Introduction to the Research Area: Homotopy Type Theory and Higher Algebra.

Louise Leclerc

September 26, 2024

1 Introduction

From now on, (Homotopy) Type Theory has become an extremely practical synthetic tool for formulating questions of (higher) algebra and homotopy theory. It is a language of mathematical discussion, conducive to formalizing questions or statements similarly to Set Theory. However, two major differences stand out between these two languages. The first notable difference is the constructivist flavor of type theory, whereas the Mathematics as usually formalized in ZFC are classical and non-constructive. While one can obtain the existence of functions (typically out of choice) in Set Theory by arguments that does not explain their construction or their computational behavior, in Type Theory functions can be used to calculate return values, in the same way as a computer program does. The second difference between the two theories is that the first allows to naturally formulate questions of Higher Category Theory, where one can think of the objects described as being simplicial complexes, or topological spaces up to homotopy. This stems from the fact that types naturally carry a structure of *weak ω-groupoid* [15]. (For a historical introduction to this homotopic understanding of type theory, see [4])

2 Types and Logic

2.1 What is a type theory ?

Historically, the story of types theory begins in the 1900s. After the discovery of his paradox, RUSSELL, with the help of FREGE, formulate the first ideas of type theory. In order to avoid self reference in a formal logic system, the key idea is to hierarchise the objects, by giving a tag to each one: its type. In logic, this gives an alternative to set theory where instead of having every object being a set, every object is now a point a of some type A, A being uniquely determined. This is to be related (but not confused) with the mebership relation $\in$ of set theory: in a *first approach*, we may understand $a : A$ as $a \in A$. But a more accurate account should be to view $a : A$ as a shape a in A, as a "generalized element" $U \to A$. Since in such systems it is not possible to have $A : A$, this prevents Russell-type paradoxes from appearing. But since its advent, types theory have become a much wider field, used in Computer Science or as a logical framework suitable for constructive mathematics.

In computer science, the notion of *type* or *typing system* is closely related to functionnal programming and more generally soundness of programs. In a large class of language, as C++ or JAVA, there is a rudimentary form of typing. It consists in flags attributed to variables, as `int`, `float` or `char`, which allows to distinguish between datas that should represent an integer, a decimal or a letter even if in the computer they all turns out to be represented as 0s and 1s. This class of language is called *weakly typed* but it is only the basic idea of type theory. There are also so-called *strongly typed* languages, and they usually corresponds to fonctionnal programming ones, such as OCaml or Haskell. In functional programming, everything is a function. You can define a function which takes in argument a function and returns a function, and even an entire program may be considered as a huge function. In those languages, when defining a function you need to specify the type of its argument and the type of its return value. If a function f eats elements of type A and spits out elements of type B, it will be given the type $A \to B$ and in this way, every object will come with a well defined type. The strong typing avoid problems of non-terminating functions. For exemple, it

is not possible to define a function which maps a function f to $f(f)$ because there is no way to attribute a type to such a function.

The notion of *typing system* that interests us is of this last form. It consists of a family of rules to derive the type of a term. This is really close (and actually closely related) to the natural deduction in logic. In order to obtain a *typing judgment* such as $a : A$, you should build a whole tree where the rules are nodes and the judgment appears as the root of the tree.

2.2 The Intuitionistic Theory of Type

I will describe now the *Intuitionistic Theory of Type*, originally formulated by MARTIN-LÖF in 1972 [6]. In what follows, conversely to Set Theory, we will not rely on the first order logic. Indeed, as we will further explain the operations on types will also act as logical operators. First, there is a list of "basic types", that we will use as building blocks to build more complicated ones. They all go with **constructors** or **native elements**, which are *the only way* to introduce elements of this type.

- There is an **empty** type, which we will denote $\bot$ or $\mathbb{0}$. There is no native element or constructor of this type.
- There is a **unit** type, which we will denote $\top$ or $\mathbb{1}$. There is one native element of this type, which we denote $*$. Hence, we may write $* : \mathbb{1}$.
- There is a type of **booleans**, which we will denote Bool or $\mathbb{2}$. There is two native elements of this type, which will be called false and true or $\mathsf{0}$ and $\mathsf{1}$. Hence, we may write $0 : \mathbb{2}$ or $1 : \mathbb{2}$.
- There is a type of **natural numbers**, which we will denote $\mathbb{N}$. There is one native element 0, so we have $0 : \mathbb{N}$. And there is one constructor succ. Given an element $n : \mathbb{N}$, we also have $\mathsf{succ}(n) : \mathbb{N}$. This is to be related with PEANO's formulation of Arithmetic. For exemple, we have $\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0))) : \mathbb{N}$ and we should also write the term $\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0)))$ as "3".

Now we will also give types operations. There are really similar to the operation that we may form with sets, such as the cartesian product or the disjoint union.

- When both A and B are types, we may form their **arrow type** $A \to B$. It is the type of maps from A to B. Their is one constructor for this type, which will be denoted λ. Given an expression $\Phi(a) : B$ which may depend on the free variable $a : A$, we may form the element $\lambda(a : A).\, \Phi(a) : A \to B$. It is akin to the standard notation $a \mapsto \Phi(a)$, which mathematicians are more familiar with. As a first exemple, we may define the function $\mathsf{plus\_two} : \mathbb{N} \to \mathbb{N}$, which adds two to a given natural number, as follows : $\lambda(n : \mathbb{N}).\, \mathsf{succ}(\mathsf{succ}(n))$. When $f : A \to B$, and $a' : A$, we may always form the expression $f(a')$, and we have $f(a') : B$. When f is $\lambda(a : A).\, \Phi(a)$, then this expression *reduces* to $\Phi(a')$. In our previous exemple, this means that our function $\mathsf{plus\_two}$ *computes* as expected. For instance: $\mathsf{plus\_two}(1)$ is the same thing as $\mathsf{succ}(\mathsf{succ}(1))$, which is, by definition, 3. We have a native element $\mathbb{N} \to \mathbb{N}$, which is the constructor succ.
- When A and B are a type, we may form their **cartesian product** type $A \times B$. The only constructor for this type is the *pairing operation* $\mathsf{pair} : A \to B \to A \times B$. This means that the only way to build an element in $A \times B$ is to give its first and second component. Hence, if $a : A$ and $b : B$, then $\mathsf{pair}(a)(b) : A \times B$. We will also more suggestively denote this element by (a, b). For instance, $(\mathsf{succ}(\mathsf{succ}(2)), *) : \mathbb{N} \times \mathbb{1}$, and $(\mathsf{plus\_two}, \mathsf{false}) : (\mathbb{N} \to \mathbb{N}) \times \mathsf{Bool}$.
- When A and B are a type, we may form their **sum** type $A + B$. This is to be related with the usual disjoint union. Their are two constructors for this type, which are $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. This means that to produce an element $c : A + B$, we may use an element from $a : A$ and define c as $\mathsf{inl}(a)$ or an element $b : B$ and define c as $\mathsf{inr}(b)$. For instance $\mathsf{inl}(*) : \mathbb{1} + \mathbb{N}$.

At this point, I need to introduce the concept of **Universes**. It is akin to the concept of universes in Set Theory, as introduced by TARSKI or GROTHENDIECK. It seems natural to ask whether there is a "type of types". However, stated like this, it will eventually leads to RUSSELL or BURALI-FORTI style paradoxes. Hence, here again we need to hierarchise. We will not assume *one* universe of "all types", but a whole "cumulative" hierarchy of such.

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$$

And we assume $\mathbb{0}$, $\mathbb{1}$, $\mathbb{2}$, $\mathbb{N}$ to be of type $\mathcal{U}_0$. Moreover, we ask for each universe to be closed under product, sum, and exponentiation (arrow types). Hence our previous rules stated as "When A and B are types, then so is $A \times B$" should be read as if A, $B : \mathcal{U}_n$, then $A \times B : \mathcal{U}_n$. We may also now consider $+$, $\times$, $\to$ to be maps $\mathcal{U}_n \to \mathcal{U}_n \to \mathcal{U}_n$. Since we think of $\mathcal{U}_{n+1}$ as beeing bigger than $\mathcal{U}_n$, then we will suppose a cumulativity. There is two ways to handle this. The first one is to allow elements to have several types in this specific case. Precisely, if $A : \mathcal{U}_n$, then $A : \mathcal{U}_{n+1}$. But this removes the desirable feature that everything gets one and only one type. A "cleaner" way of doing this is to postulate the existence of *lifting maps*, $\uparrow : \mathcal{U}_n \to \mathcal{U}_{n+1}$. preserving all the properties that we need in order that $\uparrow A$ behaves exactly as A does. But we will not worry so much about this subtlety.

Now, I will describe how to define functions out of some type. In type theory, every function is made from small bricks, namely the constructors and the recursors. Roughly, a constructor explains how to map *into* a certain type, and the recursor allows to map *out* of a certain type. So for the 7 points introduced above, I explain now what are the recursors.

- For the empty type, since there is no way to introduce an element, you can define a map out of $\mathbb{0}$ without having to specify anything. This is really similar to what happens in set theory with the empty set. Hence the recursor say the following : For each universe $\mathcal{U}$ and type $A : \mathcal{U}$, there is a map $\mathsf{rec}_{\mathbb{0}}(A) : \mathbb{0} \to A$.

- For the unit type, since there is only one native element $* : \mathbb{1}$, specifying a map out of unit is the same as picking a value for $* : \mathbb{1}$. Hence, the recursor say the following. If A is a type and $a : A$, then there is a map $\mathsf{rec}_{\mathbb{1}}(A, a) : \mathbb{1} \to A$. Moreover, we have the *reduction* (also called *computation*) rule $\mathsf{rec}_{\mathbb{1}}(A, a)(*) \equiv a$.

- For the type of booleans, there is two native elements, hence we need to specify the value of both. If A is a type, and a_0, $a_1 : A$, then $\mathsf{rec}_{\mathbb{2}}(A, a_0, a_1) : \mathbb{2} \to A$. And we have the computation rules $\mathsf{rec}_{\mathbb{2}}(A, a_0, a_1)(0) \equiv a_0$ and $\mathsf{rec}_{\mathbb{2}}(A, a_0, a_1)(1) \equiv a_1$.

- For the type of natural numbers, there is one native element and one constructor $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. Hence, in order to define a map $\mathbb{N} \to A$ for a certain type A, we should specify the image of 0, and how succ acts through the map we are defining. The recursor for $\mathbb{N}$ works as follows : Given a type A, an element $a : A$ and a map $\phi : \mathbb{N} \to A \to A$, we have a map $\mathsf{rec}_{\mathbb{N}}(A, a, \phi) : \mathbb{N} \to A$, which compute as follows:

  - $\mathsf{rec}_{\mathbb{N}}(A, a, \phi)(0) \equiv a$
  - $\mathsf{rec}_{\mathbb{N}}(A, a, \phi)(\mathsf{succ}(n)) \equiv \phi(n, \mathsf{rec}_{\mathbb{N}}(A, a, \phi)(n))$

  This is really as defining by induction a function when programming. Indeed, recursors allows us to "pattern match" as in some languages (OCAML, HASKELL, RUST...).

- For the arrow type there is not really a recursor. The way you use function is by evaluating them. But to my knowledge there is no recursor for this type.

- In order to define a map $A \times B \to C$ out of a cartesian product, what you need is a map $f : A \to B \to C$. Since the only way to introduce an element $c : A \times B$ is by pairing together two elements $a : A$ and $b : B$, then f should suffice to define the map $A \times B \to C$. If C is a type and $f : A \to B \to C$, then we have a map $\mathsf{rec}_{\times, A, B}(C, f) : A \times B \to C$ with the computation rule $\mathsf{rec}_{\times, A, B}(C, f)(pair(a)(b)) \equiv f(a)(b)$.

- if A and B are types, the elements of $A + B$ are introduced by $\mathsf{inl} : A \to A + B$ or $\mathsf{inr} : B \to A + B$. Hence when C is a type, in order to define a map $A + B \to C$ it suffice to deal with those two cases. If $f : A \to C$ and $g : B \to C$, then we have $\mathsf{rec}_{+, A, B}(C, f, g) : A + B \to C$ with the computation rules

  - $\mathsf{rec}_{+, A, B}(C, f, g)(\mathsf{inl}(a)) \equiv f(a)$
  - $\mathsf{rec}_{+, A, B}(C, f, g)(\mathsf{inr}(b)) \equiv g(b)$

Now, notice that the introduction of universes allows us to define **types families**. That is maps $A \to \mathcal{U}$ for A a type and $\mathcal{U}$ a certain universe. For exemple, we may define by recursion on $\mathbb{N}$ the finite types as

$$f(0) :\equiv \mathbb{0} \qquad \text{and} \qquad f(\mathsf{succ}(n)) :\equiv \mathbb{1} + f(n)$$

That is :

$$f :\equiv \mathsf{rec}_{\mathbb{N}}(\mathcal{U}_0, \mathbb{0}, \lambda(n:\mathbb{N}).\lambda(A:\mathcal{U}_0).(\mathbb{1}+A))$$

There are 3 more type operations that we may introduce now : the **dependant sum** Σ, the **dependant product** Π and the **identity type** Id. I describe the first two of them, which are respectively generalizations of the cartesian product and the arrow type. And we will come back to the latter in the section "Types and Geometry".

- Given a type A and a type family $B : A \to \mathcal{U}$ (we say that B is indexed by A or a type family *over* A), we may define their **dependant product**: $\prod_{(a:A)} B(a)$. It is the type of **dependant functions** f such that, for every $a : A$, $f(a) : B(a)$. The adjective "dependant" here comes from the fact that the type of $f(a)$ *depends* on the element $a : A$. It is a key feature of our type theory, and the reason why this theory is called a **dependant type theory**. This is a generalization of the type of functions, hence we will stick to the notation λ for the constructor. If $\Phi(a)$ is an expression potentially depending on the free variable $a : A$, such that, for every $a : A$, $\Phi(a) : B(a)$ then we may form the term $\lambda(a:A).\Phi(a) : \prod_{(a:A)} B(a)$. We have the expected computation rule : $(\lambda(a:A).\Phi(a))\,(a') \equiv \Phi(a')$. Notice that, for any type C (which does not depends on $a : A$), $A \to C$ is the same type as $\prod_{(a:A)} C$.
- Given a type A and a type family $B : A \to \mathcal{U}$, we may define their **dependant sum**: $\sum_{(a:A)} B(a)$. It is the type of **dependant pairs** (a, b) with $a : A$ and $b : B(a)$. Here again, the element b lives in a type which may depends on the first argument. To construct an element of this type, we need an element $a : A$ and an other one $b : B(a)$, then we may form the element $\mathsf{pair}(a)(b) : \sum_{(a:A)} B(a)$. Hence pair has the type $\prod_{(a:A)} \left(B(a) \to \sum_{(a':A)} B(a')\right)$. Notice that here again, the dependant sum and the constructor pair generalize the product in the case B constant. As with the product, we will write more suggestively (a, b) for the element $\mathsf{pair}(a)(b)$. The recursor for this type has the following form: Given a type C,

  $$\mathsf{rec}_{\Pi,A,B}(C) : \prod_{a:A}(B(a) \to C) \to \left(\sum_{a:A} B(a)\right) \to C$$

  . Which means that given, for each $a : A$, a map $\phi(a) : B(a) \to C$, we have a map $\mathsf{rec}_{\Pi,A,B}(C, \phi) : \left(\sum_{(a:A)} B(a)\right) \to C$. It satisfies the computation rule, for $a : A$ and $b : B(a)$ $\mathsf{rec}_{\Pi,A,B}(C, \phi)((a, b)) \equiv \phi(a)(b)$. This is a generalized form of *curryfication*.

Before going further, there is some conventions that I will describe. Firstly, the arrow are right associative, which means that $A \to B \to C \to D$ should be parsed as $A \to (B \to (C \to D))$. Secondly, every expression located after a binder Σ and Π is by default considered as being inside its scope. Which means that $\prod_{(a:A)} B(a) \to C$ must be parsed as $\prod_{(a:A)} (B(a) \to C)$, and not $\left(\prod_{(a:A)} B(a)\right) \to C$. Finally, we implicitely currify functions : We work with $f : A \to B \to C$, we write often $f(a, b)$ instead of $f(a)(b)$. Even if it do not work out the details (the interested reader can find the proof in [14]), there is as expected an isomorphism between $A \times B \to C$ and $A \to B \to C$, or more generally between $\left(\sum_{(a:A)} B(a)\right) \to C$ and $\prod_{(a:A)} B(a) \to C$.

2.3 The CURRY-HOWARD correpondance

Instead of giving a precise theorem or isomorphism, I will try to give the flavour of what is the CURRY-HOWARD correpondance. This is about types and logic. More precisely, there is a correspondance between writing a proof of a given *logical statement*, and defining a type theoretic expression of a certain *type*. In this philosophy, elements of a certain type will correspond to proofs of the associated logical statement.

Let A and B be two types. And think of it as being logical propositions. We also think of any element $a : A$ as a certain proof of A. Then what does it mean to proove the logical implication $A \to B$? It means than we have a way to turn an evidence of A into an evidence of B. Hence, whenever A, then we should have B to. This is exactly what we are doing when writting down a proof of $A \to B$: We suppose A, then we try to produce an evidence of B. In the same fashion, building an evidence of the proposition $A \wedge B$ will be the same as giving an element of the type $A \times B$. When *constructively* proving that $A \vee B$ holds, we give either a proof $a : A$ that A holds, or a proof $b : B$ that B holds, hence $A \vee B$ should corresponds to $A + B$. Actually, $A + B$ is not exactly the type corresponding to $A + B$ but we will not dive in those

details for now. You may also interprete the truth value as $\top$ and the false value as $\bot$. Hence, the negation of the proposition associated with the type A will be associated to the type $\neg A :\equiv (A \to \bot)$. Notice that the constructors and recursors gives us the logical laws of introduction and elimination. For exemple, the explosion principles is given by the recursor of $\bot$, and the *modus ponens* is given by the evaluation map $f \mapsto a \mapsto f(a) : (A \to B) \to A \to B$.

What about the first order logic then ? Suppose that A is a type and $P(a)$ is another type, depending on $a : A$. We think of $P(a)$ as proposition depending on $a : A$. What does it mean to prove $\forall(a : A)P(a)$? In order to derive such a statement, we assume $a : A$ is an arbitrary element, and we try to find a proof $\pi(a)$ that $P(a)$ holds. Hence, it is the same as defining a dependant map π of type $\prod_{(a:A)} P(a)$.
For the existential, it works in a similar way with the dependant sum : when *constructively* proving $\exists(a : A)P(a)$, we pick a certain $a : A$ and we show that $P(a)$ holds. Hence, it correponds to giving an element $(a, p) : \sum_{(a:A)} P(a)$. (Here we have the same issue as before, but lets keep to this interpretation for now.)

And this is how the CURRY-HOWARD correpondance works in practice. Indeed, this is the conceptual base of proofs assistants such as COQ, LEAN or AGDA. When working with a proof assistant, you translate and formalize the logic as types, translate theorems as complicated types, and try to define elements of those types. Here is an exemple of proof that the double negation of the excluded middle holds: Given a type A, the excluded middle for A is $A + \neg A$, its double negation is $\neg\neg(A + \neg A)$ Hence to give an evidence of this, we need to exhibite a map of type $((A + \neg A) \to \bot) \to \bot$. So lets pick a map $\phi : (A + \neg A) \to \bot$. Then, by composing with inl, we have a map $\phi \circ \mathsf{inl} : A \to \bot$. Hence an element $\phi \circ \mathsf{inl} : \neg A$. We may now apply our function ϕ to the argument $\mathsf{inr}(\phi \circ \mathsf{inl})$ to get an element $\phi(\mathsf{inr}(\phi \circ \mathsf{inl})) : \bot$. Finally, this means that

$$\lambda(\phi : (A + \neg A) \to \bot).\, \phi(\mathsf{inr}(\phi \circ \mathsf{inl})) \ : \ \neg\neg(A + \neg A)$$

And this is our proof that the double negation of the excluded middle holds !

3 Types and Geometry

3.1 Paths, paths between paths and the higher homotopy interpretation

3.1.1 The identity type

Pursuing in the interpretation of *types as propositions*, the logical statement "$x = y$" for x and y two terms of a given type A may also be internalised as a type. This idea come from the intuitionist type theory, as first formulated by Per MARTIN-LÖF in [6]. We should stick to this idea by defining an identity type over any type A, or more precisely a family of identy types indexed by the elements of A as follow:

$$\mathsf{Id}_A : A \to A \to \mathcal{U}$$
$$\text{for all } a : A,\ \mathsf{refl}_a : \mathsf{Id}_A(a, a)$$

So for any elements a, b of type A, we have a type of equalities in A, namely $\mathsf{Id}_A(a, b)$, and we only know one way to construct an element for the identity type: when $a \equiv b$, and using the constructor $\mathsf{refl}_a : \mathsf{Id}_A(a, a)$. And this should capture the notion of equality.

for readability, we will now write the equality type with the usual symbol "=", so $a = b$ will be a notation for the type Id $a\, b$, and we may also write "$a =_A b$" to specify the type A.

Now, as with every other type construction in type theory, one should define a recursor for this type, in order to know how to use an equality to produce other statements. We may give two recursors, and refer the reader to the HoTT reference for the proof that they are equivalent.

Path induction:

$$\mathsf{ind}_{=_A} : \prod_{(C:\prod_{(x,y:A)}(x =_A y) \to \mathcal{U})} \left(\prod_{(x:A)} C(x, x, \mathsf{refl}_x)\right) \to \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

with the computation rule, for every $a : A$, $\mathsf{ind}_{=_A}(C, f, a, a, \mathsf{refl}_a) :\equiv f(a, \mathsf{refl}_a)$

Based path induction:

$$\mathsf{ind}'_{=_A} : \prod_{(a:A)} \prod_{(C:\prod_{(x:A)}(a =_A x) \to \mathcal{U})} C(a, \mathsf{refl}_a) \to \prod_{(x:A)} \prod_{(p:a=_A x)} C(x, p)$$

with the computation rule, for every $a : A$, $\mathsf{ind}'_{=_A}(a, C, d, a, \mathsf{refl}_a) :\equiv d$

What is worth noticing from those recursor is that they tells us the equality behave as the LEIBNIZ equality : if $a =_A b$ then every property which holds for a will hold for b and inversely. But as any other type, the identity type could carry more structure that the mere propositionnal one, and should be thought as an algebraic structure. Indeed this consideration leads to *Homotopy* Type Theory, which produce the higher algebraic flavor as we will describe it later.

One can recover the usual properties of equality and the fact that it is an equivalence relation. This may be stated as the existence of the following functions (Those functions may be defined by path induction, and the reader should refer to [14] for this construction and the proofs of the following statement):

Definition 3.1 : concatenation, inversion

Given a type A and $x, y, z : A$, one may form the **concatenation** of any two elements $p : x =_A y$ and $q : y =_A z$. This concatenation is denoted $p \cdot q$ and has type $x =_A z$
Given a type A and $x, y : A$, one may form the **inverse** any $p : x =_A y$. This inverse is denoted p^{-1} and has type $y =_A x$

Moreover, given a type A, those operations respects the following properties :

unit for $\cdot$:
For every $x_0, x_1 : A$ and for every $p : x_0 = x_1$,

$$\mathsf{refl}_{x_0} \cdot p = p \quad \text{and} \quad p \cdot \mathsf{refl}_{x_1} = p$$

associativity of $\cdot$:
For every $x_0, x_1, x_2, x_3 : A$ and for every $p : x_0 = x_1, q : x_1 = x_2, r : x_2 = x_3$,

$$p \cdot (q \cdot r) = (p \cdot q) \cdot r$$

inversion :
For every $x_0, x_1 : A$ and for every $p : x_0 = x_1$,

$$p \cdot p^{-1} = \mathsf{refl}_{x_0} \quad \text{and} \quad p^{-1} \cdot p = \mathsf{refl}_{x_1}$$

And this structure may be resumed as follows:

Theorem 3.2

the families of identity types $(x =_A y)_{x y : A}$ is equiped with a structure of groupoid.

3.1.2 The homotopical point of view

As previously mentionned, one should consider that two elements of a given type may be equal in differents ways. Thinking as equality beeing paths, and types as spaces, we may imagine that two points could be linked along distincts paths. In fact, this interpretation is really accurate in our context, and provides us a convenient way to think about identity types. For example, the groupoid structure explicited in the previous section may be interpreted as the path-groupoid structure of a space. Following this interpretation, the concatenation of identity proofs now become the concatenation of paths, and the inversion become the reversal.

We may take this *space-as-types* interpretation further. Assume we have a type A, two elements $a, b : A$ and two paths $p, q : a =_A b$ between them. Then in the geometric setting, we may have a whole diversity of homotopies between p and q. This may be considered to in type theory, and it will go by the name

$$p =_{a =_A b} q$$

In fact, we already mentionned the existence of elements in this kind of type in the previous section, by claiming the unitarity, associativity and inversibility properties for identity proofs operations. And as in homotopy theory, there is no reason to stop at level two, and one would even consider types of the form

$$\alpha =_{p =_{a =_A b} q} \beta$$

Which we should write more readibly as $\alpha = \beta$. As before, inhabitants of this kind of type could be properties about the proof of associativity of types, or the proof of inversibility... The point is that more than beeing a convenient way to think about identity type, this perspective guide us towards defining objects akin to classic homotopy theoretic ones. And beautiful constructions of usual geometry may be reproduced in Homotopy Type Theory, such as loopspaces, n-spheres, homotopy groups, suspensions, universal covers, Hopf fibrations, and so on. One may also prove type theoretic formulations of standard theorems such as the WHITEHEAD principle, or the long exact sequence induced from a fibration.

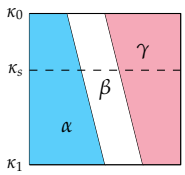
4 Higher Algebra

4.1 Homotopy associativity

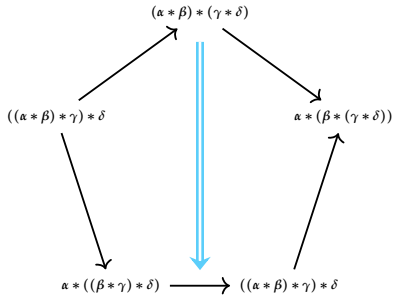
Prototypical instances of "groups" which are not strictly associative are given by loopspaces. In algebraic topolgy, there is no god-given concatenation of paths in a space, but rather a whole family of such, depending on the choice of a parametrizing of the paths. For instance, one could define the composition of two paths γ and δ as

$$\gamma * \delta : t \mapsto \begin{cases} \gamma(2 * t) & \text{if} \quad t \leq \frac{1}{2} \\ \delta(2 * (t - 1/2)) & \text{if} \quad t > \frac{1}{2}. \end{cases}$$

Moreover, the composition of three paths will only be defined up to a reparametrizing. Given a choice of a binary composition $*$, we can think of a family $(\kappa_s)_{s \in [0,1]}$ of admissible concatenations of three paths α, β and γ, transitioning from $(\alpha * \beta) * \gamma$ to $\alpha * (\beta * \gamma)$.



Now, one may push this observation on step further by noticing that even those homotopies should satisfy a coherence law on their own. Namely, there is a *pentagon homotopy* interpolating between the two ways to go from $((\alpha * \beta) * \gamma) * \delta$ to $\alpha * (\beta * (\gamma * \delta))$ by using our previous κ's.



And so on: the tower of coherence keeps growing and gets more and more complicated. Thus the main concern is to undertand and formalize all this data. The historical approach is the one of STASHEFF and goes back to the 1960's [13]. Generalizing from the previous observation, STASHEFF axiomatize a *homotopy associative H-Space* to be given by:

- a space X.
- for each $n \geq 2$, a family of operations $\mu_n : K_n \times X^n \to X$.
- having a neutral element, and satisfying compatibility conditions between the μ_n's.

And an important result is the *recognition principle* which tells us that such a homotopy associative H-space is a loopspace precisely whenever every element admits a (weak) inverse. i.e. *grouplike* homotopy associative H-space are exactly the spaces of loops $\Omega(X, x)$ of some pointed space (X, x). Such objects are also called A_∞*-groups*, E_1*-groups*, or more concisely ∞*-groups*. More generally, there is a notion of E_n*-group* (or monoid) for $n \geq 1$, corresponding to the structure of a n-fold loopspace $\Omega^n(X, x)$. They are also caled more concisely (∞, n)*-groups*.

4.2 ∞-groups in HoTT

In homotopy type theory, one may use the following trick to work with ∞-groups: Since they are precisely the loopspaces, we can identify ∞-groups with loopspaces. That is, we may let

$$\begin{aligned} \infty - \mathsf{Group} &:\equiv \Sigma_{G:\mathcal{U}} \Sigma_{X:\mathcal{U}_*^{>0}} G \simeq \Omega X \\ &\simeq \mathcal{U}_*^{>0} \end{aligned}$$

where $\mathcal{U}_*^{>0}$ is the type of *connected and pointed* types (think of pointed, connected spaces).

Note that the theory of n-groups or 1-groups (a.k.a groups) may be embedded in this formalism by asking for the delooping to be n-truncated. That is:

$$\begin{aligned} n - \mathsf{Group} &:\equiv \Sigma_{G:\mathcal{U}^{<n}} \Sigma_{X:\mathcal{U}_*^{>0}} G \simeq \Omega X \\ &\simeq \mathcal{U}_*^{>0, \leq n} \quad . \end{aligned}$$

In particular, the usual category of groups may be recover as the category of pointed and connected groupoids; by considering their automorphism group.

More generally, (∞, k)-groups may be defined as types which may be delooped k-times:

$$\begin{aligned} (\infty, k) - \mathsf{Group} &:\equiv \Sigma_{G:\mathcal{U}} \Sigma_{X:\mathcal{U}_*^{>0}} G \simeq \Omega^k X \\ &\simeq \mathcal{U}_*^{\geq k} \quad . \end{aligned}$$

The theory of higher groups is very rich, and for instances we can talk about their (higher) categories, their representations, their central extensions or cohomology. Thanks to this definition, it has been possible to formalize and prove synthetically some properties of those objects, as did BUCHHOLTZ in [2] for example.

Sadly, this method does not generalize well to other kinds of higher structures, and currently most of the higher algebra (homotopy coherent monoids, rings, or LIE algebras for instance) seems out of reach. For their usual definition involves space valued presheaves or infinite towers of coherences as described above, which we don't know how to express - or even if it is possible to do so - in HoTT. For instance, SEGAL as a method to define homotopy coherent monoids as presheaves over the augmented simplicial category Δ^+. But reproducing this construction in homotopy type theory is almost the same as defining what are called *simplicial types*. Which happens to be one of the most (if not *the most*) major open problem in HoTT.

4.3 Perspectives on higher algebra in HoTT and extensions of HoTT

Currently, it seems that several key problems obstruct a full-fledged formalization of higher algebra in HoTT, especially ∞-categories. And most of them appear to be related in some sense. The big open questions to achieve this goal are the following:

- Defining simplicial types.
- Formalizing $(\infty, 1)$-categories.
- Finding a way to encode towers of coherences.
- Making HoTT eat itself, known as "autophagy".

Where the last point means defining HoTT in HoTT (in the same way that one can define what is a model of set theory in the language of set theory for instance). And it begin to be a strong consensus that all of these problems should be correlated.

So solving one of them would probably unlock the other ones. There are several approaches to solve them, which are presented in the following paragraphs.

4.3.1 Directed Homotopy Type Theories

This approach focuses on the first two points. The idea is to enhance the fundations of HoTT to obtain a broader formalism in which types would be interpreted as ∞-catgeories. That is, one would replace the family of identity types $(x =_A y)_{x,y:A}$ on a type A by a family of *hom types* $(\mathsf{hom}_A(x,y))_{x,y:A}$. Hence types would be "directed" in the sense that it would not be possible to define an inversion $(-)^{-1} : \mathsf{hom}_A(x,y) \to \mathsf{hom}_A(y,x)$. In fact, such a theory should have a hom-induction principle really similar to the path-induction principle presented in 3.1.1:

$$\mathsf{ind}_{\mathsf{hom}_A} : \prod_{(a:A)} \prod_{(F:\prod_{(x:A)}(\mathsf{hom}_A(x,a))\to\mathcal{U})} F(a,\mathsf{id}_a) \to \prod_{(x:A)} \prod_{(u:\mathsf{hom}_A(x,a))} F(x,u)$$

with the computation rule, for every $a : A$, $\mathsf{ind}_{\mathsf{hom}_A}(a, F, d, a, \mathsf{id}_a) :\equiv d$ where id_a is a constructor of $\mathsf{hom}_A(a,a)$ correponding to the *identity* arrow $a \to a$. In this expression, howether, we should add the extra property that F is "covariant" on the variable x. In this way the hom-induction principle could be seen as an ∞-categorical version of the YONEDA lemma ! To clarify a bit the link with the YONEDA lemma, consider the case where $F : A \to \mathcal{U}$ does not depend on $\mathsf{hom}_A(-,a)$. In this case, the induction principle says that defining a map (which we should think of as a natural transformation) $\mathsf{hom}_A(-,a) \to F$ is the same as giving its value on the identity id_a.

There are several attempts to make HoTT directed. The route taken by NORTH in [8] consists in defining the notion of variance (covariance, contravariance) of maps on their variables. Taking variance into account seems to raise new problems, such as handling modalities in type theory, which is another topic of active research in the community.

There is another route, taken by RIEHL and SHULMAN in [9], which consists of axiomatizing the existence of the category of simplexes in HoTT. This way, all types can be "probed" with simplexes, and therefore inherit a structure of *simplicial type*. The resulting theory is then called *simplicial type theory*. Arguably, this is not the more satisfying solution to the problem, but it is currently one of our main progress toward the goal of directed HoTT, and already allowed the authors to prove a YONEDA lemma for $(\infty, 1)$-categories.

4.3.2 Two-level Type Theory

This approach tries to tackle the third point. The idea is to have two coexisting notions of equality in the type theory. One "strict", said *extensional*. The other one "weak", said *intensional*. So using the first one, we may define strict structures as it is done in Set Theory, and for example, the simplicial category Δ can be defined with strict coherences. For instance, the work of ANNENKOV, CAPRIOTTI, KRAUS and SATTLER in [1] yield a successful formalization of $(\infty, 1)$-categories.

4.3.3 Displayed Type Theory

Another approach is to define a notion of co-inductive types which would behave well with the homotopical structure of types. This is a very recent suggestion of KOLOMATSKAIA and SHULMAN [5], which allows a very concise definition of simplicial types, and hence unlock the path to formalizing higher algebra in type theory.

5 Other perspectives and applications of HoTT

Since its advent, HoTT as been used to formalize a very wide palette of Mathematics. Remarkably, a lot of methods have been found to talk about diverse fields of maths which are not just about homotopy theory. For instance, the intensive developement of modal type theories allowed to define *Cohesive HoTT*, usefull to formalize topological properties of spaces (not just homotopical one) as explain by SHULMAN in [12], where he formalized BROUWER's fixed-point theorem. Cohesive HoTT allows us to talk about the topology of a space as the circle $\mathbb{S}^1$, *and* about the homotopy type S^1. Both are diferent, but up to homotopy, one may recover the later from the former.

The understanding of the models of HoTT also helped us to formalize theories such as algebraic geometry in HoTT, or to find admissible extensions of the language. One of the most important advance in this direction was the definition of *Cubical Type Theory* by COHEN, COQUAND, HUBER and MÖRTBERG in [3]. Where they contribute a new constructive interpretation of the univalence axiom.

There is also an active research toward a *linear* version of HoTT, where variables should be thought as ressources, which use in an expression should be limited to some extent. See for instance the notes of VÁKÁR on the subject [16].

Finaly, one may consider the impressive effort of SHREIBER and his collaborators to formalize theoretical physics (differential geometry, quantum physics, gauge theories) in the language of Homotopy Type Theory and their aforementioned extensions. [10], [7], [11].

References

[1] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Mathematical Structures in Computer Science*, 33(8):688–743, May 2023.

[2] Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher groups in homotopy type theory, 2018.

[3] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2016.

[4] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. 04 2002.

[5] Astra Kolomatskaia and Michael Shulman. Displayed type theory and semi-simplicial types, 2024.

[6] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80, pages 73–118. Elsevier, 1975.

[7] David Jaz Myers, Hisham Sati, and Urs Schreiber. Topological quantum gates in homotopy type theory. *Communications in Mathematical Physics*, 405(7), July 2024.

[8] Paige Randall North. Towards a directed homotopy type theory, 2018.

[9] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories, 2023.

[10] Urs Schreiber. Quantization via linear homotopy types, 2014.

[11] Urs Schreiber and Michael Shulman. Quantum gauge field theory in cohesive homotopy type theory. *Electronic Proceedings in Theoretical Computer Science*, 158:109–126, July 2014.

[12] Michael Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory, 2017.

[13] James Dillon Stasheff. Homotopy associativity of h-spaces. i. *Transactions of the American Mathematical Society*, 108(2):275–292, 1963.

[14] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, 2013.

[15] Benno van den Berg and Richard Garner. Types are weak omega-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, oct 2010.

[16] Matthijs Vákár. Syntax and semantics of linear dependent types, 2015.