Rzk proof assistant and simplicial HoTT formalisation

Nikolai Kudasov, j.w.w. Emily Riehl and Jonathan Weinberger

May 26, 2023

Innopolis University

Outline

1. Synthetic theories and proof assistants
2. Synthetic ∞-categories and $\textsc{Rzk}$ language
3. Literate, explicit, visual!
4. Formalising simplicial HoTT
5. What's next?

Synthetic theories and proof assistants

Applied Category Theory

(Higher) category theory (and homotopy theory) has many applications. For example:

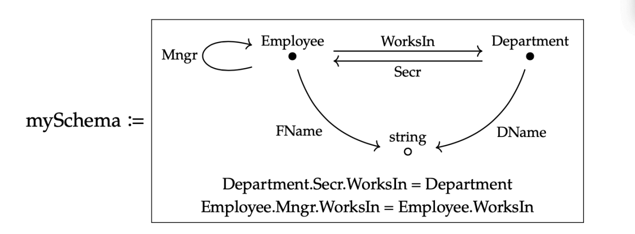
- In chemistry: "A compositional framework for reaction networks" (Baez and Pollard 2017)
- In physics: "Categorical Quantum Mechanics" (Abramsky and Coecke 2009)
- In software engineering and systems design: *Seven Sketches in Compositionality: An Invitation to Applied Category Theory* (Fong and Spivak 2018)
- In natural language processing: "Mathematical Foundations for a Compositional Distributional Model of Meaning" (Coecke, Sadrzadeh, and Clark 2010)

See many more at https://www.appliedcategorytheory.org.

Application: Categorical Databases

Basic idea is that

- database schemas are categories (Fong and Spivak 2018, Chapter 3)
- (good) migrations are (adjoint) functors ($\Sigma \dashv \Delta \dashv \Pi$)



See https://www.categoricaldata.net/ for more details.

Category Theory for Language Implementors

Algebraic concepts (including category theory) influence the tools and libraries used by language implementors:

1. free monads and similar constructions are commonly used in Haskell when implementing DSLs (Swierstra 2008);
2. monads are commonly used to abstract over imperative or SQL-like interfaces;
3. GHC.Generics are used to break down the structure of a user-defined data type to allow safe metaprogramming features.

With a richer language, we can achieve more (e.g. see examples in Licata and Harper 2011, Section 4).

Rzk in context

1. Reasoning directly in (higher) category theory (or homotopy theory) is hard, because one has to check coherences on (infinitely) many levels
2. *Synthetic theories* allow to interalize some of the arguments in such a way that (some) proofs become easier
3. *Proof assistants* check or even derive proofs in synthetic theories

Applications[1] (Physics, Biology, Computer Science, etc.)	
Homotopy Theory	(Higher) Category Theory

[1]see Applied Category Theory at https://www.appliedcategorytheory.org

Rzk in context

1. Reasoning directly in (higher) category theory (or homotopy theory) is hard, because one has to check coherences on (infinitely) many levels
2. *Synthetic theories* allow to interalize some of the arguments in such a way that (some) proofs become easier
3. *Proof assistants* check or even derive proofs in synthetic theories

Applications[1] (Physics, Biology, Computer Science, etc.)	
Homotopy Theory	(Higher) Category Theory
Homotopy Type Theory	Type Theory for Synthetic ∞-categories

[1] see Applied Category Theory at https://www.appliedcategorytheory.org

Rzk in context

1. Reasoning directly in (higher) category theory (or homotopy theory) is hard, because one has to check coherences on (infinitely) many levels
2. *Synthetic theories* allow to interalize some of the arguments in such a way that (some) proofs become easier
3. *Proof assistants* check or even derive proofs in synthetic theories

Applications[1] (Physics, Biology, Computer Science, etc.)	
Homotopy Theory	(Higher) Category Theory
Homotopy Type Theory	Type Theory for Synthetic ∞-categories
UniMath, cubical Agda, redtt, etc.	Rzk

[1] see Applied Category Theory at https://www.appliedcategorytheory.org

Synthetic ∞-categories and Rzk language

Overview

A type theory for synthetic ∞-categories (Riehl and Shulman 2017) is an extension over an (intentional) Martin-Löf Type Theory with two important features:

1. *extension types*, which rely heavily on judgemental equality;
2. *tope logic* (spatial constraints), which requires an (intuitionistic) constraint solver.

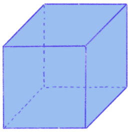
Rzk is an experimental proof assistant (and a language) based on this type theory. The language has simple syntax, but offers a few conveniences (some inspired by Agda, Coq, or Lean) .

```
#lang rzk-1    -- this presentation is a literate rzk file
```

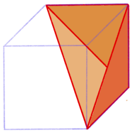
Type theory with shapes

A 3-layer type theory:

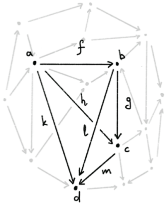
1. *cubes* provide spaces where points come from;
2. *topes* provide restrictions of those spaces;
3. *types* and *terms* are indexed by points in cubes, restricted by topes.



$(t_1, t_2, t_3) : \mathbb{2}^3$



$$(t_3 \equiv 0 \wedge t_2 \leq t_1) \vee$$
$$(t_3 \leq t_2 \wedge t_1 \equiv 1) \vee$$
$$(t_3 \leq t_2 \wedge t_2 \equiv t_1)$$



$$a, b, c, d : A$$
$$f, g, h, k, l, m$$
$$g \circ f = h$$
$$m \circ g = l$$
$$m \circ h = k$$

Cubes and topes

In this talk, we will only use directed interval space **2** ($\mathbb{2}$), directed square **2** `*` **2** ($\mathbb{2}^2$), and directed cube **2** `*` **2** `*` **2** ($\mathbb{2}^3$).

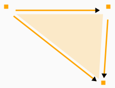
A tope is essentially an (intuitionistic) logical formula that restricts which points in a given space we consider:

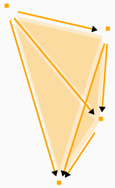
1. `TOP` selects all points in a given space (no restrictions, think true);
2. `BOT` selects nothing (think false);
3. `(psi /\ zeta)` selects all points that satisfy both `psi` and `zeta`;
4. `(psi \/ zeta)` selects all points that satisfy either `psi` or `zeta`;
5. `(t === s)` selects all points such that $\mathtt{t} = \mathtt{s}$;
6. `(t <= s)` selects all points such that $\mathtt{t} \leq \mathtt{s}$;

Basic shapes: simplices

Basic shapes over (products of) the directed interval cube:

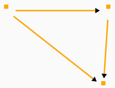
```
-- 1-simplex
#define Δ¹ : 2 -> TOPE
  := \t -> TOP

-- 2-simplex
#define Δ² : (2 * 2) -> TOPE
  := \(t, s) -> s <= t

-- 3-simplex
#define Δ³ : (2 * 2 * 2) -> TOPE
  := \((t1, t2), t3) -> t3 <= t2 /\ t2 <= t1
```





Basic shapes: boundaries

```
-- ∂ boundary of a 1-simplex
#def ∂Δ¹ : Δ¹ -> TOPE
  := \t -> (t === 0_2 \/ t === 1_2)

-- boundary of a 2-simplex
#def ∂Δ² : Δ² -> TOPE
  := \(t, s) ->
     s === 0_2
     \/ t === 1_2
     \/ s === t
```



Type layer: dependent functions

Dependent function types allow result type to depend on the *value* of a previously introduced argument. Here are some equivalent notations for an identity function:

```
#define identity : (A : U) -> (x : A) -> A
  := \A x -> x

-- curry and omit x in the type
#define identity2 : (A : U) -> (A -> A)
  := \A x -> x

-- introduce A for type and term at the same time
#define identity3 (A : U)
  : A -> A
  := \x -> x
```

Type layer: dependent functions

A dependently-typed version of flipping arguments of a function:

```
-- Flipping the arguments of a function.
#def flip
    (A B : U)                             -- For any types A and B
    (C : A -> B -> U)                     -- and a type family C
    (f : (x : A) -> (y : B) -> C x y) -- given a function f : A -> B -> C
  : (y : B) -> (x : A) -> C x y         -- we construct a function of type B -> A -> C
  := \y x -> f x y    -- by swapping the arguments

-- Flipping a function twice is the same as not doing anything
#def flip-flip-is-id
    (A B : U)                             -- For any types A and B
    (C : A -> B -> U)                     -- and a type family C
    (f : (x : A) -> (y : B) -> C x y) -- given a function f : A -> B -> C
  : f = flip B A (\y x -> C x y)
          (flip A B C f)                -- flipping f twice is the same as f
  := refl                                 -- proof by reflexivity
```

Type layer: identity/path types

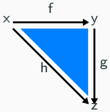
```
#variable X : U
#variable Y : X -> U

-- transport in a type family along a path in the base
#def transport
  (x y : X)
  (p : x = y)
  (u : Y x)
  : Y y
  := idJ(X, x, \y' p' -> Y y', u, y, p)
```

Simplicial types: hom

```
-- [RS17, Definition 5.1]
-- The type of arrows in A from x to y.
#def hom
  (A : U)   -- A type.
  (x y : A) -- Two points in A.
  : U
  := (t : Δ¹) -> A [
    t === 0_2 |-> x,
    t === 1_2 |-> y
  ]
```

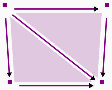
Simplicial types: hom2

```
-- [RS17, Definition 5.2]
-- the type of commutative triangles in A
#def hom2
  (A : U)
  (x y z : A)
  (f : hom A x y)
  (g : hom A y z)
  (h : hom A x z)
  : U
  := { (t1, t2) : Δ² } -> A [
    t2 === 0_2 |-> f t1,
    t1 === 1_2 |-> g t2,
    t2 === t1  |-> h t2
  ]
```



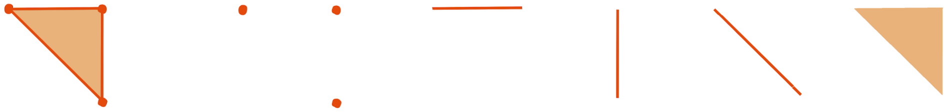
Connection squares

```
#def unfolding-square
  (A : U)
  (triangle : Δ² -> A)
  : Δ¹×Δ¹ -> A
  := \(t, s) ->
    recOR(t <= s |-> triangle (s , t),
              s <= t |-> triangle (t , s))
```



Theorem 4.4

```
-- [RS17, Theorem 4.4]
-- original form
#def cofibration-composition
   (I : CUBE)
   (chi : I -> TOPE)
   (psi : chi -> TOPE)
   (phi : psi -> TOPE)
   (X : chi -> U)
   (a : (t : phi) -> X t)
   : Eq <{t : I | chi t} -> X t [ phi t |-> a t ]>
      (Σ (f : <{t : I | psi t} -> X t [ phi t |-> a t ]>),
          <{t : I | chi t} -> X t [ psi t |-> f t ]>)
   := (\h -> (\t -> h t,
              \t -> h t),
      ((\fg t -> (second fg) t, \h -> refl),
      ((\fg t -> (second fg) t, \h -> refl)))
```



$$(\Delta^2 \to A) \simeq \sum_{x,y,z:A} \sum_{f:\hom_A(x,y)} \sum_{g:\hom_A(y,z)} \sum_{h:\hom_A(x,z)} \hom^2_A(f,g;h)$$

Literate, explicit, visual!

Literate formalisations

The main purpose of Rzk is to support formalisations for synthetic ∞-categories.

However, I believe it is important to try making formalisations understandable for the *reader*, not just easy to use for the *writer*.

To this end, Rzk supports literate programming, and current formalisations are mostly Markdown files with `rzk` code blocks.

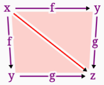
This means that formalisations are more easily browsable, for examples see:

- Rzk documentation at https://fizruk.github.io/rzk/
- Yoneda lemma formalisation project at https://emilyriehl.github.io/yoneda/
- simplicial HoTT formalisation project at https://fizruk.github.io/sHoTT/

Automatic visualisation

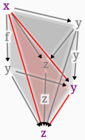
Since many formalisations have a natural accompanying visualisation in a form of commutative diagrams, Rzk implements some basic automatic rendering for topes, simplicial types and terms.

```
#def square
  (A : U)
  (x y z : A)
  (f : hom A x y)
  (g : hom A y z)
  (h : hom A x z)
  (a : Sigma (h' : hom A x z), hom2 A x y z f g h')
  : (2 * 2) -> A
  := \(t, s) -> recOR( s <= t |-> second a (t, s) , t <= s |-> second a (s
```



If a term is extracted as a part of a larger shape, generally, the whole shape will be shown (in gray):

```
#def face
  (A : U)
  (x y z : A)
  (f : hom A x y)
  (a : Sigma (g : hom A y z), {((t1, t2), t3) : 2 * 2 * 2 | t3 <= t1 \/ t2
  : Δ² -> A
  := \(t, s) -> second a ((t, t), s)
```



Explicit assumptions

RZK supports Coq-style sections and variables, with one important distinction: implicit use of variables is not allowed.

```
#variables A B C : U
#variable f : A -> B
#variable g : B -> C
#variable x : A

-- #def bad-compose : C := g (f x)
-- ERROR: implicit assumptions A and B

#def compose uses (A B) : C := g (f x)
```

See https://fizruk.github.io/rzk/site/rzk-1/sections/ for details.

Formalising simplicial HoTT

Why formalize mathematics? (one example)

How I became interested in foundations of mathematics (2014)

[V.Voevodsky] ...in 2003, twelve years after our proof was published in English, a preprint appeared on the web in which his author, Carlos Simpson, very politely claimed that he has constructed a counter-example to our theorem.
...And then in the Fall of 2013, less than a year ago, some sort of a block in my mind collapsed and I suddenly understood that Carlos Simpson was correct and that the proof which Kapranov and I published in 1991 is wrong.
Not only the proof was wrong but the main theorem of that paper was false!
...
...if it were a complex equation we would probably have checked it on a computer.
So why can not we check a solution which is a proof of a theorem?

Formalising the Yoneda Lemma

The project is a collaboration with Emily Riehl and Jonathan Weinberger.

One goal of the project is to formalise the Yoneda lemma for synthetic ∞-categories following the original paper (Riehl and Shulman 2017).

```
#def Yoneda-lemma
    (funext : FunExt)
    (A : U)                                 -- The ambient type.
    (AisSegal : isSegal A)                  -- A proof that A is Segal.
    (a  : A)                                -- The representing object.
    (C : A -> U)                            -- A type family.
    (CisCov : isCovFam A C)                 -- A covariant family.
    : isEquiv ((z : A) -> hom A a z -> C z) (C a) (evid A a C)
    := ((yon A AisSegal a C CisCov,
            yon-evid A AisSegal a C CisCov funext),
        (yon A AisSegal a C CisCov,
            evid-yon A AisSegal a C CisCov))
```

See more details at https://github.com/emilyriehl/yoneda

Formalising the Yoneda Lemma: comparing with other proof assistants

The project is a collaboration with Emily Riehl and Jonathan Weinberger.

Another goal is to compare the proof of the Yoneda lemma for ∞-categories in simplicial HoTT with proofs of the Yoneda lemma for 1-categories in other proof assistants. Sina Hazratpour has contributed formalisations in Lean to that end.

```
def yoneda_covariant {𝒞 : Type*} [category 𝒞] {F : 𝒞 ⥤ Type* } (A B : 𝒞) :
  (𝕁 A ⟶ F) ≅ F.obj A :=
{ to_fun := λ α, α.cmpt A (𝟙 A),
  inv_fun := λ a, { cmpt := λ X, λ f, (F.mor f) a,
                    naturality' :=
                    by {
                          intros X Y k,
                          funext x,
                          simp [rep_obj, rep_mor],
                          dsimp,
                          conv
                            begin
                              to_lhs,
                              change (F.mor (k ⊚ x)) a,
                            end,
                          conv
                            begin
                              to_rhs,
                              change (F.mor k) (F.mor x  a),
                            end,
                          rw [functor.resp_comp],
                          refl,
                        },
                  },
  left_inv :=  by { funext α, dsimp, ext X a, simp, rw ← cov_naturality.fibrewise },
  right_inv := by {funext, dsimp, rw functor.resp_id, refl}, }
```

See more details at https://github.com/emilyriehl/yoneda

Fixing a proof

Recently, Rzk helped find an bug in a proof of Riehl and Shulman 2017, Proposition 8.13.
Luckily, the proof could be fixed[a] in a non-trivial (for me), but fairly straightforward way.

[a] https://github.com/emilyriehl/yoneda/pull/6

Proposition 8.13. *Let A be a type and fix $a : A$. Then the type family*

$$\lambda x.\ \hom_A(a, x) : A \to \mathcal{U}$$

is covariant if and only if A is a Segal type.

Proof. The condition of Definition 8.2 asserts that for each $b, c : A$, $f : \hom_A(a, b)$, and $g : \hom_A(b, c)$, the type

$$\sum_{h:\hom_A(a,c)} \left\langle \textstyle\prod_{s:2} \hom_A(a, g(s)) \Big|^{\partial\Delta^1}_{[f,h]} \right\rangle$$

is contractible. Applying Theorem 4.4, this is easily seen to be equivalent to

$$\left\langle 2 \times 2 \to A \Big|^{d}_{[\mathsf{id}_a, f, g]} \right\rangle$$

where d is the "cubical horn"

$$\left(\begin{tikzcd} \cdot \arrow[r, "\mathsf{id}_a"] \arrow[d, "f"'] & \cdot \\ \cdot \arrow[r, "g"'] & \cdot \end{tikzcd}\right) \longrightarrow \left(\begin{tikzcd} \cdot \arrow[r, "\mathsf{id}_a"] \arrow[d, "f"'] \arrow[dr] & \cdot \arrow[d] \\ \cdot \arrow[r, "g"'] & \cdot \end{tikzcd}\right)$$

But since 2×2 is the pushout of two copies of Δ^2 over their diagonal faces, our type is now also equivalent to

$$\sum_{k:\hom_A(a,c)} \left(\hom^2_A \left(\begin{tikzcd} & b \arrow[dr, dash, "g"] & \\ a \arrow[ur, dash, "f"] \arrow[rr, dash, "k"'] & & c \end{tikzcd} \right) \times \sum_{h:\hom_A(a,c)} \hom^2_A \left(\begin{tikzcd} & a \arrow[dr, dash, "h"] & \\ a \arrow[ur, dash, "\mathsf{id}_a"] \arrow[rr, dash, "k"'] & & c \end{tikzcd} \right) \right)$$

Now by Proposition 5.10, we have

$$\left(\sum_{h:\hom_A(a,c)} \hom^2_A \left(\begin{tikzcd} & a \arrow[dr, dash, "h"] & \\ a \arrow[ur, dash, "\mathsf{id}_a"] \arrow[rr, dash, "k"'] & & c \end{tikzcd} \right) \right) \simeq \sum_{h:\hom_A(a,c)} (h = k),$$

which is contractible. Thus, it remains to consider

$$\sum_{k:\hom_A(a,c)} \hom^2_A \left(\begin{tikzcd} & b \arrow[dr, dash, "g"] & \\ a \arrow[ur, dash, "f"] \arrow[rr, dash, "k"'] & & c \end{tikzcd} \right)$$

which is contractible if and only if A is a Segal type. $\square$

What's next?

Future work: improving Rzk language

1. convenient syntax;
2. shape types;
3. user-defined cubes and topes;
4. user-defined (higher) inductive types;
5. type inference based on (Kudasov 2023);
6. implicit parameters, auto bound arguments à la Lean;

Future work: improving Rzk tools

1. Rzk InfoView for VS Code;
2. Automatic documentation with rich source code highlighting and diagrams;
3. Interactive diagrams?

Future work: formalisation

1. Complete formalisation of (Riehl and Shulman 2017);
2. Formalise synthetic fibered ∞-categories (Buchholtz and Weinberger 2023);
3. Explore formalisations of cubical, globular type theories.

Thank you!

- Abramsky, Samson and Bob Coecke (2009). "Categorical Quantum Mechanics". In: *Handbook of Quantum Logic and Quantum Structures*. Ed. by Kurt Engesser, Dov M. Gabbay, and Daniel Lehmann. Amsterdam: Elsevier, pp. 261–323. DOI: https://doi.org/10.1016/B978-0-444-52869-8.50010-4.
- Baez, John C. and Blake S. Pollard (2017). "A compositional framework for reaction networks". In: *Reviews in Mathematical Physics* 29.09, p. 1750028. DOI: 10.1142/S0129055X17500283. eprint: https://doi.org/10.1142/S0129055X17500283.
- Buchholtz, Ulrik and Jonathan Weinberger (2023). "Synthetic fibered $(\infty, 1)$-category theory". In: *Higher Structures* 7 (1), pp. 74–165. arXiv: 2105.01724 [math.CT].

Coecke, Bob, Mehrnoosh Sadrzadeh, and Stephen Clark (Mar. 2010). "Mathematical Foundations for a Compositional Distributional Model of Meaning". In: *Lambek Festschrift Linguistic Analysis* 36.

Fong, Brendan and David I Spivak (2018). *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. arXiv: `1803.05316 [math.CT]`.

Kudasov, Nikolai (2023). *E-unification for Second-Order Abstract Syntax*. To appear in *FSCD-2023*. arXiv: `2302.05815 [cs.LO]`.

- Licata, Daniel R. and Robert Harper (2011). "2-Dimensional Directed Type Theory". In: *Electronic Notes in Theoretical Computer Science* 276. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), pp. 263–289. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2011.09.026. URL: https://www.sciencedirect.com/science/article/pii/S1571066111001174.
- Riehl, Emily and Michael Shulman (2017). "A type theory for synthetic ∞-categories". In: *Higher Structures* 1 (1). arXiv: 1705.07442 [math.CT].
- Swierstra, Wouter (2008). "Data types à la carte". In: *Journal of Functional Programming* 18.4, pp. 423–436. DOI: 10.1017/S0956796808006758.