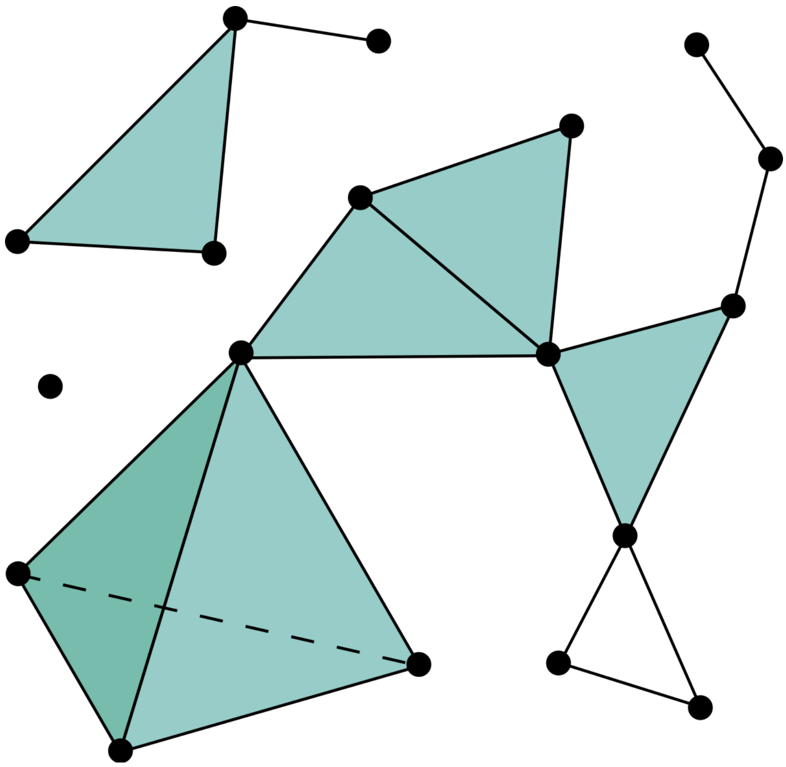
Simplicial sets in Lean

Floris Cnossen

June 20, 2021

Bachelor thesis Mathematics and Computer Science

Supervisor: dr. Benno van den Berg, dr. Jasmin Christian Blanchette, drs. Mees de Vries



Informatics Institute
Korteweg-de Vries Institute for Mathematics
Faculty of Sciences
University of Amsterdam

Abstract

The aim of this thesis is to formally verify a theorem from [6] which is a paper devoted to developing simplicial homotopy theory in a constructive way. This theorem is concerned with the geometric realization of a traversal, a certain construction in simplicial sets. The paper defines this geometric realization as a colimit and the theorem says that it can also be defined as a specific pullback. A consequence of this theorem is that two Moore structures on the category of simplicial sets from the papers "Un groupoïde simplicial comme modèle de l'espace des chemins" [2] and "Topological and simplicial models of identity types" [7] are equivalent.

In this thesis, we formalize a slightly weaker version of this theorem which says that the geometric realization is a weak pullback. This is done in the theorem prover Lean.

Title: Simplicial sets in Lean
Authors: Floris Cnossen, 12377309
Supervisors: dr. Benno van den Berg, dr. Jasmin Christian Blanchette, drs. Mees de Vries
Second graders: prof. dr. Yde Venema, dr. R.G. Belleman
End date: June 20, 2021

Informatics Institute
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.ivi.uva.nl`

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.kdvi.uva.nl`

Contents

1. Introduction

Simplicial sets are, like topological spaces, a way of describing mathematical shapes. However, unlike a topological space, a simplicial set is a combinatorial structure. It is made out of vertices, edges, triangles, etc. These are called simplices. The vertices are 0-simplices, the edges are 1-simplices, the triangles are 2-simplices, etc. Many topological concepts can be translated to simplicial concepts. In this thesis we will look at paths in simplicial sets.

In a topological space X, a path is a continuous map $p : [0, 1] \to X$. For two paths $p_1, p_2 : [0, 1] \to X$ such that $p_1(1) = p_2(0)$, we can compose these paths by first traversing p_1 and then p_2. Notice that this composition is associative and unital only up to homotopy. There is an analog of the interval in simplicial sets, called $\Delta[1]$. This is the simplicial set consisting of a single edge. For a simplicial set X we define a path as a simplicial morphism $p : \Delta[1] \to X$. However, composition of paths is only defined for a special type of simplicial set, called a Kan complex. This compositions is again only associative and unital up to homotopy. For a general simplicial set, we will look at a different notion of a path, called a Moore path.

In a topological space X, a Moore path is a continuous map $p : [0, l] \to X$ for some length parameter l. For two paths $p_1 : [0, l_1] \to X$ and $p_2 : [0, l_2] \to X$ such that $p_1(l_1) = p_2(0)$, we can compose these paths to get a path $[0, l_1 + l_2] \to X$. Notice that we do not have to rescale this path to the interval $[0, 1]$. This makes this composition strictly associative and unital. We can give the collection of Moore paths a topology, which gives us the topological space of Moore paths in X.

In a simplicial set X, a Moore path is a simplicial morphism $p : \widehat{\theta} \to X$ where θ is a parameter called a traversal and $\widehat{\theta}$ is its geometric realization. The geometric realization is a simplicial set introduced in the paper "Topological and simplicial models of identity types" [7]. Informally, the geometric realization $\widehat{\theta}$ is a sequence of n-simplices connected by $n+1$-simplices. The length of this sequence and how these simplices are connected is described by the traversal θ. In the simplest case, $n = 0$, the geometric realization is a sequence of vertices connected by edges, with a length determined by the traversal. This has similarities to the topological interval $[0, l]$, which has a length determined by the parameter l. We can define a composition of simplicial Moore paths which is associative and unital, similar to topological Moore paths. The paths in X form a simplicial set MX.

Theorem 9.11 from the paper "Effective Kan fibrations in simplicial sets" [6] says that geometric realization fits into a pullback square. This is a complex theorem with a lot of details. This theorem is important for showing that the simplicial set of Moore paths MX can be defined in a different but equivalent way, introduced in the paper "Un groupoïde simplicial comme modèle de l'espace des chemins" [2].

In this thesis we will formalize the first half of this theorem. Namely, that the geometric realization is a weak pullback. We will be using the theorem prover Lean. Lean is a functional computer language that ensures that a proof is correct by checking each step separately. Lean is based on a fundamental description of mathematics, called type theory. The main difference between type theory and the traditional set theory, is that the notion of a set is replaced by that of a type. A type in type theory is similar to a type in programming languages like C.

The Lean code can be found in the appendices and in the following GitHub repository: `https://github.com/floriscnossen/simplicial_sets_in_lean`.

1.1. Overview

In this thesis, a basic understanding of category theory is expected. In Chapter 2, we formally define simplicial sets and describe some of their basic properties. In Chapter 3, we define traversals and their geometric realization. We also formulate the main goal of the thesis. In Chapter 4, we give an introduction to type theory and Lean, based on the book "Theorem Proving in Lean" [1]. In Chapter 5, we define traversals as well as a recursive version of the geometric realization in Lean. Lastly, we formalize the theorem that this geometric realization is a weak pullback.

2. Simplicial sets

In this chapter we will introduce the notion of a *simplicial set*. We will also discuss some of the properties of simplicial sets. This introduction is based on the article "An elementary illustrated introduction to simplicial sets" [3]. Informally, a simplicial set is a mathematical structure made out of vertices, edges between these vertices, triangular faces between these edges, etc. An example of a simplicial set can be seen in Figure 2.1.

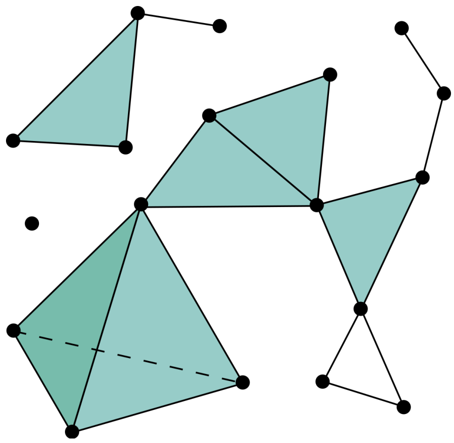


Figure 2.1.: Example of a simplicial set, Source: Wikipedia

One of the first questions that arises is how to define these objects formally. We could define a simplicial set as a topological space, but topological spaces are complicated objects. It turns out that all information we need from a simplicial set can already be encapsulated in a combinatorial definition.

For each dimension $n \in \mathbb{N}$ we have a set X_n of n-dimensional simplices. This means that X_0 is the set of vertices, X_1 the set of edges, X_2 the set of triangles, etc. In general an n-simplex is an n-dimensional pyramid with $n + 1$ vertices. Another way to write this sequence of sets is as a function $X : \mathbb{N} \to \mathbf{Set}$, where **Set** is the class of all sets. An example with labeled simplices is given in Figure 2.2.

Figure 2.2 does not show all simplices in X, because there are also implicit simplices called *degenerate* simplices. These simplices are collapsed onto lower dimesional simplices and turn out to be useful for multiple reasons. They have a similar purpose as identity maps in a category.

The sets X_n are related to each other. For example, each edge in X_1 has two vertices in X_0 as begin- and endpoints and in Figure 2.2 we can see that $\alpha \in X_2$ has 3 edges $x, y, z \in X_1$. These relations are described by maps between the sets X_n. The properties

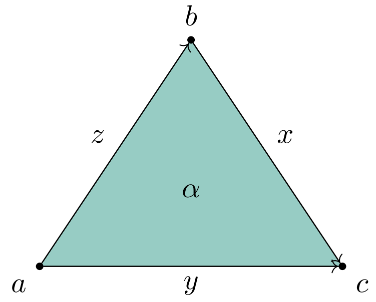


Figure 2.2.: A triangle with $X_0 = \{a, b, c\}$, $X_1 = \{x, y, z, \ldots\}$, $X_2 = \{\alpha, \ldots\}$, ...

of these maps are similar to those of the category of nonempty finite ordinals Δ. This category is called the *simplex category* and its set of objects is equivalent to $\mathbb{N}$. We will define this category and look at its properties in the next section. In Section 2.2 we will define a simplicial set as a contravariant functor from the simplex category to the category of sets.

2.1. Simplex category

Definition 2.1. *The simplex category Δ is the category with objects $[n] := \{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$ and order preserving maps as morphisms.*

This means that for a function $f : [n] \to [m]$ we have

$$f \in \mathrm{Hom}_\Delta([n], [m]) \iff \forall i \leqslant j,\ f(i) \leqslant f(j).$$

There are two types of fundamental maps in the simplex category called the standard face maps and standard degeneracies.

Definition 2.2. *For any $n \in \mathbb{N}$ and $i \in [n+1]$ we define the ith standard face map as a map $\delta_i : [n] \to [n+1]$ with*

$$\delta_i(j) = \begin{cases} j, & \text{if } j < i, \\ j+1 & \text{if } j \geqslant i. \end{cases}$$

A face map is a composition of standard face maps.

This means that a standard face map $\delta_i : [n] \to [n+1]$ is an injective map that leaves a gap at i. This is visualized in Figure 2.3.

Notice that the composition of two injective maps is also injective, so all face maps are injective. It turns out that the converse is also true.

Theorem 2.3. *For a morphism $f : [n] \to [m]$, the following are equivalent:*

1. *f is injective,*

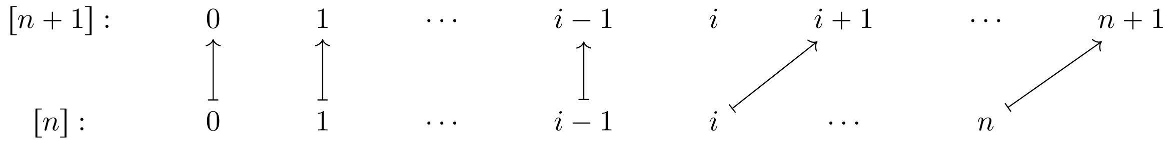


Figure 2.3.: The standard face map $\delta_i : [n] \to [n+1]$.

2. *f is a monomorphism,*
3. *f is a face map.*

Definition 2.4. *For any $n \in \mathbb{N}$ and $i \in [n]$ we define the ith standard degeneracy as a map $\sigma_i : [n+1] \to [n]$ with*

$$\sigma_i(j) = \begin{cases} j, & \text{if } j \leqslant i, \\ j-1 & \text{if } j > i. \end{cases}$$

A degeneracy is a composition of standard degeneracies.

This means that a degeneracy $\sigma_i : [n+1] \to [n]$ is a surjective map that hits i twice. This is visualized in Figure 2.4.

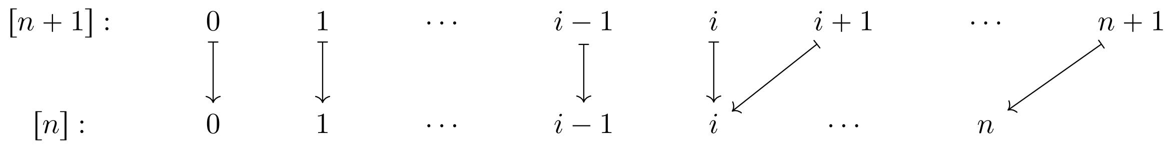


Figure 2.4.: The standard degeneracy $\sigma_i : [n+1] \to [n]$.

Similarly to face maps we have the following theorem:

Theorem 2.5. *For a morphism $f : [n] \to [m]$, the following are equivalent:*

1. *f is surjective,*
2. *f is a epimorphism,*
3. *f is a degeneracy.*

Face maps and degeneracies have special properties called the simplicial identities.

Theorem 2.6. *For any $n \in \mathbb{N}$ we have the following identities:*

$$\begin{aligned}
\delta_{j+1} \circ \delta_i &= \delta_i \circ \delta_j && \text{for } i, j \in [n+1] \text{ with } i \leqslant j, \\
\sigma_{j+1} \circ \delta_i &= \delta_i \circ \sigma_j && \text{for } i \in [n+1], j \in [n] \text{ with } i \leqslant j, \\
\sigma_i \circ \delta_i &= \sigma_i \circ \delta_{i+1} = \mathrm{id} && \text{for } i \in [n], \\
\sigma_j \circ \delta_{i+1} &= \delta_i \circ \sigma_j && \text{for } i \in [n+1], j \in [n] \text{ with } i > j, \\
\sigma_j \circ \sigma_i &= \sigma_i \circ \sigma_{j+1} && \text{for } i, j \in [n] \text{ with } i \leqslant j.
\end{aligned}$$

The face maps and degeneracies generate the simplex category. In particular, we get the following theorem:

Theorem 2.7. *Let $f : [n] \to [m]$ be a morphism in Δ. There exist a unique degeneracy $p : [n] \to [k]$ and a unique face map $i : [k] \to [m]$ such that $f = i \circ p$.*

In particular, by theorems 2.3 and 2.5, the simplex category has a unique factorization of morphisms into an epi- and a monomorphism.

2.2. Definition of simplicial sets

Definition 2.8. *A simplicial set is a functor $X : \Delta^{op} \to \textbf{\textit{Set}}$.*

For each $n \in \mathbb{N}$ the n-simplices of X are the elements of $X_n := X[n]$. These are the sets from the introduction of this chapter. A simplicial set X is a contravariant functor so any morphism $f : [n] \to [m]$ in Δ gets sent to a map $X(f) : X_m \to X_n$. In particular, the standard face maps $\delta_i : [n] \to [n+1]$ get sent to maps $X(\delta_i) : X_{n+1} \to X_n$. Each $n+1$-simplex $x \in X_{n+1}$, has $n+1$ faces. The ith face of x is defined as $X(\delta_i)(x)$. In the triangle of Figure 2.2, we get for the edges x, y and z that

$$\begin{aligned} X(\delta_0)(x) &= c, & X(\delta_1)(x) &= b, \\ X(\delta_0)(y) &= c, & X(\delta_1)(y) &= a, \\ X(\delta_0)(z) &= b, & X(\delta_1)(z) &= a, \end{aligned}$$

For the triangle α we get $X(\delta_0)(\alpha) = x, X(\delta_1)(\alpha) = y, X(\delta_1)(\alpha) = z$. Intuitively, the ith face of a simplex x is what remains after removing its ith vertex.

The standard degeneracies $\sigma_i : [n+1] \to [n]$ get sent to maps $X(\sigma_i) : X_n \to X_{n+1}$. The simplices in the images of these maps are the degenerate simplices. Intuitively, applying $X(\sigma_i)$ to a simplex x duplicates the ith vertex of x in its place. The degenerate simplices in Figure 2.2 are the simplices without a name. The simplices $X(\sigma_0)(a)$, $X(\sigma_0)(b)$ and $X(\sigma_0)(c)$, can be seen as the "constant" edge at vertices a, b and c respectively.

The Yoneda embedding of the simplex category gives a collection of simplicial sets called the *standard simplices.*

Definition 2.9. *The n-dimensional standard simplex is the simplicial set $\Delta[n] := \mathrm{Hom}(-, [n]) : \Delta^{op} \to \textbf{\textit{Set}}$. In particular, its set of m-simplices is* $\mathrm{Hom}([m], [n])$.

Intuitively, $\Delta[n]$ is a single n-dimensional simplex. For example, $\Delta[0]$ is a vertex, $\Delta[1]$ is an edge and $\Delta[2]$ is a triangle. In fact, the triangle in Figure 2.2 is equal to $\Delta[2]$ with relabeled simplices. The 2-simplex α corresponds to $\mathrm{id} \in \mathrm{Hom}([2], [2])$.

A morphism between simplicial sets X and Y is a natural transformation $\alpha : X \to Y$. In other words, it is a map $\alpha_n : X_n \to Y_n$ for each $n \in \mathbb{N}$ such that for each $f : [n] \to [m]$ we have $X(f) \circ \alpha_m = \alpha_n \circ Y(f)$. By the Yoneda lemma, morphisms $\Delta[n] \to X$ are in bijection with the set X_n.

More information and an intuitive explanation of the choice of the simplex category can be found in [3]

3. Traversals

This chapter will introduce the main topic of this thesis: *Traversals.* We will be using the definitions given in [6]. Traversals are used to define a notion of paths in simplicial sets. In a topological space X, a path is a continuous map $p : [0, 1] \to X$. The analog of the unit interval in simplicial sets is $\Delta[1]$, so for a simplicial set X we can look at morphisms $\Delta[1] \to X$. By the Yoneda lemma, this is equivalent to X_1. This is a construction commonly used for a special type of simplicial set, called a Kan complex. Composition of these paths is associative and unital only up to homotopy. For our purposes, we need a notion of path that has a strict associative and unital composition in every simplicial set.

A Moore path in a topological space X is a continuous map $p : [0, l] \to X$ for some length parameter $l \in \mathbb{R}_+$. Composition of Moore paths is done by putting one after the other. Notice that we do not rescale the length of the path, which makes this composition associative. We will define a similar construction for simplicial sets. However now each path has a more complicated parameter. This parameter is called a traversal.

Definition 3.1. *For any natural number $n \in \mathbb{N}$, an n-traversal is a list of elements in $[n] \times \{+, -\}$, called n-edges.*

We call an edge *positive* or *negative* if its second component is $+$ or $-$ respectively. A traversal can be visualized as a chain of edges. Positive edges point to the right and negative edges point to the left. An example is shown in Figure 3.1.

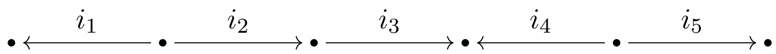


Figure 3.1.: An n-traversal $[(i_1, -), (i_2, +), (i_3, +), (i_4, -), (i_5, +)]$.

For any map $\alpha : [n] \to [m]$ in the simplex category and any positive m-edge $(p, +)$. We define $(p, +) \cdot \alpha$ as the n-traversal

$$(i, +) \cdot \alpha := [(j, +) \mid j \in [n, \ldots, 0], \alpha(j) = i]. \tag{3.1}$$

Here we take the j's in *decreasing* order. In other words $(i, +) \cdot \alpha$ is equal to $\alpha^{-1}(\{i\}) \times \{+\}$ in decreasing order. Similarly, for a negative edge $(i, -)$. We can define $(i, -) \cdot \alpha$ as an n-traversal

$$(i, -) \cdot \alpha := [(j, -) \mid j \in [0, \ldots, n], \alpha(j) = i]. \tag{3.2}$$

Now we take the j's in *increasing* order. This means that $(i, +) \cdot \alpha$ is equal to $\alpha^{-1}(\{i\}) \times \{-\}$ in increasing order. For any m-traversal θ, we define $\theta \cdot \alpha$ by applying α to each edge and concatenating the results in order.

Theorem 3.2. *For any $\alpha : [n] \to [m]$ and $\beta : [m] \to [l]$ and an l-traversal θ, we have*

$$\theta \cdot (\beta \circ \alpha) = (\theta \cdot \beta) \cdot \alpha.$$

It is also clear that $\theta \cdot \mathrm{id} = \theta$. Therefore we can define a simplicial set of traversals.

Definition 3.3. *The simplicial set $\mathbb{T}_0$ has as n-simplices all n-traversals. A map $\alpha : [n] \to [m]$ acts on m-traversals by sending θ to $\theta \cdot \alpha$.*

Let θ be an n-traversal with length l. A *position* in θ is a value $p \in [l]$. This value corresponds to one of the black dots in Figure 3.1. A pointed n-traversal is a pair (θ, p) where θ is an n-traversal and p is a position in θ. A pointed traversal can be visualized by marking one of the points in Figure 3.1. Notice that we can split a pointed traversal at its point. This gives a pair of traversals. Conversely, we can connect two traversal and remember the point of contact. This gives a bijection between pointed traversals and pairs of traversals, as can be seen in Figure 3.2.

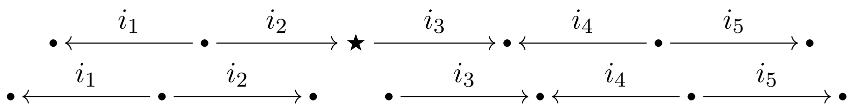


Figure 3.2.: A pointed n-traversal $([(i_1, -), (i_2, +), (i_3, +), (i_4, -), (i_5, +)], 2)$ and its corresponding pair of n-traversals $([(i_1, -), (i_2, +)], [(i_3, +), (i_4, -), (i_5, +)])$.

It turns out that it is easier to work with pairs of traversals, so this will be our final definition of pointed traversals.

Definition 3.4. *A pointed n-traversal is a tuple (θ_1, θ_2) where θ_1 and θ_2 are n-traversals.*

However, we will sometimes use the alternative definition given by the bijection above. Again, we can define a simplicial set of pointed traversals.

Definition 3.5. *The simplicial set $\mathbb{T}_1$ has as n-simplices all pointed n-traversals. The maps act component-wise on the pointed n-traversals.*

There are two important maps from $\mathbb{T}_1$ to $\mathbb{T}_0$ called *dom* and *cod*.

Definition 3.6. *The map* $\mathrm{dom} : \mathbb{T}_1 \to \mathbb{T}_0$ *is defined by* $\mathrm{dom}(\theta_1, \theta_2) = \theta_2$ *and the map* $\mathrm{cod} : \mathbb{T}_1 \to \mathbb{T}_0$ *is defined by* $\mathrm{cod}(\theta_1, \theta_2) = \theta_1 + \theta_2$*, where $+$ is concatenation of lists.*

We call these maps dom and cod because $\mathbb{T}_1$ defines a partial order on $\mathbb{T}_0$. For two traversals θ_1 and θ_2 we say that $\theta_1 \leqslant \theta_2$ if θ_1 is a tail of θ_2. In other words, if there is some traversal θ_1' such that $\theta_2 = \theta_1' + \theta_1$. This is equivalent to defining $\theta_2 = \mathrm{dom}(\theta_1, \theta_2) \leqslant \mathrm{cod}(\theta_1, \theta_2) = \theta_1 + \theta_2$ for any pointed traversal (θ_1, θ_2). This order defines a category on $\mathbb{T}_0$ such that

$$\mathrm{Hom}(\theta_2, \theta_1 + \theta_2) = \{(\theta_1, \theta_2)\}.$$

In this way, a pointed traversal (θ_1, θ_2) can be seen as a morphism from $\theta_2 = \mathrm{dom}(\theta_1, \theta_2)$ to $\theta_1 + \theta_2 = \mathrm{cod}(\theta_1, \theta_2)$.

3.1. Geometric realization

Recall that in a topological space, a Moore path is a continuous map from the interval $[0,l]$ for some parameter l. The topological space $[0,l]$ acts like a template for a Moore path in topological spaces. Similarly, for a traversal θ, we can define its geometric realization $\widehat{\theta}$. This is a simplicial set that will act as a template for a Moore path in simplicial sets.

Intuitively, for each position p in θ we take a copy $\Delta[n]_p$ of $\Delta[n]$ and for each kth edge in θ we take a copy $\Delta[n+1]_k$ of $\Delta[n+1]$. The kth edge (i,b) in θ, lies between the positions k and $k+1$. $\Delta[n]_k$ and $\Delta[n]_{k+1}$ get identified with faces of $\Delta[n+1]_k$. This is done in such a way that all vertices of $\Delta[n]_k$ and $\Delta[n]_{k+1}$ get identified pairwise except for their ith vertices. These vertices will be connected by an edge. If $b=+$ this edge will go from left$(\Delta[n]_k)$ to right$(\Delta[n]_{k+1})$ and if $b=-$ this edge will go from right to left. This is determined by the right choice of faces in $\Delta[n+1]_k$.

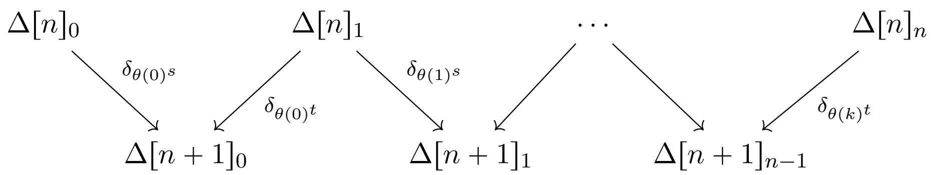
Formally, the choice of these faces is determined by the following maps.

Definition 3.7. *For some n-edge (i,b) we define $(i,b)^s, (i,b)^t \in [n+1]$ by*

$$(i,+)^s = k+1, \qquad (i,-)^s = k,$$
$$(i,+)^t = k, \qquad (i,-)^t = k+1.$$

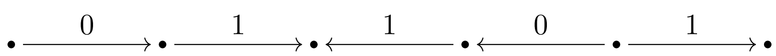
We define the geometric realization formally as follows:

Definition 3.8. *The geometric realization $\widehat{\theta}$ of a traversal θ is the colimit over the diagram*

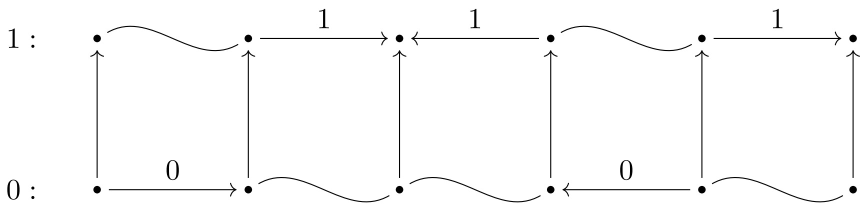


Here the maps $\delta_i : \Delta[n] \to \Delta[n+1]$ are the images of $\delta_i : [n] \to [n+1]$ under the Yoneda embedding. Notice that the subscripts after $\Delta[n]$ and $\Delta[n+1]$ do not matter for the colimit and are only useful when talking about the different copies of $\Delta[n]$ and $\Delta[n+1]$ in the geometric realization.

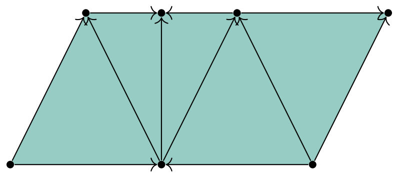
Definition 3.8 is not very intuitive, so we will look at some examples by using the intuition at the start of this section. For a 0-traversal, the geometric realization is equal to the visualization of Figure 3.1, because $\Delta[0]$ is just a single point. For the 1-traversal $[(0,+),(1,+),(1,-),(0,-),(1,+)]$ visualized by



we create 6 copies of $\Delta[1]$, which are just arrows. For each edge (i, b) in the traversal, we draw an arrow between the points with values i in direction b. The other points get marked with $\sim$ as can be seen in the following image:



After identifying the points marked with $\sim$ and filling in the triangles, we get



This is the geometric realization of the 1-traversal $[(0, +), (1, +), (1, -), (0, -), (1, +)]$.

The geometric realization is defined as a colimit. However, the geometric realization can also be described as a pullback.

Theorem 3.9. *The geometric realization $\widehat{\theta}$ of an n-traversal θ fits into the pullback square*

$$\begin{array}{ccc} \widehat{\theta} & \xrightarrow{k_\theta} & \mathbb{T}_1 \\ \big\downarrow{\scriptstyle j_\theta} & & \big\downarrow{\scriptstyle \mathrm{cod}} \\ \Delta[n] & \xrightarrow{\theta} & \mathbb{T}_0 \end{array}$$

where $\theta : \Delta[n] \to \mathbb{T}_0$ is the map obtained from the Yoneda lemma.

A proof of this theorem is given in [6]. A lot of the theory in that paper depends on the correctness of this theorem. Therefore, this theorem will be the main topic of this thesis. The fist half of this theorem says that this pullback is a weak pullback.

Theorem 3.10. *The geometric realization $\widehat{\theta}$ of an n-traversal θ is a weak pullback in the square of Theorem 3.9. This means that every pullback cone over the diagram of Theorem 3.9 has a lift to $\widehat{\theta}$.*

The difference between Theorem 3.9 and Theorem 3.10 is that Theorem 3.9 requires this lift to be unique.

We will formalize Theorem 3.10 using Lean. We will discuss this in the next chapter.

3.2. Moore paths in simplicial sets

In a topological space X, we can look at the set of all Moore paths in X. This set is equal to $\bigcup_{l \in \mathbb{R}_+} \mathrm{Hom}([0,l], X)$ and can be given a topology.

We will define a similar construction for simplicial sets. Given a simplicial set X we want to define a simplicial set of Moore paths MX. An n-simplex of MX is a morphism $p : \widehat{\theta} \to X$ for some n-traversal θ.

Definition 3.11. *Let X be a simplicial set and $n \in \mathbb{N}$. We define*

$$(MX)_n := \bigcup_{\theta \in \mathbb{T}_0(n)} \mathrm{Hom}(\widehat{\theta}, X).$$

For a map $\alpha : [m] \to [n]$ there is a map $MX(\alpha) : (MX)_n \to (MX)_m$. These maps give MX the structure of a simplicial set, but we will not discuss these maps in this thesis.

Notice that $p \in (MX)_0$ is a map from $\widehat{\theta} \to X$ for some 0-traversal θ. The geometric realization of a 0-traversal is a chain of edges pointing either to the left or right. This means that the image of p is also a chain of connected edges. In fact, if we define GX as the undirected multigraph with vertices X_0 and undirected edges X_1, then $(MX)_0$ corresponds directly to paths in the graph GX. This means that intuitively, an element of $(MX)_0$ is a path in X that only walks over the edges in X.

Take two n-dimentional Moore paths $p_1 : \widehat{\theta_1} \to X$ and $p_2 : \widehat{\theta_2} \to X$ for some n-traversals θ_1 and θ_2. Suppose that the image of the last copy of $\Delta[n]$ in $\widehat{\theta_1}$ under p_1 is the same as the image of the first copy of $\Delta[n]$ in $\widehat{\theta_2}$. In this case we can compose the paths p_1 and p_2. This results in a path $p_1 + p_2 : \widehat{\theta_1 + \theta_2} \to X$. Intuitively, this path is equal to p_1 on $\widehat{\theta_1} \subseteq \widehat{\theta_1 + \theta_2}$ and equal to p_2 on $\widehat{\theta_2} \subseteq \widehat{\theta_1 + \theta_2}$. This is well-defined, because we assumed p_1 and p_2 are the same on the intersection $\widehat{\theta_1} \cap \widehat{\theta_2} \subseteq \widehat{\theta_1 + \theta_2}$. This composition of Moore paths is associative and defines a notion of fundamental groups of simplicial sets.

4. Type Theory and Lean

Most of mathematics taught to students is based on set theory. This is a fundamental description of mathematics in which every mathematical object is a set. Using an additional layer of predicate logic we can prove theorems about sets. For any two sets A and B we can talk about whether A is an element of B or not. This is expressed as a proposition $A \in B$. For example, $-1 \in \mathbb{N}$ and $1 \in \mathbb{N}$ are two propositions that are false and true respectively. However, in set theory it is also possible to write unnatural propositions like $2 \in 1$ and $0 \in 1$. Again the first proposition is false and the second proposition is true in set theory. These propositions feel unnatural because we usually do not think of the number 1 as a set. In this chapter we will discuss a different formal system, called type theory, that does not introduce these unnatural expressions.

In type theory, we replace the notion of a set by that of a type. A type `T` can "have an object `x`" and we write `x : T`. For example, `0 : ℕ` and `1 : ℕ`. Any object has a unique type. Different from set theory, the expression `x : T` is not a proposition. This means that we do not ask questions like: "Does `x` have type `T`?" The type of an object is always known from the moment that we introduce the object. This is similar to programming languages like C. When we introduce a variable `int x = 1`, we always specify its type at the moment that we declare the variable. As a result, we cannot write `0 : 1` in type theory, because `0` and `1` both have type ℕ.

In set theory, an object can be an element of multiple sets. However in type theory, each object has a unique type. This helps us manage mathematical objects based on their type. For example, we can define a function that takes inputs of a certain type and gives outputs of another type. Again, this is similar to the language C, where each argument and return value of a function has a type.

There are multiple versions of type theory and we will be using the one from Lean. Lean is a theorem prover and programming language based on type theory. We give an introduction to Lean and type theory, based on the book "Theorem proving in Lean" [1]. In Lean's version of type theory, types themselves are objects with type `Type` and therefore can be studied as well.

We can define objects in Lean in the following way:

```
def object_name : type_name := defin
```

Here `def` is a keyword at the start of the definition. This code creates an object called `object_name` of type `type_name` defined by `defin`. For example, we could define the number `five` as the sum of 2 and 3.

```
def five : ℕ := 2 + 3
```

To formulate and prove theorems in type theory, we do not need an extra layer of predicate logic. Instead, we can do this using type theory itself. Theorems and propositions are of the type `Prop`. Every proposition is also a type itself and an object is a proof of the proposition. In Lean, we are only interested in whether a proposition is provable or not. Therefore we consider all proofs of a proposition to be equal. In this way a proposition has no objects if it is false and one object if it is true.

Formulating and proving theorems in Lean is done in a similar way as giving definitions. The difference is that we replace the keyword `def` by `theorem` or `lemma`. For example, we can prove that `five` defined above as `2 + 3` is equal to `3 + 2` as follows:

```
lemma five_eq_3_plus_2 : five = 3 + 2 := nat.add_comm 2 3
```

Here `nat.add_comm` is a proof that addition of natural numbers is commutative.

We will now look at some fundamental constructions for types.

4.1. Function types

Functions are an important concept in all parts of mathematics. For many mathematical structures, we can look at functions between these structures. To define a function in set theory, we first have to define tuples. Next we can define a function as a set of tuples that satisfy some property. This definition is quite complicated for such a fundamental concept. In practice, a function is some mathematical structure that takes some input and returns some output. In type theory, functions are defined by this property.

For two types α and β we can construct a function type $\alpha \to \beta$. An object of this type will be a function that sends objects of type α to objects of types β. For `a` $: \alpha$ and `f` $: \alpha \to \beta$, the evaluation of `f` in `a` will be `f a` $: \beta$. We can construct functions using *lambda abstraction*. Suppose that given some `a` $: \alpha$ we can construct some $\mathtt{b}_\mathrm{a} : \beta$ then we can define a function with λ `(a :` α`),` $\mathtt{b}_\mathrm{a}$. This function sends an object `a` to $\mathtt{b}_\mathrm{a}$. As an example we can define the function `f` that adds 3 to a natural number.

```
def f : ℕ → ℕ := λ (n : ℕ), n + 3
```

Alternatively, we can declare the value `n` right after the declaration of `f`.

```
def f (n : ℕ) : ℕ := n + 3
```

The idea behind this notation is that we define `f n` $: \mathbb{N}$ given some `n` $: \mathbb{N}$ instead of defining `f` directly. Note that the two functions above are equal. Their definitions are two different ways to describe the same function.

In the case that α and β are propositions, $\alpha \to \beta$ is again a proposition. Any proof `p` $: \alpha \to \beta$ sends proofs of α to proofs of β, so if α is provable then β is provable as well. This means we can think of $\alpha \to \beta$ as "α implies β".

We will now look at functions with multiple arguments. For three types α, β and γ we want to define a function that takes an object from α and an object from β and returns an object of γ. An intuitive way of doing this is using the type $(\alpha \times \beta) \to \gamma$, where $\alpha \times \beta$ is the product type that we will define in section 4.3. There is however an easier way that might be less intuitive. We can define functions with multiple arguments using the type $\alpha \to (\beta \to \gamma)$. Objects of this type are called *curried functions*. For some `f` : $\alpha \to (\beta \to \gamma)$, `a` : α and `b` : β, we can apply `f` to `a` and get `f a` : $\beta \to \gamma$. We can apply this new function to `b` and get `f a b` : γ. This is exactly what we were looking for, because now `f` takes two objects of types α and β respectively and returns an object of type γ. The type $\alpha \to (\beta \to \gamma)$ is used so often that we can remove the parentheses and just write $\alpha \to \beta \to \gamma$. An example of a function with two arguments is addition `add` : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ on the natural numbers.

4.2. Pi types

Given a type α and some $\beta : \alpha \to$ `Type` we can construct a type $\prod$ `(a :` α`),` β `a`. This is called a *Pi type* or *product type*. An object `t` of this type is a tuple of objects `t a` : β `a` for each `a` : α. Again we can use lambda abstraction to construct tuples. Suppose that given some `a` : α we can construct some $\mathtt{b}_{\mathrm{a}}$: β `a` then we can define a tuple with λ `(a :` α`),` $\mathtt{b}_{\mathrm{a}}$. This is very similar to a function type, except the type of the output depends on the input. This is why a Pi type is also called a *dependent function type*.

Given a type α and some $\beta : \alpha \to$ `Prop`, we write $\forall$ `(a :` α`),` β `a` as an abbreviation for $\prod$ `(a :` α`),` β `a`. This is a proposition that says that β `a` is true for all `a` : α. As an example we can look at the proposition that every natural number is equal to itself.

```
lemma eq_self : ∀ (n : ℕ), n = n := λ (n : ℕ), rfl
```

Here `rfl` is a proof that `n = n`.

4.3. Inductive types

So far we have seen how to construct types from other types. However we need types to begin with. This is where inductive types come in. An inductive type has a name and a finite number of constructors. In Lean this will look as follows:

```
inductive type_name : Type
| constructor₁ : ... → type_name
| constructor₂ : ... → type_name
...
| constructorₙ : ... → type_name
```

As the name implies, each constructor constructs different objects of type `type_name`. Conversely, each object of type `type_name` is constructed from one of the constructors. The dots in the constructor can be any number of arguments. If none of the constructors have any arguments, then we call it an *enumerated type*. An enumerated type has one object for each constructor. Some examples of enumerated types are

```
inductive empty : Type

inductive unit : Type
| star : unit

inductive bool : Type
| ff : bool
| tt : bool
```

Here `empty`, `unit` and `bool` have 0, 1 and 2 objects respectively. We can define functions on enumerated types by defining it for each constructor. For example,

```
def not : bool → bool
| ff := tt
| tt := ff
```

is the function that negates boolean values.

For more complex inductive types, we can use arguments in the constructors. Our first example will be that of the binary product of two types α and β. This is defined as follows:

```
inductive prod (α : Type) (β : Type)
| mk : α → β → prod
```

Each object of `prod α β` is of the form `mk a b` for `a : α` and `b : β`. In Lean we can also write this as `(a, b) : α × β`. The type `α × β` can be seen as the cartesian product of α and β. Inductive types with only one constructor, like `prod`, are called *structures*. In most structures the only constructor is called `mk`. All objects `x` of a structure have the form `mk a b ...`. Therefore Lean also has a different notation for defining structures. For `prod` this looks as follows

```
structure prod (α : Type) (β : Type) :=
(fst : α)
(snd : β)
```

This automatically creates the constructor `mk` from before. This new notation also introduces two functions `fst : prod → α` and `snd : prod → β`. For some object `x = mk a b : prod α β` we can retrieve `a` and `b` directly by writing `x.fst` and `x.snd` respectively. This is very similar to a struct in the programming language C.

In the next example we will look at the binary sum of two types α and β. This is defined as

```
inductive sum (α : Type) (β : Type)
| inl : α → sum
| inr : β → sum
```

Each object of `sum` α β is of the form `inl a` for `a` : α or `inr b` for `b` : β. Notice that if `a` : α = β then `inl a` is not equal to `inr a` because they come from different constructors. For this reason `sum` α β can be seen as the disjoint union of α and β.

So far the constructors in inductive types only contain arguments from other types. However the power of inductive types comes really from the fact that the arguments can have the type that you are defining. An example of this is the type `list`. For any type α we can define the type of lists as

```
inductive list (α : Type)
| nil  : list
| cons : α → list → list
```

Objects of type `list` α are of the form `nil` or `cons hd tl`, where `hd` : α and `tl` : `list` α. The first constructor describes the empty list `[]` : `list` α. The second constructor creates a list by taking another list and a new element. The value `hd` is considered the head of the list and `tl` is the remaining tail. Recursively `tl` was constructed again by one of the constructors. The list `[a, b, c]` is defined is Lean as `cons a (cons b (cons c nil))` or `a :: b :: c :: nil` in short. Notice that `list` α only contains finite lists, because `cons hd tl` requires `tl` to have already been constructed. This means the chain has to start somewhere with the constructor `nil`. We can define functions on the type `list` α by defining it for `nil` and `cons hd tl` separately. However, now we can use recursion by applying the same function on `tl`. For example, we will define the length of a list by

```
def length {α : Type} : list α → ℕ
| nil      := 0
| hd :: tl := length tl + 1
```

If we unfold this definition on the list `[a, b, c]` we get

$$\begin{aligned}
\texttt{length (a :: b :: c :: nil)} &= \texttt{length (b :: c :: nil) + 1} \\
&= \texttt{length (c :: nil) + 1 + 1} \\
&= \texttt{length nil + 1 + 1 + 1} \\
&= \texttt{0 + 1 + 1 + 1} = 3.
\end{aligned}$$

This is what we expect.

4.4. Natural numbers

In some of the previous examples we already used the type `nat` or ℕ in short. This type is defined as

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

Objects of this type are `zero` and `succ n` for some `n : nat`. This means we get infinitely many objects `zero`, `succ zero`, `succ (succ zero)`, ... These correspond to the natural numbers 0, 1, 2, ... We define functions and relations on ℕ inductively. For example, addition is defined by induction on the second argument

```
def add : ℕ → ℕ → ℕ
| n zero     := n
| n (succ m) := succ (add n m)
```

This is based on the fact that $n + 0 = n$ and $n + (m + 1) = (n + m) + 1$.

4.5. Categories

Given a type `obj : Type`, we can define a category with objects `obj` by giving an object of type `category obj`. Here `category obj` is defined as follows:

```
class category (obj : Type) :=
(hom       : obj → obj → Type)
(id        : Π (X : obj), hom X X)
(comp      : Π {X Y Z : obj}, (hom X Y) → (hom Y Z) → (hom X Z))
(id_comp' : ∀ {X Y : obj} (f : hom X Y), 𝟙 X ≫ f = f)
(comp_id' : ∀ {X Y : obj} (f : hom X Y), f ≫ 𝟙 Y = f)
(assoc'   : ∀ {W X Y Z : obj} (f : hom W X) (g : hom X Y) (h : hom Y Z),
(f ≫ g) ≫ h = f ≫ (g ≫ h))
```

A class is very similar to a structure with some Lean specific properties. The first three arguments give the structure of a category: the morphism type between two objects, the identity morphism and the composition of morphisms. The last three arguments are proofs of the basic properties that a category should have. Notice that in traditional mathematics, we do not see these proofs as data in the category. In Lean, this is a very common thing to do, because it nicely packs everything together.

We write `f ≫ g` for the composition of `f` and `g`. This should be interpreted as first applying `f` and then `g` instead of the other way around. We can write `X ⟶ Y` for `hom X Y`. Note that this arrow is similar to the arrow of a function type. However, these are in general very different types. First of all, a function type is always between two types and `X Y : obj` do not have to be types. Secondly, morphisms are not always interpreted as functions. That being said, for two objects `X Y : Type` in the category of types, the type of morphisms `X ⟶ Y = hom X Y` is defined as the function type `X → Y`.

For two categories `C` and `D` a functor from `C` to `D` is defined as an object of the type

```
structure functor (C : Type) [category C] (D : Type) [category D] :=
(obj       : C → D)
(map       : Π {X Y : C}, (X ⟶ Y) → (obj X ⟶ obj Y))
(map_id'   : (∀ (X : C), map (𝟙 X) = 𝟙 (obj X)))
```

```
(map_comp' : (∀ {X Y Z : C} (f : X ⟶ Y) (g : Y ⟶ Z),
map (f ≫ g) = map f ≫ map g))
```

where `obj` and `map` store the data of the functor and the other two arguments are proofs that it is a functor. Finally, we define natural transformations as objects of the type

```
structure nat_trans {C D : Type} [category C] [category D]
  (F G : C ⥤ D) :=
(app         : Π (X : C), (F.obj X) ⟶ (G.obj X))
(naturality' : ∀ {{X Y : C}} (f : X ⟶ Y),
(F.map f) ≫ (app Y) = (app X) ≫ (G.map f)
```

4.6. Simplicial sets in Lean

In this section we will describe how simplicial sets are defined in Lean. Some of the definitions have been simplified for readability. First we will define the simplex category. The objects of this category are a copy of the natural numbers.

```
def simplex_category := ℕ
```

There are two maps that distinguish these two types from each other. These are the functions `mk : ℕ → simplex_category` and `len : simplex_category → ℕ`. We write `[n] := mk n` for `n : ℕ`. However, unlike in Chapter 2, `[n]` is not the type containing the numbers $0 \leqslant i \leqslant n$. For this we use a different type called `fin`. Let `n : ℕ` be a natural number. The type `fin n` is a subtype of $\mathbb{N}$ containing the numbers $\{0, 1, \ldots, n-1\} = \{i \in \mathbb{N} \mid i < n\}$. In Lean this is defined as

```
def fin (n : ℕ) : Type := {i : ℕ // i < n}
```

This is short notation for the structure

```
structure fin (n : ℕ) :=
(val : ℕ) (property : val < n)
```

Objects of `fin n` are of the form `mk i hi`, where `i : ℕ` is a natural number and `hi : i < n` is a proof that $i < n$. Short notation for this is `⟨i, hi⟩`. For any `i : fin n` we have `i.val : ℕ` and `i.property : i < n`. The type `fin n` has four important maps

```
def cast_succ (i : fin n) : fin (n + 1) := ⟨i.val, _⟩

def succ (i : fin n) : fin (n + 1) := ⟨i.val + 1, _⟩

def cast_lt (i : fin (n+1)) (h : i.val < n) : fin n := ⟨i.val, h⟩

def pred (i : fin (n+1)) (h : i.val > 0) : fin n := ⟨i.val - 1, _⟩
```

Here the underscores stand for proofs that have been omitted. The maps `cast_succ` and `succ` go from `fin n` to `fin (n + 1)`. Given an input `i : fin n`, the map `cast_succ` returns the same value as the input. However we still need to proof that `i.val < n + 1`. Fortunately, this follows from `i.property` and the fact that

```
i.val < n < n + 1.
```

The map `succ` returns the value `i.val + 1` and similarly we can show that `i.val < n` implies `i.val + 1 < n + 1`.

The maps `cast_lt` and `pred` go from `fin (n + 1)` to `fin n`. Given an input `i` of type `fin (n+1)`, `cast_lt` returns the same value as the input. However, unlike `cast_succ`, we cannot prove that `i.val < n`. Therefore we need to add this as an extra argument to the function. The map `pred` returns the value `i.val - 1`. This is only possible if `i.val > 0`, which will be an extra argument to the function `pred`. We can prove that `i.val < n + 1` implies `i.val - 1 < n`, which finishes the function `pred`.

Now we will define the morphisms between two `a b : simplex_category` as

```
def hom (a b) := fin (a.len + 1) →m fin (b.len + 1)
```

Here $\to_m$ denotes the type of monotone or order preserving maps. We define the standard face maps and degeneracies as

```
def δ {n} (i : fin (n+2)) : [n] ⟶ [n+1] :=
λ (j : fin (n+1)), if j.cast_succ < i then j.cast_succ else j.succ

def σ {n} (i : fin (n+1)) : [n+1] ⟶ [n] :=
λ (j : fin (n+2)), if i.cast_succ < j then j.pred _ else i.cast_lt _
```

If we write out the values of these definitions, we get the same definitions given in Definition 2.2 and Definition 2.4. However, those definitions do not contain proofs that those functions are well-defined. That is, they do not contain a proof that the output of those functions are always elements of $[n + 1]$ and $[n]$ respectively. The definitions above in Lean do contains these proofs, using the maps `cast_succ`, `succ`, `cast_lt` and `pred`.

Face maps are defined as compositions of δ's. This is done using an inductive type similar to `list`. We start with the identity map and given a face map we can construct a new face map by composition with an extra δ map.

```
inductive face {n : ℕ} : Π {m : ℕ}, ([n] ⟶ [m]) → Sort*
| id                          : face (𝟙 [n])
| comp {m} (g: [n] ⟶ [m]) (i) : face g → face (g ≫ δ i)
```

Similarly we define degeneracies by

```
inductive degeneracy {n : ℕ} : Π {m : ℕ}, ([m] ⟶ [n]) → Sort*
| id                          : degeneracy (𝟙 [n])
| comp {k} (g: [k] ⟶ [n]) (i) : degeneracy g → degeneracy (σ i ≫ g)
```

We can show that every injective map is a face map. This is the nontrivial part of Theorem 2.3.

```
lemma face_of_injective {n m} (f : [n] ⟶ [m]) (hf : inj f) : face f
```

Theorem 2.7 will be formulated in Lean as

```
theorem decomp_degeneracy_face {n m} (f : [n] ⟶ [m]) :
  ∃ {k} (s : [n] ⟶ [k]) [degeneracy s] (d : [k] ⟶ [m]) [face d],
    f = s ≫ d
```

The Lean proofs of these theorems can be found in Appendix A.1.

Finally, we define a simplicial set as a contravariant functor from the simplex category to `Type`.

```
def sSet := simplex_categoryᵒᵖ ⥤ Type
```

In this definition, `Type` is the Lean-equivalent of the category **Set** in Definition 2.8.

4.7. Tactics

In Lean, an object of a certain type can be defined by giving an exact expression. This is what we have done so far in the examples of this chapter. However, sometimes these exact expressions can be long, complicated and hard to find. In these cases Lean has a shorter, clearer and easier way to give an object of a certain type. This is done using a special environment called *tactic mode*. Tactic mode starts with `begin` and ends with `end`. In this mode we write a program that generates an exact expression using commands separated by commas. These commands are called *tactics*. The exact expression generated in tactic mode is often long and unreadable. Therefore, tactic mode is mostly used for proofs. For proofs we only care about whether there exists a proof or not and we generally do not care about the exact expression.

A *tactic state* is a list of variables and a *goal*. The start of the tactic mode has a tactic state with all variables in the local environment and as goal the type of which we want to find a term. Each tactic can modify the tactic state. The last tactic has to give an exact term for the current goal. Our first tactic is `refl`. This tactic can solve goals of the form `x = y`, where `x` and `y` are definitionally equal. This means that `x` and `y` are syntactically the same after unfolding definitions. As an example, we will look at natural numbers.

```
example (n : ℕ) : n + 1 = succ n :=
begin
  refl
end
```

Indeed if we unfold the definitions of `1` and `+` we get

`n + 1 = n + (succ 0) = succ (n + 0) = succ n.`

Other tactics include `rw` for rewriting something in the goal or variables using an equality and `simp` for automatically simplifying the goal or variables using lemmas marked with `@[simp]`. The tactic `calc` can chain a number of equalities using transitivity of equality. The next example shows this using real numbers

```
example (a b : ℝ) : (a + b) + a = 2*a + b :=
begin
calc
  (a + b) + a = a + (a + b) : by rw add_comm
          ... = (a + a) + b : by rw ← add_assoc
          ... = a*2 + b     : by rw ← mul_two
          ... = 2*a + b     : by rw ← mul_comm,
end
```

However this is still a bit long. The tactic `linarith` can solve goals like these by itself. In this example this will look as follows.

```
example (a b : ℝ) : (a + b) + a = 2*a + b :=
begin
  linarith,
end
```

The tactic `linarith` will try to apply theorems to prove equalities and inequalities in specific types like $\mathbb{R}$.

There are a lot of tactics in Lean and together they improve readability of proofs.

4.8. Ethics and applications of Lean

Using Lean for proving mathematical theorems has many advantages over traditional proofs on paper, one of which is that the computer checks every single step in the proof. This means that, given the right definitions, a proof in Lean never contains mistakes. Another advantage is that Lean has many tools for simplifying and sometimes proving theorems by itself. However, this is only possible if Lean contains a general theory about the mathematical objects that the theorem is about.

This brings us right to the first trade-off with using Lean. A proof in Lean always needs every little detail. When Lean does not contain a general theory about the subject of the theorem, you have to prove all these details yourself. This can sometimes mean that it takes a lot of work to prove statements that would be considered trivial to the reader.

This thesis is very theoretical, so there are not many ethical aspects. However, we can ask ourselves whether we want Lean and computer assisted proofs to be the future of mathematics or not. An important part of mathematics is being able to communicate theorems and their proofs to other mathematicians. This can be difficult for proofs in Lean, because not every mathematician is familiar with the language. Computers can make proofs precise, but not necessarily intuitive and easy to understand. In some cases computer proofs are far from intuitive. A well known example of this is the four colour theorem. This is a theorem that says that in any map, there is a way to give each country one of four colours, such that each border has different colours on both sides. This theorem has been proved using computers. However, many mathematicians are still not satisfied with this proof, because it is not intuitive and hard to check for a human. In some cases, we need an absolute certainty that a proof is correct. In these cases Lean can be a very useful tool. However, in other cases, an intuitive proof is enough to convince the reader that a theorem is true.

Lean can be used as a programming language similar to Haskell. You can write all sorts of algorithms in Lean. However, unlike many programming languages, we can prove that these algorithms give the result we are looking for. A first example that has already been implemented in Lean is a sorting algorithm. After defining the algorithm, we can prove that the sorting algorithm always results in a sorted permutation of the original list. This has many applications in places where there is no room for errors in software. It is also used by AMD [5] and Intel [4] for proving correctness of their complex computer chips.

5. Traversals in Lean

In this chapter we will create a basic theory of traversals in Lean. This includes the application of a map to a traversal. In theory, this definition is not very complicated since it has nice properties, such as Theorem 3.2. However, it turns out to be quite difficult to prove these properties using the definitions directly. We will therefore prove some lemmas to make this easier. In the last section we prove Theorem 3.10 in Lean.

First we define the type `pm` as an enumerated type that encodes $\{+, -\}$. For a natural number `n`, we define the types `edge n` and `traversal n` like in Definition 3.1.

```
inductive pm
| plus  : pm
| minus : pm

def edge (n : ℕ) := fin (n+1) × pm

def traversal (n : ℕ) := list (edge n)
```

We define the application of a map to an edge by iterating over each value in the domain of the map and checking if this value gets mapped to the value of the edge. For a positive edge, we iterate from high values to low values in the domain. For a negative edge, we iterate from low values to high values in the domain.

```
def apply_map_to_plus {n m} (i : fin (n.len+1)) (α : m ⟶ n) :
  Π (j : ℕ), j < m.len+1 → traversal m.len
| 0       h0  := if α.to_preorder_hom 0 = i then ⟦(0, +)⟧ else ⟦⟧
| (j + 1) hj  :=
if α.to_preorder_hom ⟨j+1,hj⟩ = i
then (⟨j+1, hj⟩, +) :: (apply_map_to_plus j (nat.lt_of_succ_lt hj))
else apply_map_to_plus j (nat.lt_of_succ_lt hj)

def apply_map_to_min {n m} (i : fin (n.len+1)) (α : m ⟶ n) :
  Π (j : ℕ), j < m.len+1 → traversal m.len
| 0       h0  := if α.to_preorder_hom m.last = i then ⟦(m.last, -)⟧ else ⟦⟧
| (j + 1) hj  :=
  if α.to_preorder_hom ⟨m.len-(j+1), nat.sub_lt_succ _ _⟩ = i
  then (⟨m.len-(j+1), nat.sub_lt_succ _ _⟩, -) ::
  (apply_map_to_min j (nat.lt_of_succ_lt hj))
  else apply_map_to_min j (nat.lt_of_succ_lt hj)
```

For a general edge, we can do a case distinction on the sign and apply the right map.

```
def apply_map_to_edge {n m} (α : m ⟶ n) :
  edge n.len → traversal m.len
| (i, +) := apply_map_to_plus i α m.last.1 m.last.2
| (i, -) := apply_map_to_min i α m.last.1 m.last.2
```

For the last function `apply_map_to_edge α e`, we have the special notation `e · α`. In most proofs, we will not use the definition of `apply_map_to_edge` directly because of its complexity. Instead we will use two nice properties of this function. The first property is that the elements of `apply_map_to_edge α e` are all edges with the same sign as `e` and whose value get mapped to the value of `e`.

```
lemma edge_in_apply_map_to_edge_iff {n m} (α : m ⟶ n) :
  ∀ e₁ e₂, e₁ ∈ e₂ · α ↔ (α.to_preorder_hom e₁.1, e₁.2) = e₂
```

The second property of this map is that the values of `e · α` are strictly decreasing if `e` is a positive edge and strictly increasing if `e` is a negative edge. We will define a new order on the edges that combines these two cases such that the result of `apply_map_to_edge` is always sorted with respect to this order. In this way we do no have to repeat the same arguments for positive and negative edges. We order positive edges from high values to low values and negative edges from low values to high values. Lastly, we put negative edges before the positive edges to make the order linear.

```
def edge.lt {n} : edge n → edge n → Prop
| ⟨i, -⟩ ⟨j, -⟩ := i < j
| ⟨i, -⟩ ⟨j, +⟩ := true
| ⟨i, +⟩ ⟨j, -⟩ := false
| ⟨i, +⟩ ⟨j, +⟩ := i > j
```

Now the second property of `apply_map_to_edge` is that its result is always sorted with respect to the order above.

```
lemma apply_map_to_edge_sorted {n m : simplex_category} (α : m ⟶ n) :
  ∀ (e : edge n.len), sorted (e · α)
```

These two properties make sure that our definition of `apply_map_to_edge` is consistent with Equations (3.1) and (3.2). Strictly sorted lists are very useful because they have a few nice properties that we can use. Firstly, we can prove using induction that two sorted traversals are equal if they contain the same elements.

```
theorem eq_of_sorted_of_same_elem {n : ℕ} (θ₁ θ₂ : traversal n) :
  sorted θ₁ → sorted θ₂ → (Π e, e ∈ θ₁ ↔ e ∈ θ₂) → θ₁ = θ₂
```

The order has to be strict, because we have duplicates in the traversals θ_1 and θ_2.

Secondly, appending two sorted lists gives a sorted lists if all elements in the first list are less than all elements in the second list.

```
theorem append_sorted {n : ℕ} (θ₁ θ₂ : traversal n) :
  sorted θ₁ → sorted θ₂ → (∀ e₁ ∈ θ₁, ∀ e₂ ∈ θ₂, e₁ < e₂) →
    sorted (θ₁ ++ θ₂)
```

We now define the action of a map α on a traversal θ inductively by

```
def apply_map {n m} (α : m ⟶ n) :
  traversal n.len → traversal m.len
| ⟦⟧        := ⟦⟧
| (e :: t) := (e · α) ++ apply_map t
```

This means: apply the map to each edge and append all the resulting traversals together. We can also write this as $\theta \cdot \alpha$. Using induction and the previous two lemmas we can show that

```
lemma edge_in_apply_map_iff {n m} (α : m ⟶ n) (θ : traversal n.len) :
  ∀ (e : edge m.len), e ∈ θ · α ↔ (α.to_preorder_hom e.1, e.2) ∈ θ

lemma apply_map_preserves_sorted {n m} (α : m ⟶ n) (θ : traversal n.len) :
  sorted θ → sorted (θ · α)
```

5.1. Simplicial set of traversals

We define $\mathbb{T}_0$ as the simplicial set with n-traversals as n-simplices and the action $\theta \cdot \alpha$ of a map α on θ. For $\mathbb{T}_0$ to be a simplicial set, we need the following two properties

- Applying the identity does not change a traversal, so $\theta \cdot 𝟙$ n = θ
- Applying a composition of two maps is the same as applying them one by one, so $\theta \cdot (\alpha \gg \beta) = (\theta \cdot \beta) \cdot \alpha$.

Using induction on the traversal, it suffices to show these statements for individual edges.

Applying a map to an edge gives a sorted traversal. This means we can apply eq_of_sorted_of_same_elem. It remains to show that the traversals on both sides of each equality contain the same elements. This can be solved by the simplifier, which uses edge_in_apply_map_to_edge_iff and edge_in_apply_map_iff.

```
lemma apply_id {n} : ∀ (θ : traversal n.len), θ · 𝟙 n = θ
| ⟦⟧        := rfl
| (e :: θ) :=
begin
  unfold apply_map,                        -- (e :: θ) · 𝟙 n = e :: θ
  rw [apply_id θ], change _ = ⟦e⟧ ++ θ,  -- e · 𝟙 n ++ θ · 𝟙 n = e :: θ
  rw list.append_left_inj,                 -- e · 𝟙 n ++ θ = ⟦e⟧ ++ θ
  apply eq_of_sorted_of_same_elem,        -- e · 𝟙 n = ⟦e⟧
  { apply apply_map_to_edge_sorted },     -- (e · 𝟙 n).sorted
  { exact list.sorted_singleton h },      -- sorted ⟦e⟧
  { intro e', simp }                       -- e' ∈ e · 𝟙 n ↔ e' ∈ ⟦e⟧
end
```

```
lemma apply_comp {n m l} (α : m ⟶ n) (β : n ⟶ l) :
  ∀ (θ : traversal l.len), θ · α ≫ β = (θ · β) · α
| ⟦⟧        := rfl
| (e :: θ) :=
begin
  unfold apply_map,                        -- (e::θ) · α ≫ β
                                           -- = (e::θ) · β · α

  rw [apply_map_append, apply_comp],       -- e · α ≫ β ++ θ · α ≫ β
  rw list.append_left_inj,                 -- = e · β · α ++ θ · α ≫ β
  apply eq_of_sorted_of_same_elem,         -- e · α ≫ β = (e · β) · α
  { apply apply_map_to_edge_sorted },      -- (e · α ≫ β).sorted
  { apply apply_map_preserves_sorted,      -- (e · β · α).sorted
    apply apply_map_to_edge_sorted },      -- (e · β).sorted
  { intro e', simp }                       -- e' ∈ e · α ≫ β ↔
                                           -- e' ∈ e · β · α
end
```

In the comments of the proofs we can see what each line is trying to prove. Both proofs start by rewriting the statement and applying the induction hypothesis. Then we apply the lemma `eq_of_sorted_of_same_elem` and show that the traversals in question are sorted. The last line in both proofs shows that the traversals contain the same elements. This is done automatically using `simp`. This tactic searches for lemmas and theorems that simplify and in this cases solve the goal. Using these lemmas we can finaly define $\mathbb{T}_0$ as

```
def 𝕋₀ : sSet :=
{ obj       := λ n, traversal n.unop.len,
  map       := λ x y α, apply_map α.unop,
  map_id'   := λ n, funext (λ θ, apply_id θ),
  map_comp' := λ l n m β α, funext(λ θ, apply_comp α.unop β.unop θ)}
```

We define pointed traversals as pairs of traversals.

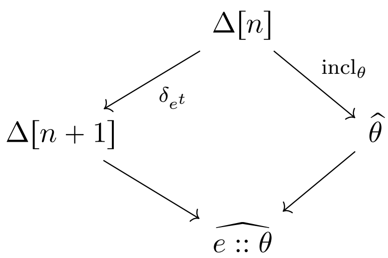
```
def pointed_traversal (n : ℕ) := traversal n × traversal n
```

Applying a map to a pointed traversal is defined by applying the map to each component of the pair. It is not hard to prove that this defines a simplicial set $\mathbb{T}_1$ of pointed traversals. For a position p between 0 and the length of the traversal, it is hard determine the corresponding position after applying a map. This is the main reason for our definition of a pointed traversal as a pair instead of a traversal with a position. We can now also define the morphisms `dom` and `cod` from $\mathbb{T}_1$ to $\mathbb{T}_0$ from Definition 3.5 and Definition 3.6.

5.2. Geometric realization in Lean

In Definition 3.8, we defined the geometric realization of a traversal as the colimit over a single diagram. In Lean, this means we first have to construct the index category for this diagram, after that we can define the diagram and finally the colimit over this diagram. Intuitively, this definition stitches all the simplices in the traversals together at once. This definition can be found in Appendix B.2. However, this definition uses list indexing. For example, in Definition 3.8, we can see multiple instances of $\theta(i)$, which means we take the ith edge in θ. Working with lists in Lean is often easier by recursion on the list. In this case, it means that we first define the geometric realization of the empty traversal. Then we define $\widehat{e :: \theta}$ recursively by stitching an extra copy of $\Delta[n+1]$ to $\widehat{\theta}$. We need the right properties of the base case and a recursive relation that is satisfied by the geometric realization.

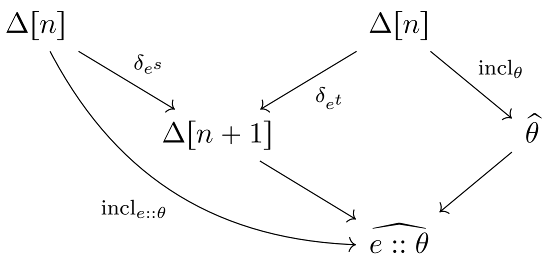
The geometric realization of the empty traversal is by Definition 3.8 the colimit over a single copy of $\Delta[n]$. This is clearly isomorphic to $\Delta[n]$. For the recursive relation, we use the fact that $\widehat{e :: \theta}$ fits into the following pushout square



(5.1)

Here incl_θ is defined as the first inclusion from $\Delta[n]_0$ into $\widehat{\theta}$ in Definition 3.8. However, for our recursive definition, we cannot use the fact that $\widehat{\theta}$ is the colimit over the diagram in Definition 3.8. A solution to this is to add the map incl_θ to the recursion. This means that we will recursively construct a simplicial set $\widehat{\theta}$ together with a map $\mathrm{incl}_\theta : \Delta[n] \to \widehat{\theta}$.

For the empty traversal we take the identity map $\mathrm{id} : \Delta[n] \to \Delta[n]$. For a traversal $e :: \theta$, we define $\widehat{e :: \theta}$ as the pushout of Diagram (5.1). We define $\mathrm{incl}_{e::\theta}$ as the composition of δ_{e^s} with the inclusion $\Delta[n+1] \to \widehat{e :: \theta}$ in Diagram (5.1). In Diagram (5.2) we can see the full recursion step for the definition of $\widehat{\theta}$ and incl_θ. This diagram looks very similar to the left part of the diagram in Definition 3.8.



(5.2)

Using this recursion, we can define $\widehat{\theta}$ and incl_θ in a very short way.

```
def bundle : Π (θ : traversal n), Σ (g : sSet), Δ[n] ⟶ g
| ⟦⟧         := ⟨Δ[n], 𝟙 Δ[n]⟩
| (e :: θ) :=
```

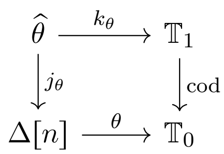
```
    let colim := sSet_pushout (to_sSet_hom (δ (t e))) (bundle θ).2 in
    ⟨colim.cocone.X, to_sSet_hom (δ (s e)) ≫ pushout_cocone.inl colim.cocone⟩

def geom_real_rec {n} (θ : traversal n) : sSet := (bundle θ).1

def geom_real_incl {n} (θ : traversal n) :
  Δ[n] ⟶ geom_real_rec θ := (bundle θ).2
```

5.3. Geometric realization as a pullback

In this section we will prove Theorem 3.10. This theorem says that the geometric realization is a weak pullback in the square

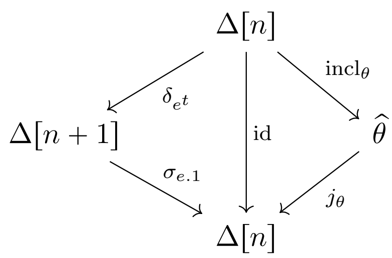


5.3.1. Construction of j_θ

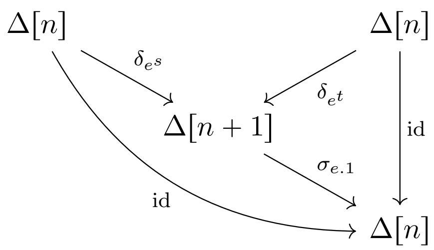
First, we construct the map $j_\theta : \widehat{\theta} \to \Delta[n]$ with the property that $j_\theta \circ \mathrm{incl}_\theta = \mathrm{id}$. We will again use recursion to construct this map. For the empty traversal we define $j_{[]} := \mathrm{id} : \Delta[n] \to \Delta[n]$ which clearly satisfies

$$j_{[]} \circ \mathrm{incl}_{[]} = \mathrm{id} \circ \mathrm{id} = \mathrm{id}\,.$$

For the recursion step, we have to construct a map $j_{e::\theta}$ from the pushout $\widehat{e :: \theta}$ to $\Delta[n]$. We can construct a map from a pushout by giving a pushout cocone over the diagram of $\widehat{e :: \theta}$. We define this cocone by



We have to prove that this diagram commutes and that $\mathrm{incl}_{e::\theta} \circ j_{e::\theta} = \mathrm{id}$. The right triangle in the diagram commutes by the induction hypothesis. After unfolding the definition of $\mathrm{incl}_{e::\theta}$ it suffices to show that the diagram



commutes. Notice that by the definition of e^s and e^t, we have that e^s and e^t are each either equal to $e.1$ or $e.1 + 1$ depending on the sign of e. Therefore, by the third simplicial identity from Theorem 2.6 we can show that the above diagram commutes. Using induction, we construct a map $j_\theta : \widehat{\theta} \to \Delta[n]$ with the property that $j_\theta \circ \mathrm{incl}_\theta = \mathrm{id}$ for each n-traversal θ. In Lean, we get the following definition of j_θ:

```
def j_rec_bundle : Π (θ : traversal n),
{j : geom_real_rec θ ⟶ Δ[n] // geom_real_incl θ ≫ j = 𝟙 Δ[n]}
| ⟦⟧        := ⟨𝟙 Δ[n], rfl⟩
| (e :: θ) :=
  ⟨(bundle_colim e θ).is_colimit.desc
    (pushout_cocone.mk (to_sSet_hom (σ e.1)) (j_rec_bundle θ).1 _), _⟩
```

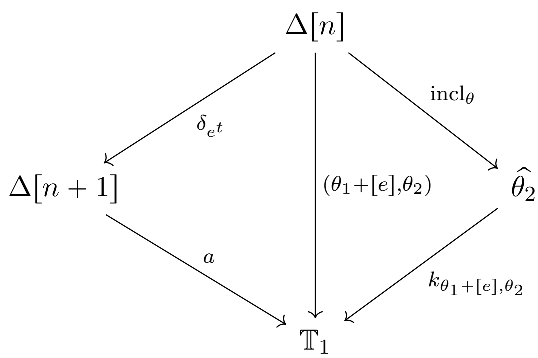
Here the proof that the diagram commutes has been replaced by two underscores for readability.

5.3.2. Construction of k_θ

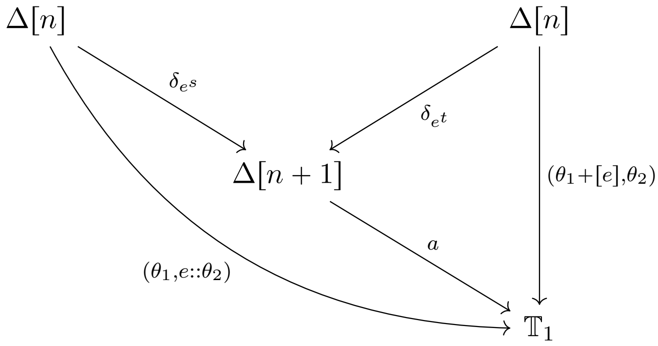
The map k_θ is more complicated than j_θ, because in the definition of $k_{e::\theta}$ we will not be using the map k_θ. Instead we will define an extra help function $k_{\theta_1,\theta_2} : \widehat{\theta_2} \to \mathbb{T}_1$. We will use recursion on θ_2 and the definition of $k_{\theta_1,e::\theta_2}$ uses the map $k_{\theta_1+[e],\theta_2}$. Again we will need an extra property for constructing a cocone. This property is $k_{\theta_1,\theta_2} \circ \mathrm{incl}_\theta = (\theta_1, \theta_2)$. Here we interpret the pointed n-traversal (θ_1, θ_2) as a morphism $(\theta_1, \theta_2) : \Delta[n] \to \mathbb{T}_1$ using the Yoneda lemma. In case $\theta_2 = []$ we define $k_{\theta_1,[]} = (\theta_1, []) : \Delta[n] \to \mathbb{T}_1$. This clearly satisfies the property because

$$k_{\theta_1,[]} \circ \mathrm{incl}_\theta = k_{\theta_1,[]} \circ \mathrm{id} = k_{\theta_1,[]} = (\theta_1, []).$$

We define $k_{\theta_1,e::\theta_2}$ by the cocone



where $a = (\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)], (e^t, e.2) :: \theta_2 \cdot \sigma_{e.1})$. Similar to j_θ, it remains to show that the following diagram commutes:



This simplifies to the following two equations:

$$(\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)], (e^t, e.2) :: \theta_2 \cdot \sigma_{e.1}) \cdot \delta_{e^s} = (\theta_1, e :: \theta_2),$$
$$(\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)], (e^t, e.2) :: \theta_2 \cdot \sigma_{e.1}) \cdot \delta_{e^t} = (\theta_1 + [e], \theta_2).$$

We show this using the simplicial identities and the following four equalities:

$$(e^s, e.2) \cdot \delta_{e^s} = [], \qquad (e^t, e.2) \cdot \delta_{e^s} = [e],$$
$$(e^s, e.2) \cdot \delta_{e^t} = [e], \qquad (e^t, e.2) \cdot \delta_{e^t} = [].$$

These can be easily proved using case distinction on the sign of e and by unfolding the definitions of s, t and δ. We can now define k_{θ_1,θ_2} inductively and $k_\theta := k_{[],\theta}$. In lean we get the following expression:

```
def k_rec_bundle : Π (θ θ' : traversal n),
{k : geom_real_rec θ ⟶ 𝕋₁ // geom_real_incl θ ≫ k = simplex_as_hom (θ', θ)}
| ⟦⟧        θ' := ⟨simplex_as_hom (θ',⟦⟧), rfl⟩
| (e :: θ) θ' :=
let a := (apply_map (σ e.1) θ' ++ ⟦(eˢ, e.2)⟧,
  (eᵗ, e.2) :: apply_map (σ e.1) θ) in
let k_θ := k_rec_bundle θ (θ' ++ ⟦e⟧) in
⟨(bundle_colim e θ).is_colimit.desc
  (pushout_cocone.mk (simplex_as_hom a) k_θ.1 _), _⟩
```

Again the proofs have been replaced by two underscores.

5.3.3. Pullback cone

The next step is to show that the maps j_θ and k_θ form a pullback cone, which is a cone over a cospan. This means that $\theta \circ j_\theta = \mathrm{cod} \circ k_\theta$. This is a special case of the equality $(\theta_1 + \theta_2) \circ j_{\theta_2} = \mathrm{cod} \circ k_{\theta_1,\theta_2}$ with $\theta_1 = []$. In other words, the diagram

$$\begin{array}{ccc}
\widehat{\theta_2} & \xrightarrow{k_{\theta_1,\theta_2}} & \mathbb{T}_1 \\
\big\downarrow{\scriptstyle j_{\theta_2}} & & \big\downarrow{\scriptstyle \mathrm{cod}} \\
\Delta[n] & \xrightarrow{\theta_1+\theta_2} & \mathbb{T}_0
\end{array}$$

commutes. We will prove this more general fact by induction on θ_2. For the empty traversal, we get

$$(\theta_1 + []) \circ j_{[]} = (\theta_1 + []) \circ \mathrm{id} = (\theta_1 + []) = \mathrm{cod} \circ (\theta_1, []) = \mathrm{cod} \circ k_{\theta_1,[]}.$$

For the induction step, we have to prove that the maps $(\theta_1+e :: \theta_2) \circ j_{e::\theta_2}$ and $\mathrm{cod} \circ k_{\theta_1,e::\theta_2}$ are equal. These are both maps from the pushout $\widehat{e :: \theta_2}$ and therefore correspond to pushout cocones. The two maps are equal if and only if these pushout cocones are equal. This means that we have to prove the following two equalities.

$$\begin{aligned}
(\theta_1 + e :: \theta_2) \circ \sigma_{e.1} &= \mathrm{cod} \ \circ \ a, \\
(\theta_1 + e :: \theta_2) \circ j_{\theta_2} &= \mathrm{cod} \ \circ \ k_{\theta_1+[e],\theta_2},
\end{aligned}$$

where $a = (\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)], (e^t, e.2) :: \theta_2 \cdot \sigma_{e.1})$. For the first equality, we use the fact that $e \cdot \sigma_{e.1} = [(e^s, e.2), (e^t, e.2)]$. Now we get

$$\begin{aligned}
(\theta_1 + e :: \theta_2) \circ \sigma_{e.1} &= \theta_1 \cdot \sigma_{e.1} + e \cdot \sigma_{e.1} + \theta_2 \cdot \sigma_{e.1} \\
&= \theta_1 \cdot \sigma_{e.1} + [(e^s, e.2), (e^t, e.2)] + \theta_2 \cdot \sigma_{e.1} \\
&= (\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)]) + ((e^t, e.2) :: \theta_2 \cdot \sigma_{e.1}) \\
&= \mathrm{cod} \circ a.
\end{aligned}$$

For the second equality, we use the induction hypothesis on $\theta_1 + [e]$. We have

$$(\theta_1 + e :: \theta_2) \circ j_{\theta_2} = ((\theta_1 + [e]) + \theta_2) \circ j_{\theta_2} = \mathrm{cod} \circ k_{\theta_1+[e],\theta_2}.$$

These two equalities show that $(\theta_1 + e :: \theta_2) \circ j_{e::\theta_2} = \mathrm{cod} \circ k_{\theta_1,e::\theta_2}$. Using induction we have $(\theta_1 + \theta_2) \circ j_{\theta_2} = \mathrm{cod} \circ k_{\theta_1,\theta_2}$ for any n-traversals θ_1 and θ_2. After translating this proof to Lean we get the following theorem.

```
lemma j_comp_θ_eq_k_comp_cod : Π (θ₁ θ₂ : traversal n),
    j_rec θ₂ ≫ (θ₁ ++ θ₂).as_hom = k_rec' θ₁ θ₂ ≫ cod
```

Filling in $\theta_1 = []$ and $\theta_2 = \theta$ gives

$$\theta \circ j_\theta = \mathrm{cod} \circ k_\theta.$$

This means that j_θ and k_θ form a pullback cone.

5.3.4. Weak pullback

To show that the pullback cone is a weak pullback, we have to show that for any other pullback cone, there exist a lift to the geometric realization. It suffices to show that the pullback cone is a weak pullback pointwise. In other words, for every $m \in \mathbb{N}$ the diagram

$$\begin{array}{ccc} \widehat{\theta}_m & \xrightarrow{(k_\theta)_m} & (\mathbb{T}_1)_m \\ \big\downarrow{\scriptstyle (j_\theta)_m} & & \big\downarrow{\scriptstyle \mathrm{cod}_m} \\ \Delta[n]_m & \xrightarrow{\theta_m} & (\mathbb{T}_0)_m \end{array}$$

is a weak pullback in the category **Set**. In Lean this will be in the category `Type`. This can be reformulated to the statement that for every $\alpha \in \Delta[n]_m$ and $(\eta_1, \eta_2) \in (\mathbb{T}_1)_m$ such that

$$\theta \cdot \alpha = \theta_m(\alpha) = \mathrm{cod}_m(\eta_1, \eta_2) = \eta_1 + \eta_2,$$

there exists a simplex $x \in \widehat{\theta}_m$ with $(j_\theta)_m(x) = \alpha$ and $(k_\theta)_m(x) = (\eta_1, \eta_2)$. We call such a simplex x a lift of α and (η_1, η_2). We will again prove this theorem using induction. However, we can only use induction on statements about the map k if we use the version with two parameters. This means we have to find a more general statement.

For any $\alpha \in \Delta[n]_m$ and $(\eta_1, \eta_2) \in (\mathbb{T}_1)_m$ such that

$$\theta_2 \cdot \alpha = \eta_1 + \eta_2,$$

we construct a simplex $x \in (\widehat{\theta_2})_m$ with $(j_{\theta_2})_m(x) = \alpha$ and $(k_{\theta_1,\theta_2})_m(x) = (\theta_1 \cdot \alpha + \eta_1, \eta_2)$. In other words, we have to find a lift in the diagram

$$\begin{array}{ccc} (\widehat{\theta_2})_m & \xrightarrow{(k_{\theta_1,\theta_2})_m} & (\mathbb{T}_1)_m \\ \big\downarrow{\scriptstyle (j_{\theta_2})_m} & & \big\downarrow{\scriptstyle \mathrm{cod}_m} \\ \Delta[n]_m & \xrightarrow{(\theta_1+\theta_2)_m} & (\mathbb{T}_0)_m \end{array}$$

for some $\alpha \in \Delta[n]_m$ and $(\theta_1 \cdot \alpha + \eta_1, \eta_2) \in (\mathbb{T}_1)_m$.

We will prove this statement by induction on θ_2. For the base case $\theta_2 = []$, the diagram simplifies to

$$\begin{array}{ccc} \Delta[n]_m & \xrightarrow{(\theta_1,[])_m} & (\mathbb{T}_1)_m \\ \big\downarrow{\scriptstyle \mathrm{id}} & & \big\downarrow{\scriptstyle \mathrm{cod}_m} \\ \Delta[n]_m & \xrightarrow{(\theta_1)_m} & (\mathbb{T}_0)_m \end{array}$$

Suppose that $\eta_1 + \eta_2 = [] \cdot \alpha = []$. Then $\eta_1 = \eta_2 = []$. We can choose $x = \alpha \in \Delta[n]_m$, because

$$\mathrm{id}(\alpha) = \alpha,$$
$$(\theta_1, [])_m(\alpha) = (\theta_1 \cdot \alpha, [] \cdot \alpha) = (\theta_1 \cdot \alpha + [], []) = (\theta_1 \cdot \alpha + \eta_1, \eta_2).$$

For the induction step, we will be looking at the diagram

$$\begin{array}{ccc} (\widehat{e :: \theta_2})_m & \xrightarrow{(k_{\theta_1, e::\theta_2})_m} & (\mathbb{T}_1)_m \\ \Big\downarrow{\scriptstyle (j_{e::\theta_2})_m} & & \Big\downarrow{\scriptstyle \mathrm{cod}_m} \\ \Delta[n]_m & \xrightarrow{(\theta_1 + e::\theta_2)_m} & (\mathbb{T}_0)_m \end{array}$$

Suppose that $\eta_1 + \eta_2 = (e :: \theta_2) \cdot \alpha = e \cdot \alpha + \theta_2 \cdot \alpha$. We will distinguish three cases: the position corresponding to (η_1, η_2) in the traversal $e \cdot \alpha + \theta_2 \cdot \alpha$ lies

- before $e \cdot \alpha$. In other words, at the start of the traversal, meaning that $\eta_1 = []$;
- after $e \cdot \alpha$. This means the position lies inside or on the edge of $\theta_2 \cdot \alpha$;
- inside $e \cdot \alpha$.

For the first case we have $\eta_1 = []$, so $\eta_2 = (e :: \theta_2) \cdot \alpha$. We choose $x = (\mathrm{incl}_{e::\theta_2})_m(\alpha)$. By the property of the map $j_{e::\theta_2}$ we have

$$(j_{e::\theta_2})_m((\mathrm{incl}_{e::\theta_2})_m(\alpha)) = (j_{e::\theta_2} \circ \mathrm{incl}_{e::\theta_2})_m(\alpha) = \mathrm{id}(\alpha) = \alpha.$$

By the property of $(k_{\theta_1, e::\theta_2})_m$ we have

$$\begin{aligned}(k_{\theta_1, e::\theta_2})_m((\mathrm{incl}_{e::\theta_2})_m(\alpha)) &= (k_{\theta_1, e::\theta_2} \circ \mathrm{incl}_{e::\theta_2})_m(\alpha) = (\theta_1, e :: \theta_2)_m(\alpha) \\ &= (\theta_1 \cdot \alpha, (e :: \theta_2) \cdot \alpha) = (\theta_1 \cdot \alpha + \eta_1, \eta_2).\end{aligned}$$

For the second case we have that $e \cdot \alpha$ is fully contained in η_1, so there exists some traversal η_1' with $\eta_1 = e \cdot \alpha + \eta_1'$. Now it follows that $e \cdot \alpha + \eta_1' + \eta_2 = e \cdot \alpha + \theta_2 \cdot \alpha$, so $\eta_1' + \eta_2 = \theta_2 \cdot \alpha$. By the induction hypothesis, we can find some $x' \in (\widehat{\theta_2})_m$ such that $(j_{\theta_2})_m(x') = \alpha$ and

$$\begin{aligned}(k_{\theta_1 + [e], \theta_2})_m(x') &= ((\theta_1 + [e]) \cdot \alpha + \eta_1', \eta_2) \\ &= (\theta_1 \cdot \alpha + e \cdot \alpha + \eta_1', \eta_2) \\ &= (\theta_1 \cdot \alpha + \eta_1, \eta_2).\end{aligned}$$

By defining x as the image of x' under the inclusion $\widehat{\theta_2} \subseteq \widehat{e :: \theta_2}$, we get by the above equations that

$$\begin{aligned}(j_{e::\theta_2})_m(x) &= (j_{\theta_2})_m(x') = \alpha, \\ (k_{\theta_1, e::\theta_2})_m(x') &= (k_{\theta_1 + [e], \theta_2})_m(x') = (\theta_1 \cdot \alpha + \eta_1, \eta_2).\end{aligned}$$

We only have to find a lift for the last case, where the position is inside $e \cdot \alpha$. This means that there is some η_2' such that $e \cdot \alpha = \eta_1 + \eta_2'$ and $\eta_2 = \eta_2' + \theta_2 \cdot \alpha$. In this cases we will find some $\beta \in \Delta[n+1]_m$ and choose $x \in (\widehat{e :: \theta_2})_m$ as the image of β under the inclusion $\Delta[n+1]_m \to (\widehat{e :: \theta_2})_m$ in diagram (5.1). This $\beta : [m] \to [n+1]$ has to satisfy the following properties:

$$\begin{aligned} \sigma_{e.1} \circ \beta &= \alpha, \\ a \cdot \beta &= (\theta_1 \cdot \alpha + \eta_1, \eta_2' + \theta_2 \cdot \alpha), \end{aligned}$$

where $a = (\theta_1 \cdot \sigma_{e.1} + [(e^s, e.2)], (e^t, e.2) :: \theta_2 \cdot \sigma_{e.1})$. Suppose we have a β with the first property, then after filling in a, the second equality simplifies to

$$([(e^s, e.2)], [(e^t, e.2)]) \cdot \beta = (\eta_1, \eta_2').$$

This means it suffices to find a β with the following three properties

$$\begin{aligned} \sigma_{e.1} \circ \beta &= \alpha, \\ (e^s, e.2) \cdot \beta &= \eta_1, \\ (e^t, e.2) \cdot \beta &= \eta_2'. \end{aligned}$$

Notice that $(e^s, e.2) \cdot \beta$, $(e^t, e.2) \cdot \beta$ and $\eta_1 + \eta_2' = e \cdot \alpha$ are sorted because applying a map to an edge gives a sorted traversal. This means that η_1 and η_2' are sorted as well. By the lemma `eq_of_sorted_of_same_elem`, it suffices to show for the last two equalities that the traversals contain the same edges. By the lemma `edge_in_apply_map_to_edge_iff`, the qualities above become

$$\sigma_{e.1} \circ \beta = \alpha, \tag{5.3}$$
$$(\beta(e'.1), e'.2) = (e^s, e.2) \iff e' \in \eta_1, \tag{5.4}$$
$$(\beta(e'.1), e'.2) = (e^t, e.2) \iff e' \in \eta_2'. \tag{5.5}$$

We can now define β such that these properties are satisfied.

$$\beta(i) := \begin{cases} \alpha(i), & \alpha(i) < e.1 \\ e^s, & (i, e.2) \in \eta_1 \\ e^t, & (i, e.2) \in \eta_2' \\ \alpha(i) + 1, & \alpha(i) > e.1 \end{cases}$$

This function is well-defined, because the middle two cases combine to

$$(i, e.2) \in \eta_1 + \eta_2' = e \cdot \alpha \iff (\alpha(i), e.2) = e \iff \alpha(i) = e.1.$$

The traversals η_1 and η_2' are disjoint because $\eta_1 + \eta_2' = e \cdot \alpha$ is strictly sorted. This means i always satisfies one of the conditions is the definition of β is satisfied.

It is not hard to show that β is order preserving. The values of e^s and e^t are each either $e.1$ or $e.1 + 1$, so in general $e.1 \leqslant e^s, e^t \leqslant e.1 + 1$. For $i \leqslant j$ we know that $\alpha(i) \leqslant \alpha(j)$ by the fact that α is order preserving. We will now do a case distinction on how these values compare to $e.1$.

- If $e.1 < \alpha(i) \leqslant \alpha(j)$ then $\beta(i) = \alpha(i) + 1 \leqslant \alpha(j) + 1 = \beta(j)$.

- If $e.1 = \alpha(i) < \alpha(j)$ then $\beta(i) \leqslant e.1 + 1 < \alpha(j) + 1 = \beta(j)$.
- If $\alpha(i) = e.1 = \alpha(j)$ then $\beta(i) \leqslant \beta(j)$ by the fact that the comparison of e^s and e^t is the same as the element wise comparison of η_1 and η_2', depending on $e.2$.
- If $\alpha(i) < \alpha(j) = e.1$ then $\beta(i) = \alpha(i) < e.1 \leqslant \beta(j)$.
- If $\alpha(i) \leqslant \alpha(j) < e.1$ then $\beta(i) = \alpha(i) \leqslant \alpha(j) = \beta(j)$.

In all cases we have $\beta(i) \leqslant \beta(j)$, so β is order preserving.

The map β was chosen in such a way that the properties (5.4) and (5.5) follow immediately from the definition. This means we only have to prove equality (5.3) which says that $\sigma_{e.1} \circ \beta = \alpha$. If $\alpha(i) < e.1$ then

$$\sigma_{e.1}(\beta(i)) = \sigma_{e.1}(\alpha(i)) = \alpha(i).$$

If $\alpha(i) > e.1$ then

$$\sigma_{e.1}(\beta(i)) = \sigma_{e.1}(\alpha(i) + 1) = \alpha(i) + 1 - 1 = \alpha(i).$$

If $\alpha(i) = e.1$ then

$$\sigma_{e.1}(\beta(i)) = \sigma_{e.1}(e^s) = \sigma_{e.1}(e^t) = e.1 = \alpha(i).$$

This means β also satisfies property (5.3). By choosing $x \in (\widehat{e :: \theta_2})_m$ as the image of β under the inclusion $\Delta[n+1]_m \to (\widehat{e :: \theta_2})_m$ in diagram (5.1), we can find a lift x in this case as well.

This means we have found a lift in each of the cases. Filling in $\theta_1 = []$ and $\theta_2 = \theta$ gives the following definition and lemmas in Lean:

```
def geom_real_rec_lift (θ : traversal n) {m} : Π (α : m ⟶ [n])
(θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (geom_real_rec θ).obj (opposite.op m)

lemma geom_real_rec_fac_j (θ : traversal n) {m} : Π (α : m ⟶ [n])
(θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (j_rec θ).app m.op (geom_real_rec_lift θ α θ₁ θ₂ hθ) = α

lemma geom_real_rec_fac_k (θ : traversal n) {m} : Π (α : m ⟶ [n])
(θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (k_rec θ).app m.op (geom_real_rec_lift θ α θ₁ θ₂ hθ) = (θ₁, θ₂)
```

The function `geom_real_rec_lift` gives the lift and two lemmas show that this lift is indeed a lift. Together, these theorems show that the geometric realization is a weak pullback. This means we have proved Theorem 3.10 in Lean, which was the main goal of this thesis.

6. Conclusion

We proved some basic properties about the simplex category in Lean. For example, the fact that a bijective morphism is the identity. We defined face maps and degeneracies in the simplex category as compositions of standard face maps and standard degeneracies respectively. We proved that a morphism in the simplex category can always be written as a composition of a face map and a degeneracy. In further research, this theorem can be extended with to fact that this decomposition is unique up to the simplicial identities. Formally, this means that the simplex category is equivalent to the quotient category of the free category generated by the standard face maps and degeneracies with the simplicial identities.

We set up a basic theory of traversals in Lean. In particular, we define traversals and the action of maps in the simplex category to a traversal. We show that this action defines a simplicial set of traversals. We also define a pointed traversal as a pair of two traversals. We also defined the geometric realization of a traversal as a repeated pushout. We constructed the maps j_θ and k_θ and formalized Theorem 3.10 which says that j_θ and k_θ form a weak pullback over the maps θ and cod in Lean. In further research, this can be extended to Theorem 3.10. This means proving that the lift constructed in the proof of Theorem 3.10 is unique. This will be the next step in this research.

Popular summary

By connecting different points, lines, triangles and pyramids, we can create all kinds of interesting objects. An example can be seen in Figure 6.1.

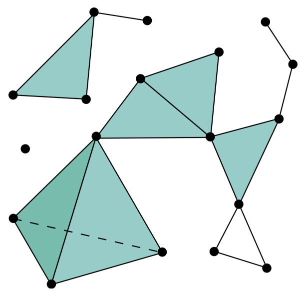


Figure 6.1.: An object constructed by points, lines, triangles and pyramids.

These objects are called *simplicial sets*. There are special simplicial sets, called *traversals*. For an example of a traversal, see Figure 6.2.

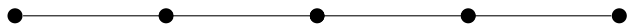


Figure 6.2.: An example of a traversal

Traversals can be used to describe paths in a simplicial set. For example, the red path in Figure 6.3 can be described by the traversal in Figure 6.2.

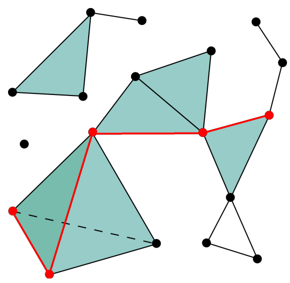


Figure 6.3.: A path in a simplicial set.

In this thesis we look at an important theorem about traversals from the paper "Effective Kan fibrations in simplicial sets" [6]. We will prove part of this theorem with a computer, using the computer language Lean. Lean is a computer language in which we can write mathematical statements and proofs. Lean checks each step in the proof and therefore ensures correctness of the proof.

Bibliography

[1] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Carnegie Mellon University, Jun 2018. URL: https://kilthub.cmu.edu/articles/journal_contribution/Theorem_Proving_in_Lean/6492902/1, doi:10.1184/R1/6492902.v1.

[2] Clemens Berger. Un groupoïde simplicial comme modèle de l'espace des chemins. *Bull. Soc. Math. France*, 123(1):1–32, 1995. URL: http://www.numdam.org/item?id=BSMF_1995__123_1_1_0.

[3] Greg Friedman. Survey Article: An elementary illustrated introduction to simplicial sets. *Rocky Mountain Journal of Mathematics*, 42(2):353 – 423, 2012. URL: https://doi.org/10.1216/RMJ-2012-42-2-353, doi:10.1216/RMJ-2012-42-2-353.

[4] John Harrison. Formal verification at Intel. In *Logic in Computer Science (LICS 2003)*, pages 45–54. IEEE Computer Society, 2003.

[5] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999.

[6] Benno van den Berg and Eric Faber. Effective kan fibrations in simplicial sets, 2020. arXiv:2009.12670.

[7] Benno van den Berg and Richard Garner. Topological and simplicial models of identity types. *ACM Trans. Comput. Log.*, 13(1):Art. 3, 44, 2012. URL: https://doi.org/10.1145/2071368.2071371, doi:10.1145/2071368.2071371.

A. Lean code: Simplicial sets

A.1. degeneracy_face.lean

```
import algebraic_topology.simplex_category
import set_theory.cardinal

open category_theory
namespace simplex_category
open_locale simplicial

/- Face maps are compositions of δ maps. -/
class inductive face {n : ℕ} : Π {m : ℕ}, ([n] ⟶ [m]) → Sort*
| id                              : face (𝟙 [n])
| comp {m} (g: [n] ⟶ [m]) (i) : face g → face (g ≫ δ i)

lemma le_of_face {n : ℕ} : Π {m : ℕ} (s : [n] ⟶ [m]) (hs : face s), n ≤ m
| n s face.id              := le_refl n
| m s (face.comp g i hg) := nat.le_succ_of_le (le_of_face g hg)

lemma face_comp_face {l m : ℕ} {g : [l] ⟶ [m]} (hg : face g) :
Π {n : ℕ} {f : [m] ⟶ [n]} (hf : face f), face (g ≫ f)
| m f face.id              := begin rw category.comp_id, exact hg, end
| n s (face.comp f i hf) :=
begin
  rw ←category.assoc g f (δ i),
  exact face.comp _ _ (face_comp_face hf),
end

instance δ_split_mono {n} (i : fin (n+2)) : split_mono (δ i) :=
begin
  by_cases hi : i < fin.last (n+1),
  { rw ←fin.cast_succ_cast_pred hi,
    exact ⟨σ i.cast_pred, δ_comp_σ_self⟩,},
  { push_neg at hi,
    have hi' : i ≠ 0, from ne_of_gt (gt_of_ge_of_gt hi fin.last_pos),
    rw ←fin.succ_pred i hi',
    exact ⟨σ (i.pred hi'), δ_comp_σ_succ⟩,},
```

```
end

lemma split_mono_of_face {n : ℕ} : Π {m} {f : [n] ⟶ [m]},
  face f → split_mono f
| n f face.id            := ⟨𝟙 [n], category.id_comp (𝟙 [n])⟩
| m f (face.comp g i hg) :=
begin
  rcases split_mono_of_face hg with ⟨g_ret, g_comp⟩,
  rcases (infer_instance : split_mono (δ i)) with ⟨δ_ret, δ_comp⟩,
  refine ⟨δ_ret ≫ g_ret, _⟩,
  simp only [auto_param_eq] at *,
  rw [category.assoc, ←category.assoc (δ i) δ_ret g_ret],
  rw [δ_comp, category.id_comp, g_comp],
end

/- Degeneracy maps are compositions of σ maps. -/
class inductive degeneracy {n : ℕ} : Π {m : ℕ}, ([m] ⟶ [n]) → Sort*
| id                          : degeneracy (𝟙 [n])
| comp {k} (g: [k] ⟶ [n]) (i) : degeneracy g → degeneracy (σ i ≫ g)

lemma le_of_degeneracy {n : ℕ} : Π {m : ℕ} (s : [m] ⟶ [n]),
  degeneracy s → n ≤ m
| n s degeneracy.id              := le_refl n
| m s (degeneracy.comp g i hg) := nat.le_succ_of_le (le_of_degeneracy g hg)

lemma degeneracy_comp_degeneracy {m n : ℕ} {f : [m] ⟶ [n]} (hf : degeneracy f) :
Π {l : ℕ} {g : [l] ⟶ [m]} (hg : degeneracy g), degeneracy (g ≫ f)
| m g degeneracy.id              := begin rw category.id_comp, exact hf, end
| l s (degeneracy.comp g i hg) :=
begin
  rw category.assoc (σ i) g f,
  exact degeneracy.comp _ _ (degeneracy_comp_degeneracy hg),
end

instance σ_split_epi {n} (i : fin (n+1)) :
  split_epi (σ i) := ⟨δ i.cast_succ, δ_comp_σ_self⟩

lemma split_epi_of_degeneracy {n : ℕ} : Π {m} {f : [m] ⟶ [n]},
  degeneracy f → split_epi f
| n f degeneracy.id              := ⟨𝟙 [n], category.id_comp (𝟙 [n])⟩
| m f (degeneracy.comp g i hg) :=
```

```
begin
  rcases split_epi_of_degeneracy hg with ⟨g_ret, g_comp⟩,
  rcases (infer_instance : split_epi (σ i)) with ⟨σ_ret, σ_comp⟩,
  refine ⟨g_ret ≫ σ_ret, _⟩,
  simp only [auto_param_eq] at *,
  rw [category.assoc, ←category.assoc σ_ret (σ i) g],
  rw [σ_comp, category.id_comp, g_comp],
end

@[reducible]
def bij {n m} (f : [n] ⟶ [m]) := function.bijective f.to_preorder_hom

/-- A bijective morphism is an isomorphism. -/
lemma iso_of_bijective {n m} (f : [n] ⟶ [m]) (hf : bij f) :
is_iso f :=
begin
  unfold bij at hf, split,
  rw function.bijective_iff_has_inverse at hf,
  rcases hf with ⟨g, hfg, hgf⟩,
  refine ⟨mk_hom ⟨g, _⟩, _⟩,
  {
    intros i j hij,
    rw le_iff_eq_or_lt at hij,
    cases hij with hij hij, rwa hij,
    by_contra hgij,
    push_neg at hgij,
    let H := f.to_preorder_hom.monotone (le_of_lt hgij),
    rw [hgf, hgf, ←not_lt] at H,
    exact H hij,
  },
  { split,
    ext1, ext1 i, simp,
    exact hfg i,
    ext1, ext1 i, simp,
    exact hgf i, }
end

/-- An isomorphism has same domain and codomain. -/
lemma auto_of_iso {n m} (f : [n] ⟶ [m]) [hf : is_iso f] : m = n :=
begin
  have h1 : fin(n+1) ≃ fin(m+1),
  { refine ⟨f.to_preorder_hom , (inv f).to_preorder_hom, _, _⟩,
    dsimp only [function.left_inverse],
    { intro i,
```

```
      suffices h : hom.to_preorder_hom (f ≫ inv f) i = i, simpa using h,
      rw [is_iso.hom_inv_id], simp, },
    { intro i,
      suffices h : hom.to_preorder_hom (inv f ≫ f) i = i, simpa using h,
      rw [is_iso.inv_hom_id], simp, }},
  have h : cardinal.mk (fin (n + 1)) = cardinal.mk (fin (m + 1)), from cardinal.eq_congr h1,
  rw [cardinal.mk_fin, cardinal.mk_fin] at h,
  norm_cast at h,
  exact (nat.succ.inj h).symm,
end

lemma id_le_iso {n} (f : [n] ⟶ [n]) [is_iso f] : ∀ i, i ≤ f.to_preorder_hom i :=
begin
  let func     := f.to_preorder_hom,
  let func_inv := (inv f).to_preorder_hom,
  intro i, apply i.induction_on, exact fin.zero_le _,
  intros j Hj,
  rw [←not_lt, ←fin.le_cast_succ_iff, not_le],
  suffices h : func  j.cast_succ ≠ func j.succ,
  exact gt_of_gt_of_ge (lt_of_le_of_ne (func.monotone (le_of_lt (fin.cast_succ_lt_succ j))) h) Hj,
  intro h,
  apply (lt_self_iff_false j.succ).mp,
  suffices h' : j.succ = j.cast_succ,
  calc j.succ = j.cast_succ : h'
          ... < j.succ : fin.cast_succ_lt_succ j,
  suffices h' : (f ≫ inv f).to_preorder_hom j.succ = (f ≫ inv f).to_preorder_hom j.cast_succ,
  rw [is_iso.hom_inv_id f] at h', simp at h', exact h',
  simp,
  exact congr_arg func_inv h.symm,
end

/-- Only automorphism is the identity. -/
lemma id_of_auto {n} (f : [n] ⟶ [n]) [is_iso f] : f = 𝟙 [n] :=
begin
  let func     := f.to_preorder_hom,
  let func_inv := (inv f).to_preorder_hom,
  ext1, apply le_antisymm,
  { have h : func.comp preorder_hom.id ≤ func.comp func_inv,
    from λ i, func.monotone (id_le_iso (inv f) i),
    change func ≤ (inv f ≫ f).to_preorder_hom at h,
```

```
    rw [is_iso.inv_hom_id f] at h,
    simpa using h,},
  { exact id_le_iso f, }
end

/-- An isomorphism is a face map. -/
lemma face_of_iso {n m} (f : [n] ⟶ [m]) [hf : is_iso f] : face f :=
begin
  tactic.unfreeze_local_instances,
  cases auto_of_iso f,
  rw @id_of_auto n f hf,
  exact face.id,
end

/-- An isomorphism is a degeneracy. -/
instance degeneracy_of_iso {n m} (f : [n] ⟶ [m]) [hf : is_iso f] : degeneracy f :=
begin
  tactic.unfreeze_local_instances,
  cases auto_of_iso f,
  rw @id_of_auto n f hf,
  exact degeneracy.id,
end

/-- A face automorphism is an isomorphism. -/
lemma iso_of_face_auto {n} : Π {m} (f : [n] ⟶ [m]), face f → n = m → is_iso f
| n f face.id h              := is_iso.id [n]
| m f (face.comp g i hg) h :=
  false.rec _ ((lt_self_iff_false n).mp (lt_of_lt_of_le
    (nat.lt_succ_of_le (le_of_face g hg)) (le_of_eq h.symm)))

/-- A degenerate automorphism is an isomorphism. -/
lemma iso_of_degeneracy_auto {n} : Π {m} (f : [m] ⟶ [n]), degeneracy f → n = m → is_iso f
| n f degeneracy.id h              := is_iso.id [n]
| m f (degeneracy.comp g i hg) h :=
  false.rec _ ((lt_self_iff_false n).mp (lt_of_lt_of_le
    (nat.lt_succ_of_le (le_of_degeneracy g hg)) (le_of_eq h.symm)))

lemma comp_σ_comp_δ {n m} (f: [n] ⟶ [m + 1]) (i : fin (m + 1))
(hi : ∀ j, f.to_preorder_hom j ≠ i.cast_succ) :
  f ≫ σ i ≫ δ i.cast_succ = f :=
begin
```

```
  ext1, ext1 j,
  simp [δ, σ, fin.succ_above, fin.pred_above],
  split_ifs with hij hji hji,
  { rw [←fin.succ_lt_succ_iff, fin.succ_pred, ←fin.le_cast_succ_iff,
   ↪  ←not_lt] at hji,
    exact absurd hij hji, },
  { rwa fin.succ_pred, },
  { rwa fin.cast_succ_cast_lt, },
  { push_neg at hij,
    rw [←fin.cast_succ_lt_cast_succ_iff, not_lt, fin.cast_succ_cast_lt]
     ↪  at hji,
    exact absurd (antisymm hij hji) (hi j),}
end

lemma comp_σ_comp_δ_succ {n m} (f: [n] ⟶ [m + 1]) (i : fin (m + 1))
(hi : ∀ j, f.to_preorder_hom j ≠ i.succ) :
  f ≫ σ i ≫ δ i.succ = f :=
begin
  ext1, ext1 j,
  simp [δ, σ, fin.succ_above, fin.pred_above],
  split_ifs with hij hji hji,
  { rw [←not_le, fin.le_cast_succ_iff, not_lt] at hij hji,
    rw fin.succ_pred at hji,
    exact absurd (antisymm hji hij) (hi j), },
  { rwa fin.succ_pred, },
  { rwa fin.cast_succ_cast_lt, },
  { push_neg at hij,
    rw [fin.cast_succ_cast_lt,←fin.le_cast_succ_iff] at hji,
    exact absurd hij hji, }
end

def inj {n m} (f : [n] ⟶ [m]) := function.injective f.to_preorder_hom

lemma comp_σ_injective {n m} (f: [n] ⟶ [m + 1]) (i : fin (m + 1))
(hi : ∀ j, f.to_preorder_hom j ≠ i.cast_succ) (hf : inj f):
  inj (f ≫ σ i) :=
begin
  intros j k hjk,
  apply hf,
  simp [σ, fin.pred_above] at hjk,
  split_ifs at hjk with hij hik hik,
  { exact fin.pred_inj.mp hjk, },
  { refine absurd (le_antisymm (not_lt.mp hik) _) (hi k),
    rw [←fin.cast_succ_inj, fin.cast_succ_cast_lt] at hjk,
```

```
    rwa [←hjk, fin.le_cast_succ_iff, fin.succ_pred], },
  { refine absurd (le_antisymm (not_lt.mp hij) _) (hi j),
    rw [←fin.cast_succ_inj, fin.cast_succ_cast_lt] at hjk,
    rwa [hjk, fin.le_cast_succ_iff, fin.succ_pred], },
  { ext, injections_and_clear, simp at h_1, exact h_1, },
end

lemma comp_σ_injective_succ {n m} (f: [n] ⟶ [m + 1]) (i : fin (m + 1))
(hi : ∀ j, f.to_preorder_hom j ≠ i.succ) (hf : inj f):
  inj (f ≫ σ i) :=
begin
  intros j k hjk,
  apply hf,
  simp [σ, fin.pred_above] at hjk,
  split_ifs at hjk with hij hik hik,
  { exact fin.pred_inj.mp hjk, },
  { refine absurd (le_antisymm _ (by rwa [←not_lt, ←fin.le_cast_succ_iff, not_le])) (hi j),
    rw [←fin.succ_inj, fin.succ_pred] at hjk,
    rw [hjk, ←not_lt, ←fin.le_cast_succ_iff, fin.cast_succ_cast_lt],
    rwa [not_le, ←fin.le_cast_succ_iff, ←not_lt],},
  { refine absurd (le_antisymm _ (by rwa [←not_lt, ←fin.le_cast_succ_iff, not_le])) (hi k),
    rw [←fin.succ_inj, fin.succ_pred] at hjk,
    rw [←hjk, ←not_lt, ←fin.le_cast_succ_iff, fin.cast_succ_cast_lt],
    rwa [not_le, ←fin.le_cast_succ_iff, ←not_lt], },
  { ext, injections_and_clear, simp at h_1, exact h_1, },
end

def fin.find_x {n : ℕ} (p : fin n → Prop) [decidable_pred p] (Hp : ∃ (i : fin n), p i) :
  {i // p i ∧ ∀ j, j < i → ¬p j} :=
begin
  let q : ℕ → Prop := λ i, ∃ (hi : i < n), p ⟨i, hi⟩,
  have Hq : ∃ (i : ℕ), q i,
  { cases Hp with i Hpi, cases i, exact ⟨i_val, i_property, Hpi⟩,},
  let i := nat.find Hq,
  have hi : i < n, cases (nat.find_spec Hq) with hi, exact hi,
  refine ⟨⟨i,hi⟩,_⟩,
  cases (nat.find_spec Hq) with hi hpi,
  split, exact hpi,
  intros j hj hpj,
  cases j,
  exact nat.find_min Hq hj ⟨j_property, hpj⟩,
```

```
end

/- An injective map is a face map. -/
lemma face_of_injective {n m} (f : [n] ⟶ [m]) (hf : inj f) : face f :=
begin
  induction m with m hm,
  { have Hf : function.surjective f.to_preorder_hom,
    { refine λ i, ⟨0,_⟩,
      rwa[(f.to_preorder_hom 0).eq_zero , i.eq_zero] },
    exact @face_of_iso _ _ f (iso_of_bijective f ⟨hf, Hf⟩) },
  by_cases Hf : function.surjective f.to_preorder_hom,
  { exact @face_of_iso _ _ f (iso_of_bijective f ⟨hf, Hf⟩), },
  { push_neg at Hf,
    let p := λ i, ∀ j, f.to_preorder_hom j ≠ i,
    let i := (fin.find_x p Hf).1,
    have hi, from (fin.find_x p Hf).2,
    cases hi with hi hi_min,
    clear hi_min,
    change ∀ j, f.to_preorder_hom j ≠ i at hi,
    by_cases Hi: i = 0,
    { have Hi : i.val < [m.succ].len, rw Hi, simp,
      let j := i.cast_lt Hi,
      rw ←fin.cast_succ_cast_lt i Hi at hi,
      let hfσ := hm (f ≫ σ j) (comp_σ_injective f j hi hf),
      rw [←comp_σ_comp_δ f j hi],
      exact face.comp (f ≫ σ j) j.cast_succ hfσ, },
    { let j := i.pred Hi,
      rw ←fin.succ_pred i Hi at hi,
      let hfσ := hm (f ≫ σ j) (comp_σ_injective_succ f j hi hf),
      rw [←comp_σ_comp_δ_succ f j hi],
      exact face.comp (f ≫ σ j) j.succ hfσ }},
end

/-- If f(i) = f(i+1) then σ i ≫ δ i+1 ≫ f = f -/
lemma σ_comp_δ_comp {n m} (f: [n + 1] ⟶ [m]) (i : fin (n + 1))
(H : f.to_preorder_hom i.cast_succ = f.to_preorder_hom i.succ) :
  σ i ≫ δ i.succ ≫ f = f :=
begin
  ext1, ext1 j,
  simp [δ, σ, fin.succ_above, fin.pred_above],
  split_ifs,
  { rw [←not_le, fin.le_cast_succ_iff, not_lt] at h h_1,
    rw fin.succ_pred at h_1,
    cases le_antisymm h_1 h,
```

```
    rwa fin.pred_succ, },
  { rw j.succ_pred,},
  { rw fin.cast_succ_cast_lt, },
  { rw [←fin.le_cast_succ_iff, fin.cast_succ_cast_lt, not_le] at h_1,
    exact absurd h_1 h,}
end

/-- Every map has a decompostion into a degeneracy and a face map. -/
theorem decomp_degeneracy_face {n m} (f : [n] ⟶ [m]) :
  ∃ {k} (s : [n] ⟶ [k]) [degeneracy s] (d : [k] ⟶ [m]) [face d], f = s ≫ d :=
begin
  induction n with n ih_n,
  { have hf : inj f, intros i j hij, rwa [i.eq_zero, j.eq_zero],
    exact ⟨0, 𝟙 [0], degeneracy.id, f, face_of_injective f hf, (category.id_comp f).symm⟩,},
  by_cases hf : function.injective f.to_preorder_hom,
  { exact ⟨n.succ, 𝟙 [n.succ], degeneracy.id, f, face_of_injective f hf, (category.id_comp f).symm⟩},
  { push_neg at hf,
    rcases hf with ⟨j₁, j₂, hfj, hj⟩,
    wlog j₁lj₂ : j₁ < j₂ := ne.lt_or_lt hj using j₁ j₂,
    let i := j₁.cast_pred,
    have hi : f.to_preorder_hom i.cast_succ = f.to_preorder_hom i.succ,
    { apply le_antisymm,
      exact f.to_preorder_hom.monotone (le_of_lt (fin.cast_succ_lt_succ i)),
      rw fin.cast_succ_cast_pred (lt_of_lt_of_le j₁lj₂ (fin.le_last j₂)),
      rw hfj,
      apply f.to_preorder_hom.monotone,
      rw [←not_lt, ←fin.le_cast_succ_iff, not_le],
      rwa fin.cast_succ_cast_pred (lt_of_lt_of_le j₁lj₂ (fin.le_last j₂)),},
    clear j₁lj₂ hfj hj j₂,
    let g := δ i.succ ≫ f,
    rcases ih_n g with ⟨k, s, hs, d, hd, hsd⟩,
    refine ⟨k, σ i ≫ s, degeneracy.comp s i hs, d, hd, _⟩,
    rw [category.assoc, ←hsd],
    exact (σ_comp_δ_comp f i hi).symm, }
end

end simplex_category
```

B. Lean code: Traversals

B.1. basic.lean

```
import algebraic_topology.simplicial_set
import category_theory.limits.has_limits
import category_theory.functor_category
import category_theory.limits.yoneda
import category_theory.limits.presheaf
import simplicial_sets.simplex_as_hom

open category_theory
open category_theory.limits
open simplex_category
open sSet
open_locale simplicial

/-!
# Traversals
Defines n-traversals, pointed n-traversals and their corresponding simplicial sets.

## Notations
* `+` for a plus,
* `-` for a minus,
* `e :: θ` for adding an edge e at the start of a traversal θ,
* `e · α` for the action of a map α on an edge e,
* `θ · α` for the action of a map α on a traversal θ.
-/

namespace traversal

@[derive decidable_eq]
inductive pm
| plus  : pm
| minus : pm

notation `±` := pm
notation `+` := pm.plus
```

```
notation `-` := pm.minus

@[reducible]
def edge (n : ℕ) := fin (n+1) × ±

def edge.lt {n} : edge n → edge n → Prop
| (i, -) (j, -) := i < j
| (i, -) (j, +) := true
| (i, +) (j, -) := false
| (i, +) (j, +) := j < i

instance {n} : has_lt (edge n) := ⟨edge.lt⟩

instance edge.has_decidable_lt {n} : Π e₁ e₂ : edge n, decidable (e₁ < e₂)
| (i, -) (j, -) := fin.decidable_lt i j
| (i, -) (j, +) := is_true trivial
| (i, +) (j, -) := is_false not_false
| (i, +) (j, +) := fin.decidable_lt j i

lemma edge.lt_asymm {n} : Π e₁ e₂ : edge n, e₁ < e₂ → e₂ < e₁ → false
| (i, -) (j, -) := nat.lt_asymm
| (i, -) (j, +) := λ h₁ h₂, h₂
| (i, +) (j, -) := λ h₁ h₂, h₁
| (i, +) (j, +) := nat.lt_asymm

instance {n} : is_asymm (edge n) edge.lt := ⟨edge.lt_asymm⟩

end traversal

@[reducible]
def traversal (n : ℕ) := list (traversal.edge n)

@[reducible]
def pointed_traversal (n : ℕ) := traversal n × traversal n

namespace traversal

notation h :: t  := list.cons h t
notation `⟦` l:(foldr `, ` (h t, list.cons h t) list.nil `⟧`) := (l : traversal _)

instance decidable_mem {n} :
```

```
  Π (e : edge n) (θ : traversal n), decidable (e ∈ θ) := list.decidable_mem


@[reducible]
def sorted {n} (θ : traversal n) := list.sorted edge.lt θ

theorem eq_of_sorted_of_same_elem {n : ℕ} : Π (θ₁ θ₂ : traversal n) (s₁ : sorted θ₁) (s₂ : sorted θ₂),
  (Π e, e ∈ θ₁ ↔ e ∈ θ₂) → θ₁ = θ₂
| [] [] := λ _ _ _, rfl
| [] (e₂ :: t₂) := λ _ _ H, begin exfalso, simpa using H e₂, end
| (e₁ :: t₁) [] := λ _ _ H, begin exfalso, simpa using H e₁, end
| (e₁ :: t₁) (e₂ :: t₂) := λ s₁ s₂ H,
begin
  simp only [sorted, list.sorted_cons] at s₁ s₂,
  cases s₁ with he₁ ht₁,
  cases s₂ with he₂ ht₂,
  have he₁e₂ : e₁ = e₂,
  { have He₁ := H e₁, simp at He₁, cases He₁ with heq He₁, from heq,
    have He₂ := H e₂, simp at He₂, cases He₂ with heq He₂, from heq.symm,
    exfalso, exact edge.lt_asymm e₁ e₂ (he₁ e₂ He₂) (he₂ e₁ He₁), },
  cases he₁e₂, simp,
  { apply eq_of_sorted_of_same_elem t₁ t₂ ht₁ ht₂,
    intro e, specialize H e, simp at H, split,
    { intro he,
      cases H.1 (or.intro_right _ he) with h, cases h,
      exfalso, exact edge.lt_asymm e₁ e₁ (he₁ e₁ he) (he₁ e₁ he),
      exact h, },
    { intro he,
      cases H.2 (or.intro_right _ he) with h, cases h,
      exfalso, exact edge.lt_asymm e₁ e₁ (he₂ e₁ he) (he₂ e₁ he),
      exact h, }}
end

theorem append_sorted {n : ℕ} : Π (θ₁ θ₂ : traversal n) (s₁ : sorted θ₁) (s₂ : sorted θ₂),
  (∀ (e₁ ∈ θ₁) (e₂ ∈ θ₂), e₁ < e₂) → sorted (θ₁ ++ θ₂)
| [] θ₂ := λ _ s₂ _, s₂
| (e₁ :: t₁) θ₂ := λ s₁ s₂ H,
begin
  simp only [sorted, list.sorted_cons] at s₁ s₂ ⊢,
  cases s₁ with he₁ ht₁,
  dsimp, rw list.sorted_cons,
```

```
  split,
  { intros e he, simp at he, cases he,
    exact he₁ e he,
    refine H e₁ (list.mem_cons_self e₁ t₁) e he },
  { apply append_sorted t₁ θ₂ ht₁ s₂,
    intros e₁' he₁' e₂' he₂',
    refine H e₁' (list.mem_cons_of_mem e₁ he₁') e₂' he₂' }
end

theorem append_sorted_iff {n : ℕ} : Π (θ₁ θ₂ : traversal n),
  sorted θ₁ ∧ sorted θ₂ ∧ (∀ (e₁ ∈ θ₁) (e₂ ∈ θ₂), e₁ < e₂) ↔ sorted (θ₁ ++ θ₂)
| ⟦⟧          θ₂ := by simp[sorted, list.sorted_nil]
| (e₁ :: t₁) θ₂ :=
begin
  split, rintro ⟨s₁, s₂, H⟩, apply append_sorted _ _ s₁ s₂ H,
  intro H, dsimp[sorted] at H, rw list.sorted_cons at H,
  change _ ∧ sorted _ at H, rw ←append_sorted_iff at H,
  split,
  { dsimp[sorted], rw list.sorted_cons, split,
    intros b hb, exact H.1 b (list.mem_append_left θ₂ hb),
    exact H.2.1 },
  split, exact H.2.2.1,
  intros e' he', simp at he', cases he', cases he',
  intros e₂ he₂, exact H.1 e₂ (list.mem_append_right t₁ he₂),
  exact H.2.2.2 e' he',
end

/-! # Applying a map to an edge -/

def apply_map_to_plus {n m : simplex_category} (i : fin (n.len+1)) (α : m ⟶ n) :
  Π (j : ℕ), j < m.len+1 → traversal m.len
| 0       h0  := if α.to_preorder_hom 0 = i then ⟦⟨0, +⟩⟧ else ⟦⟧
| (j + 1) hj  :=
  if α.to_preorder_hom ⟨j+1,hj⟩ = i
  then (⟨j+1, hj⟩, +) :: (apply_map_to_plus j (nat.lt_of_succ_lt hj))
  else apply_map_to_plus j (nat.lt_of_succ_lt hj)

lemma min_notin_apply_map_to_plus {n m : simplex_category} (α : m ⟶ n) (i : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  ∀ (k : fin (m.len + 1)), (k, -) ∉ apply_map_to_plus i α j hj :=
begin
  intros k hk,
```

```
  induction j with j,
  { simp [apply_map_to_plus] at hk,
    split_ifs at hk; simp at hk; exact hk },
  { simp [apply_map_to_plus] at hk,
    split_ifs at hk, simp at hk,
    repeat {exact j_ih _ hk }}
end

lemma plus_in_apply_map_to_plus_iff {n m : simplex_category} (α : m ⟶ n) (i : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  ∀ (k : fin (m.len + 1)), (k, +) ∈ apply_map_to_plus i α j hj ↔ k.val < j + 1 ∧ α.to_preorder_hom k = i :=
begin
  intros k,
  induction j with j,
  { simp only [apply_map_to_plus], split_ifs; simp, split,
    intro hk, cases hk, simp, exact h,
    intro hk, ext, simp, linarith,
    intro hk, replace hk : k = 0, ext, simp, linarith, cases hk, exact h, },
  { simp only [apply_map_to_plus], split_ifs; simp; rw j_ih; simp,
    split, intro hk, cases hk, cases hk, split, simp, exact h,
    split, exact nat.le_succ_of_le hk.1, exact hk.2,
    intro hk, rw hk.2, simp, cases nat.of_le_succ hk.1, right, exact h_1, left, ext, simp, exact nat.succ.inj h_1,
    intro hk, split, intro hkj, exact nat.le_succ_of_le hkj,
    intro hkj, cases nat.of_le_succ hkj, exact h_1,
    exfalso, have H : k = ⟨j + 1, hj⟩, ext, exact nat.succ.inj h_1, cases H, exact h hk, }
end

lemma apply_map_to_plus_sorted {n m : simplex_category} (α : m ⟶ n) (i : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  sorted (apply_map_to_plus i α j hj) :=
begin
  dsimp [sorted],
  induction j with j,
  { simp [apply_map_to_plus],
    split_ifs; simp, },
  { simp [apply_map_to_plus],
    split_ifs, swap, exact j_ih (nat.lt_of_succ_lt hj),
    simp only [list.sorted_cons], split, swap, exact j_ih (nat.lt_of_succ_lt hj),
    intros e he, cases e with k, cases e_snd,
```

```
    rw plus_in_apply_map_to_plus_iff at he, exact he.1,
    exact absurd he (min_notin_apply_map_to_plus α i j _ k), },
end

def apply_map_to_min {n m : simplex_category} (i : fin (n.len+1)) (α : m ⟶ n) :
Π (j : ℕ), j < m.len+1 → traversal m.len
| 0       h0  := if α.to_preorder_hom m.last = i then [⟨m.last, -⟩] else []
| (j + 1) hj  :=
  if α.to_preorder_hom ⟨m.len-(j+1), nat.sub_lt_succ _ _⟩ = i
  then (⟨m.len-(j+1), nat.sub_lt_succ _ _⟩, -) :: (apply_map_to_min j (nat.lt_of_succ_lt hj))
  else apply_map_to_min j (nat.lt_of_succ_lt hj)

lemma plus_notin_apply_map_to_min {n m : simplex_category} (α : m ⟶ n) (i : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  ∀ (k : fin (m.len + 1)), (k, +) ∉ apply_map_to_min i α j hj :=
begin
  intros k hk,
  induction j with j,
  { simp [apply_map_to_min] at hk,
    split_ifs at hk; simp at hk; exact hk },
  { simp [apply_map_to_min] at hk,
    split_ifs at hk, simp at hk,
    repeat {exact j_ih _ hk }}
end

lemma min_in_apply_map_to_min_iff {n m : simplex_category} (α : m ⟶ n) (i : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  ∀ (k : fin (m.len + 1)), (k, -) ∈ apply_map_to_min i α j hj ↔ k.val ⩾ m.len - j ∧ α.to_preorder_hom k = i :=
begin
  intros k,
  induction j with j,
  { simp only [apply_map_to_min], split_ifs; simp, split,
    intro hk, cases hk, simp, split, refl, exact h,
    intro hk, ext, exact le_antisymm (fin.le_last k) hk.1,
    intro hk, replace hk : k = m.last, ext, exact le_antisymm (fin.le_last k) hk,
    cases hk, exact h, },
  {
    have Hk : ∀ k, m.len - j.succ ⩽ k ↔ m.len - j ⩽ k ∨ m.len - j.succ = k,
```

```
    { have hmj_pos : 0 < m.len - j, from nat.sub_pos_of_lt
    ↪  (nat.succ_lt_succ_iff.mp hj),
      rw nat.lt_succ_iff at hj, intro k,
      rw [nat.sub_succ, ←nat.succ_le_succ_iff, ←nat.succ_inj',
      ↪  nat.succ_pred_eq_of_pos hmj_pos],
      exact nat.le_add_one_iff, },
    simp only [apply_map_to_min], split_ifs; simp; rw j_ih; simp,
    split, intro hk, cases hk, cases hk, split, simp, exact h,
    split, rw nat.sub_succ, exact nat.le_trans (nat.pred_le _) hk.1,
    ↪  exact hk.2,
    intro hk, rw hk.2, simp, cases (Hk k).mp hk.1, right, exact h_1,
    ↪  left, ext, exact h_1.symm,
    intro hk, rw Hk k, split, intro hkj, left, exact hkj,
    intro hkj, cases hkj, exact hkj,
    have Hk' : k = ⟨m.len - (j + 1), apply_map_to_min._main._proof_1 _⟩,
    ↪  ext, exact hkj.symm,
    cases Hk', exact absurd hk h,}
end

lemma apply_map_to_min_sorted {n m : simplex_category} (α : m ⟶ n) (i
↪  : fin (n.len+1)) (j : ℕ) (hj : j < m.len + 1) :
  list.sorted edge.lt (apply_map_to_min i α j hj) :=
begin
  induction j with j,
  { simp [apply_map_to_min],
    split_ifs; simp, },
  { simp [apply_map_to_min],
    split_ifs, swap, exact j_ih (nat.lt_of_succ_lt hj),
    simp only [list.sorted_cons], split, swap, exact j_ih
    ↪  (nat.lt_of_succ_lt hj),
    intros e he, cases e with k, cases e_snd,
    exact absurd he (plus_notin_apply_map_to_min α i j _ k),
    rw min_in_apply_map_to_min_iff at he,
    replace he : k.val ≥ m.len - j := he.1,
    change m.len - (j + 1) < k.val,
    refine lt_of_lt_of_le _ he, rw nat.sub_succ,
    refine nat.pred_lt _, simp,
    rwa [nat.sub_eq_zero_iff_le, not_le, ←nat.succ_lt_succ_iff], },
end

def apply_map_to_edge {n m : simplex_category} (α : m ⟶ n) : edge
↪  n.len → traversal m.len
| (i, +) := apply_map_to_plus i α m.last.1 m.last.2
| (i, -) := apply_map_to_min i α m.last.1 m.last.2
```

```
notation e · α := apply_map_to_edge α e

example (p : Prop) (h : p) : p ↔ true := iff_of_true h trivial

@[simp]
lemma edge_in_apply_map_to_edge_iff {n m : simplex_category} (α : m ⟶
↪ n) :
  ∀ (e₁ : edge m.len) (e₂), e₁ ∈ e₂ · α ↔ (α.to_preorder_hom e₁.1, e₁.2)
   ↪ = e₂ :=
begin
  intros e₁ e₂, cases e₁ with i₁ b₁, cases e₂ with i₂ b₂,
  cases b₁; cases b₂; simp [apply_map_to_edge],
  { simp [plus_in_apply_map_to_plus_iff],
    exact λ _, i₁.2, },
  { apply plus_notin_apply_map_to_min, },
  { apply min_notin_apply_map_to_plus, },
  { simp [min_in_apply_map_to_min_iff, simplex_category.last], },
end

lemma apply_map_to_edge_sorted {n m : simplex_category} (α : m ⟶ n) :
  ∀ (e : edge n.len), sorted (e · α)
| (i, +) := apply_map_to_plus_sorted α i _ _
| (i, -) := apply_map_to_min_sorted α i _ _

/-! # Applying a map to a traversal -/

def apply_map {n m : simplex_category} (α : m ⟶ n) : traversal n.len
↪  → traversal m.len
| ⟦⟧        := ⟦⟧
| (e :: t) := (e · α) ++ apply_map t

notation θ · α := apply_map α θ

@[simp]
lemma edge_in_apply_map_iff {n m : simplex_category} (α : m ⟶ n) (θ :
↪  traversal n.len) :
  ∀ (e : edge m.len), e ∈ θ · α ↔ (α.to_preorder_hom e.1, e.2) ∈ θ :=
begin
  intros e, induction θ;
  simp [apply_map, list.mem_append],
  simp [edge_in_apply_map_to_edge_iff, θ_ih],
end
```

```
def apply_map_preserves_sorted {n m : simplex_category} (α : m ⟶ n) (θ : traversal n.len) :
  sorted θ → sorted (θ · α) :=
begin
  intro sθ, induction θ; dsimp [sorted, apply_map],
  { exact list.sorted_nil },
  simp only [sorted, list.sorted_cons] at sθ,
  apply append_sorted,
  apply apply_map_to_edge_sorted,
  apply θ_ih sθ.2,
  intros e₁ he₁ e₂ he₂,
  rw edge_in_apply_map_to_edge_iff at he₁,
  rw edge_in_apply_map_iff at he₂,
  replace sθ := sθ.1 (α.to_preorder_hom e₂.fst, e₂.snd) he₂,
  cases he₁, clear he₁ he₂,
  cases e₁ with i₁ b₁, cases e₂ with i₂ b₂,
  cases b₁; cases b₂; simp [edge.lt] at sθ ⊢;
  try {change i₂ < i₁}; try {trivial}; try {change i₁ < i₂};
  rw ←not_le at sθ ⊢;
  exact λ H, sθ (α.to_preorder_hom.monotone H),
end

@[simp]
lemma apply_map_append {n m : simplex_category} (α : m ⟶ n) : Π (θ₁ θ₂ : traversal n.len),
  apply_map α (θ₁ ++ θ₂) = (apply_map α θ₁) ++ (apply_map α θ₂)
| ⟦⟧ θ₂          := rfl
| (h :: θ₁) θ₂ :=
begin
  dsimp[apply_map],
  rw apply_map_append,
  rw list.append_assoc,
end

@[simp]
lemma apply_id {n : simplex_category} : ∀ (θ : traversal n.len),
  apply_map (𝟙 n) θ = θ
| ⟦⟧          := rfl
| (e :: θ) :=
begin
  unfold apply_map,
  rw [apply_id θ], change _ = ⟦e⟧ ++ θ,
  rw list.append_left_inj,
  apply eq_of_sorted_of_same_elem,
```

```
  { apply apply_map_to_edge_sorted },
  { exact list.sorted_singleton e },
  { intro e, simp }
end

@[simp]
lemma apply_comp {n m l : simplex_category} (α : m ⟶ n) (β : n ⟶ l) :
  ∀ (θ : traversal l.len), apply_map (α ≫ β) θ = apply_map α (apply_map β θ)
| ⟦⟧       := rfl
| (e :: θ) :=
begin
  unfold apply_map,
  rw [apply_map_append, ←apply_comp, list.append_left_inj],
  apply eq_of_sorted_of_same_elem,
  { apply apply_map_to_edge_sorted },
  { apply apply_map_preserves_sorted,
    apply apply_map_to_edge_sorted },
  { intro e, simp, }
end

/-! # The application of the standard face maps and standard degeneracies. -/

@[simp] lemma apply_δ_self {n} (i : fin (n + 2)) (b : ±) :
  apply_map_to_edge (δ i) (i, b) = ⟦⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  apply apply_map_to_edge_sorted,
  exact list.sorted_nil,
  intro e, cases e, simp,
  intro h, exfalso,
  simp [δ, fin.succ_above] at h,
  split_ifs at h,
  finish,
  rw [not_lt, fin.le_cast_succ_iff] at h_1, finish,
end

@[simp] lemma apply_δ_succ_cast_succ {n} (i : fin (n + 1)) (b : ±) :
  apply_map_to_edge (δ i.succ) (i.cast_succ, b) = ⟦(i, b)⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  apply apply_map_to_edge_sorted,
```

```
  exact list.sorted_singleton (i, b),
  intro e, cases e, simp,
  intro hb, cases hb,
  split,
  { intro he,
    have H : (δ i.succ ≫ σ i).to_preorder_hom e_fst = (σ
    ↪  i).to_preorder_hom i.cast_succ,
    { rw ←he, simp, },
    rw δ_comp_σ_succ at H,
    simpa [σ, fin.pred_above] using H, },
  { intro he, cases he,
    simp [δ, fin.succ_above, fin.cast_succ_lt_succ], }
end

@[simp] lemma apply_δ_cast_succ_succ {n} (i : fin (n + 1)) (b : ±) :
  apply_map_to_edge (δ i.cast_succ) (i.succ, b) = ⟦(i, b)⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  apply apply_map_to_edge_sorted,
  exact list.sorted_singleton (i, b),
  intro e, cases e, simp,
  intro hb, cases hb,
  split,
  { intro he,
    have H : (δ i.cast_succ ≫ σ i).to_preorder_hom e_fst = (σ
    ↪  i).to_preorder_hom i.succ,
    { rw ←he, simp, },
    rw δ_comp_σ_self at H,
    simp [σ, fin.pred_above] at H,
    split_ifs at H, from H,
    exact absurd (fin.cast_succ_lt_succ i) h, },
  { intro he, cases he,
    simp [δ, fin.succ_above, fin.cast_succ_lt_succ], }
end

@[simp] lemma apply_σ_to_plus {n} (i : fin (n + 1)) :
  apply_map_to_edge (σ i) (i, +) = ⟦(i.succ, +), (i.cast_succ, +)⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  { apply apply_map_to_edge_sorted,},
  { simp [sorted], intros a b ha hb, rw ha, rw hb,
    exact fin.cast_succ_lt_succ i, },
  { intro e, cases e with l b,
    rw edge_in_apply_map_to_edge_iff,
```

```
    simp, rw ←or_and_distrib_right, simp, intro hb, clear hb b,
    simp [σ, fin.pred_above],
    split,
    { intro H, split_ifs at H,
      rw ←fin.succ_inj at H, simp at H,
      left, exact H,
      rw ←fin.cast_succ_inj at H, simp at H,
      right, exact H, },
    { intro H, cases H; rw H; simp[fin.cast_succ_lt_succ], }}
end

@[simp] lemma apply_σ_to_min {n} (i : fin (n + 1)) :
  apply_map_to_edge (σ i) (i, -) = ⟦(i.cast_succ, -), (i.succ, -)⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  { apply apply_map_to_edge_sorted, },
  { simp[sorted],
    intros a b ha hb, rw [ha, hb],
    exact fin.cast_succ_lt_succ i, },
  { intro e, cases e with l b,
    rw edge_in_apply_map_to_edge_iff,
    simp, rw ←or_and_distrib_right, simp, intro hb, clear hb b,
    simp [σ, fin.pred_above],
    split,
    { intro H, split_ifs at H,
      rw ←fin.succ_inj at H, simp at H,
      right, exact H,
      rw ←fin.cast_succ_inj at H, simp at H,
      left, exact H, },
    { intro H, cases H; rw H; simp[fin.cast_succ_lt_succ], }}
end

def edge.s {n} : edge n → fin (n+2)
| ⟨k, +⟩ := k.succ
| ⟨k, -⟩ := k.cast_succ

def edge.t {n} : edge n → fin (n+2)
| ⟨k, +⟩ := k.cast_succ
| ⟨k, -⟩ := k.succ

notation e`ˢ` := e.s
notation e`ᵗ` := e.t

lemma apply_σ_to_self {n} (e : edge n) :
```

```
  apply_map_to_edge (σ e.1) e = ⟦(eˢ, e.2), (eᵗ, e.2)⟧ :=
begin
  apply eq_of_sorted_of_same_elem,
  { apply apply_map_to_edge_sorted, },
  { dsimp [sorted],
    rw [list.sorted_cons],
    split, swap, apply list.sorted_singleton,
    intro e', simp, intro he', cases he',
    cases e with i b, cases b;
    exact fin.cast_succ_lt_succ i },
  { intro e', simp,
    cases e with i b, cases i with i hi,
    cases e' with i' b', cases i' with i' hi',
    cases b; cases b';
    simp [σ, fin.pred_above, edge.s, edge.t];
    split_ifs;
    try { rw ←fin.succ_inj, simp [h] };
    split; intro hi;
    cases hi;
    try { linarith };
    simp }
end

/- Simplicial set of traversals. -/
def 𝕋₀ : sSet :=
{ obj       := λ n, traversal n.unop.len,
  map       := λ x y α, apply_map α.unop,
  map_id'   := λ n, funext (λ θ, apply_id θ),
  map_comp' := λ l n m β α, funext (λ θ, apply_comp α.unop β.unop θ) }

lemma 𝕋₀_map_apply {n m : simplex_categoryᵒᵖ} {f : n ⟶ m} {θ : traversal n.unop.len} :
  𝕋₀.map f θ = θ.apply_map f.unop := rfl

/- Simplicial set of pointed traversals. -/
def 𝕋₁ : sSet :=
{ obj       := λ x, pointed_traversal x.unop.len,
  map       := λ _ _ α θ, (𝕋₀.map α θ.1, 𝕋₀.map α θ.2),
  map_id'   := λ _, by ext1 θ; simp,
  map_comp' := λ _ _ _ _ _, by ext1 θ; simp }

@[simp] lemma 𝕋₁_map_apply {n m : simplex_categoryᵒᵖ} {f : n ⟶ m} {θ₁ θ₂ : traversal n.unop.len} :
  𝕋₁.map f (θ₁, θ₂) = (𝕋₀.map f θ₁, 𝕋₀.map f θ₂) := rfl
```

```
@[simp] lemma 𝕋₁_map_apply_fst {n m : simplex_categoryᵒᵖ} {f : n ⟶ m} {θ : pointed_traversal n.unop.len} :
  (𝕋₁.map f θ).1 = 𝕋₀.map f θ.1 := rfl

@[simp] lemma 𝕋₁_map_apply_snd {n m : simplex_categoryᵒᵖ} {f : n ⟶ m} {θ : pointed_traversal n.unop.len} :
  (𝕋₁.map f θ).2 = 𝕋₀.map f θ.2 := rfl

def dom : 𝕋₁ ⟶ 𝕋₀ :=
{ app          := λ n θ, θ.2,
  naturality' := λ n m α, rfl }

def cod : 𝕋₁ ⟶ 𝕋₀ :=
{ app          := λ n θ, list.append θ.1 θ.2,
  naturality' := λ m m α, funext (λ θ, (traversal.apply_map_append α.unop θ.1 θ.2).symm) }

def as_hom {n} (θ : traversal n) : Δ[n] ⟶ 𝕋₀ := simplex_as_hom θ

end traversal

def pointed_traversal.as_hom {n} (θ : pointed_traversal n) :
  Δ[n] ⟶ traversal.𝕋₁ := simplex_as_hom θ
```

B.2. geom_real.lean

```
import traversals.basic

open category_theory
open category_theory.limits
open simplex_category
open sSet
open_locale simplicial

/-! # Geometric realisation of a traversal -/

namespace traversal

namespace geom_real

variables {n : ℕ} (θ : traversal n)

```

```
@[reducible]
def shape := fin(θ.length + 1) ⊕ fin(θ.length)

namespace shape

inductive hom : shape θ → shape θ → Type*
| id (X)                    : hom X X
| s  (i : fin(θ.length)) : hom (sum.inl i.cast_succ) (sum.inr i)
| t  (i : fin(θ.length)) : hom (sum.inl i.succ)      (sum.inr i)

instance category : small_category (shape θ) :=
{ hom := hom θ,
  id := λ j, hom.id j,
  comp := λ j₁ j₂ j₃ f g,
  begin
    cases f, exact g,
    cases g, exact hom.s f_1,
    cases g, exact hom.t f_1,
  end,
  id_comp' := λ j₁ j₂ f, rfl,
  comp_id' := λ j₁ j₂ f, by cases f; refl,
  assoc'   := λ j₁ j₂ j₃ j₄ f g h,  by cases f; cases g; refl,
}

end shape

def diagram : shape θ ⥤ sSet :=
{ obj := λ j, sum.cases_on j (λ j, Δ[n]) (λ j, Δ[n+1]),
  map := λ _ _ f,
  begin
    cases f with _ j j,
    exact 𝟙 _,
    exact to_sSet_hom (δ (list.nth_le θ j.1 j.2).s),
    exact to_sSet_hom (δ (list.nth_le θ j.1 j.2).t),
  end,
  map_id'   := λ j, rfl,
  map_comp' := λ _ _ _ f g, by cases f; cases g; refl, }

def colimit : colimit_cocone (diagram θ) :=
{ cocone := combine_cocones (diagram θ) (λ n,
  { cocone := types.colimit_cocone _,
    is_colimit := types.colimit_cocone_is_colimit _ }),
  is_colimit := combined_is_colimit _ _,
}
```

```

end geom_real

def geom_real {n} (θ : traversal n) : sSet := (geom_real.colimit θ).cocone.X

end traversal
```

B.3. geom_real_rec.lean

```
import traversals.basic
import category_theory.currying

open category_theory
open category_theory.limits
open simplex_category
open sSet
open_locale simplicial

namespace traversal

namespace geom_real_rec

variables {n : ℕ}

def sSet_colimit {sh : Type*} [small_category sh] (diag : sh ⥤ sSet) :
  colimit_cocone (diag) :=
{ cocone := combine_cocones (diag) (λ n,
  { cocone := types.colimit_cocone _,
    is_colimit := types.colimit_cocone_is_colimit _ }),
  is_colimit := combined_is_colimit _ _, }

def sSet_pushout {X Y Z : sSet} (f : X ⟶ Y) (g : X ⟶ Z) := sSet_colimit (span f g)

def bundle : Π (θ : traversal n), Σ (g : sSet), Δ[n] ⟶ g
| ⟦⟧       := ⟨Δ[n], 𝟙 _⟩
| (e :: θ) :=
  let colim := sSet_pushout (to_sSet_hom (δ e.t)) (bundle θ).2 in
  ⟨colim.cocone.X, to_sSet_hom (δ e.s) ≫ pushout_cocone.inl colim.cocone⟩

end geom_real_rec
```

```
def geom_real_rec {n} (θ : traversal n) : sSet := (geom_real_rec.bundle θ).1

namespace geom_real_rec
variables {n : ℕ}

def geom_real_incl (θ : traversal n) : Δ[n] ⟶ geom_real_rec θ := (geom_real_rec.bundle θ).2

def bundle_colim (e : edge n) (θ : traversal n) :=
  sSet_pushout (to_sSet_hom (δ e.t)) (bundle θ).2

@[simp]
lemma geom_real_rec_nil : geom_real_rec ([[]] : traversal n) = Δ[n] := rfl

@[simp]
lemma geom_real_incl_nil : geom_real_incl ([[]] : traversal n) = 𝟙 Δ[n] := rfl

lemma geom_real_rec_cons (e : edge n) (θ : traversal n) :
  geom_real_rec (e :: θ) = (bundle_colim e θ).cocone.X := rfl

lemma geom_real_incl_cons (e : edge n) (θ : traversal n) :
  geom_real_incl (e :: θ) = to_sSet_hom (δ e.s)
    ≫ pushout_cocone.inl (bundle_colim e θ).cocone := rfl

def j_rec_bundle : Π (θ : traversal n),
  {j : geom_real_rec θ ⟶ Δ[n] // geom_real_incl θ ≫ j = 𝟙 Δ[n]}
| [[]]        := ⟨𝟙 Δ[n], rfl⟩
| (e :: θ) :=
begin
  let j_θ := j_rec_bundle θ,
  refine ⟨(bundle_colim e θ).is_colimit.desc (pushout_cocone.mk (to_sSet_hom (σ e.1)) j_θ.1 _), _⟩,
  change _ = geom_real_incl θ ≫ j_θ.val, rw j_θ.2,
  swap,
  rw [geom_real_incl_cons, category.assoc],
  rw [(bundle_colim e θ).is_colimit.fac _ walking_span.left, pushout_cocone.mk_ι_app_left],
  all_goals
  { dsimp [to_sSet_hom],
    rw [←standard_simplex.map_comp, ←standard_simplex.map_id],
    apply congr_arg,
```

```
    cases e with i b, cases b;
    try { exact δ_comp_σ_self };
    try { exact δ_comp_σ_succ }},
end

def j_rec (θ : traversal n) : geom_real_rec θ ⟶ Δ[n] := (j_rec_bundle θ).1

def j_prop (θ : traversal n) : geom_real_incl θ ≫ j_rec θ = 𝟙 Δ[n] := (j_rec_bundle θ).2

def k_rec_bundle : Π (θ θ' : traversal n),
  {k : geom_real_rec θ ⟶ 𝕋₁ // geom_real_incl θ ≫ k = simplex_as_hom (θ', θ)}
| ⟦⟧        θ' := ⟨simplex_as_hom (θ',⟦⟧), rfl⟩
| (e :: θ) θ' :=
begin
  let k_θ := k_rec_bundle θ (θ' ++ ⟦e⟧),
  refine ⟨(bundle_colim e θ).is_colimit.desc (pushout_cocone.mk _ k_θ.1 _), _⟩,
  { apply simplex_as_hom,
    --Special position
    exact (apply_map (σ e.1) θ' ++ ⟦(eˢ, e.2)⟧, (eᵗ, e.2) :: apply_map (σ e.1) θ)},
  change _ = geom_real_incl θ ≫ k_θ.val, rw k_θ.2,
  swap,
  rw [geom_real_incl_cons, category.assoc],
  rw [(bundle_colim e θ).is_colimit.fac _ walking_span.left, pushout_cocone.mk_ι_app_left],
  all_goals
  { rw [hom_comp_simplex_as_hom],
    rw simplex_as_hom_eq_iff,
    cases e with i b, cases b; simp;
    rw [𝕋₀_map_apply, 𝕋₀_map_apply, has_hom.hom.unop_op];
    simp[edge.s, edge.t, apply_map];
    rw [←apply_comp, ←apply_comp];
    try { rw δ_comp_σ_self }; try { rw δ_comp_σ_succ };
    rw [apply_id, apply_id];
    simp },
end

def k_rec' (θ θ' : traversal n) := (k_rec_bundle θ θ').1

def k_prop' (θ θ' : traversal n) :
```

```
  geom_real_incl θ ≫ k_rec' θ θ' = simplex_as_hom (θ', θ) := (k_rec_bundle θ θ').2

def k_rec (θ : traversal n) : geom_real_rec θ ⟶ 𝕋₁ := (k_rec_bundle θ ⟦⟧).1

def k_prop (θ : traversal n) :
  geom_real_incl θ ≫ k_rec θ = simplex_as_hom (⟦⟧, θ) := (k_rec_bundle θ ⟦⟧).2

lemma j_comp_θ_eq_k_comp_cod : Π (θ θ' : traversal n),
  j_rec θ ≫ (θ' ++ θ).as_hom = (k_rec_bundle θ θ').1 ≫ cod
| ⟦⟧       θ' :=
  begin
    change simplex_as_hom _ = simplex_as_hom _ ≫ cod,
    rw [simplex_as_hom_comp_hom], refl,
  end
| (e :: θ) θ' :=
begin
  apply pushout_cocone.is_colimit.hom_ext (bundle_colim e θ).is_colimit,
  { change to_sSet_hom (σ e.fst) ≫ simplex_as_hom _ = simplex_as_hom _ ≫ cod,
    rw [simplex_as_hom_comp_hom, hom_comp_simplex_as_hom],
    rw simplex_as_hom_eq_iff,
    dsimp [𝕋₀, apply_map, cod], cases e with i b, cases b; simp,
    all_goals { simp [apply_map], change _ = _ ++ _, rw list.append_assoc, refl,}},
  { change j_rec θ ≫ (θ' ++ (⟦e⟧ ++ θ)).as_hom = (k_rec_bundle θ (θ' ++ ⟦e⟧)).1 ≫ cod,
    rw ←list.append_assoc,
    apply j_comp_θ_eq_k_comp_cod }
end

def pullback_cone_rec' (θ θ' : traversal n) : pullback_cone ((θ' ++ θ).as_hom) cod :=
  pullback_cone.mk (j_rec θ) (k_rec_bundle θ θ').1 (j_comp_θ_eq_k_comp_cod θ θ')

def pullback_cone_rec (θ : traversal n) : pullback_cone (θ.as_hom) cod :=
  pullback_cone.mk (j_rec θ) (k_rec θ) (j_comp_θ_eq_k_comp_cod θ ⟦⟧)

def append_eq_append_split {n} {a b c d : traversal n} :
```

```
  a ++ b = c ++ d → {a' // c = a ++ a' ∧ b = a' ++ d} ⊕ {c' // a = c
  ↪  ++ c' ∧ d = c' ++ b} :=
begin
  induction c generalizing a,
  case nil { rw list.nil_append, rintro rfl, right, exact ⟨a, rfl, rfl⟩
  ↪  },
  case cons : c cs ih {
    intro h, cases a,
    { left, use c :: cs, simpa using h },
    { simp at h, cases h, cases h_left, simp,
      exact ih h_right }}
end

section beta

variables {m : simplex_category} {α : m ⟶ [n]} {e : edge n} {θ₁ θ₂ :
↪  traversal m.len} (H : apply_map_to_edge α e = θ₁ ++ θ₂)

def β_conditions (i) (H : apply_map_to_edge α e = θ₁ ++ θ₂) :
  α.to_preorder_hom i < e.1 ∨ α.to_preorder_hom i > e.1 ∨ (i, e.2) ∈ θ₁
  ↪  ∨ (i, e.2) ∈ θ₂ :=
begin
  cases lt_trichotomy (α.to_preorder_hom i) e.fst, left, exact h,
  cases h, swap, right, left, exact h,
  right, right,
  replace h : (α.to_preorder_hom (i, e.2).1, (i, e.2).2) = e, rw h,
  ↪  cases e, refl,
  rw [←edge_in_apply_map_to_edge_iff, H] at h, simp at h, exact h,
end

def β_fun (H : apply_map_to_edge α e = θ₁ ++ θ₂) : fin (m.len + 1) →
↪  fin ([n + 1].len + 1) := λ i,
  if h₁ : α.to_preorder_hom i < e.1 then (α.to_preorder_hom i).cast_succ
  else if h₂ : α.to_preorder_hom i > e.1 then (α.to_preorder_hom i).succ
  else if h₃ : (i, e.2) ∈ θ₁ then eˢ
  else eᵗ

lemma β_eq_es_iff (i) : β_fun H i = (eˢ) ↔ (i, e.2) ∈ θ₁ :=
begin
  simp[β_fun],
  split;
  intro h',
  { by_contra, split_ifs at h',
    rw [←fin.cast_succ_lt_cast_succ_iff, h'] at h_1,
```

```
    cases e with j b, cases b,
    exact lt_asymm (fin.cast_succ_lt_succ j) h_1,
    exact lt_irrefl _ h_1,
    rw [←fin.succ_lt_succ_iff, h'] at h_2,
    cases e with j b, cases b,
    exact lt_irrefl _ h_2,
    exact lt_asymm (fin.cast_succ_lt_succ j) h_2,
    cases e with j b, cases b,
    exact ne_of_lt (fin.cast_succ_lt_succ j) h',
    exact ne_of_gt (fin.cast_succ_lt_succ j) h' },
  { have hi : (i, e.2) ∈ apply_map_to_edge α e, rw H, exact list.mem_append_left θ₂ h',
    rw edge_in_apply_map_to_edge_iff at hi,
    cases e with j b, cases b;
    simp at hi h'; simp[hi, h'] }
end

lemma β_eq_et_iff (i) : β_fun H i = (eᵗ) ↔ (i, e.2) ∈ θ₂ :=
begin
  simp[β_fun],
  have hf: e.s = e.t ↔ false,
  { split; intro hf, cases e with j b, cases b,
    exact ne_of_lt (fin.cast_succ_lt_succ j) hf.symm,
    exact ne_of_lt (fin.cast_succ_lt_succ j) hf,
    exfalso, exact hf },
  split; intro h',
  { by_contra, split_ifs at h',
    rw [←fin.cast_succ_lt_cast_succ_iff, h'] at h_1,
    cases e with j b, cases b,
    exact lt_irrefl _ h_1,
    exact lt_asymm (fin.cast_succ_lt_succ j) h_1,
    rw [←fin.succ_lt_succ_iff, h'] at h_2,
    cases e with j b, cases b,
    exact lt_asymm (fin.cast_succ_lt_succ j) h_2,
    exact lt_irrefl _ h_2,
    cases e with j b, cases b,
    exact ne_of_gt (fin.cast_succ_lt_succ j) h',
    exact ne_of_lt (fin.cast_succ_lt_succ j) h',
    cases β_conditions i H, exact h_1 h_4,
    cases h_4, exact h_2 h_4,
    cases h_4, exact h_3 h_4,
    exact h h_4 },
  { have hi : (α.to_preorder_hom (i, e.2).1, (i, e.2).2) = e,
```

```
    { rw [←edge_in_apply_map_to_edge_iff, H], exact list.mem_append_right θ₁ h', },
    cases e; simp at hi ⊢ h', simp [hi],
    have H' : sorted (θ₁ ++ θ₂), rw ←H, apply apply_map_to_edge_sorted,
    rw ←append_sorted_iff at H',
    intro hi', exfalso, have h'' := (H'.2.2 _ hi' _ h'),
    exact edge.lt_asymm _ _ h'' h'', }
end

lemma β_monotone : monotone (β_fun H) := λ i j hij,
begin
  simp [β_fun], split_ifs; try { apply le_refl },
  { exact α.to_preorder_hom.monotone hij },
  { apply le_of_lt, rw ←fin.le_cast_succ_iff, exact α.to_preorder_hom.monotone hij },
  { rw ←fin.cast_succ_lt_cast_succ_iff at h, cases e with j b, cases b,
    exact le_trans (le_of_lt h) (le_of_lt (fin.cast_succ_lt_succ _)),
    exact le_of_lt h },
  { rw ←fin.cast_succ_lt_cast_succ_iff at h, cases e with j b, cases b,
    exact le_of_lt h,
    exact le_trans (le_of_lt h) (le_of_lt (fin.cast_succ_lt_succ _)) },
  { refine absurd (α.to_preorder_hom.monotone hij) (not_le.mpr _),
    exact lt_of_lt_of_le h_2 (not_lt.mp h) },
  { simp, exact α.to_preorder_hom.monotone hij },
  { refine absurd (α.to_preorder_hom.monotone hij) (not_le.mpr _),
    exact lt_of_le_of_lt (not_lt.mp h_3) h_1 },
  { refine absurd (α.to_preorder_hom.monotone hij) (not_le.mpr _),
    exact lt_of_le_of_lt (not_lt.mp h_3) h_1 },
  all_goals { have hi := le_antisymm (not_lt.mp h) (not_lt.mp h_1) },
  { refine absurd (α.to_preorder_hom.monotone hij) (not_le.mpr _),
    rwa hi at h_3 },
  { cases e with k b, cases b; simp [edge.s, edge.t],
    exact le_of_lt h_4,
    apply le_of_lt, rw[←fin.le_cast_succ_iff], simp, exact le_of_lt h_4 },
  swap 3, swap 3,
  { refine absurd (α.to_preorder_hom.monotone hij) (not_le.mpr _),
    rwa hi at h_3 },
  { cases e with k b, cases b; simp [edge.s, edge.t],
    apply le_of_lt, rw[←fin.le_cast_succ_iff], simp, exact le_of_lt h_4,
    exact le_of_lt h_4 },
  all_goals
  { cases e with k b, cases b; dsimp[edge.s, edge.t];
```

```
    try { exact le_of_lt (fin.cast_succ_lt_succ k) },
    exfalso, have hj := le_antisymm (not_lt.mp h_4) (not_lt.mp h_3),
    have H' : sorted (θ₁ ++ θ₂), rw ←H, apply apply_map_to_edge_sorted,
    rw ←append_sorted_iff at H', },
  { have hj' : (α.to_preorder_hom (j, +).1, (j, +).2) = (k, +), simp,
  ↪  exact hj,
    rw [←edge_in_apply_map_to_edge_iff, H] at hj',
    simp [h_5] at hj' h_2,
    refine absurd hij (not_le.mpr _),
    exact H'.2.2 _ h_2 _ hj', },
  { have hi' : (α.to_preorder_hom (i, -).1, (i, -).2) = (k, -), simp,
  ↪  exact hi.symm,
    rw [←edge_in_apply_map_to_edge_iff, H] at hi',
    simp [h_2] at hi' h_5,
    refine absurd hij (not_le.mpr _),
    exact H'.2.2 _ h_5 _ hi', },
end

def β (H : apply_map_to_edge α e = θ₁ ++ θ₂) : m ⟶ [n+1] := hom.mk
{ to_fun    := β_fun H,
  monotone' := β_monotone H, }

lemma β_comp_σ : β H ≫ σ e.1 = α :=
begin
  ext1, ext1 i, simp [β, β_fun], split_ifs;
  simp [σ, fin.pred_above]; split_ifs; try { refl };
  try { push_neg at * },
  { exfalso, exact lt_asymm h h_1 },
  { exfalso, rw ←fin.le_cast_succ_iff at h_2, simp at h_2, exact h h_2
  ↪  },
  { push_neg at h h_1, rw le_antisymm h_1 h, cases e with j b, cases b;
    simp[edge.s, edge.t] at h_3 ⊢,
    exfalso, exact h_3 },
  { push_neg at h h_1 h_3, rw le_antisymm h_1 h, cases e with j b, cases
  ↪  b;
    simp[edge.s, edge.t] at h_3 ⊢,
    exact absurd (fin.cast_succ_lt_succ j) (not_lt.mpr h_3) },
  { push_neg at h h_1, rw le_antisymm h_1 h, cases e with j b, cases b;
    simp[edge.s, edge.t] at h_3 ⊢,
    exfalso, exact h_3 },
  { push_neg at h h_1 h_3, rw le_antisymm h_1 h, cases e with j b, cases
  ↪  b;
    simp[edge.s, edge.t] at h_3 ⊢,
    exact absurd (fin.cast_succ_lt_succ j) (not_lt.mpr h_3) },
```

```
end

end beta


def geom_real_rec_lift' : Π (θ θ' : traversal n) {m} (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (geom_real_rec θ).obj (opposite.op m)
| ⟦⟧       θ' m α θ₁ θ₂ hθ := α
| (e :: θ) θ' m α θ₁ θ₂ hθ :=
  begin
    cases append_eq_append_split hθ with a' c',
    { rcases a' with ⟨θ₂', hθ₂', hθ₂⟩,
      let p : pushout_cocone _ _ := (bundle_colim e θ).cocone,
      apply p.inl.app (opposite.op m),
      exact β hθ₂',
    },
    { cases c' with c' hc',
      let p : pushout_cocone _ _ := (bundle_colim e θ).cocone,
      exact p.inr.app (opposite.op m) (geom_real_rec_lift' θ (θ' ++ ⟦e⟧) α c' θ₂ hc'.2.symm) },
  end

lemma geom_real_rec_fac_j' : Π (θ θ' : traversal n) {m} (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (j_rec θ).app (opposite.op m) (geom_real_rec_lift' θ θ' α θ₁ θ₂ hθ) = α
| ⟦⟧       θ' m α θ₁ θ₂ hθ := rfl
| (e :: θ) θ' m α θ₁ θ₂ hθ :=
  begin
    simp [geom_real_rec_lift'],
    cases append_eq_append_split hθ with a' c',
    { rcases a' with ⟨θ₂', H, hθ₂⟩,
      cases e with i b, simp,
      exact β_comp_σ H },
    { cases c' with c' hc',
      exact geom_real_rec_fac_j' θ (θ' ++ ⟦e⟧) α c' θ₂ _ }
  end

lemma geom_real_rec_fac_k' : Π (θ θ' : traversal n) {m} (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (k_rec_bundle θ θ').1.app (opposite.op m) (geom_real_rec_lift' θ θ' α θ₁ θ₂ hθ) = ((apply_map α θ') ++ θ₁, θ₂)
| ⟦⟧       θ' m α θ₁ θ₂ hθ :=
```

```
  begin
    simp [apply_map] at hθ, cases hθ.1, cases hθ.2,
    simp[geom_real_rec_lift'], refl
  end
| (e :: θ) θ' m α θ₁ θ₂ hθ :=
  begin
    simp [geom_real_rec_lift'],
    cases append_eq_append_split hθ with a' c',
    { rcases a' with ⟨θ₂', H, hθ₂⟩,
      change (simplex_as_hom _).app (opposite.op m) (β H) = _,
      simp [simplex_as_hom],
      change apply_map _ _  = _ ∧ apply_map _ _  = _ ,
      rw [apply_map_append],
      simp [apply_map],
      rw [←apply_comp, ←apply_comp, β_comp_σ H],
      cases hθ₂,
      change _ ∧ _ = θ₂' ++ _,
      rw [list.append_left_inj],
      rw [list.append_right_inj],
      have h₁ : sorted (θ₁ ++ θ₂'), rw ←H, apply
       ↪  apply_map_to_edge_sorted,
      have h₂ := h₁, rw ←append_sorted_iff at h₂,
      split;
      refine eq_of_sorted_of_same_elem _ _ (apply_map_to_edge_sorted _
       ↪  _) (by simp[h₂.1, h₂.2]) _;
      intro e'; rw edge_in_apply_map_to_edge_iff; simp; split; simp [β];
      try {rw β_eq_es_iff H}; try {rw β_eq_et_iff H}; intro h,
      { intro h', rw ←h' at h, simpa using h },
      { have he' : e' ∈ apply_map_to_edge α e, rw H, exact
       ↪  list.mem_append_left θ₂' h,
        rw edge_in_apply_map_to_edge_iff at he', cases e, cases e', simp
         ↪  at he' ⊢,
        cases he'.2, simp, exact h },
      { intro h', rw ←h' at h, simpa using h },
      { have he' : e' ∈ apply_map_to_edge α e, rw H, exact
       ↪  list.mem_append_right θ₁ h,
        rw edge_in_apply_map_to_edge_iff at he', cases e, cases e', simp
         ↪  at he' ⊢,
        cases he'.2, simp, exact h }},
    { cases c' with c' hc', simp,
      dsimp [k_rec, k_rec_bundle],
      change (k_rec_bundle θ (θ' ++ ⟦e⟧)).1.app (opposite.op m)
       ↪  (geom_real_rec_lift' θ (θ' ++ ⟦e⟧) α c' θ₂ _) = _,
      cases hc'.1,
```

```
      specialize geom_real_rec_fac_k' θ (θ' ++ ⟦e⟧) α c' θ₂ hc'.2.symm,
      rw [geom_real_rec_fac_k', apply_map_append], simp [apply_map], refl, }
  end

def geom_real_rec_lift (θ : traversal n) {m} : Π (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (geom_real_rec θ).obj (opposite.op m) := geom_real_rec_lift' θ ⟦⟧

lemma geom_real_rec_fac_j (θ : traversal n) {m} : Π (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (j_rec θ).app (opposite.op m) (geom_real_rec_lift θ α θ₁ θ₂ hθ) = α := geom_real_rec_fac_j' θ ⟦⟧

lemma geom_real_rec_fac_k (θ : traversal n) {m} : Π (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ),
  (k_rec θ).app (opposite.op m) (geom_real_rec_lift θ α θ₁ θ₂ hθ) = (θ₁, θ₂) := geom_real_rec_fac_k' θ ⟦⟧

lemma geom_real_rec_unique : Π (θ : traversal n) {m} (α : m ⟶ [n]) (θ₁ θ₂ : traversal m.len) (hθ : θ₁ ++ θ₂ = apply_map α θ)
  (x : (geom_real_rec θ).obj (opposite.op m)),
  (j_rec θ).app (opposite.op m) x = α →
  (k_rec θ).app (opposite.op m) x = (θ₁, θ₂) →
  x = geom_real_rec_lift θ α θ₁ θ₂ hθ
| ⟦⟧ m α θ₁ θ₂ hθ x hx₁ hx₂ := by dsimp [geom_real_rec_lift]; rw ←hx₁; refl
| (e :: θ) m α ⟦⟧         θ₂ hθ x hx₁ hx₂ := sorry
| (e :: θ) m α (e₁ :: θ₁) θ₂ hθ x hx₁ hx₂ := sorry

theorem geom_real_is_pullback_θ_cod (θ : traversal n) : is_limit (pullback_cone_rec θ) :=
begin
  apply evaluation_jointly_reflects_limits,
  intro m, exact
  { lift := λ c,
    begin
      let c_fst : c.X ⟶ Δ[n].obj m := c.π.app walking_cospan.left,
      let c_snd : c.X ⟶ 𝕋₁.obj m   := c.π.app walking_cospan.right,
      have hθ : c_fst ≫ (as_hom θ).app m = c_snd ≫ cod.app m,
      { change c_fst ≫ (cospan θ.as_hom cod ⋙ (evaluation simplex_categoryᵒᵖ Type).obj m).map walking_cospan.hom.inl
          = c_snd ≫ (cospan θ.as_hom cod ⋙ (evaluation simplex_categoryᵒᵖ Type).obj m).map walking_cospan.hom.inr,
```

```
      rw [←c.π.naturality, ←c.π.naturality], refl },
    exact λ x, geom_real_rec_lift θ (c_fst x) _ _ (congr_fun hθ x).symm,
  end,
  fac' := λ c,
  begin
    intro j, cases j;
    let c_fst : c.X ⟶ Δ[n].obj m := c.π.app walking_cospan.left;
    let c_snd : c.X ⟶ pointed_traversal m.unop.len := c.π.app walking_cospan.right,

    { let p := pullback_cone_rec θ,
      let const_c := (category_theory.functor.const walking_cospan).obj c.X,
      let const_c_inl := const_c.map walking_cospan.hom.inl,
      let const_p := (category_theory.functor.const walking_cospan).obj p.X,
      let const_p_inl := const_p.map walking_cospan.hom.inl,
      let eval_cospan := cospan θ.as_hom cod ⋙ (evaluation simplex_categoryᵒᵖ Type).obj m,
      change _ = const_c_inl ≫ c.π.app none,
      have H : const_c_inl ≫ c.π.app none = _, apply c.π.naturality',
      refine trans _ H.symm, clear H,
      suffices H : ((pullback_cone_rec θ).π.app none).app m
        = ((pullback_cone_rec θ).π.app walking_cospan.left).app m
        ≫ eval_cospan.map walking_cospan.hom.inl,
      { simp, rw H, ext1 x,
        let α : m.unop ⟶ [n] := c_fst x,
        simp, apply congr_arg,
        exact geom_real_rec_fac_j θ α _ _ _ },
      change (const_p_inl ≫ (pullback_cone_rec θ).π.app none).app m = _,
      rw p.π.naturality', refl },
    ext1 x, let α : m.unop ⟶ [n] := c_fst x,
    cases j, simp,
    { exact geom_real_rec_fac_j θ α _ _ _ },
    { change (k_rec θ).app (opposite.op m.unop) (geom_real_rec_lift θ α (c_snd x).1 (c_snd x).2 _) = c_snd x,
      rw [geom_real_rec_fac_k θ α (c_snd x).1 (c_snd x).2 _], simp }
  end,
  uniq' := λ c,
  begin
    intros lift' hlift',
```

```
      change c.X ⟶ (geom_real_rec θ).obj (opposite.op m.unop) at
      ↪  lift',
      ext1 x, simp,
      apply geom_real_rec_unique θ,
      specialize hlift' walking_cospan.left, simp at hlift',
      rw ←hlift', refl,
      specialize hlift' walking_cospan.right, simp at hlift',
      rw ←hlift',
      simp, refl,
    end }
end

end geom_real_rec

end traversal
```