Homotopy Arithmeticy

by Robert M, Sven Nilsen, 2025

In this paper we introduce a simple model for Homotopy Theory.

Consider the sets of functions of type `bool → nat & (< n)` for some `n`.

Naturally, a function `f : bool → nat` corresponds to a tuple `(a, b)` where:

f(false) = a f(true) = b

In Category Theory, it is natural to think about functions as morphisms. Geometrically, this intuition corresponds to paths between points. However, in Homotopy Theory, morphisms are naturally homotopies, which are higher order paths between paths. There are advantages in language design to jump over points as the objects and choose to start with paths, or functions, as the objects:

- Avoids the need for n-Category Theory, just to get started
- Explicit symmetry and transitivity on objects, which can be weakened when needed
- Free proof semantics by normal Category Theory

Proof semantics comes from paths and higher order paths, not from individual points. For example, in classical logic, a proof `f : bool$^0$ → bool` might get ambiguous when we say “for all inputs”. In most theories, `bool$^0$` normalizes to the `unit` type which only element is `()`, so that `f() : bool`. However, in theories where this is not specified, “for all inputs” can become ambiguous. In Path Semantics, we avoid this problem by saying a proof `f` in classical logic is:

f <=> \true

This means, we do not care about the definition of inputs, but talk about the function identity itself.

In this model, we are considering functions of type `bool → nat`, so there is no notion of “truth” necessarily in the sense of normal logic. One could have different notions of proofs:

f <=> \0 f <=> \1 f <=> \2 … f <=> \n

Geometrically, this corresponds to a loop starting at some point `n` and returning to `n`.

In Category Theory, a category has transitivity of morphisms, written `g · f` for two morphisms `g` and `f`. This binary operator is called “composition”. Transitivity comes from the property that both morphisms map between objects, such that `f : A → B` and `g : B → C`, results in `g · f : A → C`.

Since our objects are functions, or pairs, we can write the kind of transitivity we get in our category:

$f : (a_0, a_1) \to (b_0, b_1)$ & $g : (b_0, b_1) \to (c_0, c_1)$ → $g \cdot f : (a_0, a_1) \to (c_0, c_1)$

This form of transitivity is at the level of homotopies. While this might sound sophisticated mathematical terminology, in geometry this simply corresponds to concatenating two quads that share one edge into a mesh with two quads. With other words, we just glue quads together into meshes and view these meshes as coherent surfaces. Basically: What we mean by surface geometry.

Every category has identity morphisms for every object. In the geometric interpretation of our model, identity morphisms corresponds to a surface looping from an edge back to itself. This is very useful when you want to prove that some mesh describing a surface loops back on itself. However, you should watch out for getting the orientation of the surface right, because in a Möbius topology:

$$(1, 2) \quad \rightarrow \quad (2, 1)$$

This is not the same as an identity morphism and you have to loop twice around the surface to get:

$$(1, 2) \quad \rightarrow \quad (1, 2)$$

Because we get homotopies of lowest order for free by starting with functions as objects, in 3 dimensions, we do not need any n-Category Theory to talk about the orientation of surfaces.

Now, the orientation of a surface can determine whether some continuous mesh has one or two sides. In the case of Möbius topology, there is only one side. On a hollow cylinder with zero thickness, there are two sides, the inside and the outside. However, when we give a hollow cylinder some thickness, there is only one side, because you can “move over the edge” continuously from the outside to the inside and vice versa. By hollowing out the inside of this shape, we get a torus, which has two sides, but the “outside” of the torus is both the inside and outside of a corresponding cylinder. The inside of a torus is not visible in 3 dimensions.

How do we count the surfaces? We collect all edges by quads into meshes, such that no two distinct meshes share a quad. The number of distinct meshes gives the number of surfaces.

This method has a weakness: By manipulating the orientation of edges, we can create a single “front” surface that has no corresponding “back” surface. This can be easily misunderstood, because “one surface” can be interpreted in different ways. To prevent this from happening, we can add symmetry to edges, such that if some edge exists, pointing in one direction `(a, b)`, then the edge pointing in reverse direction also exists `(b, a)`. It guarantees that when a shape has two surfaces, this will result in two distinct meshes. This means we often want symmetry on objects.

In General Relativity, one often works with manifolds that have geodesic curves. This means, for every two points A and B, there is a “shortest” path or “most straightforward” path between them. If there are two edges `(a, b)` and `(b, c)`, then we can define the geodesic from `(a, c)` alone, whatever this means in the definition of the manifold. Now, this might seem only relevant in physics, but geodesics are also important in geometric surgery. When gluing together surfaces, it simplifies the gluing operation significantly, when gluing some edge to `(a, c)`, this automatically glues perfectly to `(a, b)` and `(b, c)` in the precise way we intended to, by choosing the definition of the manifold carefully to our needs. This means we often want transitivity on objects.

While choosing functions as objects for our category does not imply that there are symmetry and transitivity on objects, this is often desirable. We already know that by using a category, we have transitivity on homotopies. However, do we want symmetry on homotopies?

In geometry, a quad from `($a_0$, $a_1$)` to `($b_0$, $b_1$)`, might also be thought of as a quad from `($b_0$, $b_1$)` to `($a_0$, $a_1$)`. Often, it is nice to think about the first quad and identical to the second quad, because when we need a back surface to our quad, we can use `($a_1$, $a_0$)` to `($b_1$, $b_0$)` instead. Swapping the direction of edges is nicer when gluing edges together in topology. However, who says that we only want to glue edges? We might also glue quads together, for example by using quads to form a cube with the source and target quads as top and bottom of the cube. This gluing operation might be along the diagonal of the cube when seen from the side.

With other words, a quad that swap edges by symmetry, might be seen as a generalization of swapping direction of an edge. Whatever algorithm we use to count these volumetric entities (which are called “hyper surfaces” in n-dimensional geometry), we need a way to avoid ambiguity. By using symmetry, we force away an edge case with a Möbius topology on cubes instead of quads. You might not want to do this if you are sure in advance about the topology, because it generates a lot of extra data you might not need. For every cube, there will be additional cubes, just like there is a “front” and “back” surface of a quad. However, we need to think carefully about what this means.

First, going back to edges can help us to think about n-dimensional shapes. A single edge is a pair `(a, b)`. Going up one dimension, we get a quad from `($a_0$, $a_1$)` to `($b_0$, $b_1$)`. Now, we have not actually said where `$a_0$, $a_1$, $b_0$, $b_1$` are located in some space. In principle, you could take any 4 coordinates in space, make any permutation between them, and any permutation would be a valid map. Since we are used to think of quads as approximately flat, when we start with a flat quad here, there will be some other permutation where you get crossings over from diagonal to diagonal. You can twist one edge once, it crosses over; twice in same direction and and it goes back to the original shape. The geometry does not remember how many time you twist something in the same direction. This operation is not continuous, otherwise we might be able to create a helix shape.

One can think about twisting a shape into fixed end points as operations on a group. When there is a finite number of elements in a group and there is no memory of the number of twistings in the shape, this group corresponds to all permutations between the points. It means, every possible configuration of the shape can be derived directly by enumerating all permutations. The index in this discrete space, together with an algorithm that determines the shape, will contain all information we need to identify any configuration. So, this loss of memory, while seemingly sounding as a downside, is an actual advantage we can exploit to improve the efficiency of the algorithms we use. Now, how do you add memory about the number of twistings? You simply add numbers as extra information. You can have one number for the index, plus numbers that tells twistings along various dimensions. However, depending on topology, you should normalize these extra twisting numbers so you have a way to count distinct shapes accurately.

Therefore, the highest possible number of quads we can have connecting `($a_0$, $a_1$)` to `($b_0$, $b_1$)`, is the permutations of `$a_0$, $a_1$, $b_0$, $b_1$`. Any map we create between these permutations can only lower the number of quads. When we assume symmetry over all permutations, it means that once we have one permutation, we get all other permutations somewhere in the model, but these other permutations might not be connected through the same hypersurface. When we only have symmetry between edges `(a, b)` and `(b, a)`, this is like only adding permutations that satisfy some dimensional constraint. With other words, we do not add symmetry over all permutations. When we add a symmetry between quads from `($a_0$, $a_1$)` to `($b_0$, $b_1$)` to quads from `($b_0$, $b_1$)` to `($a_0$, $a_1$)`, this adds another set of permutations constrained to another dimension than the first one for edges.

While this makes it clear what we mean by hypersurfaces, it is still only possible to study edges in our model, while all higher dimensional shapes exists only in the meta-theory about the model.

Remember that all our objects in this model are edges. We just started talking about hypersurfaces in higher dimensions. An edge is a one-dimensional hypersurface, a quad is a two-dimensional hypersurface, a cube is a three-dimensional hypersurface etc. Now, a good question is: Can we use quads and cubes as objects in our model? Yes!

`(($a_0$, $a_1$), ($b_0$, $b_1$))`	We can model a quad by inserting edges inside an edge

Naturally, since a pair is a function of type `bool → nat`, when we model quads this way we get a function of type `$bool^2$ → nat`. For an n-dimensional hypersurface we get `$bool^n$ → nat`.

When we are thinking about natural numbers returned from functions as addresses to geometrical points in some space, we automatically want symmetry over all permutations. This is because any point in space might be transformed into any other, by choice of coordinate systems. Many of these coordinate systems are weird ones, but still, when we want to prove something holds no matter where the points are located in space, we should use symmetry over all permutations.

For proof semantics in this model this insight is significant,
because for every proof `f : $bool^n \to nat$` we can take the permutations of all inputs over an ordered enumeration of the inputs (e.g. increments in binary) and get another proof `g : $bool^n \to nat$` which is not logically equivalent to `f`, but yet `g` encodes the same proof semantics as `f`.

Wait! Hold on! There are proofs which are not logically equivalent, but they mean the same thing?

Yes! However, remember we are used to proofs in logic that only returns `true` for all inputs. These proofs are naturally symmetric over all permutations. From a proof semantical perspective of geometry, all these proofs are geometrically connected through the same hypersurface. With other words, there is only one side to this shape.

When we generalize from logic to proof semantics of Homotopy Theory in this model, we are no longer talking about proofs that only returns `true` for all inputs. Remember, the functions return natural numbers, not booleans. Yet, there is no formal definition that is used in our model to tell us how to consider two proofs as equal. The trick we use instead, is to add a hypersurface connecting two objects in the model whenever we consider them equivalent in some sense, plus a symmetry such that some hyper surface `(a, b)` where `a, b` can be nested, implies that `(b, a)` also exists in the model. What we do not say is that all symmetries over permutations exist, because that is only needed for the meta-theory of that higher-order proof. When we do not need to go to higher level proof semantics, we just stop adding new symmetries and stick to the most trivial symmetry. This trivial symmetry encodes a logical equivalence along the “highest edge”, which means the first boolean parameter of the function:

$$(f : bool^n \to nat) \in \mathbf{S} \quad \to \quad (f^{-1} : bool^n \to nat) \in \mathbf{S}$$

where `$f(0) \iff f^{-1}(1)$ & $f(1) \iff f^{-1}(0)$`

This way, we do not have to worry whether `(b, a)` exists, when we know `(a, b)` exists.
We can just write whatever objects we want to check existence of, in any other.

The reason we do not add all symmetries over permutations, is because it generates a lot of extra data and wastes memory and time in algorithms. We do not want our model to answer every possible question, because we would quickly run out of resources. Instead, we want to focus on the questions that are interesting to us. This means, we use the rules of symmetries that are practical. Including when we do not want edge direction symmetry. Remember, objects in our category are edges. Symmetry is not necessary, neither for objects nor for morphisms in a category. However, it can be nice to have. We can add symmetry for objects. We can also add symmetry for morphisms. The latter turns the category into a groupoid. By starting with edges as objects, we do not have to use n-Category Theory and we only need to “summon” it by need, when it is practical.

Where do morphisms live in memory? When we only work with edges that are pairs of natural numbers, we do not define where morphisms are in memory, because our set `**S**` only contains edges and no higher hypersurfaces. This means that morphisms might live in the meta-theory and be used logically, e.g. when `**S**` is treated as an unknown set (uninterpreted) from some set of axioms. When we add hypersurfaces to `**S**`, we can study morphisms as objects in the model.

We can also weaken symmetry. For example, when we have a rule `(a, b) => (b, a)`, this does not necessarily mean that `(b, a) => (a, b)`. For some particular modeling session of some topology in the model, when you have `(a, b)` and `(a, b) => (b, a)`, you get `(a, b) & (b, a)`. So, you can assume that `(a, b) == (b, a)`. Yet, this only holds in that particular session, not in general for all sessions. In Path Semantics we say that `((a, b) == (b, a))^true`, these two pairs are tautologically equal, when it holds for all sessions. However, this does not imply that you can substitute them for each other as sub-expressions in some hypersurface. You can only add one of them at the top level and the rules will generate the other side by symmetry. No matter which rules are used, you will get the same result as before, but you will not necessarily get some hypersurface with that particular sub-expression substituted for the other.

As a consequence, substitution in sub-expressions is not easy to reason about. When you can substitute `(a, b)` with `(b, a)` and vice versa everywhere, this is geometrically a gluing operation. Gluing operations are not just ways of constructing a geometric shape. From a proof semantical perspective, gluing operations can function as way of substituting sub-expressions. In Path Semantics, tautological equivalence does not mean that you can substitute everywhere. Most operators are normal congruent and only requires normal equivalence `_ == _`, a few operators are only tautological congruent and requires tautological equivalence `(_ == _)^true`, but in principle there can also be operators that are not even tautological congruent. There is no formal theory about when gluing operations are allowed or not in general. When you work with this model of Homotopy Theory, you have to be careful and think through what kind of gluing operations you want to do.

In constructive logic, the proof semantics is language biased toward a single surface. You can make various assumptions and reason about how things are connected, but only from within one surface at a time, or, by adding the HOOO EP axioms, through meta-theory about other surfaces. However, you can not count the number of surfaces in the model. This means you should be aware of the limitations of constructive logic by mathematical language design. However, you can model our model of Homotopy Theory in constructive logic and study it, including counting the number of surfaces. Yet, if the structure within this model “leaks out” to the language of constructive logic, it will “pollute” the environment and might cause logical contradictions. Thinks about the model as studying nuclear physics in some laboratory. Make sure the experiments are controlled and safe.

In summary so far: We use nested pairs to describe hypersurfaces. The symmetry we choose is optional and arbitrary and depends on what is practical for any given project. The main reason we use symmetry is to avoid edge cases where the number of pieces in the space we are studying can be interpreted ambiguously. However, for practical reasons such as limited computational capacity, we can not add all symmetries that are relevant for higher level proof semantics, because it would require all permutations and lead to explosion in combinatorics.

Now it is time to take a step back and start formalizing n-groupoids in this model.

A groupoid structure with morphisms of type `$\text{bool}^n \to \text{nat}$` in some set `**S**`:

$$f \in \mathbf{S} \quad \to \quad f^{-1} \in \mathbf{S}$$

Where `$f^{-1}$` is defined as following:

$$x : \text{bool} \quad \to \quad f^{-1}(x) := f(!x)$$

For example:

$$(1, 2) \in \mathbf{S} \quad \to \quad (2, 1) \in \mathbf{S}$$

The groupoid structure here is on natural numbers as objects, with functions as morphisms. Notice that we do not impose this structure on the model by default, because in the model, objects are functions in the set and morphisms might exist at the level of the meta-theory. However, when we want to formalize n-groupoids in this model, we go down to the level of natural numbers as objects.

Since a groupoid is a category, it also has transitivity:

$$(a, b) \in \mathbf{S} \ \& \ (b, c) \in \mathbf{S} \quad \rightarrow \quad (a, c) \in \mathbf{S}$$

Transitivity and symmetry is the same as a partial equivalence. When we have `(a, b)`, this implies `(b, a)` by symmetry. From transitivity, `(a, b)` and `(b, a)` implies `(a, a) and `(b, b)`. This means, from the perspective of groupoids we have identity morphisms by partial equivalence. We do not need to add reflexivity `(a, a)` for any `a`.

We have not specified what we mean that some natural number `a` is in this groupoid. We can only talk about the morphisms as members of the set `**S**`. There are no objects as natural numbers.

However, there is a trick: When `(a, a) ∈ **S**`, one can write this as `a ∈ **S**`.

Instead of defining what objects are as natural numbers in the groupoid here, we avoid the entire problem by using `a` as a morphism `(a, a)`. It means, we just leave objects undefined. There is no axiom of Category Theory that says explicitly that objects need to defined in some category, hence also for groupoids in general. It might seem like a downside at first, but this is actually an advantage. Since these objects are undefined, we also do not need to waste memory and time on them in the algorithms we use. We simply do not write any algorithms for them. Just ignore them.

To create a model of Homotopy Theory, we want to extend the notion of groupoids to n-groupoids. The natural way to do this is by allowing substitution in the following way in sub-expressions:

$$a \ <=> \ (a, a)$$

For example, for a tuple `(a, b)`, one can get `((a, a), b)`.

This can be thought of as extending and contracting functions `f : bool → nat, f' : $bool^2$ → nat`.

$$f \quad <=> \quad f'$$

From the perspective of theorem proving, this is like adding or removing a parameter.

By considering all possible inputs, one can add a normalization procedure for meta-theory, invariant over permutations, which collects all leaves in sub-expressions and constructs an ordered set.

For example, `((a, b), (c, a))` gets normalized to `(a, (b, c))`.
The normalization algorithm corresponds to treating functions as equal by their codomains.
We use this algorithm in meta-theory, while using it for gluing operations is optional and arbitrary.

Remember that the objects in `**S**` are tuples, so although one can normalize to `a`, this does not mean that the n-groupoid contain all possible objects `a, b, c, d, ...`. We need to add every object explicitly to the n-groupoid. The semantics of `a` will always be a tuple `(a, a)`.

The structure of such n-groupoids with natural numbers as objects is partial equivalence,
because there is symmetry and transitivity, but no reflexivity.

Now, we have a model of Homotopy Theory with proof semantics of n-groupoids.

We can also move on to do Path Semantics in this model. Path Semantics goes deeper than Homotopy Theory, because instead of just a single set `**S**`, which for n-groupoids is with respect to natural numbers, we construct multiple n-groupoids `$\mathbf{G_0}$, $\mathbf{G_1}$, $\mathbf{G_2}$, …, $\mathbf{G_n}$`. Each of these n-groupoids is like an `**S**` before, but now we use `**S**` to mean a set of such n-groupoids instead.

Consider a set `**S**` of such n-groupoids which do not share any object among themselves:

$$f \in \mathbf{G_n} \;\&\; g \in \mathbf{G_m} \;\&\; (f == g) \quad \rightarrow \quad \mathbf{G_n} == \mathbf{G_m} \quad \text{for all } \mathbf{G_n}, \mathbf{G_m} \in \mathbf{S}$$

For every `f : bool^(n : nat & (> 0)) → nat`, we can figure out which n-groupoid in `**S**` it belongs to. Often, it is sufficent to know that `f` belongs to some n-groupoid, using path semantical qubit `~`:

$$f \in \mathbf{G_n} \quad <=> \quad \sim f \quad \text{for all } \mathbf{G_n} \in \mathbf{S}, \text{ where } <=> \text{ is a bit “magical”}$$

Translating the property of `**S**` to using path semantical qubit `~`:

$$\sim f \;\&\; \sim g \;\&\; (f == g) \quad \rightarrow \quad \mathbf{G_n} == \mathbf{G_m} \quad \text{for all } \mathbf{G_n}, \mathbf{G_m} \in \mathbf{S}$$

Or, using the definition of path semantical quality `~~` and adding recursion on higher sets:

$$f \sim\sim g \quad \rightarrow \quad \mathbf{G_n} \sim\sim \mathbf{G_m} \quad \text{for all } f \in \mathbf{G_n}, g \in \mathbf{G_m}, \mathbf{G_n}, \mathbf{G_m} \in \mathbf{S}$$

Path semantical quality `~~` preserves the partial equivalence structure in these n-groupoids.

Now, there is a particular interesting class of such sets `**S**`, which has the following property:

$$a \mathrel{!}= b \quad \rightarrow \quad \sim(a, b) \quad \text{for all } a, b, \text{ where } (a, b) \in \mathbf{G_n}, \mathbf{G_n} \in \mathbf{S}$$

This means, whenever `a` and `b` are distinct, the tuple `(a, b)` belongs to some n-groupoid in `**S**`. By symmetry and transitivity, if `(a, b)` belongs to some n-groupoid in `**S**`, then `(a, a)` and `(b, b)` belongs to the same n-groupoid in `**S**`. Notice that inequality `_ != _` does not need to be fully defined. It might be a temporal phenomena where e.g. a person distinguishes objects over time, e.g. influenced by culture in an arbitrary way. We will get to the issue of totally defined inequality later.

The path semantical structure of `**S**` captures the notion that each n-groupoid consists of a number of pieces. Each piece only belongs to one particular n-groupoid and none of the others. When one consider a topology on such pieces, ignoring the numeric values, the n-groupoids become the classifying spaces of topology.

For example, if `!~(a, a)`, then it can not belong to any n-groupoid,
plus that no tuples can contain `a` in any of the n-groupoids, anywhere in `**S**`.
With other words `!~(a, a)` means a hole.

A hole does not belong to any particular n-groupoid in `**S**`. Furthermore, we are not stating our notion of continuity explicitly here. Continuity in topology is often using geometry and continuous functions. However, continuity might also describe any kind of preservation of dimensional structure along some particular transformation. For example, matrix transforms with non-zero determinants preserve geometric volume, which might be thought of as a notion of continuity. The definition of topology might vary between pieces, even within the same n-groupoid. We only know that whatever topology we use, holes tell us the global constraints on these topologies.

For example, when some topology on a piece is a dougnut (volume) or torus (surface), a dougnut as a hole in the middle and a torus has two holes: One in the middle and one in the interior. This means, you can not assign a dougnut to the same set `**S**` as to a torus where the torus is the surface of the doughnut. Why not? Because when the torus is the surface of the dougnut, they share the hole in the middle, but the dougnut connects to the hole in the interior, which is not allowed.

Similarly, you can not use some topology where a ball (volume) and sphere (surface) in the same set `**S**`, where the sphere is the surface of the ball. However, you are allowed to do it when the sphere is the surface of another ball. Therefore, when we have a ball and a sphere in the same set `**S**`, we know that they belong to two different topologies. They can not share the same inside.

This way of reasoning makes it possible to deduce properties of the set `**S**` with some added topology per piece in its n-groupoids, without knowing which choice of topology we have made.

In physics, a “hole” can be e.g. a wall that no object can penetrate through. We do not know, nor care, about the specific topology that these objects have, but we know for sure that they can not pass through that specific wall. If we had to define the topology for every object each time, then life would become very hard for Path Semanticists. However, Path Semantics usually do not worry about topology, but about the constraints and language biases of some mathematical language.

For example, you can use a container, e.g. a bucket and fill it with water. The bucket can be empty or filled with water, but it can not be both empty and filled at the same time. Here, the empty bucket functions like a “surface” on the filled bucket as “volume”. They can not both belong to the same set `**S**`. However, you might create a map of type `bool → **S**` to describe empty/filled buckets. This adds a time interpretation to your project, but this time interpretation does not need to e.g. reflect the time interpretation of path semantical levels. You can have multiple dimensions of time.

Remember that we only consider functions that maps to natural numbers up to some `n : nat`.

For `n = 2`, we have:

(0, 1) (1, 0)

Using symmetry and transitivity, one can show that there is only one n-groupoid that contains:

(0, 0) (0, 1) (1, 0) (1, 1)

In fact, this holds for any set `**S**` when we use totally defined inequality with the rule:

a != b → ~(a, b) for all `a, b`, where `(a, b) ∈ $\mathbf{G_n}$, $\mathbf{G_n}$ ∈ **S**`

When inequality is not total, we might have multiple n-groupoids in `**S**`.

Furthermore, since a total defined inequality operator spawns tuples for every pair, it means that everything in the single n-groupoid gets connected to everything else. It fills the space entirely.

Such spaces behave like propositions. Any member is as good as any other member, just like a proof of some proposition is just as good as any other proof of the same proposition. Still, propositional spaces can contain a hole. When there is a hole, there is just one n-groupoid and it contains zero objects. This propositional space corresponds to `false` in logic. All functions belong to the same n-groupoid, but we did not say there could not be zero functions involved. In a such case, the inequality is not well defined for any input and can be ignored.

This means that there is an analogue of propositions in any choice of topology over such sets `**S**`.

When interpreting this physically, one can think about it as no matter how we interpret physical reality, there will be some concept that corresponds to truth, just by thinking about physical reality.

Now, by thinking of `true` and `false` as propositional spaces, they might be thought of as the extreme maximum and minimum spaces, which both contain only a single n-groupoid. Any other partially defined inequality puts distinct numbers in some amount of n-groupoids, but we do not know how many there are. All we know is that the numbers are paired, so all their connections belong to the same n-groupoid.

It is kind of like a person who searches for a soulmate among the other living beings on the same planet. One person might have multiple soulmates, but there exists at least one soulmate per person.

In biology, for every species that reproduces sexually, there has to be at least one individual per biological gender for the species to survive. Otherwise, the species will go extinct or is already extinct. Since reproduction happens sexually, the species needs instincts programmed into the DNA of individuals to seek out reproduction this way. While not all species raise children as couples, there must at least be some method a species uses to reproduce using instincts or behavior.

So, the idea of using inequality to spawn connections into n-groupoids, is very useful when interpreting the physical world. A lot of this mathematics makes sense as physical structure.

There is also an element of the unknown when inequality is partially defined. When a person is born, it is not certain from the perspective of the person which planet they live on. The person has to learn and explore to figure it out. In a sense, it could be possible in principle that all living beings existed only on a single planet. However, there can be also multiple planets. The undefined behavior of inequality prevents us from assuming that we know these things, when we in fact do not know them. We always have to investigate the particular constraints on topology to learn more.

For example, there could be a constraint in biology such that carbon based life forms are incompatible with silicon life forms and vice versa. This could produce an instable equilibrium where a planet tends to support carbon based life forms, but not silicon life forms, or supports silicon life forms, but not carbon based life forms. These two life forms might be compatible for a short period of time, but perhaps getting more and more unstable in the long run. We do not know. When stated in general, this has nothing to do with AI versus humans in particular, but for biology.

In constructive logic, we can not prevent anyone from adding new axioms, unless the language the constructive logic is implemented in does not support introducing axioms. This means, we can only prove theorems in the context of an unknown amount of n-groupoids, using this model of Homotopy Theory.

Notice that this model is much simpler than constructive logic. The utility of using a such simple model is to learn what Homotopy Theory means in the context of theorem proving. It can help build intuition about what language biases there are in constructive logic.

If two pieces within the same n-groupoid are separated, then one might move one piece to a second n-groupoid. Or, one might move two pieces from two n-groupoids into a one n-groupoid. There can be an infinite number of empty n-groupoids. Kind of like a diary that has a large number of blank pages. The meaning of each n-groupoid is chosen arbitrarily. You can add your own meaning to the model. However, one piece can only be assigned to one n-groupoid. To work around this problem, you can use indices paired with pieces, e.g. `(0, a), (1, a), (2, a), …, (n, a)`.

While the limitation of keeping one n-groupoid per piece might seem overly restrictive at first, it is relatively easy to work around this limitation. Furthermore, when only looking at the pieces of the entire set `**S**`, by adding a time parameter, `time → **S**`, one might think of these pieces as moving around dynamically inside `**S**`. When there is no change in the definition of partial inequality, the pieces can move around but not be glued together or teared apart. This is a natural property of topological spaces. Therefore, `time → **S**` is a way to model topology easily. The rules that constrain how pieces move around between n-groupoids define the particular topology being used.

To describe an initial configuration with a topology `time → **S**`, one needs two pieces of information: The partial defined inequality operator, plus some map `init : nat$^2$ → nat` that tells which n-groupoid to put `(a, b)`, by calling `init(a, b)`. This returns the index of the n-groupoid to use. Now, this is a set `**S**`, so this technique does not tell you how to look up the n-groupoid. You can use a list of n-groupoids instead, which gives a natural way to initialize a state.

A rule can be thought of as a map: `rule : **S**$^2$ → bool`. It tells whether first state, first argument, can change into a second state, second argument. If you want to do probabilistic simulations using Markov chains, then you can return a probability instead of boolean: `rule : **S**$^2$ → prob`.

In many applications, e.g. video games, there is no reason to glue or tear apart objects. This means, the number of pieces can remain constant over time. Some game engines simply hide objects that are not visible on the screen and perform the same logic on both dead and alive entities, since the cost of CPU branching per entity can be more expensive than simulating a fixed array of entities. This technique also gives better predictability regarding performance, since the worst case of the algorithm gets easier to reason about.

Programmers rarely care about the topology of entities in their video games. Since the topology in this model is undefined, but also one can reason about holes, there is no need to add a specific topology but concentrate of what matters for the particular task at hand. This is usually reasoning about holes, instead of reasoning about how objects are connected.

In addition, when doing computer graphics, one rarely need to be concerned about volume vs surface at the same time. For example, it is common to do volumetric and surface ray tracing separately and combine the result afterwards into final rendering. The intuition that you can not put both the volume and the surface of the same geometric object into the same set `**S**`, helps to develop intuition why we tend to separate those two concerns in practical programming.

A geometric piece does not need to have specific coordinates, nor do we need to specify in full how we address the piece. This model cares very little about such implementation details.

Nor do you need to code this specific model to make programming easier. It helps to use the model mentally, as a language tool to focus on your project from a perspective of Homotopy Theory. Still, this is only to make reasoning easier. The purpose is to get stuff done, not over-think your design.

Homotopy Theory often focuses on higher morphisms, but in this model, this is just more complex sub-expressions. We usually do not care that much about what particular complex sub-expressions the model contains. Instead, we tend to write code that works for all cases. Many people feel the need to learn about these higher levels of abstractions and connections between things, but in practice, it is about reasoning about the number of pieces and how they relate to global constraints that is most useful.

Finally, all connections to a single point, e.g. `a` exists only within a single n-groupoid. Therefore, it can be useful to think of the point as belonging to that n-groupoid. Not just the piece, but its points.