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 \textit{sum}: 

\begin{align*}
\text{sum} :: & [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum} [ ] & = 0 \\
\text{sum} (x : xs) & = x + \text{sum} \; xs
\end{align*}
2. Functions that consume lists

Two equations, indirectly defining \textit{sum}:

\begin{align*}
sum &:: [Integer] \rightarrow Integer \\
sum [ ] & = 0 \\
sum (x : xs) & = x + sum xs
\end{align*}

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

\begin{align*}
h [ ] & = e \\
h (x : xs) & = f \ x \ (h \ xs)
\end{align*}
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 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

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

etc etc
4. What’s special about lists?

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

\[
\textbf{data} \quad \text{List}\ a = \text{Nil} \mid \text{Cons}\ a\ (\text{List}\ a)
\]

Then we could have

\[
\begin{align*}
\text{foldList} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List}\ a \rightarrow b \\
\text{foldList}\ f\ e\ \text{Nil} &= e \\
\text{foldList}\ f\ e\ (\text{Cons}\ x\ xs) &= f\ x\ (\text{foldList}\ f\ e\ xs)
\end{align*}
\]
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):

```haskell
data Rose a = Node a [ Rose a ]
```
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))
```
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))
```
5. It’s not always so obvious

Rose trees (eg for games, or XML):

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

\[
foldRose_1 :: (a \rightarrow c \rightarrow b) \rightarrow (b \rightarrow c \rightarrow c) \rightarrow c \rightarrow \text{Rose } a \rightarrow b
\]

\[
foldRose_1 f g e (\text{Node } x \ ts) = f \ x \ (\text{foldr } g e \ (\text{map } (\text{foldRose}_1 f g e) \ ts))
\]

\[
foldRose_2 :: (a \rightarrow b \rightarrow b) \rightarrow ([b] \rightarrow b) \rightarrow \text{Rose } a \rightarrow b
\]

\[
foldRose_2 f g (\text{Node } x \ ts) = f \ x \ (g \ (\text{map } (\text{foldRose}_2 f g) \ ts))
\]

\[
foldRose_3 :: (a \rightarrow [b] \rightarrow b) \rightarrow \text{Rose } a \rightarrow b
\]

\[
foldRose_3 f (\text{Node } x \ ts) = f \ x \ (\text{map } (\text{foldRose}_3 f) \ ts)
\]

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

Rose trees (eg for games, or XML):

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

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

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

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

\[
foldRose_2 f g (\text{Node } x \ ts) = f \ x \ (g \ (\text{map} \ (foldRose_2 f g) \ ts))
\]

\[
foldRose_3 :: (a \rightarrow [b] \rightarrow b) \rightarrow Rose a \rightarrow b
\]

\[
foldRose_3 f (\text{Node } x \ ts) = f \ x \ (\text{map} \ (foldRose_3 f) \ ts)
\]

Which should we choose?

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

\[
foldr1, foldl1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a
\]
6. Preparing for genericity

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

\[
\begin{align*}
\text{data } & \text{ListS } a \ b = \text{NilS} \mid \text{ConsS } a \ b \\
\text{data } & \text{Fix } s \ a = \text{In } (s \ a (\text{Fix } s \ a)) \\
\text{type } & \text{List } a = \text{Fix } \text{ListS } a
\end{align*}
\]

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

\[
\text{In } (\text{ConsS } 1 (\text{In } (\text{ConsS } 2 (\text{In } (\text{ConsS } 3 (\text{In } \text{NilS})))))))
\]

For convenience, define inverse \textit{out} to \textit{In}:

\[
\begin{align*}
\text{out } :: & \text{Fix } s \ a \to s \ a (\text{Fix } s \ a) \\
\text{out } (\text{In } x) & = x
\end{align*}
\]
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

Shape is mostly opaque; just need to ‘locate’ the \(a\)s and \(b\)s:

```haskell
bimap :: (a \rightarrow a') \rightarrow (b \rightarrow b') \rightarrow ListS a b \rightarrow ListS a' b'

\[
bimap f g \text{ NilS} = \text{ NilS} \\
bimap f g (\text{ConsS} a b) = \text{ConsS} (f a) (g b)
\]
```
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

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

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

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

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

```haskell
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 \circ \textit{bimap \textit{id}} (fold f) \circ \textit{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 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 *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 $(C, g)$.
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 \( f :: S A B \rightarrow B \). A \textit{homomorphism} between \( (B, f) \) and \( (C, g) \) is a function \( h :: B \rightarrow C \) such that

\[ h \circ f = g \circ \text{bimap id} h \]

Algebra \( (B, f) \) is \textit{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 \textit{ListS Integer}:

\[
\begin{align*}
\text{In} & :: \text{ListS Integer} (\text{List Integer}) \rightarrow \text{List Integer} \\
\text{add} & :: \text{ListS Integer Integer} \rightarrow \text{Integer}
\end{align*}
\]

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

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

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

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

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

Recall

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

Reverse certain arrows:

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

Recall

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

Reverse certain arrows:

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

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

\[
\text{unfoldr} :: (b \rightarrow \text{Maybe } (a, b)) \rightarrow b \rightarrow [a]
\]
\[
\text{unfoldr } f \ b = \text{case } f \ b \text{ of }
\]
\[
\begin{align*}
\text{Nothing} & \rightarrow [ ] \\
\text{Just } (a, b') & \rightarrow a : \text{unfoldr } f \ b'
\end{align*}
\]
9. Duality

Recall

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

Reverse certain arrows:

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

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

\[
\text{unfoldr} :: (b \rightarrow \text{Maybe } (a, b)) \rightarrow b \rightarrow [a]
\]
\[
\text{unfoldr } f \ b = \text{case } f \ b \text{ of }
\]
\[
\text{Nothing} \rightarrow [ ] \quad \text{Just } (a, b') \rightarrow a : \text{unfoldr } f \ b'
\]

Categorically, \textit{coalgebras} \((B, f)\) with \(f :: B \rightarrow S A B\), \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. 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 $A \to B$ 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 \( A \rightarrow B \) of *arrows* (functions)
- 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

(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 \textit{List} is simultaneously

\begin{itemize}
\item an operation on objects \((\text{List } A = [A])\)
\item an operation on arrows \((\text{List } f = \text{map } f)\)
\end{itemize}

such that

\begin{itemize}
\item \text{List } f : \text{List } A \rightarrow \text{List } B \text{ when } f : A \rightarrow B
\item \text{List } id = id
\item \text{List } (f \circ g) = \text{List } f \circ \text{List } g
\end{itemize}
13. ‘Functor’

Functor $ListS A$ is simultaneously

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

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 *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 \textit{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 $(\text{Integer}, \text{add})$:

$$\text{sum} \circ \text{In} = \text{add} \circ \text{bimap id} \ \text{sum}$$
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 $(\text{Integer}, \text{add})$:

$$\text{sum} \circ \text{In} = \text{add} \circ \text{bimap id sum}$$

(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:** $(\text{List Integer}, \text{In})$ 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:** $(\text{List Integer, In})$ 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:** $(\text{List Integer}, \text{In})$ 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.

(More generally, an *initial object* in a category is one with a unique arrow to every other object. In $\text{SET}$, 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 \( \text{SET} \), but not some other settings
- not quite true for realistic Haskell
  
  *undefined* values, *infinite* data structures, *strictness*...
- defining equations do not always uniquely define \textit{foldr}—consider

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

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

\[
\begin{align*}
h[ ] &= 3 \\
h(x:xs) &= \text{const}\left(\text{const } 3\right) \times (h \ xs)
\end{align*}
\]

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