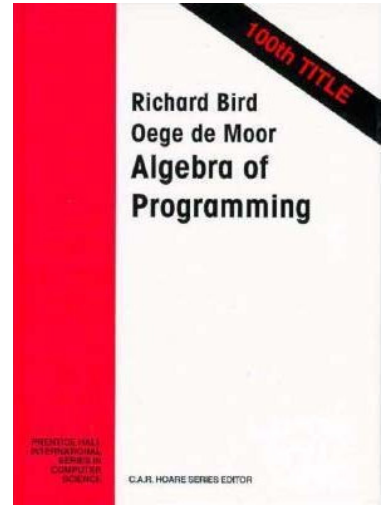
Category theory concepts appearing in functional programming

Janis Voigtländer

07.08.2019



Wholemeal programming

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- **“Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; …”**

  Ralf Hinze

- **“Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction.”**

  Richard Bird

- Algebraic data types can be recursive. For example:

```
data Nat = Zero | Succ Nat
```

- Values of this type:

```
Zero, Succ Zero, Succ (Succ Zero), ...
```

- Computation by recursive function definitions:

```
add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

- With several recursive occurrences, tree structures:

```
data Tree = Leaf | Node Tree Integer Tree
```

- Values: `Leaf`, `Node Leaf 2 Leaf`, ...

- Computation:

```
height :: Tree -> Integer
height Leaf
  = 0
height (Node left _ right)
  = 1 + max (height left) (height right)
```

What is "algebraic data" about these types?

Recall:

$$\text{data Nat} = \text{Zero} \mid \text{Succ Nat}$$

$$\text{data Tree} = \text{Leaf} \mid \text{Node Tree Integer Tree}$$

Each has an underlying (polynomial) functor in **Set**:

$$\begin{aligned}
\mathbf{F}_{\text{Nat}}\ A &= \{\text{Zero}\} \cup \{(\text{Succ}, a) \mid a \in A\} \\
\mathbf{F}_{\text{Nat}}\ h &: \text{Zero} \mapsto \text{Zero} \\
&\quad\ (\text{Succ}, a) \mapsto (\text{Succ}, h(a))
\end{aligned}$$

$$\begin{aligned}
\mathbf{F}_{\text{Tree}}\ A &= \{\text{Leaf}\} \cup \{(\text{Node}, a_1, z, a_2) \mid a_1 \in A, z \in \mathbb{Z}, a_2 \in A\} \\
\mathbf{F}_{\text{Tree}}\ h &: \text{Leaf} \mapsto \text{Leaf} \\
&\quad\ (\text{Node}, a_1, z, a_2) \mapsto (\text{Node}, h(a_1), z, h(a_2))
\end{aligned}$$

For these we can consider algebras, that is, arrows in **Set** of types $\mathbf{F}_{\text{Nat}}\ A \to A$ and $\mathbf{F}_{\text{Tree}}\ A \to A$.

What is "algebraic data" about these types?

Let us consider two such algebras as examples:

- For $\mathbf{F}_{\mathsf{Nat}}$, an algebra with carrier $\mathbb{B} = \{ff, tt\}$:

$$\begin{aligned} alg_1 : \mathsf{Zero} &\mapsto ff \\ (\mathsf{Succ}, b) &\mapsto \neg b \end{aligned}$$

- For $\mathbf{F}_{\mathsf{Tree}}$, an algebra with carrier $\mathbb{Z}$:

$$\begin{aligned} alg_2 : \mathsf{Leaf} &\mapsto 1 \\ (\mathsf{Node}, z_1, z, z_2) &\mapsto z_1 * z * z_2 \end{aligned}$$

Between two algebras for the same functor, there is a notion of homomorphism:

$$\begin{array}{ccc} B & \xleftarrow{g} & \mathsf{F}B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{F}h} \\ A & \xleftarrow[f]{} & \mathsf{F}A \end{array}$$

What is "algebraic data" about these types?

Beside the $\mathbf{F}_{\mathsf{Nat}}$-algebra alg_1 with carrier $\mathbb{B}$ and

$$\begin{aligned} alg_1 : \mathsf{Zero} &\mapsto \mathit{ff} \\ (\mathsf{Succ}, b) &\mapsto \neg b \end{aligned}$$

let us also consider the $\mathbf{F}_{\mathsf{Nat}}$-algebra alg_3 with carrier $\mathbb{Z}$ and

$$\begin{aligned} alg_3 : \mathsf{Zero} &\mapsto 0 \\ (\mathsf{Succ}, z) &\mapsto z + 2 \end{aligned}$$

Exercise:

How many algebra homomorphisms are there

- ... from alg_1 to alg_3?
- ... from alg_3 to alg_1?

What is "algebraic data" about these types?

Beside the $\mathbf{F}_{\mathsf{Tree}}$-algebra alg_2 with carrier $\mathbb{Z}$ and

$$\begin{aligned} alg_2 : \mathsf{Leaf} &\mapsto 1 \\ (\mathsf{Node}, z_1, z, z_2) &\mapsto z_1 * z * z_2 \end{aligned}$$

let us also consider the $\mathbf{F}_{\mathsf{Tree}}$-algebra alg_4 with carrier $\mathbb{B}$ and

$$\begin{aligned} alg_4 : \mathsf{Leaf} &\mapsto tt \\ (\mathsf{Node}, b_1, z, b_2) &\mapsto b_1 \wedge isOdd(z) \wedge b_2 \end{aligned}$$

Exercise:

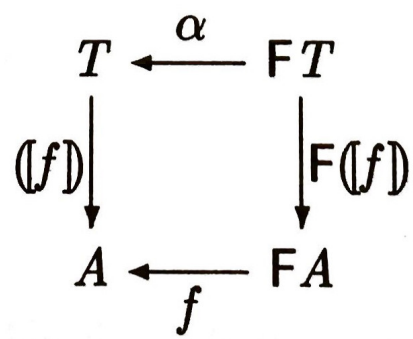
How many algebra homomorphisms are there

- ... from alg_2 to alg_4?
- ... from alg_4 to alg_2?

What is "algebraic data" about these types?

Since identity arrows are homomorphisms and the composition of two homomorphisms is again a homomorphism, the algebras for a functor **F** form a category, **Alg**(**F**).

Often, in particular for polynomial functors in **Set**, this category has an initial object, that is, an **F**-algebra α such that for each **F**-algebra there is exactly one algebra homomorphism from α to it.



In the setting considered here, the carrier of the initial algebra can always be obtained (up to isomorphism) from the following construction:

$$T = \mathbf{F}\varnothing \cup \mathbf{F}\left(\mathbf{F}\varnothing\right) \cup \mathbf{F}\left(\mathbf{F}\left(\mathbf{F}\varnothing\right)\right) \cup \cdots$$

What is "algebraic data" about these types?

Let us instantiate this for $\mathbf{F}_{\mathsf{Nat}}$:

$$\begin{aligned} T_{\mathsf{Nat}} &= \mathbf{F}_{\mathsf{Nat}}\,\varnothing \cup \mathbf{F}_{\mathsf{Nat}}\,(\mathbf{F}_{\mathsf{Nat}}\,\varnothing) \cup \mathbf{F}_{\mathsf{Nat}}\,(\mathbf{F}_{\mathsf{Nat}}\,(\mathbf{F}_{\mathsf{Nat}}\,\varnothing)) \cup \cdots \\ &= \{\mathsf{Zero}\} \cup \mathbf{F}_{\mathsf{Nat}}\,(\{\mathsf{Zero}\}) \cup \mathbf{F}_{\mathsf{Nat}}\,(\mathbf{F}_{\mathsf{Nat}}\,(\{\mathsf{Zero}\})) \cup \cdots \\ &= \{\mathsf{Zero}, (\mathsf{Succ}, \mathsf{Zero})\} \cup \mathbf{F}_{\mathsf{Nat}}\,(\{\mathsf{Zero}, (\mathsf{Succ}, \mathsf{Zero})\}) \cup \cdots \\ &= \{\mathsf{Zero}, (\mathsf{Succ}, \mathsf{Zero}), (\mathsf{Succ}, (\mathsf{Succ}, \mathsf{Zero}))\} \cup \cdots \\ &\vdots \end{aligned}$$

Modulo syntax, this corresponds to

$$\mathsf{Zero},\ \mathsf{Succ\ Zero},\ \mathsf{Succ\ (Succ\ Zero)},\ \ldots$$

as the values for the algebraic data type

$$\mathsf{data\ Nat} = \mathsf{Zero} \mid \mathsf{Succ\ Nat}$$

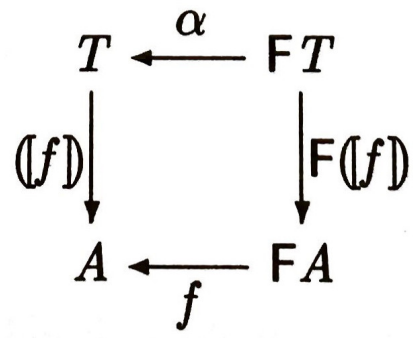
Similarly,

$$T_{\mathsf{Tree}} = \mathbf{F}_{\mathsf{Tree}}\,\varnothing \cup \mathbf{F}_{\mathsf{Tree}}\,(\mathbf{F}_{\mathsf{Tree}}\,\varnothing) \cup \mathbf{F}_{\mathsf{Tree}}\,(\mathbf{F}_{\mathsf{Tree}}\,(\mathbf{F}_{\mathsf{Tree}}\,\varnothing)) \cup \cdots$$

explains what the values are for the algebraic data type Tree.

What is “algebraic data” about these types?

But there is other data in



than just the T. In particular, to every algebra f we get a unique homomorphism from the initial algebra, called “catamorphism”.

Let us look at this for $\mathbf{F}_{\mathsf{Nat}}$ and our alg_1 of type $\mathbf{F}_{\mathsf{Nat}}\ \mathbb{B} \to \mathbb{B}$ with

$$alg_1\ :\ \mathsf{Zero} \mapsto \mathit{ff},\ (\mathsf{Succ}, b) \mapsto \neg b$$

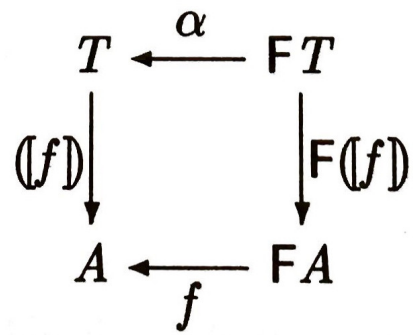
What is $(\!|alg_1|\!)$?

The unique k of type $T_{\mathsf{Nat}} \to \mathbb{B}$ such that $k \circ \alpha = alg_1 \circ \mathbf{F}_{\mathsf{Nat}}\ k$.

But that means we first have to work out what α of type $\mathbf{F}_{\mathsf{Nat}}\ T_{\mathsf{Nat}} \to T_{\mathsf{Nat}}$ is? Spoiler: It is syntactic identity!

What is "algebraic data" about these types?

So, to determine what $(\!|alg_1|\!)$ is, as the unique k of type $T_{\mathsf{Nat}} \to \mathbb{B}$ such that $k \circ \alpha = alg_1 \circ \mathbf{F}_{\mathsf{Nat}}\ k$, we can look at what the diagram



requires for elements of $T = T_{\mathsf{Nat}} = \mathbf{F}_{\mathsf{Nat}}\ T_{\mathsf{Nat}}$.

Recall that

$$\mathbf{F}_{\mathsf{Nat}}\ k\ :\ \mathsf{Zero} \mapsto \mathsf{Zero},\ (\mathsf{Succ}, a) \mapsto (\mathsf{Succ}, k(a))$$

and

$$alg_1\ :\ \mathsf{Zero} \mapsto ff,\ (\mathsf{Succ}, b) \mapsto \neg b$$

So we get:

$$k\ :\ \mathsf{Zero} \mapsto ff,\ (\mathsf{Succ}, a) \mapsto \neg k(a)$$

What is "algebraic data" about these types?

In Haskell syntax, we get that $(\!|alg_1|\!)$ is the following function:

$$
\begin{aligned}
&k :: \mathsf{Nat} \to \mathsf{Bool} \\
&k\ \mathsf{Zero} &&= \mathsf{False} \\
&k\ (\mathsf{Succ}\ a) &&= \mathit{not}\ (k\ a)
\end{aligned}
$$

Analogously, we get that $(\!|alg_3|\!)$ is the following function:

$$
\begin{aligned}
&k :: \mathsf{Nat} \to \mathsf{Integer} \\
&k\ \mathsf{Zero} &&= 0 \\
&k\ (\mathsf{Succ}\ a) &&= (k\ a) + 2
\end{aligned}
$$

What is "algebraic data" about these types?

And $(\!|alg_2|\!)$, for $\mathbf{F}_{\mathsf{Tree}}$ instead of $\mathbf{F}_{\mathsf{Nat}}$, is the following function:

$$
\begin{aligned}
&k :: \mathsf{Tree} \to \mathsf{Integer} \\
&k\ \mathsf{Leaf} &&= 1 \\
&k\ (\mathsf{Node}\ a_1\ z\ a_2) &&= (k\ a_1) * z * (k\ a_2)
\end{aligned}
$$

While $(\!|alg_4|\!)$ is the following function:

$$
\begin{aligned}
&k :: \mathsf{Tree} \to \mathsf{Bool} \\
&k\ \mathsf{Leaf} &&= \mathsf{True} \\
&k\ (\mathsf{Node}\ a_1\ z\ a_2) &&= (k\ a_1)\ \&\&\ (\mathit{isOdd}\ z)\ \&\&\ (k\ a_2)
\end{aligned}
$$

Indeed every structurally recursive function arises as a catamorphism.

For example, the function $\mathit{height} :: \mathsf{Tree} \to \mathsf{Integer}$ that appeared on the lecture slide is the catamorphism for the $\mathbf{F}_{\mathsf{Tree}}$-algebra mapping Leaf to 0 and $(\mathsf{Node}, z_1, \_, z_2)$ to $1 + \mathit{max}\ z_1\ z_2$.

Is any of this actually useful in programming?

Well, for one thing, the constructions can be expressed in Haskell itself, e.g.

$$\text{data F } a = \text{ZeroF} \mid \text{SuccF } a$$

as representation of

$$\mathbf{F}_{\text{Nat}}\ A = \{\,\text{Zero}\,\} \cup \{\,(\text{Succ}, a) \mid a \in A\,\}$$

and

$$\begin{aligned}
&\mathit{cata} :: (\text{F } a \to a) \to (\text{Nat} \to a)\\
&\mathit{cata}\ f = k\\
&\quad \text{where } k\ \text{Zero} \quad\ = f\ \text{ZeroF}\\
&\qquad\quad\ k\ (\text{Succ } a) = f\ (\text{SuccF } (k\ a))
\end{aligned}$$

as the realisation of $(\!|\cdot|\!)$.

Which does not, in this form, look particularly appealing. But if we recognise that F $a \to a$, for "data F a = ZeroF | SuccF a", is actually isomorphic to $(a, a \to a)$, then ...

Is any of this actually useful in programming?

. . . we arrive at this version:

$$
\begin{aligned}
&\mathit{cata} :: (a, a \to a) \to (\mathsf{Nat} \to a) \\
&\mathit{cata}\ (z, s) = k \\
&\quad \text{where } k\ \mathsf{Zero} \quad\ = z \\
&\qquad\quad\ \ k\ (\mathsf{Succ}\ a) = s\ (k\ a)
\end{aligned}
$$

and can then abbreviate

$$
\begin{aligned}
&k :: \mathsf{Nat} \to \mathsf{Bool} \\
&k\ \mathsf{Zero} \quad\ = \mathsf{False} \\
&k\ (\mathsf{Succ}\ a) = \mathit{not}\ (k\ a)
\end{aligned}
$$

to just $\mathit{cata}\ (\mathsf{False}, \mathit{not})$ — essentially $(\!|\mathit{alg}_1|\!)$, as well as abbreviate

$$
\begin{aligned}
&k :: \mathsf{Nat} \to \mathsf{Integer} \\
&k\ \mathsf{Zero} \quad\ = 0 \\
&k\ (\mathsf{Succ}\ a) = (k\ a) + 2
\end{aligned}
$$

to just $\mathit{cata}\ (0, \lambda z \to z + 2)$ — essentially $(\!|\mathit{alg}_3|\!)$.

Is any of this actually useful in programming?

Likewise, for "data Tree = Leaf | Node Tree Integer Tree" we have

$$
\begin{aligned}
&\mathit{cata} :: (a, a \to \mathsf{Integer} \to a \to a) \to (\mathsf{Tree} \to a) \\
&\mathit{cata}\ (l, n) = k \\
&\quad \text{where } k\ \mathsf{Leaf} = l \\
&\qquad\quad k\ (\mathsf{Node}\ a_1\ z\ a_2) = n\ (k\ a_1)\ z\ (k\ a_2)
\end{aligned}
$$

and can then abbreviate the three structurally recursive functions on Tree we saw, to just:

- $\mathit{cata}\ (1, \lambda z_1\ z\ z_2 \to z_1 * z * z_2)$ — for $(\!|alg_2|\!)$
- $\mathit{cata}\ (\mathsf{True}, \lambda b_1\ z\ b_2 \to b_1\ \&\&\ (\mathit{isOdd}\ z)\ \&\&\ b_2)$ — for $(\!|alg_4|\!)$
- $\mathit{cata}\ (0, \lambda z_1\ \_\ z_2 \to 1 + \mathit{max}\ z_1\ z_2)$ — for *height*

Moreover, it is not really necessary to explicitly program the different versions of *cata* for Nat, Tree, etc. The principles are so generic that the compiler can be made to support catamorphisms automatically for any algebraic data type (of a polynomial functor).

- Also remember the function

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

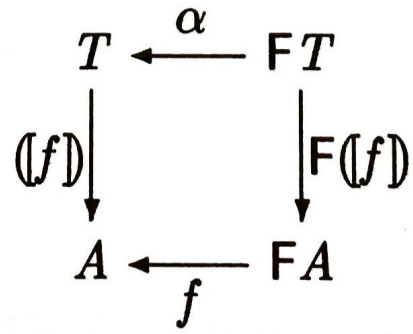
  which puts a (left-associative) function/operator between all elements of a non-empty list.

- It is a member of a whole family of related functions, the most prominent of which is `foldr`, defined thus:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ n []     = n
foldr c n (x:xs) = c x (foldr c n xs)
```

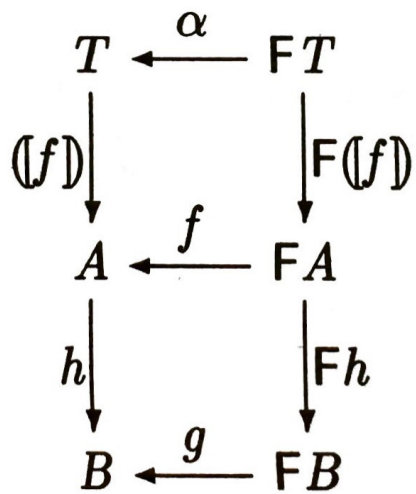
What about "lawful program construction"?

Recall:



The universal property $k = (\!|f|\!) \Leftrightarrow k \circ \alpha = f \circ \mathsf{F}\ k$ has further useful consequences.

In particular, there is the fusion law:



$$h \circ (\!|f|\!) = (\!|g|\!) \Leftarrow h \circ f = g \circ \mathsf{F}\ h$$

What about "lawful program construction"?

So, since *isOdd* was identified as an algebra homomorphism from alg_2 to alg_4, we know that $isOdd \circ (\!|alg_2|\!) = (\!|alg_4|\!)$.

Put differently, for functions

$$
\begin{aligned}
&product :: \mathsf{Tree} \to \mathsf{Integer} \\
&product\ \mathsf{Leaf} &&= 1 \\
&product\ (\mathsf{Node}\ a_1\ z\ a_2) &&= (product\ a_1) * z * (product\ a_2)
\end{aligned}
$$

and

$$
\begin{aligned}
&allOdd :: \mathsf{Tree} \to \mathsf{Bool} \\
&allOdd\ \mathsf{Leaf} &&= \mathsf{True} \\
&allOdd\ (\mathsf{Node}\ a_1\ z\ a_2) &&= (allOdd\ a_1)\ \&\&\ (isOdd\ z)\ \&\&\ (allOdd\ a_2)
\end{aligned}
$$

it holds: $isOdd \circ product = allOdd$.

This can be used for proving properties, or for deriving programs from specifications, or for optimising the efficiency of programs, . . . , for explicitly recursive functions or for functions expressed as catamorphisms, . . . , generically for different algebraic data types.

Some comments on duality

All we have seen can be dualised:

- coalgebras, arrows of types like $A \to \mathbf{F}_{\mathsf{Nat}}\ A$ and $A \to \mathbf{F}_{\mathsf{Tree}}\ A$
- coalgebra homomorphisms
- category of coalgebras
- the final coalgebra for a functor — What will be the carrier?
- unique homomorphisms to the final coalgebra, called "anamorphisms" or "unfolds"
- a fusion law

Concretely, functions like

$$
\begin{aligned}
&k :: \mathsf{Integer} \to \mathsf{Nat} \\
&k\ 1 = \mathsf{Zero} \\
&k\ z = \mathsf{Succ}\ (k\ (\mathit{div}\ z\ 2))
\end{aligned}
$$

and

$$
\begin{aligned}
&k :: (\mathsf{Integer}, \mathsf{Integer}) \to \mathsf{List\ Integer} \\
&k\ (a, b) = \mathsf{Cons}\ a\ (k\ (b, a + b))
\end{aligned}
$$

More direct appearances of functors

Data types can be polymorphic, that is, instead of

$$\text{data Tree} = \text{Leaf} \mid \text{Node Tree Integer Tree}$$

we can have

$$\text{data Tree } a = \text{Leaf} \mid \text{Node (Tree } a\text{) } a \text{ (Tree } a\text{)}$$

That is an action on objects in **Set**. Is there also a corresponding action on arrows that turns Tree into a functor?

Indeed, laws like

$$(\mathit{treeMap}\ f) \circ (\mathit{treeMap}\ g) = \mathit{treeMap}\ (f \circ g)$$

hold, and can also be used for program calculation etc.

Also, as another example of a function we have used:

```
map :: (a -> b) -> [a] -> [b]
map h []     = []
map h (x:xs) = h x : map h xs
```

And indeed related:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap h Leaf = Leaf
treeMap h (Node left x right)
  = Node (treeMap h left)
         (h x)
         (treeMap h right)
```

- **Another useful function:**

```
map :: (a -> b) -> [a] -> [b]
```

which applies a function to all elements of a list.

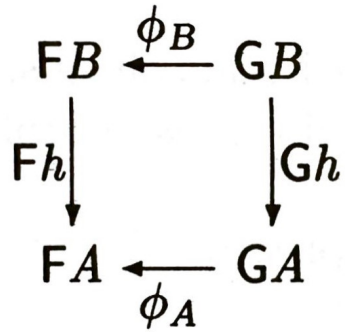
- **For example:**

```
map even [1..10]

map (dilated 5) [ pic1, pic2, pic3 ]
```

Semantic consequences of polymorphism

Moreover, functions of polymorphic type satisfy interesting laws, having to do with the concept of natural transformation:



which we will look at in a moment.

Of course we do not in the lecture, but what we do consider there are instances in disguise . . .

Contrast to for-loops in Java, C, etc.

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- In contrast, it is not remotely true that in an imperative language we can always replace a piece of code written like this:

```
for (a = 0; a <= n; a++)
   result[a] = h(a);
```

  by this:

```
for (a = n; a >= 0; a--)
   result[a] = h(a);
```

- And even for the cases where commands as above <u>are</u> equivalent, a formulation given that way is less useful than the Haskell equation we saw, or indeed its more general version:

```
[ h a | a <- reverse list ]
≡ reverse [ h a | a <- list ]
```

Consequences of polymorphic types

- Polymorphism has really interesting semantic consequences.
- For example, in the lecture last week, I mentioned that always:

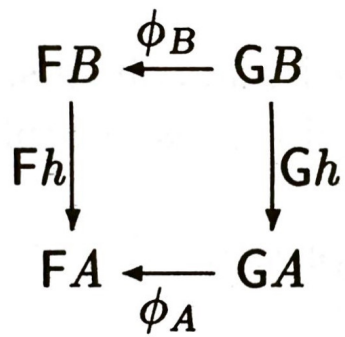
```
    [ h a | a <- reverse list ]
≡   reverse [ h a | a <- list ]
```

- What if I told you that this holds, for arbitrary `h` and `list`, not only for `reverse`, but for any function with type `[a] -> [a]`, no matter how it is defined?
- Can you give some such functions (and check the above claim)?

Semantic consequences of polymorphism

So what is it that *reverse* and other functions of type $[a] \to [a]$ have in common?

A natural transformation, between two functors **F** and **G** (both of the same kind, in terms of source and target categories), is an indexed collection of arrows ϕ_A of types $\mathbf{G}\,A \to \mathbf{F}\,A$ such that for every arrow h (in the source category) this diagram commutes:



Let us instantiate this for $\phi = \mathit{reverse}$ and $\mathbf{F} = \mathbf{G} = [\cdot]$.

The condition becomes that for every function h of type B $\to A$ it holds:

$$\mathit{map}\ h \circ \mathit{reverse}_B = \mathit{reverse}_A \circ \mathit{map}\ h$$

Polymorphic functions on lists

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- We have already seen a lot of functions that fit this pattern:

```
head   :: [a] -> a
tail   :: [a] -> [a]
last   :: [a] -> a
init   :: [a] -> [a]
length :: [a] -> Int
null   :: [a] -> Bool
concat :: [[a]] -> [a]
```

- In concrete applications, the type variable gets instantiated appropriately: `head "abc" :: Char`.

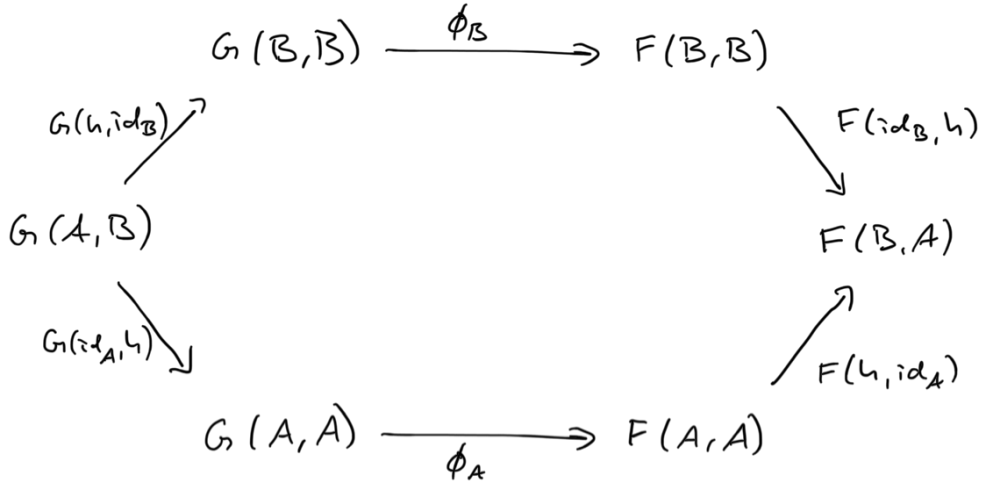
Semantic consequences of polymorphism

However, not every polymorphic function can be made to fit the pattern $\mathbf{G}\,A \to \mathbf{F}\,A$ for some functors $\mathbf{F}$ and $\mathbf{G}$.

For example, consider *filter* :: $(a \to \text{Bool}) \to [a] \to [a]$.

What is some naturality-like property that every function of *filter*'s type satisfies?

A dinatural transformation, between two bifunctors $\mathbf{F}$ and $\mathbf{G}$ of the same mixed contravariant/covariant kind over the same source category, is an indexed collection of arrows ϕ_A of types $\mathbf{G}\,(A, A) \to \mathbf{F}\,(A, A)$ such that for every h this diagram commutes:



- And another one:

```
filter :: (a -> Bool) -> [a] -> [a]
```

  which selects list elements that satisfy a certain predicate.

- For example,

```
filter isPalindrome completeDictionary

filter (> 0.5) bonusPercentageList
```

Relationship to list comprehensions

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- While the following are not the actual definitions of `map` and `filter`, we can think of them as such:

```
map :: (a -> b) -> [a] -> [b]
map h list = [ h a | a <- list ]

filter :: (a -> Bool) -> [a] -> [a]
filter p list = [ a | a <- list, p a ]
```

- Conversely, every list comprehension expression, no matter how complicated with several generators, guards, etc., can be implemented via `map`, `filter`, and `concat`.

Expressing laws

- Also, a law like (mentioned earlier):

```
[ h a | a <- reverse list ]
≡ reverse [ h a | a <- list ]
```

  can nicely be expressed as:

```
map h . reverse ≡ reverse . map h
```

- Then we can also ask under which conditions this holds:

```
map h . filter p ≡ filter q . map h
```

- Generally, higher-order functions are a boon for “lawful program construction” (see the Richard Bird quote).

Semantic consequences of polymorphism

Let us instantiate this for $\phi = \mathit{filter}$. There are actually at least two ways to consider the type $(a \to \mathrm{Bool}) \to [a] \to [a]$ as something like $\mathbf{G}\,(A, A) \to \mathbf{F}\,(A, A)$ for bifunctors of the required kind.

One possibility is:

- $\mathbf{G}\,(X, Y) = \mathit{Hom}(X, \mathrm{Bool})$
- $\mathbf{F}\,(X, Y) = \mathit{Hom}([X], [Y])$

Now we can take an element q of $\mathbf{G}\,(A, B) = \mathit{Hom}(A, \mathrm{Bool})$, that is, a function $q :: A \to \mathrm{Bool}$, and chase it around the diagram (while taking into account that the action of Hom on arrows is that $\mathit{Hom}(f, g)$ is the function $k \mapsto g \circ k \circ f$) . . . on the whiteboard.