Refining the meaning of types

Noam Zeilberger[1]

MSR-Inria Joint Centre

Mathematics, Algorithms and Proofs
Institut Henri Poincaré, Paris, 27 May 2014

[1] based on joint work with Paul-André Melliès

Complete the sequence

- number theorists study numbers
- group theorists study groups
- knot theorists study knots
- type theorists study _____

Complete the sequence

- number theorists study numbers
- group theorists study groups
- knot theorists study knots
- type theorists study _____

Solution: answers to the question, *"What in the world are types??"*

An unsatisfactory answer

Why not this old idea relating type theory and category theory?

$$\text{type systems} \quad \overset{\text{“syntactic category”}}{\underset{\text{“internal language”}}{\rightleftarrows}} \quad \text{categories}$$

One sign of trouble: why do we need two separate “theories” at all?

This work

Revisits the "orthodox interpretation" of type theory through the lens of category theory.

Based on quite simple ideas. (This talk will try to stay elementary.)

Rough draft (Oct 2013): `http://arxiv.org/abs/1310.0263`

In one (strict!) equation:

$$\boxed{\text{type systems} = \text{functors}}$$

Preview

1. An unsettled question
2. A quick refresher on lambda calculus and types
3. Type Systems = Functors
4. On [A000168]

Part 1: An unsettled question

From: Vladimir Voevodsky <vladimir@ias.edu>
Date: Mon, 12 May 2014 19:47:26 +0200
To: types-list@lists.seas.upenn.edu
Subject: [TYPES] types

Hello,

I am reading Russell's texts and the more I investigate them the more it seems to me that the concept of types as we use it today has very little with how types where perceived by Russell or Church.

For them types were a restriction mechanism.

As Russell and Whitehead write:

"It should be observed that the whole effect of the doctrine of types is negative: it forbids certain inferences which would otherwise be valid, but does not permit any which would otherwise be invalid."

The types which we use today are a constructive tool. For example, types in Ocaml are a device without which writing many programs would be extremely inconvenient.

I am looking for a historic advice - when and where did types appear in programming languages which were enabling rather than forbidding in nature?

Morris (1973)

James H. Morris, "Types are not sets":

> *The title is not a statement of fact, of course, but an opinion about how language designers should think about types. There has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas.*

Milner (1978)

Robin Milner, "A Theory of Type Polymorphism in Programming":

> *We now proceed, in outline, as follows. We define a new class of expressions which we shall call types; then we say what is meant by a value possessing a type. Some values have many types, and some have no type at all. In fact "wrong" has no type. But if a functional value has a type, then as long as it is applied to the right kind (type) of argument it will produce the right kind (type) of result- which cannot be "wrong"!*

Reynolds (1983)

John C. Reynolds, "Types, Abstraction, and Parametric Polymorphism":

> *We explore the thesis that type structure is a syntactic discipline for maintaining levels of abstraction. Traditionally, this view has been formalized algebraically, but the algebraic approach fails to encompass higher-order functions. For this purpose, it is necessary to generalize homomorphic functions to relations; the result is an "abstraction theorem" that is applicable to the typed lambda calculus and various extensions, including user-defined types.*
>
> *Finally, we consider polymorphic functions, and show that the abstraction theorem captures Strachey's concept of parametric, as opposed to ad hoc, polymorphism.*

Pierce (2002)

Benjamin C. Pierce, *Types and Programming Languages* (§1.1):

> *As with many terms shared by large communities, it is difficult to define "type system" in a way that covers its informal usage by programming language designers and implementors but is still specific enough to have any bite. One plausible definition is this:*
>
> > *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

Understanding disagreement

My thesis: the real reason *why* type theory hasn't settled on formal definitions of "type" and "type system" is a fundamental tension between two opposing views of what types mean in general.

Reynolds (2000)

John C. Reynolds, "The Meaning of Types: From Intrinsic to Extrinsic Semantics":

> *A definition of a typed language is said to be "intrinsic" if it assigns meanings to typings rather than arbitrary phrases, so that ill-typed phrases are meaningless. In contrast, a definition is said to be "extrinsic" if all phrases have meanings that are independent of their typings, while typings represent properties of these meanings.*

Girard (2004)

Jean-Yves Girard, *The Blind Spot* (§6.F, emphasis added):

> *There are indeed two readings of polymorphism, depending one starts with essence or existence.*
>
> **Essence.** *In this reading, the type is primitive, one constructs it, then one takes care of the objects. There is no real polymorphism, there is only one form (essence) for a given object. This is the viewpoint followed by the category-theoretic interpretations of logic. [...]*
>
> **Existence.** *But one can instead contend that objects are anterior to their type, seen as an essence. This is the viewpoint of subtyping, this is also the viewpoint of ludics: an object may have several types, be representative of several essences.*

(See also Chapter 1 on "Existence vs. essence".)

Part 2: A quick refresher on lambda calculus and types

The untyped lambda calculus

Terms are either variables, applications, or abstractions:

$$t, u ::= x \mid t(u) \mid \lambda x.t$$

Two equations (or rewriting rules):

$$(\lambda x.t)(u) = u[t/x]$$
$$t = \lambda x.t(x)$$

Plus implicit "α-conversion" (e.g., $\lambda x.x = \lambda y.y$)

Theorem: lambda calculus is Turing-complete.

Church's (simplified) theory of simple types

The set of **types** is defined inductively:

1. ι is a (base) type.
2. If A and B are types, then $A \to B$ is a type.

For each type A, we define the **well-formed terms of type** A:

1. Any variable x_A is a wf term of type A
2. If x_A is a variable and t is a wf term of type B then $\lambda x_A.t$ is a wf term of type $A \to B$
3. If t is a wf term of type $A \to B$ and u is a wf term of type A then $t(u)$ is a wf term of type B

Proposition: it is decidable whether a term is well-formed, and the type of a well-formed term is unique.

Intersection type discipline

Terms:

$$t, u ::= x \mid t(u) \mid \lambda x.t$$

Types:

$$S, T ::= \alpha, \beta \cdots \mid S \to T \mid S \cap T$$

Typing:

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : S \to T}\ (\to I) \qquad \frac{\Gamma \vdash t : S \to T \quad \Gamma \vdash u : S}{\Gamma \vdash t(u) : T}\ (\to E)$$

$$\frac{\Gamma \vdash t : S \quad \Gamma \vdash t : T}{\Gamma \vdash t : S \cap T}\ (\cap I) \quad \frac{\Gamma \vdash t : S \cap T}{\Gamma \vdash t : S}\ (\cap E_1) \quad \frac{\Gamma \vdash t : S \cap T}{\Gamma \vdash t : T}\ (\cap E_2)$$

Example: $\vdash \lambda x.x : (\alpha \to \alpha) \cap ((\alpha \to \beta) \to (\alpha \to \beta))$
Theorem: t is strongly normalizable iff $\Gamma \vdash t : A$ for some Γ, A

Intersection type discipline (cont.)

Subtyping axioms:

$$\frac{S \leq T \quad S \leq U}{S \leq T \cap U} \qquad \frac{S_2 \leq S_1 \quad T_1 \leq T_2}{S_1 \to T_1 \leq S_2 \to T_2} \qquad \text{(etc.)}$$

Rule of subsumption:

$$\frac{\Gamma \vdash t : S \quad S \leq T}{\Gamma \vdash t : T}\ (sub)$$

Part 3: Type Systems = Functors

Categories

Definition

A category consists of:

- A collection of *objects* $(A, B, \dots)$.
- A collection of *morphisms* $(f, g, \dots)$, together with operations dom and cod assigning to each morphism a unique source and target. We write $f : A \to B$ to indicate that $\mathrm{dom}(f) = A$ and $\mathrm{cod}(f) = B$.
- Composition and identity: for any pair of morphisms $f : A \to B$ and $g : B \to C$, a morphism $(f ; g) : A \to C$, as well as for every object A, a morphism $id_A : A \to A$.
- Such that associativity and unicity laws hold:

$$(f ; g) ; h = f ; (g ; h)$$
$$f ; id = f = id ; f$$

Type Systems $\rightleftarrows$ Categories

General idea: interpret well-typed terms (of some type system)

$$\Gamma \vdash t : A$$

as morphisms (in some category)

$$t : \Gamma \to A$$

Fact: Church's STLC may be interpreted in any *cartesian closed* category.

The problem of coherence

Problem: any morphism has a fixed domain and codomain, so how should we interpret these rules?

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B}$$

Idea (Reynolds): assign meanings to *typing derivations* rather than terms, and ask for *coherence*.

Our idea is to turn this around: define a type system as the ("forgetful") functor from typing derivations to terms!

First a note on type refinement

Freeman and Pfenning (1991) introduced a two-level approach:

- take an existing typed language (e.g., ML) and add a layer of *refinement types* (intersections, etc.)
- use type system to check more precise properties of programs (i.e., proofs are *reconstructed* by type checker)

Practically motivated, but has lead to much beautiful theory:

- Six dissertations (Freeman, Xi, Davies, Dunfield, me, Lovas)
- See also Frank Pfenning's "Church and Curry: Combining Intrinsic and Extrinsic Typing" (Peter Andrews Festschrift)

In a sense, what I will describe is just a synthesis of the refinement approach to type theory with ideas from categorical logic.

Functors

Definition

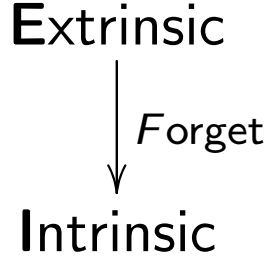
Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A functor $F : \mathbf{C} \to \mathbf{D}$ determines the following:

- for each object A of $\mathbf{C}$, an object $F(A)$ of $\mathbf{D}$, and
- for each morphism $f : A \to B$ of $\mathbf{C}$, a morphism $F(f) : F(A) \to F(B)$ of $\mathbf{D}$,
- such that composition and identity are preserved:

$$F(f; g) = F(f); F(g)$$
$$F(id_A) = id_{F(A)}$$

Setup

Fix (arbitrary) categories **E** and **I**, together with a(n arbitrary) functor $F : \mathbf{E} \to \mathbf{I}$. Our overall reading of this situation:

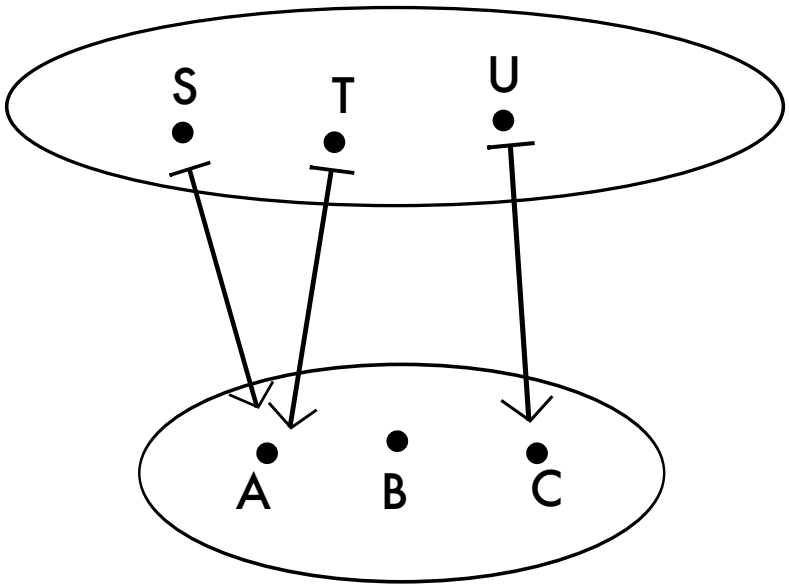


Now let us develop some additional vocabulary.

Type refinement

Definition

We say that an object S of **E** refines an object A of **I** if $F(S) = A$.

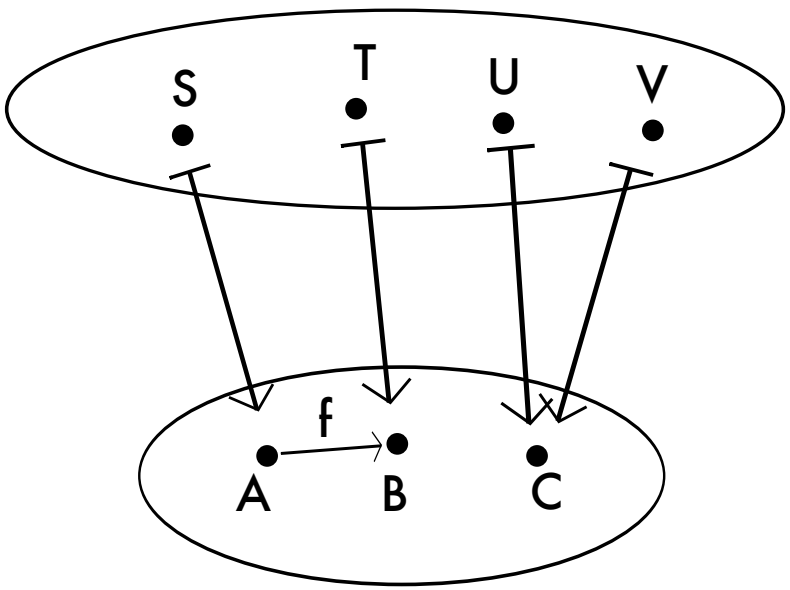


$$S, T \sqsubset A$$
$$U \sqsubset C$$

Typing and subtyping judgments

Definition

A typing judgment is a triple (S, f, T) such that $F(S) = \mathrm{dom}(f)$ and $F(T) = \mathrm{cod}(f)$. In the special case where $f = id$, we call this a subtyping judgment.

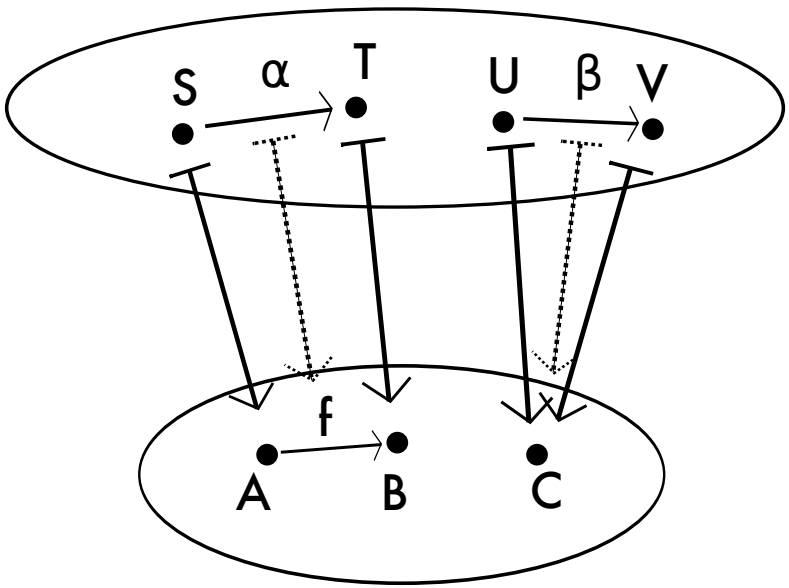


$$S \xrightarrow[f]{} T \qquad U \leq V$$

Typing derivations

Definition

A derivation of a typing judgment (S, f, T) is a morphism $\alpha : S \to T$ such that $F(\alpha) = f$.



$$S \xrightarrow[f]{\alpha} T \qquad U \overset{\beta}{\leq} V$$

We say that a typing judgment $J = (S, f, T)$ is derivable if there is a derivation of J, written $\vdash J$.

Typing rules

Functoriality corresponds to rules for composing typing derivations:

$$\frac{S \underset{f}{\longrightarrow} T \quad T \underset{g}{\longrightarrow} U}{S \underset{f;g}{\longrightarrow} U} \qquad \frac{}{S \underset{id_A}{\longrightarrow} S}$$

Various rules involving subtyping are also valid, e.g.:

$$\frac{S \underset{f}{\longrightarrow} T \quad T \leq U}{S \underset{f}{\longrightarrow} U} \qquad \frac{S \leq T \quad T \underset{g}{\longrightarrow} U}{S \underset{g}{\longrightarrow} U}$$

What next?

All of this makes sense for any functor $F : \mathbf{E} \to \mathbf{I}$.

Asking for additional properties of F corresponds to asking for the existence of additional *logical connectives*. For example, the following rules (presented in type-theoretic language) precisely convey the definition of a *fibration* of categories:

$$\frac{f : A \to B \quad T \sqsubset B}{f^* T \sqsubset A}$$

$$\frac{}{f^* T \underset{f}{\longrightarrow} T} \qquad \frac{S \underset{g;f}{\longrightarrow} T}{S \underset{g}{\longrightarrow} f^* T}$$

$$\frac{\dfrac{S \overset{\beta}{\underset{g;f}{\longrightarrow}} T}{S \underset{g}{\longrightarrow} f^* T} \quad \dfrac{}{f^* T \underset{f}{\longrightarrow} T}}{S \underset{g;f}{\longrightarrow} T} = S \overset{\beta}{\underset{g;f}{\longrightarrow}} T \qquad S \overset{\eta}{\underset{g}{\longrightarrow}} f^* T = \frac{S \overset{\eta}{\underset{g}{\longrightarrow}} f^* T \quad \dfrac{}{f^* T \underset{f}{\longrightarrow} T}}{S \underset{g;f}{\longrightarrow} T}}{S \underset{g}{\longrightarrow} f^* T}$$

In `http://arxiv.org/abs/1310.0263`, we studied situations in which F is *monoidal closed* in addition to being a (bi)fibration.

Part 4: On [A000168]

Scott

The untyped lambda calculus

$$t ::= x \mid t(u) \mid \lambda x.t$$

may be modelled *internally* to any cartesian closed category $\mathbf{I}$, as an object D equipped with inverse operations

$$lam : D^D \to D$$

$$app : D \to D^D$$

Putting the equations aside, this definition may be compared to the encoding of λ-calculus in LF via "higher-order abstract syntax":

```
d : type.
lam : (d -> d) -> d.
app : d -> (d -> d).
```

e.g., where the term $t = \lambda x.\lambda y.x(y)$ is encoded by

```
t = lam [x] lam [y] app x y : d.
```

Carving out the normal forms

On the other hand there is also a well-known direct description of normal forms, in mutual induction with "neutral" terms:

$$N ::= R \mid \lambda x.N$$
$$R ::= x \mid R(N)$$

Formally, this can be seen as a *refinement* of the lambda calculus (cf. Lovas PhD, §2.4). We write the signature

$$N, R \sqsubset D$$
$$sw : R \leq N$$
$$\supset I : N^R \xrightarrow[lam]{} N$$
$$\supset E : R \xrightarrow[app]{} R^N$$

as a compact way of presenting a cartesian closed functor $\mathbf{E} \to \mathbf{I}$.

Imposing some order

Now let's weaken the assumption that **E** is a CCC to the more general monoidal setting, using the following signature:

$$N, R \sqsubset D$$

$$sw : R \leq N$$

$$\supset I : \overset{N}{\underset{R}{\multimap}} \xrightarrow[lam]{} N$$

$$\supset E : R \xrightarrow[app]{} \overset{R}{\underset{N}{\multimap}}$$

Any monoidal closed functor meeting this signature validates the following typing rules:

$$\frac{}{R \xrightarrow[id]{} R} \qquad \frac{\Omega \xrightarrow[t]{} R}{\Omega \xrightarrow[t]{} N} \qquad \frac{R \bullet \Omega \xrightarrow[t]{} N}{\Omega \xrightarrow[lam(\lambda[t])]{} N} \qquad \frac{\Omega_1 \xrightarrow[t]{} R \quad \Omega_2 \xrightarrow[u]{} N}{\Omega_1 \bullet \Omega_2 \xrightarrow[app(t) \lhd u]{} R}$$

In this situation, typing derivations correspond to proofs that lambda terms are normal/neutral and use all variables *exactly once and in (LIFO) order*.

But what does this mean?

Trying to come up with natural examples of such functors $\mathbf{E} \to \mathbf{I}$ (besides the syntactic one), I decided to *count* normal forms and see what emerges. The inductive characterization leads to a simple recurrence for the number of (linear, ordered) normal and neutral terms of a given "length" ℓ and with a given number of free variables k.

Counting as refinement

In fact, *length* can be neatly defined using another monoidal closed functor $\mathbf{E} \to \mathbf{Set}$:

$$
\begin{aligned}
&N, R \mapsto \mathbb{N} \\
&\quad sw \mapsto \ell \mapsto 1 + \ell && (: \mathbb{N} \to \mathbb{N}) \\
&\quad \supset I \mapsto f \mapsto f(0) && (: \mathbb{N}^{\mathbb{N}} \to \mathbb{N}) \\
&\supset E \mapsto \ell \mapsto \lambda m.\ell + m && (: \mathbb{N} \to \mathbb{N}^{\mathbb{N}})
\end{aligned}
$$

In this language, the number of normal forms of a given length ℓ and with a given number of free neutral variables k corresponds to the number of derivations of the judgment

$$\underbrace{R \bullet \cdots \bullet R}_{k \text{ times}} \underset{\ell + \sum -}{\longrightarrow} N$$

where "$\ell + \sum -$" denotes the function of type $\mathbb{N}^k \to \mathbb{N}$ which sums its arguments and adds ℓ.

Feeling lucky?

Using a wee bit of Haskell, I computed $\#(\epsilon \xrightarrow[\ell]{} N)$ for $\ell = 1..7$, obtaining the sequence

$$1, 2, 9, 54, 378, 2916, 24057$$

Entering this into the OEIS strongly suggested that the sequence I was dealing with was A000168, which counts the

"Number of rooted planar maps with n edges"

or equivalently the

"Number of rooted 4-regular planar maps with n vertices".

Further computation confirmed that $\ell = 8$ gives the next entry of the sequence, 208494. (NB: set $\ell = n + 1$ to get the sequences to line up.)

For example...

Here are all the closed, LIFO normal forms of length three:

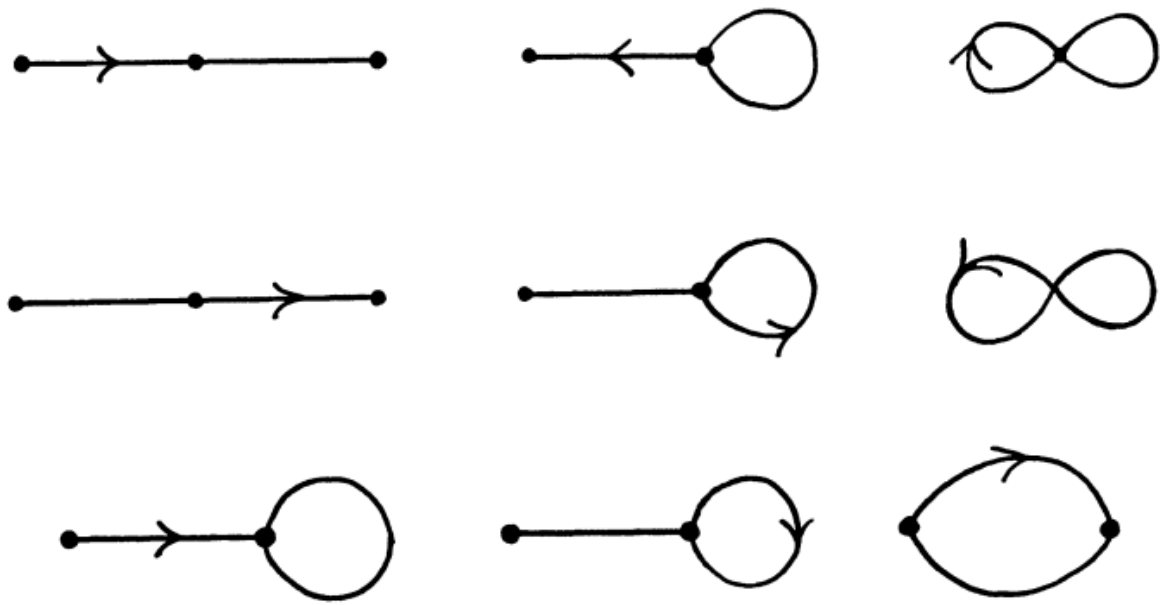
$$\lambda x.x(\lambda y.y(\lambda z.z)) \quad \lambda x.x(\lambda y.\lambda z.z(y)) \quad \lambda x.x(\lambda y.y)(\lambda z.z)$$

$$\lambda x.\lambda y.y(x(\lambda z.z)) \quad \lambda x.\lambda y.y(\lambda z.z(x)) \quad \lambda x.\lambda y.y(\lambda z.z)x$$

$$\lambda x.\lambda y.y(x)(\lambda z.z) \quad \lambda x.\lambda y.\lambda z.z(y(x)) \quad \lambda x.\lambda y.\lambda z.z(y)x$$

For example...

And here are all the rooted planar maps with two edges:[2]



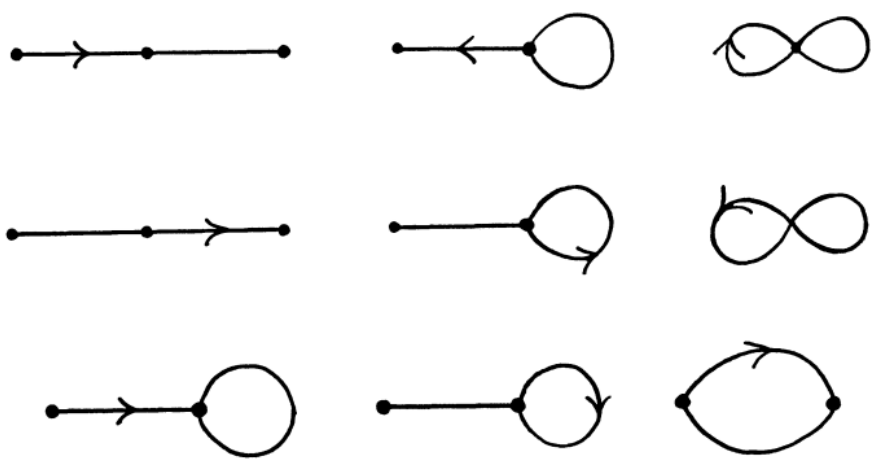
[2] From W. T. Tutte, "A Census of Planar Maps" (1962)

Do you see the resemblance?

$\lambda x.x(\lambda y.y(\lambda z.z))$ $\lambda x.x(\lambda y.\lambda z.z(y))$ $\lambda x.x(\lambda y.y)(\lambda z.z)$

$\lambda x.\lambda y.y(x(\lambda z.z))$ $\lambda x.\lambda y.y(\lambda z.z(x))$ $\lambda x.\lambda y.y(\lambda z.z)x$

$\lambda x.\lambda y.y(x)(\lambda z.z)$ $\lambda x.\lambda y.\lambda z.z(y(x))$ $\lambda x.\lambda y.\lambda z.z(y)x$



Lightning strikes twice

Yesterday, in conversation with Alain Giorgetti, I found out that he was also (!) planning to use A000168 as an example during his talk tomorrow about a tool for enumerative combinatorics.

Conclusion

André Boileau and André Joyal, "La logique des topos" (1981):

> *Il est maintenant clair qu'une catégorie n'est souvent qu'un système formel généralisé...*

This is a powerful slogan, and what we are advocating is just a subtle refinement:

> *In many situations, the concept of "type system" may be usefully identified with the concept of "functor".*

Real motivation: to ultimately get a better understanding of the interaction of logic (and proof theory) with computational effects.

On the lookout for concrete applications!