GPU Lab

Functional Programming

or applied deep magic

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

λ

D. Berényi – M. F. Nagy-Egri

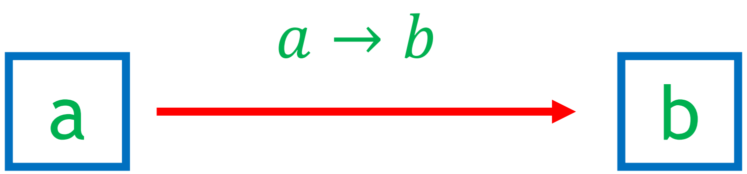
Functor

A type that has the Functor property comes with a function:

$$(a \to b) \to F\ a \to F\ b$$

called `fmap`.

fmap takes a function, and a value with a context and applies the function inside that context. Graphically:



Functor

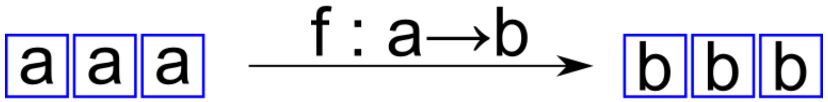
Almost everything in programming is
(or could be in C++) a functor...

- All containers like `std::vector`
- Many helper objects like `std::future`

λ

Functor

Especially with containers, you may want to think of `map` as the following:



Example:

$F(a)$:

```
std::vector<int>
```

$f : a \to b$:

```
std::string f(int x)
{
        return std::to_string(x);
}
```

$F(b)$:

```
std::vector<std::string>
```

Functor

map must satisfy the following relations:

$$\text{map id} = \text{id}$$

mapping the identity does nothing

$$\text{map } (f \circ g) = (\text{map } f) \circ (\text{map } g)$$

map is distributive with composition

Functor

The map is an interface of a type template, like `std::vector<T>`.

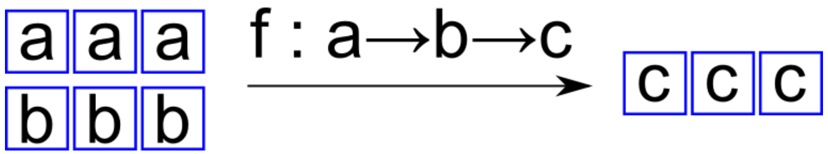
In C++ map may be implemented for `std::vector<T>` like this:

```
template<typename Func, typename A,
         typename B = typename std::result_of<Func(A)>::type>
std::vector<B> map( Func f, std::vector<A> const& va )
{
   std::vector<B> res(va.size());
   std::transform(va.cbegin(), va.cend(), res.cbegin(), f);
   return res;
}
```

Functor

A useful generalization is to extend map to multiple arguments:

This is called zip:



$$(a \to b \to c) \to F\ a \to F\ b \to F\ c$$

It takes a binary function, two containers, and return one container.

Functor, Applicative, Monad

Two other similar concepts, extending the construction:

Functor's map was: $\boxed{a} \xrightarrow{f\,:\,a\to b} \boxed{b}$

Applicative:
the function is also in a context: $\boxed{a} \xrightarrow{\boxed{f\,:\,a\to b}} \boxed{b}$

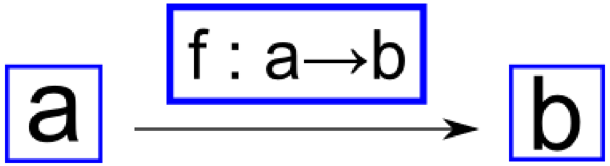
Monad:
The function returns a context: $\boxed{a} \xrightarrow{f\,:\,a\to \boxed{b}} \boxed{b}$

λ

Applicative

Applicative extends Functor with:

```
pure:   a -> F a
apply: F (a->b) -> F a -> F b
```



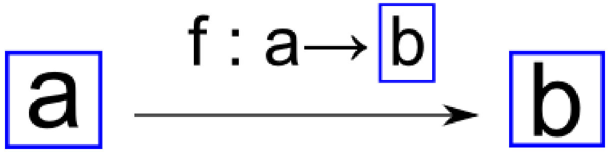
pure packages a pure value inside such a context.

Without applicative's `apply` (that is an extension of `map`), it is not possible to apply a packed function to the packed argument

Monad

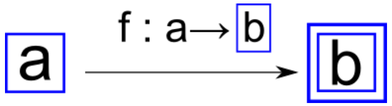
Monad extends Applicative with:

```
return: a -> F a
bind: F a -> (a -> F b) -> F b
```



`return` is an analogue of Applicative's `pure`:
it packages a pure value inside a context.

`bind` extends apply such that
when the function get applied inside the context,
the result will <u>not</u> get double packaged like this:



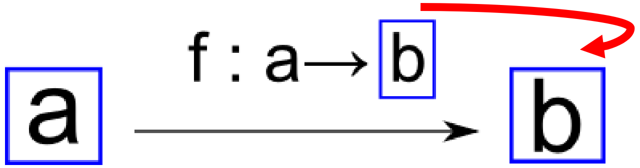
Functional Programming

λ

Monad

Applicatives can be chained (not like Functors)
but they always get evaluated.

Monads can be chained too, but the functions have the possibility to short-circuit the evaluation
by choosing the state of the context it returns!



Monad

Haskell's complete I/O system is built using Monads.

The monadic state represents e.g. the state of a keyboard, and the functions that extract characters return new context each time that represents the new state (e.g. key pressed).

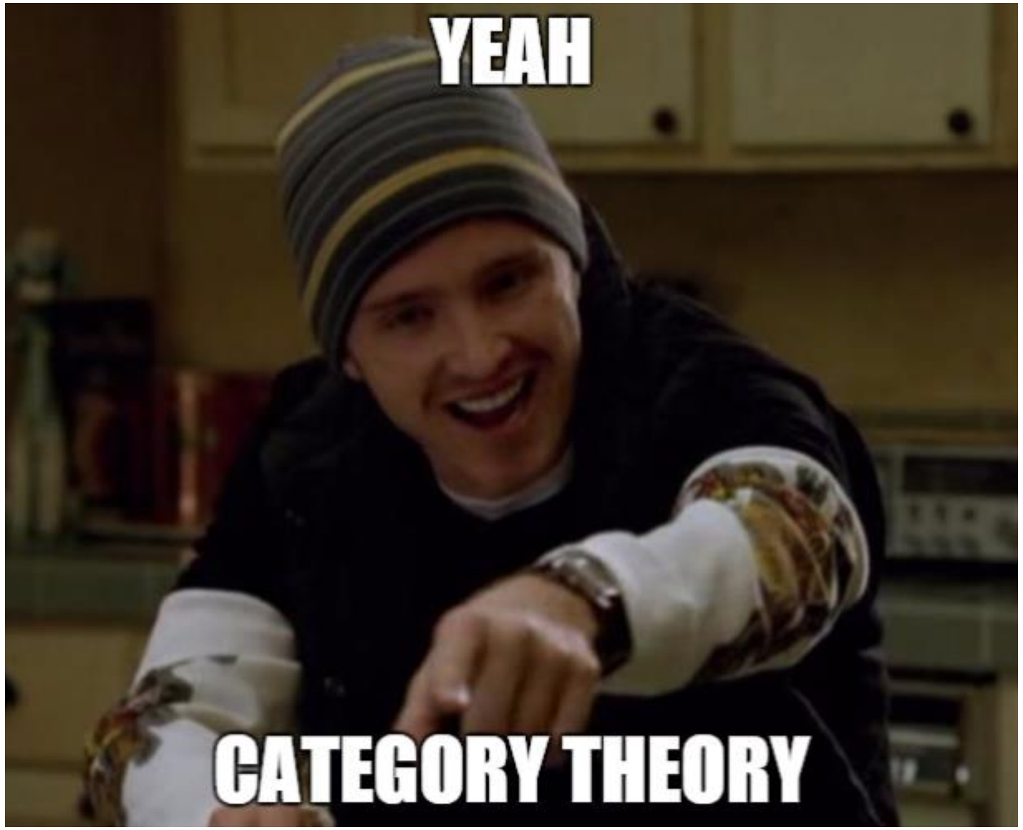
Functor, Applicative, Monad

Sum and Product type combinators and composition interacts with these concepts as follows:

- Functor preserves all three
  (sum of functors is a functor, same for prod and comp.)
- Applicative only closed for product and composition
- Monad is only closed for product types.

Functor, Applicative, Monad

Where these constructs and algebraic properties come from?



Category theory... It works!

Functor, Applicative, Monad

An intuitive explanation if you still need it:

[Functor, Applicative, Monad explained]

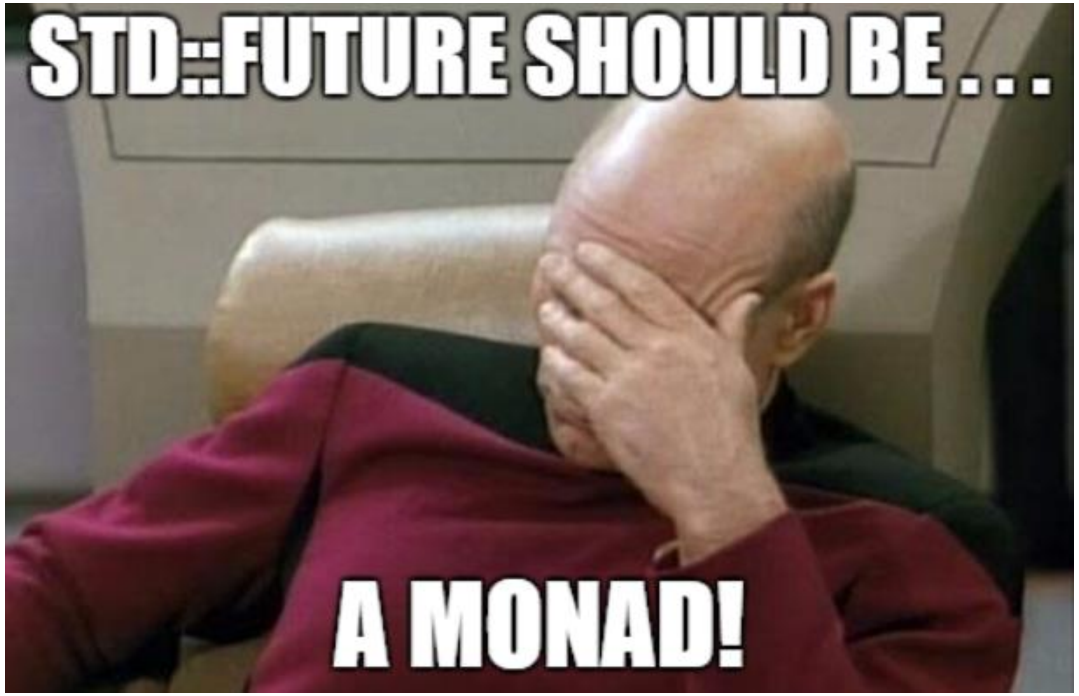
„How to learn about Monads:

1. Get a PhD in computer science.
2. Throw it away because you don’t need it for this section!”

Functor, Applicative, Monad

It is sad, that functional programming is completely alien in the C++ committee.

This leads to awkward situations, like reinventing the wheel in a slow and painful manner…



[Link](), [video]()

More...

This was just the tip of the iceberg...
We've not even implemented a linear algebra library so far ☺

Foldable

The Foldable concept necessities a following function on a type (container):

$$(a \to b \to a) \to a \ \to F\ b \to a$$

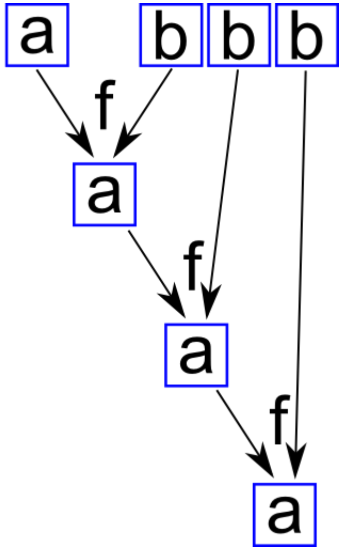
Called `foldl`.

What it is good for?

Foldable

`foldl:` $(a \to b \to a) \to a \to F\ b \to a$

f : a→b→a



`foldl` takes a binary function f,
a zero element of type a,
and a container of types b,
and aggregates the container by the
repeated application of f.

Foldable

`foldl`: $(a \to b \to a) \to a \ \to F\, b \to a$

`foldl` can be used to calculate sums, products, min, max, avg, etc. on containers.

The signature in C++ would be like:

```
template<typename Func,
         typename A,
         typename B>
A foldl( Func f, A zero, std::vector<B> const& fa );
```

usage:

```
sum: foldl( [](B a, B b){ return a+b; }, (B)0, fa );
```

Foldable

Note: it can be observed, that the requirements are a slight generalizations of a monoid!

`foldl:` $(a \to a \to a) \to a \ \to F\ a \to a$

This is the unit element of the monoid

This is the monoid binary operation

Foldable

Note: it can be observed, that the requirements are a slight generalizations of a monoid!

So we could say also that:

`foldl:` $M\,a \rightarrow F\,a \rightarrow a$

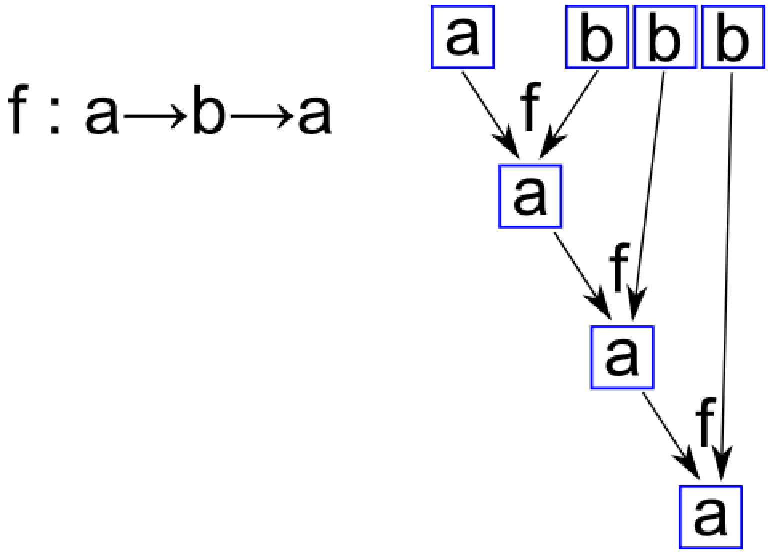
Where M is a monoid structure over type a
(the analogue of the underlying set)

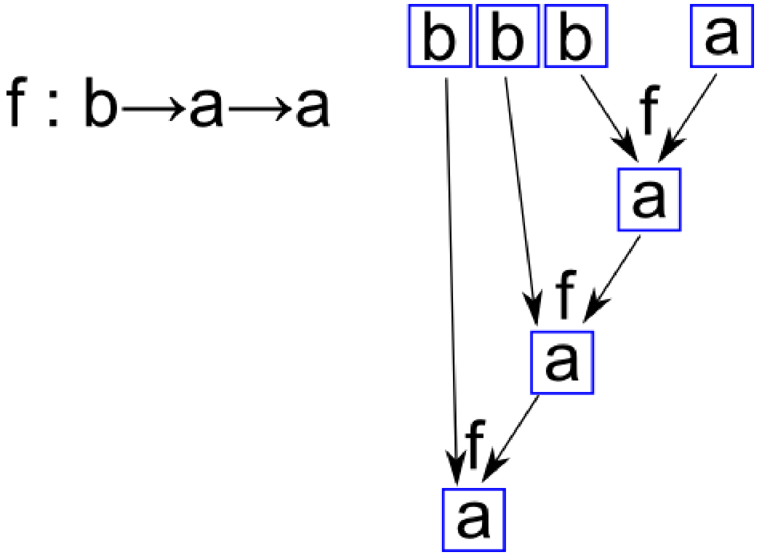
Foldable

Note 2.: why is it called foldl? Because there is a foldr!

`foldl:` $(a \to b \to a) \to a \ \to F\ b \to a$ inspects from left

`foldr:` $(b \to a \to a) \to a \ \to F\ b \to a$ inspects from right





Map, Fold, Zip

Why this is good for us?

We can implement the basic linear algebra operations (and much more) by using just map, fold, zip

- add/subtract matrices, vectors? It's just a zip
- multiply, divide by scalars? It's just a map
- dot product? `foldl (+) 0 (zip (*) u v)`
- matrix-multiplication?
  View the matrix as a container of rows/cols, that have map, fold, zip, and then it is simply a map of dot.

And now, category theory kicks in

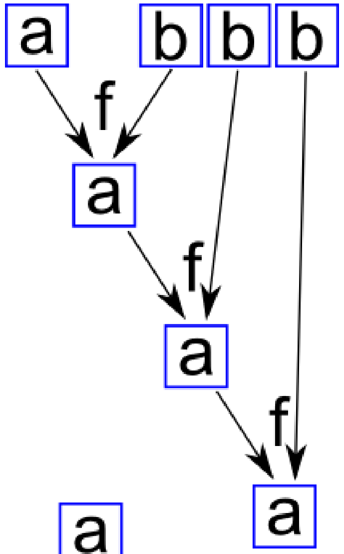
In category theory,
concepts come in pairs…

And pairs of pairs ☺

Unfolds

Things can be reversed and looked the other way.
Consider folds:

f : a→b→a

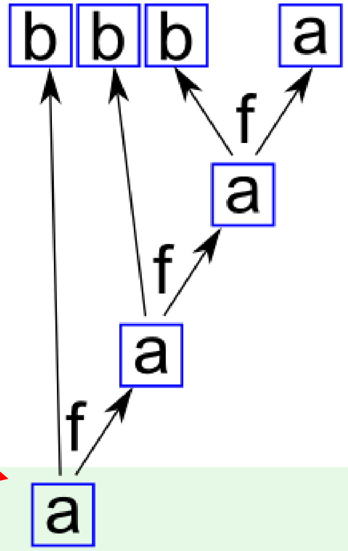


`foldl`: $(a \to b \to a) \to a \to F\ b \to a$
consumes from left

Dual:

`unfoldr`: $(a \to (b,\ a)) \to a \to (F\ b,\ a)$
produces to right

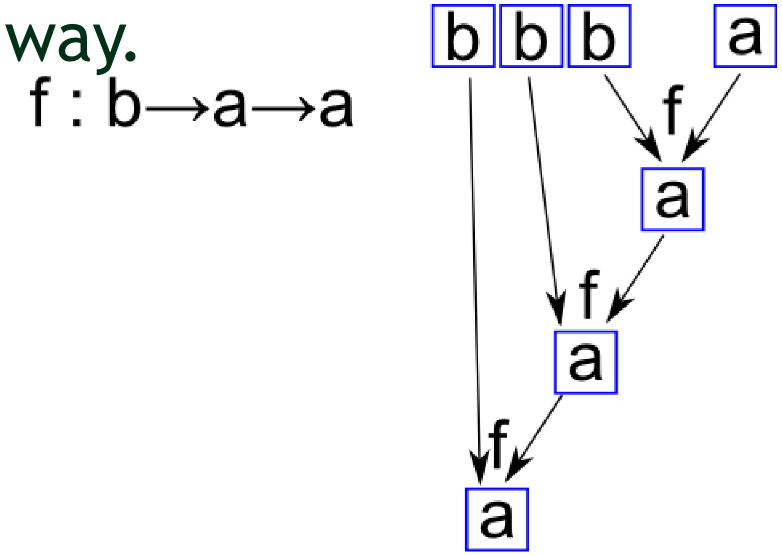
f : a→(b, a)



The zero element changes role:
it will be called „the seed”

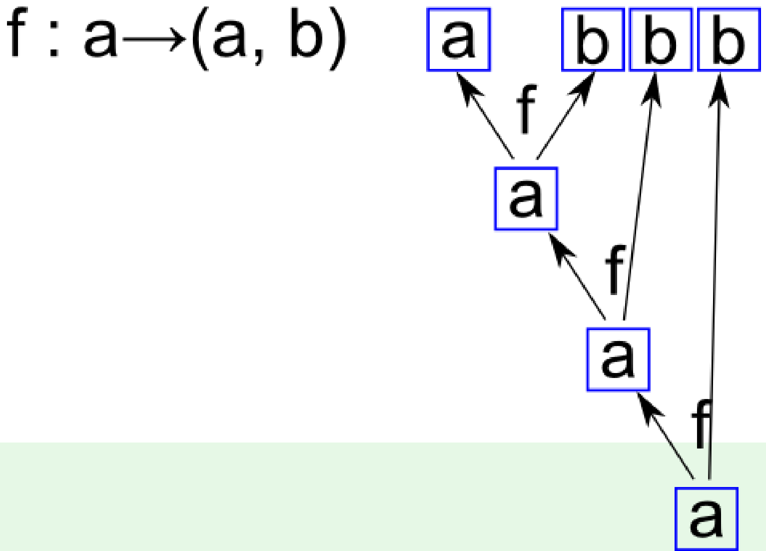
Unfolds

Things can be reversed and looked the other way.
Consider folds:



`foldr:` $(b \to a \to a) \to a \ \to F\ b \to a$
consumes from right

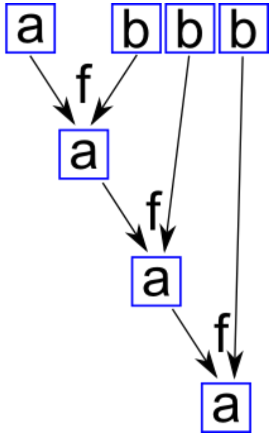
Dual:



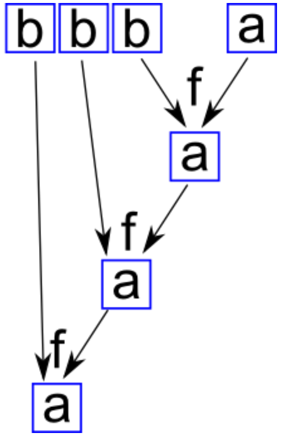
`unfoldl:` $(a \to (a,\ b)) \to a \ \to (a,\ F\ b)$
produces to left

Folds, Unfolds

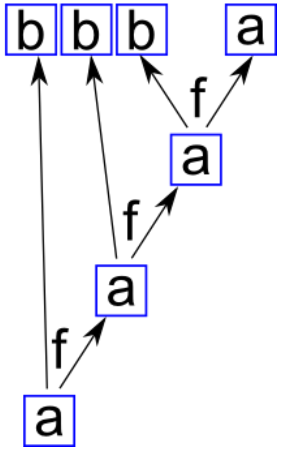
f : a→b→a



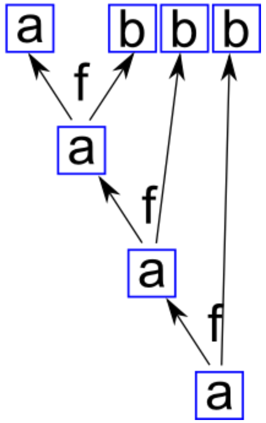
f : b→a→a



f : a→(b, a)



f : a→(a, b)



λ

Folds, Unfolds

Nice, but there is a problem... The termination guarantees are also changed:

folds finish, when the container is exhausted.

When unfolds finish?

λ

Folds, Unfolds

Nice, but there is a problem... The termination guarantees are also changed:

folds finish, when the container is exhausted.

When unfolds finish? **NEVER!**

Unfolds

Unfolds generate infinite structures.
You cannot ever ask for the whole structure.

However, we always need a finite portion of it at the end...

Remember Lazy Evaluation? We can have unfolds in the expression trees, until we do not take all of their values in the end.

Usually there is a function like `take`, that retrieves the first n elements of a structure. If we compose this somewhere after an unfold, we are safe.

Unfolds

What are unfolds good for?

Well, they are usually the ones that create the vectors, matrices etc. for you (except when you write out all the elements yourself)

- Initialize to a specific number all items in container
- Read data from file and construct a container from it
- Generate sequence of elements
- Generate random numbers
- Etc.

In fold, we always did the same thing at each step.

What if we could do different things at each application where we need to apply a binary function?

Hell starts to break loose...

λ

Catamorphisms

Catamorphisms generalize fold:

- Instead of a simple container,
  we have a recursive datatype (call it tree)
- Instead one function we have n
- Instead of one type in the container, we have a Sum type (!) of n elements.

And we have:

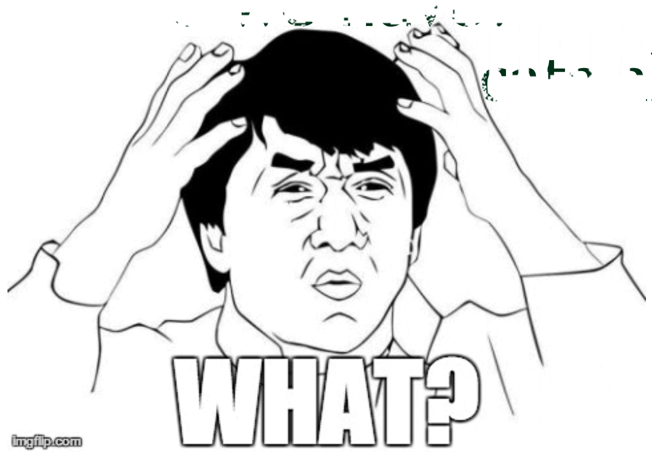
$$\text{cata alg tree} = \text{alg} \circ \text{map (cata alg)} \circ \text{unfix tree}$$

What???

Catamorphisms

Catamorphisms generalize fold:

- Instead of a simple container,
  we have a recursive datatype (call it tree)
- Instead one function we have n
- Instead of one type in the container, we have a Sum type (!) of n elements.



$$\text{cata alg tree} = \text{alg} \circ \text{map (cata alg)} \circ \text{unfix tree}$$

Catamorphisms

Simple recursive datatype: a List of type a elements.
(think of it as a template with 1 parameter: typename a)

A list may be empty:	`List a = Empty`	or
may contain 1 element:	`a`	or
may contain 2 elements:	`a, a`	or ...
may contain 3 elements:	`a, a, a`	etc...

It looks like a recursion, isn't it?

λ

Catamorphisms

Simple recursive datatype: a List of type a elements.

A list may be empty:	`List a = Empty`	or
may contain 1 element:	`a`	or
may contain 2 elements:	`a, a`	or ...
may contain 3 elements:	`a, a, a`	`etc...`

Think of it as tuples

It looks like a recursion, isn't it?

Let's do the same thing, like we did for the factorial:
A new argument will represent the recursive symbol, and we'll use the Y-combinator to tie the knot.

Catamorphisms

Simple recursive datatype: a List of type a elements.

Self!

A list may be empty: `List_proto self a = Empty  or`
may contain 1 element of a and a type s: `a, self`

And now: List a = Y (List_proto self a)

Writing out the recursion:

Think of it as a pair:
(a, self)

Empty	
a, Empty	self = Empty
a, (a, Empty)	self = List_proto Empty a
a, (a, (a, Empty))	self = List_proto (List_proto Empty a) etc...

Catamorphisms

Recursion may happen at the type level, thus we can describe repeating infinite structures, like trees.

These can be defined by creating a parametric (templated) non recursive type, and making it recursive by applying the fix-point combinatory on it.

Catamorphisms

Now we have trees, what do we do on them?

The recursive type has a condition hidden in the sum type:

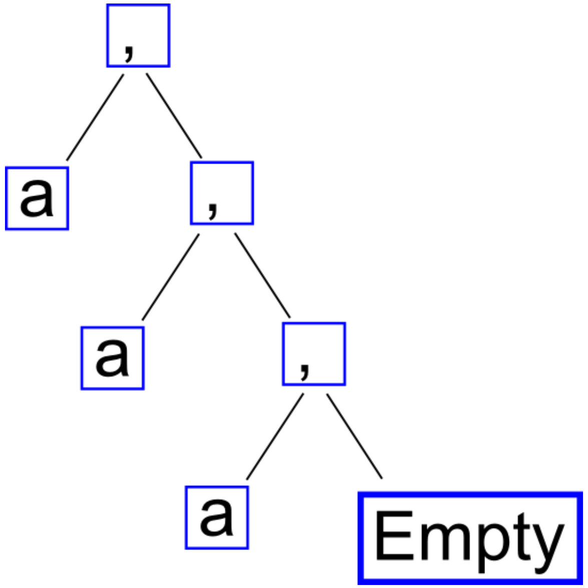
List a = Empty *or* (a, self)

When doing whatever on a tree, we should be prepared to handle both cases: in this case two functions are needed.

But! To be meaningful, they must return the same type!

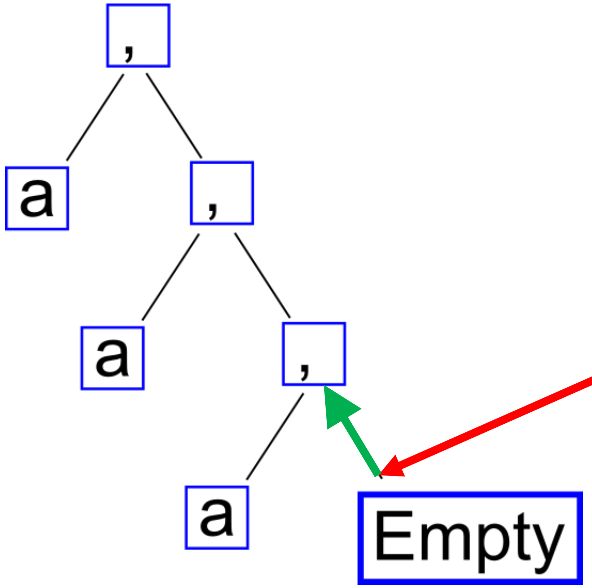
Catamorphisms

Consuming the structure:



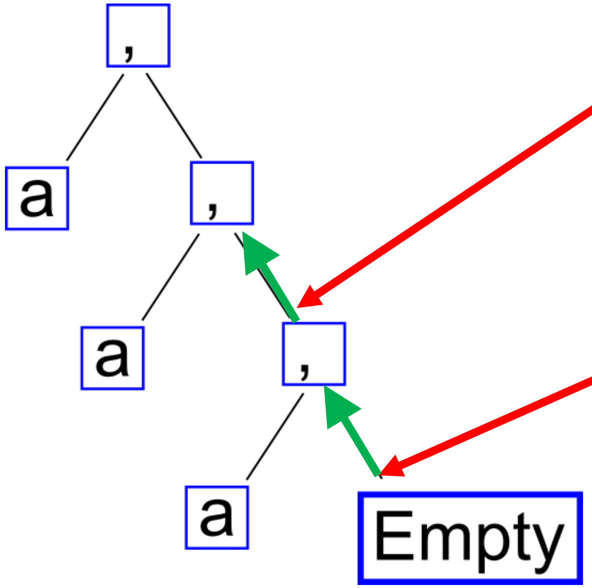
Catamorphisms

Consuming the structure:



Function 1
returning something on Empty case

Consuming the structure:

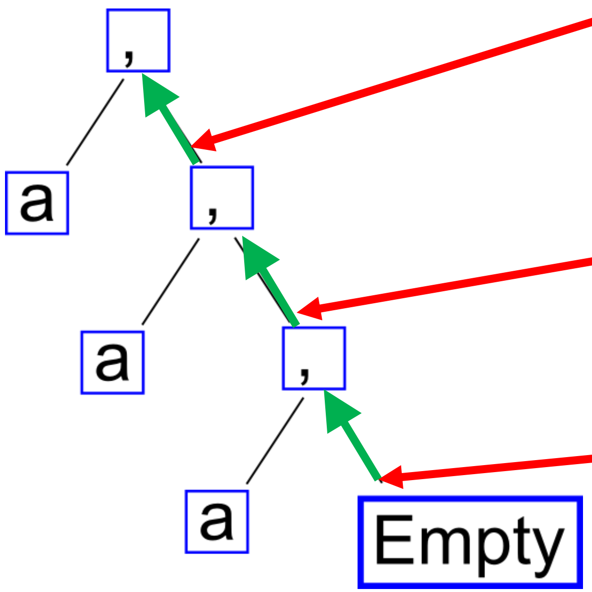


Function 2
returning something on (a, self) case
But the self part is already processed by Function 1.

Function 1
returning something on Empty case

Catamorphisms

Consuming the structure:



Again Function 2
returning something on (a, self) case
But the self part is already processed by Function 2.

Function 2
returning something on (a, self) case
But the self part is already processed by Function 1.

Function 1
returning something on Empty case

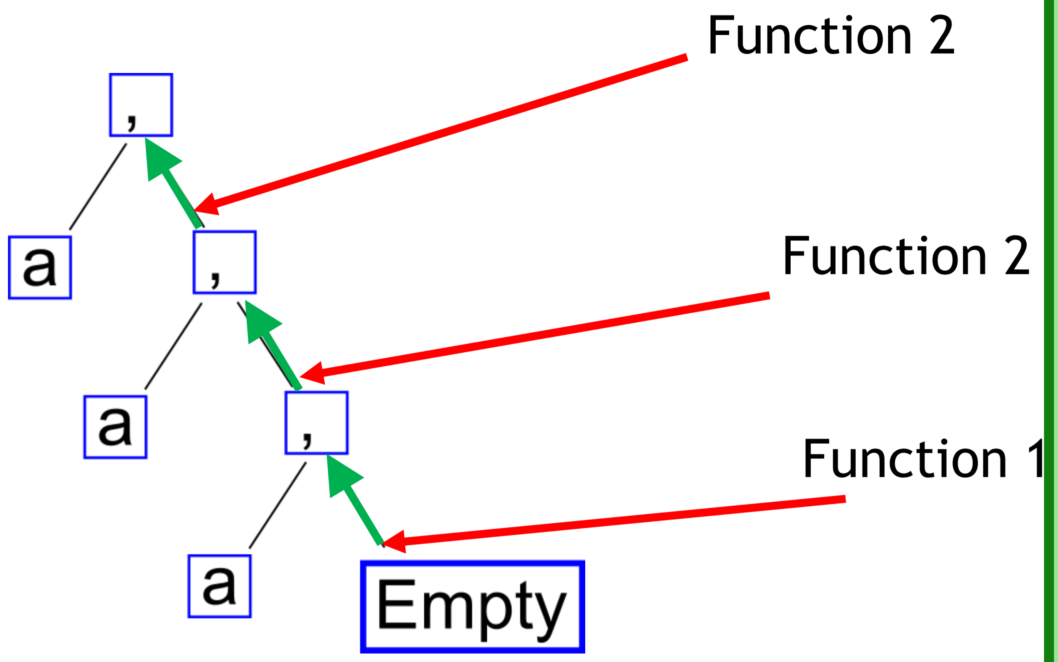
Catamorphisms

Consuming the structure:

The concept is known in Category Theory as F-Algebras

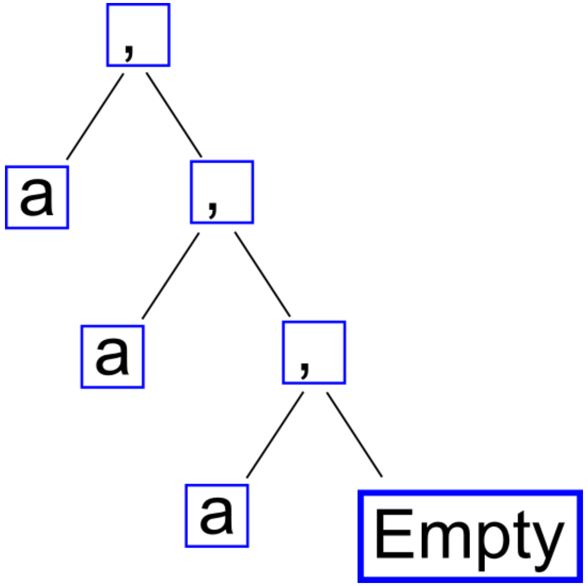
The set of functions that handle the different cases in the Sum type, is called *algebra*

The common return type that the functions share is called the *carrier type* of the algebra.



Catamorphisms

Consuming the structure:



The function, that takes a recursive type, and an algebra and consumes the structure by applying the algebra recursively is called:

Catamorphism:

The algebra

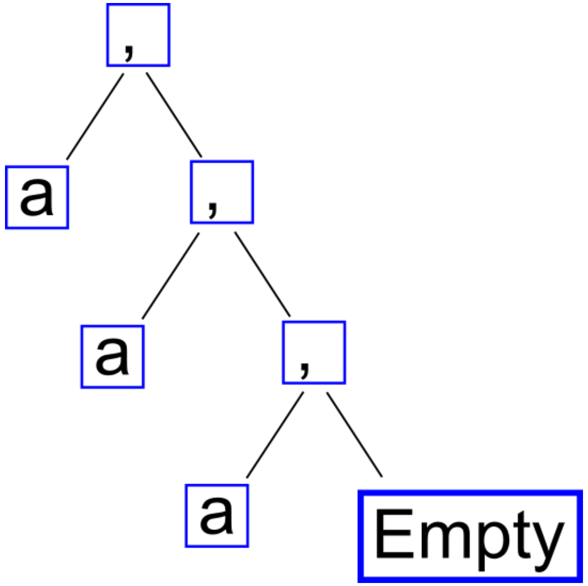
The recursive type

$$\text{cata alg tree} = \text{alg} \circ \text{map (cata alg)} \circ \text{unfix tree}$$

Catamorphisms

Catamorphism:

$$\text{cata alg tree} = \text{alg} \circ \text{map (cata alg)} \circ \text{unfix tree}$$



Of course,
cata returns
the carrier type

Evaluate the current level

Recursively evaluate the deeper levels

Reveal the type of the structure one level deeper

Catamorphisms

$$\text{cata alg tree} = \text{alg} \circ \text{map (cata alg)} \circ \text{unfix tree}$$

Not going into the details the following can be implemented in C++

```
template<typename A, template<typename> F>
auto cata(A algebra, Fix<F> tree)->typename A::Carrier
{
  return algebra( map( [&]( auto subtree )
  {
     return cata<A, F>(algebra, subtree);
  }, unfix(tree) ) );
}
```

This *is* actual code!

Functional Programming

Catamorphisms

What can we do with catamorphisms?

Catamorphisms

What can we do with catamorphisms?

Evaluate and transform any tree-like recursive structure!

Stuff like:

- Evaluating, transforming expression trees
- Writing trees to streams (file, network)
- Everything that folds can do, but much much more.

  In fact: folds are the special cases of catamorphisms, where the algebra has only one element.

Catamorphisms

The Boost library has a part, called Boost proto, that implements embedded domain specific languages inside C++.
It is a large and complex C++ library.

It was noted by Bartosz Milewski, that it may be significantly simplified by using F-algebras. The main developer of Boost Proto, Eric Niebler is credited by doing the first implementation in C++.

We managed to simplify it even more, down to around 200 lines.

Catamorphisms

Oh, and there is more!
Catamorphisms compose (in a special, but useful case)!

An algebra

$$f\colon F\ a \ \rightarrow \ a$$

A tree transformation

$$h\colon G\ a \ \rightarrow \ F\ a$$

$$\mathrm{cata\ f} \circ \mathrm{cata\ (fix} \circ \mathrm{h)} = \mathrm{cata\ (f} \circ \mathrm{h)}$$

This makes it possible to merge two traversals of the tree into a single one. This is widely used in optimization passes in functional compilers.
There are also compositions of algebras...

λ

More morphisms

What about duals?

The dual of catamorphism is an anamorphism:

$$\text{ana coalg seed} = \text{fix} \circ \text{map (ana coalg)} \circ \text{coalg seed}$$

It takes a seed and recursively generates a tree structure by applying a co-algebra...

Examples:

- reading a tree structure from file to memory (XML, json, HTML etc)

λ

More morphisms

Compositions:

- Hylomorphisms:
  composition of an anamorphism and a catamorphism
  The first builds, the second immediately consumes the structure without temporaries

Examples:

- numerical integration schemes
- Merge sort (divide-and-conquer algorithms)
- Certain program optimization strategies

More morphisms

Compositions:

- Hylomorphisms:
  composition of an anamorphism and a catamorphism
  The first builds, the second immediately consumes the structure without temporaries

Examples:

- Merge sort in 5 tokens:

$$\text{msort} = \text{hylo merge unflatten}$$

More morphisms

Compositions:

- Hylomorphisms:
  composition of an anamorphism and a catamorphism
  The first builds, the second immediately consumes the structure without temporaries

Examples:

- Merge sort in 5 tokens:

Merge two sorted lists (branch in the tree)

Generate a tree level by splitting in two

$$\text{msort} = \text{hylo merge unflatten}$$

Functional Programming

More morphisms

Paramorphisms:
like factorial: eats its argument, but keep it too
(generally: any subresult)

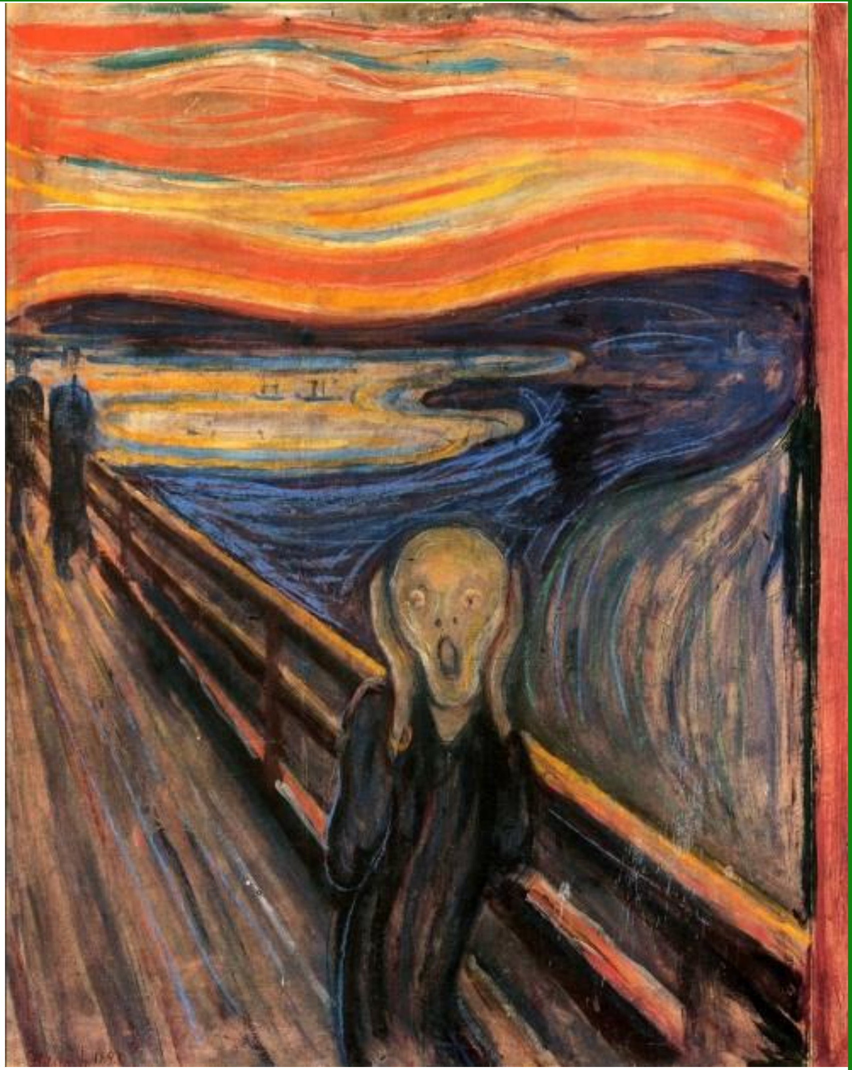
certain optimizations cannot work for these kinds of recursive functions, precisely because they have different algebraic properties

Dual: apomorhisms…

λ

The zoo-of-morphisms

So when this is going to end?????



The zoo-of-morphisms

A collection of recursion schemes is assembled from the literature by Edward Kmett and implemented in Haskell.

Fold Schemes	Description	Unfold Schemes	Description
Catamorphism	Consume structure level by level	Anamorphism	Create structure level by level
Paramorphism	Consume with primitive recursion	Apomorphism	Create structure, may stop and return with a branch or level
Zygomorphism	Consume with the aid of a helper function		
Histomorphism	Consume, possibly multiple levels at once	Futumorphism	Create structure, possibly multiple levels at once
Prepromorphism	Consume, by repeatedly applying a natural transformation	Postpro-morphism	Create, by repeatedly applying a natural transformation

The zoo-of-morphisms

Okay, but why on Earth?

The zoo-of-morphisms

The reason is nothing complicated...

Precisely the same way we moved from goto to `for` loops and then to algorithms...

We less likely to make a mistake, easier to read, comprehend, manipulate, reason about.

The zoo-of-morphisms

C / C++	Functional
goto	Y-combinator based recursion
for loops	folds / unfolds
std::algorithms	structured recursion schemes

Outlook

Outlook

Outlook

The correspondences does not and at Curry-Howard...

Logic	Type Theory	C++
Conjunction	Product type	~ struct
Disjunction	Sum type	~ union+enum
Implication	Function type	R f(A…)
Implication introduction	Function definition	R f(A a, A b){ return a*b; }
Implication elimination	Function call	f(a, b);

The correspondences does not and at Curry-Howard...

Category Theory	Physics	Topology	Logic	Computation
object X	Hilbert space X	manifold X	proposition X	data type X
morphism $f\colon X \to Y$	operator $f\colon X \to Y$	cobordism $f\colon X \to Y$	proof $f\colon X \to Y$	program $f\colon X \to Y$
tensor product of objects: $X \otimes Y$	Hilbert space of joint system: $X \otimes Y$	disjoint union of manifolds: $X \otimes Y$	conjunction of propositions: $X \otimes Y$	product of data types: $X \otimes Y$
tensor product of morphisms: $f \otimes g$	parallel processes: $f \otimes g$	disjoint union of cobordisms: $f \otimes g$	proofs carried out in parallel: $f \otimes g$	programs executing in parallel: $f \otimes g$
internal hom: $X \multimap Y$	Hilbert space of 'anti-X and Y': $X^* \otimes Y$	disjoint union of orientation-reversed X and Y: $X^* \otimes Y$	conditional proposition: $X \multimap Y$	function type: $X \multimap Y$

Table 4: The Rosetta Stone (larger version)

http://arxiv.org/abs/0903.0340

Outlook

The correspondences does not and at Curry-Howard...

Category Theory	Physics	Topology	Logic	Computation
object X	Hilbert space X	manifold X	proposition X	data type X
morphism $f: X \to Y$	operator $f: X \to Y$	cobordism $f: X \to Y$	proof $f: X \to Y$	program $f: X \to Y$
tensor product of objects: $X \otimes Y$	Hilbert space of joint system: $X \otimes Y$	disjoint union of manifolds: $X \otimes Y$	conjunction of propositions: $X \otimes Y$	product of data types: $X \otimes Y$
tensor product of morphisms: $f \otimes g$	parallel processes: $f \otimes g$	disjoint union of cobordisms: $f \otimes g$	proofs carried out in parallel: $f \otimes g$	programs executing in parallel: $f \otimes g$
internal hom: $X \multimap Y$	Hilbert space of 'anti-X and Y': $X^* \otimes Y$	disjoint union of orientation-reversed X and Y: $X^* \otimes Y$	conditional proposition: $X \multimap Y$	function type: $X \multimap Y$

Table 4: The Rosetta Stone (larger version)

http://arxiv.org/abs/0903.0340

Outlook

WIGNER GPU Lab

Introductory readings for the interested:

- Philip Wadler
  - Propositions as Types, and other writings
- Benjamin C. Pierce – Types and Programming Languages
- Bob Coecke - Introducing categories to the practicing physicist
- Michael Barr, Charles Wells – Category Theory for Computing Science
- John C. Baez
  - Physics, Topology, Logic and Computation: A Rosetta Stone
  - A Prehistory of n-Categorical Physics
- Eric Meijer
  Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Functional Programming

λ

Outlook

Recommended reading to get used to functional languages:

Learn you a Haskell for great good!

Tim Williams: Recursion Schemes by Example   Video

If you want to know how to base mathematics on homotopy type theory: link

If you want to know how to base physics along same lines: link video