Algebra of Functional Programs

Filip Pawlak

filippawlak@gmail.com

University of Wrocław

May 20, 2015

In this talk we will isolate four common recursion schemes (naturally associated with recursive data types), give them a generic theoretical treatment and prove a number of general laws about them. Then we will use these laws in an example.

Motivation

It is sometimes quite hard to reason about arbitrary recursively defined functions.

By sticking to well-defined recursion patterns, we are able to:

- use already proven theorems to optimize or prove properties
- calculate programs and reason about them more easily (chiefly because we can now refer to the recursion scheme in isolation)
- reuse code and ideas

Introducing fold

```
sum  []     = 0
sum  (x:xs) = x + sum  xs
```

This sort of pattern is very common in functional programs.

```
h  []     = e
h  (x:xs) = f  x  (h  xs)
```

It's so common that it deserves its own higher-order function in the standard library:

```
foldr  ::  (a -> b -> b) -> b -> [a] -> b
foldr  f  e  []      = e
foldr  f  e  (x : xs) = f  x  (foldr  f  e  xs)
```

Introducing fold

But is it only for lists?

We could've defined our own list type:

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

Then the fold function for that type would be:

```
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)
```

What about other data structures?

Introducing fold

```
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)
```

Introducing fold

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

foldRose :: (a -> [b] -> b) -> Rose a -> b
foldRose f (Node x ts) = f x (map (foldRose f) ts)
```

Do you see the pattern?

Folds in a sense "replace" data constructors with the functions/values that you pass as arguments.

Datatype-generic fold

We separate the "list shape" from type recursion:

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

As an example, list [1, 2] is represented by

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

We also define an inverse to in named out:

```
out :: Fix s a -> s a (Fix s a)
out (In x) = x
```

Datatype-generic fold

We separate the "list shape" from type recursion:

```
data ListS a b = NilS | ConsS a b
data Fix s a = In {out :: s a (Fix s a)}
type List a = Fix ListS a
```

Now we define a function bimap which applies its arguments f and g to all the a's and b's in an argument of type ListS a b:

```
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)
```

```
data ListS a b = NilS | ConsS a b
data Fix s a = In {out :: s a (Fix s a)}
type List a = Fix ListS a

bimap :: (a -> a') -> (b -> b') ->
         ListS a b -> ListS a' b'
```

Now we can write a different version of fold on List:

```
foldList :: (ListS a b -> b) -> List a -> b
foldList f = f . bimap id (foldList f ) . out
```

f gives the "interpretation" of constructors. For example sum = foldList add :: List Integer -> Integer, where

```
add :: ListS Integer Integer -> Integer
add NilS = 0
add (ConsS m n) = m + n
```

Datatype-generic fold

Now we also want to abstract away from the specific ListS shape. To be suitable, a shape must support bimap (so it must be a bifunctor):

```
class Bifunctor s where
bimap :: (a -> a') -> (b -> b') -> s a b -> s a' b'
```

Then fold works for any suitable shape:

```
fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
fold f = f . bimap id (fold f) . out
```

...and one of these shapes is ListS:

```
instance Bifunctor ListS where
  bimap f g NilS = NilS
  bimap f g (ConsS a b) = ConsS (f a) (g b)
```

Datatype-generic fold

Now we also want to abstract away from the specific ListS shape. To be suitable, a shape must support bimap (so it must be a bifunctor):

```
class Bifunctor s where
bimap :: (a -> a') -> (b -> b') -> s a b -> s a' b'
```

Then fold works for any suitable shape:

```
fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
fold f = f . bimap id (fold f) . out
```

...but binary trees also fit:

```
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)
```

A **category** consists of:

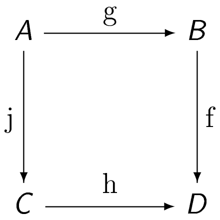
- a collection of objects
- for each pair A, B of objects, a collection Mor(A, B) of arrows (also called morphisms)
- an identity arrow id A : $A \to A$ for each object A
- composition $f \circ g : A \to C$ of compatible arrows $f : B \to C$ and $g : A \to B$
- composition is associative, and identities are neutral elements

The categorical view

The category **SET** consists of:

- a collection of objects (**sets**, or in our case types)
- for each pair A, B of objects, a collection Mor(A, B) of arrows **(total 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

Before we go any further...



We compose morphisms along the arrows. We say that the diagram **commutes** when $f \circ g = h \circ j$.

A functor F is simultaneously

- an operation on objects
- an operation on arrows

such that

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

(think fmap in Haskell)

The categorical view

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

The categorical view

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

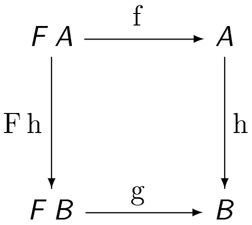
An **F-algebra** for functor F is a pair (A, f) where $f : F\ A \rightarrow A$.

An **F-algebra** for functor F is a pair (A, f) where f : F A→A.

Eg. (List Integer, In) and (Integer, add) are both F-algebras for ListS Integer:

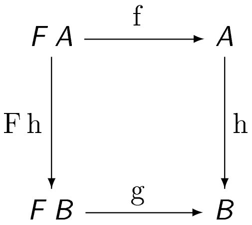
```
In  :: ListS Integer (List Integer) -> List Integer
add :: ListS Integer Integer -> Integer
```

A **homomorphism** between F-algebras (A, f) and (B, g) is a morphism h : A→B such that the following diagram commutes:



...or, equivalently, that $h \circ f = g \circ F\ h$.

A **homomorphism** between F-algebras (A, f) and (B, g) is a morphism h : A→B such that the following diagram commutes:



...or, equivalently, that $h \circ f = g \circ F\ h$.

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

The categorical view

It turns out that for an initial F-algebra (A, f) the action f has an inverse, so F A is isomorphic to A (which is what we really want for our inductive data types). In this sense A is a fixed point of F.

It turns out that for an initial F-algebra (A, f) the action f has an inverse, so F A is isomorphic to A (which is what we really want for our inductive data types). In this sense A is a fixed point of F.

We usually call the f function for an initial algebra **in**.

The categorical view

It turns out that for an initial F-algebra (A, f) the action f has an inverse, so F A is isomorphic to A (which is what we really want for our inductive data types). In this sense A is a fixed point of F.

We usually call the f function for an initial algebra **in**. We usually call the inverse function **out**.

The categorical view

It turns out that for an initial F-algebra (A, f) the action f has an inverse, so F A is isomorphic to A (which is what we really want for our inductive data types). In this sense A is a fixed point of F.

We usually call the f function for an initial algebra **in**. We usually call the inverse function **out**.

Sidenote 1: (List Integer, In) is an initial algebra for the functor ListS Integer. (Integer, add) is not, as add doesn't even have an inverse.

The categorical view

It turns out that for an initial F-algebra (A, f) the action f has an inverse, so F A is isomorphic to A (which is what we really want for our inductive data types). In this sense A is a fixed point of F.

We usually call the f function for an initial algebra **in**. We usually call the inverse function **out**.

Sidenote 1: (List Integer, In) is an initial algebra for the functor ListS Integer. (Integer, add) is not, as add doesn't even have an inverse.
Sidenote 2: the existence of initial algebras is guaranteed for so-called polynomial functors (so all functors that we consider in this talk).

The categorical view

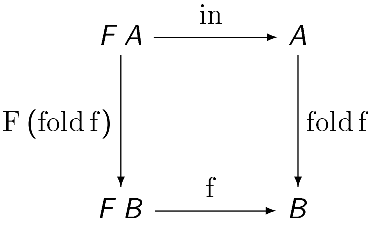
Once again, for an initial F-algebra (A, in), there is a unique homomorphism to each F-algebra (B, f). We call this homomorphism **fold f**.

The categorical view

Once again, for an initial F-algebra (A, in), there is a unique homomorphism to each F-algebra (B, f). We call this homomorphism **fold f**.

Sidenote: this homomorphism is also called a **catamorphism**.

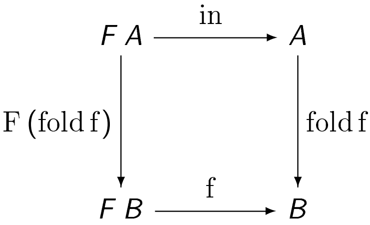
For an initial F-algebra (A, in), there is a unique homomorphism to each F-algebra (B, f). We call this homomorphism **fold f**.



Its defining property is that fold f $\circ$ in $=$ f $\circ$ F fold f.

The categorical view

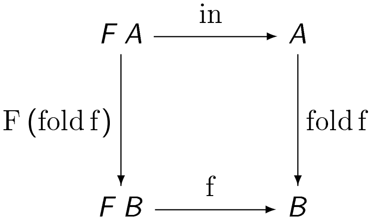
For an initial F-algebra (A, in), there is a unique homomorphism to each F-algebra (B, f). We call this homomorphism **fold f**.



Its defining property is that fold f ∘ in = f ∘ F fold f.

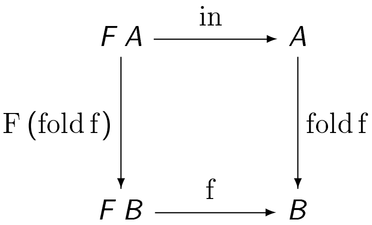
Intuitively, it means that it behaves nicely with F.

The categorical view



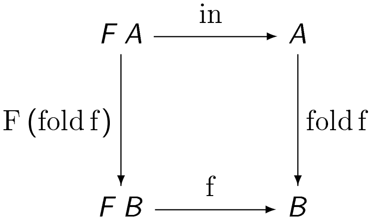
We have that fold ∘ in = f ∘ F fold f.

The categorical view



We have that fold ∘ in = f ∘ F fold f.

Because in ∘ out = id, we can also write fold f = f ∘ F fold f ∘ out.



We have that fold ∘ in = f ∘ F fold f.

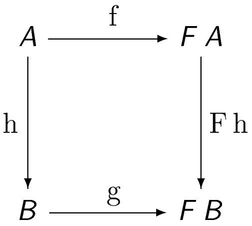
Because in ∘ out = id, we can also write fold f = f ∘ F fold f ∘ out.

```
fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
fold f = f . bimap id (fold f) . out
```

An **F-coalgebra** is a pair (A, f) where $f : A \to F\ A$.

The categorical view

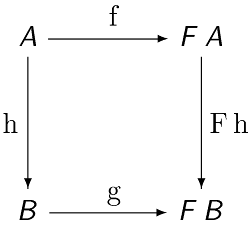
An **F-coalgebra** is a pair (A, f) where f : A $\rightarrow$ F A.
A **homomorphism** between F-coalgebras (A, f) and (B, g) is a morphism h : A $\rightarrow$ B such that the following diagram commutes:



...or, equivalently, that $g \circ h = F\ h \circ f$.

The categorical view

An **F-coalgebra** is a pair (A, f) where f : A $\rightarrow$ F A.
A **homomorphism** between F-coalgebras (A, f) and (B, g) is a morphism h : A $\rightarrow$ B such that the following diagram commutes:



...or, equivalently, that g $\circ$ h = F h $\circ$ f.

An F-coalgebra (B, g) is **terminal** if there is a unique homomorphism to it **from** every F-coalgebra (A, f).

The categorical view

It turns out that for a terminal F-coalgebra (B, g) the action g has an inverse, so B is isomorphic to F B (which is what we really want for our inductive data types). In this sense B is a fixed point of F.

It turns out that for a terminal F-coalgebra (B, g) the action g has an inverse, so B is isomorphic to F B (which is what we really want for our inductive data types). In this sense B is a fixed point of F.

We usually call the g function for a terminal coalgebra **out**.

The categorical view

It turns out that for a terminal F-coalgebra (B, g) the action g has an inverse, so B is isomorphic to F B (which is what we really want for our inductive data types). In this sense B is a fixed point of F.

We usually call the g function for a terminal coalgebra **out**. We usually call the inverse function **in**.

The categorical view

It turns out that for a terminal F-coalgebra (B, g) the action g has an inverse, so B is isomorphic to F B (which is what we really want for our inductive data types). In this sense B is a fixed point of F.

We usually call the g function for a terminal coalgebra **out**. We usually call the inverse function **in**.

Sidenote: the existence of terminal coalgebras is guaranteed for all functors that we consider in this talk.

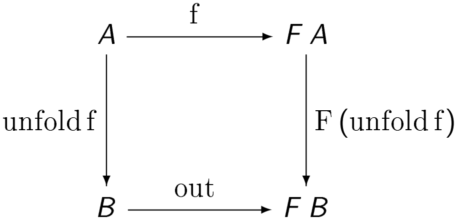
Once again, for a terminal F-coalgebra (B, out), there is a unique homomorphism to it from each F-algebra (A, f). We call this homomorphism **unfold** f.

Once again, for a terminal F-coalgebra (B, out), there is a unique homomorphism to it from each F-algebra (A, f). We call this homomorphism **unfold f**.

Sidenote: this homomorphism is also called an **anamorphism**.

The categorical view

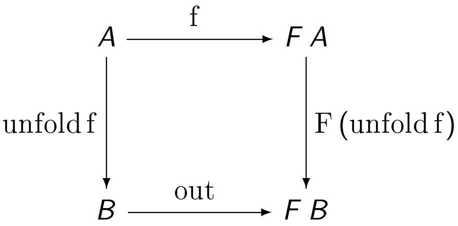
For a terminal F-coalgebra (B, out), there is a unique homomorphism to it from each F-algebra (A, f). We call this homomorphism **unfold f**.



Its defining property is that out $\circ$ unfold f = F unfold f $\circ$ f.

The categorical view

For a terminal F-coalgebra (B, out), there is a unique homomorphism to it from each F-algebra (A, f). We call this homomorphism **unfold f**.



Its defining property is that out ∘ unfold f = F unfold f ∘ f.

Intuitively, unfold f takes some seed of type A and generates something of type B (eg. a list). f takes a seed and generates both an intermediate value (eg. a list element) and a seed for unfold to generate the rest of the structure (the tail of the list).

(Note: there can be many values and many seeds, depending on the functor/the shape of the inductive type definition - take binary trees as an example.)

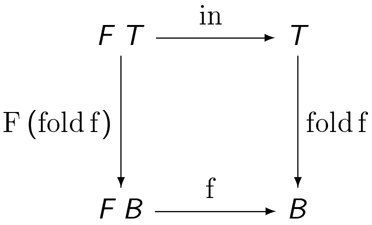
(Note: there can be many values and many seeds, depending on the functor/the shape of the inductive type definition - take binary trees as an example.)

We have that $\mathrm{out} \circ \mathrm{unfold}\ f = F\ \mathrm{unfold}\ f \circ f$.
Because $\mathrm{in} \circ \mathrm{out} = \mathrm{id}$, we also have $\mathrm{unfold}\ f = \mathrm{in} \circ F\ \mathrm{unfold}\ f \circ f$.

(Note: there can be many values and many seeds, depending on the functor/the shape of the inductive type definition - take binary trees as an example.)

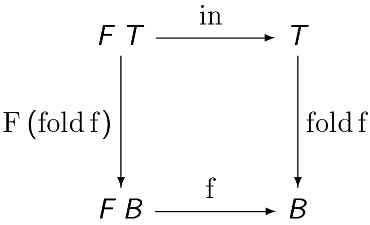
We have that out $\circ$ unfold f = F unfold f $\circ$ f.
Because in $\circ$ out = id, we also have unfold f = in $\circ$ F unfold f $\circ$ f.

```
unfold :: Bifunctor s => (b -> s a b) ->
          (b -> Fix s a)
unfold f = In . bimap id (unfold f) . f
```



Now we can finally prove some laws about these recursion schemes. We begin with fold. We will often make use of the following universal property, which is a consequence of the uniqueness of fold:

$$h = \mathrm{fold}_T f \iff h \circ \mathrm{in}_T = f \circ Fh$$



Now we can finally prove some laws about these recursion schemes. We begin with fold. We will often make use of the following universal property, which is a consequence of the uniqueness of fold:

$$h = \mathsf{fold}_T f \iff h \circ \mathsf{in}_T = f \circ Fh$$

It is a sort of "canned induction proof".

Intuitively, the evaluation rule shows "one step of evaluation" of a fold.

$$\mathsf{fold}_T f \circ \mathsf{in}_T$$
$$= \quad \{\text{universal property, letting h = fold f}\}$$
$$f \circ \mathsf{F}(\mathsf{fold}_T f)$$

Fusion (exact version)

$$
\begin{aligned}
& h \circ \mathsf{fold}_T f = \mathsf{fold}_T g \\
\iff \quad & \{\text{universal property}\} \\
& h \circ \mathsf{fold}_T f \circ \mathsf{in}_T = g \circ \mathsf{F}(h \circ \mathsf{fold}_T f) \\
\iff \quad & \{\text{functors}\} \\
& h \circ \mathsf{fold}_T f \circ \mathsf{in}_T = g \circ \mathsf{F}h \circ \mathsf{F}(\mathsf{fold}_T f) \\
\iff \quad & \{\text{evaluation rule}\} \\
& h \circ f \circ \mathsf{F}(\mathsf{fold}_T f) = g \circ \mathsf{F}h \circ \mathsf{F}(\mathsf{fold}_T f)
\end{aligned}
$$

Again, it's a kind of a "canned induction proof".

Fusion (weaker version)

$$
\begin{aligned}
& h \circ \mathrm{fold}_T f = \mathrm{fold}_T g \\
\Longleftrightarrow \quad & \{\text{exact fusion}\} \\
& h \circ f \circ \mathrm{F}(\mathrm{fold}_T f) = g \circ \mathrm{F}h \circ \mathrm{F}(\mathrm{fold}_T f) \\
\Longleftarrow \quad & \{\text{Leibniz}\} \\
& h \circ f = g \circ \mathrm{F}h
\end{aligned}
$$

Much easier to use.

Identity

The identity function id is a fold:

$$\begin{aligned} & \mathsf{id} = \mathsf{fold}_T\, f \\ \iff & \quad \{\text{universal property}\} \\ & \mathsf{id} \circ \mathsf{in}_T = f \circ \mathsf{F}\ \mathsf{id} \\ \iff & \quad \{\text{identity}\} \\ & f = \mathsf{in}_T \end{aligned}$$

That is, $\mathsf{fold}_T\, \mathsf{in}_T = id$. Not very suprising, actually.

Also, the destructor out_T of a datatype, the inverse of the constructor out_T, can be written a a fold.

$$
\begin{aligned}
& \mathsf{in}_T \circ \mathsf{fold}_T f = \mathsf{id} \\
\iff \quad & \{\text{identity as a fold}\} \\
& \mathsf{in}_T \circ \mathsf{fold}_T f = \mathsf{fold}_T \mathsf{in}_T \\
\Longleftarrow \quad & \{\text{weak fusion}\} \\
& \mathsf{in}_T \circ f = \mathsf{in}_T \circ \mathsf{F}\ \mathsf{in}_T \\
\Longleftarrow \quad & \{\text{Leibniz}\} \\
& f = \mathsf{F}\ \mathsf{in}_T
\end{aligned}
$$

Therefore we can define $\mathsf{out}_T = \mathsf{fold}_T\,(F\,\mathsf{in}_T)$.

Destructors

We should check that this also makes out the inverse of in when the composition is reversed:

$$
\begin{aligned}
& out_T \circ \mathrm{in}_T \\
= & \quad \{\text{above}\} \\
& \mathrm{fold}_T(\mathrm{F}\ \mathrm{in}_T) \circ \mathrm{in}_T \\
= & \quad \{\text{evaluation rule}\} \\
& \mathrm{F}\ \mathrm{in}_T \circ \mathrm{F} out_T \\
= & \quad \{\text{functors}\} \\
& \mathrm{F}(\mathrm{in}_T \circ out_T) \\
= & \quad \{\mathrm{in} \circ \mathrm{out} = \mathrm{id}\} \\
& \mathrm{id}
\end{aligned}
$$

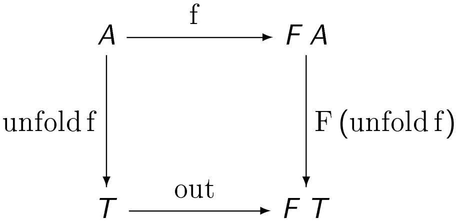
Destructors

We should check that this also makes out the inverse of in when the composition is reversed:

$$
\begin{aligned}
& out_T \circ \mathrm{in}_T \\
= \quad & \{\text{above}\} \\
& \mathrm{fold}_T(\mathrm{F}\ \mathrm{in}_T) \circ \mathrm{in}_T \\
= \quad & \{\text{evaluation rule}\} \\
& \mathrm{F}\ \mathrm{in}_T \circ \mathrm{F} out_T \\
= \quad & \{\text{functors}\} \\
& \mathrm{F}(\mathrm{in}_T \circ out_T) \\
= \quad & \{\mathrm{in} \circ \mathrm{out} = \mathrm{id}\} \\
& \mathrm{id}
\end{aligned}
$$

(Note: this is a corollary of a more general theorem, stating that every injective function on a recursive data type is a fold).

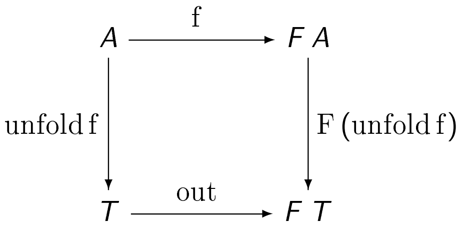
Universal property for unfold



The universal property for unfold is:

$$h = \mathsf{unfold}_T f \iff \mathsf{out}_T \circ h = F\ h \circ f$$

Universal property for unfold



The universal property for unfold is:

$$h = \text{unfold}_T f \iff \text{out}_T \circ h = F\ h \circ f$$

Again, a sort of "canned induction proof".

Properties of unfolds

The laws are simply duals to the laws for fold, so we just present them without proof.

Evaluation rule: $\text{out}_T \circ \text{unfold}_T\, f = F\, \text{unfold}_T\, f \circ f$
Exact and weak fusion: $\text{unfold}_T\, f \circ h = \text{unfold}_T\, g$
$\iff F\, (\text{unfold}_T\, f) \circ f \circ h = F\, (\text{unfold}_T\, f) \circ F\, h \circ g$
$\Longleftarrow f \circ h = F\, h \circ g$
Identity: $\text{unfold}_T\, \text{out}_T = id$
Constructors: $\text{in}_T = \text{unfold}_T\, (F\, \text{out}_T)$
The last law is a corollary of a more general law, stating that any surjective function to a recursive data type is an unfold.

The slide in which we move to CPO

Unfortunately, the category SET doesn't really suit us: we'd like to do things like $\textit{fold}\ f \circ \textit{unfold}\ g$, but in this category initial F-algebras and terminal F-coalgebras can be different objects. Moreover, it contains only total functions, so we can't express nontermination.

The slide in which we move to CPO

Unfortunately, the category SET doesn't really suit us: we'd like to do things like $\textit{fold } f \circ \textit{unfold } g$, but in this category initial F-algebras and terminal F-coalgebras can be different objects. Moreover, it contains only total functions, so we can't express nontermination.

We solve these problems by moving to the category CPO, where the objects are pointed complete partial orders and the arrows are continuous functions. Now initial F-algebras and terminal F-coalgebras are the same objects, up to isomorphism, and are fixed points of the functor F (for so-called locally continuous functors, so all functors that appear in this talk).

The slide in which we move to CPO

Unfortunately, the category SET doesn't really suit us: we'd like to do things like *fold f* $\circ$ *unfold g*, but in this category initial F-algebras and terminal F-coalgebras can be different objects. Moreover, it contains only total functions, so we can't express nontermination.

We solve these problems by moving to the category CPO, where the objects are pointed complete partial orders and the arrows are continuous functions. Now initial F-algebras and terminal F-coalgebras are the same objects, up to isomorphism, and are fixed points of the functor F (for so-called locally continuous functors, so all functors that appear in this talk). Also we have to include strictness conditions in some of our laws. So for example the universal property for fold becomes:

$$h = \mathrm{fold}_T f \iff h \circ \mathrm{in}_T = f \circ F\, h \quad \textit{for strict } f \textit{ and } h$$

Hylomorphisms

A **hylomorphism** h is a function which can be expressed as a composition of a fold following an unfold:

$h = \text{fold}_T\, g \circ \text{unfold}_T\, f$

A **hylomorphism** h is a function which can be expressed as a composition of a fold following an unfold:

$h = \mathrm{fold}_T\, g \circ \mathrm{unfold}_T\, f$

If h is of that form, then it can also be written as $g \circ F\, h \circ f$:

$$
\begin{aligned}
& \mathrm{h} \\
= \quad & \{\text{definition}\} \\
& \mathrm{fold}_T\, g \circ \mathrm{unfold}_T\, f \\
= \quad & \{\text{recursive definitions of fold and unfold}\} \\
& g \circ F\, \mathrm{fold}_T\, g \circ \mathrm{out}_T \circ \mathrm{in}_T \circ F\, \mathrm{unfold}_T\, f \circ f \\
= \quad & \{\mathrm{out} \circ \mathrm{in} = \mathrm{id}\} \\
& g \circ F\, \mathrm{fold}_T\, g \circ F\, \mathrm{unfold}_T\, f \circ f \\
= \quad & \{\text{functors, definition}\} \\
& g \circ F\, h \circ f
\end{aligned}
$$

Hylomorphisms

$$h = \text{fold}_T\, g \circ \text{unfold}_T\, f \implies h = g \circ F\, h \circ f$$

The law we've just derived is the *deforestation* law. It sometimes allows us to compute h without creating the intermediate structure returned by unfold.

The implication in the other direction also holds, but the proof requires more machinery.

Paramorphisms

A **paramorphism** is like a fold, but on every level of the recursion we have access to both the original value and the result of applying the paramorphism (so it "eats its argument and keeps it too"). The factorial function is a natural example of a paramorphism.

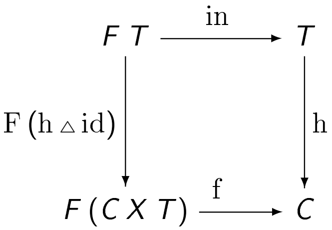
Formally, for an initial F-algebra (T, in_T) and f : $F\,(C \times T) \to C$, $\mathrm{para}_T f : T \to C$ is defined as follows:

$$\mathrm{para}_T\, f = \mathrm{exl} \circ \mathrm{fold}_T\, (f \vartriangle (\mathrm{in}_T \circ F\, \mathrm{exr}))$$

...where $\mathrm{exl}, \mathrm{exr}$ are pair projections and $(g \vartriangle h)\, x = (g\, x, h\, x)$.

Paramorphisms

$$\text{para}_T\, f = \text{exl} \circ \text{fold}_T\, (f \vartriangle (\text{in}_T \circ F\, \text{exr}))$$



It enjoys the following universal property:

$$h = \text{para}_T\, f \iff h \circ \text{in}_T = f \circ F\,(h \vartriangle id) \wedge h \perp = f \perp$$

Unsurprisingly, we can derive for paramorphisms similar laws as those listed for catamorphisms.

Example: banana split theorem

The theorem states that the fork of two folds is a fold (so we can traverse the data structure only once instead of twice):

$$\mathrm{fold}_T\, f \vartriangle \mathrm{fold}_T\, g = \mathrm{fold}_T\, ((f \circ F\, exl) \vartriangle (g \circ F\, exr))$$

We will use it to derive a one-pass solution for the problem of calculating the average of a list.

Example: banana split theorem

$$
\begin{aligned}
& \mathrm{fold}_T\, f \vartriangle \mathrm{fold}_T\, g = \mathrm{fold}_T\, ((f \circ F\, exl) \vartriangle (g \circ F\, exr)) \\
\iff & \quad \{\text{universal property}\} \\
& (\mathrm{fold}_T\, f \vartriangle \mathrm{fold}_T\, g) \circ in = \\
& ((f \circ F\, exl) \vartriangle (g \circ F\, exr)) \circ F\, (\mathrm{fold}_T\, f \vartriangle \mathrm{fold}_T\, g) \\
\iff & \quad \{\text{property of fork, functors, property of extractions}\} \\
& (\mathrm{fold}_T\, f \vartriangle \mathrm{fold}_T\, g) \circ in = ((f \circ F\, (\mathrm{fold}_T\, f)) \vartriangle (g \circ F\, (\mathrm{fold}_T\, g))) \\
\iff & \quad \{\text{property of fork}\} \\
& ((\mathrm{fold}_T\, f \circ in) \vartriangle (\mathrm{fold}_T\, g \circ in)) = \\
& ((f \circ F\, (\mathrm{fold}_T\, f)) \vartriangle (g \circ F\, (\mathrm{fold}_T\, g))) \\
\iff & \quad \{\text{evaluation rule}\} \\
& ((f \circ F\, (\mathrm{fold}_T\, f)) \vartriangle (g \circ F\, (\mathrm{fold}_T\, g))) = \\
& ((f \circ F\, (\mathrm{fold}_T\, f)) \vartriangle (g \circ F\, (\mathrm{fold}_T\, g)))
\end{aligned}
$$

$$average = DIV \circ sum \vartriangle length$$
$$sum = \mathrm{fold}_{ListInt}(const\, 0 \triangledown +)$$
$$length = \mathrm{fold}_{ListInt}(const\, 0 \triangledown (1+))$$

The banana split theorem lets us write:

$$sum \vartriangle length =$$
$$\mathrm{fold}_{ListInt}(((const\, 0 \triangledown +) \circ F\, exl) \vartriangle (const\, 0 \triangledown (1+)) \circ F\, exr)$$

($\triangledown$ is like either in Haskell)

The end.

References

Jeremy Gibbons, *Calculating Functional Programs*, 2002.

Erik Meijer et al., *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*, 1991.