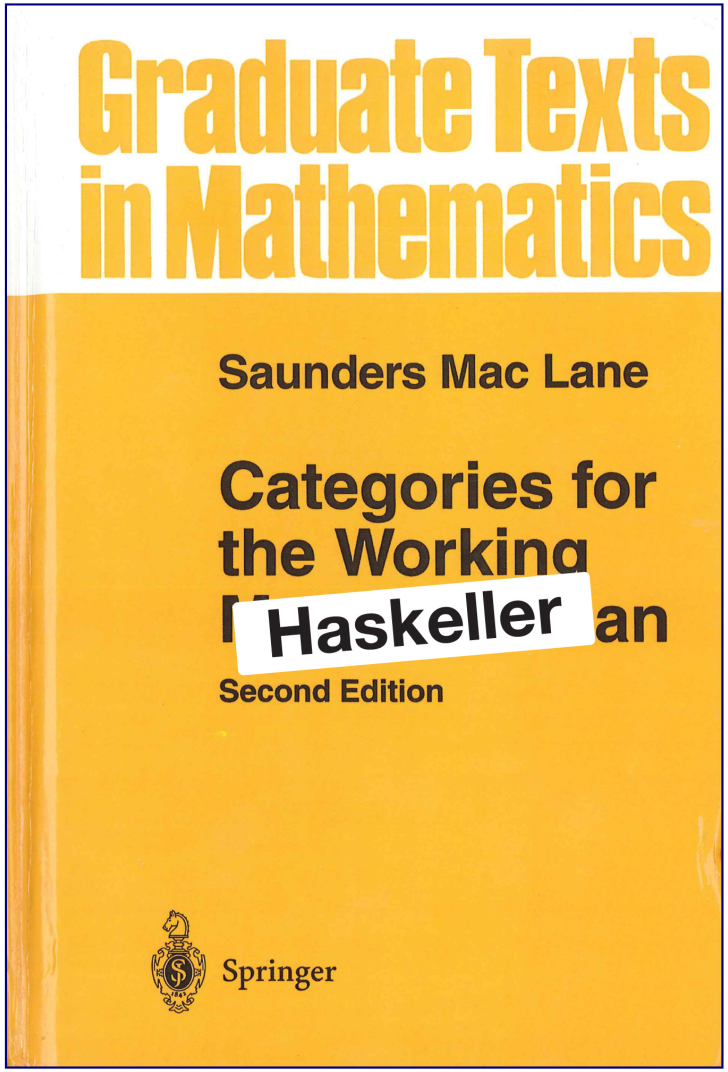
> *monads are just monoids in*
> *the category of endofunctors*

don't you understand?"

I'll try to show how

> *category theory*
> *inspires better code.*

But you don't really need the category theory: it all makes sense in Haskell too.



2. Functions that consume lists

Two equations, indirectly defining *sum*:

$$\begin{aligned} &sum :: [\,Integer\,] \to Integer \\ &sum\ [\,] \qquad\quad = 0 \\ &sum\ (x : xs) = x + sum\ xs \end{aligned}$$

2. Functions that consume lists

Two equations, indirectly defining *sum*:

$$sum :: [\,Integer\,] \to Integer$$
$$sum\ [\,] \quad\quad = 0$$
$$sum\ (x : xs) = x + sum\ xs$$

Not just $+$. For *any* given f and e, these equations uniquely determine h:

$$h\ [\,] \quad\quad = e$$
$$h\ (x : xs) = f\ x\ (h\ xs)$$

2. Functions that consume lists

Two equations, indirectly defining *sum*:

$$sum :: [\,Integer\,] \to Integer$$
$$sum\,[\,] \qquad = 0$$
$$sum\,(x : xs) = x + sum\ xs$$

Not just $+$. For *any* given f and e, these equations uniquely determine h:

$$h\,[\,] \qquad = e$$
$$h\,(x : xs) = f\ x\ (h\ xs)$$

The unique solution is called $foldr\ f\ e$ in the Haskell libraries:

$$foldr :: (a \to b \to b) \to b \to [\,a\,] \to b$$
$$foldr\ f\ e\,[\,] \qquad = e$$
$$foldr\ f\ e\,(x : xs) = f\ x\ (foldr\ f\ e\ xs)$$

3. Some applications of foldr

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

etc etc

4. What's special about lists?

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

$$\mathbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

Then we could have

$$\begin{aligned}
&foldList :: (a \to b \to b) \to b \to List\ a \to b\\
&foldList\ f\ e\ Nil \qquad\qquad\quad = e\\
&foldList\ f\ e\ (Cons\ x\ xs) = f\ x\ (foldList\ f\ e\ xs)
\end{aligned}$$

4. What's special about lists?

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

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

Then we could have

$$\begin{aligned}
&foldList :: (a \to b \to b) \to b \to List\ a \to b \\
&foldList\ f\ e\ Nil \qquad\qquad\quad = e \\
&foldList\ f\ e\ (Cons\ x\ xs) = f\ x\ (foldList\ f\ e\ xs)
\end{aligned}$$

Similarly,

$$\textbf{data}\ Tree\ a = Tip\ a \mid Bin\ (Tree\ a)\ (Tree\ a)$$

$$\begin{aligned}
&foldTree :: (a \to b) \to (b \to b \to b) \to Tree\ a \to b \\
&foldTree\ f\ g\ (Tip\ x) \qquad\ = f\ x \\
&foldTree\ f\ g\ (Bin\ xs\ ys) = g\ (foldTree\ f\ g\ xs)\ (foldTree\ f\ g\ ys)
\end{aligned}$$

5. It's not always so obvious

Rose trees (eg for games, or XML):

$$\textbf{data}\ \mathit{Rose}\ a = \mathit{Node}\ a\ [\mathit{Rose}\ a]$$

5. It's not always so obvious

Rose trees (eg for games, or XML):

$$\mathbf{data}\ Rose\ a = Node\ a\ [\,Rose\ a\,]$$

$$foldRose_1 :: (a \to c \to b) \to (b \to c \to c) \to c \to Rose\ a \to b$$
$$foldRose_1\ f\ g\ e\ (Node\ x\ ts) = f\ x\ (foldr\ g\ e\ (map\ (foldRose_1\ f\ g\ e)\ ts))$$

5. It's not always so obvious

Rose trees (eg for games, or XML):

$$\mathbf{data}\ \mathit{Rose}\ a = \mathit{Node}\ a\ [\,\mathit{Rose}\ a\,]$$

$$\mathit{foldRose}_1 :: (a \to c \to b) \to (b \to c \to c) \to c \to \mathit{Rose}\ a \to b$$
$$\mathit{foldRose}_1\ f\ g\ e\ (\mathit{Node}\ x\ ts) = f\ x\ (\mathit{foldr}\ g\ e\ (\mathit{map}\ (\mathit{foldRose}_1\ f\ g\ e)\ ts))$$

$$\mathit{foldRose}_2 :: (a \to b \to b) \to ([\,b\,] \to b) \to \mathit{Rose}\ a \to b$$
$$\mathit{foldRose}_2\ f\ g\ (\mathit{Node}\ x\ ts) = f\ x\ (g\ (\mathit{map}\ (\mathit{foldRose}_2\ f\ g)\ ts))$$

5. It's not always so obvious

Rose trees (eg for games, or XML):

$$\mathbf{data}\ Rose\ a = Node\ a\ [Rose\ a]$$

$$foldRose_1 :: (a \to c \to b) \to (b \to c \to c) \to c \to Rose\ a \to b$$
$$foldRose_1\ f\ g\ e\ (Node\ x\ ts) = f\ x\ (foldr\ g\ e\ (map\ (foldRose_1\ f\ g\ e)\ ts))$$

$$foldRose_2 :: (a \to b \to b) \to ([b] \to b) \to Rose\ a \to b$$
$$foldRose_2\ f\ g\ (Node\ x\ ts) = f\ x\ (g\ (map\ (foldRose_2\ f\ g)\ ts))$$

$$foldRose_3 :: (a \to [b] \to b) \to Rose\ a \to b$$
$$foldRose_3\ f\ (Node\ x\ ts) = f\ x\ (map\ (foldRose_3\ f)\ ts)$$

Which should we choose?

5. It's not always so obvious

Rose trees (eg for games, or XML):

$$\mathbf{data}\ \mathit{Rose}\ a = \mathit{Node}\ a\ [\mathit{Rose}\ a]$$

$$\begin{aligned}
&\mathit{foldRose}_1 :: (a \to c \to b) \to (b \to c \to c) \to c \to \mathit{Rose}\ a \to b \\
&\mathit{foldRose}_1\ f\ g\ e\ (\mathit{Node}\ x\ \mathit{ts}) = f\ x\ (\mathit{foldr}\ g\ e\ (\mathit{map}\ (\mathit{foldRose}_1\ f\ g\ e)\ \mathit{ts}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldRose}_2 :: (a \to b \to b) \to ([b] \to b) \to \mathit{Rose}\ a \to b \\
&\mathit{foldRose}_2\ f\ g\ (\mathit{Node}\ x\ \mathit{ts}) = f\ x\ (g\ (\mathit{map}\ (\mathit{foldRose}_2\ f\ g)\ \mathit{ts}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldRose}_3 :: (a \to [b] \to b) \to \mathit{Rose}\ a \to b \\
&\mathit{foldRose}_3\ f\ (\mathit{Node}\ x\ \mathit{ts}) = f\ x\ (\mathit{map}\ (\mathit{foldRose}_3\ f)\ \mathit{ts})
\end{aligned}$$

Which should we choose?

Haskell libraries get folds for non-empty lists 'wrong'!

$$\mathit{foldr}1, \mathit{foldl}1 :: (a \to a \to a) \to [a] \to a$$

6. Preparing for genericity

Separate out list-specific 'shape' from type recursion:

$$\textbf{data}\ \mathit{ListS}\ a\ b = \mathit{NilS} \mid \mathit{ConsS}\ a\ b$$
$$\textbf{data}\ \mathit{Fix}\ s\ a = \mathit{In}\ (s\ a\ (\mathit{Fix}\ s\ a))$$
$$\textbf{type}\ \mathit{List}\ a = \mathit{Fix}\ \mathit{ListS}\ a$$

For example, list $[1, 2, 3]$ is represented by

$$\mathit{In}\ (\mathit{ConsS}\ 1\ (\mathit{In}\ (\mathit{ConsS}\ 2\ (\mathit{In}\ (\mathit{ConsS}\ 3\ (\mathit{In}\ \mathit{NilS}))))))$$

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

$$\mathit{out} :: \mathit{Fix}\ s\ a \to s\ a\ (\mathit{Fix}\ s\ a)$$
$$\mathit{out}\ (\mathit{In}\ x) = x$$

6. Preparing for genericity

Separate out list-specific 'shape' from type recursion:

$$\mathbf{data}\ \mathit{ListS}\ a\ b = \mathit{NilS} \mid \mathit{ConsS}\ a\ b$$
$$\mathbf{data}\ \mathit{Fix}\ s\ a = \mathit{In}\ \{\, \mathit{out} :: s\ a\ (\mathit{Fix}\ s\ a)\,\} \quad \text{-- } \mathit{In} \text{ and } \mathit{out} \text{ together}$$
$$\mathbf{type}\ \mathit{List}\ a = \mathit{Fix}\ \mathit{ListS}\ a$$

Shape is mostly opaque; just need to 'locate' the *a*s and *b*s:

$$\mathit{bimap} :: (a \to a') \to (b \to b') \to \mathit{ListS}\ a\ b \to \mathit{ListS}\ a'\ b'$$
$$\mathit{bimap}\ f\ g\ \mathit{NilS} = \mathit{NilS}$$
$$\mathit{bimap}\ f\ g\ (\mathit{ConsS}\ a\ b) = \mathit{ConsS}\ (f\ a)\ (g\ b)$$

6. Preparing for genericity

Separate out list-specific 'shape' from type recursion:

$$\textbf{data}\ \mathit{ListS}\ a\ b = \mathit{NilS} \mid \mathit{ConsS}\ a\ b$$
$$\textbf{data}\ \mathit{Fix}\ s\ a = \mathit{In}\ \{\ \mathit{out} :: s\ a\ (\mathit{Fix}\ s\ a)\ \} \quad \text{-- } \mathit{In} \text{ and } \mathit{out} \text{ together}$$
$$\textbf{type}\ \mathit{List}\ a = \mathit{Fix}\ \mathit{ListS}\ a$$
$$\mathit{bimap} :: (a \to a') \to (b \to b') \to \mathit{ListS}\ a\ b \to \mathit{ListS}\ a'\ b'$$

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

$$\mathit{foldList} :: (\mathit{ListS}\ a\ b \to b) \to \mathit{List}\ a \to b$$
$$\mathit{foldList}\ f = f \circ \mathit{bimap}\ \mathit{id}\ (\mathit{foldList}\ f) \circ \mathit{out}$$

eg $\mathit{foldList}\ \mathit{add} :: \mathit{List}\ \mathit{Integer} \to \mathit{Integer}$, where

$$\mathit{add} :: \mathit{ListS}\ \mathit{Integer}\ \mathit{Integer} \to \mathit{Integer}$$
$$\mathit{add}\ \mathit{NilS} = 0$$
$$\mathit{add}\ (\mathit{ConsS}\ m\ n) = m + n$$

7. Going datatype-generic

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

$$\textbf{class}\ \mathit{Bifunctor}\ s\ \textbf{where}$$
$$\quad \mathit{bimap} :: (a \to a') \to (b \to b') \to s\ a\ b \to s\ a'\ b'$$

Then *fold* works for any suitable shape:

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

Of course, *ListS* is a suitable shape...

$$\begin{array}{l}\textbf{instance}\ \mathit{Bifunctor}\ \mathit{ListS}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{bimap}\ f\ g\ \mathit{NilS} & = & \mathit{NilS} \\ \mathit{bimap}\ f\ g\ (\mathit{ConsS}\ a\ b) & = & \mathit{ConsS}\ (f\ a)\ (g\ b) \end{array}\end{array}$$

7. Going datatype-generic

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

$$
\begin{aligned}
&\textbf{class}\ \mathit{Bifunctor}\ s\ \textbf{where} \\
&\quad \mathit{bimap} :: (a \to a') \to (b \to b') \to s\ a\ b \to s\ a'\ b'
\end{aligned}
$$

Then *fold* works for any suitable shape:

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b \\
&\mathit{fold}\ f = f \circ \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \circ \mathit{out}
\end{aligned}
$$

...but binary trees are also suitable:

$$
\textbf{data}\ \mathit{TreeS}\ a\ b = \mathit{TipS}\ a \mid \mathit{BinS}\ b\ b
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Bifunctor}\ \mathit{TreeS}\ \textbf{where} \\
&\quad \mathit{bimap}\ f\ g\ (\mathit{TipS}\ a) \qquad\ \ = \mathit{TipS}\ (f\ a) \\
&\quad \mathit{bimap}\ f\ g\ (\mathit{BinS}\ b_1\ b_2) = \mathit{BinS}\ (g\ b_1)\ (g\ b_2)
\end{aligned}
$$

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

$$h \circ f = g \circ bimap\ id\ h$$

8. The categorical view, in a nutshell

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

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

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

Algebra (B, f) is *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

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

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

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

$$\begin{aligned} \mathit{In} \quad &:: \mathit{ListS\ Integer}\ (\mathit{List\ Integer}) \to \mathit{List\ Integer} \\ \mathit{add} &:: \mathit{ListS\ Integer\ Integer} \to \mathit{Integer} \end{aligned}$$

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

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

$$h \circ f = g \circ bimap\ id\ h$$

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

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

$$\begin{aligned} In \quad &:: ListS\ Integer\ (List\ Integer) \to List\ Integer \\ add &:: ListS\ Integer\ Integer \to Integer \end{aligned}$$

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

Theorem: for all sensible shape functors S, initial algebras exist.

9. Duality

Recall

$$fold :: Bifunctor\ s \Rightarrow (s\ a\ b \to b) \to (Fix\ s\ a \to b)$$
$$fold\ f = f \circ bimap\ id\ (fold\ f) \circ out$$

9. Duality

Recall

$$\begin{aligned}
&\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to (\mathit{Fix}\ s\ a \to b)\\
&\mathit{fold}\ f = f \circ \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \circ \mathit{out}
\end{aligned}$$

Reverse certain arrows:

$$\begin{aligned}
&\mathit{unfold} :: \mathit{Bifunctor}\ s \Rightarrow (b \to s\ a\ b) \to (b \to \mathit{Fix}\ s\ a)\\
&\mathit{unfold}\ f = \mathit{In} \circ \mathit{bimap}\ \mathit{id}\ (\mathit{unfold}\ f) \circ f
\end{aligned}$$

9. Duality

Recall

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to (\mathit{Fix}\ s\ a \to b)\\
&\mathit{fold}\ f = f \circ \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \circ \mathit{out}
\end{aligned}
$$

Reverse certain arrows:

$$
\begin{aligned}
&\mathit{unfold} :: \mathit{Bifunctor}\ s \Rightarrow (b \to s\ a\ b) \to (b \to \mathit{Fix}\ s\ a)\\
&\mathit{unfold}\ f = \mathit{In} \circ \mathit{bimap}\ \mathit{id}\ (\mathit{unfold}\ f) \circ f
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{unfoldr} :: (b \to \mathit{Maybe}\ (a, b)) \to b \to [\,a\,]\\
&\mathit{unfoldr}\ f\ b = \mathbf{case}\ f\ b\ \mathbf{of}\\
&\qquad \mathit{Nothing} \to [\,]\\
&\qquad \mathit{Just}\ (a, b') \to a : \mathit{unfoldr}\ f\ b'
\end{aligned}
$$

9. Duality

Recall

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

Reverse certain arrows:

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

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

$$\begin{aligned}
&\mathit{unfoldr} :: (b \to \mathit{Maybe}\ (a, b)) \to b \to [\,a\,] \\
&\mathit{unfoldr}\ f\ b = \mathbf{case}\ f\ b\ \mathbf{of} \\
&\qquad\qquad \mathit{Nothing} \quad\ \to [\,] \\
&\qquad\qquad \mathit{Just}\ (a, b') \to a : \mathit{unfoldr}\ f\ b'
\end{aligned}$$

Categorically, *coalgebras* (B, f) with $f :: B \to S\ A\ B$, *finality*.

10. Conclusions

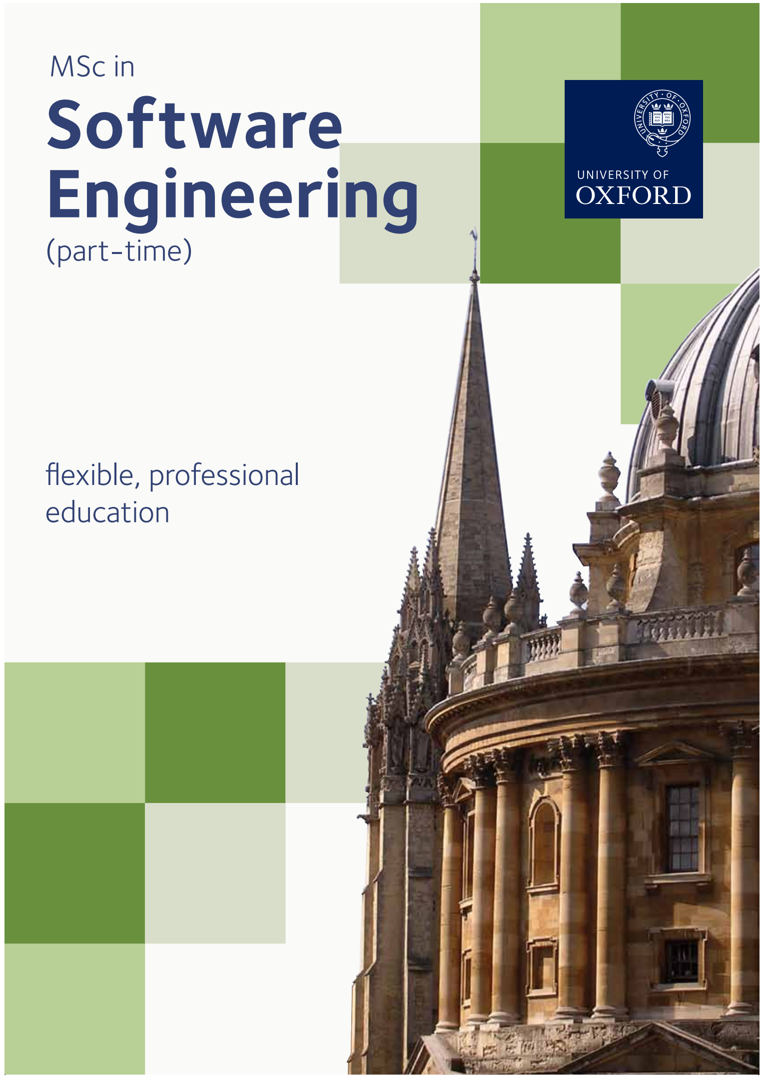
- category theory as an organisational tool, not for intimidation
- helping you to write better code, with *less mess*
- the mathematics is really quite pretty
- . . . but the Haskell makes sense on its own too

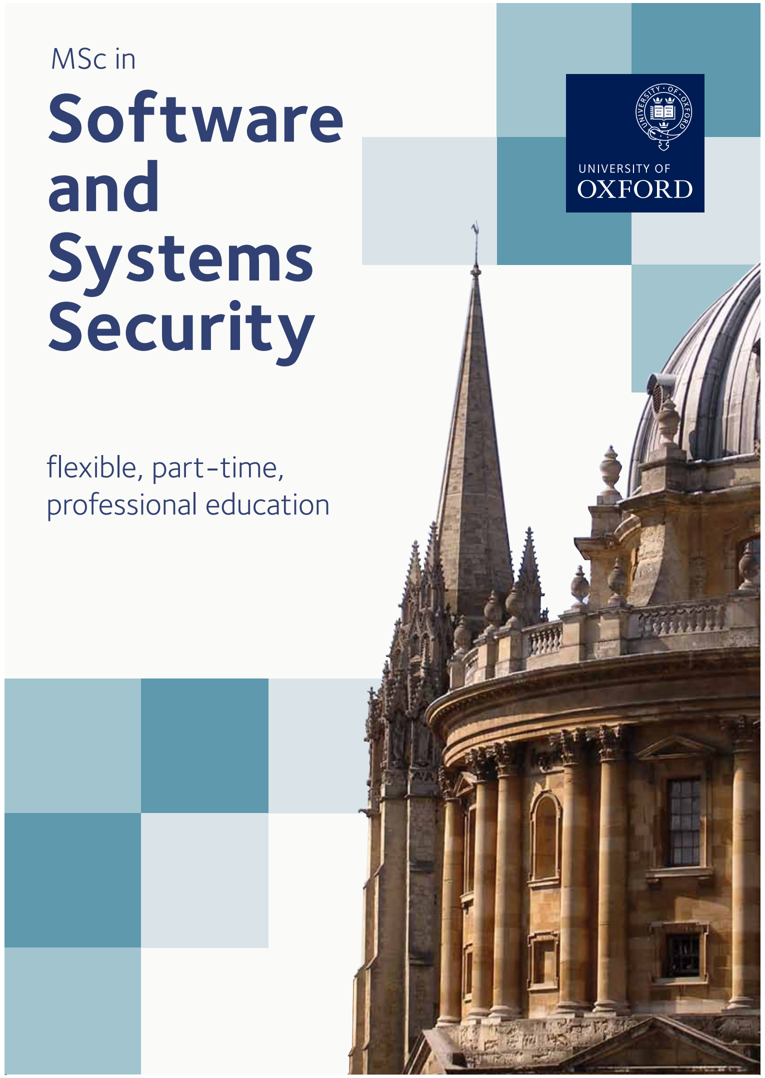
10. Conclusions

- 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. Software Engineering Programme



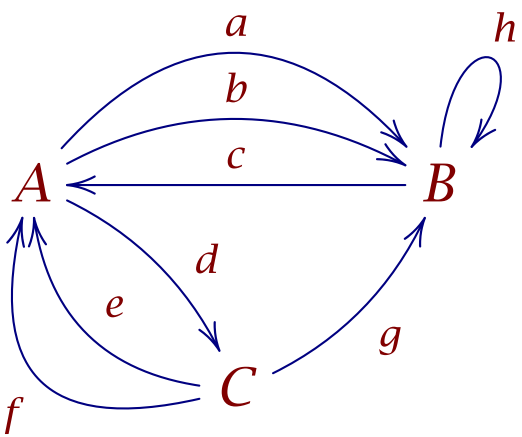


Appendix: category theory

12. ‘Category’

A category consists of

- a collection of *objects*
- for each pair A, B of objects, a collection $A \to B$ of *arrows*
- 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

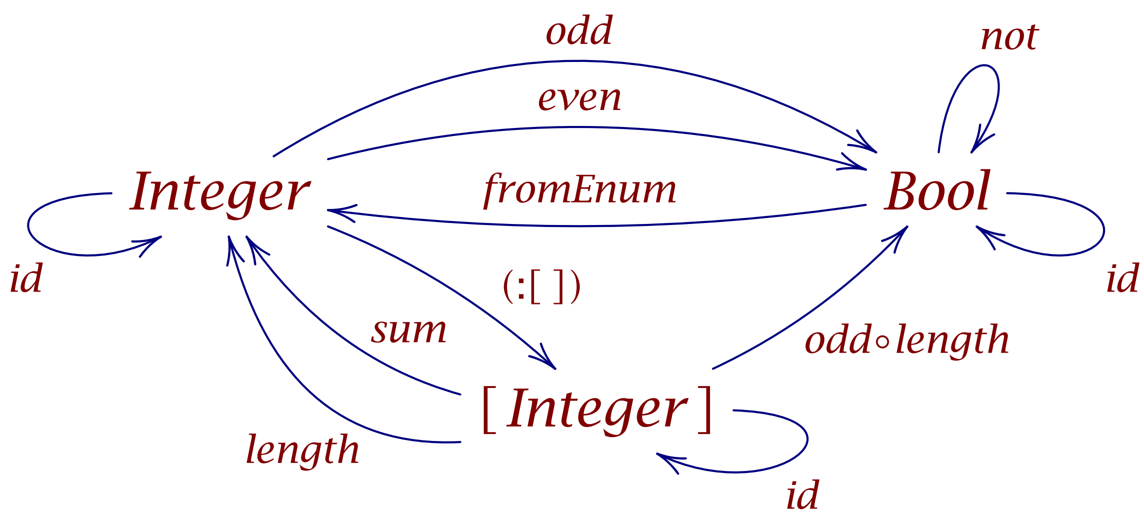


(think of *paths* in labelled directed graphs)

12. 'Category'

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



(some of category *SET*, in which objects are sets and arrows are total functions)

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 *List* is simultaneously

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

such that

- $List\ f : List\ A \rightarrow List\ B$ when $f : A \rightarrow B$
- $List\ id = id$
- $List\ (f \circ g) = List\ f \circ List\ g$

13. ‘Functor’

Functor $\mathit{ListS}\ A$ is simultaneously

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

such that

- $(\mathit{ListS}\ A)\ f : \mathit{ListS}\ A\ B \to \mathit{ListS}\ A\ B'$ when $f : B \to B'$
- $(\mathit{ListS}\ A)\ \mathit{id} = \mathit{id}$
- $(\mathit{ListS}\ A)\ (f \circ g) = (\mathit{ListS}\ A)\ f \circ (\mathit{ListS}\ A)\ g$

14. ‘Algebra’

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

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

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

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

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

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

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

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

$$\begin{tikzcd} F\ A \arrow[r, "f"] \arrow[d, "F\ h"'] \arrow[dr, phantom, "☺"] & A \arrow[d, "h"] \\ F\ B \arrow[r, "g"'] & B \end{tikzcd}$$

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

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

$$\begin{tikzcd} F\ A \arrow[r, "f"] \arrow[d, "F\ h"'] \arrow[dr, phantom, "☺"] & A \arrow[d, "h"] \\ F\ B \arrow[r, "g"'] & B \end{tikzcd}$$

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

$$sum \circ In = add \circ bimap\ id\ sum$$

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

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

$$\begin{tikzcd} F\ A \arrow[r, "f"] \arrow[d, "F\ h"'] \arrow[dr, phantom, "☺"] & A \arrow[d, "h"] \\ F\ B \arrow[r, "g"'] & B \end{tikzcd}$$

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

$$sum \circ In = add \circ bimap\ id\ sum$$

(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: $(\mathit{List\ Integer}, \mathit{In})$ is the initial $(\mathit{ListS\ Integer})$-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: $(List\ Integer, In)$ is the initial $(ListS\ Integer)$-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.

(*polynomial*: constructed from sums and products, like simple algebraic datatypes)*

Datatype-generically, too.

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**: $(List\ Integer, In)$ is the initial $(ListS\ Integer)$-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.

(*polynomial**: *constructed from sums and products, like simple algebraic datatypes*)

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 $\varnothing$, 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

$$
\begin{aligned}
h\ [\,] &= 3 \\
h\ (x : xs) &= \mathit{const}\ (\mathit{const}\ 3)\ x\ (h\ xs)
\end{aligned}
$$

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

$$\begin{aligned} h\,[\,] &= 3 \\ h\,(x : xs) &= const\ (const\ 3)\ x\ (h\ xs) \end{aligned}$$

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