Deep Magic

λ-calculus, Type Theory, Category Theory, Functional Programming

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

D. Berényi – M. F. Nagy-Egri

"The Jargon File makes a distinction between ***deep magic***, which refers to code based on *esoteric theoretical knowledge*, and ***black magic***, which refers to code based on techniques that appear to work but which lack a theoretical explanation.

It also defines ***heavy wizardry***, which refers to code based on obscure or undocumented intricacies of particular hardware or software."

Wikipedia

Formal systems

Formal systems

Formal systems

A *formal system* is somehow the “formalization” of mathematical abstract thinking, model construction and manipulation.

Its representation is a formal language, that consist of:

- A finite number of symbols
- A grammar — which sequence of symbols is a well-formed formula, which expression is meaningful in the system
- A finite set of axioms — all of them are well-formed formulas
- A finite set of inference rules — that transform well-formed formulas into well-formed formulas

λ

Propositional Logic

Propositional Logic consists of:

- Propositions: atomic formulas of logical statements. They truth value is either true or false.
- Composite expressions built by logical connectives:
  - Negation ($\neg$)
  - And ($\wedge$)
  - Or ($\vee$)
  - Implication ($\rightarrow$)
  - Equivalence ($\equiv, \leftrightarrow$)

Propositional Logic

Propositional Logic is closed for the above operations:
composite expressions are also propositions

Argument: list of propositions, where the last one can be derived from the earlier ones.

Propositional Logic is a formal system:

- The symbols represent the atomic propositions
- Grammar is made from the logical connectives
- The axioms (if any) are considered true without premises
- The inference rules are truth preserving transformations, and the new propositions derived this way are the theorems.

Propositional Logic

The main application of Propositional Logic is to decide if two logical formulae have the same truth value by repeatedly applying the inference rules (proof)

There two things to be proven of such a logical forma system:

- Soundness - the rules does not contradict each other
- Completeness - all proofs can be carried out without the need for another inference rule.

Propositional Logic

Extensions:

- First order logic:

  logical functions: $P(x)$

  quantification over a group of objects that are subject of the propositions:
  - Existential: $\exists x(P(x))$: there exists an x, such that $P(x)$ is true
  - Universal: $\forall x(P(x))$: for all x proposition $P(x)$ is always true

- Higher-order Logic:
  Quantification can happen over functions or propositions, or sets of these…

λ

λ-calculus

λ-calculus

λ-calculus

λ-calculus is a formal system that is built to study function abstraction and application

by Alonso Church and his students:
Stephen Kleene and John Barkley Rosser

Note: function application is just "whitespace", otherwise the number of parentheses would be enormous…

The elements of the grammar:

- Variable symbols: x, y
- Lambda abstraction: $\lambda x. x^2$
- Lambda application: $(\lambda x. x^2)\ 4 \ \rightarrow 16$

λ-calculus

Properties:

- α – equivalence: the symbol of the parameter (bound variable) does not matter: $\lambda x. x^2 \equiv \lambda z. z^2$

- β – reduction: function application is just “find-and-replace” in the lambda abstraction: $\left(\lambda x. \frac{x^2+x}{2^x-x}\right) 4 \equiv \frac{4^2+4}{2^4-4}$

  assuming that the name of the variable being substituted does not clash with an existing name

- η – equivalence : $\lambda x. f x \equiv f$

λ-calculus

In a well-behaving formal system, there are normal forms, that cannot be simplified further by the transformation rules.

If a system is strongly normalizing, every expression can be simplified to such a normal form...

λ −calculus is not such a system.
Consider this expression
and try to β − reduce it:

Note:
function applications at every space!

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \rightarrow (\lambda x.x\ x)\ (\lambda x.x\ x)$$

λ-calculus

Natural numbers can be encoded:

$$0 \coloneqq \lambda f.\lambda x.x$$
$$1 \coloneqq \lambda f.\lambda x.f\ x$$
$$2 \coloneqq \lambda f.\lambda x.f\ (f\ x)$$
$$3 \coloneqq \lambda f.\lambda x.f\ \big(f\ (f\ x)\big)$$
$$n \coloneqq \lambda f.\lambda x.f^n\ x$$

These are higher order functions:
they apply a given f function n times on x: $3\ f\ x = f\ (\ f\ (\ f\ x)\ )$

λ-calculus

- Increment by one (succ):

  $\lambda n. \lambda f. \lambda x. f\ (n\ f\ x)$
- Addition: $f^{m+n}(x) = f^m(f^n(x))$:

  $\lambda m. \lambda n. \lambda f. \lambda x. m\ f\ (n\ f\ x)$
- Multiplication: $f^{m*n}(x) = (f^n)^m\ (x)$

  $\lambda m. \lambda n. \lambda f. m\ (n\ f)$
- Raise to power:
  by definition: $n\ f\ x = f^n\ x$, and now take $f \to m, x \to f$ to have:

  $\lambda m. \lambda n. n\ m$
- Decrement by one (pred):

  $\lambda n. \lambda f. \lambda x. n\ (\lambda g. \lambda h. h\ (g\ f)\ )\ (\lambda u. x)\ (\lambda u. u)$
- Subtraction: $\lambda m. \lambda n. (n\ pred)\ m$

Similarly Boolean logic and list operations can be represented...

λ-calculus

The most interesting construction is the Y-combinator:

$$\mathrm{Y} \coloneqq \lambda g.\left(\lambda x.\, g\ (x\ x)\right)\left(\lambda x.\, g\ (x\ x)\right)$$

- After two $\beta -$ reductions we have: $Y\ f = f\ (Y\ f)$
  Continuing: $Y\ f = f\ (Y\ f) = f\ (\ f\ (Y\ f)\ )$
- Such a function can be used to create a recursive function as follows:

Note: The first argument will be its self!

$$\mathrm{Factorial\_proto} \coloneqq \lambda f.\lambda n.\,(n == 1\ ?\ 1 : n \times (f\ f\ (n-1)\ )\ )$$

$$\mathrm{Factorial} \coloneqq Y\ \mathrm{Factorial\_proto}$$

Pl.: $\mathrm{Factorial}\ 4\ \rightarrow 24$

λ-calculus

Such recursions are infinite, they cannot be normalized.

This means, that $\lambda-$calculus cannot be interpreted as a consistent logical system…

Brief history

Now we can speak a little bit about history...

Brief history

At the beginning of the XX. Century Whitehead’s and Russel’s Principia Mathematica made it obvious, that most of then used mathematics is expressible with the tools of formal logic.

This lead David Hilbert to set the goal of constructing a complete and consistent formal system of mathematics, where given any statement it can be efficiently decided whether it is true or false. (*Entscheidungsproblem)*

Brief history

Of course, the devil is in the details:

What is efficient? What is decidable?

Is it really true, that every statement is either true or false?

Also: in 2000 years it was not possible to define "efficiently decidable"...

...and now out of nowhere appeared three definitions!

Brief history

Church and his students constructed the $\lambda-$calculus. They said, that any statement, that is expressible and normalizable can be considered calculable (decidable)

Gödel didn't like this, he devised the so called generalized recursive functions. Church's team quickly showed the equivalence of this and their definition.

Alan Turing made the abstract model of the computer: the Turing machine. It turns out, that the Halting Problem is also equivalent with the above formulations…

λ

Brief history

These developments started to undermine Hilbert’s program, but it was completely proved impossible by Gödel:

- In any formal system, that is rich enough to encode the natural number arithmetic, there exists a statement that cannot be proven or disproven. These systems are not complete.
- Such a system cannot prove it’s own consistency

Brief history

Something had to be done with such irreducible expressions and recursions... Especially in logic, where it lead to paradoxes.

Russels's paradox:

- Let H be a set, whose elements are those sets that do not contain themselves...
- Then H is not a member of its self, so by definition it should contain its self, but this is a contradiction.

λ

Brief history

Some further related results were found:

- If a formal system is complex enough to encode arithmetic, then it cannot solve its own halting problem (such as expression normalizing).

- If we restrict the system, such as we take out recursion, then the halting problem will be trivial, but we cannot express certain calculations.

Brief history

Paradoxes in logic were resolved in two ways:

- The restriction of set theory axioms and the related formal system: Zermelo-Fraenkel set theory (together with the axiom of choice), this is the foundation of modern set theory.

- Introduction of ***types*** to annotate propositions by what kind of element collections can they operate on.

λ

Type theory

Type theory

Type theory

Basic properties:

- We extend the formal system with types: any expression will have exactly one type

  $\mathrm{x} : \mathrm{T}$ example: $2 : \mathrm{integer}$

- Such a statement is a typing judgement. If the lhs is a composite expression, then it must be typed recursively by inspecting the subexpressions first.
- example: $2 + \mathrm{true}$ cannot be typed.

Type theory

The formal system must be augmented by typing judgements, written as: $: \frac{\text{Type premises}}{\text{Resulting type judgement}}$

Example:

$$\frac{}{42 : \text{integer}}$$ read: the type of 42 is integer. (no premises)

Functions are written by the → operator:

$$\frac{}{+ : \text{integer} \rightarrow \text{integer} \rightarrow \text{integer}}$$ (no premises)

Type theory

Note: the → operator is understood to be in Curry-ed form:

Example:

$$+ : \text{integer} \rightarrow \text{integer} \rightarrow \text{integer}$$

3+4 first step:

$$(3\,+) : \text{integer} \rightarrow \text{integer}$$

The → operator is right associative:
$A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

second step:

$$(3\,+)\;4 : \text{integer}$$

λ

Type theory

The function calls are typed as:

$$\frac{f : A \to B \quad x : A}{f\ x : B}$$

Again the example of 3+4 is typed as:

$$\cfrac{\cfrac{\cfrac{}{+ : \text{integer} \to \text{integer} \to \text{integer}} \qquad \cfrac{}{3 : \text{integer}}}{+\ 3 : \text{integer} \to \text{integer}} \qquad \cfrac{}{4 : \text{integer}}}{+\ 3\ 4 : \text{integer}}$$

Type theory

We need variables, but this means, we need to have a context: what is the type of a given symbol in the given context?

$$\Gamma = \{\mathrm{x} : \mathrm{integer}, \mathrm{f} : \mathrm{integer} \rightarrow \mathrm{integer}\,\}$$

All previous judgements should be augmented:

$$\Gamma \vdash \mathrm{e} : \mathrm{T} \qquad \text{(type of } e \text{ is } T \text{ in } \Gamma\text{).}$$

Type theory

The typing judgement: $\dfrac{}{\Gamma \vdash x : T}$

Earlier examples become like: $\dfrac{}{\Gamma \vdash 42 : \text{integer}}$

Function abstraction: $\dfrac{\Gamma, x : T \vdash e : U}{\Gamma \vdash (\lambda x : T\,.\,e) : T \to U}$

λ

Type theory

Now:

- Unicity:
  All expressions can have at most 1 type.
  If an expression cannot be typed, it is an error.

- Soundness:
  If an expression can be typed, then it can be safely evaluated!

- Completeness?
  Can we type anything that is not erroneous at evaluation?

Related reading

λ

Type theory

Why we did this whole thing in the first place???

If we equip the $\lambda-$calculus with this type system we get a logically consistent formal system (the simply typed $\lambda-$ calculus) that is *strongly normalizing*!!!

But, we cannot type the Y-combinator and similar recursive expressions...

Type theory

Now some type combinators:

- Sum type: $\mathrm{A} + \mathrm{B}$ (tagged union, variant, ...)
  Contains either A or B (decided at runtime) constructed by either one.

- Product type: $\mathrm{A} \times \mathrm{B}$ (tuple, record, ...)
  Contains A and B, constructed by both of them.

If the number of all possible values of type A is denoted by $|A|$, then:

$$|A + B| = |A| + |B|$$
$$|A \times B| = |A| \cdot |B|$$

λ

Type theory

Types can be differentiated, see: link, link

The function type has: $|A \to B| = |B^A| = |\mathrm{B}|^{|\mathrm{A}|}$

This gives the number of all possible functions taking A and returning B.

Type theory

In 1934 Haskell Brooks Curry observed that:

- The function signatures can be interpreted as implications.
  So all function types are provable statements.
- Conversely: for all provable statements there exists a function type
- Same kind of relation hold between expressions and proofs.

In 1969 William Alvin Howard extended this:

- There exists a logical system called Natural Deduction (by Gerhard Gentzen) that corresponds to the simply typed λ −calculus
- Evaluation of expressions corresponds to the simplification of logical proofs.

Type theory

Howard found that:

- $A \lor B$ corresponds to the sum type: $A + B$
- $A \land B$ corresponds to the product type: $A \times B$
- $A \rightarrow B$ corresponds to the function type: $A \rightarrow B$

- The fact, that sub-proofs can be composed by symbolic manipulations was known (Brouwer-Heyting-Kolmogorov), but its connection to programming was not.

This correspondence is known after Curry-Howard

Type theory

Howard went further:

- Logical quantors then should have a counter part in type theory!

- These are the Existential and Universal Types,
  today commonly known as Dependent Types.

λ

Type theory

The Curry-Howard correspondence inspired Jean-Yves Girard and John Reynolds to construct System F (or polymorphic λ – calculus)

It was theoretically investigated further by Per Martin-Löf in the framework of Intuitionistic Type Theory to include dependent types.

Π – types correspond to universal quantification:
It is a dependent type, that maps a type to a type:
Example: an array that stores n elements of type T

$$\prod_{\mathrm{n}:\mathbb{N}} \mathrm{Vector}(\mathrm{T},\mathrm{n}) \leftrightarrow \forall n \in \mathbb{N}\,.\,T:\mathcal{U} \to \mathrm{Vector}(\mathrm{T},\mathrm{n}) : \mathcal{U}$$

Σ – types correspond to existential quantification:

$$\sum_{\mathrm{n}:\mathbb{N}} \mathrm{List}(\mathrm{T},\mathrm{n}) \leftrightarrow \exists n \in \mathbb{N}\,.\,\mathrm{List}(\mathrm{T},\mathrm{n}) : \mathcal{U}$$

Deep Magic

λ

Type theory

The problem with System F is that while the rewrite rules are strongly normalizing, the type judgements are not decidable!

In general: dependently typed languages must balance between expressiveness and decidability!

This whole line lead to the investigation of different type systems and corresponding logical systems culminating in the development of proof assistants.

The system F and its extensions (Hindley-Milner) is the base of the modern functional languages like Haskell or the ML family, and many ideas started to propagate to Java and C#

The Martin-Löf theory is the base of the modern proof-assistants like Coq, Agda, Idris

Yet another direction

Abstract algebra and category theory

Abstract algebra

The beginning of the XX. century saw the axiomatization of algebraic structures also.

How do we define an algebraic structure?

- One (or more) underlying sets
- One (or more) operations on the elements of the underlying sets
- Some axioms on the operations

Abstract algebra

Minimal example:

- Set S without operations and axioms☺

Less minimal example:

- Magma

  A set S with one binary operation, that is closed:
  applied to any two elements from S, the result is again in the set S.

Abstract algebra

Physicists mainly familiar with groups:

Group:

An algebraic structure with one binary operation (usually denoted by $\cdot$)

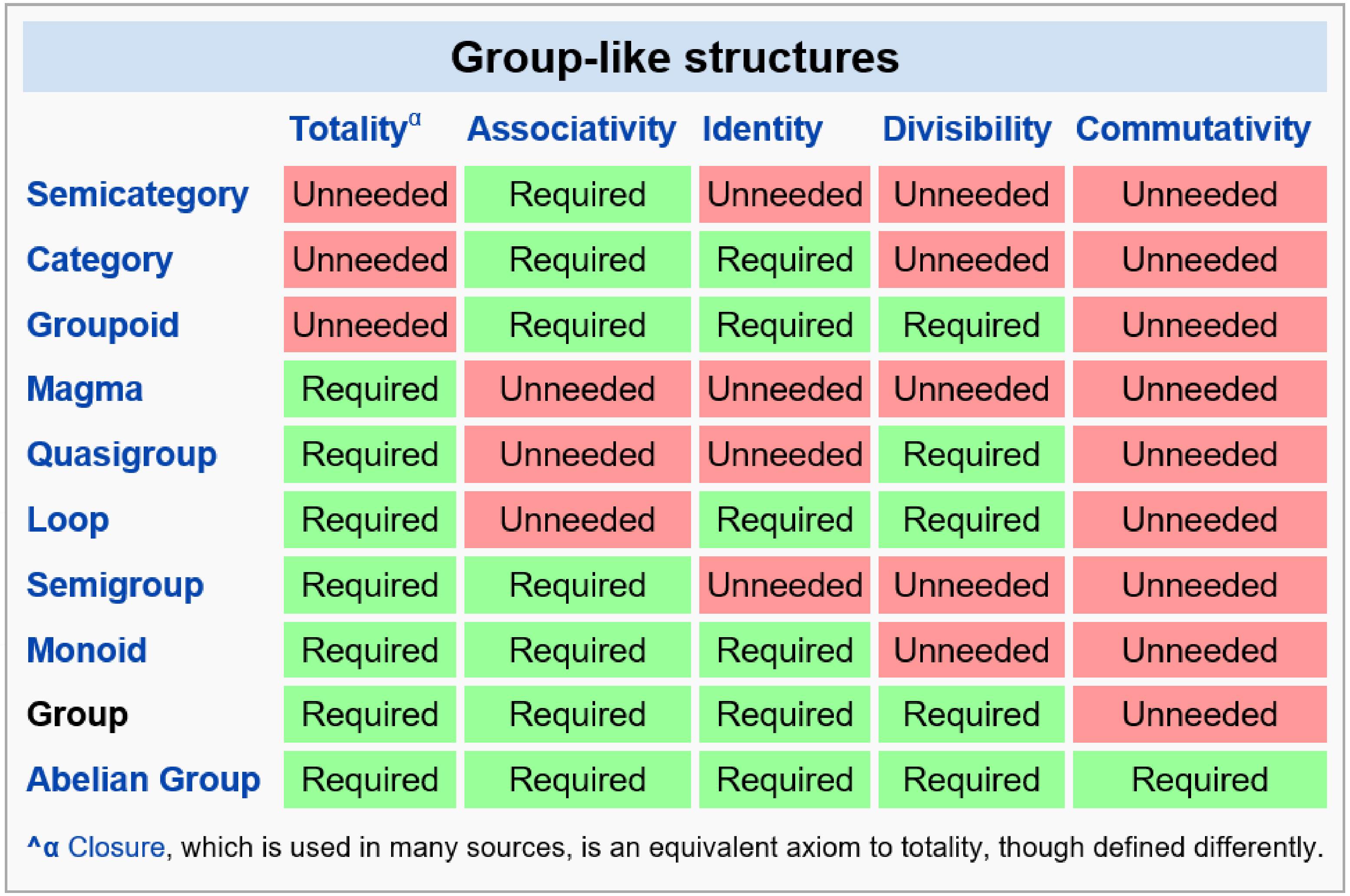
Group axioms:

- Closure: $\forall a, b \in G\ .\ \exists a \cdot b \in G$
- Associativity: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Unity: $\exists 1 \in G\ \ \forall a \in G\ .\ 1 \cdot a = a \cdot 1 = a$
- Inverse: $\forall a \in G\ \exists a^{-1} \in G\ .\ a \cdot a^{-1} = a^{-1} \cdot a = 1$

λ

Abstract algebra

But there are may other structures based on what axioms we impose:



Group-like structures

	Totality[α]	Associativity	Identity	Divisibility	Commutativity
Semicategory	Unneeded	Required	Unneeded	Unneeded	Unneeded
Category	Unneeded	Required	Required	Unneeded	Unneeded
Groupoid	Unneeded	Required	Required	Required	Unneeded
Magma	Required	Unneeded	Unneeded	Unneeded	Unneeded
Quasigroup	Required	Unneeded	Unneeded	Required	Unneeded
Loop	Required	Unneeded	Required	Required	Unneeded
Semigroup	Required	Required	Unneeded	Unneeded	Unneeded
Monoid	Required	Required	Required	Unneeded	Unneeded
Group	Required	Required	Required	Required	Unneeded
Abelian Group	Required	Required	Required	Required	Required

^α Closure, which is used in many sources, is an equivalent axiom to totality, though defined differently.

Deep Magic

λ

Abstract algebra

But there is more:
Ring-like structures:

There are two operations over S: $+$ and $\cdot$, where the axioms:

- S is an abelian-group with $+$ (commutative, associative, unity, inverse)
- S is a monoid with $\cdot$: (associative and has unity)
- $+$ and $\cdot$ are distributive: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

If division is possible also: division ring

If division is possible and the multiplication is commutative: field

Abstract algebra

Structures continue with:

Vector space (V) over field F if

- V is an abelian group with $+$,
- there is a $\cdot : V \times V \to F$ that has unity in V,
    - compatible with the product in F: $*$: $a \cdot (b \cdot \vec{v}) = (a * b) \cdot \vec{v}$ distributive over the addition in F: $+$ and in V: $+$

V is a Module if F is just a ring

Normed vector spaces

Topologic Vector spaces

Complete Topologic Spaces (Banach-, Hilbert-spaces)

If there is also a multiplication in the vector space: it is called an algebra.

λ

Abstract algebra

The problem with Abstract Algebra is that is deals with the structures as if they are isolated from the rest of the world.

Well... this is not exactly true...

Abstract algebra

Group homomorphisms:

Transformations that map from one group to another,
while respecting the group structure:

Let $G(S, \cdot)$ and $H(P, *)$ be two groups, then

- $\phi: G \to H$ is a homomorphism, if:

$$\phi(g_1 \cdot g_2) = \phi(g_1) * \phi(g_2)$$

- It follows that unity maps to unity and inverses map to inverses:
$$\phi(g^{-1}) = \phi(g)^{-1}$$

Abstract algebra

Group homomorphisms:

Transformations that map from one group to another, while **respecting the group structure**:

Let $G(S, \cdot)$ and $H(P, *)$ be two groups, then

- $\phi: G \to H$ is a homomorphism, if:

$$\phi(g_1 \cdot g_2) = \phi(g_1) * \phi(g_2)$$

- It follows that unity maps to unity and inverses map to inverses:

$$\phi(g^{-1}) = \phi(g)^{-1}$$

λ

Abstract algebra

Group homomorphisms:

Transformations that map from one group to another, while **respecting the group structure**:

Let $G(S,\ \cdot)$ and $H(P,\ *)$ be two groups, then

- $\phi : G \rightarrow H$ is a homomorphism, if:

$$\phi(g_1 \cdot g_2) = \phi(g_1) * \phi(g_2)$$

- It follows that unity maps to unity and inverses map to inverses:

$$\phi(g^{-1}) = \phi(g)^{-1}$$

λ

Category Theory

Category Theory

Category Theory

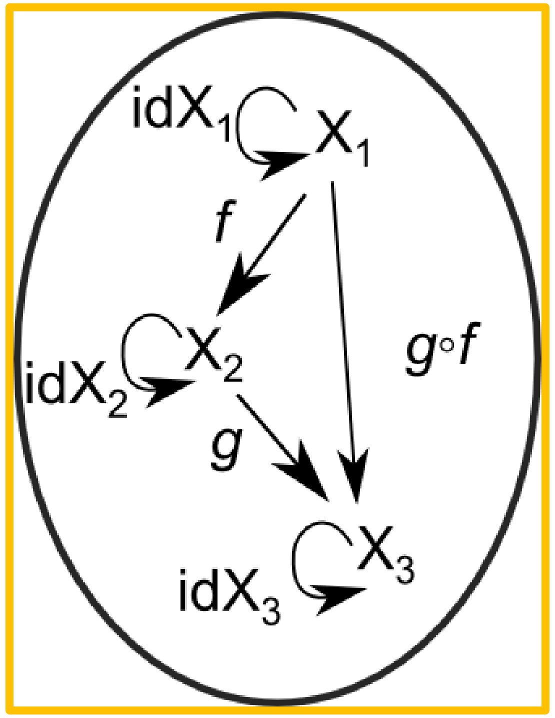
Abstract algebraic structures can be studied by investigating the structure preserving mappings between the structures

How to map from one structure from another, while preserving axioms?

Category Theory

Minimal definition: $\mathcal{C}$ is a category if:

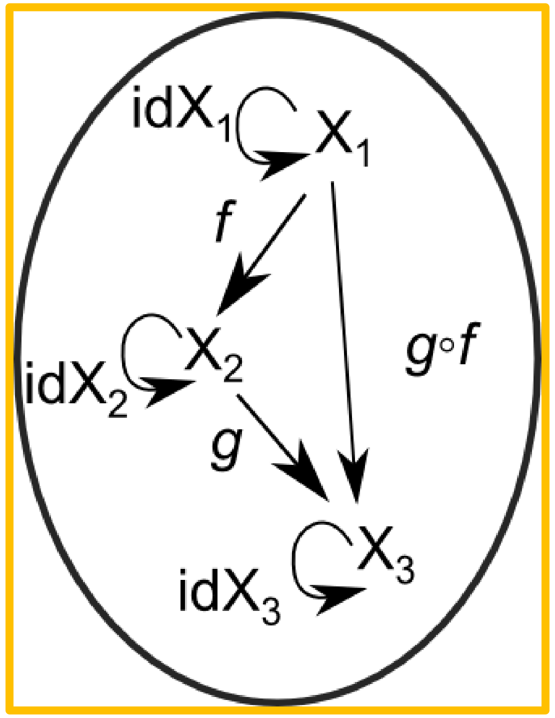
- There exist an ob(C) class,
  collection of "objects"
- There exist a hom(C) class
  collection of morphisms
  (directed connections between objects)
  - There exist a binary operation on morphisms that is: associative and has unity. This is called *composition*.



Category Theory

Some notations:

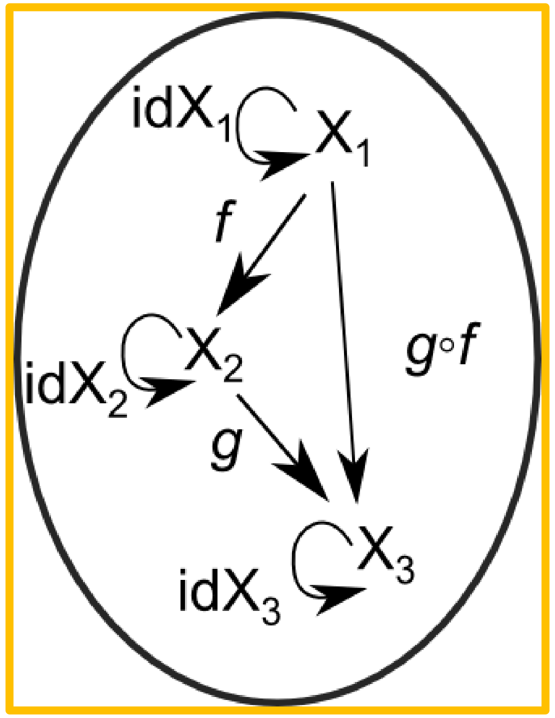
- Categories: $C, D, \ldots$
- Objects: $X, Y, \ldots$
- Morphisms: $f, g, \ldots$
- Identity morphisms on objects: $id_X, id_Y, \ldots$
- Morphism composition: $\circ$


$$\begin{tikzcd} X_1 \arrow[loop left, "idX_1"] \arrow[d, "f"'] \arrow[dr, "g\circ f"] & \\ X_2 \arrow[loop left, "idX_2"] \arrow[r, "g"'] & X_3 \arrow[loop left, "idX_3"] \end{tikzcd}$$

Category Theory

Morphisms can be:

- Isomorphism:
  There exists an inverse, that composes to unity:
  $$f \circ f^{-1} = id_X$$
- Endomorphism:
  The start and end object is the same
- Automorphism:
  the two above at the same time



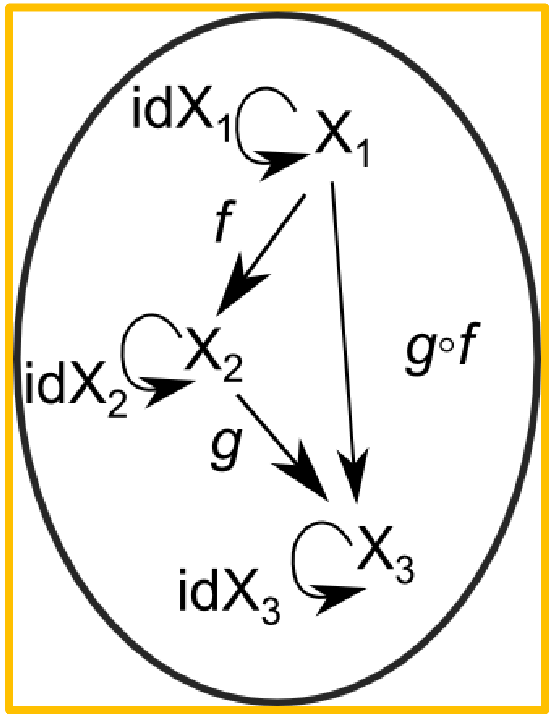
hom(X, Y) denotes the collection of all morphisms between X and Y

Category Theory

Simple example:

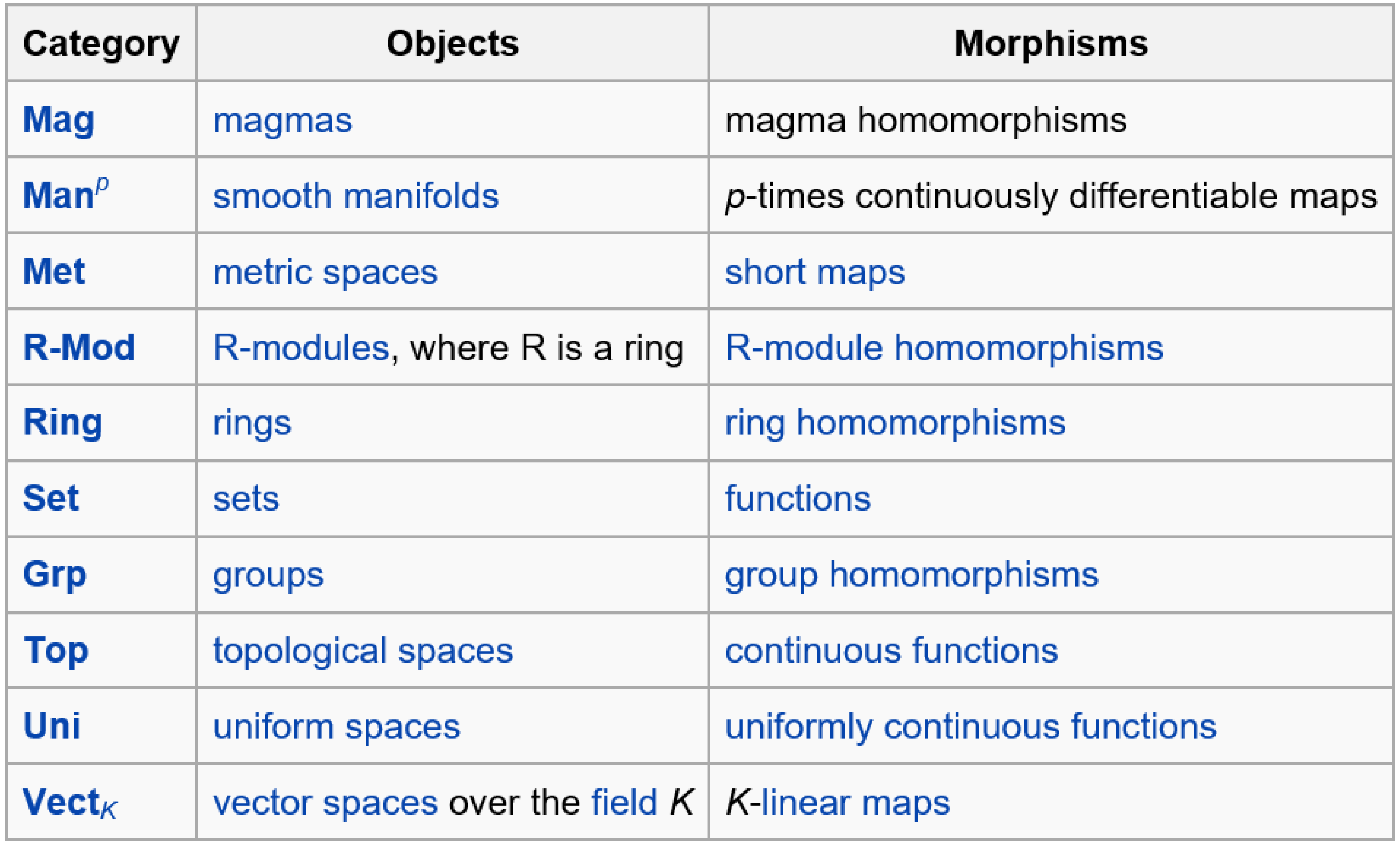
Set:

- Objects are sets
- Morphisms are functions between sets
- Morphisms composition is the usual function composition



Category Theory

Other categories:



Category	Objects	Morphisms
Mag	magmas	magma homomorphisms
Man^p	smooth manifolds	p-times continuously differentiable maps
Met	metric spaces	short maps
R-Mod	R-modules, where R is a ring	R-module homomorphisms
Ring	rings	ring homomorphisms
Set	sets	functions
Grp	groups	group homomorphisms
Top	topological spaces	continuous functions
Uni	uniform spaces	uniformly continuous functions
Vect$_K$	vector spaces over the field K	K-linear maps

Category Theory

Product of objects:

$$\begin{tikzcd} & Y \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\ X_1 & X_1 \times X_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & X_2 \end{tikzcd}$$

- $X = X_1 \times X_2$ if the shown morphisms exist for any Y, f_1, f_2-triple and the diagram commutes.

- f is then called the product of f_1 f_2

Category Theory

Most basic construct over categories: *Functor*

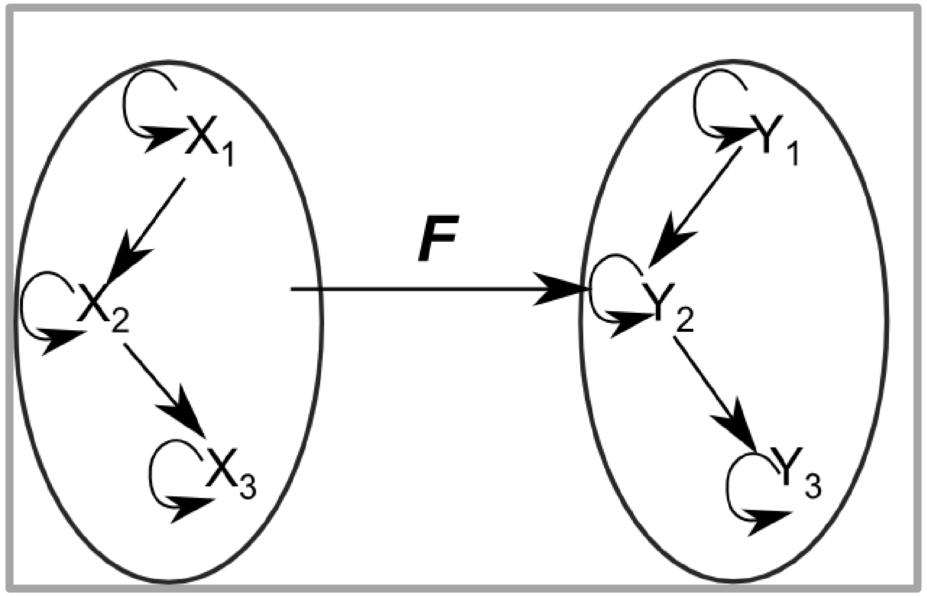
the generalization of homomorphisms to categories

F is a Functor between categories $C \to D$ if:

- $X \in C \Rightarrow F(X) \in D$
- $f: X \to Y \in C \Rightarrow F(f): F(X) \to F(Y) \in D$

such that:

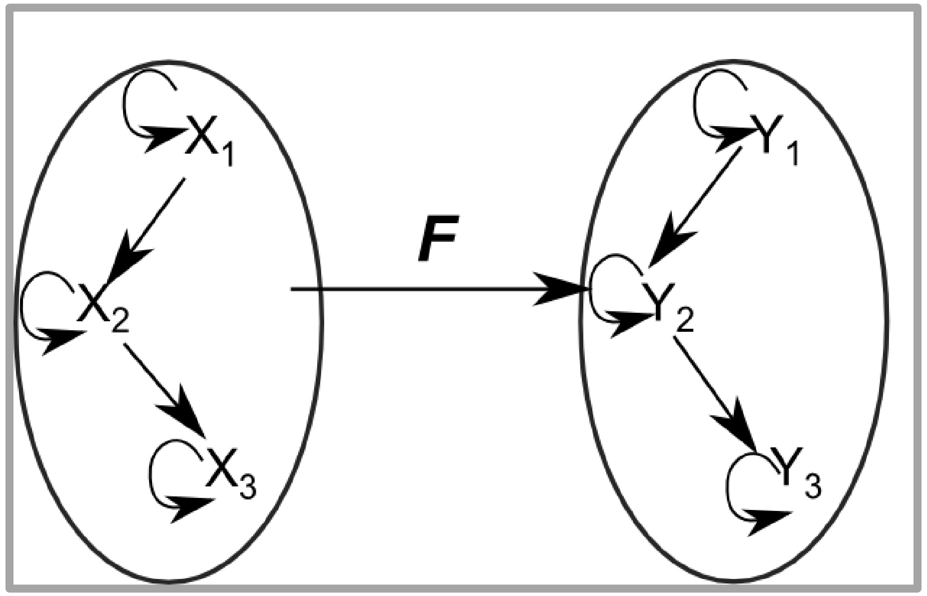
- $F(id_X) \Rightarrow id_{F(X)}\ \forall X \in C$
- $F(g \circ f) = F(g) \circ F(f)\ \forall f, g \in C$



Category Theory

So a Functor maps the objects and morphisms of one category to another, such that:

- It maps unity to unity
- Preserves morphism composition



Category Theory

Example from physics:

Let C be a category, where:

- objects are abstract vector bases
- Morphisms are basis transformations

Let D be a category, where these are represented by coordinates.

Then, $F: C \to D$ is a Functor if:

- it maps bases to coordinate vector sets,
- and transformations to basis transformation matrices (composition: matrix product)

λ

Category Theory

If we fix a vector $\vec{v}$ and the Functor $F_{\vec{v}}$ maps:

- bases to the coordinates of vector $\vec{v}$,
- basis transformations to the matrices acting on $\vec{v}$

we find that the successive basis change of B_1 and B_2 should act on $\vec{v}$ like $\underline{\underline{B_2}}^{-1} \cdot \underline{\underline{B_1}}^{-1}$

While repeating the same for linear functionals we find:

$\underline{\underline{B_1}} \cdot \underline{\underline{B_2}}$

Category Theory

It turns out, that Functors can be

- Covariant $F: C \rightarrow D$ if:
  - $f: X \rightarrow Y \ \in C \rightarrow F(f): F(\mathbf{X}) \rightarrow F(\mathbf{Y}) \in \ D$
  - $F(g \ \circ \ f) = F(\mathbf{g}) \circ F(\boldsymbol{f}) \ \forall f, g \in C$
- Contravariant $G: C \rightarrow D$ if:
  - $f: X \rightarrow Y \ \in C \rightarrow G(f): G(\mathbf{Y}) \rightarrow G(\mathbf{X}) \in \ D$
  - $G(g \ \circ \ f) = G(\boldsymbol{f}) \circ G(\mathbf{g}) \ \forall f, g \in C$

So contravariant functors reverse morphisms and the order in composition.

Category Theory

Opposite category: C^{op}

All morphisms are reversed.

It preserves functors:

$$(F : C \to D)^{op} \cong F : C^{op} \to D^{op}$$

Category Theory

Exponentiation in a category:

- Let C be a category where there is a product, and $Y, Z \in \mathrm{obj}(C)$ objects
- Let $(\times Y): C \to C$ be a Functor, that
  $\forall X \in C \to X \times Y$ and $\forall f \in C \to f \times id_Y$
- Now:
  Z^Y together with the morphism $Z^Y \times Y \to Z$ is the exponential of Y and Z if:
- $\forall X \in C$ there exists a morphism $X \to Z^Y$ unique to $(\times Y)(X)$.

If this is valid for all $Z \in C$, then there is a bijection between

$$\mathrm{hom}(X \times Y, Z) \text{ and } \mathrm{hom}(X, Z^Y)$$

Category Theory

What does this bijection mean?

It turns out, that:

- $Z^Y \times Y \rightarrow Z$ morphism represents the evaluation of the function $Z \rightarrow Y$ at argument Y
- $X \rightarrow Z^Y$ represents the lambda abstraction:

$$\lambda X \simeq (\times Y)(X) = X \rightarrow Z^Y$$

So the bijection between $\mathrm{hom}(X \times Y, Z)$ and $\mathrm{hom}(X, Z^Y)$ is actually the curried form of the two argument function $f\colon X \times Y \rightarrow Z$ that is: $\lambda f\colon X \rightarrow Z^Y$

λ

Category Theory

This means,
we just encoded the λ −calculus inside category theory!

Category Theory

Bear with me, abstract nonsense finishes in a few minutes!

Category Theory

The Functor structure can be repeated again one level higher:

`Cat`: is the category of categories, where:

- The objects are categories
- The morphisms are Functors

Okay, then what are the Functors at this higher level that map `Cat` morphisms to each other?

Category Theory

Natural Transformation (NT):

- If F and G are two Functors between categories C and D, then
- η is a NT, or a collection of morphisms, such that
  - $X \in C \Rightarrow \eta_X : F(X) \to G(X) \in D$
  - Such that $\forall f : X \to Y \in C$ it holds, that

$$\eta_Y \circ F(f) = G(f) \circ \eta_X$$

  - Or, to be more comprehensible,
    this diagram commutes (start from $F(X)$)

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\eta_X"'] & F(Y) \arrow[d, "\eta_Y"] \\
G(X) \arrow[r, "G(f)"'] & G(Y)
\end{tikzcd}$$

Category Theory

Natural Isomorphism (NI):

- If for all objects in C the result of η in D is an isomorphism

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\eta_X"'] & F(Y) \arrow[d, "\eta_Y"] \\
G(X) \arrow[r, "G(f)"'] & G(Y)
\end{tikzcd}$$

Category Theory

Cartesian Closed Categories (CCC):

- Product and Exponential exists for all two objects, in the earlier sense
- There is a terminal object (there is a morphisms from all objects into the terminal)

Monoidal Categories: (generalize the monoid structure to categories)

- $\otimes : \mathrm{C} \times \mathrm{C} \to \mathrm{C}$ bifunctor (a functor mapping from two categories to one)
- $1 \in C$ unit element
- Three natural isomorphisms, that state the monoidal axioms (associativity and unit element behaviour)

Category Theory

Functorial Strength:

- If M is a monoidal category, and $F: M \to M$ is an endofunctor
- The strength of F is the following natural transformation:

  - $v \otimes F(w) \to F(v \otimes w)$ The monoidal product can be applied before the functor
  - such that $u \otimes v \otimes F(w) = F(u \otimes v \otimes w)$
  - and $1 \otimes F(w) = F(w)$

Also multiple products and the unity

λ

Category Theory

If C is a Cartesian Closed Category and it has Monoid structure too, then

- The statement that functor F has a strength is equivalent with the existence of a Natural Transformation that maps $f \in C$ morphisms to $F(f)$ morphisms.

- Put it otherwise: $\forall f: X \to Y \in C \;\; \exists F(X) \to F(Y) \in C$

Category Theory

This means that, taking the product of a morphism $a \to b$ with $F(a)$:

$\hom(a,b) \otimes F\,(a) \Rightarrow F(\hom(a,b) \otimes a)$ Because of the Strength property

$F(\hom(a,b) \otimes a) \Rightarrow F(b)$ Because of the exponentiation of CCC

This means that: a function $a \to b$ can be raised to the functor structure and applied on the image of a ($F(a)$) resulting in an image of b ($F(b)$)

λ

Category Theory

Why is this generalized abstract nonsense is good for us really?

Functional Programming

Programs can be seen as categories, where

- Objects are types
- Morphisms are functions between types
  - Morphism composition is the function composition like: `f(g(x));`
  - Units are the identity functions: like: `int id(int x){ return x; }`

And the Functors… ?
…Here begins the programming part!