Formalization of Non-Abelian Topology for Homotopy Type Theory

Master Thesis

Author: Jakob von Raumer

Supervisors at CMU: Prof. Jeremy Avigad
Prof. Steve Awodey

Supervisors at KIT: Prof. Dr. Gregor Snelting
Prof. Dr. Frank Herrlich

May 28, 2015

Abstract | Kurzfassung

In this thesis, I present a translation of some essential algebraic structures from non-abelian algebraic topology to a setting of homotopy type theory and an application of these structures in the form of a fundamental double groupoid of a 2-truncated type which is presented by a set and a 1-type. I furthermore describe how I formalized parts of these definitions and theorems in the newly developed interactive theorem prover Lean.

In dieser Arbeit präsentiere ich eine Übersetzung einiger essentieller algebraischer Strukturen der nichtabelschen algebraischen Topologie in Homotopietypentheorie und eine Anwendung dieser Strukturen in Gestalt eines fundamentalen Doppelgruppoids eines 2-abgestumpften Typs, welcher durch eine Menge und einen 1-Typen dargestellt ist. Desweiteren beschreibe ich, wie ich Teile dieser Definitionen und Sätze in dem neu entwickelten interaktiven Theorembeweiser Lean formalisiert habe.

I declare that I have developed and written the enclosed thesis by myself, and have not used sources or means without declaration in the text.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Jakob von Raumer

Acknowledgements

First and foremost, I would like to express my deep gratitude to Prof. Jeremy Avigad, who has been an excellent advisor for this thesis and has been a great help in every stage of this thesis project. He supported me in finding a topic, and gave lots of helpful advice for the formalization as well as for the write up.

Also, I want to thank Prof. Steve Awodey for giving me the idea to formalize structures from non-abelian topology and who gave me useful advice on what formalization goals to pursue.

Another great help was to be included in the reading group on non-abelian algebraic topology whose other members Ulrik Buchholtz, Kuen-Bang Hou (Favonia), Ed Morehouse, Clive Newstead, Egbert Rijke, and Bas Spitters I also want to thank for helping me to understand the matter.

Another big thank you goes out to Floris van Doorn who was also working with the Lean theorem prover during my stay in Pittsburgh and who has always been a great help and support whenever I ran into problems and needed fast advice.

As the developer of the Lean theorem prover, Leonardo de Moura has also provided the best support imaginable. Most of the times, he gave answers to issues with Lean or bug fixes in just a few minutes. I furthermore have to thank Soonho Kong for providing good support for his Lean user interface and for helping me to set up the environment for the code listings in this thesis.

For their contribution to make my stay in Pittsburgh possible, I want to thank Prof. Dr. Gregor Snelting for supervising my thesis from Karlsruhe, Prof. Dr. Frank Herrlich for allowing the thesis to count for my mathematics major, and Prof. Dr. Alexander Waibel and the interACT exchange program for financing my stay at Carnegie Mellon University. Last but not least, the thesis project would not have happened without my friend Felix Maurer pointing me to the videos of Prof. Robert Harper's homotopy type theory course and the latter redirecting me to Jeremy Avigad and Steve Awodey.

Contents

Chapter 1

Introduction

Making mathematical definitions and theorem proofs readable and verifiable by computers has become increasingly important in the last years, not only since there are proofs that are hard or impossible to be checked by a single person due to their size (one example being Tom Hales' proof of the Kepler conjecture [HAB+15]). With the rise of formally verified software, one also wants the same level of trust for the mathematical theories whose soundness guarantee the correct functionality of the program. Fields where formal verification has been successfully used to certify computer programs include cryptography and aerospace industry. These rely heavily on results from algebra and calculus and differential equations.

Homotopy type theory (HoTT) can serve as a foundation of mathematics that is better suited to fit the needs of formalizing certain branches of mathematics, especially the ones of *topology*. In traditional, set-based approaches to formalizing the world of mathematical knowledge, topological spaces and their properties have to be modeled with much effort by referring to the type of real numbers. In contrast to this, homotopy type theory contains topologically motivated objects like fibrations and homotopy types as primitives. This makes it much easier and more natural to reason about topological properties of these objects. Homotopy type theory is a relatively new field but it already has produced several useful implementations and libraries in interactive theorem provers like Agda [hotb] and Coq [hota]. One important feature of homotopy type theory is that it is *constructive* and thus allows to extract programs from definitions and proofs.

Homotopy type theory is *proof relevant* which means that there can be distinct (and internally distinguishable) proofs for one statement. This leads to the fact that types in HoTT bear the structure of a higher groupoid in their identities. The essential problem in the field of *homotopy* is to analyze this structure of paths and iterated paths between paths in topological spaces or, in the world of HoTT, in higher types. This happens by considering the algebraic properties of the homotopy groups or *homotopy groupoids* of the spaces resp. types.

In his book "Nonabelian Algebraic Topology" [BHS11], Ronald Brown introduces the notion of *double groupoids with thin structures* and *crossed modules over groupoids* to describe the interaction between the first and the second homotopy groupoid of a space algebraically. Brown's approach, preceding the discovery of homotopy type theory by a few decades, is formulated entirely classically and set-based.

In this thesis I describe how I translated some of the central definitions and lemmas from his book to dependently typed algebraic structures in homotopy type theory, made them applicable to the analysis of 2-truncated types by creating the notion of a *fundamental double groupoid of a presented 2-type*, and then formalized them in the newly built interactive theorem proving system Lean [dMKA$^+$].

The structure of this thesis is as follows: Chapter 2 gives a short introduction to some basics of homotopy type theory. Chapter 3 summarizes the considered categories as they are presented in Ronald Brown's book. Then, Chapter 4 describes, how we can translate these definitions to the setting of homotopy type theory. Eventually, Chapter 5 tells my experiences in formalizing the definitions in Lean.

Chapter 2

Homotopy Type Theory

This chapter shall serve to provide the reader with the necessary basic knowledge about homotopy type theory. Most of this knowledge was gathered and written up during the "special year on univalent foundations" which took place in the years 2012 and 2013 at the Institute for Advanced Study in Princeton. It resulted in the collaborative effort to write and publish a first *book* on homotopy type theory [Uni13] which is still being improved and open for suggestions at GitHub. [1] My description of homotopy type theory will stick to the notation and terminology used in this book.

Furthermore, I will not make the effort to distinguish between what elements of the theory were there in earlier approaches to intensional type theory, most prominently the one of Per Martin-Löf [ML98], as this would defy the purpose of a concise and coherent introduction to the current "state of the art" of homotopy type theory.

2.1 Some Basic Non-Dependent Type Theory

A type theoretical foundation of mathematics uses **Types** wherever, in an approach based on set theory and predicate logic, sets and propositions are used. Homotopy type theory adds to this logical interpretation and set interpretation the point of view of a topological space or its homotopy type. **Objects** (or **instances**) of a type thus correspond to elements of a set, to proofs of a proposition, as well as to points in a space.

The judgment that that some object a is an instance of a type A will be written as $a : A$. Opposed to set theory it is always a priori determined,

[1] `https://github.com/HoTT/book`

what type some constructed object will be an instance of, and this type is, up to definitional equality of types, fixed. If an object or a type can be written in two different ways, we will express the fact that two expressions coincide using "$\equiv$" (since "$=$" will later denote propositional equality). Likewise, "$:\equiv$" is the notation for abbreviating the the right hand side by the expression on the left hand side. It is important to note that it is decidable to check whether $a \equiv b$ holds for two given terms a and b.

Types in homotopy type theory are organized in **universes**. For every $i \in \mathbb{N}$ we assume to have a universe $\mathcal{U}_i$ which is itself, as an object, contained in the universe $\mathcal{U}_{i+1}$. In this way, all types, including the universes, can be seen as objects in some greater type. Often, it is assumed that universes are *cumulative* in the sense that if $A : \mathcal{U}_i$, then $A : \mathcal{U}_{i+1}$ (and thus, $A : \mathcal{U}_j$ for every $j \geq i$). Since this entails some computational difficulties for theorem provers, the language Lean will not incorporate universe cumulativity. A replacement for the cumulativity is an inductively defined lifting function $\mathcal{U}_i \to \mathcal{U}_{i+1}$. In the following, I will most of the time leave the index of a universe implicit and just denote it by $\mathcal{U}$. This means to say that these definitions are applicable to all (combinations of) universe indices. The reason for the need of multiple universes is that a system that simply assumed that $\mathcal{U} : \mathcal{U}$ would be inconsistent.

We will now take a look at some of the non-dependent type formers, some of which will later be extended to dependent ones. We will introduce these type formers by giving semi-formal rules for the formation of the type, the introduction of the type's instances and for their elimination.

The most basic type is the type $A \to B$ of **non-dependent** functions between two types $A, B : \mathcal{U}$. The inference rules that come with it are exactly those known from types λ-calculus:

$$\to\text{-}\textsc{Form}\ \frac{A, B : \mathcal{U}}{A \to B : \mathcal{U}} \qquad \to\text{-}\textsc{Intro}\ \frac{a : A \vdash \Phi[a/x] : B}{(\lambda x.\Phi) : A \to B}$$
$$\to\text{-}\textsc{Elim}\ \frac{f : A \to B \qquad a : A}{f(a) : B} \tag{2.1}$$

Here, Φ is a term that may have x as a free variable. $\Phi[a/x]$ denotes the replacement by every appearance of x by a. Of course, we have the rules of η-conversion and β-reduction:

$$\eta\ \frac{f : A \to B}{(\lambda x.f(x)) \equiv f} \qquad \beta\ \frac{(\lambda x.\Phi) : A \to B \qquad a : A}{(\lambda x.\Phi)(a) \equiv \Phi[a/x]} \tag{2.2}$$

These definitional equalities will from now on be used "silently" to replace subterms. In the logic interpretation of HoTT, function types model implication of propositions while in the set and topology interpretation they represent (arbitrary resp. continuous) maps.

One important special case of non-dependent function types are those of the form $A \to \mathcal{U}$ for a type $A : \mathcal{U}$. We call it the type of **type families** indexed by A. With propositions as types, these represent propositions that depend on a variable $x : A$. Topologically, assigning to each point of a space another space "above" it in a continuous fashion reflects the notion of a *fibration*.

Especially when thinking of types modeling propositions, it is important to find types that correspond to to the absolute truth values true and false. A false statement should not be provable, so it should be represented by the **empty type 0** which has no introduction rule at all:

$$\mathbf{0}\text{-}\textsc{Form}\ \frac{}{\mathbf{0} : \mathcal{U}} \qquad \mathbf{0}\text{-}\textsc{Elim}\ \frac{C : \mathbf{0} \to \mathcal{U} \qquad x : \mathbf{0}}{\mathsf{ind}_{\mathbf{0}}(C, x) : C(x)}$$

Here, ind_0 stands for "induction". The name indicates that **0** is generated *inductively* on an empty collection of constructors. The rule asserts that we can infer every possible statement once we constructed an instance of **0** ("ex falso quodlibet"). Just like for every induction principle we will state in the rest of this chapter, we can obtain a non-dependent version of the rule, called *recursion rule*, by assuming C to be a constant type family or, in other words, a type $A : \mathcal{U}$:

$$\mathsf{rec}_{\mathbf{0}}(A, x) :\equiv \mathsf{ind}_{\mathbf{0}}((\lambda y.A), x) : A.$$

The type corresponding to "true" is the **unit type 1** which provides exactly one way of constructing an instance of it:

$$\mathbf{1}\text{-}\textsc{Form}\ \frac{}{\mathbf{1} : \mathcal{U}} \qquad \mathbf{1}\text{-}\textsc{Intro}\ \frac{}{\star : \mathbf{1}}$$

$$\mathbf{1}\text{-}\textsc{Elim}\ \frac{C : \mathbf{1} \to \mathcal{U} \qquad p : C(\star) \qquad x : \mathbf{1}}{\mathsf{ind}_{\mathbf{1}}(C, p, x) : C(x)}$$

The elimination rule can be thought of ensuring that we can prove a statement about an arbitrary element of **1** by just proving it for the constructor $\star$. Additionally to the rule, we furthermore specify the behavior of the induction on the constructor itself, by saying that $\mathsf{ind}_{\mathbf{1}}(C, p, \star) \equiv p$. In a set theoretic context, **1** would be a (or "the") singleton, from the topological point of view it stands for a contractible space. Again, if we set $C(x) :\equiv A$ for some type $A : \mathcal{U}$, we obtain a non-dependent recursor $\mathsf{rec}_{\mathbf{1}}(A) : A \to \mathbf{1} \to A$, selecting for each point $a : A$ the function that maps to a.

Looking at the type of some of the recursors, for example $\mathsf{rec}_{\mathbf{1}}$, before giving all their arguments, we struggle to express their type using only non-dependent functions, since e.g. $\mathsf{rec}_{\mathbf{1}}(A)$ does not have the same type for every choice of $A : \mathcal{U}$. This is where dependent functions come into play.

2.2 Dependent Functions and Pairs

Dependent Functions, also called **Π-types**, are the core of dependent type theory. Opposed to a non-dependent function $f : A \to B$ between two types $A, B : \mathcal{U}$, which returns an instance of B when applied to whatever instance of A, the return type of a dependent function on a type family $B : A \to \mathcal{U}$ is $B(a)$ when the function is evaluated at some instance $a : A$. Of course, we can rediscover non-dependent function types as Π-types on constant type families. We write $\prod_{(a:A)} B(a)$ for the type of dependent functions on the type family $B : A \to \mathcal{U}$. Since to construct such a dependent function, we need to give an element of $B(a)$ for *every* $a : A$, we can think of $\prod_{(a:A)} B(a)$ as the statement "For all $a : A$, the statement $B(a)$ holds." In the topological interpretation, we can say that this corresponds to giving a point in each fiber of the fibration $B : A \to \mathcal{U}$ varying continuously on the chosen $a : A$. this is called a *section* of the fibration. The rules to form the type of Π-types and to introduce and apply dependent functions generalize the rules for non-dependent functions as follows:

$$\Pi\text{-}\textsc{Form}\ \frac{A : \mathcal{U}_i \qquad B : A \to \mathcal{U}_j}{(\prod_{(a:A)} B(a)) : \mathcal{U}_{max\{i,j\}}} \qquad \Pi\text{-}\textsc{Intro}\ \frac{a : A \vdash \Phi[a/x] : B(a)}{(\lambda x.\Phi) : \prod_{(a:A)} B(a)}$$
$$\Pi\text{-}\textsc{Elim}\ \frac{f : \prod_{(a:A)} B(a) \qquad a : A}{f(a) : B(a)} \tag{2.3}$$

Again, we have the rules for β-reduction and η-conversion like in the non-dependent case (2.2), yielding judgmental equalities $(\lambda x.f(x)) \equiv f$ and $(\lambda x.\Phi)(a) \equiv \Phi[a/x]$.

Having defined Π-types we are able to state the recursor for other basic, essentially non-dependent, type formers: Product and coproduct types. These take the type theoretic role of conjunction and disjunction of propositions, and of cartesian products and and disjoint unions of sets or topological spaces. The inference rules for **product types** are the following:

$$\times\text{-}\textsc{Form}\ \frac{A, B : \mathcal{U}}{A \times B : \mathcal{U}} \qquad \times\text{-}\textsc{Intro}\ \frac{a : A \qquad b : B}{(a,b) : A \times B}$$

$$\times\text{-}\textsc{Elim}\ \frac{C : A \times B \to \mathcal{U} \qquad p : \prod_{(a:A)} \prod_{(b:B)} C((a,b)) \qquad x : A \times B}{\mathsf{ind}_{A\times B}(C, p, x) : C(x)}$$

with the definitional equality $\mathsf{ind}_{A\times B}(C, p, (a,b)) \equiv p(a,b)$. Note that the type of $\mathsf{ind}_{A\times B}$ can now be expressed as

$$\prod_{C:A\times B\to\mathcal{U}} \left(\prod_{a:A} \prod_{b:B} C((a,b)) \right) \to \prod_{x:A\times B} C(x).$$

The first and second projection of an instance $x : A \times B$ is then simply defined by

$$\begin{aligned} \mathsf{pr}_1(x) &:\equiv \mathsf{ind}_{A\times B}((\lambda y.A), (\lambda a.\lambda b.a), x) : A \\ \mathsf{pr}_2(x) &:\equiv \mathsf{ind}_{A\times B}((\lambda y.A), (\lambda a.\lambda b.b), x) : B, \end{aligned}$$

yielding $\mathsf{pr}_1((a,b)) \equiv a$ and $\mathsf{pr}_2((a,b)) \equiv b$ judgmentally. An informal way of stating the meaning of ×-Elim would be: "To show a statement about all instances of a product type, is suffices to prove it for all pairs".

Dually to products we define the **coproduct** or **sum** of two types by giving the following rules:

$$+\text{-}\textsc{Form}\ \frac{A : \mathcal{U} \qquad B : \mathcal{U}}{A + B : \mathcal{U}}$$

$$+\text{-}\textsc{Intro1}\ \frac{a : A}{\mathsf{inl}(a) : A + B} \qquad +\text{-}\textsc{Intro2}\ \frac{b : B}{\mathsf{inr}(b) : A + B}$$

$$+\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : (A + B) \to \mathcal{U} \\ p : \prod_{(a:A)} C(\mathsf{inl}(a)) \qquad q : \prod_{(b:B)} C(\mathsf{inr}(b)) \qquad x : A + B \end{array}}{\mathsf{ind}_{A+B}(C, p, q, x) : C(x)}$$

Note that this is the first type former for which there is more than just one introduction rule. As a consequence, there is more than one "base case" to prove to use the induction rule.

Going back and looking at the product type, we recognize that we can, just as for the function type, find a more general, dependent version, the **dependent product type** or **Σ-type**. The gain of generality consists of the fact that, for the pairs that make up the type, the type of their second component may depend on the concrete value of the first component. This can be used to model existentially quantified statements like "there exists an $a : A$ such that $B(a)$ holds." Topologically, the Σ-type of a fibration $B : A \to \mathcal{U}$ represents the *total space* of B. The first projection of a dependent pair corresponds to the projection of a point in a fiber onto its base. (This is the map which topologists would traditionally refer to as "the fibration".) The inference rules for dependent products are:

$$\Sigma\text{-}\textsc{Form}\ \frac{A : \mathcal{U} \qquad B : A \to \mathcal{U}}{(\sum_{(a:A)} B(a)) : \mathcal{U}} \qquad \Sigma\text{-}\textsc{Intro}\ \frac{a : A \qquad b : B(a)}{(a,b) : (\sum_{(a:A)} B(a))}$$

$$\Sigma\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : ((\sum_{(a:A)} B(a))) \to \mathcal{U} \\ p : \prod_{(a:A)} \prod_{(b:B(a))} C((a,b)) \qquad x : (\sum_{(a:A)} B(a)) \end{array}}{\mathsf{ind}_{(\sum_{(a:A)} B(a))}(C, p, x) : C(x)}$$

Again, we assume the definitional equality

$$\mathsf{ind}_{(\sum_{(a:A)} B(a))}(C, p, (a,b)) \equiv p(a,b).$$

We can recover non-dependent products by setting B to be a constant type family. The projections to the first and second component are defined similar to the ones of non-dependent products for a dependent pair $x : \prod_{(a:A)} B(a)$:

$$\mathsf{pr}_1(x) :\equiv \mathsf{ind}_{(\prod_{(a:A)} B(a))}((\lambda x.A), (\lambda a.\lambda b.a), x) \text{ and}$$
$$\mathsf{pr}_2(x) :\equiv \mathsf{ind}_{(\prod_{(a:A)} B(a))}((\lambda x.B(\mathsf{pr}_1(x))), (\lambda a.\lambda b.b), x).$$

Product types, Σ-types, coproduct types, the unit type and the empty type all are examples for the larger class of **inductive types**. I will abstain from giving the general definition of inductive types and inductive type families and again refer to definitions in either the HoTT book [Uni13] and the introduction of inductive families by Peter Dybier [Dyb94], which provided the blueprint for the implementation in Lean. Instead, I will another a common example for an inductive type:

Just as the unit type was defined inductively on its constructor $\star$ and the coproduct on two constructor inl and inr, we can obtain the **natural numbers** $\mathbb{N}$ as the inductive type of a zero element $0 : \mathbb{N}$ and the successor function $S : \mathbb{N} \to \mathbb{N}$. These data yield, besides the obvious introduction rules, the following well-known elimination rule:

$$\mathbb{N}\text{-}\textsc{Elim}\ \frac{C : \mathbb{N} \to \mathcal{U} \qquad p : C(0) \qquad q : \prod_{(n:\mathbb{N})} C(n) \to C(S(n)) \qquad x : \mathbb{N}}{\mathsf{ind}_{\mathbb{N}}(C, p, q, x) : C(x)}$$

The judgmental equalities of the rule are $\mathsf{ind}_{\mathbb{N}}(C, p, q, 0) \equiv p$ and the recursive equation

$$\mathsf{ind}_{\mathbb{N}}(C, p, q, S(x)) \equiv q(x, \mathsf{ind}_{\mathbb{N}}(C, p, q, x)).$$

For a constant type family we get the non-dependent recursion

$$\mathsf{rec}_{\mathbb{N}}(A) :\equiv \mathsf{ind}(\lambda x.A) : A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A,$$

which is exactly the intuitive way to define a function $\mathbb{N} \to A$ by recursion. If we were to define what it means for a natural number to be odd, we could do this by

$$\mathsf{isodd} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathcal{U}, \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0})) : \mathbb{N} \to \mathcal{U},$$

where $A \to \mathbf{0}$ is inhabited if A is not, and thus models the negation of statements. For example, this gives us the statement that the number one is odd, witnessed by the following term:

$$\begin{aligned}(\lambda x.x) :\ & \mathbf{0} \to \mathbf{0} \\ & \equiv \mathsf{ind}_{\mathbb{N}}((\lambda x.\mathcal{U}), \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0}), 0) \to \mathbf{0} \\ & \equiv \mathsf{ind}_{\mathbb{N}}((\lambda x.\mathcal{U}), \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0}), S(0)) \\ & \equiv \mathsf{isodd}(S(0)).\end{aligned}$$

2.3 Propositional Equality

So far, all equalities were judgmental or definitional: They resulted from defining new notations and from the judgmental equalities that come with each induction rule. As mentioned before, it is decidable to check two terms for equality. But this means that with this kind of equality, we will not be able to reason about undecidable equality statements, or express equality *inside* the theory. Since in the logical interpretation of type theory, every statement should be represented by a type, so should the equality of two objects of the same type. This is why we introduce the **equality type** or **identity type** as our next type. It is obvious that for a type $A : \mathcal{U}$ and two objects $a, b : A$ there should be a type $a =_A b$ of which we need so construct an instance to show that a and b are "equal" in some sense. But what should the introduction and the elimination rule for such a type look like? It turns out that it should be the relation that is defined inductively on the witnesses for its reflexivity:

$$=\text{-}\textsc{Form}\ \frac{A : \mathcal{U} \qquad a, b : A}{a =_A b : \mathcal{U}} \qquad =\text{-}\textsc{Intro}\ \frac{A : \mathcal{U} \qquad a : A}{\mathsf{refl}_a : a =_A a}$$

$$=\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : \prod_{(a,b:A)}(a =_A b) \to \mathcal{U} \\ c : \prod_{(a:A)} C(a, a, \mathsf{refl}_a) \qquad a, b : A \qquad p : a =_A b \end{array}}{\mathsf{ind}_{=_A}(C, c, a, b, p) : C(x, y, p)}$$

The introduction rule gives us a canonical proof for the equality of two definitionally equal objects. The recursor says that to prove a statement about all pairs of points in a type and all equalities between them, it suffices to show it for reflexive case and the canonical equality proof refl. An elimination rule equivalent to the *unbased equality induction* or *J-rule* stated above is the following *based equality induction* where we fix the start point of each equality we reason about:

$$\textsc{Based}\text{-}=\text{-}\textsc{Elim}\ \frac{\begin{array}{c} a : A \qquad C : \prod_{(b:A)}(a =_A b) \to \mathcal{U} \\ c : C(a, \mathsf{refl}_a) \qquad b : A \qquad p : a =_A b \end{array}}{\mathsf{ind}'_{=_A}(a, C, c, b, p) : C(b, p)}$$

Both elimination rules are assumed to yield judgmental equalities when applied to the constructor refl itself:

$$\begin{aligned} &\mathsf{ind}_{=_A}(C, c, a, a, \mathsf{refl}_a) \equiv c(a) \text{ and} \\ &\mathsf{ind}'_{=_A}(a, C, c, a, \mathsf{refl}_a) \equiv c. \end{aligned}$$

We will see that for the topological interpretation of types, the identity type should not necessarily be seen as representing equality of points but rather *paths between points*.

Equal objects should be indiscernible in the sense that every property $C : A \to \mathcal{U}$ that is provable for some $a : A$ by giving an instance of $C(a)$

should also be provable for $b : A$, given an equality $p : a =_A b$. As a first lemma about equality we prove that this is indeed the case. (Note that the distinction between a definition and a lemma or theorem together with its proof is less clear in type theory than it is for set based mathematics.)

Lemma 2.3.1 (Transport)**.** *Let $A : \mathcal{U}$ be a type, $C : A \to \mathcal{U}$ a type family, and $a, b : A$ be equal by $p : a =_A b$. Then, for each $c : C(a)$, we obtain an object $p_*(c) : C(b)$. We call this the **transport** of c along the path p.*

Proof. We can use (based or unbased) path induction to reduce the statement to the one where $p \equiv \mathsf{refl}_a$. In this case, we simply to chose $p_*(c) :\equiv c$. Formally, we define

$$p_*(c) :\equiv \mathsf{ind}'_{=_A}(a, (\lambda b.\lambda p.C(b)), c, b, p).$$

$\square$

In the statement of this lemma, like in all the following, the list of preconditions should be thought of as an iterated Π-type, so that the full statement is of the type

$$\prod_{C:A\to\mathcal{U}} \prod_{a,b:A} \prod_{p:a=_A b} C(a) \to C(b).$$

We have seen that the propositional equality we defined above is, by its constructor reflexive. But since equality should be an equivalence relation, we have to prove its symmetry and transitivity, or, in the language of paths, provide the inverse and concatenation of paths.

Definition 2.3.2 (Inverse paths)**.** Let $A : \mathcal{U}$ and $a, b : A$. For every path $p : a =_A b$, there is a path $p^{-1} : b =_A a$ such that $\mathsf{refl}_a^{-1} \equiv \mathsf{refl}_a$ for every $a : A$.

Proof. By path induction, it suffices to provide $\mathsf{refl}_a^{-1} : a =_A a$, which we provide by giving refl_a itself. Formally: $p^{-1} :\equiv \mathsf{ind}'_{=_A}(a, (\lambda b.\lambda p.b =_A a), \mathsf{refl}_a, b, p)$. $\square$

Definition 2.3.3 (Concatenation of paths)**.** For a type $A : \mathcal{U}$, $a, b, c : A$ and paths $p : a = b$ and $q : b = c$ there is a path $p \cdot q : a = c$ which can be chosen in a way such that $\mathsf{refl}_a \cdot \mathsf{refl}_a \equiv \mathsf{refl}_a$.

Proof. We again apply induction on the equalities involved to end up having only to specify an instance of $a = a$ which we choose to be refl_a. I will from now on abstain from giving the type families and applications of the equality elimination rule explicitly. More formal proofs for these definitions can be found in any formalization or in the HoTT book [Uni13]. $\square$

We see that this inversion and concatenation are exactly the operations which are part of a groupoid. It turns out that they also fulfill the defining properties of a groupoid:

Definition 2.3.4 (Groupoid laws)**.** Let $A : \mathcal{U}$, $a, b, c, d : A$ and $p : a = b$, $q : b = c$ and $r : c = d$. Then,

- $p = p \cdot \mathsf{refl}_b = \mathsf{refl}_a \cdot p$,
- $p^{-1} \cdot p = \mathsf{refl}_b$, $p \cdot p^{-1} = \mathsf{refl}_a$,
- $(p^{-1})^{-1} = p$ and
- $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

Proof. We can prove all these by applying path induction to all the three paths involved. $\square$

Note that all these propositional equalities are *between equalities*. For example, if we wrote the first equation mentioned in the previous definition noting the type it was in, it would be $p =_{a =_A b} p \cdot \mathsf{refl}_b$. We call these equalities **2-paths** or **2-dimensional paths**. These make the equality in a type an ∞-groupoid.

To be a suitable notion of equality, it should be respected by functions in the sense that for a function $f : A \to B$ and an equality $p : a = b$ in its domain we should be able to obtain an equality $f(a) = f(b)$ in the codomain. This is indeed the case, which we can prove by induction on the equality:

Lemma 2.3.5. *Let $A, B : \mathcal{U}$, $f : A \to B$, and $a, b : A$. Then, there is a function $\mathsf{ap}_f : (a = b) \to (f(a) = f(b))$ such that $\mathsf{ap}_f(\mathsf{refl}_a) \equiv \mathsf{refl}_{f(a)}$.* $\square$

We call ap_f the application of f on paths. ap is functorial with respect to the path in the sense that it respects the concatenation and inversion of paths, and with respect to the function by respecting composition of functions and acting neutral on the identity function.

Lemma 2.3.6. *Let $A, B, C : \mathcal{U}$, $f : A \to B$ and $g : B \to C$. For paths $p : a =_A b$ and $q : b =_A c$ we have equalities*

- $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$,
- $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$,
- $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$, *and*
- $\mathsf{ap}_{\mathsf{id}_A}(p) = p$. □

But what if f is a dependent function? Then, the $f(a)$ and $f(b)$ would not necessarily be instances of the same type and so $f(a) = f(b)$ would not be a valid type. But we can use the transport (Lemma 2.3.1) to turn $f(a)$ into a point in the same fiber as $f(b)$.

Lemma 2.3.7. *Let $A : \mathcal{U}$, $P : A \to \mathcal{U}$ and $f : \prod_{(a:A)} P(a)$. Then, we can construct a dependent function*

$$\mathsf{apd}_f : \prod_{a,b:A} \prod_{p:a=b} (p_*(f(a)) = f(b)),$$

such that $\mathsf{apd}_f(a, a, \mathsf{refl}_a) \equiv \mathsf{refl}_{f(a)}$. □

2.4 Equivalences and Univalence

In set based mathematics we clearly do not care about whether two proofs for the equality of two elements are themselves equal or not. Different from that, in the topological setting, where equalities correspond to paths and equalities between equalities correspond to *homotopies* between paths, we might want to consider types, where not all paths are homotopic. But what about equalities between types? From the logical viewpoint, a good notion of equality would be the *biconditional*: Two statements $A, B : \mathcal{U}$ should be equal, if and only if we can find instances of $A \to B$ and $B \to A$. A notion of equality for sets, which is weaker than definitional equality, would be their isomorphicness in the category of sets — that is, finding a bijection between them. When talking about the homotopy type of a topological space however, spaces are considered equivalent, and their homotopy type equal, when they are *homotopy equivalent*. There is a notion of equivalence that captures all these three viewpoints. To formulate it, we first need the definition of homotopic functions, as known from classic homotopy theory:

Definition 2.4.1 (Homotopy of functions)**.** Two maps $f, g : A \to B$ are called **homotopic** if for all $a : A$ we have $f(a) = g(a)$. We define the notation

$$f \sim g :\equiv \prod_{a:A} (f(a) = g(a)).$$

We can define the same for two dependent functions $f, g : \prod_{(a:A)} B(a)$ over the same type family.

Definition 2.4.2 (Equivalences)**.** Let $A, B : \mathcal{U}$. A function $f : A \to B$ is called an **equivalence** between A and B, if there is a $g : B \to A$ such that $\eta : g \circ f \sim \mathrm{id}_A$ and $\epsilon : f \circ g \sim \mathrm{id}_B$ and furthermore

$$\tau : \prod_{a:A} \mathsf{ap}_f(\eta(a)) =_{f(g(f(a)))=a} \epsilon(f(a)) \equiv \mathsf{ap}_f \circ \eta \sim \epsilon \circ f.$$

We need τ to make sure that each two witnesses f is an equivalence are equal. Since τ looks like the one of the two commutativity conditions for pairs of adjoint functors, this kind of equivalence is also called a **half adjoint equivalence**. The type of witnesses for f to be an equivalence shall be

$$\mathsf{isequiv}(f) :\equiv \sum_{g:B \to A} \; \sum_{\eta : g \circ f \sim \mathrm{id}_A} \; \sum_{\epsilon : f \circ g \sim \mathrm{id}_B} \mathsf{ap}_f \circ \eta \sim \epsilon \circ f.$$

The type of all equivalences between two types $A, B : \mathcal{U}$ is denoted by

$$A \simeq B :\equiv \sum_{f : A \to B} \mathsf{isequiv}(f).$$

It is an easy exercise to show that equivalence of types is indeed an equivalence relation. $\mathsf{isequiv}(f)$ itself is equivalent to other known definitions of equivalence, like for example the existence of a left inverse and of a right inverse of f. Easy examples for equivalent types include $(\mathbf{1} \times A) \simeq A$, $(\mathbf{1} \to A) \simeq A$, $(\mathbf{0} + A) \simeq A$ and $(\mathbf{0} \times A) \simeq \mathbf{0}$, for a type $A : \mathcal{U}$.

But how do equality and equivalence of types relate to each other? Clearly equal types can be proven to be equivalent, just using path induction on the equality between them and the reflexivity of equivalence:

Lemma 2.4.3. *Let $A, B : \mathcal{U}$. Then there is a function*

$$\mathsf{idtoeqv}_{A,B} : (A =_{\mathcal{U}} B) \to (A \simeq B),$$

which for the reflexivity on $\mathcal{U}$ is defined by

$$\mathsf{idtoeqv}_{A,A}(\mathsf{refl}_A) \equiv (\mathrm{id}_A, \mathrm{id}_A, (\lambda a.\mathsf{refl}_a), (\lambda a.\mathsf{refl}_a), (\lambda a.\mathsf{refl}_{\mathsf{refl}_a})).$$

□

On the other hand, there is no way to obtain an equality of types from an equivalence. Vladimir Voevodsky proposed the following axiom to make it possible:

Axiom 2.4.4 (Univalence)**.** *For* $\mathsf{idtoeqv}_{A,B}$ *is itself an equivalence for every choice of* $A, B : \mathcal{U}$.

This implies that

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B)$$

and yields an inverse to $\mathsf{idtoeqv}_{A,B}$ *which we call*

$$\mathsf{ua}_{A,B} : (A \simeq B) \to (A =_{\mathcal{U}} B).$$

From now on, we will always assume this axiom since as it is essential for the existence of higher types and thus for homotopy type theory in general. Another important consequence of assuming univalence is that we can prove the equality of two dependent functions by giving a homotopy between them:

Theorem 2.4.5. *Let* $A : \mathcal{U}$, $B : A \to \mathcal{U}$ *and* $f, g : \prod_{(a:A)} B(a)$. *Then,*

$$(f \sim g) \to (f =_{A \to B} g).$$

We omit the non-trivial proof for this theorem and refer to the slightly distinct approaches in the HoTT book [Uni13] and the Lean library.

2.5 Truncated Types

As already considered in the previous sections, every type $A : \mathcal{U}$ comes with a type of paths $a =_A b$ for each $a, b : A$ and iterated, higher dimensional paths between these paths. Sometimes we want a type to not contain any information in higher paths by assuming that all its n-dimensional paths are equal. This leads to the concept of **truncated types**.

The first definition shall describe what it means for a type to contain only one point. We already defined the unit type $\mathbf{1}$ as the type with exactly one constructor, but we want a property that we can check for any given type. This property will then be, as a type, equivalent to being equivalent to the unit type. When thinking about types in the logical context, this should remind the reader of the fact that a true statement which does not depend on any free variables, is logically equivalent to the canonical "true" statement. In topology, spaces that are homotopy equivalent to a single point are called contractible. They are equivalently characterized as those spaces which contain a point serving as *center* for a contraction — that is, a continuous choice of paths from each point in the space to the center. This is the definition of contractibility which is directly translated to its counterpart in homotopy type theory:

Definition 2.5.1 (Contractibility). A type $A : \mathcal{U}$ is called **contractible** if

$$\mathsf{isContr}(A) :\equiv \sum_{x:A} \prod_{a:A} (a =_A x).$$

Analyzing this definition, we first observe that a contractible type is inhabited by $\mathsf{pr}_1(\mathsf{isContr}(A))$ and that for each $a, b : A$ we can find a path between a and b by

$$\mathsf{pr}_2(\mathsf{isContr}(A))(a) \cdot \mathsf{pr}_2(\mathsf{isContr}(A))(b)^{-1} : a =_A b.$$

This equality of all objects in contractible types makes them the class of types we want to use for *true propositions*. As mentioned above, contractible types are equivalent to the unit type:

Lemma 2.5.2. *Let $A : \mathcal{U}$ be a contractible type. Then, $A \simeq \mathbf{1}$.*

Proof. Let $p : \mathsf{isContr}(A)$. We obtain a function $f : A \to \mathbf{1}$ by setting $f(a) :\equiv \star$ for each $a : A$ and its inverse $g : \mathbf{1} \to A$, which by induction we only have to define on the constructor of $\mathbf{1}$, by $g(\star) :\equiv \mathsf{pr}_1(p)$. The proof that f and g are indeed inverse to each other is then done using induction on the unit type and the paths yielded by $\mathsf{pr}_2(p)$. □

If we want to build a class for all propositions, whether true or not, we can use this consequence of contractibility as a definition and remove the inhabitedness condition:

Definition 2.5.3 (Mere proposition). A type $A : \mathcal{U}$ is a **mere proposition** if

$$\mathsf{isProp}(A) :\equiv \prod_{a,b:A} (a =_A b).$$

Besides all contractible types, more concrete examples for mere propositions are $\mathbf{0}$, $\mathbf{1}$, function types with merely propositional codomains, and many of the defined properties like being an equivalence $\mathsf{isequiv}(f)$, being a contractible type $\mathsf{isContr}(A)$ and even being a mere proposition $\mathsf{isProp}(A)$. In the definition of equivalence, the coherence condition τ is needed to make it a mere proposition. Examples for types which are not mere propositions are also easy to find: $A + B$, whenever A and B are both non-empty types, and $\mathbb{N}$ clearly contain non-equal objects.

Since the proof that two functions are inverses of each is trivial on a pair of mere propositions, function types on mere propositions works just as one would expect from propositions:

Lemma 2.5.4. *Let $A, B : \mathcal{U}$ be mere propositions and $f : A \to B$, $g : B \to A$. Then, $A \simeq B$.* □

When dealing with sets, we do not require all their objects to be equal, but only the proofs of equality:

Definition 2.5.5 (Sets)**.** A type $A : \mathcal{U}$ is called a **set**, if there is an object in

$$\mathsf{isSet}(A) :\equiv \prod_{a,b:A} \prod_{p,q:a=_A b} (p =_{a=_A b} q).$$

So far most types we encountered were sets, but we can use univalence to show that each of our universes is an example for a type that is not a set:

Lemma 2.5.6. *For each i, the universe $\mathcal{U}_i$ is not a set.*

Proof. Consider the type $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1} : \mathcal{U}_i$ and let $0_\mathbf{2} :\equiv \mathsf{inl}(\star)$ and $1_\mathbf{2} :\equiv \mathsf{inr}(\star)$ be its two constructors. As true for every (non-higher) inductive type with multiple constructors, distinct constructors are unequal:

$$(0_\mathbf{2} \neq 1_\mathbf{2}) \equiv (0_\mathbf{2} = 1_\mathbf{2} \to \mathbf{0}).$$

We define a function $f : \mathbf{2} \to \mathbf{2}$ by setting $f(0_\mathbf{2}) :\equiv 1_\mathbf{2}$ and $f(1_\mathbf{2}) :\equiv 0_\mathbf{2}$. It's easy to see that f, as an involution, is an equivalence, which, by univalence results in a path $p : \mathbf{2} =_{\mathcal{U}_i} \mathbf{2}$. If we assume $\mathcal{U}_i$ to be a set, we receive an equality $p = \mathsf{refl}_\mathbf{2}$. But transporting along that equality, f would be equal to $\mathrm{id}_f rm{-}e$ and thus

$$0_\mathbf{2} = f(1_\mathbf{2}) = \mathrm{id}_2(1_\mathbf{2}) = 1_\mathbf{2},$$

which, with the above inequality gives the desired result. □

Just like $\mathsf{isContr}(A)$ and $\mathsf{isProp}(A)$, $\mathsf{isSet}(A)$ itself is a mere proposition. We can also prove that each mere proposition is a set, by which we get

$$\mathsf{isContr}(A) \to \mathsf{isProp}(A) \to \mathsf{isSet}(A)$$

for each type $A : \mathcal{U}$. Calling a set a 0-type, a mere proposition a (-1)-type and a contractible type a (-2)-type, we can extend this chain by the following definition of an arbitrary n-type or n-truncated type:

Definition 2.5.7 (Truncated types)**.** By recursion on the index we define $\mathsf{is}\text{-}n\text{-}\mathsf{type}$ as a function $\mathcal{U} \to \mathcal{U}$ by

$$\mathsf{is}\text{-}n\text{-}\mathsf{type}(A) :\equiv \begin{cases} \mathsf{isContr}(A) & \text{if } n = -2 \text{ and} \\ \prod_{(a,b,:A)} \mathsf{is}\text{-}n'\text{-}\mathsf{type}(a =_A b) & \text{if } n = n' + 1 \text{ otherwise.} \end{cases}$$

For example, for $n = 1$ this yields

$$\mathsf{is}\text{-}1\text{-}\mathsf{type}(A) \simeq \prod_{a,b:A} \; \prod_{p,q:a=_A b} \; \prod_{\alpha,\beta:p=_{a=_A b}q} (\alpha =_{p=_{a=_A b}q} \beta).$$

Collecting basic facts about n-types, we have

Lemma 2.5.8.

- *If $A : \mathcal{U}$ is an n-type, then A is an $(n+1)$-type.*
- *For each $n \geq -2$ and $A : \mathcal{U}$, the type* $\mathsf{is}\text{-}n\text{-}\mathsf{type}(A)$ *is a mere proposition.*
- *If $A : \mathcal{U}$ is an n-type and $B : A \to \mathcal{U}$ such that for each $a : A$, $B(a)$ is an n-type, then $\sum_{(a:A)} B(a)$ is an n-type as well.*
- *If $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ are such that $B(a)$ is an n-type for each $a : A$, then $\prod_{(a:A)} B(a)$ is an n-type as well.*
- *Let $n \geq -1$. Then, $A : \mathcal{U}$ is an $(n+1)$-type if and only if for all $a : A$, the equality type $a =_A a$ is an n-type.*
- *Let $n \geq 0$. Then, $A : \mathcal{U}$ is an n-type, if and only if for all $a : A$, the n-*fold iterated loop space $\Omega^{n+1}(A, a)$, *which is defined using recursion on n by*

$$\Omega^1(A, a) :\equiv (a =_A a) \text{ and}$$
$$\Omega^{n+1}(A, a) :\equiv (\mathsf{refl}_{\ddots_a} =_{\Omega^n(A,a)} \mathsf{refl}_{\ddots_a}),$$

is contractible.

2.6 Higher Inductive Types

Besides "normal" inductive types, which have constructors that objects of or functions to the type, there is the concept of a **higher inductive type**. Higher inductive types can also contain constructors which do not yield instances of the type itself, but instead propositional equalities between instances. Like for basic inductive types, I will abstain from giving rules for general higher inductive type but instead we will take a look at some common higher inductive types.

We can picture the path constructors as *gluing* a pair of other constructors together by a new path. One of the simplest examples for such a case is the one where the type is constructed by just two instances 0_I and 1_I, together with a path $\mathsf{seg} : 0_I = 1_I$. The inference rules for this **interval type** I are the following:

$$I\text{-}\textsc{Form}\ \frac{}{I : \mathcal{U}} \qquad I\text{-}\textsc{Intro1}\ \frac{}{0_I : I} \qquad I\text{-}\textsc{Intro2}\ \frac{}{1_I : I} \qquad I\text{-}\textsc{Intro3}\ \frac{}{\mathsf{seg} : 0_I = 1_I}$$

$$I\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : I \to \mathcal{U} \\ c_0 : C(0_I) \qquad c_1 : C(1_I) \qquad p : \mathsf{seg}_*(c_0) = c_1 \qquad x : I \end{array}}{\mathsf{ind}_I(C, c_0, c_1, p, x) : C(x)}$$

Of course, we again get judgmental equalities for the case where x is one of the constructors. As a suitable counterpart for the third constructor, we assume that $\mathsf{apd}_f(seg) = s$, using the dependent application defined in 2.3.7, where $f(x) :\equiv \mathsf{ind}_I(C, c_0, c_1, p, x)$.

In words, to prove a statement for an arbitrary point on the interval, one has to prove it for both endpoints and show that the proofs can be connected by a path "over" seg. In the case of the non-dependent recursor, the transport is not necessary. The type I is a very uninteresting type because of the following:

Lemma 2.6.1. *The interval I is contractible, and therefore $I \simeq \mathbf{1}$.* □

It turns out that if we don't assume the path to be between to distinct constructor instances but instead be a *loop* based in a single constructor we obtain the homotopy type $\mathbb{S}^1$ of a **circle** or **1-sphere**:

$$\mathbb{S}^1\text{-}\textsc{Form}\ \frac{}{\mathbb{S}^1 : \mathcal{U}} \qquad \mathbb{S}^1\text{-}\textsc{Intro1}\ \frac{}{\mathsf{base} : \mathbb{S}^1} \qquad \mathbb{S}^1\text{-}\textsc{Intro2}\ \frac{}{\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}}$$

$$\mathbb{S}^1\text{-}\textsc{Elim}\ \frac{C : \mathbb{S}^1 \to \mathcal{U} \qquad c : C(\mathsf{base}) \qquad p : \mathsf{loop}_*(c) = c \qquad x : \mathbb{S}^1}{\mathsf{ind}_{\mathbb{S}^1}(C, c, p, x) : C(x)}$$

Topologically speaking, the elimination rule tells us that we can find all sections of a fibration above the circle up to homotopy by appending "constant sections" to paths that only lie in one fiber. Again, if we take C to be the constant type family, we obtain a non-dependent recursor that lets us define a function $f : \mathbb{S}^1 \to B$ by providing a point b, which becomes $f(\mathsf{base})$ and a loop $b =_B b$ which is propositionally equal to $\mathsf{ap}_f(\mathsf{loop})$.

We can show that $\mathsf{loop} \neq \mathsf{refl}_{\mathsf{base}}$ and thus, $\mathbb{S}^1$ provides another example for a type which is not a set. Just as we used the type $\mathbf{2}$, which is not a mere proposition, to show that its surrounding universe is not a set, we can show that a universe which contains the circle cannot be a 1-type. We can continue this correspondence by introducing higher dimensional spheres. As is topology, they can be built as the **suspension** of a lower dimensional sphere. Since suspension is an important operation, especially for homology, we define it as a type former on arbitrary types (its notation Σ should not be confused with the notation for the type of dependent paris):

$$\textsc{Susp-Form}\ \frac{A : \mathcal{U}}{\Sigma A : \mathcal{U}} \qquad \textsc{Susp-Intro1}\ \frac{A : \mathcal{U}}{N : \Sigma A} \qquad \textsc{Susp-Intro2}\ \frac{A : \mathcal{U}}{S : \Sigma A}$$

$$\textsc{Susp-Intro3}\ \frac{A : \mathcal{U}}{\mathsf{merid} : A \to N =_{\Sigma A} S}$$

$$\textsc{Susp-Elim}\ \frac{\begin{array}{c} C : \Sigma A \to \mathcal{U} \qquad n : C(N) \\ s : C(S) \qquad m : \prod_{(a:A)} \mathsf{merid}(a)_*(n) =_{C(S)} s \qquad x : \Sigma A \end{array}}{\mathsf{ind}_{\Sigma A}(C, n, s, m, x) : C(x)}$$

We have judgmental equalities for the point constructors and a propositional equality to apd for the path constructor as in the previous examples. It's an easy task to prove that $\Sigma\mathbf{2} \simeq \mathbb{S}^1$ which leads us to defining

$$\begin{aligned} \mathbb{S}^0 &:\equiv \mathbf{2} \text{ and} \\ \mathbb{S}^{n+1} &:\equiv \Sigma\mathbb{S}^n \end{aligned}$$

recursively. For each $n : \mathbb{N}$, the type S^{n+1} is not an n-type, the 2-sphere, for example, is not truncated at any level.

Other important topological operations like pushouts, quotients, and colimits can be defined in a similar way. When using higher types we often want to make a type n-truncated for some n by "collapsing" all equalities above dimension n while preserving its properties below that level. This operation is called n-**truncation** and is defined by the following formation, introduction and elimination rules:

$$\textsc{Trunc-Form}\ \frac{A : \mathcal{U} \qquad n : \mathbb{N}}{\|A\|_n : \mathcal{U}} \qquad \textsc{Trunc-Intro}\ \frac{a : A}{|a|_n : \|A\|_n}$$

$$\textsc{Trunc-Elim}\ \frac{\begin{array}{c} C : \|A\|_n \to \mathcal{U} \qquad p : \prod_{(x:\|A\|_n)} \mathsf{is}\text{-}n\text{-}\mathsf{type}(C(x)) \\ g : \prod_{(a:A)} P(|a|_n) \qquad x : \|A\|_n \end{array}}{\mathsf{ind}_{\|A\|_n}(C, p, g, x) : C(x)}$$

As one can easily derive from the elimination rule, $\|A\|_n$ is an n-type and, together with the introduction rule, we can prove that every map in $A \to B$, for an n-type B, factors through $\|A\|_n$ by $|\cdot|_n$ and $\mathsf{ind}_{\|A\|_n}$.

Of course there are many interactions between all the elements of homotopy type theory that were introduced in this chapter. A deeper analysis of these can be found in the HoTT book [Uni13] as well as in many subsequent publications. This introduction should suffice to give enough understanding for the reader to know all definitions used in Chapter 4. Describing the general syntax and semantics of higher inductive types still remains an open problem.

Chapter 3

Non-Abelian Topology

This section describes the basic notions of non-abelian topology which I formalized and applied to higher types in homotopy type theory instead of topological spaces. Most of the definitions are taken from the book "Non-abelian Algebraic Topology" by Ronald Brown, Philip J. Higgins and Rafael Sivera [BHS11]. The structures used extend classical homotopy theory by considering *fundamental groupoids* with multiple base points, characterizing the interaction between the first and the second homotopy group of a space by *crossed modules* as well as *n-fold categories*, for which we will only consider the case $n = 2$.

3.1 Double Categories

To make the precise definition of a double category easier, we observe that we can define a (small) category C by giving a tuple $(ob_C, hom_C, \partial^-, \partial^+, \epsilon, \circ_C)$ where

- ob_C is the set of objects,
- hom_C is a set that contains all morphisms,
- ∂^- and $\partial^+ : hom_C \to ob_C$ are maps assigning to each morphism f its domain and codomain,
- $\epsilon : ob_C \to mor_C$ gives the identity morphism at each element (this implies $\partial^- \circ_C \epsilon = \partial^+ \circ_C \epsilon = \text{id}$), and

- $\circ_C$ denotes the composition of morphisms as a partial function $hom_C \times hom_C \to hom_C$, defined for all $(g, f) \in hom_C \times hom_C$ where $\partial^+(f) = \partial^-(g)$.

In accordance with the geometric interpretation that objects correspond to points while morphisms correspond to lines we will call ∂^- and ∂^+ **boundary** or **face maps** while ϵ will often be referred to as **degeneracy map**.

Extending this idea, we want a double category to be an algebraic structure that does not only contain points and lines but also squares, bounded by four bounding sides, which we will also call **faces**. Like shown in Figure 3.1, we must impose conditions on the face maps so that we can really speak of squares. Furthermore, just as we needed degenerate lines at each point, we will have degenerate squares that have a given line at both of their vertical or both of their horizontal faces. In our definiton, $\partial_1^-(u)$, $\partial_1^+(u)$, $\partial_2^-(u)$ and $\partial_2^+(u)$ will correspond to the upper, lower, left and right face of a square u, respectively. The whole set of faces of a given square $u \in D_2$ will be referred to as **shell** of this square.

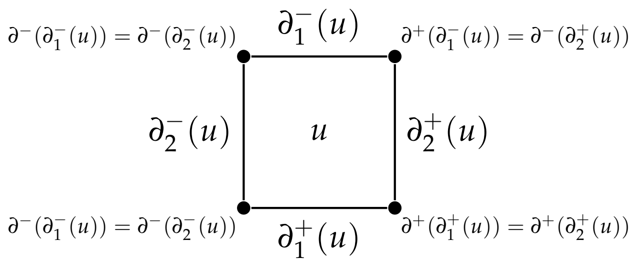

$\partial^-(\partial_1^-(u)) = \partial^-(\partial_2^-(u))$
$\partial_1^-(u)$
$\partial^+(\partial_1^-(u)) = \partial^-(\partial_2^+(u))$
$\partial_2^-(u)$
u
$\partial_2^+(u)$
$\partial^-(\partial_1^-(u)) = \partial^-(\partial_2^-(u))$
$\partial_1^+(u)$
$\partial^+(\partial_1^+(u)) = \partial^+(\partial_2^+(u))$

Figure 3.1: A square $u \in D_2$, its faces, and its corners.

Definition 3.1.1 (Double category)**.** A **double category** D is given by the following data: Three sets D_0, D_1, and D_2, the elements of which are respectively called **0-, 1- and 2-cells**, together with maps ∂^-, ∂^+, ϵ, $\circ_D$, ∂_1^-, ∂_1^+, ϵ_1, $\circ_1$, ∂_2^-, ∂_2^+, ϵ_2, and $\circ_2$ that make these sets form three categories:

- a category $(D_0, D_1, \partial^-, \partial^+, \epsilon, \circ_D)$ on D_0, often called the **(1-)skeleton** of the double category,
- a **vertical category** $(D_1, D_2, \partial_1^-, \partial_1^+, \epsilon_1, \circ_1)$, and
- a **horizontal category** $(D_1, D_2, \partial_2^-, \partial_2^+, \epsilon_2, \circ_2)$.

The mentioned maps are required to satisfy the following **cubical identities**:

$$\begin{aligned}\partial^- \circ \partial_1^- &= \partial^- \circ \partial_2^-,\\ \partial^- \circ \partial_1^+ &= \partial^+ \circ \partial_2^-,\\ \partial^+ \circ \partial_1^- &= \partial^- \circ \partial_2^+,\\ \partial^+ \circ \partial_1^+ &= \partial^+ \circ \partial_2^+,\end{aligned} \tag{3.1}$$

$$\begin{aligned}\partial_1^- \circ \epsilon_2 &= \epsilon \circ \partial^-,\\ \partial_1^+ \circ \epsilon_2 &= \epsilon \circ \partial^+,\\ \partial_2^- \circ \epsilon_1 &= \epsilon \circ \partial^-,\\ \partial_2^+ \circ \epsilon_1 &= \epsilon \circ \partial^+, \text{ and}\end{aligned} \tag{3.2}$$

$$\epsilon_1 \circ \epsilon = \epsilon_2 \circ \epsilon =: 0. \tag{3.3}$$

The boundary and degeneracy maps of the vertical category are furthermore assumed to be a homomorphism with respect to the composition of the horizontal category, and vice versa:

$$\begin{aligned}\partial_2^-(v \circ_1 u) &= \partial_2^-(v) \circ_D \partial_2^-(u),\\ \partial_2^+(v \circ_1 u) &= \partial_2^+(v) \circ_D \partial_2^+(u),\\ \partial_1^-(v \circ_2 u) &= \partial_1^-(v) \circ_D \partial_1^-(u),\\ \partial_1^+(v \circ_2 u) &= \partial_1^+(v) \circ_D \partial_1^+(u),\\ \epsilon_2(g \circ_D f) &= \epsilon_2(g) \circ_1 \epsilon_2(f), \text{ and}\\ \epsilon_1(g \circ_D f) &= \epsilon_1(g) \circ_2 \epsilon_1(g),\end{aligned} \tag{3.4}$$

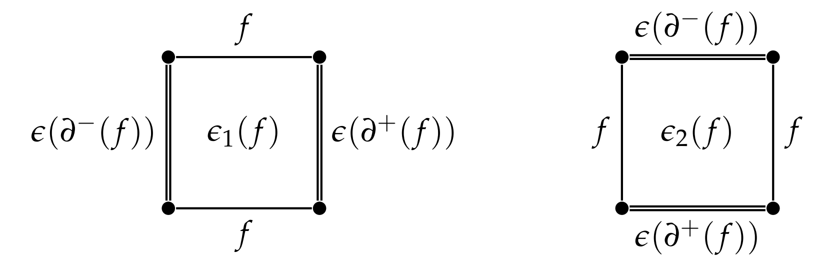
for each $f, g \in D_1$ and $u, v \in D_2$ where the compositions are defined.

A last condition, called the **interchange law**, has to be fulfilled: For each $u, v, w, x \in D_2$,

$$(x \circ_2 w) \circ_1 (v \circ_2 u) = (x \circ_1 v) \circ_2 (w \circ_1 u) \tag{3.5}$$

has to hold if it is well-defined.

As seen in Figure 3.1, the four identities (3.1) correspond to the well-definedness of the corners of a given square $u \in D_2$. The next four equations (3.2) tell us that for any line $f \in D_1$, besides the identities $\partial_1^\pm(\epsilon_1(f)) = f$ and $\partial_2^\pm(\epsilon_2(f)) = f$ (which follow from the definition of a category), the remaining two faces of a degenerate square are defined as consisting of the suitable degenerate lines. Figure 3.2 illustrates the cases of both the vertical and the horizontal category. Equation (3.3) is to make sure that in the case



f
$\epsilon(\partial^-(f))$
$\epsilon_1(f)$
$\epsilon(\partial^+(f))$
f
$\epsilon(\partial^-(f))$
f
$\epsilon_2(f)$
f
$\epsilon(\partial^+(f))$

Figure 3.2: Degenerate squares of the vertical and horizontal category for a given line $f \in D_1$. Degenerate lines are drawn as double lines.

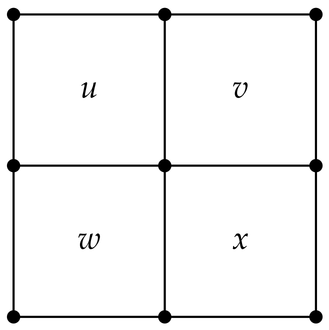


Figure 3.3: The grid we use to illustrate the composition $(x \circ_2 w) \circ_1 (v \circ_2 u)$ as well as $(x \circ_1 v) \circ_2 (w \circ_1 u)$, which are identical by the interchange law.

where we take the degenerate square of a line which is itself degenerate, and end up with a square with all four faces degenerate, it doesn't matter whether we choose the vertical or the horizontal degeneracy but that instead we receive a unique zero-element for each $x \in D_0$.

The linearity condition (3.4) serves to define the two faces of a composite square that are not already fixed by the definition of vertical and horizontal category.

The interchange law can be applied to four squares that are composable as a 2×2-grid to another square. It makes sure that the result of the composition does not depend on whether we first compose horizontally and then vertically or the other way round. This justifies illustrating such compositions, as well as larger, iterated ones, by "grids" like the one shown in Figure 3.3.

Starting from a given 1-skeleton, it is easy to find some very simple but nevertheless very useful and recurring examples for double categories:

Example 3.1.2. Let $C = (C_0, C_1, \partial^-, \partial^+, \epsilon, \circ_C)$ be a category. The **square**

double category on C is defined by setting $D_0 := C_0$, $D_1 := C_1$ and

$$\begin{aligned} D_2 := \{(f,g,h,i) | \partial^+(f) = \partial^-(i), \partial^+(i) = \partial^+(g), \\ \partial^-(g) = \partial^+(h), \partial^-(h) = \partial^-(f)\} \subseteq D_1^4 \end{aligned}$$

and ∂_1^-, ∂_1^+, ∂_2^-, ∂_2^+ to be the four projections on this set. (To keep things consistent, I will always state the faces of a square in the order upper, lower, left and right.) The degenerate squares are the obvious quadruples $(f,f,\mathrm{id},\mathrm{id})$ and $(\mathrm{id},\mathrm{id},f,f)$ for a morphism $f \in C_1$. Composing two squares (f,g_1,h_1,i_1) and (g_1,g_2,h_2,i_2) vertically yields a square $(f,g_2,h_2 \circ h_1, i_2 \circ i_1)$.

Note that f, g, h, and i do not have to form a *commutative* square — the square double category on C rather collects all possible squares in C. We denote the square double category on C as $\square' C$.

Example 3.1.3. Let again be $C = (C_0, C_1, \partial^-, \partial^+, \epsilon, \circ_C)$ a category. We restrict the square double category on C to commutative squares and obtain the **commutative square double category** or **shell double category** on C:

$$\begin{aligned} &D_0 := C_0, \\ &D_1 := C_1 \text{ and} \\ &D_2 := \left\{(f,g,h,i) \in (\square' C)_2 \,\middle|\, g \circ_C h = i \circ_C f\right\}. \end{aligned}$$

Faces and degeneracies are trivial, for defining the the vertical composition of two squares (f,g_1,h_1,i_1) and (g_1,g_2,h_2,i_2), one obtains the commutativity of the composed square by

$$\begin{aligned} g_2 \circ_C h_2 \circ_C h_1 &= i_2 \circ_C g_1 \circ_C h_1 \\ &= i_2 \circ_C i_1 \circ_C f \end{aligned}$$

and analogously for the horizontal composition. We write $\square C$ for the commutative square double category on C.

For the purpose of building a category of double categories we have to define what it means for a map to preserve the structure of a double category:

Definition 3.1.4. A **double functor** F between double categories D and E is a triple of maps (F_0, F_1, F_2) where $F_0 : D_0 \to E_0$, $F_1 : D_1 \to E_1$ and $F_2 : D_2 \to E_2$ such that (F_0, F_1) is a functor between the 1-skeleton of D and E and (F_1, F_2) is a functor between both the vertical and horizontal category of D and E. That means that all faces and degeneracies commute with with F_1 resp. F_2.

Lemma 3.1.5. *Double functors turn the set of all double categories into a category* **DCat**. *Its initial object is the empty double category, its terminal object consists of the double category D with $D_0 = D_1 = D_2 = \{*\}$.* $\square$

3.2 Thin Structures and Connections

We will now enrich double category with even more data: We need a notion of what it means for a square to be *thin*. When defining the fundamental double category of a space these thin squares will correspond to those actual geometric squares inside the considered space, which are homotopic to degenerate squares.

Definition 3.2.1. Let D be a double category on a category $C = (C_0, C_1)$. Then, a **thin structure** on D is a double functor $T : \square C \to D$ which on the 1-skeleton is the identity.

Equivalently, we can describe a thin structure by marking certain two-cells as **thin** in way such that the following hold:

1. Every commutative shell has a unique thin filler.

2. The horizontal and vertical composition of two thin squares is thin.

3. Degenerate squares are thin.

In a double category D without a defined thin structure we have $\epsilon_1(f)$ and $\epsilon_2(f)$ as two ways to receive a square from a given morphism $f \in D_1$. A thin structure adds two more canonical ways of turning a line into a square:

Definition 3.2.2. Let D be a double category with a thin structure and $f \in D_1$ a morphism. Then, the **lower right connection** of f is the thin square with f as its upper and left face and $\epsilon(\partial^+(f))$ as its lower and right face. Analogously, the **upper left connection** of f is the thin square with f on the bottom and right face and $\epsilon(\partial^-(f))$ on the upper and left side. (See Figure 3.4.) We denote the lower right connection of f with $\Gamma^-(f)$ and the upper left connection with $\Gamma^+(f)$.

We observe that connections are composable in the following sense:

Lemma 3.2.3 (S-laws). *Let D be a double category with thin structure and $f \in D_1$. Then,*

$$\Gamma^-(f) \circ_2 \Gamma^+(f) = \epsilon_1(f) \textit{ and}$$
$$\Gamma^-(f) \circ_1 \Gamma^-(f) = \epsilon_2(f).$$

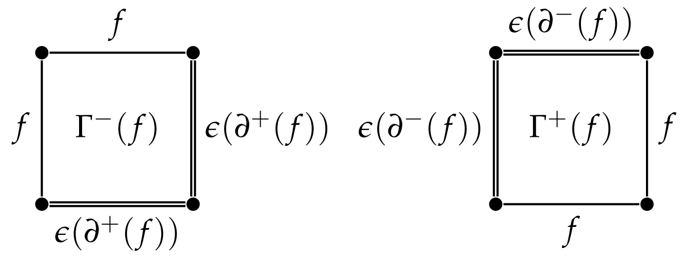

f
f
$\Gamma^-(f)$
$\epsilon(\partial^+(f))$
$\epsilon(\partial^+(f))$
$\epsilon(\partial^-(f))$
$\epsilon(\partial^-(f))$
$\Gamma^+(f)$
f
f

Figure 3.4: Lower right and upper left connection of a morphism f.

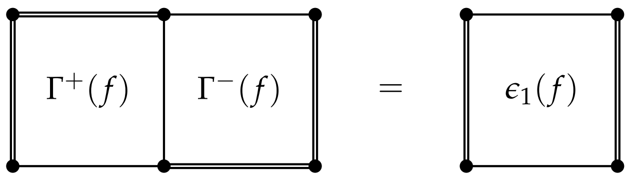


Figure 3.5: The horizontal S-law.

Lemma 3.2.4 (Transport laws). *Let D be a double category with thin structure and $f, g \in D_1$ with $\partial^+(f) = \partial^-(g)$. Then,*

$$\Gamma^-(g \circ f) = (\Gamma^-(g) \circ_2 \epsilon_2(g)) \circ_1 (\epsilon_1(g) \circ_2 \Gamma^-(f)) \text{ and}$$
$$\Gamma^+(g \circ f) = (\Gamma^+(g) \circ_2 \epsilon_1(f)) \circ_1 (\epsilon_2(f) \circ_2 \Gamma^+(f)).$$

Proof. As one can easily check (see Figures 3.5 – 3.7) for each of the equations, the composed squares are well defined and have the faces of the square on the left hand side coincide with those of the square on the right hand side. The composite squares are thin because they are a composition of thin squares. Then, both theorems follow from the uniqueness of thin fillers and the thinness of identity squares. □

3.3 Double Groupoids

In general, paths in topological spaces are non-oriented or can be reversed. Our algebraic structures describing paths and squares should reflect this behavior by the property that also its morphisms and two-cells should be reversible. Categories C in which for each morphism $f \in C_1$ there exists an inverse $f^{-1} \in C_1$ are called *groupoids*. We apply this concept not only to the 1-skeleton but also to the two-cells of a double category, which can be inverted vertically as well as horizontally.

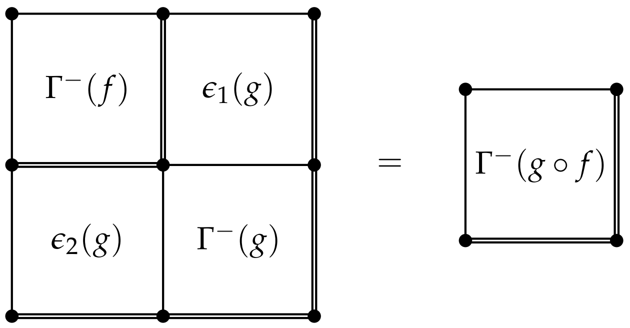


Figure 3.6: The transport law for $\Gamma^-(g \circ f)$.

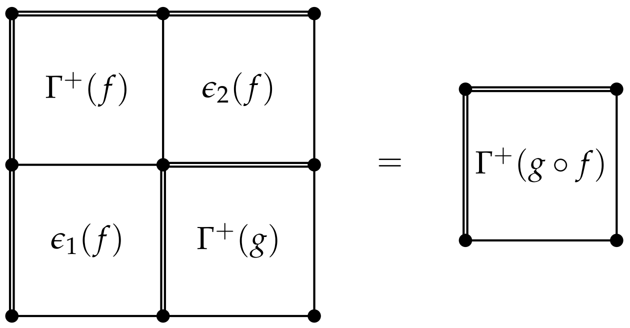


Figure 3.7: The transport law for $\Gamma^+(g \circ f)$.

Definition 3.3.1. A **weak double groupoid** is a double category D where all three categories — the 1-skeleton, the vertical category and the horizontal category — are groupoids. The inverses of a square $u \in D_2$ in the vertical and horizontal category will be denoted by inv_1 and inv_2 and can be seen as flipping the square along horizontal, resp. vertical, line.

For a double category to be a **double groupoid** we further require it to come with a fixed thin structure.

Note, that the three notions of inversion interact with each other by yielding the following laws:

Lemma 3.3.2 (Coherence of inverses). *Let D be a weak double groupoid, $a \in D_1$ and $u \in D_2$. Then,*

$$\begin{aligned}\epsilon_1(a) \circ_2 \epsilon_1(a^{-1}) = \epsilon_2(a) \circ_1 \epsilon_2(a^{-1}) = 0(\partial^-(a)),\\ \epsilon_1(a^{-1}) \circ_2 \epsilon_1(a) = \epsilon_2(a^{-1}) \circ_1 \epsilon_2(a) = 0(\partial^+(a)),\end{aligned} \tag{3.6}$$

$$\begin{aligned}\partial_1^-(\mathrm{inv}_2(u)) &= \partial_1^-(u)^{-1},\\ \partial_1^+(\mathrm{inv}_2(u)) &= \partial_1^+(u)^{-1},\\ \partial_2^-(\mathrm{inv}_1(u)) &= \partial_2^-(u)^{-1},\\ \partial_2^+(\mathrm{inv}_1(u)) &= \partial_2^+(u)^{-1},\end{aligned} \tag{3.7}$$

$$\begin{aligned}\epsilon_1(a^{-1}) &= \mathrm{inv}_2(\epsilon_1(a)), \textit{ and}\\ \epsilon_2(a^{-1}) &= \mathrm{inv}_1(\epsilon_2(a)).\end{aligned} \tag{3.8}$$

Proof. All equations follow from the fact that the face maps and degeneracies are homomorphic and thus also respect inverses. □

We can furthermore prove that our intuition is right assuming that horizontally inverting the vertical composition of squares is equal to composing the inverted squares:

Lemma 3.3.3 (Distributivity of inverses). *For a weak double groupoid D and $v, u \in D_2$ the following equations hold as soon as they are well defined:*

$$\begin{aligned}\mathrm{inv}_1(v \circ_2 u) &= \mathrm{inv}_1(v) \circ_2 \mathrm{inv}_1(u) \textit{ and}\\ \mathrm{inv}_2(v \circ_1 u) &= \mathrm{inv}_2(v) \circ_1 \mathrm{inv}_2(u).\end{aligned} \tag{3.9}$$

Proof. We only prove the first equation since the second one results from transposition of the situation and is provable analogously. Using the previous calculations and the interchange law we see that

$$\begin{aligned}(\mathrm{inv}_1(v) \circ_2 \mathrm{inv}_1(u)) \circ_1 (v \circ_2 u) =& (\mathrm{inv}_1(v) \circ_1 v) \circ_2 (\mathrm{inv}_1(u) \circ_1 u)\\ =& \epsilon_1(\partial_1^-(v)) \circ_2 \epsilon_1(\partial_1^-(u))\\ =& \epsilon_1(\partial_1^-(v) \circ \partial_1^-(u))\\ =& \epsilon_1(\partial_1^-(v \circ_2 u))\\ =& \mathrm{inv}_1(v \circ_2 u) \circ_1 (v \circ_2 u).\end{aligned}$$

Cancelling out $v \circ_2 u$ gives the desired result. □

Observing that we can "rotate" a square by 180 degrees in two ways, by first taking the vertical and then the horizontal inverse or vice versa, we can prove that those are actually one and the same:

Lemma 3.3.4. *For any weak double groupoid D and square $u \in D_2$,*

$$\mathrm{inv}_1(\mathrm{inv}_2(u)) = \mathrm{inv}_2(\mathrm{inv}_1(u)).$$

Proof. Similar to the last proof we calculate that

$$\begin{aligned}\mathrm{inv}_1(\mathrm{inv}_2(u)) \circ_2 \mathrm{inv}_1(u) &= \mathrm{inv}_1(\mathrm{inv}_2(u) \circ_2 u)\\ &= \mathrm{inv}_1(\epsilon_2(\partial_2^-(u)))\\ &= \epsilon_2(\partial_2^-(u)^{-1})\\ &= \mathrm{inv}_2(\mathrm{inv}_1(u)) \circ_2 \mathrm{inv}_1(u).\end{aligned}$$

□

Of course, the two basic examples we saw for double categories extend to double groupoids:

Lemma 3.3.5 (Square and shell double groupoids)**.** *If C is a groupoid, then the square double category $\Box' C$ and the shell double category $\Box C$ are double groupoids.*

Proof. It is easily seen for both cases that $\mathrm{inv}_1(f, g, h, i) := (g, f, h^{-1}, i^{-1})$ and $\mathrm{inv}_2(f, g, h, i) := (f^{-1}, g^{-1}, i, h)$ provide valid inverses. Thin squares in both double categories are exactly those in $(\Box C)_2$, which makes $\Box C$ a double groupoids with all squares thin. □

We will now take a look at the most important example of a double groupoid: The fundamental double groupoid of a triple of spaces. It extends and generalizes the definition of the fundamental group and the second homotopy group of a space. We start by extending the idea of the set of loops based at a point by allowing multiple base points and adding squares. Just like its classical homotopy theoretic counterpart, the following structure does *not* already define a double groupoid until we quotient out homotopy classes.

Definition 3.3.6 (Filtered maps)**.** Let $C \subseteq A \subseteq X$ be a nested triple of topological spaces. We obtain sets of points, lines and squares by defining

- $R(X, A, C)_0 := C$,
- $R(X, A, C)_1 := \{\sigma : (I, \partial I) \to (A, C)\}$, and
- $R(X, A, C)_2 := \{\alpha : (I^2, \partial I^2, \partial^2 I^2) \to (X, A, C)\}$, where

the maps are meant to be *based* in the sense that they should map the components of the indicated tuples to their counterparts. In other words, the points in $R(X, A, C)$ are the points in C , the lines are paths in A with endpoints located in C and the two-cells are squares in X which have faces in A and corners in C.

Note that $R(X, A, C)_1$ in the case of $C = \{*\}$ becomes the loop space based in $*$ and in the case of $C = A = \{*\}$ the elements of $R(X, A, C)_2$ end up to be maps $\mathbb{S}^2 \to X$ based in $*$.

Just as for a double groupoid we give faces, degeneracies, compositions and inversions. They might seem familiar from classical homotopy theory since they are just an extension of the operation that defines the loop group:

For a line $\sigma : (I, \partial I) \to (A, C)$, the left and right face are simply given by $\sigma(0)$ and $\sigma(1)$. For a square α we define $\partial_1^-(\alpha)(x) = \alpha(0, x)$, $\partial_1^+(\alpha)(x) = \alpha(1, x)$, $\partial_2^-(\alpha)(x) = \alpha(x, 0)$ and $\partial_2^+(\alpha)(x) = \alpha(x, 1)$.

Degenerate lines are constant maps $I \to C$, degenerate squares α are those where $\alpha(x, y) = \sigma(x)$ or $\alpha(x, y) = \sigma(y)$ for some line σ.

Composition of two composable squares α and β is defined analogously to the composition of paths in the loop group:

$$(\beta \circ_1 \alpha) = \begin{cases} \alpha(2x, y) & \text{if } 0 \leq x \leq \frac{1}{2}, \\ \beta(2x - 1, y) & \text{if } \frac{1}{2} \leq x \leq 1 \text{ as well as} \end{cases}$$

$$(\beta \circ_2 \alpha) = \begin{cases} \alpha(x, 2y) & \text{if } 0 \leq y \leq \frac{1}{2}, \\ \beta(x, 2y - 1) & \text{if } \frac{1}{2} \leq y \leq 1. \end{cases}$$

Inversion is given by $\sigma^{-1}(x) = \sigma(1-x)$ for a line σ and $\mathrm{inv}_1(\alpha)(x,y) = (1-x,y)$, $\mathrm{inv}_2(\alpha)(x,y) = (y,1-x)$ for a square α.

The next step is to define the fundamental double groupoid of a triple of spaces by modding out equivalence classes of filtered maps:

Definition 3.3.7. We define the **fundamental double groupoid** $\Pi_2(X,A,C)$ of a nested triple of spaces $C \subseteq A \subseteq X$ by defining

$$\begin{aligned}
&\Pi_2(X,A,C)_0 := C,\\
&\Pi_2(X,A,C)_1 := R(X,A,C)_1/\equiv, \text{ and}\\
&\Pi_2(X,A,C)_2 := R(X,A,C)_2/\equiv,
\end{aligned}$$

where in the first case $\equiv$ denotes homotopy rel vertices meaning that two paths σ and σ' are equivalent iff there is a homotopy $H : I^2 \to A$ that leaves both endpoints fixed:

$$\forall t \in I : H(t,0) = H(0,0) \in C \text{ and } H(t,1) = H(0,1) \in C.$$

For the squares, $\equiv$ we also require the considered homotopies $H : I^3 \to X$ to keep the corners fixed. But more, we require the homotopies *thin* in the sense that for all $x \in \partial I^2$ we have $H(t,x) \in A$ for all $t \in I$.

It can easily be seen that the operations defined on squares and lines in Definition 3.3.6 respect this definition of equivalence and that the resulting structure is, indeed, a weak double groupoid.

This is furthermore the point where the reason for the name “thin structure” becomes clear: To define a thin structure on $\Pi_2(X,A,C)$ we put the predicate “thin” on those equivalence classes of squares that contain a squre $\alpha : I^2 \to X$ where $\alpha(x,y) \in A$ for all $x,y \in I$. It is obvious that this property is closed under the composition of squares and it is true by definition that for all $\sigma \in \Pi_2(X,A,C)_1$ the condition is fulfilled for $\epsilon_1(\sigma)$ and $\epsilon_2(\sigma)$.

If for lines $f,g,h,i : I \to A$ we have $g \circ h = i \circ f$ in $\Pi_2(X,A,C)$, there is a homotopy $H : I^2 \to A$ rel vertices between $g \circ h$ and $i \circ f$. It is true that in this case we can also choose H such that f, g, h, and i lie on the four faces of a square. We define such a choice of H to be our thin filler for that quadruple of faces.

As a last step we have to prove that this choice is unique up to thin homotopy. We assume that we are not only given a thin square H for lines f, g, h, and i but also another thin square H' for a set of lines f', g', h',

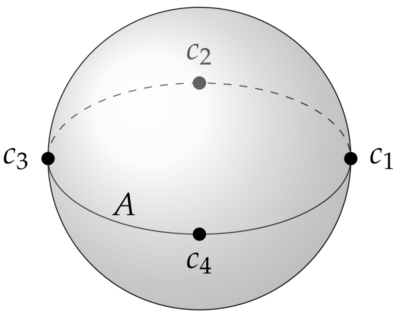
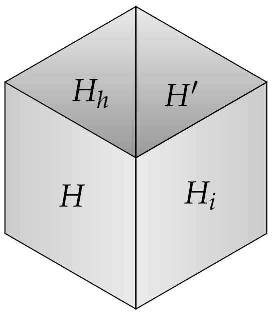


Figure 3.8: Filtered presentation of the sphere.

i', respectively equivalent to f, g, h and i. To see that H and H' can be "connected" with a thin homotopy, we consider them, and the homotopies H_g, H_h, and H_i between three of the faces, as five faces of a a cube:



We can fill this cube and receive a thin homotopy $I^3 \to X$ by simply choosing a point p above the cube and mapping each point in I^3 to its image under the projection onto the box centered in p.

Before we move on to introduce another algebraic structure that is useful for the analysis of the first and second homotopy group of a space, here is an example for a space and its fundamental double groupoid:

Example 3.3.8 (The fundamental double groupoid of the sphere). Let $X := S^2$ be the 2-sphere, $A \subset X$ its equator and $C = \{c_1, c_2, c_3, c_4\} \subset A$ four points on the equator (see Figure 3.8).

Then, $\Pi_2(X, A, C)$ has C as point set, lines are generated by the segments $\overline{c_0c_1}$, $\overline{c_1c_2}$, $\overline{c_2c_3}$ and $\overline{c_3c_4}$. All two-cells are the result of composition of the upper and lower hemisphere and degenerate squares on the equator.

Definition 3.3.9 (Category of double groupoids). Since morphisms between groupoids are nothing but functors between their underlying categories we

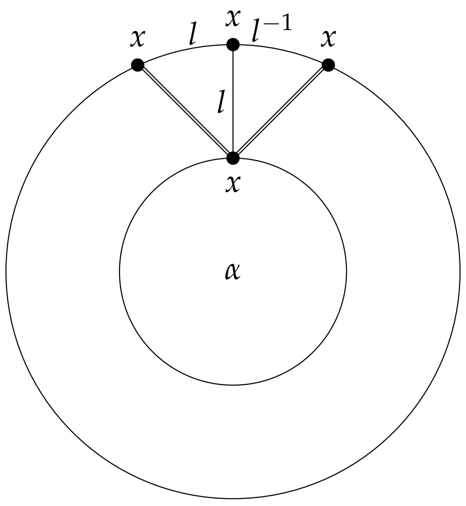


Figure 3.9: Pasting a loop l to a disk α.

can, without changing the definition of morphisms, enhance our category **DCat** of double categories to a **category of weak double groupoids**. This category is thus a full subcategory of **DCat**.

But to a greater degree we are interested in the **category of double groupoids** : This is the category **DGpd** containing as objects all double groupoids and as morphisms all double functors that preserve the attached thin structure in the sense that they map thin squares to thin squares.

3.4 Crossed Modules

The motivation to introduce crossed modules as a tool for the analysis of the homotopy properties of a topological space comes from observing that in the long exact sequence of relative homotopy groups for the constellation $x \in A \subseteq X$, the second relative homotopy group $\pi_2(X, A, x)$ and the fundamental group $\pi_1(A, x)$ of the subspace are related in the following two ways:

1. There is a boundary map $\pi_2(X, A, x) \to \pi_1(A, x)$ which is induced by mapping a representative $\alpha : (\mathbb{D}^2, \partial\mathbb{D}^2) \to (X, A)$ to its restriction to the boundary $\alpha|_{\partial\mathbb{D}^2} : \mathbb{S}^1 \to A$.

2. The group $\pi_1(A, x)$ acts on $\pi_2(X, A, x)$ by "glueing" the representative l of $\pi_1(A, x)$ on the disk that represents the given element $[\alpha] \in$

$\pi_2(X, A, x)$. We receive the resulting disk by extending α in a degenerate way along l as illustrated in Figure 3.9.

These two means of interaction can furthermore be observed to fulfil more algebraic requirements that will be captured in the definition of a crossed module:

- The boundary of a representative which was created by pasting $l \in \pi_1(A, x)$ to a disk $[\alpha] \in \pi_2(X, A, x)$ is the concatenation of paths $l^{-1} \cdot \partial\alpha \cdot l$.
- The disk resulting from pasting the boundary of a disk $\beta \in \pi_2(X, A, x)$ to a disk $\alpha \in \pi_2(X, A, x)$ is homotopic to the composition $\beta^{-1} \cdot \alpha \cdot \beta$ in $\pi_2(X, A, x)$.

So the boundary map and the action resemble the *conjugation* of group elements in two different ways. This motivates bundling these properties into a new algebraic structure:

Definition 3.4.1 (Crossed module over a group)**.** Let P be a group. A **crossed module** on P is another group M together with a group homomorphism $\mu : M \to P$ and a group action ϕ of P on M such that:

1. For all $a \in P, x \in M$:
$$\mu(\phi(a, x)) = a \cdot \mu(x) \cdot a^{-1}. \tag{3.10}$$
2. For all $x, c \in M$:
$$\phi(\mu(c), x) = c \cdot x \cdot c^{-1}. \tag{3.11}$$

Crossed modules are not only motivated by geometric examples but also used to capture a very common, purely group theoretic configuration:

Example 3.4.2 (Normal subgroup crossed module)**.** Let G be a group and $N \subseteq G$ a normal subgroup of G. Then N is made a crossed module on G by the inclusion map $i : N \to G$ and the conjugation action of G on N.

Now we are not interested in the absolute and relative homotopy groups based in one point but want to adapt the structure to fit well to the concept of fundamental *groupoids* instead of fundamental groups. This obvious choice to achieve this is, unsurprisingly, to replace the base group P in the definition of a crossed module by a groupoid:

Definition 3.4.3 (Crossed module over a groupoid)**.** Let P be a groupoid. A **crossed module** over P is a group M_p for every $p \in P$ together with with a family of group homomorphisms $(\mu_p : M_p \to \hom_P(p, p))_{p \in P}$ and a map ϕ which is a *groupoid action* of P on M in the sense that it maps a pair (a, x), where $a \in \hom_P(p, q)$ and $x \in M_p$, to an element of M_q, such that $\phi(\mathrm{id}_p, x) = x$, $\phi(b \circ_P a, x) = \phi(b, (\phi(a, x))$ and $\phi(a, y \cdot_{M_p} x) = \phi(a, y) \cdot_{M_q} \phi(a, x)$ for all $p, q, r \in P, x, y \in M_p, a \in \hom_P(p, q)$ and $b \in \hom_P(q, r)$.

And, just in the case of crossed modules over a group, we require μ and ϕ to fulfil the following two essential equations:

1. For all $a \in \hom_P(p, q)$ and $x \in M_p$:

$$\mu_q(\phi(a, x)) = a \circ \mu_p(x) \circ a^{-1} \in \hom_P(q, q). \qquad (3.12)$$

2. For all $c, x \in M_p$:

$$\phi(\mu_p(c), x) = c \cdot x \cdot c^{-1} \in M_p. \qquad (3.13)$$

With this definition we can extend the example from the beginning of this chapter by allowing not only x as an endpoint of the paths considered but every point in a set $C \subseteq A$. This generalization results in the definition of the **fundamental crossed module** of a triple of spaces $C \subseteq A \subseteq X$.

To make the set of crossed module a category, we need to define what a morphism between two crossed modules should be:

Definition 3.4.4 (Morphisms between crossed modules)**.** Let $(M_p)_{p \in P}$ be a crossed module over a groupoid P, with homomorphism μ and action ϕ and let $(N_q)_{q \in Q}$ be a crossed module over Q with morphism μ' and action ϕ'. A **morphism** between $(M_p)_{p \in P}$ and $(N_q)_{q \in Q}$ is a functor F between P and Q and a family of group homomorphisms $(\psi_p)_{p \in P}$ with $\psi_p : M_p \to N_{F(p)}$ such that

- $F \circ \mu_p = \mu'_{F(p)} \circ \psi_p$ for all $p \in P$ and
- for all $p, q : P$, $a \in \hom_P(p, q)$ and $x : M_p$, the action is preserved:

$$\psi_q(\phi(a, x)) = \phi(F(a), \psi_p(x)).$$

Definition 3.4.5 (Category of crossed modules)**.** All crossed modules on groupoids form a category **XMod** using the previously defined morphisms. Crossed modules over groups are a full subcategory of **XMod**.

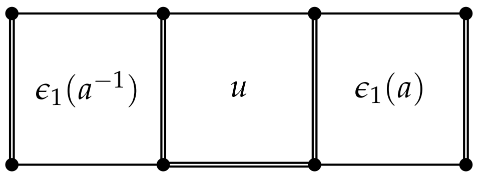


Figure 3.10: The result $\phi(a,u) \in M_q$ of a morphism $a \in \hom_P(p,q)$ acting on a two-cell $u \in M_p$ three of whose faces are degenerate.

3.5 Double Groupoids and Crossed Modules are Equivalent

Comparing the fundamental crossed module and the fundamental double groupoid we observe that these two structures contain basically the same information. In fact, not only those particular examples do, but we can prove that the categories **DGpd** and **XMod** are equivalent.

In this chapter, the functors $\gamma : \mathbf{DGpd} \to \mathbf{XMod}$ and $\lambda : \mathbf{XMod} \to \mathbf{DGpd}$ will be presented as well as a proof that $\gamma\lambda$ and $\lambda\gamma$ are naturally isomorphic to the respective identity functors.

Lemma 3.5.1 (The crossed module associated to a double groupoid). *Let G be a double groupoid. We set*

$$\begin{aligned} P &:= (G_0, G_1) \text{ and} \\ M_p &:= \left\{ u \in G_2 \middle| \partial_1^+(u) = \partial_2^-(u) = \partial_2^+(u) = \epsilon(p) \right\} \text{ for } p \in G_0. \end{aligned}$$

Then, M_p is a group with composition $\circ_2$, neutral element $0(p)$ and inverse inv_2. Let further be $\mu = \partial_1^-$ and let $\phi(a,u) = \epsilon_1(a) \circ_2 u \circ_2 \epsilon_1(a^{-1}) \in M_q$ for $a \in$ $\hom_P(p,q)$ and $u \in M_p$.

The given data P, $(M_p)_{p\in P}$, μ and ϕ form a crossed module γG.

Proof. We easily check that $\mu(\phi(a,u)) = a \circ u \circ a^{-1}$ for any $u \in M_p$ and $a \in hom_P(p,q)$. To see, that $\phi(\mu_p(c),u) = c \circ_2 u \circ_2 c^{-1}$ for $u, c \in M_p$, we consider the composite square

$$(c \circ_2 0(p) \circ_2 \mathrm{inv}_2(c)) \circ_1 \left(\epsilon_1(\partial_1^-(c)) \circ_2 u \circ_2 \partial_1^-(c^{-1})\right),$$

which, when evaluated as is, resolves to $\phi(\mu_p(c),u)$, but after applying the interchange law twice becomes

$$(c \circ_1 \epsilon_1(\partial_1^-(c))) \circ_2 (0(p) \circ_1 u) \circ_2 \left(\mathrm{inv}_2(c) \circ_1 \partial_1^-(c^{-1})\right)$$

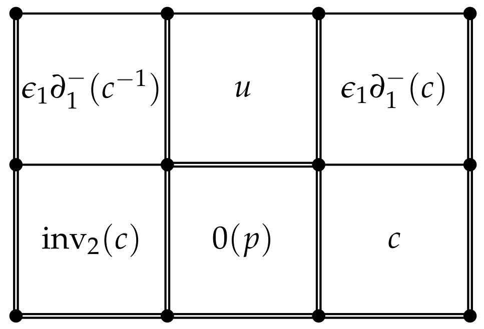


Figure 3.11: A composite square proving the second crossed module axiom for γG.

and can be simplified to $c \circ_2 u \circ_2 c$ (see Figure 3.11). □

Definition 3.5.2. The construction of γG can be extended to a map from double functors to morphisms of crossed modules and we obtain a functor γ : **DGpd** $\to$ **XMod**.

But how can we recover a double groupoid given a crossed module on a groupoid?

Lemma 3.5.3 (The double groupoid associated to a crossed module)**.** *Let $(M_p)_{p\in P}$ be a crossed module on a groupoid P. We define*

$$G_2 := \left\{(f,g,h,i,m)\middle|\mu(m) = i \circ f \circ h^{-1} \circ g^{-1}\right\}.$$

Then, G_2 forms the set of two cells of a double groupoid $\lambda(P,(M_p))$ with the first four projections as face maps.

Proof. Defining the vertical composition of squares $u = (f, g_1, h_1, i_1, m)$ and $v = (g_1, g_2, h_2, i_2, n)$ as $v \circ_1 u := (f, g_2, h_2 \circ h_1, i_2 \circ i_2, \phi(i_2, m) \cdot n)$ is well-defined since

$$\begin{aligned}\mu(\phi(i_2,m)\cdot n) &= i_2 \circ \mu(m) \circ i_2^{-1} \circ \mu(n)\\ &= i_2 \circ i_1 \circ f \circ h_1^{-1} \circ g_1^{-1} \circ i_2^{-1} \circ i_2 \circ g_1 \circ h_2^{-1} \circ g_2^{-1}\\ &= (i_2 \circ i_1) \circ f \circ (h_2 \circ h_1)^{-1} \circ g_2^{-1}\end{aligned}$$

and the horizontal composition of $u = (f_1, g_1, h, i_1, m)$ and $v = (f_2, g_2, i_1, i_2, n)$ likewise by $v\circ_2 := (f_2 \circ f_1, g_2 \circ g_1, h, i_2, n \cdot \phi(g_2, m))$ with

$$\begin{aligned}\mu(n\cdot\phi(g_2,m)) &= \mu(n) \circ g_2 \circ \mu(m) \circ g_2^{-1}\\ &= i_2 \circ f_2 \circ i_1^{-1} \circ g_2^{-1} \circ g_2 \circ i_1 \circ f_1 \circ h^{-1} \circ g_1^{-1} \circ g_2^{-1}\\ &= i_1 \circ (f_2 \circ f_1) \circ h \circ (g_2 \circ g_1)^{-1}\end{aligned}$$

Identity squares are given by $(f, f, \epsilon(p), \epsilon(q), 1)$ and $(\epsilon(p), \epsilon(q), f, f, 1)$ for $f \in \hom_P(p, q)$. Identity and associativity laws follow easily from the fact that ϕ respects identity morphisms as well as the composition in M_p and in P. □

Definition 3.5.4. The map λ can be turned into a functor $\lambda : \mathbf{XMod} \to \mathbf{DGpd}$ by extending it to morphisms of crossed modules.

We conclude this chapter by stating:

Theorem 3.5.5. *The categories* **DGpd** *and* **XMod** *are equivalent.*

Proof. We show that there are isomorphic natural transfromations $\gamma\lambda \simeq \mathrm{id}_{\mathbf{XMod}}$ and $\lambda\gamma \simeq \mathrm{id}_{\mathbf{DGpd}}$.

These transformation are just the identity on points an morphisms so we only have to consider their definition on two-cells. For a crossed module $(M_p)_{p \in P}$,

$$\begin{aligned}\gamma(\lambda(M_p)) &= \left\{(f, g, h, i, m) \;\middle|\; \mu(m) = i \circ f \circ h^{-1} \circ g^{-1}, g = h = i = \mathrm{id}_p\right\} \\ &= \{(f, \mathrm{id}_p, \mathrm{id}_p, \mathrm{id}_p, m) \mid \mu(m) = f\} \\ &\cong M_p.\end{aligned}$$

It is easy to check that this isomorphism of groups extends to an isomorphism of crossed modules and that it is natural in $(M_p)_{p \in P}$.

To find a suitable natural isomorphism $\mathrm{id}_{\mathbf{DGpd}} \simeq \lambda\gamma$, we have to match the structure of two-cells in a double groupoid, which are compsable in two dimensions, to the one-dimensional nature of a group: Assume a double groupoid G. We have to map each two-cell $u \in G_2$ with faces f, g, h, and i to some tuple (f, g, h, i, m) where m is a square with identity on all but its top face. But how do we turn a square with arbitrary faces to a square with such constraints in a revertible way? It turns out that we can use the connections (Definition 3.2.2) to define such a *folding* Φ of squares (cf. Figure 3.12):

$$\Phi(u) := \Gamma^-(i) \circ_2 u \circ_2 \mathrm{inv}_2(\Gamma^-(h)) \circ_2 \mathrm{inv}_2(\mathrm{id}_1(g))$$

ϕ is bijective since we can just rewrite the previous equation to

$$u = \mathrm{inv}_2(\Gamma^-(i)) \circ_2 \Phi(u) \circ_2 \mathrm{id}_1(g) \circ_2 \Gamma^-(h)$$

Calculations tell us furthermore that Φ is a homomorphism. Φ preserves the thin structure on G because it maps every thin square u to the zero

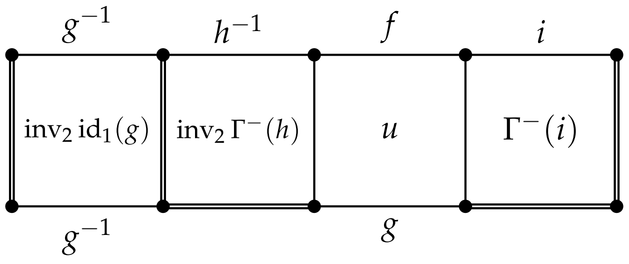

g^{-1}
h^{-1}
f
i
$\mathrm{inv}_2\,\mathrm{id}_1(g)$
$\mathrm{inv}_2\,\Gamma^-(h)$
u
$\Gamma^-(i)$
g^{-1}
g

Figure 3.12: Folding the faces of a square u to its upper face.

square based in the lower right corner of u. This can be proved by showing that all thin squares are compositions of identity squares and connections and the fact that these get mapped to zero. Thus, we obtain a bijective double functor between G and $\lambda(\gamma(G))$, whose naturality in G is shown by a proof we omit for the sake of brevity. □

Chapter 4

Translation and Use in Homotopy Type Theory

In this this chapter, I will describe how the structures introduced in the previous chapter can be translated to homotopy type theory. Besides formulating the concepts using dependent types this involves caring about the effects of univalence and proof relevance on these definitions. What does it mean for two instances of a structure to be propositionally equal? What truncation levels should be imposed on the parameters of said structure such that algebraic structures bear no unwanted information in their iterated equality types?

We start off with defining the notion of a categories, and then continue to translate the definitions from the previous chapter appropriately. Finally we will show how to apply the definitions to 2-types and define the fundamental double groupoid and fundamental crossed module of a presented type.

The standard references for the implementation of categories I will use in this chapter are the respective chapter of the HoTT-Book [Uni13] as well as the paper about “Univalent Categories and the Rezk Completion” by Benedikt Ahrens, Chris Kapulkin and Mike Shulman [AKS13]. While most of the time I will stick to the (consistent) notation and terminology of both of these, I will deviate sometimes to bring the presentation more in line with the actual Lean implementation presented in the next chapter.

4.1 Categories in Homotopy Type Theory

Definition 4.1.1 (Precategory). Let $A : \mathcal{U}$ be a type (the **object type** or **carrier**). A **precategory** C on A is constructed by giving the following data:

- For each $a, b : A$ a type of morphisms $\hom_C(a,b) : \mathcal{U}$ for which we furthermore require that $\prod_{(a,b:A)} \mathsf{isSet}(\hom_C(a,b))$.
- The composition of morphisms
$$\mathrm{comp}_C : \prod_{a,b,c:A} \hom_C(b,c) \to \hom_C(a,b) \to \hom_C(a,c).$$
We will most of the leave the first three arguments implicit and just write $g \circ_C f$ or gf for $\mathrm{comp}_C(a,b,c,g,f)$.
- An identity operator $\mathrm{id}_C : \prod_{(a:A)} \hom_C(a,a)$.
- A witness ensuring associativity for all morphisms:
$$\prod_{a,b,c,d:A} \prod_{h:\hom_C(c,d)} \prod_{g:\hom_C(b,c)} \prod_{f:\hom_C(a,b)} h \circ_C (g \circ_C f) = (h \circ_C g) \circ_C f$$
- Witnesses that the identity morphisms are neutral with respect to composition from the left and from the right:
$$\prod_{a,b:A} \prod_{f:\hom_C(a,b)} (\mathrm{id}_C(b) \circ_C f = f) \times (f \circ_C \mathrm{id}(a) = f)$$

As with all the definitions given in this semi-informal style, it is, from a theoretical standpoint, equivalent whether to see them as a description of an iterated Σ-Type or as the only constructor of an inductive type. We will later (Section 5.3) see that in formalization practice it is favorable to choose to introduce them as new inductive types instead of Σ-Types.

We also observe that, since equalities in sets are mere propositions, we have the following lemma:

Lemma 4.1.2 (Equality of precategories). *Let C and D Precategories on A with* $\hom_D :\equiv \hom_C$, $\mathrm{comp}_C = \mathrm{comp}_D$ *and* $\mathrm{id}_C = \mathrm{id}_D$. *Then, $C = D$.* $\square$

This also justifies the fact that we do not require further coherence conditions on associativity (the "pentagon coherence law") and identity laws.

Definition 4.1.3 (Functors)**.** Let C_A and C_B be precategories on types A and B. A **functor** F between C_A and C_B is constructed by giving the following:

- Its definition on objects as an instance of $F_{\text{obj}} : A \to B$.
- Its definition on morphisms

  $$F_{\text{hom}} : \prod_{a,b:A} \prod_{f:\text{hom}_{C_A}(a,b)} \text{hom}_{C_B}(F_{\text{obj}}(a), F_{\text{obj}}(b)).$$

  Again we will often leave out the first two arguments for F_{hom} and moreover abbreviating F_{hom} and F_{obj} to F whenever the distinction is clear.
- A proof in $\prod_{(a:A)} F(\text{id}_{C_A}(a)) = \text{id}_{C_B}(F(a))$ that the identies are preserved and
- a proof the respects the composition in the respective categories, as an instance of

  $$\prod_{a,b,c:A} \prod_{g:\text{hom}_{C_A}(b,c)} \prod_{f:\text{hom}_{C_B}(a,b)} F(g \circ_{C_A} f) = F(g) \circ_{C_B} F(f)$$

Again, the last two ingredients turn out to be mere propositions, as they are of Π-types over equalities in sets. This leads us to the observation that to prove the equality of two functors, it suffices to check it on their definitions on objects and morphisms:

Lemma 4.1.4 (Equality of functors)**.** *Let $A, B : \mathcal{U}$ and let C, D be categories on A and B, respectively. Let F and G be two functors from C to D. If we have*

$$\begin{aligned} p &: \prod_{a:A} F(a) = G(a) \text{ and} \\ q &: \prod_{a,b:A} \prod_{f:\text{hom}_C(a,b)} p(b)_* \left(p(a)_*(F(f))\right) = G(f), \end{aligned}$$

then $F = G$. □

With this definition of precategories and functors, a lot of structures can be instantiated as such. For example, the 1-types of a universe $\mathcal{U}_i$ give us a precategory with morphisms between $A, B : \mathcal{U}_i$ being $A = B$, composition being concatenation of equalities and identity being reflexivity. But often we will only have to deal with precategories whose carrier is a set.

Definition 4.1.5 (Strict precategory)**.** A precategory with a set as carrier is called **strict**.

One primary use for strict precategories is the following: If we wanted to build a precategory of precategories we encounter the problem that functors between two given precategories don't generally form a set! Restricting ourselves to strict precategories solves this problem:

Lemma 4.1.6. *Let C be a precategory and D be a strict precategory. Then, the type of functors between C and D forms a set.*

Proof. For the type of functors to be a set, all parameters should be sets. Since the definition on morphisms is a set by definition and, as we already observed, the identity witnesses are mere propositions, the only critical parameter is the object function. But turning the codomain of this function into a set obviously solves the problem. □

Corollary 4.1.7. *For each pair of universes $(\mathcal{U}_i, \mathcal{U}_j)$ there is a precategory of strict precategories with carrier in $\mathcal{U}_i$ and morphism types in $\mathcal{U}_j$. Morphisms of this category are functors, with composition of functors and identity functors being defined as obvious.* □

Another precategory we can consider is the precategory of sets:

Lemma 4.1.8. *Let $\mathcal{U}_i$ be a universe. Then, the 1-type of sets in $\mathcal{U}_i$ forms a precategory with morphisms being arbitrary functions.*

But the truncation level of the carrier is not the only thing that could be bothersome when dealing with precategories: if we look at the previous two examples of precategories where each object itself are types, we can conclude that if two objects are isomorphic in the algebraic sense they are equivalent types and thus, by the univalence axiom, equal. We want to transfer this as a requirement to all categories, extending the general idea of univalence which says that isomorphic objects should be treated as equals. To make this definition of a *category* as smooth as possible, some auxiliary definitions will be necessary.

Definition 4.1.9 (Isomorphisms)**.** Let C be a precategory on a type $A : \mathcal{U}$. A morphism $f : \hom_C(a, b)$ is called an **isomorphism** if there is a morphism $g : \hom_C(b, a)$ that is both a left and a right inverse to f. Define furthermore

$a \cong b$ for $a, b : A$ as the type of all isomorphisms in $\mathrm{hom}_C(a, b)$. Formally:

$$\begin{aligned} \mathsf{isIso}_C(f) &:\equiv \sum_{g:\mathrm{hom}_C(b,a)} (g \circ_C f = \mathrm{id}_C(a)) \times (f \circ_C g = \mathrm{id}_C(b)) \\ a \cong b &:\equiv \sum_{f:\mathrm{hom}_C(b,a)} \mathsf{isIso}_C(f). \end{aligned}$$

Lemma 4.1.10. *For each $a, b : A$, $a \cong b$ is a set and for each $f : \mathrm{hom}_C(a, b)$, $\mathsf{isIso}(f)$ is a mere proposition.* □

Equal objects of a category are always isomorphic:

Lemma 4.1.11. *If for two objects $a, b : A$ of a precategory C we have $p : a = b$, then there exists*

$$\mathsf{idtoiso}_{a,b}(p) : a \cong b,$$

such that $\mathsf{idtoiso}_{a,a}(\mathsf{refl}_a) = (\mathrm{id}_C(a), \ldots)$.

Proof. Induction on p lets us assume that $p \equiv \mathsf{refl}_a$. But since $\mathrm{id}_C(a) \circ_C \mathrm{id}_C(a) = \mathrm{id}_C(a)$ by either of the cancellation laws for identities, we get an instance of $a \cong a$. □

Definition 4.1.12 (Category)**.** A **category** C on A is defined to be a precategory on A which is *univalent*: it is accompanied by a proof that for all $a, b : A$, $\mathsf{idtoiso}_{a,b}$ is an equivalence.

The essential use of this extension is, of course, the inverse that idtoiso is assumed to have: Statements about isomorphic objects become statements about equal objects and can thus be proven using induction. There are some important examples for univalent categories:

Lemma 4.1.13. *Precategories on $\mathcal{U}$ and precategories on subtypes of $\mathcal{U}$ (i.e. on $\sum_{(A:\mathcal{U})} P(A)$ where $P : \mathcal{U} \to \mathcal{U}$ and $\prod_{(A:\mathcal{U})} \mathsf{isProp}(P(A))$) with function types as morphisms are univalent.* □

Lemma 4.1.14. *The precategory of strict precategories is univalent.*

Proof. If two precategories are isomorphic, the functors witnessing this relation restrict to an equivalence on the carrier which, by univalence, gives us an equality between the carriers. Transported along this equality the functors also give an equivalence between the morphism types of two objects. Again, we use univalence, this time to gain an equality between the sets of morphisms. By lemma we conclude that two isomorphic precategories are equal. □

Lemma 4.1.15. *If there is a univalent category on a type $A : \mathcal{U}$, then A is a 1-type.*

Proof. For all $a, b : A$, the fact that $a \cong b$ is a set implies that, since truncation levels are preserved by equivalences, also $a = b$ is a set. But by definition, this proves that A is 1-truncated. $\square$

A last definition is the one of a groupoid. Here, the univalent case is rather uninteresting since the structure of the groupoid is completely determined by the carrier.

Definition 4.1.16 ((Pre-)Groupoids)**.** A **(pre-)groupoid** is a (pre-)category C on $A : \mathcal{U}$ together with

$$\mathsf{alliso}_C : \prod_{a,b:A} \prod_{f:\hom_C(a,b)} \mathsf{isIso}(f).$$

4.2 Double groupoids in Homotopy Type Theory

As we saw when introducing (pre-)categories in homotopy type theory, we use dependent types to model the type of morphisms. Instead, we could have said that a category is defined on two types $A, H \in \mathcal{U}$ where A represents the objects and H is *one* type of morphisms. This would have required us to give functions $\partial^-, \partial^+ : H \to A$ specifying domain and codomain of morphisms and the composition to be of the type

$$\prod_{g,f:H} \left(\partial^+(f) = \partial^-(g) \to \sum_{h:H} \left(\partial^-(h) = \partial^-(f)\right) \times \left(\partial^+(h) = \partial^+(g)\right) \right).$$

The difference between this approach and the one we chose is that morphisms are only composable if their respective codomain and domain are *definitionally* the same. Each approach is advantageous in some situations: Using dependent types will force us more often to use transports to achieve definitional equality on codomain and domain, using ∂^- and ∂^+ requires an identity proof for each composition we want to construct.

For the implementation of double categories and double groupoids I decided to adopt the dependently typed concept that is commonly also used to formalize categories. Also note that in the following definition we will *not* require the 1-skeleton of a double category to be univalent since in our intended main use of the structures the object type will not be 1-truncated and thus the 1-skeleton, as seen in Lemma 4.1.15, not a univalent category.

Definition 4.2.1 (Double category)**.** A **double category** D is constructed by giving the following components:

- A *set* $D_0 : \mathcal{U}$ of objects.
- A precategory C on D_0. To be consistent with the notation in the last chapter, we will write $D_1(a, b) : \mathcal{U}$ for $\hom_C(a, b)$.
- A dependent type of two-cells:

  $$D_2 : \prod_{a,b,c,d:D_0} \prod_{f:D_1(a,b)} \prod_{g:D_1(c,d)} \prod_{h:D_1(a,c)} \prod_{i:D_1(b,d)} \mathcal{U}$$

  We will always leave the first four parameters implicit and write $D_2(f, g, h, i)$ for the type of two-cells with f as their upper face, g as their bottom face, h as their left face, and i as their right face.
- The vertical composition operation: For all $a, b, c_1, d_1, c_2, d_2 : D_0$ and $f_1 : D_1(a, b), g_1 : D_1(c_1, d_1), h_1 : D_1(a, c_1), i_1 : D_1(b, d_1), g_2 : D_1(c_2, d_2)$, $h_2 : D_1(c_1, c_2)$, and $i_2 : D_1(d_1, d_2)$ the composition of two cells

  $$D_2(g_1, g_2, h_2, i_2) \to D_2(f, g_1, h_1, i_1) \to D_2(f_1, g_2, h_2 \circ h_1, i_2 \circ i_1).$$

  As this is pretty verbose, we will, from now on, refrain from writing out parameters that only serve to can easily be inferred from the rest of the term.

  We will denote the vertical composition of $v : D_2(g_1, g_2, h_2, i_2)$ and $u : D_2(f, g_1, h_1, i_1)$ with $v \circ_1 u$ leaving all other information implicit.
- The vertical identity $\mathrm{id}_1 : \prod_{(a,b:D_0)} \prod_{(f:D_1(a,b))} D_2(f, f, \mathrm{id}_C(a), \mathrm{id}_C(b))$.
- For all $w : D_2(g_2, g_3, h_3, i_3), v : D_2(g_1, g_2, h_2, i_2)$, and $u : D_2(f, g_1, h_1, i_1)$ a witness for the associativity of the vertical composition $\mathsf{assoc}_1(w, v, u)$ in

  $$\mathsf{assoc}(i_3, i_2, i_1)_*(\mathsf{assoc}(h_3, h_2, h_1)_*(w \circ_1 (v \circ_1 u))) = (w \circ_1 v) \circ_1 u,$$

  where assoc is the associativity proof in the 1-skeleton. The transport is required since the cells at the left and right side of the equation do not definitionally have the same set of faces.

- For every $u : D_2(f,g,h,i)$ we need proofs $\mathsf{idLeft}_1(u)$ and $\mathsf{idRight}_1(u)$ that the following equations hold:

$$\begin{aligned}\mathsf{idLeft}(i)_*(\mathsf{idLeft}(h)_*(\mathrm{id}_1 \circ_1 u)) &= u \text{ and}\\ \mathsf{idRight}(i)_*(\mathsf{idRight}(h)_*(\mathrm{id}_1 \circ_1 u) &= u.\end{aligned}$$

  Again, the transports are needed to account for the difference in the faces of the two squares we compare.

- Of course, we need a horizontal composition and identity:

$$\begin{aligned}\circ_2 &: \prod_{\dots} D_2(f,g,h,i) \to D_2(f_2,g_2,i,i_2) \to D_2(f_2 \circ f, g_2 \circ, h, i_2)\\ \mathrm{id}_2 &: \prod_{a,b:D_0} \prod_{f:D_1(a,b)} D_2(\mathrm{id}_C(a), \mathrm{id}_C(b), f, f)\end{aligned}$$

- Analogously to the ones above, we need the associativiy and identitiy proofs for the horizontal composition:

$$\begin{aligned}\mathsf{assoc}_2 &: \mathsf{assoc}(g_3,g_2,g_1)_*(\mathsf{assoc}(f_3,f_2,f_1)_*(w \circ_2 (v \circ_2 u)))\\ &\qquad = (w \circ_2 v) \circ_1 u,\\ \mathsf{idLeft}_2 &: \mathsf{idLeft}(g)_*(\mathsf{idLeft}(f)_*(\mathrm{id}_2 \circ_2 u)) = u, \text{ and}\\ \mathsf{idRight}_2 &: \mathsf{idRight}(g)_*(\mathsf{idRight}(f)_*(\mathrm{id}_2 \circ_2 u) = u,\end{aligned}$$

  for every $u : D_2(f,g,h,i)$, $v : D_2(f_2,g_2,i,i_2)$, and $w : D_2(f_3,g_3,i_2,i_3)$.

- For every f, g, h, and i, the type of two-cells $D_2(f,g,h,i)$ must be a set.

- The identities should distribute over the respective other composition (compare (3.4)):

$$\begin{aligned}&\prod_{a,b,c:D_0} \prod_{f:D_1(a,b)} \prod_{g:D_1(b,c)} \mathrm{id}_2(g \circ f) = \mathrm{id}_2(g) \circ_1 \mathrm{id}_2(f)\\ &\prod_{a,b,c:D_0} \prod_{f:D_1(a,b)} \prod_{g:D_1(b,c)} \mathrm{id}_1(g \circ f) = \mathrm{id}_1(g) \circ_2 \mathrm{id}_1(f)\end{aligned}$$

- Corresponding to the equation (3.3), we need a proof that there is only one, unique, zero-square for each point:

$$\prod_{a:D_0} \mathrm{id}_1(\mathrm{id}_C(a)) = \mathrm{id}_2(\mathrm{id}_C(a))$$

- Finally, as introduced in equation (3.5), the interchange law should hold:

$$\prod_{\dots} (x \circ_2 w) \circ_1 (v \circ_2 u) = (x \circ_1 v) \circ_2 (w \circ_1 u)$$

  Consider that here, the "…" indexing the iterated Π-type hide a list of 9 points in D_0, 12 morphisms and four squares. We will continue to hide the indexing of two-cells for the sake of readability.

This list of necessary parameters of a constructor to a double category might seem long at first glance, but the fact that we implemented the two-cells as dependent types released us from the duty of adding the cubical identities (3.1) and (3.2) propositionally. Not only those, but also the four first equations in (3.4) hold by definition! Formulating the definition using three different categories would have rendered this impossible. But the notion of the two categories of two-cells is still accessible in the formalization:

Definition 4.2.2. We can recover what in Definition 3.1.1 we called the **vertical precategory** of a double category D as a category V on the type $A :\equiv \sum_{(a,b:D_0)} D_1(a,b)$ with morphisms

$$\hom_V((a,b,f),(c,d,g)) :\equiv \sum_{h:D_1(a,c)} \sum_{i:D_1(b,d)} D_2(f,g,h,i)$$

and composition $(h_2, i_2, v) \circ_V (h_1, i_1, u) :\equiv (h_2 \circ h_1, i_2 \circ i_1, v \circ_1 u)$. The corresponding category axioms can easily be proved to follow from the ones of the 1-skeleton of D and their counterparts in Definition 4.2.1.

The **horizontal precategory** H of D is defined likewise.

But how do the basic examples 3.1.3 of a square double category and a shell double category translate into HoTT? Stating them is surprisingly straight-forward:

Example 4.2.3 (Square and shell double category)**.** The square double category on a precategory C on a Type $A : \mathcal{U}$ can be instantiated as the double category having, of course, C as a one skeleton, and setting $D_2(f,g,h,i) :\equiv \mathbf{1}$. By doing this, the type of squares does not contain any more information than its arguments. For every quadruple of morphisms forming a square, there is exactly one two-cell. All necessary conditions to make this a double category hold trivially after defining $\mathrm{id}_1(f) \equiv \mathrm{id}_2(f) \equiv (u \circ_1 v) \equiv (u \circ_2 v) :\equiv \star : \mathbf{1}$.

For the commutative square double category, one might be inclined to set $D_2(f,g,h,i) :\equiv \|g \circ h = i \circ f\|$ since each commuting set of faces should not be inhabited by more than one element. But since morphisms between given objects form a set, the commutativity witness is already a mere proposition and we can, without any doubts, define $D_2(f,g,h,i) :\equiv (g \circ h = i \circ f)$. Composition and identities can be defined like stated in 3.1.3, the remaining properties follow easily by the truncation imposed on morphisms and two-cells.

When defining thin structures, we want the uniqueness of a thin filler of a commutative shell to be represented by a functional dependency. By this, we will have more definitional equalities between thin squares than we would get if we defined thin squares to be a mere proposition depending on a quadruple of morphisms.

Definition 4.2.4 (Thin structure)**.** We define a **thin structure** T on a double category D to consist of:

- A dependent function selecting a thin square for each commuting square:

$$\mathsf{thin} : \prod_{a,b,c,d:D_0} \prod_{f:D_1(a,b)} \prod_{g:D_1(c,d)} \prod_{h:D_1(a,c)} \prod_{i:D_1(b,c)} g \circ h = i \circ f \\ \to D_2(f,g,h,i)$$

- For each $a,b : D_0$, $f : D_1(a,b)$, and $p : f \circ \mathrm{id}(a) = \mathrm{id}(b) \circ f$, $q : \mathrm{id}(b) \circ f = f \circ \mathrm{id}(a)$, we have $\mathsf{thin}(f,f,\mathrm{id}(a),\mathrm{id}(b),p) = \mathrm{id}_1(f)$ and $\mathsf{thin}(\mathrm{id}(a),\mathrm{id}(b),f,f,q) = \mathrm{id}_2(f)$. We could have abstained from quantifying over the commutativity proofs and just used $\mathsf{idRight}(f) \cdot \mathsf{idLeft}(f)^{-1}$ and $\mathsf{idLeft}(f) \cdot \mathsf{idRight}(f)^{-1}$ as canonical choices for p and q. But since p and q are proofs for a mere proposition this would yield in an equivalent definition which is a bit easier to instantiate but much less convenient to use.

- For any adjacent squares u and v, $\mathsf{thin}(v) \circ_1 \mathsf{thin}(u) = \mathsf{thin}(v \circ_1 u)$ and $\mathsf{thin}(v) \circ_2 \mathsf{thin}(u) = \mathsf{thin}(v \circ_2 u)$ where appropriate. Here, besides quantifying over 6 objects, 7 morphisms, and two squares, we also quantify over every commutativity proof for the shell of u, v, as well as their respective composite.

Definition 4.2.5 (Weak double groupoid)**.** A **weak double groupoid** is constructed by giving:

- A double category D with objects D_0, morphisms D_1 and two-cells D_2.
- A proof that the 1-skeleton of D is a pregroupoid.
- Vertical and horizontal inverses

$$\begin{aligned} \mathrm{inv}_1 &: \prod_{a,b,c,d:D_0} \prod_{f,g,h,i} D_2(f,g,h,i) \to D_2(g,f,h^{-1},i^{-1}) \text{ and} \\ \mathrm{inv}_2 &: \prod_{a,b,c,d:D_0} \prod_{f,g,h,i} D_2(f,g,h,i) \to D_2(f^{-1},g^{-1},i,h) \end{aligned}$$

- Proofs that inv_1 and inv_2 actually are inverses with respect to vertical and horizontal composition:

$$\begin{aligned} \mathsf{leftInv}_1(u) &: \mathsf{leftInv}(i)_*\big(\mathsf{leftInv}(h)_*(\mathrm{inv}_1(u) \circ_1 u)\big) = \mathrm{id}_1(f), \\ \mathsf{rightInv}_1(u) &: \mathsf{rightInv}(i)_*\big(\mathsf{rightInv}(h)_*(u \circ_1 \mathrm{inv}_1(u))\big) = \mathrm{id}_1(g), \\ \mathsf{leftInv}_2(u) &: \mathsf{leftInv}(g)_*\big(\mathsf{leftInv}(f)_*(\mathrm{inv}_2(u) \circ_2 u)\big) = \mathrm{id}_2(h), \text{ and} \\ \mathsf{rightInv}_2(u) &: \mathsf{rightInv}(g)_*\big(\mathsf{rightInv}(f)_*(u \circ_2 \mathrm{inv}_2(u))\big) = \mathrm{id}_2(i), \end{aligned}$$

  for every $u : D_2(f,g,h,i)$. Here we again leave implicit most of the arguments. $\mathsf{leftInv}$ and $\mathsf{rightInv}$ are the respective witnesses for id in the 1-skeleton of D along which we again have to transport to make the statement well-typed.

Finally, we can define what a double groupoid in homotopy type theory should be:

Definition 4.2.6 (Double groupoid)**.** A **double groupoid** is a weak double groupoid together with a thin structure on it.

We conclude by noting that our double categories and double groupoids here are *strict*, since we defined them to be on set-truncated carriers. Of course, we could omit this condition to obtain a notion of non-strict double categories and double groupoids. Most of my formalization does not assume those structures to be strict, but we need to include the strictness whenever we want to deal with the *category of double categories* or the *category of double groupoids*, because double functors will only form a set when their codomain is strict.

Another question that could be asked is what it means for a double category to be *univalent*. The most reasonable condition would be that, besides the 1-skeleton, also the vertical and horizontal precategory should be univalent. While our main example of a fundamental double groupoid will turn out to be univalent, I could not find a way to gain advantage by restricting general considerations on double groupoids on univalent ones.

Definition 4.2.7 (Double functor)**.** A **double functor** F between double categories D and E shall consist of the following data:

- A functor between the respective 1-skeleton of D and E. We will write F_0 for the function on objects of D and F_1 for the function on morphisms of D.
- For all shells (f, g, h, i) we have a function
$$F_2 : D_2(f, g, h, i) \to D_2(F_1(f), F_1(g), F_1(h), F_1(i))$$
- F respects the vertical and horizontal identities: For all $a, b : D_0$ and $f : D_1(a, b)$, we have proofs $\mathsf{respectId}_1(f)$ and $\mathsf{respectId}_2(f)$ for
$$\mathsf{respectId}(b)_*\big(\mathsf{respectId}(a)_*(F_2(\mathrm{id}_1(f)))\big) = \mathrm{id}_1(F_1(f)) \text{ and}$$
$$\mathsf{respectId}(b)_*\big(\mathsf{respectId}(a)_*(F_2(\mathrm{id}_2(f)))\big) = \mathrm{id}_2(F_1(f)).$$
- F is linear with respect to vertical and horizontal composition:
$$\begin{aligned} &\mathsf{respectComp}(i_2, i_1)_*\big(\mathsf{respectComp}(h_2, h_1)_*(F_2(v \circ_1 u))\big) \\ =&F_2(v) \circ_1 F_2(u) \text{ and} \\ &\mathsf{respectComp}(g_2, g_1)_*\big(\mathsf{respectComp}(f_2, f_1)_*(F_2(v \circ_2 u))\big) \\ =&F_2(v) \circ_2 F_2(u), \end{aligned}$$

  wherever the respective composition of u and v is defined and where $\mathsf{respectComp}$ is the witness that the functor on the 1-skeletons is linear with respect to morphisms in D_1.

Lemma 4.2.8. *Double categories and double functors form a univalent category. Weak double groupoids are a full subcategory of this category.*

Proof. The proof can be done like the one of Lemma 4.1.14. □

4.3 Crossed Modules in Homotopy Type Theory

When defining crossed modules, there is less room for decisions, like in what extent to rely on dependent types, than in the case of double categories and double groupoids. We use *strict* groupoids as base as well as a family of groups that have a set as their carrier:

Definition 4.3.1 (Crossed module)**.** A **crossed module** is defined as comprised of the following information:

- A strict groupoid P on a carrier $P_0 : \mathcal{U}$.
- A family of types $M : P_0 \to \mathcal{U}$ where for each $p : P_0$ we have $\mathsf{isSet}(M_p)$ and a group structure on M_p, whose operation we will denote with "$\cdot$".
- A family $\mu : \prod_{(p:P_0)} M_p \to \hom_P(p,p)$ of functions which all are group homomorphisms:
$$\prod_{p:P_0} \prod_{y,x:M(p)} \mu_p(y \cdot x) = \mu_p(y) \circ \mu_p(x),$$
$$\prod_{p:P_0} \mu_p(1) = \mathrm{id}_P(p).$$
- An action $\phi : \prod_{(p,q:P_0)} \hom_P(p,q) \to M_p \to M_q$ of P on M, which means that
$$\prod_{p:P_0} \prod_{x:M_p} \phi(\mathrm{id}_P(p), x) = x,$$
$$\prod_{p,q,r:P_0} \prod_{g:\hom_P(q,r)} \prod_{h:\hom_P(p,q)} \prod_{x:M_p} \phi(g \circ f, x) = \phi(g, \phi(f,x)), \text{ and}$$
$$\prod_{p,q:P_0} \prod_{f:\hom_P(p,q)} \prod_{y,x:M_p} \phi(f, y \cdot x) = \phi(f,y) \cdot \phi(f,x).$$
- Finally, proofs for the required relation between the action ϕ and conjugation in the respective structures:
$$\prod_{p,q:P_0} \prod_{f:\hom_P(p,q)} \prod_{x:M_p} \mu_q(\phi(f,x)) = f \circ \mu_p(x) \circ f^{-1} \text{ and}$$
$$\prod_{p:P_0} \prod_{c,x:M_p} \phi(\mu_p(c), x) = c \cdot x \cdot c^{-1}.$$

Here, in both equations we have to decide for one of the two ways to place parentheses in the right-hand side of the equation, because associativity only holds propositionally. Either way will cause us to require transporting, in the formalization I went with binding always to the right.

Another definition which will cause no surprise is the one of morphisms between crossed modules:

Definition 4.3.2 (Morphisms of crossed modules)**.** A **morphism between two crossed modules** X and Y on base groupoids P and Q and group families M and N is defined to be comprised of a functor F between the respective base groupoids and a family of functions $\psi : \prod_{p:P_0} M_p \to N_{F(p)}$ which should satisfy the following equations:

$$\prod_{p:P_0} \prod_{y,x:M_p} \psi_p(y \cdot x) = \psi_p(y) \cdot \psi_p(x),$$

$$\prod_{p:P_0} \prod_{x:M_p} F(\mu_p(x)) = \mu_{F(p)}(\psi_p(x)), \text{ and}$$

$$\prod_{p,q:P_0} \prod_{f:\hom_P(p,q)} \prod_{x:M_p} \psi_q(\phi(f,x)) = \phi(F(f), \psi_p(x)).$$

4.4 Presented Types

In this section we want to transfer what in Chapter 3 were the fundamental double groupoid and the fundamental crossed module of a space to the world of higher types. This, of course involves more than just replacing each occurrence of the word "space" with the word "type" but requires more restriction to the kind of information one has to provide to characterize a type by the introduced algebraic structures.

In the topological setting we did not impose any conditions on the topological properties of the components of the nested triple of spaces – even if we pictured C as a disjoint union of points, A as one-dimensional and X as two-dimensional in our Example 3.3.8 of the fundamental double groupoid of a 2-sphere. To meet the truncation level requirements when instantiating the fundamental double groupoid of a triple of types we have to make the truncation levels of the types increase in order of the inclusions. This leads us to the following definition:

Definition 4.4.1. A **presented 2-type** is a triple (X, A, C) of types $X, A, C : \mathcal{U}$ together with functions $\iota : C \to A$ and $\iota' : A \to X$ where X is a 2-type, A is a 1-type and C is a set.

Example 3.3.8 matches these requirements:

Example 4.4.2. The 2-sphere $\mathbb{S}^2$ can be defined as the higher inductive type on

- Four points $c_1, c_2, c_3, c_4 : \mathbb{S}^2$,
- equalities $p_{12} : c_1 = c_2$, $p_{23} : c_2 = c_3$, $p_{34} : c_3 = c_4$, and $p_{41} : c_4 = c_1$, representing the equator, and
- two higher equalities $n, s : p_{12} \cdot p_{23} \cdot p_{34} \cdot p_{41} = \mathsf{refl}_{c_1}$, representing the northern and southern hemisphere.

Using this definition we can define C to be the set $\{c_1, c_2, c_3, c_4\}$, A to be the higher inductive type $\mathbb{S}^1$ generated only by the points $c_1, \ldots, c_4$ and the equalities $p_{12}, \ldots, p_{41}$, and $X :\equiv \mathbb{S}^2$. $\iota : C \to A$ is the obvious embedding mapping $c_i \mapsto c_i$ and $\iota' : A \to X$ is defined by induction on A with $\iota'(c_i) :\equiv c_i : X$ and $\mathsf{ap}_{\iota'}(p_i) = p_i$.

Then, $(\left\|\mathbb{S}^2\right\|_2, A, C)$ is a presented 2-type.

We can now define the fundamental double groupoid of a presented type. As it can be derived from the example above, the objects in C will correspond to the objects of the groupoid, while morphisms in the groupoid will be equalities A and two-cells will be equalities between equalities in X. We will start by first considering the 1-skeleton of this double groupoid.

Definition 4.4.3 (Fundamental groupoid)**.** Let $A : \mathcal{U}$ be a 1-type, $C : \mathcal{U}$ be a set and $\iota : C \to A$. The **fundamental double groupoid** $G_1(A, C)$ associated to this data is a groupoid on the carrier C with $\hom_{G_1(A,C)}(a, b) :\equiv \big(\iota(a) = \iota(b)\big)$ for all $a, b : C$ and $g \circ f :\equiv f \cdot g$ for all $f : \iota(a) = \iota(b)$ and $g : \iota(b) = \iota(c)$.

Proof. The obvious choice for identity morphisms is setting $\mathrm{id}_{G_1(A,C)} :\equiv \prod_{(a:C)} \mathsf{refl}_{\iota(a)}$. Associativity as well as neutrality of $\mathrm{id}_{G_1(A,C)}$ follows directly from the respective properties of equalities. Inverses of morphisms are defined to be the reversed paths. $\square$

Definition 4.4.4 (Fundamental double groupoid)**.** Let (X, A, C) be a presented 2-type related by $\iota : C \to A$ and $\iota' : A \to X$. The **fundamental double groupoid** $G_2(X, A, C)$ of this triple is defined as having the fundamental groupoid $G_1(A, C)$ as 1-skeleton while the dependent type $G_2(X, A, C)_2$ of two-cells is defined as:

$$\prod_{a,b,c,d:C} \prod_{f:\iota a=\iota b} \prod_{g:\iota c=\iota d} \prod_{h:\iota a=\iota c} \prod_{i:\iota b=\iota d} \mathsf{ap}_{\iota'}(h) \cdot \mathsf{ap}_{\iota'}(g) = \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i) \tag{4.1}$$

Proof. Let us start by defining the vertical composition of two-cells: Let

$$u : \mathsf{ap}_{\iota'}(h_1) \cdot \mathsf{ap}_{\iota'}(g_1) = \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i_1) \text{ and}$$
$$v : \mathsf{ap}_{\iota'}(h_2) \cdot \mathsf{ap}_{\iota'}(g_2) = \mathsf{ap}_{\iota'}(g_1) \cdot \mathsf{ap}_{\iota'}(i_2).$$

Then, we obtain $v \circ_1 u$ as the following concatenation of paths:

$$\begin{aligned}
\mathsf{ap}_{\iota'}(h_1 \cdot h_2) \cdot \mathsf{ap}_{\iota'}(g_2) &= (\mathsf{ap}_{\iota'}(h_1) \cdot \mathsf{ap}_{\iota'}(h_2)) \cdot \mathsf{ap}_{\iota'}(g_2) \\
&= \mathsf{ap}_{\iota'}(h_1) \cdot (\mathsf{ap}_{\iota'}(h_2) \cdot \mathsf{ap}_{\iota'}(g_2)) \\
&= \mathsf{ap}_{\iota'}(h_2) \cdot (\mathsf{ap}_{\iota'}(g_1) \cdot \mathsf{ap}_{\iota'}(i_2)) \\
&= (\mathsf{ap}_{\iota'}(h_2) \cdot \mathsf{ap}_{\iota'}(g_1)) \cdot \mathsf{ap}_{\iota'}(i_2) \\
&= (\mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i_1)) \cdot \mathsf{ap}_{\iota'}(i_2) \\
&= \mathsf{ap}_{\iota'}(f) \cdot (\mathsf{ap}_{\iota'}(i_1) \cdot \mathsf{ap}_{\iota'}(i_2)) \\
&= \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i_1 \cdot i_2).
\end{aligned} \tag{4.2}$$

Here, the first and last equation (an instance of the general theorem saying that ap is distributive over the concatenation of paths) is what keeps this definition from being a special case of the shell double category (Definition 4.2.3) and makes its formalization a lot more difficult. Horizontal composition is given analogously, a vertical identity square for a morphism $f : \iota(a) = \iota(b)$ consists of

$$\begin{aligned}
\mathsf{ap}_{\iota'}(\mathsf{refl}_{\iota a}) \cdot \mathsf{ap}_{\iota'}(f) &\equiv \mathsf{refl}_{\iota'\iota a} \cdot \mathsf{ap}_{\iota'}(f) \\
&= \mathsf{ap}_{\iota'}(f) \\
&\equiv \mathsf{ap}_{\iota'}(f) \cdot \mathsf{refl}_{\iota'\iota b} \\
&\equiv \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(\mathsf{refl}_{\iota b}),
\end{aligned}$$

where it depends on the exact definition of equality as an inductive type whether the second and third equality are judgmental.

Proving associativity, identity laws and the interchange law can be done all by using the following scheme:

1. First, we prove a version of the law where points, paths and two-cells all lie in one 2-type X. In this setting we can apply induction to the first and second order paths involved which makes all of the laws reduce to the form $\mathsf{refl} = \mathsf{refl}$.

2. Then, we use the that proof to show the actual instance of the laws. Because of transports along laws in for the 1-skeleton (i.e. assoc, idLeft and $\mathsf{idRight}$) and transports along the theorem that equates $\mathsf{ap}_{\iota'}(p \cdot q)$ an $\mathsf{ap}_{\iota'}(p) \cdot \mathsf{ap}_{\iota'}(q)$, we first end up with an equation that contains lots of "unnecessary" transport.

3. We can eliminate these transports by then assuming that there is no ι, but points and paths lie in A while two cells are equalities in X. Here, we can apply induction to the first order paths involved but not the iterated ones.

I will refrain from stating these proofs in detail and I will give an example in Section 5.7 when presenting the formalization in Lean.

The vertically inverse of a square $u : \mathsf{ap}_{\iota'}(h) \cdot \mathsf{ap}_{\iota'}(g) = \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i)$ is again given by using the functoriality of $\mathsf{ap}_{\iota'}$:

$$\begin{aligned}
\mathsf{ap}_{\iota'}(h^{-1}) \cdot \mathsf{ap}_{\iota'}(f) &= \mathsf{ap}_{\iota'}(h)^{-1} \cdot \mathsf{ap}_{\iota'}(f) \\
&= \mathsf{ap}_{\iota'}(g) \cdot \mathsf{ap}_{\iota'}(i)^{-1} \\
&= \mathsf{ap}_{\iota'}(g) \cdot \mathsf{ap}_{\iota'}(i^{-1}).
\end{aligned}$$

The final question that remains is which squares should be defined as thin. The answer is to regard a square in $G_2(X, A, C)_2(f, g, h, i)$ as thin as soon as it is already be "filled" in A: For every shell (f, g, h, i) and $p : h \cdot g = f \cdot i$ we get a thin square from

$$\begin{aligned}
\mathsf{ap}_{\iota'}(h) \cdot \mathsf{ap}_{\iota'}(i) &= \mathsf{ap}_{\iota'}(h \cdot i) \\
&= \mathsf{ap}_{\iota'}(f \cdot i) \\
&= \mathsf{ap}_{\iota'}(f) \cdot \mathsf{ap}_{\iota'}(i).
\end{aligned}$$

This again connects the name "thin" to its actual geometric interpretation.

□

Chapter 5

A Formalization in the Lean Theorem Prover

My main goal in this thesis project was the formalization and application of Ronald Brown's structures for non-abelian algebraic topology in the theorem prover Lean. Lean, at the point of time when I started working on my project, was still in a very early stage of development and did not only lack any automation but also a basic library for homotopy type theory. Thus, we will first take a look at the basic language elements and technologies used in Lean and then describe the strategies, the structure and the pitfalls we encountered when building up a library for basic homotopy type theory, for categories in homotopy type theory, and finally for double groupoids and crossed modules.

5.1 The Lean Theorem Prover

The development of the theorem prover Lean was initiated in 2013 by Leonardo de Moura. De Moura had previously been working on the automated theorem prover Z3, one of the leading solvers for problem sets in the SMT standard. With Lean, he intends to create a interactive theorem proving system that connects the strength of solvers like Z3 with the expressiveness and flexibility of interactive systems like Agda, Coq or Isabelle. While in the world of automated theorem proving the verification of a statement results in a yes-or-no answer at best accompanied by a counterexample in the case that the statement gets refuted, in interactive theorem proving, we are interested in an actual proof that a statement is correct. Since in homotopy type theory it is relevant which proof of a theorem we consider and since proofs of theorems can be part of another definition or theorem, the proofs in an interactive theorem prover suitable for homotopy type theory should even be objects in the language itself. Lean has two modes: One for standard, proof irrelevant mathematics and one for homtopy type theory. In the following, I will only explain the features of the latter. The explanations are not intended to be a tutorial for the system, but should equip the reader with the knowledge necessary to read the code excerpts in the later chapters.

A first ingredient to the language are **type universes**. Instead of using $\mathcal{U}$, universes in Lean are denoted as `Type.{l}`, where `l` is the level of the universe. Of course, `Type.{l}` is an object of `Type.{l+1}`. But in contrast to homotopy type theory as presented in the HoTT book [Uni13], type universes in Lean are *non-cumulative*, i.e. `A : Type.{l}` does not entail `A : Type.{l+1}`.

Definitions can be *universe polymorphic*, which means that, when no concrete universe levels are given, Lean will keep the definition as general as possible regarding universe levels of arguments and return type. The instantiation of a definition `A` at a universe `l` can be received manually by writing `A.{l}`. To have manual control over the coherence of universe levels of definitions in a certain scope, variable universe levels can be declared using the command **`universe variable`**. The following snippet shows universe polymorphism (introducing an implicit universe placeholder `l_1`) and the use of universe variables:

```
check Type -- Prints Type.{l_1} : Type.{l_1+1}

universe variable l
check Type.{l} -- Prints Type.{l} : Type.{l+1}
```

The only built-in type formers are (dependent and non-dependent) function types, structures, and inductive families.

The **type of functions** between types `A` and `B` is written as `A → B`. `A` and `B` do not have to lie in the same universe to form this type and the universe level of the function type is the maximum of the level of domain and codomain type. Lambda abstraction and function application can be written like known from e.g. Haskell. β reduction is applied automatically for each output, η conversion is applied when necessary in the unification process.

```
variables (A B : Type) (a : A) (b : B) (f : A → B)

check A → B -- Prints A → B : Type.{max l_1 l_2}
check (λ (x : A), b) -- Prints λ (x : A), b : A → B
check (f a) -- Prints f a : B
check (λ x, f x) a -- Prints f a : B
```

An important special case of non-dependent function types are type families of the form `A → Type`. For every `P : A → Type` we can form the **Π-type** `Π (x : A), P x` over `P`. Actually, non-dependent function types are just treated as the special case of dependent functions where `P` is constant. The Π-type `Π (x : A), B` for `A B : Type` is automatically reduced to `A → B`.

```
variables (A B : Type) (P : A → Type) (Q : Π (x : A) , P x → Type)
variables (p : Π (x : A), P x) (a : A)

check p a -- Prints p a : P a
check Q a -- Prints Q a : P a → Type
check (λ (x : A), Q x (p x)) -- Prints λ (x : A), Q x (p x) : A → Type
```

Lean furthermore allows the definition of inductive types and inductive families. To construct an **inductive type**, one must give a list of parameters the type should depend on and a list of constructors. This makes the definition of important types like the natrual numbers or the identity type possible. The dependent recursor for inductive types is generated automatically by the kernel:

```
inductive nat : Type :=
  zero : nat,
  succ : nat → nat
check nat.succ nat.zero -- Prints nat.succ nat.zero : nat
check @nat.rec_on -- Prints Π {C : nat → Type} (n : nat), C nat.zero →
                  --        (Π (a : nat), C a → C (nat.succ a)) → C n
```

```
inductive eq (A : Type) (a : A) : A → Type :=
  refl : eq A a a
variables (A : Type) (a : A)
check @eq.refl A a -- Prints eq.refl a : eq A a a
check @eq.rec_on A a -- Prints Π {C : Π (a_1 : A), eq A a a_1 → Type}
                     --        {a_1 : A} (n : eq A a a_1),
                     --        C a (eq.refl a) → C a_1 n
```

We can not only define single inductive types but also **families of inductive types** [Dyb94], which we can define by recursion on the index of the family:

```
open nat

inductive vec (A : Type) : ℕ → Type :=
  nil : vec A 0,
  cons : Π (n : ℕ), A → vec A n → vec A (n+1)

open vec
variables (A : Type) (a : A)
check @vec.rec_on A -- Prints Π {C : Π (a : ℕ), vec A a → Type} {a : ℕ}
                    --        (n : vec A a), C 0 (nil A) →
                    --        (Π (n : ℕ) (a : A) (a_1 : vec A n), C n a_1
                    --          → C (n + 1) (cons n a a_1)) →
                    --        C a n
check cons 0 a (nil A) -- Prints cons 0 a (nil A) : vec A (0+1)
```

A widely used subclass of inductive types are **structures**, which are similar to what is often referred to as “records”. Structures are inductive types which are non-recursive and only have one constructor. That means that they are equivalent to iterated sigma types but, among other advantages, have named projections. Structures provide a basic inheritance mechanism as they can extend arbitrarily many other structures with **coercions** to each parent structure being added automatically:

```
structure graph (V : Type₀) :=
  (E : V → V → Type₀)

structure refl_graph (V : Type₀) extends graph V :=
  (refl : Π (v : V), E v v)

structure trans_graph (V : Type₀) extends graph V :=
  (trans : Π (u v w : V), E u v → E v w → E u w)

structure refl_trans_graph (V : Type₀) extends refl_graph V, trans_graph V

variables (V : Type₀)
check graph.E (refl_graph.mk (λ (a b : V), a = b) eq.refl)
/- Prints
  graph.E (refl_graph.to_graph
    (refl_graph.mk (λ (a b : V), a = b) eq.refl)) :
    V → V → Type₀ -/
```

As one can already seen in these small examples, writing out the full names and the complete list of parameters for each call of a defined function can be very tedious. Lean implements some features that allow its users to make theory files more succinct and readable by leaving out information that can be inferred automatically by Lean.

One feature that allows more brevity are **implicit arguments**. To mark an argument to a definition to be automatically inferred by Lean, the user can mark it with curly brackets instead of round brackets when listing it in the signature of the definition. Of course the missing arguments have to be such that they can be uniquely determined by the unification process, otherwise the unifier will return an appropriate error message whenever the definition is used. The user can make all arguments in a function call implicit by prepending a `!` to the definition name. The opposite, making all arguments explicit, can be achieved by prepending the symbol `@`. By default, implicit arguments are *maximally inserted*. That means, that any expression of a Π-type with its first argument implicit is not interpreted as is but already further applied by inference of that argument. By using double curly brackets `⦃...⦄` instead of single curly brackets, the user can change this behaviour to be more passive and only infer an argument automatically if it precedes a explicitely stated one:

```
definition inverse {P : Type} {x y : P} (p : x = y) : y = x :=
  eq.rec (eq.refl x) p
check inverse -- Prints inverse : ?x = ?y → ?y = ?x

definition id ⦃A : Type⦄ (a : A) := A
check id -- Prints id : Π ⦃A : Type⦄, A → Type

variables {A : Type} (a : A)
check id a -- Prints id a : Type
```

Another useful feature to organize formalizations and shorten code is the interaction between namespaces and overloading. In general there are no restrictions on **overloading** theorem names but each additional overload with the same name makes it harder for the elaborator to figure out which one it needs to use in a certain case. To make overloads more organized, definitions can be put in hierarchical **namespaces** which can be opened selectively. Opening a namespace makes each theorem in that namespace available to be called with its name as stated in the definition inside that namespace, as opposed to giving its absolute name. Definitions can be marked as `protected` to prevent their names from being pruned and as `private` to exclude them from exporting completely. Furthermore, opening namespaces enables the use of **notations** defined in that namespace.

```
open nat

namespace exponentiation
  definition squared (a : ℕ) := a * a
  notation a`²`:100 := squared a
end exponentiation

open exponentiation
check squared -- Prints squared : ℕ → ℕ
eval 4² -- Prints 16
```

Often, consecutive definitions share a lot of arguments. To avoid the need to repeat those arguments for each definition, Lean provides **sections** which serve as scopes for common variables. Besides `variables`, there are `parameters` which will not be generalized until the context or section is closed.

One last feature helping to create short and readable files are **type classes**. After marking an inductive type or a family of inductive type as `[class]`, we can declare definitions which are objects of this type as `[instance]`. Then, we can use a fourth mode of argument implicitness (besides `(...)`, `{...}`, and `⦃...⦄`) denoted by `[...]` to tell Lean that it should infer this argument by filling in one of the instances of the required type. The selection of one out of several fitting instances can be influenced by assigning priorities to the instance declarations. Type classes are a tool for achieving *canonicity*. One important application is the fixation of instances of algebraic structures on certain types or their closure under certain type formers, for example the structure of an abelian group on the type of integers or the direct product of groups as a canonical group structure on a product type. By marking theorems as instances which themselves have type class parameters, we can put quite some automation load on the type class resolution algorithm. In the following example we give a canonical graph structure on the natural numbers and prove that the resulting graph is serial:

```
open nat eq sigma.ops

structure graph [class] (V : Type₀) := (E : V → V → Type₀)

definition nat_graph [instance] : graph nat := graph.mk (λ x y, y = x+1)

definition is_serial (V : Type₀) [G : graph V] := Π x, Σ y, graph.E x y

definition nat_serial : is_serial nat := λ x, ⟨x+1, idp⟩
```

In all previous examples, we stated all definitions and proofs by giving the whole term at once. In some theorem provers like Agda [Nor09], this method, support by the presence of `let` and `where` terms, is used for all proofs while others, like Coq [BBC$^+$97], rely on the use of **tactics**. Tactics are sequentially executed commands that transform a given proof goal into a set of (potentially) easier to solve goals. Lean allows both approaches to be used purely or even intertwined, using tactics for subterms of a declarative proofs. The user can declare a proof in tactic mode with `begin ... end`. As soon as the tactic block has been successfully filled in with tactic calls, the full proof term gets elaborated, type checked, and is, from there on, in distinguishable from a declaratively entered proof.

The tactics in Lean are still under development and will be extended in near future. Thus, I will only give a list of the tactics that were used at the time of writing the formalizations in this thesis:

- The tactic `exact` takes a proof term as argument which it uses to solve the first subgoal completely. If Lean fails to infer every argument that is not stated explicitly or if Lean cannot unify the expression with the goal, the tactic fails.

- If one wants to work on the current first subgoal with an arbitrary theorem without already solving it completely, one can use `apply`. It unifies the theorem with the subgoal but opposed to failing it creates a new subgoal for each undetermined or unresolved argument of the stated theorem. The new subgoals are added from the last argument to the first. Arguments, that can be inferred from later arguments will not be converted to a new subgoal.

- Sometimes, one wants to prove premises or give arguments in the order they appear in the statement of the applied theorem. This can be achieved by using `fapply`. In general, it results in more subgoals than `apply`.

- Using `assert` lets the user prove a named hypothesis which then is added to the context for the rest of the main proof.
- If the current subgoal is a (dependent or non-dependent) function, `intro` adds the variable, which the goal quantifies over, to the context and applies it to the goal. `intros` does the same iteratively for multiple variables. `revert` does the opposite: It selects a variable in the context and replaces the goal by its quantification over this variable. A similar job is done by `generalize`. It takes an arbitrary term as argument and quantifies the goal over a variable of that term's type, replacing all its occurrences by the quantifying variable.
- The tactic `clear` is used to delete unused variables from the context. This is especially useful to speed up type class resolution and to shorten the output of the current context.
- `cases` destructs a given context variable of an inductive type using the automatically generated `cases_on` theorem. Optionally, a list of the desired names for the newly generated variables can be appended.
- Often, a goal is not expressed in its simplest form. `esimp` simplifies the goal after unfolding a given list of definitions.
- The `rewrite` tactic takes a theorem yielding a (propositional) equality and replaces occurrences of the equation's left hand side by the right hand side. The user can specify in what pattern or how often the replacement should be made.
- By using `rotate`, one can rotate the list of subgoals by a given number of steps. This is useful when the user wants to postpone the goal at the first position to be solved at the end of the proof.
- Tactics can furthermore be concatenated to a single tactic using the and-then composition `(tactic1 ; tactic2)`. The resulting tactic fails if either one of the tactics fail.
  The notation `[tactic1 | tactic2]` specifies an or-else branching of tactics: Lean first tries to apply the first tactic and if it fails the second one. The composed tactic only fails if both of them fail.

- Adding to this composition of tactics one can use the **`repeat`** tactic as a loop command. The tactic given to **`repeat`** as an argument is applied over and over until it fails. So, by giving it an or-else composition of tactics Lean tries each one from a set of tactics until none of them is applicable. If one wants to specify the number of repetitions manually (e.g. for performance reasons), **`do`** is the tactic of choice.

- Curly brackets can be used to create a scope which only aims to solve the first subgoal. If the subgoal is not solved within that scope, an error will appear at the line of the closing bracket.

5.2 Basic Homotopy Type Theory in Lean

Lean's homotopy type theory library is split into two parts: The content of a folder `init` is imported by default and provides theories which are used in built-in features like **`rewrite`**. Besides this folder there are other theories which are not required at startup and which the user can import manually. They include characterizations of path spaces and basic theories about common algebraic structures which will be presented in the next chapter. The extension and maintenance of the library is an ongoing joint effort by Jeremy Avigad, Floris van Doorn, Leonardo de Moura and me.

One important definition in the initialization folder is, of course, the one of **propositional equality**. As mentioned above, equality is just a very basic example for an inductive type. Here, we define equality `eq` as inductive family of types and, with `idp`, give an alternative name for refl with the base being an implicit argument:

```
universe variable l
inductive eq.{l} {A : Type.{l}} (a : A) : A → Type.{l} :=
refl : eq a a

definition idp {a : A} := refl a
```

Concatenation and inversion of paths are defined using path induction and abbreviated with the obvious notation:

```
definition concat (p : x = y) (q : y = z) : x = z := eq.rec (λu, u) q p
definition inverse (p : x = y) : y = x := eq.rec (refl x) p

notation p₁ ⬝ p₂ := concat p₁ p₂
notation p ⁻¹ := inverse p
```

By this definition, $p \cdot \mathsf{refl}$ is definitionally equal to p while $\mathsf{refl} \cdot p = p$ can only be proved by induction on p. A lot of basic calculations in the path groupoid can also be proved by just using path induction. Here is one example for this kind of lemma:

```
definition eq_of_idp_eq_inv_con (p q : x = y) : idp = p⁻¹ ⬝ q → p = q :=
eq.rec_on p (take q h, h ⬝ (idp_con _)) q
```

Two other basic definitions for paths are the one of transports and ap, the application of a non-dependent function to a path.

```
definition transport [reducible] (P : A → Type) {x y : A}
  (p : x = y) (u : P x) : P y :=
eq.rec_on p u

definition ap ⦃A B : Type⦄ (f : A → B) {x y:A} (p : x = y) : f x = f y :=
eq.rec_on p idp
```

To be able to express univalence it is important to have a useful definition of the **equivalence of types**. We use structures to express the type of proofs $\mathsf{isequiv}(f)$ that a function f is a half-adjoint equivalence as well as for the type of equivalences between two given types ($\sim$ denotes the type of homotopies between two functions) :

```
structure is_equiv [class] {A B : Type} (f : A → B) :=
  (inv : B → A)
  (retr : (f ∘ inv) ∼ id)
  (sect : (inv ∘ f) ∼ id)
  (adj : Π x, retr (f x) = ap f (sect x))

structure equiv (A B : Type) :=
  (to_fun : A → B)
  (to_is_equiv : is_equiv to_fun)
```

In practice, we will almost never use the generated constructor of `is_equiv` but instead we proved an alternative constructor `adjointify` which does not require the adjointness proof `adj`.

Univalence then states that the function `equiv_of_eq`, which lets us gain an equivalence from an equality between two types in the same universe, is itself an equivalence. We mark the axiom as instance so that, whenever there is mention of `equiv_of_eq`, we have access to its inverse. This inverse `ua` is also the only common form in which univalence appears elsewhere.

```
section
  universe variable l
  variables {A B : Type.{l}}

  definition is_equiv_tr_of_eq (H : A = B) :
    is_equiv (transport (λ X, X) H) :=
  @is_equiv_tr Type (λX, X) A B H

  definition equiv_of_eq (H : A = B) : A ≃ B :=
  equiv.mk _ (is_equiv_tr_of_eq H)
end

axiom univalence (A B : Type) : is_equiv (@equiv_of_eq A B)
attribute univalence [instance]

definition ua {A B : Type} : A ≃ B → A = B := (@equiv_of_eq A B)⁻¹
```

We use the univalence axiom to prove **function extensionality** of first non-dependent and then dependent functions. We first define three varieties of function extensionality and then proof that the desired definition of function extensionality (stating that the function apD10, which returns for each equality between functions a homotopy between those functions, is an equivalence) follows from the other ones. This approach has been ported from the Coq HoTT library [hota].

```
definition funext.{l k} :=
Π ⦃A : Type.{l}⦄ {P : A → Type.{k}} (f g : Π x, P x),
  is_equiv (@apD10 A P f g)

-- Naive funext is the assertion that pointwise equal functions are equal.
definition naive_funext :=
Π ⦃A : Type⦄ {P : A → Type} (f g : Πx, P x), (f ∼ g) → f = g

-- Weak funext says that a product of contractible types is contractible.
definition weak_funext :=
Π ⦃A : Type⦄ (P : A → Type) [H: Πx, is_contr (P x)], is_contr (Πx, P x)
```

Truncation levels are implemented by first creating a version of the natural numbers that “start at -2” together with a coercion from the actual type nat, then defining internal versions of contractability and truncatedness, and eventually defining a type class structure holding a proof of the internal truncatedness:

```
inductive trunc_index : Type₁ :=
  minus_two : trunc_index,
  succ : trunc_index → trunc_index
...
structure contr_internal (A : Type) :=
  (center : A)
  (contr : Π(a : A), center = a)

definition is_trunc_internal (n : trunc_index) : Type → Type :=
trunc_index.rec_on n (λA, contr_internal A)
  (λn trunc_n A, (Π(x y : A), trunc_n (x = y)))
...
structure is_trunc [class] (n : trunc_index) (A : Type) :=
  (to_internal : is_trunc_internal n A)

abbreviation is_contr := is_trunc -2
abbreviation is_hprop := is_trunc -1
abbreviation is_hset  := is_trunc 0
```

By this trick we can then define contractability as the special case of being truncated at level -2. This makes it a lot easier to write theorems that hold for all levels of truncatedness. The type class `is_trunc` has many instances that, by calling type class resolution themselves, automatize the process of finding the truncation level of a given iterated Σ-type or Π-type (proofs and surrounding context omitted, definitions gathered from multiple files):

```
--Equalities in contractible types are contractible.
definition is_contr_eq {A : Type} [H : is_contr A] (x y : A) :
  is_contr (x = y) := ...

-- n-types are also (n+1)-types.
definition is_trunc_succ [instance] [priority 100]
  (A : Type) (n : trunc_index) [H : is_trunc n A] : is_trunc (n.+1) A := ...

-- The unit type is contractible.
definition is_contr_unit [instance] : is_contr unit := ...

-- The empty type is a mere proposition.
definition is_hprop_empty [instance] : is_hprop empty := ...

-- A Sigma type is n-truncated if the type of all possible projections is.
definition is_trunc_sigma [instance] (B : A → Type) (n : trunc_index)
  [HA : is_trunc n A] [HB : Πa, is_trunc n (B a)] :
  is_trunc n (Σ a, B a) := ...

```

```
-- Any dependent product of n-types is an n-type.
definition is_trunc_pi [instance] (B : A → Type) (n : trunc_index)
  [H : Πa, is_trunc n (B a)] : is_trunc n (Πa, B a) := ...

-- Being an equivalence is a mere proposition.
theorem is_hprop_is_equiv [instance] : is_hprop (is_equiv f) := ...
```

An important part of the library, besides the initialization files, consists of theorems characterizing paths between instances of different type formers. One example for such a characterization is that a path between two dependent pairs can be built out of paths between their projections:

```
definition dpair_eq_dpair (p : a = a') (q : p ▹ b = b') : ⟨a, b⟩ = ⟨a', b'⟩ :=
by cases p; cases q; apply idp

definition sigma_eq (p : u.1 = v.1) (q : p ▹ u.2 = v.2) : u = v :=
by cases u; cases v; apply (dpair_eq_dpair p q)
```

Tables 5.1 and 5.2 show the line count and the compilation time for each theory in Lean's HoTT library excluding the library for category theory.

5.3 Category Theory in Lean

Our library of basic category theoretical definitions and theorems, which is still work in progress, also mimics the structure of Coq's HoTT implementation of categories while aiming to be more succinct than it. We use structures and type classes for most definitions of algebraic structures and their closure properties.

The central structure our formalization revolves around is, of course, the one of a precategory:

```
structure precategory [class] (ob : Type) : Type :=
  (hom : ob → ob → Type)
  (homH : Π(a b : ob), is_hset (hom a b))
  (comp : Π⦃a b c : ob⦄, hom b c → hom a b → hom a c)
  (ID : Π (a : ob), hom a a)
  (assoc : Π ⦃a b c d : ob⦄ (h : hom c d) (g : hom b c) (f : hom a b),
     comp h (comp g f) = comp (comp h g) f)
  (id_left : Π ⦃a b : ob⦄ (f : hom a b), comp !ID f = f)
  (id_right : Π ⦃a b : ob⦄ (f : hom a b), comp f !ID = f)
```

Since usually the domain of an identity morphism is determined by the context, we will most often use a shorter notation for the identity:

Theory	Line Count	Compilation Time in s
`init.`		
`bool`	28	0.043
`datatypes`	90	0.044
`default`	15	0.421
`equiv`	276	1.082
`function`	61	0.042
`hedberg`	47	0.399
`logic`	359	0.173
`nat`	345	0.565
`num`	135	0.073
`path`	648	2.683
`priority`	12	0.036
`relation`	43	0.072
`reserved_notation`	103	0.035
`tactic`	106	0.067
`trunc`	262	0.821
`util`	18	0.324
`wf`	162	0.394
`axioms.`		
`funext_of_ua`	162	0.674
`funext_varieties`	111	0.483
`ua`	51	0.302
`types.`		
`empty`	23	0.055
`prod`	99	0.225
`sigma`	26	0.058
`sum`	19	0.036

Table 5.1: Theories imported in Lean's initial startup.

Theory	Line Count	Compilation Time in s
`arity`	188	0.797
`algebra.`		
`binary`	74	0.156
`group`	570	1.317
`relation`	122	0.330
`types.`		
`arrow`	49	0.143
`eq`	271	0.556
`equiv`	98	0.652
`fiber`	51	0.199
`pi`	198	1.177
`pointed`	40	0.121
`prod`	48	0.144
`sigma`	397	1.599
`trunc`	140	0.371
`W`	157	0.128

Table 5.2: Theories in Lean's standard library for its homotopy type theory mode. (Category theory excluded.)

```
definition id [reducible] := ID a
```

We introduce a shortcut for the type class instance postulating that each equality of morphisms is a mere proposition. We use this instance, for example, to prove that it is sufficient to give equalities between the morphism types, composition and identities of two precategories on the same type of objects, to show that these precategories are equal:

```
definition is_hprop_eq_hom [instance] : is_hprop (f = f') := !is_trunc_eq

definition precategory_eq_mk' (ob : Type) (C D : precategory ob)
  (p : @hom ob C = @hom ob D)
  (q : transport (λ x, Πa b c, x b c → x a b → x a c) p
    (@comp ob C) = @comp ob D)
  (r : transport (λ x, Πa, x a a) p (@ID ob C) = @ID ob D) : C = D :=
begin
  cases C, cases D,
  apply precategory_eq_mk, apply q, apply r,
end
```

We also define a *bundled* version of a category, as the structure containing the object type and a precategory on it as fields:

```
structure Precategory : Type :=
  (carrier : Type)
  (struct : precategory carrier)
```

Using typeclasses for split monos, split epis and isomorphisms enables us to access the left, right or both-sided inverse filling in the precategory structure as well as the invertability witness by type class instance resolution:

```
structure split_mono [class]
    {ob : Type} [C : precategory ob] {a b : ob} (f : a ⟶ b) :=
  {retraction_of : b ⟶ a}
  (retraction_comp : retraction_of ∘ f = id)

structure split_epi [class]
    {ob : Type} [C : precategory ob] {a b : ob} (f : a ⟶ b) :=
  {section_of : b ⟶ a}
  (comp_section : f ∘ section_of = id)

structure is_iso [class]
    {ob : Type} [C : precategory ob] {a b : ob} (f : a ⟶ b) :=
  {inverse : b ⟶ a}
  (left_inverse  : inverse ∘ f = id)
  (right_inverse : f ∘ inverse = id)
```

On purpose, we weaken some theorems to accepting arbitrary witnesses for mere propositions that would actually be derivable from the other witnesses. By this, we can have class instance resolution fill in a witness selected by highest instance priority. In the following example, even if there is a theorem obtaining `is_iso (f⁻¹)` from `is_iso f`, we might want the witness to be the axiom that in a groupoid all morphisms are isomorphisms. This spares us the need of transporting the statement to the right witness each time we use it:

```
definition inverse_involutive (f : a ⟶ b) [H : is_iso f] [H : is_iso (f⁻¹)]
  : (f⁻¹)⁻¹ = f :=
inverse_eq_right !left_inverse

definition id_inverse (a : ob) [H : is_iso (ID a)] : (ID a)⁻¹ = id :=
inverse_eq_left !id_comp

definition comp_inverse [Hp : is_iso p] [Hpq : is_iso (q ∘ p)] :
  (q ∘ p)⁻¹ʰ = p⁻¹ʰ ∘ q⁻¹ʰ :=
inverse_eq_left (show (p⁻¹ʰ ∘ q⁻¹ʰ) ∘ q ∘ p = id, from
  by rewrite [-assoc, inverse_comp_cancel_left, left_inverse])
```

Not only the the property of a morphism f being an isomorphism but also the type of morphisms between two given objects is encapsulated in a structure that helps to fill in proofs automatically.

```
structure iso (a b : ob) :=
  (to_hom : hom a b)
  [struct : is_iso to_hom]

infix `≅`:50 := iso.iso
attribute iso.struct [instance] [priority 400]

-- The type of isomorphisms between two objects is a set.
definition is_hset_iso [instance] : is_hset (a ≅ b) :=
begin
  apply is_trunc_is_equiv_closed,
    apply (equiv.to_is_equiv (!iso.sigma_char)),
end
```

Another definition we later want to generalize to the two dimensional case is the one of a functor. We introduce functors as a structure and add coercions to its function on objects and to its function on morphisms. By doing so, we will be able to write `F a` for its evaluation at an object `a` and `F f` for its evaluation at a morphism `f`.

```
structure functor (C D : Precategory) : Type :=
  (to_fun_ob : C → D)
  (to_fun_hom : Π ⦃a b : C⦄, hom a b → hom (to_fun_ob a) (to_fun_ob b))
  (respect_id : Π (a : C), to_fun_hom (ID a) = ID (to_fun_ob a))
  (respect_comp : Π {a b c : C} (g : hom b c) (f : hom a b),
    to_fun_hom (g ∘ f) = to_fun_hom g ∘ to_fun_hom f)

infixl `⇒`:25 := functor
attribute to_fun_ob [coercion]
attribute to_fun_hom [coercion]
```

One example where these coercions are used is the definition of the composition of functors:

```
definition compose [reducible] (G : functor D E) (F : functor C D) :
  functor C E :=
functor.mk
  (λ x, G (F x))
  (λ a b f, G (F f))
  (λ a, calc
    G (F (ID a)) = G (ID (F a)) : by rewrite respect_id
             ... = ID (G (F a)) : by rewrite respect_id)
  (λ a b c g f, calc
    G (F (g ∘ f)) = G (F g ∘ F f)     : by rewrite respect_comp
              ... = G (F g) ∘ G (F f) : by rewrite respect_comp)
```

Like mentioned above, it is useful to still have a representation of a structure as an iterated product and Σ-type. With this characterization we can formalize Lemma 4.1.6 easily using type class resolution:

```
protected definition sigma_char :
  (Σ (to_fun_ob : C → D)
  (to_fun_hom : Π ⦃a b : C⦄, hom a b → hom (to_fun_ob a) (to_fun_ob b)),
  (Π (a : C), to_fun_hom (ID a) = ID (to_fun_ob a)) ×
  (Π {a b c : C} (g : hom b c) (f : hom a b),
    to_fun_hom (g ∘ f) = to_fun_hom g ∘ to_fun_hom f)) ≃ (functor C D) :=
...

section
  local attribute precategory.homH [priority 1001]
  protected theorem is_hset_functor [instance]
    [HD : is_hset D] : is_hset (functor C D) :=
  by apply is_trunc_equiv_closed; apply functor.sigma_char
end
```

This enables us to implement the precategory of strict precategories (compare Corollary 4.1.7). The proof that this precategory is univalent (Lemma 4.1.14) still lacks a formalization.

```
structure strict_precategory [class] (ob : Type) extends precategory ob :=
  (is_hset_ob : is_hset ob)

structure Strict_precategory : Type :=
  (carrier : Type)
  (struct : strict_precategory carrier)

definition precat_strict_precat : precategory Strict_precategory :=
precategory.mk (λ a b, functor a b)
  (λ a b, @functor.is_hset_functor a b _)
  (λ a b c g f, functor.compose g f)
  (λ a, functor.id)
  (λ a b c d h g f, !functor.assoc)
  (λ a b f, !functor.id_left)
  (λ a b f, !functor.id_right)
```

5.4 Formalizing Double Groupoids

When formalizing the structures presented in Chapters 3 and 4, I proceeded in the order the concepts are presented in this thesis. Since it was

Theory	Line Count	Compilation Time in s
algebra.		
groupoid	119	0.419
category.		
basic	74	0.265
constructions	144	1.180
precategory.		
adjoints	143	*0.638
basic	236	1.870
constructions	268	1.862
functor	253	*2.981
iso	350	2.131
nat_trans	114	1.033
strict	53	0.246
yoneda	208	3.852

Table 5.3: The theories in Lean's category theory library. Theories marked with * contain unfinished proofs.

necessary to change several definitions to improve compatibility with the library and to improve performance I had to change most of the definitions repeatedly during the process. In the following, I will only present the final version of the definitions.

The first structure is the one of a **double category**. Since it has many fields, it was useful to come up with an idea to shorten the definition: We first define what it means to be a **worm category**. This structure consists of objects, morphisms and two-cells the same way a double category does, but it only allows for composition in one direction. In contrast to an actual double category there are essentially two, instead of three, categories involved in the definition of a worm category.

```
structure worm_precat {D₀ : Type} (C  : precategory D₀)
  (D₂ : Π ⦃a b c d : D₀⦄
    (f : hom a b) (g : hom c d) (h : hom a c) (i : hom b d), Type) :=
  (comp₁ : proof Π ⦃a b c₁ d₁ c₂ d₂ : D₀⦄
    ⦃f₁ : hom a b⦄ ⦃g₁ : hom c₁ d₁⦄ ⦃h₁ : hom a c₁⦄ ⦃i₁ : hom b d₁⦄
    ⦃g₂ : hom c₂ d₂⦄ ⦃h₂ : hom c₁ c₂⦄ ⦃i₂ : hom d₁ d₂⦄,
    (D₂ g₁ g₂ h₂ i₂) → (D₂ f₁ g₁ h₁ i₁)
    → (@D₂ a b c₂ d₂ f₁ g₂ (h₂ ∘ h₁) (i₂ ∘ i₁)) qed)
  (ID₁ : proof Π ⦃a b : D₀⦄ (f : hom a b), D₂ f f (ID a) (ID b) qed)
  (assoc₁ : proof Π ⦃a b c₁ d₁ c₂ d₂ c₃ d₃ : D₀⦄
    ⦃f  : hom a b⦄   ⦃g₁ : hom c₁ d₁⦄ ⦃h₁ : hom a c₁⦄ ⦃i₁ : hom b d₁⦄
    ⦃g₂ : hom c₂ d₂⦄ ⦃h₂ : hom c₁ c₂⦄ ⦃i₂ : hom d₁ d₂⦄
    ⦃g₃ : hom c₃ d₃⦄ ⦃h₃ : hom c₂ c₃⦄ ⦃i₃ : hom d₂ d₃⦄
    (w : D₂ g₂ g₃ h₃ i₃) (v : D₂ g₁ g₂ h₂ i₂) (u : D₂ f g₁ h₁ i₁),
    (assoc i₃ i₂ i₁) ▹ ((assoc h₃ h₂ h₁) ▹
        (comp₁ w (comp₁ v u))) = (comp₁ (comp₁ w v) u) qed)
  (id_left₁ : proof Π ⦃a b c d : D₀⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄
    (u : D₂ f g h i),
    (id_left i) ▹ ((id_left h) ▹ (comp₁ (ID₁ g) u)) = u qed)
  (id_right₁ : proof Π ⦃a b c d : D₀⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄
    (u : D₂ f g h i),
    (id_right i) ▹ ((id_right h) ▹ (comp₁ u (ID₁ f))) = u qed)
  (homH' : proof Π ⦃a b c d : D₀⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄,
    is_hset (D₂ f g h i) qed)
```

We then use the inheritance mechanism for structures to define a double category as extending two worm precategories on the same object type `D₀`, the same 1-skeleton `C` and dependent types of two-cells that differ by *transposition* in the sense that if `D₂` is the type of two-cells of the “vertical” worm category, `(λ ⦃a b c d : D₀⦄ f g h i, D₂ h i f g)` is the respective dependent type for the “horizontal” one. To prevent the fields of the two worm precategories from being merged (which is the default for structure fields with identical names), we have to rename the fields of the horizontal worm category. Then we add the laws that could not be expressed in terms of only one direction of two-cell composition.

```
structure dbl_precat {D₀ : Type} (C : precategory D₀)
  (D₂ : Π ⦃a b c d : D₀⦄
    (f : hom a b) (g : hom c d) (h : hom a c) (i : hom b d), Type)
  extends worm_precat C D₂,
    worm_precat C (λ ⦃a b c d : D₀⦄ f g h i, D₂ h i f g)
  renaming comp₁→comp₂ ID₁→ID₂ assoc₁→assoc₂
    id_left₁→id_left₂ id_right₁→id_right₂ homH'→homH'_dontuse :=
  (id_comp₁ : proof Π {a b c : D₀} (f : hom a b) (g : hom b c),
    ID₂ (g ∘ f) = comp₁ (ID₂ g) (ID₂ f) qed)
  (id_comp₂ : proof Π {a b c : D₀} (f : hom a b) (g : hom b c),
    ID₁ (g ∘ f) = comp₂ (ID₁ g) (ID₁ f) qed)
  (zero_unique : proof Π (a : D₀), ID₁ (ID a) = ID₂ (ID a) qed)
  (interchange : proof Π {a₀₀ a₀₁ a₀₂ a₁₀ a₁₁ a₁₂ a₂₀ a₂₁ a₂₂ : D₀}
    {f₀₀ : hom a₀₀ a₀₁} {f₀₁ : hom a₀₁ a₀₂} {f₁₀ : hom a₁₀ a₁₁}
    {f₁₁ : hom a₁₁ a₁₂} {f₂₀ : hom a₂₀ a₂₁} {f₂₁ : hom a₂₁ a₂₂}
    {g₀₀ : hom a₀₀ a₁₀} {g₀₁ : hom a₀₁ a₁₁} {g₀₂ : hom a₀₂ a₁₂}
    {g₁₀ : hom a₁₀ a₂₀} {g₁₁ : hom a₁₁ a₂₁} {g₁₂ : hom a₁₂ a₂₂}
    (x : D₂ f₁₁ f₂₁ g₁₁ g₁₂) (w : D₂ f₁₀ f₂₀ g₁₀ g₁₁)
    (v : D₂ f₀₁ f₁₁ g₀₁ g₀₂) (u : D₂ f₀₀ f₁₀ g₀₀ g₀₁),
    comp₁ (comp₂ x w) (comp₂ v u) = comp₂ (comp₁ x v) (comp₁ w u) qed)
```

This unbundled definition of a double category is useful when making statements about all double category structures on a certain pair of 1-skeletons and two-cell types. But since we often want to talk about *all* double categories (in a certain universe), we add a bundled up version that also adds the strictness condition.

```
structure Dbl_precat : Type :=
  (cat : Precategory)
  (two_cell : Π ⦃a b c d : cat⦄ (f : hom a b)
    (g : hom c d) (h : hom a c) (i : hom b d), Type)
  (struct : dbl_precat cat two_cell)
  (obj_set : is_hset (carrier cat))
```

With that definition, we can start instantiating first simple examples. When implementing the square double category (Definition 4.2.3), we can make use of the `repeat` tactic to fill in each of the many arguments of the double category constructor with either the constructor $\star$ of **1**, the fact that **1** is a mere proposition, or the fact mere propositions are sets:

```
definition square_dbl_precat : dbl_precat C
  (λ ⦃a b c d : D₀⦄ (f : hom a b) (g : hom c d)
    (h : hom a c) (i : hom b d), unit) :=
begin
  fapply dbl_precat.mk,
    repeat (intros; (rexact ⋆ |  apply is_hprop.elim | apply is_trunc_succ)),
    repeat (intros;  apply idp),
end
```

The definition of the shell double category is not as straightforward, since we have to perform the calculations from 3.1.3 for the commutativity of composite squares:

```
definition comm_square_dbl_precat : dbl_precat C
  (λ ⦃a b c d : D₀⦄ (f : hom a b) (g : hom c d)
    (h : hom a c) (i : hom b d), g ∘ h = i ∘ f) :=
begin
  fapply dbl_precat.mk,
    intros, exact (calc g₂ ∘ h₂ ∘ h₁ = (g₂ ∘ h₂) ∘ h₁ : assoc
                                 ... = (i₂ ∘ g₁) ∘ h₁ : a_1
                                 ... = i₂ ∘ g₁ ∘ h₁ : assoc
                                 ... = i₂ ∘ i₁ ∘ f₁ : a_2
                                 ... = (i₂ ∘ i₁) ∘ f₁ : assoc),
    intros, exact (calc f ∘ ID a = f : id_right
                              ... = ID b ∘ f : id_left),
    repeat (intros; apply is_hset.elim),
    intros, apply is_trunc_eq,
    intros, exact (calc (i₂ ∘ i₁) ∘ f₁ = i₂ ∘ i₁ ∘ f₁ : assoc
                                    ... = i₂ ∘ g₁ ∘ h₁ : a_2
                                    ... = (i₂ ∘ g₁) ∘ h₁ : assoc
                                    ... = (g₂ ∘ h₂) ∘ h₁ : a_1
                                    ... = g₂ ∘ h₂ ∘ h₁ : assoc),
    intros, exact (calc ID b ∘ f = f : id_left
                              ... = f ∘ ID a : id_right),
    repeat (intros; apply is_hset.elim),
    intros, apply is_trunc_eq,
    repeat (intros; apply is_hset.elim),
end
```

Extracting the horizontal and vertical precategory of a double category is something that can already be defined on a worm precategory where we have *one* category of two-cells. We encapsulate the objects and the morphisms of this category in their own structure definitions:

```
parameters {D₀ : Type} [C : precategory D₀] {D₂ : ...} (D : worm_precat C D₂)

structure two_cell_ob :=
  (vo1 vo2 : D₀)
  (vo3 : hom vo1 vo2)

structure two_cell_connect (Sf Sg : two_cell_ob) :=
  (vc1 : hom (two_cell_ob.vo1 Sf) (two_cell_ob.vo1 Sg))
  (vc2 : hom (two_cell_ob.vo2 Sf) (two_cell_ob.vo2 Sg))
  (vc3 : D₂ (two_cell_ob.vo3 Sf) (two_cell_ob.vo3 Sg) vc1 vc2)
```

After characterizing those types by sigma types, characterizing equalities between them and examining the truncation level of the structures we can define the two-cell precategory of a worm category: (Note the relations between the universe levels involved.)

```
universe variables l₀ l₁ l₂
variables {D₀ : Type.{l₀}} [C : precategory.{l₀ (max l₀ l₁)} D₀]
  {D₂ : Π ..., Type.{max l₀ l₁ l₂}}
```

```

definition two_cell_precat (D : worm_precat C D₂)
  : precategory.{(max l₀ l₁) (max l₀ l₁ l₂)} (two_cell_ob D) :=
begin
  fapply precategory.mk.{(max l₀ l₁) (max l₀ l₁ l₂)},
    intros [Sf, Sg], exact (two_cell_connect D Sf Sg),
    intros [Sf, Sg], apply is_trunc_is_equiv_closed, apply equiv.to_is_equiv,
      exact (two_cell_connect_sigma_char D Sf Sg),
      apply is_trunc_sigma, intros,
      apply is_trunc_sigma, intros, apply (homH' D),
    intros [Sf, Sg, Sh, Sv, Su], apply (two_cell_comp D Sv Su),
    intro Sf, exact (two_cell_id D Sf),
    intros, exact (two_cell_assoc D h g f),
    intros [Sf, Sg, Su], exact (two_cell_id_left D Su),
    intros [Sf, Sg, Su], exact (two_cell_id_right D Su),
end
```

Now we can obtain the vertical and horizontal precategory as the two-cell precategory of each parent worm precategory:

```
definition vert_precat (D : dbl_precat C D₂) :=
worm_precat.two_cell_precat.{l₀ l₁ l₂} (to_worm_precat_1 D)

definition horiz_precat (D : dbl_precat C D₂) :=
worm_precat.two_cell_precat.{l₀ l₁ l₂} (to_worm_precat_2 D)
```

Before we continue to define thin structures and double groupoids, we create a library of helper lemmas that will often be used in the other theories. In the definition of a double category we already saw that often composite squares have to be transported along an equality in one of their faces. Continuing to prove more complex equations of squares, these transports can best be managed when on the *outside* of a composition. So we create a library of theorems that equate a square composition involving a transport on one or both of the squares with a another composition where the transport is pulled to the outside or eliminated. Some examples for these lemmas are the following:

```
variables {a b c d b₂ d₂ : D₀} {E : Type}
  {f : hom a b} {g : hom c d} {h : hom a c} {i : hom b d}
  {f₂ : hom b b₂} {g₂ : hom d d₂} {i₂ : hom b₂ d₂}

definition transp_comp₂_eq_comp₂_transp_l_l {e : E → hom a c}
  {h h' : E} (q : h = h')
  (u : D₂ f g (e h) i) (v : D₂ f₂ g₂ i i₂) :
```

```
  transport (λ x, D₂ _ _ (e x) _) q (comp₂ D v u)
  = comp₂ D v (transport (λ x, D₂ _ _ (e x) _) q u) :=
by cases q; apply idp

definition transp_comp₂_inner_deal1 {e : E → hom b d}
  {i i' : E} (q : i = i')
  (u : D₂ f g h (e i)) (v : D₂ f₂ g₂ (e i') i₂) :
  comp₂ D v (transport (λ x, D₂ _ _ _ (e x)) q u)
  = comp₂ D (transport (λ x, D₂ _ _ (e x) _) q⁻¹ v) u :=
by cases q; apply idp

definition transp_comp₂_eq_comp₂_transp_inner {e : E → hom b d}
  {i i' : E} (q : i = i')
  (u : D₂ f g h (e i)) (v : D₂ f₂ g₂ (e i) i₂) :
  comp₂ D v u = comp₂ D (transport (λ x, D₂ _ _ (e x) _) q v)
    (transport (λ x, D₂ _ _ _ (e x)) q u) :=
by cases q; apply idp
```

Since for a given double category there will, in all relevant cases, be one canonical thin structure, we use the type class mechanism to find that thin structure as an instance of the following **type class of thin structures**:

```
structure thin_structure [class] {D₀ : Type} [C : precategory D₀]
    {D₂ : Π ⦃a b c d : D₀⦄, hom a b → hom c d → hom a c → hom b d → Type}
    (D : dbl_precat C D₂) :=
  (thin : Π ⦃a b c d : D₀⦄
    (f : hom a b) (g : hom c d) (h : hom a c) (i : hom b d), g ∘ h = i ∘ f
    → D₂ f g h i)
  (thin_id₁ : proof Π ⦃a b : D₀⦄ (f : hom a b),
    thin f f (ID a) (ID b) ((id_right f) ⬝ (id_left f)⁻¹) = ID₁ D f qed)
  (thin_id₂ : proof Π ⦃a b : D₀⦄ (f : hom a b),
    thin (ID a) (ID b) f f ((id_left f) ⬝ (id_right f)⁻¹) = ID₂ D f qed)
  (thin_comp₁ : proof Π ⦃a b c₁ d₁ c₂ d₂ : D₀⦄
    ⦃f₁ : hom a b⦄ ⦃g₁ : hom c₁ d₁⦄ ⦃h₁ : hom a c₁⦄ ⦃i₁ : hom b d₁⦄
    ⦃g₂ : hom c₂ d₂⦄ ⦃h₂ : hom c₁ c₂⦄ ⦃i₂ : hom d₁ d₂⦄
    (pv : g₂ ∘ h₂ = i₂ ∘ g₁) (pu : g₁ ∘ h₁ = i₁ ∘ f₁)
    (px : g₂ ∘ h₂ ∘ h₁ = (i₂ ∘ i₁) ∘ f₁),
    comp₁ D (thin g₁ g₂ h₂ i₂ pv) (thin f₁ g₁ h₁ i₁ pu)
    = thin f₁ g₂ (h₂ ∘ h₁) (i₂ ∘ i₁) px qed)
  (thin_comp₂ : proof Π ⦃a b c₁ d₁ c₂ d₂ : D₀⦄
    ⦃f₁ : hom a b⦄ ⦃g₁ : hom c₁ d₁⦄ ⦃h₁ : hom a c₁⦄ ⦃i₁ : hom b d₁⦄
    ⦃g₂ : hom c₂ d₂⦄ ⦃h₂ : hom c₁ c₂⦄ ⦃i₂ : hom d₁ d₂⦄
    (pv : i₂ ∘ g₁ = g₂ ∘ h₂) (pu : i₁ ∘ f₁ = g₁ ∘ h₁)
    (px : (i₂ ∘ i₁) ∘ f₁ = g₂ ∘ h₂ ∘ h₁),
    comp₂ D (thin h₂ i₂ g₁ g₂ pv) (thin h₁ i₁ f₁ g₁ pu)
    = thin (h₂ ∘ h₁) (i₂ ∘ i₁) f₁ g₂ px qed)
```

```

open thin_structure
check @thin_id₁ /- Prints
  thin_id₁ :
  Π {D₀ : Type} {C : precategory D₀}
    {D₂ : Π ⦃a b c d : D₀⦄, hom a b → hom c d → hom a c → hom b d → Type}
    (D : dbl_precat C D₂) [c : thin_structure D] ⦃a b : D₀⦄ (f : hom a b),
        thin D f f (ID a) (ID b) (id_right f ⬝ (id_left f)⁻¹) = ID₁ D f -/
```

Defining a connection is, of course, very straightforward:

```
definition br_connect ⦃a b : D₀⦄ (f : hom a b) : D₂ f (ID b) f (ID b) :=
thin D f (ID b) f (ID b) idp

definition ul_connect ⦃a b : D₀⦄ (f : hom a b) : D₂ (ID a) f (ID a) f :=
thin D (ID a) f (ID a) f idp
```

In contrast, proving the S-law (3.2.3) takes a surprising amount of effort. It is the first of many theorems for which we need helper lemmas that generalize paths to make a statement provable by path induction.

```
definition ID₁_of_ul_br_aux {a b : D₀} (f g h : hom a b)
  (p : g = f) (q : h = f)
  (r1 : h ∘ id = id ∘ g) (r2 : f ∘ id = id ∘ f)
  (rr : q ▹ (p ▹ r1) = r2) :
  q ▹ (p ▹ thin D g h id id r1) = thin D f f id id r2 :=
by cases rr; cases p; cases q; apply idp

definition ID₁_of_ul_br ⦃a b : D₀⦄ (f : hom a b) :
  (id_left f) ▹ ((id_right f) ▹
  (comp₂ D (br_connect f) (ul_connect f))) = ID₁ D f :=
begin
  -- Bring transports to right hand side
  apply tr_eq_of_eq_inv_tr, apply tr_eq_of_eq_inv_tr,
  -- Work on left hand side
  apply concat,
    -- Composites of thin squares are thin
    apply thin_comp₂,
    -- Commutativity of composite square
    apply inverse, apply assoc,
  -- Bring transports to left hand side
  apply eq_inv_tr_of_tr_eq, apply eq_inv_tr_of_tr_eq,
  apply concat,
    -- Apply helper lemma eliminating transports
    apply ID₁_of_ul_br_aux, apply is_hset.elim,
    exact ((id_right f) ⬝ (id_left f)⁻¹),
  -- Identity squares are thin
  apply thin_id₁,
end
```

To prove the transport laws, we use a similar auxiliary lemma and the **assert** command to provide proofs of commutativity and for the different rows to the context:

```
definition br_of_br_square_aux {a c : D₀} (gf : hom a c)
  (h₁ : hom c c) (p : h₁ = ID c)
  (r1 : h₁ ∘ gf = h₁ ∘ gf) (r2 : (ID c) ∘ gf = (ID c) ∘ gf) :
  (p ▹ thin D gf h₁ gf h₁ r1) = thin D gf (ID c) gf (ID c) r2 :=
by cases p; apply (ap (λ x, thin D _ _ _ _ x) !is_hset.elim)

definition br_of_br_square ⦃a b c : D₀⦄ (f : hom a b) (g : hom b c) :
  (id_left id) ▹ (comp₁ D (comp₂ D (br_connect g) (ID₂ D g))
    (comp₂ D (ID₁ D g) (br_connect f)))
  = br_connect (g ∘ f) :=
begin
  apply tr_eq_of_eq_inv_tr,
  -- Prove commutativity of second row
  assert line2_commute : (id ∘ id) ∘ g = id ∘ g ∘ id,
    exact (calc (id ∘ id) ∘ g = id ∘ g : @id_left D₀ C
                         ... = (id ∘ g) ∘ id : id_right
                         ... = id ∘ (g ∘ id) : assoc),
  -- Prove thinness of second row
  assert line2_thin : comp₂ D (br_connect g) (ID₂ D g)
    = thin D (g ∘ id) (id ∘ id) g id line2_commute,
    apply concat, apply (ap (λx, comp₂ D _ x)), apply inverse, apply thin_id₂,
    apply thin_comp₂,
  -- Prove commutativity of first row
  assert line1_commute : (g ∘ id) ∘ f = id ∘ g ∘ f,
    exact (calc (g ∘ ID b) ∘ f = g ∘ f : @id_right D₀ C
                          ... = ID c ∘ g ∘ f : id_left),
  -- Prove thinness of first row
  assert line1_thin : comp₂ D (ID₁ D g) (br_connect f)
    = thin D (g ∘ f) (g ∘ id) f id line1_commute,
    apply concat, apply (ap (λx, comp₂ D x _)), apply inverse, apply thin_id₁,
    apply thin_comp₂,
  -- Replace composite squares by thin squares
  apply concat, exact (ap (λx, comp₁ D x _) line2_thin),
  apply concat, exact (ap (λx, comp₁ D _ x) line1_thin),
  -- Thinness of the entire 2x2 grid
  apply concat, apply thin_comp₁, apply idp,
  apply eq_inv_tr_of_tr_eq,
  apply br_of_br_square_aux,
end
```

Now we can start defining double groupoids. The definition of a weak double groupoid poses a greater task to Lean's type checker than the one of a double category which is why we first define the the type of the arguments that are required to make a weak double groupoid out of a double category as separate definitions to give Lean the chance to elaborate and type check the terms before using them in the actual definition. Note that in the **extends** command in structure definition, `C` is automatically cast from a groupoid to a general precategory.

```
context
  parameters
    {D₀ : Type}
    (C : groupoid D₀)
    (D₂ : Π ⦃a b c d : D₀⦄, hom a b → hom c d → hom a c →  hom b d → Type)
    (D : dbl_precat C D₂)

  definition inv₁_type : Type :=
  Π ⦃a b c d : D₀⦄ {f : hom a b} {g : hom c d} {h : hom a c} {i : hom b d},
    D₂ f g h i → D₂ g f (h⁻¹) (i⁻¹)

  definition left_inverse₁_type (inv₁ : inv₁_type) : Type :=
  Π ⦃a b c d : D₀⦄ {f : hom a b} {g : hom c d} {h : hom a c} {i : hom b d}
    (u : D₂ f g h i),
    (left_inverse i) ▹ (left_inverse h) ▹ (comp₁ D (inv₁ u) u) = ID₁ D f
  ...
end

structure weak_dbl_gpd {D₀ : Type} (C : groupoid D₀)
  (D₂ : Π ⦃a b c d : D₀⦄, hom a b → hom c d → hom a c →  hom b d → Type)
  extends D : dbl_precat C D₂ :=
  (inv₁ : inv₁_type C D₂)
  (left_inverse₁ : left_inverse₁_type C D₂ D inv₁)
  (right_inverse₁ : right_inverse₁_type C D₂ D inv₁)
  (inv₂ : inv₂_type C D₂)
  (left_inverse₂ : left_inverse₂_type C D₂ D inv₂)
  (right_inverse₂ : right_inverse₂_type C D₂ D inv₂)
```

To implement Definition 4.2.6 of a double groupoid, we only have to add a thin structure to a weak double groupoid. For the definition of the category of double groupoids we again add a strict and bundled version:

```
structure dbl_gpd {D₀ : Type} (C : groupoid D₀) (D₂ : Π ⦃a b c d : D₀⦄,
    hom a b → hom c d → hom a c →  hom b d → Type)
```

```
  extends D : weak_dbl_gpd C D₂:=
  (T : thin_structure (weak_dbl_gpd.to_dbl_precat D))

structure Dbl_gpd : Type :=
  (gpd : Groupoid)
  (two_cell : Π ⦃a b c d : gpd⦄,
    hom a b → hom c d → hom a c →  hom b d → Type)
  (struct : dbl_gpd gpd two_cell)
  (obj_set : is_hset (carrier gpd))
```

For the definition of a double functor we again have to pull out the definition of the argument types to make it easier for the elaborator:

```
structure dbl_functor (D E : Dbl_gpd) :=
  (catF : functor (gpd D) (gpd E))
  (twoF : Π ⦃a b c d : gpd D⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄,
    two_cell D f g h i → two_cell E (catF f) (catF g) (catF h) (catF i))
  (respect_id₁ : respect_id₁_type D E catF twoF)
  (respect_comp₁ : respect_comp₁_type D E catF twoF)
  (respect_id₂ : respect_id₂_type D E catF twoF)
  (respect_comp₂ : respect_comp₂_type D E catF twoF)
```

As a first lemma characterizing equalities between double functors we have the following:

```
parameters (D E : Dbl_gpd)
  (catF1 catF2 : functor (gpd D) (gpd E))
  (twoF1 : Π ⦃a b c d : gpd D⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄,
    two_cell D f g h i → two_cell E (catF1 f) (catF1 g) (catF1 h) (catF1 i))
  (twoF2 : Π ⦃a b c d : gpd D⦄
    ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄,
    two_cell D f g h i → two_cell E (catF2 f) (catF2 g) (catF2 h) (catF2 i))
  (respect_id₁1 : proof respect_id₁_type D E catF1 qed twoF1)
  (respect_id₁2 : proof respect_id₁_type D E catF2 qed twoF2)
  (respect_comp₁1 : proof respect_comp₁_type D E catF1 qed twoF1)
  (respect_comp₁2 : proof respect_comp₁_type D E catF2 qed twoF2)
  (respect_id₂1 : proof respect_id₂_type D E catF1 qed twoF1)
  (respect_id₂2 : proof respect_id₂_type D E catF2 qed twoF2)
  (respect_comp₂1 : proof respect_comp₂_type D E catF1 qed twoF1)
  (respect_comp₂2 : proof respect_comp₂_type D E catF2 qed twoF2)

definition dbl_functor.congr (p1 : catF1 = catF2) (p2 : p1 ▹ twoF1 = twoF2) :
  dbl_functor.mk catF1 twoF1
```

```
    respect_id₁1 respect_comp₁1 respect_id₂1 respect_comp₂1
  = dbl_functor.mk catF2 twoF2
    respect_id₁2 respect_comp₁2 respect_id₂2 respect_comp₂2 :=
begin
  cases p1, cases p2,
  intros, apply (ap01111 (λ f g h i, dbl_functor.mk catF2 twoF2 f g h i)),
    repeat (
      repeat ( apply eq_of_homotopy ; intros ) ;
      apply (@is_hset.elim _ (!(homH' E))) ),
end
```

But in practice, this formalization turned out to be less useful than one where we don't have a path between the category functors but instead between their objects and morphisms, separately. The parameters of such a theorem `dbl_functor.congr'` are the following. (`apD011` is the lemma that equates $f(a, b)$ and $f(a', b')$ for an arbitrary dependent function f and equalities between the respective arguments.)

```
(p1 : to_fun_ob catF1 = to_fun_ob catF2)
(p2 : transport
  (λ x, Π (a b : carrier (gpd D)), hom a b → hom (x a) (x b)) p1
  (to_fun_hom catF1) = to_fun_hom catF2)
(p3 : apD011 (λ Hob Hhom,
              Π ⦃a b c d : carrier (gpd D)⦄
                ⦃f : hom a b⦄ ⦃g : hom c d⦄ ⦃h : hom a c⦄ ⦃i : hom b d⦄,
                two_cell D f g h i →
                @two_cell E (Hob a) (Hob b) (Hob c) (Hob d)
                 (Hhom a b f) (Hhom c d g) (Hhom a c h) (Hhom b d i))
      p1 p2 ▷ twoF1 = twoF2)
```

Using these more fine grained prerequisites for equality between double functors we can prove the associativity of double functors in a pretty straightforward way:

```
definition dbl_functor.assoc {B C D E : Dbl_gpd}
  (H : dbl_functor D E) (G : dbl_functor C D) (F : dbl_functor B C) :
  dbl_functor.compose H (dbl_functor.compose G F)
  = dbl_functor.compose (dbl_functor.compose H G) F :=
begin
  fapply (dbl_functor.congr' B E),
      apply idp,
    apply idp,
  apply idp,
end
```

The composition mentioned in that theorem is defined as obvious on the 1-skeleton and on the two-cells, but it is quite a bit of work to show that it respects identities and composition. Let's take a look at the part of the proof that shows that the composite functor respects the vertical identity:

```
intros, apply tr_eq_of_eq_inv_tr, apply tr_eq_of_eq_inv_tr,
apply concat, apply (ap (λ x, twoF G x)), apply respect_id₁',
apply concat, apply twoF_transport_l, apply tr_eq_of_eq_inv_tr,
apply concat, apply twoF_transport_r, apply tr_eq_of_eq_inv_tr,
apply concat, apply respect_id₁',
apply inv_tr_eq_of_eq_tr, apply inv_tr_eq_of_eq_tr,
apply inverse,
apply concat, apply (transport_eq_transport4 (λ f g h i, two_cell E f g h i)),
apply concat, apply transport4_transport_acc,
apply concat, apply transport4_transport_acc,
apply concat, apply transport4_transport_acc,
apply concat, apply transport4_transport_acc,
apply concat, apply transport4_transport_acc,
apply transport4_set_reduce,
```

The two calls to `apply respect_id₁'` are the use of the fact that each of the double functors respects identity squares. The lemmas `twoF_transport_l` and `twoF_transport_r` serve to pull a transport out of the application of a double functor to a two-cell. Just as in many proofs, we then end up with a goal that consists of an equation with the same term on its left and right hand side but with lots of transport terms on one side. Since the transports are all on the different faces of a two-cell, we create an auxiliary definition `transport4` holding transports for each face. Then we turn the regular transports into instances of this new definition using `transport_eq_transport4` and `transport4_transport_acc` so that in the end we can use the fact that morphisms between two objects form a set to eliminate the `transport4` term.

`transport4` itself ist defined as follows:

```
parameters {A B C D : Type} (P : A → B → C → D → Type)
definition transport4 {a0 a1 : A} {b0 b1 : B} {c0 c1 : C} {d0 d1 : D}
  (pa : a0 = a1) (pb : b0 = b1) (pc : c0 = c1) (pd : d0 = d1)
  (u : P a0 b0 c0 d0) : P a1 b1 c1 d1 :=
pd ▹ pc ▹ pb ▹ pa ▹ u
```

After turning the outermost `transport` into a `transport4` we can accumulate other transport by just using the following lemma:

```
definition transport4_transport_acc {E : Type}
  {a0 : A} {b0 : B} {c0 : C} {d0 : D}
  {e0 e1 : E} {f : E → A} {g : E → B} {h : E → C} {i : E → D}
  (pa : f e1 = a0) (pb : g e1 = b0) (pc : h e1 = c0) (pd : i e1 = d0)
  (p : e0 = e1) (u : P (f e0) (g e0) (h e0) (i e0)) :
transport4 pa pb pc pd (transport (λ (x : E), P (f x) (g x) (h x) (i x)) p u)
= transport4 (ap f p ⬝ pa) (ap g p ⬝ pb) (ap h p ⬝ pc) (ap i p ⬝ pd) u :=
by cases pa; cases pb; cases pc; cases pd; cases p; apply idp
```

The final reduction step is done by assuming that all parameter types A, B, C, and D are sets:

```
definition transport4_set_reduce [HA : is_hset A] [HB : is_hset B]
  [HC : is_hset C] [HD : is_hset D]
  {a0 : A} {b0 : B} {c0 : C} {d0 : D}
  {pa : a0 = a0} {pb : b0 = b0} (pc : c0 = c0) (pd : d0 = d0)
  (u : P a0 b0 c0 d0) :
  transport4 pa pb pc pd u = u :=
begin
  assert Ppa : pa = idp, apply is_hset.elim,
  assert Ppb : pb = idp, apply is_hset.elim,
  assert Ppc : pc = idp, apply is_hset.elim,
  assert Ppd : pd = idp, apply is_hset.elim,
  rewrite [Ppa, Ppb, Ppc, Ppd],
end
```

Finally, we can instantiate the precategory of double groupoids:

```
universe variables l₁ l₂ l₃
definition cat_dbl_gpd [reducible] :
  precategory.{(max l₁ l₂ l₃)+1 (max l₁ l₂ l₃)} Dbl_gpd.{l₁ l₂ l₃} :=
begin
  fapply precategory.mk,
    intros [D, E], apply (dbl_functor D E),
    intros [D, E], apply (is_hset_dbl_functor D E),
    intros [C, D, E, G, F], apply (dbl_functor.compose G F),
    intro D, apply (dbl_functor.id D),
    intros [B, C, D, E, H, G, F], apply (dbl_functor.assoc),
    intros [B, C, F], apply (dbl_functor.id_left),
    intros [B, C, F], apply (dbl_functor.id_right),
end
```

5.5 Formalizing Crossed Modules

The definition of a crossed module involves stating the fact that the given map μ is a group homomorphism and that ϕ is a groupoid action. Since the algebra library did not contain any definitions for these algebraic notions and since their properties are only rarely used, I decided to add them to the definition of a crossed module as additional fields:

```
structure xmod {P₀ : Type} [P : groupoid P₀] (M : P₀ → Group) :=
  (P₀_hset : is_hset P₀)
  (μ : Π ⦃p : P₀⦄, M p → hom p p)
  (μ_respect_comp : Π ⦃p : P₀⦄ (b a : M p), μ (b * a) = μ b ∘ μ a)
  (μ_respect_id : Π (p : P₀), μ 1 = ID p)
  (φ : Π ⦃p q : P₀⦄, hom p q → M p → M q)
  (φ_respect_id : Π ⦃p : P₀⦄ (x : M p), φ (ID p) x = x)
  (φ_respect_P_comp : Π ⦃p q r : P₀⦄ (b : hom q r) (a : hom p q) (x : M p),
    φ (b ∘ a) x = φ b (φ a x))
  (φ_respect_M_comp : Π ⦃p q : P₀⦄ (a : hom p q) (y x : M p),
    φ a (y * x) = (φ a y) * (φ a x))
  (CM1 : Π ⦃p q : P₀⦄ (a : hom p q) (x : M p), μ (φ a x) = a ∘ (μ x) ∘ a⁻¹)
  (CM2 : Π ⦃p : P₀⦄ (c x : M p), φ (μ c) x = c * (x * c⁻¹ᵍ))
```

The fact that μ respects inverses and ϕ respects the neutral element of the group can then be derived from the fields listed in the definition of a crossed module:

```
definition μ_respect_inv ⦃p : P₀⦄ (a : M p) : μ MM a⁻¹ᵍ = (μ MM a)⁻¹ :=
begin
  assert H : μ MM a⁻¹ᵍ ∘ μ MM a = (μ MM a)⁻¹ ∘ μ MM a,
    exact calc μ MM a⁻¹ᵍ ∘ μ MM a = μ MM (a⁻¹ᵍ * a) : μ_respect_comp
                              ... = μ MM 1 : by rewrite mul_left_inv
                              ... = id : μ_respect_id
                              ... = (μ MM a)⁻¹ ∘ μ MM a : left_inverse,
  apply epi.elim, exact H,
end

definition φ_respect_one ⦃p q : P₀⦄ (a : hom p q) : φ MM a 1 = 1 :=
begin
  assert H : φ MM a 1 * 1 = φ MM a 1 * φ MM a 1,
    exact calc φ MM a 1 * 1 = φ MM a 1 : mul_one
                        ... = φ MM a (1 * 1)  : one_mul
                        ... = φ MM a 1 * φ MM a 1 : φ_respect_M_comp,
  apply eq.inverse, apply (mul_left_cancel H),
end
```

Just like we did for double functors, we define morphisms of crossed modules as their own structures, with lemmas to state a representation by iterated sigma and product types, their truncation level and a lemma to build an equality between two of them:

```
structure xmod_morphism : Type :=
  (gpd_functor : functor (Groupoid.mk X (gpd X)) (Groupoid.mk Y (gpd Y)))
  (hom_family : Π (p : X), (groups X p) → (groups Y (gpd_functor p)))
  (hom_family_hom : Π (p : X) (x y : groups X p),
    hom_family p (x * y) = hom_family p x * hom_family p y)
  (mu_commute : Π (p : X) (x : groups X p),
    gpd_functor (μ X x) = μ Y (hom_family p x))
  (phi_commute : Π (p q : X) (a : hom p q) (x : groups X p),
    hom_family q (φ X a x) = φ Y (gpd_functor a) (hom_family p x))

definition xmod_morphism_sigma_char :
  (Σ (gpd_functor : functor (Groupoid.mk X (gpd X)) (Groupoid.mk Y (gpd Y)))
    (hom_family : Π (p : X), (groups X p) → (groups Y (gpd_functor p))),
    (Π (p : X) (x y : groups X p),
      hom_family p (x * y) = (hom_family p x) * (hom_family p y))
    × (Π (p : X) (x : groups X p),
    to_fun_hom gpd_functor (μ X x) = μ Y (hom_family p x))
    × (Π (p q : X) (a : @hom X _ p q) (x : groups X p),
    hom_family q (φ X a x) = φ Y (gpd_functor a) (hom_family p x)))
      ≃ xmod_morphism := ...

definition xmod_morphism_hset : is_hset xmod_morphism := ...

parameters ...
  (p : to_fun_ob gpd_functor1 = to_fun_ob gpd_functor2)
  (q : transport (λ x, Π a b, hom a b → hom (x a) (x b)) p
    (to_fun_hom gpd_functor1) = to_fun_hom gpd_functor2)
  (r : transport (λ x, Π p', (groups X p') → (groups Y (x p'))) p
    hom_family1 = hom_family2)

definition xmod_morphism_congr :
  xmod_morphism.mk gpd_functor1 hom_family1
    hom_family_hom1 mu_commute1 phi_commute1
  = xmod_morphism.mk gpd_functor2 hom_family2
    hom_family_hom2 mu_commute2 phi_commute2 := ...
```

This equality lemma can then be used to prove the identity and associativity of crossed module morphisms to build the precategory of crossed modules.

```
definition xmod_morphism_id_left :
  xmod_morphism_comp (xmod_morphism_id Y) f = f :=
begin
  cases f,
  fapply xmod_morphism_congr,
      apply idp,
    apply idp,
  repeat (apply eq_of_homotopy ; intros),
  apply idp,
end

universe variables l₁ l₂ l₃
definition cat_xmod :
  precategory.{(max l₁ l₂ l₃)+1 (max l₁ l₂ l₃)} Xmod.{l₁ l₂ l₃} :=
begin
  fapply precategory.mk,
    intros [X, Y], apply (xmod_morphism X Y),
    intros [X, Y], apply xmod_morphism_hset,
    intros [X, Y, Z, g, f], apply (xmod_morphism_comp g f),
    intro X, apply xmod_morphism_id,
    intros [X, Y, Z, W, h, g, f], apply xmod_morphism_assoc,
    intros [X, Y, f], apply xmod_morphism_id_left,
  intros [X, Y, f], apply xmod_morphism_id_right,
end
```

5.6 Proving the Equivalence

Proving the equivalence between the categories **DGpd** and **XMod** involves the following steps:

1. Define γ and λ on objects – as functions mapping double groupoids to crossed modules and vice versa.

2. Define γ and λ on the morphisms of the respective categories: On double functors and crossed module morphisms. This is a step that we glossed over when defining the functors in the original topological and set theoretic setting.

3. Instantiate γ and λ as actual functors by proving that they respect identities and composition.

4. Build natural transformations $\gamma\lambda \to \mathrm{id}_{\mathbf{XMod}}$ and $\lambda\gamma \to \mathrm{id}_{\mathbf{DGpd}}$.

5. Show that these natural transformations are isomorphic by showing that they map each object in the respective category to an isomorphism in that category.

We start by implementing γ as a function `Dbl_gpd → Xmod`. This first of all involved building the family of groups in the desired crossed module. We create a further structure to hold the objects of each of these groups: The type of two-cells which have the identity on all but their upper face.

```
structure folded_sq (a : D₀) :=
  (lid : hom a a)
  (filler : D₂ lid id id id)
```

After proving the group axioms for `folded_sq a`, a task which again involves extensive transport management, we can instantiate the actual group of "folded squares" over a point `a : D₀`. Here, `l₂` is the the universe level of morphisms in the double groupoid in the context and `l₃` is the level of two-cells.

```
protected definition folded_sq_group [instance] (a : D₀) :
  group (folded_sq a) :=
begin
  fapply group.mk,
    intros [u, v], apply (folded_sq.comp u v),
    apply (folded_sq.is_hset a),
    intros [u, v, w], apply ((folded_sq.assoc u v w)⁻¹),
    apply folded_sq.one,
    intro u, apply (folded_sq.id_left u),
    intro u, apply (folded_sq.id_right u),
    intro u, apply (folded_sq.inv u),
    intro u, apply (folded_sq.left_inverse u),
end

protected definition folded_sq_Group [reducible] (a : D₀) :
  Group.{max l₂ l₃} :=
Group.mk (folded_sq a) (folded_sq_group a)
```

The definitions for μ and ϕ themselves are again very straightforward, while some of the axioms are surprisingly hard to prove, one example being the proof that ϕ respects the group composition, which takes more than 100 lines due to the massive need to transform `transport` terms. In the end we can define the bundled crossed module that results from γ:

```
protected definition xmod [reducible] :
  xmod (λ x, gamma.folded_sq_Group G x) :=
begin
  fapply xmod.mk,
    exact D₀set,
    intros [x, u], apply (gamma.mu G u),
    intros [x, v, u], apply (gamma.mu_respect_comp G v u),
    intro x, apply gamma.mu_respect_id,
    intros [x, y, a, u], apply (gamma.phi G a u),
    intros [x, u], apply (gamma.phi_respect_id G u),
    intros [x, y, z, b, a, u], apply (gamma.phi_respect_P_comp G b a u),
    intros [x, y, a, v, u], apply (gamma.phi_respect_M_comp G a v u),
    intros [x, y, a, u], apply (gamma_CM1 a u),
    intros [x, v, u], apply (gamma_CM2 v u),
end

end

open Dbl_gpd
protected definition on_objects [reducible] (G : Dbl_gpd) : Xmod :=
Xmod.mk (λ x, gamma.folded_sq_Group G x) (gamma.xmod G)
```

Defining the functor γ on morphisms we have to map a double functor to a morphism of crossed modules. We first define the morphism on the group family by applying the double functor to the “folded squares” with transports to keep three of their faces the identity. Proving that this defines a group homomorphism, and proving the further axioms again makes use of the statements that functors respect composition and identities and of the helper lemmas to move the resulting transport terms.

```
context
parameters {G H : Dbl_gpd} (F : dbl_functor G H)

protected definition on_morphisms_on_base [reducible]
  (p : gamma.on_objects G) (x : Xmod.groups (gamma.on_objects G) p) :
  Xmod.groups (gamma.on_objects H) (to_fun_ob (dbl_functor.catF F) p) :=
begin
  cases F with [catF, twoF, F3, F4, F5, F6],
  cases G with [gpdG,sqG,structG,carrierG_hset],
  cases H with [gpdH,sqH,structH,carrierH_hset],
  cases x with [lid,filler],
  fapply folded_sq.mk, apply (to_fun_hom catF lid),
  apply (transport (λ x, sqH _ x id id) (respect_id catF p)),
  apply (transport (λ x, sqH _ _ x id) (respect_id catF p)),
  apply (transport (λ x, sqH _ _ _ x) (respect_id catF p)),
  apply (twoF filler),
end

set_option unifier.max_steps 30000
protected definition on_morphisms_hom_family [reducible]
  (p : Xmod.carrier (gamma.on_objects G))
  (x y : Xmod.groups (gamma.on_objects G) p) :
  gamma.on_morphisms_on_base F p (x * y) =
  (gamma.on_morphisms_on_base F p x) * (gamma.on_morphisms_on_base F p y) :=
begin
  ...
end

protected definition on_morphisms :
  xmod_morphism (gamma.on_objects G) (gamma.on_objects H) :=
begin
  ...
end
```

We show the final instantiation of γ as a functor using function extensionality for the proof that γ respects composition and identity.

```
protected definition functor :
  functor Cat_dbl_gpd.{l₁ l₂ l₃} Cat_xmod.{(max l₁ l₂) l₂ l₃} :=
begin
  fapply functor.mk,
    intro G, apply (gamma.on_objects G),
    intros [G, H, F], apply (gamma.on_morphisms F),
    intro G, cases G,
      fapply xmod_morphism_congr, apply idp, apply idp,
      repeat ( apply eq_of_homotopy ; intros), cases x_1, apply idp,
  ...
end
```

Just like for γ, we create a structure to hold the two-cell of the double groupoid we create when defining the functor λ:

```
parameters {P₀ : Type} [P : groupoid P₀] {M : P₀ → Group} (MM : xmod M)
...
structure lambda_morphism ⦃a b c d : P₀⦄
  (f : hom a b) (g : hom c d) (h : hom a c) (i : hom b d) :=
(m : M d) (comm : μ MM m = i ∘ f ∘ h⁻¹ ∘ g⁻¹)
```

Compositions, identity squares, thin squares and axioms that are needed to build a double groupoid are defined without major difficulties. In the end we collect all 27 of these definitions to form the double groupoid we want:

```
protected definition dbl_gpd [reducible] : dbl_gpd P lambda_morphism :=
begin
  fapply dbl_gpd.mk,
    intros, apply (lambda_morphism.comp₁ a_1 a_2),
    intros, apply (lambda_morphism.ID₁ f),
    intros, apply lambda_morphism.assoc₁,
    intros, apply lambda_morphism.id_left₁,
    intros, apply lambda_morphism.id_right₁,
    intros, apply lambda_morphism.is_hset,
    intros, apply (lambda_morphism.comp₂ a_1 a_2),
    intros, apply (lambda_morphism.ID₂ f),
    intros, apply lambda_morphism.assoc₂,
    intros, apply lambda_morphism.id_left₂,
    intros, apply lambda_morphism.id_right₂,
    intros, apply lambda_morphism.is_hset,
    intros, apply lambda_morphism.id_comp₁,
    intros, apply lambda_morphism.id_comp₂,
    intros, apply lambda_morphism.zero_unique,
    intros, apply lambda_morphism.interchange,
    intros, apply (lambda_morphism.inv₁ a_1),
    intros, apply lambda_morphism.left_inverse₁,
    intros, apply lambda_morphism.right_inverse₁,
    intros, apply (lambda_morphism.inv₂ a_1),
    intros, apply lambda_morphism.left_inverse₂,
    intros, apply lambda_morphism.right_inverse₂,
    intros, fapply thin_structure.mk,
      intros, apply (lambda_morphism.T f g h i a_1),
      intros, apply lambda_morphism.thin_ID₁,
      intros, apply lambda_morphism.thin_ID₂,
      intros, apply lambda_morphism.thin_comp₁,
      intros, apply lambda_morphism.thin_comp₂,
end
```

After defining λ on morphisms we can instantiate it as a functor between categories with the same restriction to universe levels that we already saw in the definition of γ.

```
protected definition functor :
  functor Cat_xmod.{l₁ l₂ l₃} Cat_dbl_gpd.{(max l₁ l₂) l₂ l₃} :=
begin
  fapply functor.mk,
    intro X, apply (lambda.on_objects X),
    intros [X, Y, f], apply (lambda.on_morphisms f),
    intro X, cases X,
      fapply dbl_functor.congr', apply idp, apply idp,
      repeat ( apply eq_of_homotopy ; intros), cases x_8,
      fapply lambda_morphism.congr', apply idp,
      apply is_hset.elim,
    intros [X, Y, Z, g, f], cases X, cases Y, cases Z, cases g, cases f,
      fapply dbl_functor.congr', apply idp, apply idp,
      repeat ( apply eq_of_homotopy ; intros), cases x_8,
      fapply lambda_morphism.congr', apply idp,
      apply is_hset.elim,
end
```

5.7 Instantiating the Fundamental Double Groupoid

The general strategy for the instantiation of the fundamental double groupoid of a presented 2-type has been laid out in Definition 4.4.4. We start by building the fundamental groupoid of a 1-type A, a set C and a function $\iota : C \to A$ relating them:

```
definition fundamental_groupoid [reducible] : groupoid C :=
groupoid.mk
  (λ (a b : C), ι a =  ι b)
  (λ (a b : C), is_trunc_eq nat.zero (ι a) (ι b))
  (λ (a b c : C) (p : ι b = ι c) (q : ι a = ι b), q ⬝ p)
  (λ (a : C), refl (ι a))
  (λ (a b c d : C) (p : ι c = ι d) (q : ι b = ι c) (r : ι a = ι b),
    con.assoc r q p)
  (λ (a b : C) (p : ι a = ι b), con_idp p)
  (λ (a b : C) (p : ι a = ι b), idp_con p)
  (λ ⦃a b : C⦄ (p : ι a = ι b),
    @is_iso.mk C _ a b p (eq.inverse p) (!con.right_inv) (!con.left_inv))
```

We then continue to provide all the components for the double groupoid in the "flat" setting where we only consider points, equalities and iterated equalities in the 2-type X. This is done by basic manipulation of the given equalities. The vertical composition, for example, is given by the following calculation:

```
definition fund_dbl_precat_flat_comp₁ {a₁ b₁ a₂ b₂ a₃ b₃ : X}
  {f₁ : a₁ = b₁} {g₁ : a₂ = b₂} {h₁ : a₁ = a₂} {i₁ : b₁ = b₂}
  {g₂ : a₃ = b₃} {h₂ : a₂ = a₃} {i₂ : b₂ = b₃}
  (v : h₂ ⬝ g₂ = g₁ ⬝ i₂) (u : h₁ ⬝ g₁ = f₁ ⬝ i₁) :
  (h₁ ⬝ h₂) ⬝ g₂ = f₁ ⬝ (i₁ ⬝ i₂) :=
calc (h₁ ⬝ h₂) ⬝ g₂ = h₁ ⬝ (h₂ ⬝ g₂) : by rewrite con.assoc
               ... = h₁ ⬝ (g₁ ⬝ i₂) : by rewrite v
               ... = (h₁ ⬝ g₁) ⬝ i₂ : by rewrite con.assoc'
               ... = (f₁ ⬝ i₁) ⬝ i₂ : by rewrite u
               ... = f₁ ⬝ (i₁ ⬝ i₂) : by rewrite con.assoc
```

The corresponding axioms are proven by induction over the equalities involved. Here, one has to carefully choose the order in which to destruct the equalities, since the iterated equalities can only be destructed if their right hand side is atomic and since in Lean $p \cdot \mathsf{refl}$, but not $\mathsf{refl} \cdot p$ is judgmentally equal to p. The following example shows the proof of the interchange law by destruction in the order which is illustrated in Figure 5.1 (with non-reflexivity equalities shown as paths and iterated equalities as bounded areas).

```
variables
  {a₀₀ a₀₁ a₀₂ a₁₀ a₁₁ a₁₂ a₂₀ a₂₁ a₂₂ : X}
  {f₀₀ : a₀₀ = a₀₁} {f₀₁ : a₀₁ = a₀₂} {f₁₀ : a₁₀ = a₁₁} {f₁₁ : a₁₁ = a₁₂}
  {f₂₀ : a₂₀ = a₂₁} {f₂₁ : a₂₁ = a₂₂} {g₀₀ : a₀₀ = a₁₀} {g₀₁ : a₀₁ = a₁₁}
  {g₀₂ : a₀₂ = a₁₂} {g₁₀ : a₁₀ = a₂₀} {g₁₁ : a₁₁ = a₂₁} {g₁₂ : a₁₂ = a₂₂}
  (x : g₁₁ ⬝ f₂₁ = f₁₁ ⬝ g₁₂) (w : g₁₀ ⬝ f₂₀ = f₁₀ ⬝ g₁₁)
  (v : g₀₁ ⬝ f₁₁ = f₀₁ ⬝ g₀₂) (u : g₀₀ ⬝ f₁₀ = f₀₀ ⬝ g₀₁)

definition fund_dbl_precat_flat_interchange :
  fund_dbl_precat_flat_interchange_vert_horiz x w v u
  = fund_dbl_precat_flat_interchange_horiz_vert x w v u :=
begin
  revert v, revert f₀₁, revert g₀₂,
  revert u, revert f₀₀, revert g₀₁, revert g₀₀,
  revert x, revert f₁₁, revert g₁₂, revert f₂₁,
  revert w, revert f₁₀, revert g₁₁, revert g₁₀,
  cases f₂₀,
  intro g₁₀, cases g₁₀,
  intro g₁₁, cases g₁₁,
  intro f₁₀, intro w, cases w,
  intro f₂₁, cases f₂₁,
  intro g₁₂, cases g₁₂,
  intro f₁₁, intro x, cases x,
  intro g₀₀, cases g₀₀,
  intro g₀₁, cases g₀₁,
  intro f₀₀, intro u, cases u,
  intro g₀₂, cases g₀₂,
  intro f₀₁, intro v, cases v,
```

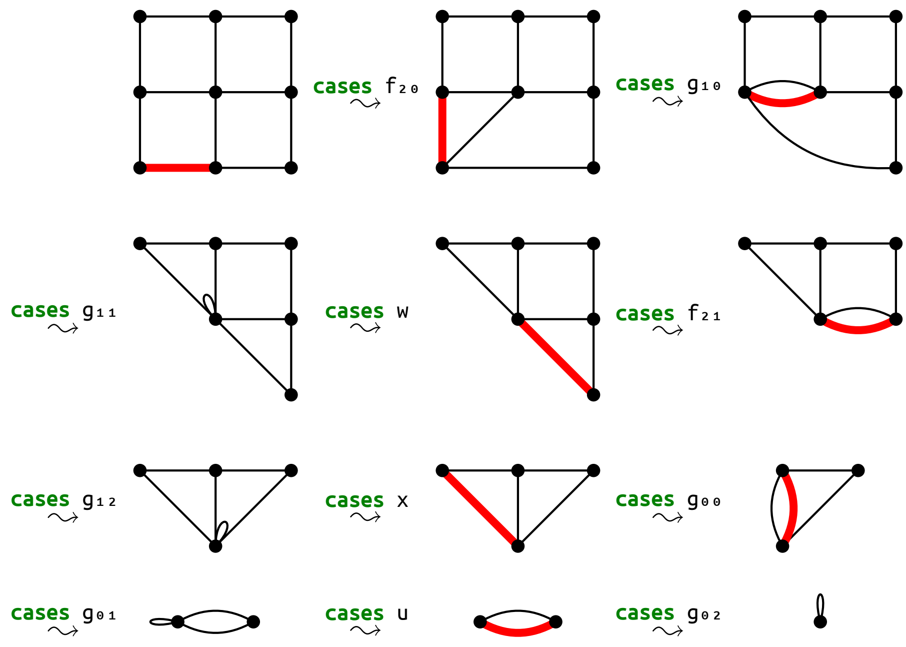


Figure 5.1: Destructing a grid of squares to prove the "flat" interchange law.

```
  apply idp,
end
```

The actual vertical composition is then derived from the "flat" version by transporting twice along the functoriality ap_con of path concatenation.

```
definition fund_dbl_precat_comp₁ ...
  (v : ap ι' h₂ ⬝ ap ι' g₂ = ap ι' g₁ ⬝ ap ι' i₂)
  (u : ap ι' h₁ ⬝ ap ι' g₁ = ap ι' f₁ ⬝ ap ι' i₁) :
    ap ι' (h₁ ⬝ h₂) ⬝ ap ι' g₂ = ap ι' f₁ ⬝ ap ι' (i₁ ⬝ i₂) :=
((ap_con ι' i₁ i₂)⁻¹) ▹ ((ap_con ι' h₁ h₂)⁻¹) ▹
@fund_dbl_precat_flat_comp₁ X A C Xtrunc Atrunc Cset
  (ι' (ι a₁)) (ι' (ι b₁)) (ι' (ι a₂)) (ι' (ι b₂)) (ι' (ι a₃)) (ι' (ι b₃))
  (ap ι' f₁) (ap ι' g₁) (ap ι' h₁) (ap ι' i₁)
  (ap ι' g₂) (ap ι' h₂) (ap ι' i₂) v u
```

Then we can prove the axioms for the double groupoid performing step 2 and 3 from the proof of Definition 4.4.4. Like in the following example of vertical associativity, we first use the flat version of the axiom and in the end refer to another auxiliary lemma that allows for path induction in the type A.

```
definition fund_dbl_precat_assoc₁_aux {a₁ a₂ a₃ a₄ b₁ b₂ b₃ b₄ : A}
  (f₁ : a₁ = b₁) (g₁ : a₂ = b₂) (h₁ : a₁ = a₂) (i₁ : b₁ = b₂)
  (g₂ : a₃ = b₃) (h₂ : a₂ = a₃) (i₂ : b₂ = b₃) (g₃ : a₄ = b₄)
  (h₃ : a₃ = a₄) (i₃ : b₃ = b₄)
  (w : (ap ι' h₃) ⬝ (ap ι' g₃) = (ap ι' g₂) ⬝ (ap ι' i₃))
  (v : (ap ι' h₂) ⬝ (ap ι' g₂) = (ap ι' g₁) ⬝ (ap ι' i₂))
  (u : (ap ι' h₁) ⬝ (ap ι' g₁) = (ap ι' f₁) ⬝ (ap ι' i₁)) :
  (transport (λ x, ((ap ι' h₁) ⬝ x) ⬝ (ap ι' g₃) = _) (ap_con ι' h₂ h₃)
   (transport (λ x, _ =  ((ap ι' f₁) ⬝ ((ap ι' i₁) ⬝ x))) (ap_con ι' i₂ i₃)
    (transport (λ x, (x ⬝ (ap ι' g₃)) = _) (ap_con ι' h₁ (concat h₂ h₃))
     (transport (λ x, _ = (ap ι' f₁) ⬝ x) (ap_con ι' i₁ (concat i₂ i₃))
      (transport (λ x, _ = (ap ι' f₁) ⬝ (ap ι' x)) (con.assoc i₁ i₂ i₃)
       (transport (λ x, (ap ι' x) ⬝ _ = _) (con.assoc h₁ h₂ h₃)
        (transport (λ x, _ =  (ap ι' f₁) ⬝ x) (ap_con ι' (concat i₁ i₂) i₃)⁻¹
         (transport (λ x, x ⬝ (ap ι' g₃) = _) (ap_con ι' (concat h₁ h₂) h₃)⁻¹
          (transport (λ x, _ = _ ⬝ (x ⬝ _)) (ap_con ι' i₁ i₂)⁻¹
           (transport (λ x, (x ⬝ _) ⬝ _ = _) (ap_con ι' h₁ h₂)⁻¹
            (transport (λ x, x ⬝ _ = _)
              (con.assoc (ap ι' h₁) (ap ι' h₂) (ap ι' h₃))⁻¹
              (transport (λ x, _ = _ ⬝ x)
                 (con.assoc (ap ι' i₁) (ap ι' i₂) (ap ι' i₃))⁻¹
                 (fund_dbl_precat_flat_comp₁
                   (fund_dbl_precat_flat_comp₁ w v) u))))))))))))
   = (fund_dbl_precat_flat_comp₁ (fund_dbl_precat_flat_comp₁ w v) u) :=
begin
  ...
end

definition fund_dbl_precat_assoc₁ :
  (con.assoc i₁ i₂ i₃) ▹ (con.assoc h₁ h₂ h₃) ▹
     (fund_dbl_precat_comp₁ w (fund_dbl_precat_comp₁ v u))
  = fund_dbl_precat_comp₁ (fund_dbl_precat_comp₁ w v) u :=
begin
  unfold fund_dbl_precat_comp₁,
  apply tr_eq_of_eq_inv_tr, apply tr_eq_of_eq_inv_tr,
  apply inv_tr_eq_of_eq_tr, apply inv_tr_eq_of_eq_tr,
  apply concat, apply fund_dbl_precat_flat_transp1, apply inv_tr_eq_of_eq_tr,
  apply concat, apply fund_dbl_precat_flat_transp2, apply inv_tr_eq_of_eq_tr,
  apply concat, apply fund_dbl_precat_flat_assoc₁', -- Call flat version
  apply eq_tr_of_inv_tr_eq, apply eq_tr_of_inv_tr_eq,
  apply eq_tr_of_inv_tr_eq, apply eq_tr_of_inv_tr_eq,
  apply eq_inv_tr_of_tr_eq, apply eq_inv_tr_of_tr_eq,
  apply eq_inv_tr_of_tr_eq, apply eq_inv_tr_of_tr_eq,
  apply inverse,
  apply concat, apply fund_dbl_precat_flat_transp3, apply inv_tr_eq_of_eq_tr,
  apply concat, apply fund_dbl_precat_flat_transp4, apply inv_tr_eq_of_eq_tr,
  apply inverse, apply fund_dbl_precat_assoc₁_aux, -- Call half-flat lemma
end
```

Thin squares are defined by the following calculation using, again, the functoriality of ap.

```
definition fund_dbl_precat_thin {a b c d : C}
  {f : ι a = ι b} {g : ι c = ι d} {h : ι a = ι c} {i : ι b = ι d}
  (comm : h ⬝ g = f ⬝ i) :
  ap ι' h ⬝ ap ι' g = ap ι' f ⬝ ap ι' i :=
calc ap ι' h ⬝ ap ι' g = ap ι' (h ⬝ g) : ap_con
                   ... = ap ι' (f ⬝ i) : comm
                   ... = ap ι' f ⬝ ap ι' i : ap_con
```

Using the same strategy for the axioms for this thin structure as for the axioms of the weak double groupoid lets us finally conclude by defining the fundamental double groupoid:

```
definition fundamental_dbl_precat : dbl_gpd (fundamental_groupoid)
  (λ (a b c d : C)
    (f : ι a = ι b) (g : ι c = ι d) (h : ι a = ι c) (i : ι b = ι d),
    ap ι' h ⬝ ap ι' g = ap ι' f ⬝ ap ι' i) :=
begin
  fapply dbl_gpd.mk, ...
  fapply thin_structure.mk, ...
end
```

Theory	Line Count	Compilation Time in s
`transport4`	49	0.333
`dbl_cat.`		
`basic`	224	5.272
`decl`	58	4.253
`dbl_gpd.`		
`basic`	199	6.375
`category_of`	41	1.314
`decl`	69	8.856
`functor`	582	47.997
`fundamental`	1270	21.091
`equivalence.`		
`equivalence`	371	*30.049
`gamma_functor`	51	6.109
`gamma`	164	6.054
`gamma_group`	229	5.973
`gamma_morphisms`	167	5.986
`gamma_mu_phi`	407	24.828
`lambda_functor`	27	4.086
`lambda`	569	16.148
`lambda_morphisms`	70	7.914
`thin_structure.`		
`basic`	154	3.091
`decl`	33	0.766
`xmod.`		
`category_of`	25	0.327
`decl`	53	0.794
`morphism`	209	4.542

Table 5.4: The theories of my formalization project. Theories marked with * contain unfinished proofs.

Chapter 6

Conclusion and Future Work

Summarizing the previous chapters, my work consists of translating the algebraic structures of double groupoids and crossed module to set based structures in homotopy type theory. I transferred the principal example of a fundamental double groupoid of a triple of spaces from the topological setting to its equivalent in the world of higher types by defining the fundamental double groupoid of a presented 2-type. I formalized the structures of double categories, double groupoids and crossed modules in the new theorem proving language Lean and mechanized the essential parts of the proof that double groupoids with thin structures and crossed modules form equivalent categories. I furthermore made the formalized structures applicable to 2-truncated higher types by instantiating the fundamental double groupoid of such types.

I hereby made it possible to analyze presented 2-types in purely set-based algebraic structures. This opens up the analysis of second homotopy groups or second homotopy groupoids of 2-types and the characteristics of these groups and groupoids to the use of formalized group theoretical and category theoretical knowledge. This could lead to direct computation of several homotopy invariants. Being one of the first greater formalization projects in Lean, the problems encountered during the process of writing the formal definitions and proofs led to improvements in the performance and usability of Lean. With respect to their compilation time, my theory files also serve as a benchmark for the elaboration and type checking algorithms used in Lean.

What are the main insights and experiences gained from this work? Many of the difficulties in writing the formalization are certainly due to the early development stage of the system used. During my work, Lean's features for structure definition, type class inference, tactics, as well as troubleshooting and output were vastly enhanced and improved. Also, the time Lean needs to elaborate and type check the theory files have decreased drastically since the start of the project. The library of category theory and group theory that were developed alongside the actual formalization project and are still work in progress. Another big hurdle was the management of transport terms in my proofs. Even actually trivial proofs involving equalities of two-cells in double categories and double groupoids turned out to be long and tedious due to the need of moving transport terms from the inside of an operator or a function to its outside. This is, of course, the price one has to pay for the heavy use of dependently typed two-cells in double categories. After gaining experience on what auxiliary lemmas were needed and how to use them, this burden of moving transport terms was reduced to a mere strain on the theorem prover and a part of the theories that made the files longer and less readable. Finally, some parts of the proofs which were not stated explicitly but instead left out as "trivial" in my main reference [BHS11] turned out to be more sophisticated than initially anticipated.

There are several points where I could have made a different decision that would have led to different results in the complexity and the character of my formalization. As mentioned in Chapter 4, the decision on whether to formalize the higher cells of a double category as dependent types or as flat types with face operators is a difficult one. Deciding for flat types would have prevented the need for many auxiliary lemmas involved in the effort to control transport terms in equalities of two-cells. It is hard to judge whether a flat typed approach would have led to longer proofs and less readable definitions or if it made the formalization cleaner and shorter. Another way of preventing the need of said auxiliary lemmas and of applying lemmas to move transport terms to different sides of equations would be using what Daniel Licata calls "pathovers" and "squareovers" in his paper [LB] describing a strategy for the proof that, as a higher inductive type, the torus is equivalent to the product of two circles. These encapsulate the type of equalities like those of the form $p_*(x) = y$ as objects, formalized as an inductive type. Another solution would be to switch to a different logic that allow postulating judgmental equalities, e.g. the cubical identities (3.1). One example for such a logical framework might by Vladimir Voevodsky's Homotopy Type System (HTS) [Voe]. This system might make it possible to generalize my formalization to the case of cubical ω-groupoids and crossed complexes – something which, with my current approach – is not possible since there is no uniform way to describe the dependent types of n-cells for all $n \in \mathbb{N}$.

This leads to the question what could be a possible way to continue and extend my project. The most obvious use of the formalization would be a 2-dimensional Seifert-van Kampen theorem for presented 2-types. In its most common form, such a theorem would state that the category theoretical pushout of the fundamental double groupoid of two presented 2-types is isomorphic to the fundamental double groupoid of the pushout of those presented 2-types in the form of a higher inductive type. Then, one could search for ways to find "reasonable" presentations for 2-types, homotopy surjective ones come to mind, either manually or automatically at the time of the definition of a higher inductive type. Such an automation could then be part of the definitional package of an interactive theorem prover that provides these higher inductive types as a primitive. As mentioned in the above paragraph, the most important generalization of my work would consist of replacing the "double" in "double groupoid" by "n-fold" for an arbitrary $n \in \mathbb{N}$ or, as an ultimate goal, by the case of ω-groupoids that contain higher cells for every dimension. Finally, one could ask if there are any applications of Ronald Brown's attempts to "compute" crossed modules induced by subgroups [BW96] to computable homotopy characteristics of higher types.

Bibliography

[AKS13] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. *Mathematical Structures in Computer Science*, pages 1–30, 2013. (Cited on page 43.)

[BBC$^+$97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997. (Cited on page 67.)

[BHS11] Ronald Brown, Philip J Higgins, and Rafael Sivera. *Nonabelian Algebraic Topology: Filtered spaces, crossed complexes, cubical homotopy groupoids*. European Mathematical Society, 2011. (Cited on pages 2, 23, and 108.)

[BW96] Ronald Brown and Christopher D Wensley. Computing crossed modules induced by an inclusion of a normal subgroup, with applications to homotopy 2-types. *Theory Appl. Categ*, 2(1):3–16, 1996. (Cited on page 109.)

[dMKA$^+$] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover. submitted. (Cited on page 2.)

[Dyb94] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994. (Cited on pages 8 and 64.)

[HAB$^+$15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, et al. A formal proof of the Kepler conjecture. *arXiv preprint arXiv:1501.02155*, 2015. (Cited on page 1.)

[hota] Homotopy type theory. `https://github.com/HoTT/HoTT`. Accessed: April 24, 2015. (Cited on pages 1 and 71.)

[hotb] Homotopy type theory in agda. `https://github.com/HoTT/HoTT-Agda`. Accessed: April 24, 2015. (Cited on page 1.)

[LB] Daniel R Licata and Guillaume Brunerie. A cubical approach to synthetic homotopy theory. (Cited on page 108.)

[ML98] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998. (Cited on page 3.)

[Nor09] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009. (Cited on page 67.)

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. (Cited on pages 3, 8, 12, 15, 21, 43, and 62.)

[Voe] Vladimir Voevodsky. A simple type system with two identity types. `http://uf-ias-2012.wikispaces.com/file/view/HTS.pdf`. Accessed: April 22, 2015. (Cited on page 108.)