Exact Couples in Cubical Type Theory

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Michael Zhang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Favonia, Advisor

August 2025

Acknowledgements

I am deeply grateful to my advisor Favonia for introducing me to the wonderful world of higher-dimensional type theory, and for their patience, guidance, and support throughout my master's degree. I would like to thank Axel Ljungström for guiding me toward this research direction. I would like to thank the members of the programming languages lab at UMN for the advice and companionship. Finally, I would like to thank my girlfriend and family for their love and encouragement during this time.

Abstract

In homotopy theory, spectral sequences are widely used tools for computing homology and cohomology groups by approximating them iteratively. Exact couples are a key algebraic abstraction at the heart of many spectral sequences, such as the Serre and Adams spectral sequences, handling the complexity of the iteration process. Constructing these spectral sequences by hand is often a complex process, so computer verification of the results is critical to achieving a high level of mathematical rigor.

Cubical type theory is a formal language for computer proof assistants. It is based on homotopy type theory, whose notion of equality is borrowed from paths in homotopy theory. As such, it is a natural setting for computer-verified constructions in homotopy theory. We present a construction for exact couples using cubical type theory. Our construction is mechanized in Cubical Agda, a proof assistant that can check the correctness of code. Our work establishes the necessary type theoretic formulations for exact couples and sets the foundation for further formalization of spectral sequences in cubical type theory.

Table of Contents

1 Introduction

In algebraic topology and homotopy theory, spectral sequences, first introduced by Jean Leray [Ler46a, Ler46b] are tools for iteratively computing challenging (co)homology groups from known groups. Throughout the 20th century, many different spectral sequences have been discovered, such as the Serre spectral sequence, which computes (co)homology groups of the total space of a fibration, and the Adams spectral sequence, which computes homotopy groups of spheres. Construction of many of these spectral sequences involves finding the initial conditions for algebraic structure known as an exact couple, which can be iterated to produce the entire spectral sequence. Exact couples [W. 52, W. 53] serve as an important abstraction for iterating a spectral sequence. Due to the complexity, computer verification is useful for confirming the correctness of constructions to a high level of rigor. This thesis presents a mechanized construction of exact couples in cubical type theory using the proof assistant Cubical Agda, laying the groundwork for future mechanized constructions of spectral sequences.

Type theory is a formal language, similar to Zermelo-Fraenkel (ZF) set theory, that acts as a foundation upon which mathematical objects can be built. Unlike ZF set theory, type theory appeals more to the computational nature of programming languages. Due to the syntactic nature of the inference rules, checking the correctness of type is often a much more efficient process than checking correctness of proofs in set theory since it can more quickly reject ill-typed terms. [Bau20]

One version of type theory is homotopy type theory (HoTT) [Uni13], which inspiration from homotopy theory. HoTT features higher-dimensional identity types known as paths, which borrows directly from paths in homotopy theory. Additionally, HoTT introduced the univalence axiom, relating weak homotopy equivalence with identity in a type theoretic sense. This makes reasoning about higher-dimensional structures much more natural than in other type theories. HoTT also introduced higher inductive types, which give yet another

way of constructing higher-dimensional paths, and also give a clean formalization of quotients types which we will use. However, these concepts are only introduced axiomatically in HoTT, meaning the theory itself cannot reduce terms derived from univalence or path constructors from higher inductive types. As a consequence, HoTT does not exhibit canonicity, a property of programming languages that all terms reduce to a canonical form for its type. Cubical type theory [Coh+15] is a type theory that improves on HoTT by providing a computational interpretation for univalence and higher inductive types, making it well-suited for doing mechanized proof.

This thesis follows the results and future directions of Floris van Doorn's dissertation [Doo18]. Our results consist of a formal construction of exact couples and proof of exactness of the derived couples. These results have been mechanized using Cubical Agda [VMA21], a proof assistant that implements a variant of cubical type theory that includes computational univalence and higher inductive types. Code snippets from Agda may be included in the paper for demonstration – readers are not expected to understand Agda syntax to understand the paper. The code repository containing the results can be found at https://git.sr.ht/~mzhang/msthesis, and have been verified with Agda v2.7.0.

The structure of this thesis is: Section 2 gives the background in type theory, and algebra necessary for the rest of the paper, Section 3 describes the construction itself, and Section 4 concludes with some remarks about mechanization in general.

2 Background

2.1 Type theory

The current established language of formal mathematics is set theory with the Zermelo-Fraenkel (ZF) axioms. However, the rise of computer involvement in proof checking and development revealed additional demands. Type theory is an alternative foundation for mathematics that has properties more amenable to computer programs. It is built on a system of syntactic inference rules, which allows for more efficient type checking algorithms. Strict typing ensures that ill-typed expressions are rejected early.

Earlier interest in computerized mathematical proof focused primarily on automatic theorem proving (ATP), where the program automatically completes the proof [HUW14]. However, there are limits to the complexity of proofs that even the most powerful automated solvers are able to handle. Thus, ATP style solvers are typically deployed in less expressive logics or limited scenarios where search is tractable. These limitations – particularly for proofs requiring creative insight or novel constructions – necessitated a different approach. Interactive theorem proving (ITP) is primarily human-driven, but is "interactive" or "computer-aided" in the sense that computer programs known as proof assistants (or theorem provers) are deeply intertwined in the proof process. ITP allows for more expressive languages such as dependent type theory, since the burden is on verification rather than finding a solution.

One of the cornerstone developments in type theory is Per Martin-Löf's intuitionistic type theory, also known as Martin-Löf type theory (MLTT) [Mar75]. MLTT is rooted in the idea that computation and logic are fundamentally the same, known as the Curry-Howard correspondence [Cur34, How80]. Under this correspondence, types represent logical propositions, and terms of those types represent proofs. This provides the justification that type checking algorithms indeed have mathematical value. There has been extensive ongoing

research efforts to mechanize existing mathematical results, which has led to many libraries of mathematical knowledge [Bru83, Com20, Con+86, The24].

The results in this paper are mechanized using Agda, an ITP-style proof assistant that uses a variant of type theory called cubical type theory [Coh+15]. Cubical type theory is primarily based on homotopy type theory (HoTT) [Uni13], with which it shares the majority of its semantics. HoTT interprets identity types in type theory as paths in homotopy theory, and provides ways of constructing paths such as with higher inductive types and the univalence principle, which unifies the ideas of equivalence and identity. However, cubical type theory has better computational properties: whereas in HoTT univalence and higher inductive types must be assumed as axioms, cubical type theory allows it to be proven as a theorem, since univalence directly computes on the paths. Where the model of HoTT is simplicial sets, cubical type theory uses cubical sets, which makes it easier to give paths computational value. We will give a brief introduction to cubical type theory below.

2.1.1 Inference rules and contexts

Before introducing any mathematical objects, we must first discuss the logic behind the rules. Unlike ZF set theory, which is based on first-order logic, type theory comes with its own deduction system consisting of judgements and rules. Judgements are assertions of truth. Inference rules describe how certain judgements can be achieved. For example, the following rule shows that judgements J_1, J_2, and J_3 are required for judgement J to hold:

$$\frac{J_1 \quad J_2 \quad J_3}{J}$$

We say J is "derivable" if there exists a derivation tree with J at the bottom. Judgements primarily deal with meta-theoretic statements such as manipulation of contexts and well-formedness of objects.

Since type theory is syntactic, we must have a context to hold syntactic variables. The conventional variable name for contexts is Γ. We call information that contexts hold "assumptions." The empty context is known as $(\cdot)$, and (Γ, x) represents adding the assumption x to the context. Although this may look like a list at first, many type theories typically come with rules governing contexts, known as structural rules, to allow for more flexible manipulation. The standard structural rules are:

- Weakening, allowing the addition of unused assumptions
- Contraction, allowing for the removal of duplicate assumptions
- Exchange, allowing for re-ordering of assumptions

With this understanding, we can describe the primary judgements used in type theory.

Γ ctx	Γ is a valid context. All further rules involving Γ will implicitly assume that Γ is a valid context.
$\Gamma \vdash A$ type	Under context Γ, A represents a valid type.
$\Gamma \vdash x : A$	Under context Γ, x represents an element of A. Implicitly assumes A is valid under Γ.
$\Gamma \vdash x \equiv y : A$	Under context Γ, x and y represent definitionally equal terms of type A. Implicitly assumes A is valid under Γ.

The context is frequently elided for brevity when there is a single obvious ambient context. For instance, the two rules below are understood to have the same meaning:

$$\frac{A \text{ type} \quad B \text{ type}}{A \times B \text{ type}} \qquad \frac{\Gamma \text{ ctx} \quad \Gamma \vdash A \text{ type} \quad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \times B \text{ type}}$$

Under these judgements, types are created using four kinds of rules:

Name	**Description**
Type formation	Describes when a type is valid.
Constructor	Describes how to create terms of this type.
Eliminator	Describes the ways of "using" a term of the type.

Name	Description
Computation rule	Describes what the result of applying the eliminator will be.

There is often also a *uniqueness* rule for certain types, stating that all terms of that type can be represented in a canonical form. For instance, any function f is equal to the corresponding term using its canonical constructor $\lambda x.f(x)$.

A full formal presentation of all the rules necessary to describe MLTT including structural rules as well as basic types is given in the Appendix A.2 of the homotopy type theory book (HoTT book) [Uni13], so we will briefly walk through the notation of types we will encounter in the rest of the paper.

- $A \to B$. Function type. Terms are constructed using lambda notation $\lambda x.(x + 1)$, and eliminated by applying it: $f(1)$.
- $(x : A) \to B(x)$. Dependent function type. This is a generalization of the previous function type. In other accounts, this is written as $\prod_{x:A} B(x)$; however, we will stick to the arrow notation, since it is slightly easier to read, and is more similar to the notation used by the Agda proof assistant.
- $A \times B$. Cartesian pair type. Terms are constructed as (a, b) with $a : A$ and $b : B$, and eliminated by projections fst and snd.
- $\sum_{x:A} B(x)$. Dependent pair type. This is a generalization of the previous pair type, but the type of the second element can depend on the value of the first. The pattern of having the first element be a type and the second being a proposition of that type is common in subtypes. For example, $\sum_{x:\mathbb{N}} \mathsf{isEven}(x)$ may represent the type of all even numbers since odd x would have the empty type for the second element.
- $\mathbb{0}$. Empty type. This is the type with no elements. In type theory, under the Curry-Howard correspondence, it represents logical falsity, as a proposition that cannot be proven will have no inhabitants.

- $\mathbb{1}, \mathbb{2}, ...$ Unit type, boolean type, etc. These are types with a fixed number of elements in them. There is a single constructor for each term. The type is eliminated by case matching.
- $\mathbb{N}$. Natural number type. This is an inductively defined type with constructors $\mathsf{zero} : \mathbb{N}$, representing zero, and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$, representing the successor operation.
- $x =_A y$. Path type. We will discuss this in more detail in the next section.

Aside from several of the above types, we are also able to define inductive types by specifying constructors that take a term of the type as input. For instance, a tree structure may be defined by creating a base constructor "leaf", and inductive constructor "branch" which holds two trees. As we will see later, it is also possible to define inductive types that allow construction of paths in the type, known as higher inductive types.

2.1.2 Paths

One of the most important notions in mathematics is equality. An intensional type theory such as cubical type theory has two primary notions of equality.

Definitional equality is defined by the reductions built into the inference rules. It is also known as judgemental equality as it operates on the judgement level. Nothing inside the theory can distinguish two objects that are definitionally equal. We use this notation $x \equiv y$ to denote that terms x and y are definitionally equal.

On the other hand, **propositional** equality is internal to the language. The propositional equality type $x =_A y$ is a standard type that can be constructed, manipulated, and eliminated from within the theory. Cubical type theory is based on homotopy type theory, which interprets types as topological spaces and identity types as paths in those spaces. It turns out paths in topological spaces are particularly good for representing identity types, since they capture both the expected properties of equality and provide higher-dimensional paths for applications such as quotients and doing synthetic homotopy theory.

Given some type A, a **path** between x and y is a type $x = y$ that represents their propositional equality. The path definition mirrors that from homotopy theory – it can be thought of as a map from the continuous interval $[0, 1]$ to A. The interval is denoted $\mathbb{I}$. When the path is applied at 0, it yields x, and when applied at 1, it yields y. We write this as $p(0) \equiv x$ and $p(1) \equiv y$.

The interval can also be a variable. Conventionally, we use $i, j, k, ...$ to represent interval variables. Creating paths is done by specifying an expression over the interval variables. For example, the identity path, or reflexive path also called refl for short, is a path that goes from some point x back to itself. It can be defined by ignoring the interval variable, as follows:

$$\mathsf{refl} : x =_A x$$

$$\mathsf{refl}(i) \equiv x$$

Remark 2.1.2.1: Although this seems to behave similarly to a function $\mathbb{I} \to A$, and Cubical Agda actually uses the same function notation to interface with it, paths are not simply the same as the function. Critically, paths come with extra Kan operations for composition and filling which we will discuss more later. We will continue to use function notation to refer to path application.

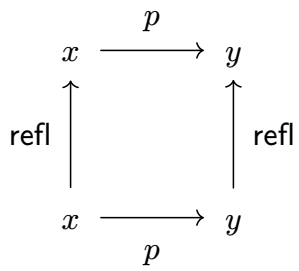
We also have a dependent version of the path type, or "path over path", named PathP. Instead of having a fixed constant $A : \mathsf{Type}$, the dependent path operates over a type family $A : \mathbb{I} \to \mathsf{Type}$. Then, given $x : A(0)$ and $y : A(1)$, we have a path $\mathsf{PathP}(A, x, y)$. A non-dependent path is simply a special case of a dependent path, where the type family A is simply a constant function that does not depend on the argument. Dependent paths allow for expressing propositional equalities between values of paths that have been identified, and are important for representing the equality between dependent pairs, as we will see later.

Since paths are continuous by the topological definition, conceptually we should be able to apply it at any real number point between 0 and 1. However, in the type theory we only care about the endpoints and do not need any meaningful way of accessing the middle points. Path expressions must be continuous over their domain, meaning they cannot arbitrarily choose one expression for the case of 0 and another expression for the case of 1. Thus, we are restricted in the ways we can construct paths; we can only construct paths between equal terms.

We also have several ways of extending a path to a higher dimension. A trivial method is to create a higher-dimensional path that ignores one of the interval coordinate variables. For example, given a one-dimensional path $p : x = y$, we can trivially create a two-dimensional path with:

$$p'(i, j) \equiv p(i)$$

This can be visualized by a square that looks like this:



This path is actually definitionally the same as p. In order to create more interesting paths, we will need some new techniques. First, there are some operations on the interval. The **negation** of an interval variable $\neg i$ turns 0 to 1 and 1 to 0. The **meet** ($\wedge$) of two variables represents their minimum. The **join** ($\vee$) of two variables represents their maximum.

We can use these operations when constructing higher-dimensional paths to create "squares", or 1-cubes, out of a path. For example, given a path $p : x = y$, we can create the following squares:

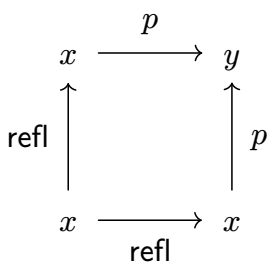


Figure 4: $p(i \wedge j)$

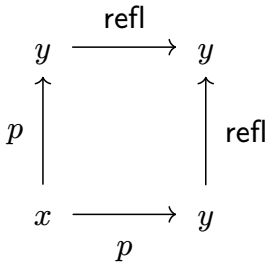
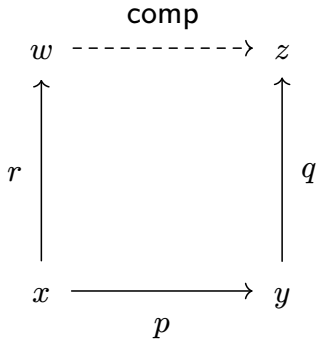


Figure 5: $p(i \vee j)$

We can create squares using any combination of these operations, like $p(\neg i \wedge j)$ will create a version of the meet square with the i direction flipped. Again, all of these operations reduce definitionally at the specified endpoints.

There are a few operations related to cube-filling. These represent the Kan operations of composition and filling from the underlying cubical set model. We will present these operations directly from the cubical type theory perspective rather than appealing to the cubical set notation.

The core Kan operation is the **composition** operation, which completes a cube given enough boundary data from other faces. The final face represents the composition of the paths. For example, consider the following square:



Given paths $p : x = y$, $q : y = z$, and $r : x = w$, the composition operation gives us the missing face of the square. This idea extends into higher-dimensional boxes, which require more faces to constrain the missing face. This is what is used to define the **filling** operation,

which completes the interior of the cube given enough boundary data. In the diagram above, this corresponds to filling the interior of the square. Intuitively, this is done by extending the cube into a higher dimension, and then using the composition operation to complete the missing face corresponding to the interior.

Next are some common functions used to manipulate paths. The **transport** function captures the notion of *indiscernibility of identicals* – that two identical objects can be interchanged in any expression. If P is a type family indexed over A, and $p : x =_A y$, then we say $\mathsf{transport}^P(p, -) : P(x) \to P(y)$. This is also sometimes called substitution, since it is essentially replacing instances of x with y. Its definition is also derived from composition – it is filling the missing face of a one-dimensional cube.

Next, **path induction** is the standard eliminator for identity types that have been defined inductively in the style of HoTT. While path types aren't inductive, they still obey the same rule, which states that in order to eliminate an identity type $p : x = y$, it suffices to provide the action taken in the case of $\mathsf{refl} : x = x$. This is also known as the J rule. In cubical type theory, J can be implemented using $\mathsf{transport}$.

There are also some important interactions between paths and functions. First, ap (read: application) captures the functoriality of functions over paths. That is to say, applying a function to both sides of an identity preserves the identity. For some function $f : A \to B$, and path $p : x =_A y$, we have $\mathsf{ap}_f(p) : f(x) =_B f(y)$.

The **function extensionality** property identifies two functions as long as they are point-wise identical in behavior for every possible input. For some functions $f, g : A \to B$, if for any $x : A$, $f(x) =_B g(x)$, then we say $f = g$. This extensionality property was not possible to prove with the inductive identity types used in intensional type theories such as HoTT. However, with cubical paths being function-like, the point-wise identity is $A \to \mathbb{I} \to B$ while the function-level identity is $\mathbb{I} \to A \to B$, we can simply swap the variables to achieve the

property.[1]

2.1.3 Universes

In order to treat types as first-class objects in the theory, we must have a type of all types. Allowing this type of all types naively leads to contradiction, due to a discovery known as Girard's paradox. This led to the stratification of types of types, also known as *universes*. The type of all types is known as Type_0, which itself is an element of Type_1, which is of type Type_2, and so on. (Some other accounts use the notation $\mathcal{U}_i$ for universe)

Unfortunately, this means types themselves can only be a member of a single universe. If we want to describe a universe-1 natural number type, this would not be the same as the universe-0 natural number type. Proof assistants typically employ extra rules in their elaborator to relieve this issue, as simply allowing a type to inhabit multiple universes leads back to the contradiction. The standard way to deal with this is through *universe polymorphism*, which allows quantification over and manipulation of universe levels. Of course, there are rules governing how universe levels can be used. For instance, consider the function type $A \to B$. If A lives in universe Type_a and B lives in universe Type_b, the function must live in universe $\mathsf{Type}_{a \sqcup b}$, where $a \sqcup b$ represents the least upper bound of a and b. There is also a type Lift, which is able to increase the universe level of any type. However, this is an explicit coercion.

The mechanization requires careful handling of universe levels in order to be blessed by the proof assistant's type checker. But we will not need to do anything interesting with universe levels, so for the rest of the paper, we will elide the universe levels for the sake of brevity.

2.1.4 Higher inductive types

Higher inductive types (HITs) are among one of the only sources of non-trivial higher-dimensional paths in homotopy type theory. An ordinary inductive type, such as $\mathbb{N}$,

[1] Notation abuse here. As noted in Remark 2.1.2.1 the path isn't strictly a function, but the method still works.

consists of constructors that may depend on the entire inductive type. We call these constructors "point" constructors, since they construct points in the space corresponding to the inductive type. However, the only paths that can exist in this space are trivial paths. HITs give us "path" constructors, which create non-trivial paths.

For instance, the circle S^1 is defined as a single point with a non-trivial path, which represents the loop. As a higher inductive type, we represent this as two constructors:

$$\mathsf{base} : S^1$$

$$\mathsf{loop} : \mathsf{base} = \mathsf{base}$$

An eliminator for the circle must handle both cases. Suppose we are writing an eliminator $f : S^1 \to C$. Intuitively speaking, an element of S^1 can be either base or lie somewhere along the loop. Our base case goes to $f(\mathsf{base})$, and the loop must be mapped to a path in C such that the endpoints agree with $f(\mathsf{base})$. Thus, we must provide the loop's action on paths: $\mathsf{ap}_f(\mathsf{loop})$.

Based on the version of HoTT described by the HoTT book (known as "book HoTT"), the path constructors, eliminators, and computation rules for higher inductive types needed to be axiomatized. Getting the eliminators and computation rules correct takes meticulous effort, and is incredibly error-prone, since it is not checked by the proof assistant. Additionally, the computation rules can only be propositional.

However, one of the major developments of cubical type theory is computational semantics for higher inductive types. Since paths are simply maps over the interval, a higher inductive type's eliminator computes definitionally on the endpoints.

Higher inductive types are central to constructing many of the spaces studied in algebraic topology, such as spheres, wedges, and Eilenberg-MacLane spaces, as well as other useful types such as truncations and quotients.

2.1.5 Homotopy levels

Homotopy levels, also known from algebraic topology as n-types, describe the amount of "interesting" homotopical information exists in a type. As mentioned previously, some spaces can exhibit higher-dimensional paths – paths over paths. Describing a space as an n-type characterizes what is the highest dimension of path that is not trivial in this space.

The n in n-type refers to integers starting at -2. This is due to a 0-type being a set. However, in a formal system, this definition incredibly inconvenient for proving properties by induction, since we would have to carry around the proof of $n \geq -2$. It is much easier to shift the definition such that the base case is -2, which makes this indexing set the same as the natural numbers but shifted by -2. The name homotopy level, originating from HoTT, simply uses the natural numbers, with 0 denoting contractible types and 2 denoting sets. To avoid this confusion, we will use homotopy level numbering going forward.

Here is a table of the first few homotopy levels, and a description of types that they classify:

Level	**Name**	**Description**
0	*contractible type*	This type is equivalent to the unit type.
1	*mere proposition*	Cannot distinguish elements apart.
2	*set*	Cannot distinguish paths between elements apart.
3	*groupoid*	Cannot distinguish 2-paths apart.

Mere propositions and sets are of particular interest to the mechanization of mathematics, specifically algebra. Mere propositions (or "Props") are so named because they exhibit the property that if inhabited, all elements are the same. This aligns with our usual notion of propositions – we typically do not care *how* the proposition is proven, simply that it is either inhabited (true) or uninhabited (false). In fact, we will often say we will prove a proposition to simply mean constructing an element of a mere proposition type.

Sets are so named because they mirror the sets of set theory. One of the differences between sets of set theory and types of homotopy type theories is that types come with an infinite

hierarchy of path types, while set theory has the principle of *uniqueness of identity proofs* (UIP), which says any two elements can only be proven identical in one way. A type with homotopy level 2 exhibits this same property, since paths in such types are mere propositions.

Property 2.1.5.1: If a type A has homotopy level n, then paths in A will have homotopy level $n-1$.

Truncations can be used to artificially reduce the homotopy level of a type. A truncation can be thought of as a higher inductive type that adds path constructors for all paths above a particular dimension. We use $\|A\|_n$ to denote the nth truncation of A, and for some $a : A$, we use $|a|_n$ to denote the inclusion of a into $\|A\|_n$.

Why would we want to reduce the homotopy level of a type? One big reason would be to avoid having to solve coherence between higher paths. For example, consider the type representing "the image of $f : A \to B$":

$$\mathsf{Image} : \sum_{y:B} \left(\left\| \sum_{x:A} (f(x) = y) \right\|_1 \right)$$

If we wanted to compare elements of this type, identity is functorial over products, so we would need to compare y_1 and y_2, which we may have techniques for. However, comparing the equality term $f(x) = b$ may be extremely difficult, or even impossible if we allow an infinite hierarchy of path types. In this case (and many such cases in mathematics), we only care *that* an element y is in the image, not which particular x it was mapped from. In fact, a devastating effect of allowing multiple distinct $\sum_{x:A} f(x) = b$ elements is that there could possibly be more elements in the image of f than in B itself.

Truncating paths to a certain homotopy level – for example, mere propositions – allows us to safely ignore higher path data while not completely losing access to it. For example, we can map out of a mere proposition to produce another mere proposition. In a sense,

as long as we are not "leaking" more information, we can perform computations on the underlying data.

2.2 Algebra

2.2.1 Algebra in Agda

Working with algebra in a computer proof assistant tends to be more cumbersome than doing algebra on pen and paper. Equations need to be solved by rewriting the current proof goal using identities either proved directly for types such as natural numbers, or assumed as part of axioms for algebraic structures such as groups or rings. For example, when solving $x - 2 = 5$, it might be natural for us to add 2 to both sides to get 7 in our heads, but we are actually using `ap` to add 2 to both sides, then using the left inverse property of integers to cancel the -2 and 2 to get 0, and then using the right identity property of integers to indicate that $x + 0$ is the same as x. In few cases, proof assistants may come equipped with a *ring solver* [GM05], which can automatically solve some equations in rings using a more syntax-directed approach. In all other cases, equations will need to be solved manually, which in tactic-based proof assistants is done by writing a series of tactics to apply in order to complete the proof. However, this often leads to cryptic-looking proofs, and is intractable without having access to the theorem prover. The Agda proof assistant takes a different approach – while employing tactics, they are primarily part of the interactive tooling and not directly written into the code. Instead, proof terms are explicit in the final code, and the library provides syntactic sugar to create a powerful "equational reasoning" syntax that combines the interactivity with a more readable notation akin to two-column proofs. For example, consider this Agda proof of a lemma.

```
lemma : invEq d 0g ≡ - d .fst 0g
lemma =
  invEq d 0g                                ≡⟨ sym (+IdL _) ⟩
  0g + invEq d 0g                           ≡⟨ cong (_+ invEq d 0g) (sym (+InvL (d .fst 0g))) ⟩
  (- d .fst 0g + d .fst 0g) + invEq d 0g    ≡⟨ sym (+Assoc _ _ _) ⟩
  - d .fst 0g + (d .fst 0g + invEq d 0g)    ≡⟨ cong ((- d .fst 0g) +_) (+Comm _ _) ⟩
  - d .fst 0g + (invEq d 0g + d .fst 0g)    ≡⟨ cong ((- d .fst 0g) +_) (sym (pd (invEq d 0g))) ⟩
  - d .fst 0g + d .fst (invEq d 0g)         ≡⟨ cong ((- d .fst 0g) +_) (secEq d 0g) ⟩
  - d .fst 0g + 0g                          ≡⟨ +IdR (- d .fst 0g) ⟩
  - d .fst 0g                               ∎
```

The left side is a restatement of the *type* of the expression, while the right side is the proof term that shows why the term to its left is equal to the term on the next line. Under the hood, the `_≡⟨_⟩` is stitching together all the proof terms on the right side using path concatenation into a single path. Later in this paper we will use a two-column proof format for algebraic proofs. It should be understood that this constitutes a sequence of path manipulations, using the path operators as well as group axioms we will define.

Next, defining algebraic structures in type theory is done through sigma (dependent pair) types, which contain not only the underlying types of the structures but also all the operations and their properties. The first element of the pair is the underlying type, also known as the "carrier," and the second element of the pair expresses properties of the underlying type. In order to construct the overall structure, it is required to not only provide the underlying type, but also the properties. This is akin to requiring all the properties to be proven in order to even show that the structure is well-defined.

For example, consider the monoid. It could be defined this way:

```
record Monoid : Type (ℓ-suc ℓ) where
  field
    A : Type ℓ                                        -- underlying type
    _·_ : A → A → A                                   -- op
    e : A                                             -- identity
    ·IdL : (x : A) → e · x ≡ x                        -- left identity
    ·IdR : (x : A) → x · e ≡ x                        -- right identity
    ·Assoc : (x y z : A) → x · (y · z) ≡ (x · y) · z  -- associativity
    is-set : isSet A                                  -- UIP
```

In Agda, a record is equivalent to a nested sequence of dependent pairs. Each field is able to depend on all previous fields. Defining a monoid using this record type requires providing each of the specified proofs, and having an element of type `Monoid` gives us access to all of the properties we would expect a monoid to exhibit.

Although putting all the properties in the same record as the carrier is concise, it is more customary in practice to separate the carrier from its properties, since when working with

sets, most properties will be mere propositions. For example, the `·IdL` property above is a mere proposition because its underlying type A is a set, which means all paths in A are mere propositions. The benefit of defining properties this way is that the entire properties record becomes a mere proposition, and when performing comparisons between pairs where the second element of the pair is a mere proposition, the second element can be safely ignored. In fact, this is a pattern that we will use frequently, and the Cubical Agda library provides a helper known as `Σ≡Prop` whose type is:

$$\texttt{Σ≡Prop} : ((x : A) \to \mathsf{isProp}(B(x))) \to \left(u, v : \sum_A B\right) \to (\mathsf{fst}(u) = \mathsf{fst}(v)) \to u = v$$

This encodes the property that it suffices to show that the first elements of a pair are identical, when the type of second element of the pair is a mere proposition.

2.2.2 Abelian groups

Definition 2.2.2.1 (Abelian group): An abelian group G consists of a set G, equipped with an identity element 0, inverse function $(-)$, and a commutative binary operation, $(+)$ satisfying the following axioms:

- associativity $a + (b + c) = (a + b) + c$
- left- (`IdL`) and right- (`IdR`) identity $0 + a = a$ and $a + 0 = a$
- left- (`InvL`) and right- (`InvR`) inverse $a + (-a) = 0$ and $(-a) + a = 0$
- commutativity (`+Comm`) $a + b = b + a$

In type theory, we use a sigma (dependent pair) type to represent a structure like this. The 0 element of a group G may be denoted 0_G when it benefits to know which group is being discussed.

The binary operation is abstracted but since it behaves so much like addition in most contexts, we will use the plus + operator to represent it rather than the traditional dot $(\cdot)$. Likewise, we may use $+_G$ to explicitly declare the group being used.

Definition 2.2.2.2 (Group homomorphism): A group homomorphism f between abelian groups A and B is a function between the carrier sets A and B, with the property that it preserves the group operation:

$$f(a+b) = f(a) + f(b)$$

It directly follows from this that the group homomorphism also respects identities and inverses. As this is a basic result in group theory, we will omit the proof below.

2.2.3 Subgroups

Definition 2.2.3.1 (Subgroup): A **subgroup** H of a group G is itself a group whose underlying subset $\langle H \rangle$ is a subset of the underlying set $\langle G \rangle$, such that the identity element is in the subset, and group and inverse operations remain closed in the subset.

In type theory, we encode subsets of a set S as elements of the power set $S \to \mathsf{Prop}$, where Prop is the type of all mere propositions. For example, even numbers are expressed as a function of type $\mathbb{N} \to \mathsf{Prop}$, defined by $n \mapsto n \bmod 2 = 0$. Then, inhabitants of the subset will need to produce a witness of the subset property. Suppose T is a subset of S. Then, we say $x \in T$ to mean $T(x)$.

For H to be a subgroup of G, we require three properties to hold:

- $\mathsf{id\text{-}closed} : 0_G \in H$
- $\mathsf{op\text{-}closed} : (x, y : G) \to x \in H \to y \in H \to x + y \in H$
- $\mathsf{inv\text{-}closed} : (x : G) \to x \in H \to -x \in H$

There is also a notion of a *normal* subgroup, which has the additional condition that for any $g : G$ of some group G and $s : S$ of its subgroup S, that $g \cdot s \cdot g^{-1} = s$. This condition is required for quotients to be well-defined. However, as we are primarily working with abelian groups, we get this property "for free", since we can re-order the group operations such that $g \cdot g^{-1}$ cancels out.

2.2.4 Images and kernels

A couple examples of subgroups we will encounter quite often are *images* and *kernels*.

The image of a homomorphism (or more generally, any function) is the set of all elements of the codomain that are mapped by the homomorphism. The only way to provide a witness of this fact is to provide the corresponding member of the domain. For some homomorphism $f : A \to B$, we may want to define $\mathsf{isInIm}(f, a)$, then, as $\sum_{a:A} f(a) = b$. However, this leads us to a problem: if there happened to be different proofs of $f(a) = b$, we would have different elements of the image, leading to potentially *more* elements in the image than the codomain. In fact, we don't even want to know which elements a of the domain map to b, since there may be multiple such a's. Thus the best way to define the image is by propositionally truncating the proof.

Definition 2.2.4.1 (Image): Let f be a homomorphism from A to B, and b be some element of B. The proposition isInIm is defined as:

$$\mathsf{isInIm}(f, b) \equiv \left\| \sum_{a:A} f(a) = b \right\|_1$$

The image subtype of B is:

$$\mathsf{Im}(f) \equiv \sum_{b:B} \mathsf{isInIm}(f, b)$$

Definition 2.2.4.2 (Kernel): Let f be a homomorphism from A to B, and a some element of A. The proposition $\mathsf{isInKer}$ is defined as:

$$\mathsf{isInKer}(f, a) \equiv (f(a) = 0_B)$$

The kernel subtype of A is:

$$\mathsf{Ker}(f) \equiv \sum_{a:A} \mathsf{isInKer}(f, a)$$

An important property of both images and kernels is that both isInIm and $\mathsf{isInKer}$ are mere propositions, meaning that they are contractible. This contributes to the fact that images and kernels are subtypes of the underlying type (B for isInIm and A for $\mathsf{isInKer}$). The contractibility of the second item in the pair ensures that the subtype cannot contain more elements than the full type. In fact, not only are images and kernels subtypes, they are *subgroups*, as they exhibit the closure properties for the identity, inverse, and group operations.

2.2.5 Quotients

By grouping elements that are related to each other by some criteria of our choosing, we can partition a set into disjoint subsets of equivalence classes. The resulting set is a quotient, and the criteria is known as a relation. For a set A, and a relation $R : A \to A \to \mathsf{Type}$, A/R denotes the quotient of A by R, and $x \sim_R y$ (or simply $x \sim y$) denotes that elements $x, y : A$ are related by R.

Quotients are central to homotopy theory, because they capture the notion of gluing or collapsing, by identifying a relation which relates the elements being glued. For example, the wedge sum construction involves gluing two pointed spaces together at their basepoint. Its definition involves quotienting the disjoint union of the two spaces by the relation that identifies the two basepoints.

Quotients are defined in HoTT as a higher inductive type, consisting of the function that trivially embeds any element of the total type into the quotient (the "quotient map"), as well as path constructors for those that are related by the relation. In this work, we will focus on **set** quotients, where there is an additional constructor squashing higher paths; however, there also do exist untruncated **type** quotients without this condition.

A slightly simplified version of the Agda code (without universe levels) corresponding to set quotients is:

```
data _/_ (A : Type) (R : A → A → Type) : Type where
  [_] : (a : A) → A / R
  eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
  squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

Eliminating set quotients must also be done with care. Suppose we are defining an eliminator f into some type family $P : A/R \to \mathsf{Type}$. Since elements of set quotients are equivalence classes, not only must we determine the behavior of the eliminator f on a direct inclusion $[a]$, it must be shown that for all $a, b : A$ such that there exists a proof $r : a \sim_R b$, there exists a dependent path $\mathsf{PathP}(\lambda i.\mathsf{eq}/(a, b, r, i), f(a), f(b))$ showing that $f(a)$ and $f(b)$ can be identified. This is required, since the eliminator must respect the relation.

A pattern that frequently comes up when working with quotients is picking a representative element out of the equivalence class, proving the lemma for that representative element, and then transporting it to any member of that equivalence class. This representative element is in the form the inclusion of an element of the underlying type.

Theorem 2.2.5.1 (Surjectivity of the quotient map): Let A be any type, and let $R : A \to A \to \mathsf{Type}$ be a relation over A. Then, for any element q of A/R, there is a corresponding element a of A for which $[a] = q$.

This avoids the hassle of needing to use the full eliminator every time, at the cost of introducing some transports into proof terms, which may require an additional transport

lemma. It also helps us with diagram chasing proofs involving quotient types, as it is easier to work with a single representative element. We can pick such a representative element because the canonical map of a set into its quotient is surjective. This theorem appears as 6.10.2 in the HoTT book [Uni13].

There is another important property we will use later – effectiveness of prop-valued quotients. In the case that $R(x, y)$ happens to be a mere proposition, we can actually extract the *way* x and y are related. An inhabitant of the relation $R(x, y)$ tells us that the equivalence class of x is the same as the equivalence class of y. Formally, this is $R(x, y) \to [x] = [y]$. However, just knowing $[x] = [y]$ does not tell us anything about the original relation. This is because for non-prop-valued relations, $R(x, y)$ would not be unique, so we cannot automatically determine a map $[x] = [y] \to R(x, y)$. But if $R(x, y)$ is a mere proposition, this is simply a mapping from a mere proposition into another mere proposition.

Theorem 2.2.5.2 (Effectivity of prop-valued quotients): For any relation $R : A \to A \to \mathsf{Type}$ such that for all $x, y : A$, $R(x, y)$ is a mere proposition, we have a canonical map:

$$[x] = [y] \to R(x, y)$$

2.2.6 Exactness and exact sequences

Exactness is an important concept related to sequences of maps in homological algebra. To be precise, it measures if the image of the previous map is the same as the kernel of the next. Exactness is typically interesting for discussing chain complexes in homology, where discrepancies between the images and kernels tell us about higher-dimensional holes in the space.

In order to define exactness, we must first discuss pointed types. A **pointed type** has a single distinguished point, known as the "base" point. Any element of the type can be

chosen to be the base point. Typically, the specific choice doesn't matter, so there are techniques to make the choice of base point irrelevant, for example by assuming the space is *connected*, making all elements connected by a path. Pointed types are typically denoted A_* and their base point a_0.

Maps between pointed types are known as pointed maps (or pointed functions), and they are just maps between the underlying types with the added stipulation that the base point of the domain is mapped to the base point of the codomain. Formally speaking:

$$A_* \to_* B_* \equiv \sum_{f:A\to B} (f(a_0) = b_0)$$

Definition 2.2.6.1 (Exactness): Given pointed types A, B, C and pointed functions $f : A \to B$ and $g : B \to C$, we say f and g are **exact** if the image $\mathsf{Im}(f)$ is equivalent to the kernel $\mathsf{Ker}(g)$.

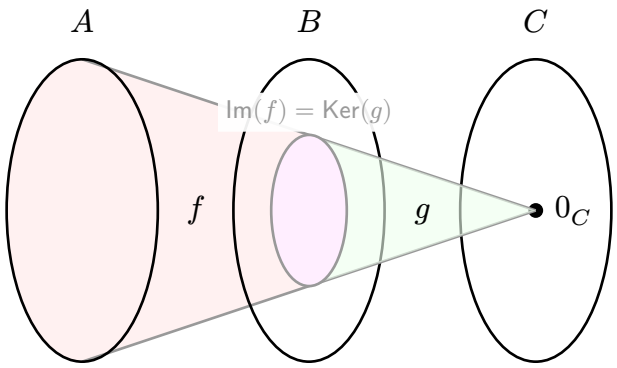


Diagram 1: $\mathsf{Im}(f) = \mathsf{Ker}(g)$

In the formalization, we represent this equivalence as a pair of functions, as shown below, since the conversion functions are the only parts that are important to our formalization.

```
im∈ker : isInIm f b → isInKer g b
ker∈im : isInKer g b → isInIm f b
```

Remark 2.2.6.2: It can be shown that having these two functions are sufficient for proving the entire equivalence, due to the fact that C is a set and images are propositionally truncated.

This definition of exactness also extends to groups, since those are simply pointed types with more structure.

A series of functions where every pair of consecutive functions are exact is called an *exact sequence*. Exact sequences give us a way to concisely represent certain properties of functions. As an example, consider this exact sequence:

$$0 \longrightarrow A \xrightarrow{f} B \longrightarrow 0$$

The exactness at A tells us that the kernel of f is *only* 0_A. This corresponds to the fact that f is **injective** (meaning no element of the codomain is mapped to from more than one element of the domain). The exactness at B tells us that the image of f is all of B. This corresponds to the fact that f is **surjective** (meaning every element of the domain is being mapped to some element of the codomain). These two facts combined tell us that f is an **isomorphism**.

An exact sequence is considered *long* if it spans infinitely in one or both directions. For example, the long exact sequence of homotopy groups relates the nth homotopy groups derived from a pointed function f:

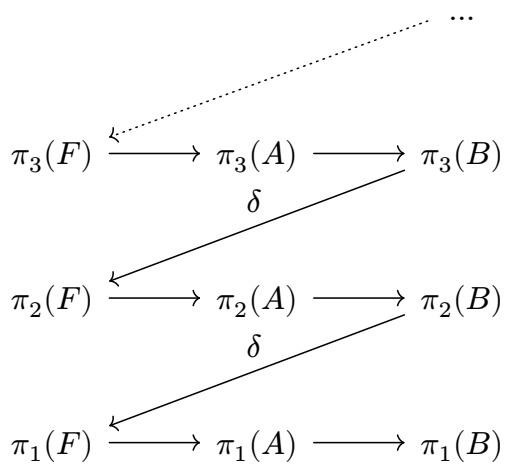


Diagram 3: The long exact sequence of homotopy groups

Long exact sequences have many applications in homotopy theory. As an example, the long exact sequence of homotopy groups for the Hopf fibration $S^3 \hookrightarrow S^7 \to S^4$ has been used to compute the homotopy group $\pi_3(S^4)$.

3 Exact Couples

Exact couples are an algebraic structure discovered by [W. 52] that cleverly separates the algebraic side of spectral sequences from the homological side. To motivate discussion of exact couples, let us begin with a brief overview of spectral sequences.

3.1 Spectral sequences

At its core, a spectral sequence is a bookkeeping tool for computing with long exact sequences of (co)homology groups. It was originally introduced by Jean Leray [Ler46a, Ler46b] for computing sheaf cohomology, with popular textbook accounts in [Hat04, McC01]. Spectral sequences are made out of pages of 2 dimensional grids of (usually) abelian groups, with differentials between the groups. There are many different kinds of spectral sequences, characterized by their *second* page, and an ∞-page, which is the information we want to compute. The ∞-page describes the "convergent" page of the spectral sequence, when pages no longer change due to all differentials becoming trivial. The statement of the spectral sequence usually looks like:

$$E_2^{p,q} = G \Rightarrow H$$

where G is the definition of the E_2 page at coordinates (p, q), and H is the definition of the E_∞ page at (p, q). (This notation is for a cohomological spectral sequence. For a *homological* spectral sequence, we write the scripts in the opposite order: $E^2_{p,q}$) Each subsequent page's E groups are computed as the homology of its previous page.

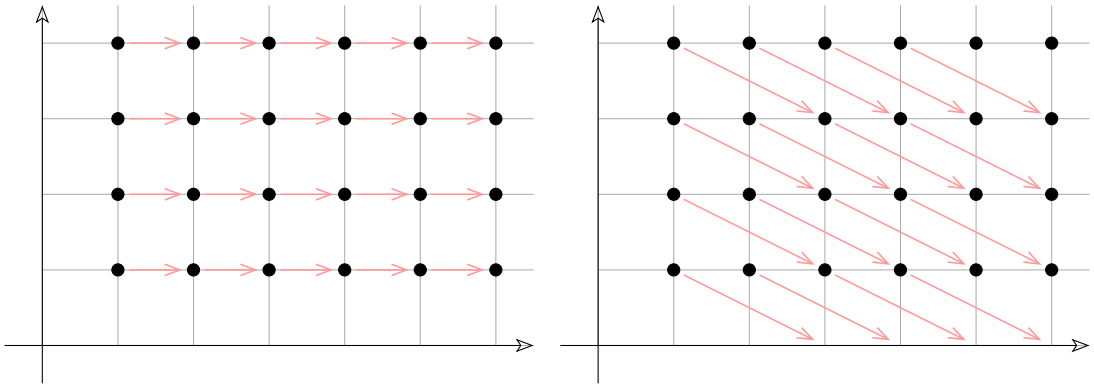


Diagram 4: Pages $E_{1,2}$ of a cohomological spectral sequence

Exact couples thus exploit the fact that the morphisms in the long exact sequences used to construct the spectral sequences follow a pattern, making it able to be represented uniformly as graded homomorphisms. To get from an exact couple to a spectral sequence, we can simply "forget" extra data from our exact couple. This extra data is needed because each page of the spectral sequence only gives us the next page's groups E_r, but not its differentials d_r.

A notable aspect of this approach is that it separates the algebraic machinery of iterating the spectral sequence from the homotopy theory, giving us a purely algebraic way of referring to this. While all exact couples give rise to a spectral sequence, not all spectral sequences are necessarily derived from an exact couple. In this section, we will discuss how to construct the derived exact couple of an exact couple, and relate it with constructing spectral sequences.

3.2 Basic definition of exact couples

An exact couple[2] (A, C, f, g, h) consists of two abelian groups A, C as well as morphisms

$$f : A \to A \qquad g : A \to C \qquad h : C \to A$$

[2]Other accounts use D, E as the groups and i, j, k as the morphisms, but the names here were chosen to disambiguate from other naming conventions. For example, for an element of D one would typically write d, but this conflicts with the name of the differential. And i, j, k are frequently used for degree indexing throughout.

such that they are exact in every vertex of the following non-commuting diagram:

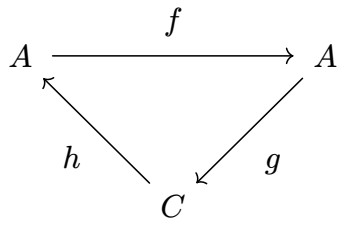


Diagram 5: Exact couple

The purpose of arranging data in this fashion is that we can create a *derived* exact couple. The derived exact couple represents the next page of the spectral sequence, which depending on the actual spectral sequence, is typically a closer approximation to the (co)homology group being computed. However, the derivation process is a purely algebraic process, and relies only on the group structure that we up above.

To form the derived exact couple (A', C', f', g', h'), we define:

- A' is the image of f
- C' is the homology $\mathsf{Ker}(d)/\ \mathsf{Im}(d)$
- f' is the induced map of f under A'
- g' is roughly defined as $g \circ f^{-1}$, although we will make this more precise in the next section
- h' is the induced map of h

These definitions were chosen because they reflect the iteration process of the spectral sequence. The maps f, g, and h typically come from the long exact sequences of (co)homology groups of some filtration. Since C ends up being exactly one of the E pages, the derived C' must be its homology. Finally, the proof of exactness of for the derived couple is typically done through diagram chasing. We will discuss the diagram chasing proof below, but with special care to the type theoretic details.

3.3 Mechanization of exact couples over abelian groups

Although the definition of f', g', and h' have very concise descriptions in prose, type theory as a formal system requires a higher degree of rigor, for instance in how truncations and quotients are handled. In this section we will walk through the definition of all components of the exact couple, with special care for handling type theory structures.

3.3.1 Defining the groups

A' is an abelian group that is defined to be the image of f. The underlying set of A' is therefore defined as $\mathsf{Im}(f)$. As a reminder, this is defined as $\sum_{a:A} \left\| \sum_{a':A} f(a') = a \right\|_1$, the set of all a in A such that there exists some a' in A such that $f(a') = a$. The image forms a subgroup with the same group operation as A. Thus, the set $\mathsf{Im}(f)$ is indeed a group. The final piece to show that A' is an abelian group is to prove commutativity of the group operation, i.e for any $x, y : \mathsf{Im}(f)$, it is true that $x + y = y + x$. If x is a pair (x', p) where p is a proof that x' is in the image of f, and similarly y is a pair (y', q), then the group operation $x +_{A'} y$ is defined to be a pair $(x' +_A y', r)$, where r is a proof that $x' +_A y'$ is in the image of f, derived from p and q. Similarly, $y +_{A'} x$ results in $(y' +_A x', s)$, where s is a proof that $y' +_A x'$ lies in the image of f. It is required to produce a proof $v_1 : x' +_A y' = y' +_A x'$, and then under this equality, that there exists a dependent path from r to s, written $v_2 : \mathsf{PathP}(\lambda i.\ \mathsf{isInIm}(f, v_1(i)), \mathsf{op\text{-}closed}(p, q), \mathsf{op\text{-}closed}(q, p))$, where $\mathsf{op\text{-}closed}(p, q)$ combines p and q using the subgroup property that the group operation remains closed in the subgroup. We can produce a definition for v_1 by using the commutative property of the group operation of the abelian group A. For v_2, we can produce a path between them since op-closed is a mere proposition.[3]

To define C', we will need to first define a map $d : C \to C$, defined as $d \equiv g \circ h$. Note that this function takes g *after* h, so it is not exact from the assumptions of the exact couple.

[3]Though this is the overall idea, formally there is some more work required to reconcile the fact that the two endpoints are in different op-closeds. The mechanization uses the `Σ≡Prop` library function which handles comparing dependent pairs where the second element is a mere proposition.

However, composing d with itself *does* result in the zero map, since $d \circ d = (g \circ h) \circ (g \circ h) = g \circ (h \circ g) \circ h = g \circ 0 \circ h = 0$. Because d exhibits the property $d^2 = 0$, it is known as a *differential*. In fact, this d represents the differentials of the spectral sequence that is constructed from the exact couple. C' is then roughly defined as the homology of this differential:

$$C' \equiv \mathsf{Ker}(d)/\ \mathsf{Im}(d)$$

Of course, there are some things we must verify in order to show that this definition is well-founded, and indeed an abelian group. Type theory forces us to perform these verifications since structures such as abelian groups and quotients are made up of records consisting of not only the sets themselves but also the properties, so in order to construct the structure we must produce proof terms for all properties. First, we must show that $\mathsf{Ker}(d)$ is even an abelian group. This result simply relies on the fact that the group operation of $\mathsf{Ker}(d)$ is defined element-wise, and $\mathsf{isInKer}(d, -)$ is a mere proposition. Therefore, for any $x, y : \mathsf{Ker}(d)$, to prove that $x + y = y + x$ it suffices to show that the first element of $x + y$ and $y + x$ exhibits the commutative property, and the second element is identified automatically. Of course, since $\mathsf{fst}(x), \mathsf{fst}(y) : C$ and C is an abelian group, it must be true that $\mathsf{fst}(x + y) = \mathsf{fst}(y + x)$.

Next, in order for our quotient to be well-defined, it must be shown that the image of d' is necessarily a *normal subgroup* of the kernel. Is this statement even well-defined? To be precise, d has type $C \to C$. The image subgroup of any function $A \to B$ will necessarily be a subgroup of B, so we must operate with a separate version of d, called d', which has type $C \to \mathsf{Ker}(d)$. This function can be defined from d by first showing that any $d(c)$ must lie in the kernel of d, since $d^2(c) = 0$ by the differential property. It is also trivial to show that d' is a group homomorphism, by ignoring the mere proposition second element. Finally, since both the image and the kernel remain abelian groups, all subgroups are normal.

Remark 3.3.1.1: Going forward, we will use both $\mathsf{Ker}(d)$ and $\mathsf{Im}(d)$ in the definition of C'. It should be understood that the divisor is actually $\mathsf{Im}(d')$, and that inhabitants of this group have a first projection that lies in $\mathsf{Ker}(d)$ rather than C.

Finally, we must prove that the type $\mathsf{Ker}(d)/\ \mathsf{Im}(d)$ indeed forms an abelian group. The quotient of an abelian group by a subgroup always forms a group. The quotient group's operation is induced from the group operation of the kernel of d, which itself is induced from the group operation of C.

$\mathsf{Ker}(d)$'s group operation takes (x, p) and (y, q) where $x, y : C$ and $p : d(x) = 0_C$ and $q : d(y) = 0_C$, and must produce a new element of $\mathsf{Ker}(d)$ that respects the group operation. The first element of the pair is simply $x +_C y$. The second element of the pair needs to show $d(x +_C y) = 0_C$. We can use d's group homomorphism property to show $d(x +_C y) = d(x) +_C d(y)$, after which we can simply substitute p and q to complete the proof.

The group operation for $\mathsf{Ker}(d)/\ \mathsf{Im}(d)$ is defined by eliminating the two operands. Suppose we start with $x, y : \mathsf{Ker}(d)/\ \mathsf{Im}(d)$. To eliminate x and y, we start by providing the behavior at $x \equiv [x']$ and $y \equiv [y']$ such that $x', y' : \mathsf{Ker}(d)$. In this case, we can simply use the group operation of $\mathsf{Ker}(d)$ that we defined above, resulting in $[x'] +_{\mathsf{Ker}(d)/\ \mathsf{Im}(d)} [y'] \equiv \left[x' +_{\mathsf{Ker}(d)} y'\right]$. We must then show that for some other $x_1 : \mathsf{Ker}(d)/\ \mathsf{Im}(d)$ such that $x \sim_{\mathsf{Im}(d)} x_1$, our group operation produces a propositionally equal result: $\left[x' +_{\mathsf{Ker}(d)} y'\right] = x_1 +_{\mathsf{Ker}(d)/\ \mathsf{Im}(d)} [y']$. In other words, the group operation respects the relation. Similarly we must do this for some $y_1 : \mathsf{Ker}(d)/\ \mathsf{Im}(d)$ such that $y \sim_{\mathsf{Im}(d)} y_1$. The justification for this is that for $[x_1] \sim_{\mathsf{Im}(d)} [x_2]$ to hold, it must be the case that $\mathsf{isInIm}(d, x_2 - x_1)$. Thus, it suffices to show that the outcome of the group operation with some x_1 only differs by some element of $\mathsf{Im}(d)$. But since our group operation is exactly addition, this fact will always hold.

Finally, we must prove that our new group operation is commutative, to satisfy the requirement that C' is an abelian group. We know that C's group operation is commutative, so the remaining work is navigating the wrappers around C.

3.3.2 Defining the morphisms

$f' : A' \to A'$ is defined as the abelian group homomorphism induced by f. Recall from Definition 2.2.2.2 that defining a group homomorphism requires two pieces of data – the underlying function from $\langle A' \rangle$ to $\langle A' \rangle$, and then a proof that the function satisfies the group homomorphism property.

The underlying function f' takes some element of $\langle A' \rangle$ as input. Since A' is defined as the image of f, it is a pair (a, p) consisting of $a : A$ and $p : \mathsf{isInIm}(f, a)$. We already know how to transform a; since f' is induced by f, the output is simply $f(a)$. Additionally, $f(a)$ is trivially in the image of f, with a as its witness, so we can use the term $|(a, \mathsf{refl})|_1 : \mathsf{isInIm}(f, f(a))$ as the second element of the output pair.

To show f' satisfies the group homomorphism property, we must show that for some $x, y : A'$, that $f'(x +_{A'} y) = f'(x) +_{A'} f'(y)$. If we consider only the first element of each pair, $\mathsf{fst}(f'(x +_{A'} y)) \equiv f(\mathsf{fst}(x) +_A \mathsf{fst}(y))$, $\mathsf{fst}(f'(x)) \equiv f(\mathsf{fst}(x'))$, and $\mathsf{fst}(f'(y)) \equiv f(\mathsf{fst}(y))$. But we already know $f(\mathsf{fst}(x) +_A \mathsf{fst}(y)) = f(\mathsf{fst}(x)) +_A f(\mathsf{fst}(y))$ from the fact that f itself is a group homomorphism. The second element of the pair is a proof that the first element lies in the image of f. However, since that proof is truncated to a mere proposition, we can simply produce a path by using the truncation.

As mentioned above, $g' : A' \to C'$ consists of two parts – for some $a' : A'$, we first take the pre-image of a' under f, and then apply g to it. It helps to break the construction of g' into two parts – starting with a map pre-g', which is a map $A \to C'$ which starts with the pre-image, and then turning that into our final g' by turning the image into the pre-image properly.

Our definition of pre-g' is induced from composing g (which is $A \to C$) with an inclusion map from $C \to C'$. Recall that $C' \equiv \mathsf{Ker}(d)/\ \mathsf{Im}(d)$. So the first step is to take our $g(a)$ and show that it lies in the kernel of d. As it turns out, $g(a)$ is in the kernel of d for any a, and this is a useful lemma, so we will prove it generally.

Lemma 3.3.2.1: For any a, there is a term $\mathsf{isInKer}(d, g(a))$.

Proof: The expression $\mathsf{isInKer}(d, g(a))$ is equivalent to showing that $d(g(a)) = 0_C$. However, $d \equiv g \circ h$, so $d(g(a)) = g(h(g(a)))$. By exactness, $h(g(a)) = 0$, and by the property of group homomorphisms, $g(0) = 0$. Thus, we can construct a term of type $\mathsf{Ker}(d)$ with $g(a)$ as the first element. ■

To get from $\mathsf{Ker}(d)$ to $\mathsf{Ker}(d)/\ \mathsf{Im}(d)$ we can use the canonical quotient inclusion map. In summary, if we give the name p to the second element of the pair given by Lemma 3.3.2.1, then the definition of pre-g' is:

$$\text{pre-}g'(a) \equiv [g(a), p]$$

We can also show that pre-g' is a group homomorphism by showing it satisfies the group homomorphism property: for any a_1, a_2, we must show $\text{pre-}g'(a_1 +_A a_2) = \text{pre-}g'(a_1) +_{C'} \text{pre-}g'(a_2) = [g(a_1), p_1] +_{C'} [g(a_2), p_2]$. Recall that the group operation of C' is defined such that $[x] +_{C'} [y] \equiv \left[x +_{\mathsf{Ker}(d)} y\right]$. We can rewrite the expression as $\left[g(a_1 +_A a_2), p_{a_1+a_2}\right] = \left[(g(a_1), p_1) +_{\mathsf{Ker}(d)} (g(a_2), p_2)\right]$. Because both sides use the point constructor of the quotient, we can use ap to construct a path for it so long as we can prove $\left(g(a_1 +_A a_2), p_{a_1+a_2}\right) =_{\mathsf{Ker}(d)} (g(a_1), p_1) +_{\mathsf{Ker}(d)} (g(a_2), p_2)$. Since the group operation of $\mathsf{Ker}(d)$ operates pointwise on each of the pair elements, we can separately show that $g(a_1 +_A a_2) = g(a_1) +_C g(a_2)$, and that the proofs that they lie in the kernel of d are equal. The first equality is due to the group homomorphism property of g, and the second is due to the fact that proof of kernel membership is a mere proposition.

Now we have a map $A \to C'$ and we are trying to construct $A' \to C'$. How can we bridge this gap? The A in the pre-g' map defined above refers to the pre-image of some element of A'. So, for some $a' : A'$, we must somehow extract the proof term from the second element of a'. However, this element is of type $\mathsf{isInIm}(f, \mathsf{fst}(a')) \equiv \left\| \sum_{a:A} f(a) = \mathsf{fst}(a') \right\|_1$, which is propositionally truncated. Typically, n-truncated types can only be eliminated to n-types

or lower, otherwise information about higher homotopies would be leaked. However, it has been shown by Kraus [Kra15] that a weakly constant eliminator function can eliminate from a lower n-type to a higher n-type.

Theorem 3.3.2.2: Given a function $f : A \to B$ such that:

- B is a set (homotopy level 2)
- f is *weakly constant* (in other words, for all $x, y : A$, $f(x) = f(y)$)

There exists a function $f' : \|A\|_1 \to B$ such that $f' \equiv |-|_1 \circ f$.

As long as we can show that our definition of g' behaves uniformly on different pre-images of a', we can safely work with the the pre-image. To be precise, we would like to construct a map $A' \to C'$. Let $a' : A'$ be the input, and we must produce an element of C'. We have a function pre-g', which is of type $A \to C'$, but we do not have direct access to the pre-image, which is locked behind the truncation. We will construct a function

$$\mathsf{glmage} : \left(\sum_{a:A} f(a) = \mathsf{fst}(a') \right) \to C'$$

that satisfies the requirements above. This glmage function will internally use pre-g' to create an element of C'. Theorem 3.3.2.2 gives us a map $\mathsf{glmage}' : \left\| \sum_{a:A} f(a) = \mathsf{fst}(a') \right\|_1 \to C'$, which is definitionally equal to $\mathsf{isInIm}(f, \mathsf{fst}(a')) \to C'$. The domain of this map is exactly the data given by $\mathsf{snd}(a')$, so $\mathsf{glmage}'(\mathsf{snd}(a'))$ is the correct element of C' needed to complete the definition of g'.

It remains to give a definition of glmage that satisfies the requirements of Theorem 3.3.2.2. We have previously defined pre-$g' : A \to C'$, so we can define glmage as follows:

$$\mathsf{glmage}((a, p)) \equiv \text{ pre-}g'(a)$$

The first requirement of glmage is that the codomain C' must be a set. Being a set is a precondition for being an abelian group, so we already know this is true. The second requirement is that glmage must be weakly constant. Consider two distinct elements of $\sum_{a:A} f(a) = \mathsf{fst}(a')$, named (a_1, p_1) and (a_2, p_2). To show that glmage is weakly constant, we must produce a proof that $\text{pre-}g'(a_1) = \text{pre-}g'(a_2)$.

To do this, we must use p_1 and p_2: concatenating the paths results in $p_1 \cdot p_2 : f(a_1) = f(a_2)$, which means it is true that $f(a_1) - f(a_2) = 0_A$. By the group homomorphism property of f, we can rewrite this as $f(a_1 - a_2) = 0_A$. Note that this means $a_1 - a_2$ lies in the kernel of f, which is the same as the image of h, by exactness of f and h. Thus, $a_1 - a_2$ lies in the image of h – in other words, we have an element of type $\mathsf{isInIm}(h, a_1 - a_2)$. We do not have direct access to the pre-image, but we can eliminate the proof that $a_1 - a_2$ lies in the image of h using Theorem 3.3.2.2. (We will revisit the conditions for this invocation of the theorem momentarily.) Let c be the pre-image of $a_1 - a_2$ under h, and we also have a proof that $h(c) = a_1 - a_2$. Applying g on both sides yields $g(h(c)) = g(a_1 - a_2)$. We can rewrite this as $d(c) = g(a_1) - g(a_2)$ by applying the group homomorphism property of g. This indicates that $g(a_1) - g(a_2)$ lies in the image of d. Recall that $\text{pre-}g'$ wraps g, quotienting its result by the image of d. This means $\text{pre-}g'(a_1)$ and $\text{pre-}g'(a_2)$ lie in the same equivalence class in C', which means $\text{pre-}g'(a_1) = \text{pre-}g'(a_2)$, completing the requirement for weak constancy.

To complete this proof, we must provide justification that we can eliminate the term of type $\mathsf{isInIm}(h, a_1 - a_2)$ to a term of type $\text{pre-}g'(a_1) = \text{pre-}g'(a_2)$. First, note that $\text{pre-}g'(a_1) = \text{pre-}g'(a_2)$ is a mere proposition, since it is an identity type in C', which is a set. Mere propositions are also sets, as any n-type is also an $(n+1)$-type. Second, we must show that the final part of the above proof is weakly constant. Just as above, consider two distinct elements of $\sum_{c:C} h(c) = a_1 - a_2$, named (c_1, q_1) and (c_2, q_2). To show that our proof is weakly constant, it must be shown that our proofs are propositionally equal. But since our codomain $\text{pre-}g'(a_1) = \text{pre-}g'(a_2)$ is a mere proposition, all elements are by

definition propositionally equal. With this, we have completed the definition for the map g'.

$h' : C' \to A'$ is induced by h, but again, the definition of C' complicates this process quite a bit. The overall construction strategy involves identifying some c that allows us to invoke h, showing that it respects the relation of being in the image of d, and showing that $h(c)$ is in the image of f, which is a requirement for A'.

Let $c' : C'$ be the input. We must now produce an element of A'. Since C' is defined as $\mathsf{Ker}(d)/\ \mathsf{Im}(d)$, c' is actually an equivalence class. To eliminate an equivalence class, it is necessary to give a map from $\mathsf{Ker}(d) \to A'$, and then show that the map respects the relation – any two related members of $\mathsf{Ker}(d)$ will be mapped to propositionally equal elements of A'. Let us call this map pre-$h' : \mathsf{Ker}(d) \to A'$.

To define pre-h', let us consider an input $(c, p) : \mathsf{Ker}(d)$, where p is a proof that c is in the kernel of d. Applying h yields an element of A, but this is not sufficient – to produce an element of A', we must prove that $h(c)$ lies in the image of f. Fortunately, we know c is in the kernel of d, which means $d(c) = g(h(c)) = 0_C$. This means $h(c)$ is in the kernel of g, which by exactness of f and g means that $h(c)$ lies in the image of f.

Now we must show that pre-h' respects the relation. This amounts to showing that for any $x, y : \mathsf{Ker}(d)$ such that $\mathsf{isInIm}(d, x - y)$, it must be true that pre-$h'(x) =$ pre-$h'(y)$. $x - y$ is also in the image of g, due to the definition of d as $g \circ h$. This means $x - y$ is in the kernel of h, by exactness of g and h. In other words, $h(x - y) = 0_A$. Because pre-h' is simply h but with the added $\mathsf{isInKer}(d, -)$ term, this implies that pre-$h'(x - y) = 0_{A'}$. We also need an additional lemma to show that pre-h' in fact observes the group homomorphism property, which can be proved using h's group homomorphism property and the fact that kernel is a mere proposition. The rest of the proof follows:

$$
\begin{aligned}
\text{pre-}h'(x) &= (\text{pre-}h'(x) -_{A'} \text{pre-}h'(y)) + \text{pre-}h'(y) \\
&= \text{pre-}h'\left(x -_{\mathsf{Ker}(d)} y\right) + \text{pre-}h'(y) \qquad \text{by pre-}h' \text{ lemma} \\
&= 0_{A'} + \text{pre-}h'(y) \\
&= \text{pre-}h'(y) \qquad \blacksquare
\end{aligned}
$$

We have now eliminated c' using pre-h' and shown that it respects the relation of C'. It remains to show that this definition for h' is indeed a group homomorphism. This means for any two $c_1, c_2 : C'$, it is true that $h'(c_1 +_{C'} c_2) = h'(c_1) +_{A'} h'(c_2)$. We can eliminate c_1 and c_2 using the quotient eliminator, showing that the final result respects the relation by using the fact that A' is a set. Then, we need only to use the fact that $\mathsf{isInKer}(d, -)$ is a mere proposition and h's group homomorphism property to show the underlying principle. This concludes the construction of h'.

3.3.3 Proof of exactness

As explained earlier in Remark 2.2.6.2, it suffices to show that f' and g' are exact solely by giving a pair of functions $\mathsf{isInIm}(f', a') \to \mathsf{isInKer}(g', a')$ and $\mathsf{isInKer}(g', a') \to \mathsf{isInIm}(f', a')$ for any $a' : A'$.

We begin by defining the first function. Let $a' : A'$, and $p : \mathsf{isInIm}(f', a')$. The goal is to provide a proof for $\mathsf{isInKer}(g', a')$, which is definitionally equal to $g'(a') = 0_{C'}$. In order to access data behind propositional truncations, we must show that we are eliminating *into* a mere proposition as well, or else we will leak data. In this case, C' is a set, by virtue of being a group. Thus, paths in C' such as $g'(a') = 0_{C'}$ must be mere propositions. Now that we have justified accessing propositionally truncated data, we can unpack p into a pair (a'_1, q), with $a'_1 : A'$ and $q : f'(a'_1) = a'$. Let $a'_1 \equiv (a_1, r)$, where $r : \mathsf{isInIm}(f, a_1)$. Since we are still eliminating into a mere proposition, we can unpack r into (a_2, s), with $a_2 : A$ and $s : f(a_2) = a_1$. With all the pieces in hand, the proof can be constructed as follows:

$$
\begin{aligned}
g'(a') &= g'(f'(a'_1)) && \text{by } q \\
&= g'(f'((a_1, \mid a_2, s \mid_1))) && \text{by definition of } a'_1 \text{ and } r \\
&= g'((f(a_1), \mid a_1, \mathsf{refl} \mid_1)) && \text{by definition of } f', \text{ ignoring } s \\
&= [g(a_1), t] && \text{by Lemma 3.3.2.1} \\
&= [g(f(a_2)), t] && \text{by } s \\
&= [0_C, t] && \text{by exactness of } f \text{ and } g \\
&= 0_{C'} && \text{by } \mathsf{isInIm}(d, 0_C) \text{ being a mere proposition}
\end{aligned}
$$

Next, we prove $\mathsf{isInKer}(g', a') \to \mathsf{isInIm}(f', a')$. Let $a' : A'$, and let $p : \mathsf{isInKer}(g', a')$. The goal is to show that $\mathsf{isInIm}(f', a')$. First, let us extract all the data that is available to us. As before, since our goal $\mathsf{isInIm}(f', a')$ is a mere proposition, we can safely extract any data that has been propositionally truncated. Starting with a', let the first element be $a : A$. We can eliminate the second element, which is of type $\mathsf{isInIm}(f, a)$, into a pair with the first element $a_1 : A$ and second element $q : f(a_1) = a$. Next, we have p, which tells us $g'(a') = 0_{C'}$. By the definition of g', we know this means $[g(a_1), y] = 0_{C'}$, where y is the proof obtained from Lemma 3.3.2.1. C' is defined as a quotient, so this is really telling us that $(g(a_1), y)$ is in the same equivalence class as $0_{\mathsf{Ker}(d)}$. In other words, $(g(a_1), y)$ is *related* to $0_{\mathsf{Ker}(d)}$ by the relation of being in the image of d. Since being in the image of d is a mere proposition, we can use Theorem 2.2.5.2 to obtain an inhabitant of $\mathsf{isInIm}\Big(d', (g(a_1), y) -_{\mathsf{Ker}(d)} 0_{\mathsf{Ker}(d)}\Big)$. We can eliminate this term to obtain $c : C$ and a proof $d'(c) = (g(a_1), y) -_{\mathsf{Ker}(d)} 0_{\mathsf{Ker}(d)}$. Then, doing a bit of algebra on this:

$$
\begin{aligned}
d'(c) &= (g(a_1), y) -_{\mathsf{Ker}(d)} 0_{\mathsf{Ker}(d)} \\
d(c) &= g(a_1) -_C 0_C && \text{by taking first projection} \\
d(c) &= g(a_1) \\
0 &= g(a_1) - d(c) \\
0 &= g(a_1) - g(h(c)) && \text{by definition of } d \\
0 &= g(a_1 - h(c)) && \text{group homomorphism property}
\end{aligned}
$$

This tells us that $a_1 - h(c)$ lies in the kernel of g. But by exactness of f and g, this means $a_1 - h(c)$ lies in the image of f as well. We can eliminate the proof of this into a term $a_2 : A$ and a proof $f(a_2) = a_1 - h(c)$. Doing some more algebra:

$$
\begin{aligned}
f(a_2) &= a_1 - h(c) \\
f(f(a_2)) &= f(a_1 - h(c)) && \text{applying } f \text{ to both sides} \\
&= f(a_1) - f(h(c)) && \text{group homomorphism property} \\
&= f(a_1) - 0_A && \text{exactness of } h \text{ and } f \\
&= f(a_1)
\end{aligned}
$$

So, now we have a term, say called $r : f(f(a_2)) = f(a_1)$, that looks very close to $f(a_2) = a_1$, which would help us construct a term of type $\mathsf{isInIm}(f', a')$, but because f is not necessarily injective, we cannot directly make this leap. Instead, we will first construct $a_1' : A'$, defined by a pair $(a, \mid f(a_2), r \cdot q \mid_1)$. Now we know that a_1' lies in the image of f', with a pre-image of $(f(a_2), \mid a_2, \mathsf{refl} \mid_1)$. We can show that $a_1' = a'$, by showing that these terms are actually pairs with an identical first element a, and the type of the second element $\mathsf{isInIm}(f, a)$ is a mere proposition. We can then transport $\mathsf{isInIm}(f', a_1')$ over this path to finally obtain an element of type $\mathsf{isInIm}(f', a')$.

To show exactness of g' and h', we must give functions $\mathsf{isInIm}(g', c') \to \mathsf{isInKer}(h', c')$ and $\mathsf{isInKer}(h', c') \to \mathsf{isInIm}(g', c')$ for any $c' : C'$. For both of these functions, it is convenient to first start by using Theorem 2.2.5.1 to obtain some $(c, p) : \mathsf{Ker}(d)$ such that $p : \mathsf{isInKer}(d, c)$

and $[c,p]=c'$. Notice that this involved eliminating out of a propositional truncation, but just as before, this is valid because both $\mathsf{isInIm}(g',c')$ and $\mathsf{isInKer}(h',c')$ are mere propositions. We can transport the inputs and outputs along this path so we can work with a concrete C directly.

Let us begin with the forward direction. We have a term of type $\mathsf{isInIm}(g',[c,p])$, which we can eliminate (due to our goal being a mere proposition) into a pair consisting of $a':A'$, and $q:g'(a')=[c,p]$, and trying to produce a term of type $\mathsf{isInKer}(h',[c,p])$, or $h'(g'(a))=0_{A'}$. The term a' is a pair which we will call the first projection a and the second projection is a proof of $\mathsf{isInIm}(f,a)$. We can eliminate that proof, giving us $a_1:A$ such that $f(a_1)=a$. By using the dependent eliminator, we can change the goal term slightly – rather than producing a term of type $h'(g'(a))=0_{A'}$, it suffices to show $h'(g'((a,y)))=0_{A'}$, where y is obtained from Lemma 3.3.2.1. Unwrapping the definitions of g' and h', we see that the goal is really pre-$h'((g(a),y))=0_{A'}$. This significantly simplifies the task, since A' is a pair where the second element is a mere proposition, so we only need to show that the first elements of the pairs can be identified. In this case, we are looking for $h(g(a))=0_A$, which is true by exactness of g and h.

For the backwards direction, we again begin with $c:C$ such that $\mathsf{isInKer}(d,c)$ and $[c,p]=c'$. We also have the assumption $p:h'([c,p])=0_{A'}$, and are trying to show that $\mathsf{isInIm}(g',[c,p])$. Taking the first element of each side of p yields $h(c)=0_A$ – in other words, c lies in the kernel of h. Applying exactness of g and h yields $q:\mathsf{isInIm}(g,c)$. To go from this to $\mathsf{isInIm}(g',c')$, we must propositionally eliminate q into its parts: $a:A$ such that $g(a)=c$. Now we can construct the pre-image of c' under g', which is a pair consisting of $f(a)$, and a trivial proof that $f(a)$ lies in the image of f. Then, we can show applying g' yields the equivalence class of $g(a)$, which we know is c. The second element does not matter since it is a mere proposition.

Finally, we prove exactness of h' and f'. This involves some $a' : A'$ for which we must show $\mathsf{isInIm}(h', a') \to \mathsf{isInKer}(f', a')$ and $\mathsf{isInKer}(f', a') \to \mathsf{isInIm}(h', a')$. Let us give the name a to the first element of a'.

For the forward direction, we have $p : \mathsf{isInIm}(h', a')$ and we are trying to show $\mathsf{isInKer}(f', a')$. Note that this expands to a path between elements of A', which is a pair where the second element is $\mathsf{isInIm}(f', -)$, which is a mere proposition. This means we simply have to show $f(a) = 0_A$. Since h and f are exact, this is equivalent to showing that a lies in the image of h. We can use propositional elimination on p to obtain $c' : C'$ such that $h'(c') = a'$. Using surjectivity of the quotient, Theorem 2.2.5.1, we can pick some representative element $c : C$ such that $[c] = c'$, so we have $h'([c]) = a'$. Projecting the first element out from each side yields $h(c) = a$. We can now construct an element of $\mathsf{isInIm}(h, a)$ using c as the pre-image, and convert this over to $\mathsf{isInKer}(f, a)$ to complete the proof.

Finally, in the backward direction, we have $p : \mathsf{isInKer}(f', a')$ and we are trying to show $\mathsf{isInIm}(h', a')$. As with the forward direction, extracting just the first component of p gives us $\mathsf{isInKer}(f, a)$, which is the same as $\mathsf{isInIm}(h, a)$. We must now derive $\mathsf{isInIm}(h', a')$ from $\mathsf{isInIm}(h, a)$. Unpacking $\mathsf{isInIm}(h, a)$ gives us $c : C$ and a proof that $h(c) = a$. In order to show that c can be included into C', we must show that c lies in the kernel of d. Fortunately, we know that a is in the image of f, which means it's in the kernel of g, or $g(a) = 0_C$. Combining this with $h(c) = a$ gives us $g(h(c)) = 0_C$, which is precisely $d(c) = 0_C$. We can use this proof to construct our pre-image $[c]$. The proof of $h'([c]) = a'$ is a pair with the second element a propositional truncation, and the first element has already been proved earlier with $\mathsf{isInIm}(h, a)$.

4 Conclusion

We have presented a formal construction of exact couples with a detailed emphasis on the type theoretic aspects. The formal construction has additionally been mechanized and checked for correctness using the Cubical Agda proof assistant. This provides a foundation upon which we can build spectral sequences in Cubical Agda.

The process of formalization itself is quite a journey. Firstly, learning a proof assistant as well as associated libraries to the point of familiarity takes quite some time. For this reason, I believe it is still incredibly challenging for those not already familiar with coding or proof assistants to produce a working proof that will be accepted by the type checker.

Performance also becomes a bottleneck for proofs of certain complexity. It is quite common when doing homological algebra to have types contain entire proofs. Even though sometimes we can use propositional truncation to ignore it, the compiler will still have to process that truncation, and it will take an immense amount of time. Without more runtime flags, Agda may consume enormous amounts of memory to represent these expressions. In an interactive development environment (IDE), producing contexts and cubical boundaries requires even more resources. One method to mitigate this problem during the development phase is use of a feature known as "lossy unification", or "injective for inference". When evaluating comparisons this avoids expanding $f(x) = f(y)$ when it decides f is the same on both sides. While this changes the type checking behavior, developers believe Agda should only produce false negatives, rather than false positives. This makes it a useful heuristic during development, but as the feature is still experimental, the code cannot be trusted to be correct unless it has been checked without the feature enabled.

Still, an increase in the repertoire of mechanized mathematics, as well as awareness of mechanized proof assistants as well as improvements to those proof assistants results in an elevated level of trust in mathematical results.

References

[Ler46] J. Leray, "L'anneau d'homologie d'une représentation," *C. R. Acad. Sci., Paris*, vol. 222, pp. 1366–1368, 1946a.

[Ler46] J. Leray, "Structure de l'anneau d'homologie d'une représentation," *C. R. Acad. Sci., Paris*, vol. 222, pp. 1419–1422, 1946b.

[W. 52] W. S. Massey, "Exact Couples in Algebraic Topology (Parts I and II)," *The Annals of Mathematics*, vol. 56, no. 2, p. 363, Sep. 1952, doi: 10.2307/1969805.

[W. 53] W. S. Massey, "Exact Couples in Algebraic Topology (Parts III, IV, and V)." Accessed: Jul. 26, 2025. [Online]. Available: https://webhomes.maths.ed.ac.uk/~v1ranick/papers/massey7.pdf

[Bau20] A. Bauer, "What makes dependent type theory more suitable than set theory for proof assistants?." Accessed: Aug. 21, 2025. [Online]. Available: https://mathoverflow.net/q/376973

[Uni13] T. Univalent Foundations Program, *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study, 2013. [Online]. Available: https://homotopytypetheory.org/book

[Coh+15] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," in *21st international conference on types for proofs and programs (TYPES 2015)*, T. Uustalu, Ed., in Leibniz international proceedings in informatics (lipics), vol. 69. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 1–34. doi: 10.4230/LIPIcs.TYPES.2015.5.

[Doo18] F. van Doorn, "On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory." Accessed: Sep. 25, 2024. [Online]. Available: http://arxiv.org/abs/1808.10690

[VMA21] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical Agda: A dependently typed programming language with univalence and higher inductive types," *Journal of Functional Programming*, vol. 31, p. e8, 2021, doi: 10.1017/S0956796821000034.

[HUW14] J. Harrison, J. Urban, and F. Wiedijk, "History of Interactive Theorem Proving," *Computational Logic*, vol. 9, pp. 135–214, 2014, Accessed: Jun. 30, 2025. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf

[Mar75] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," *Logic colloquium '73*, vol. 80. in Studies in logic and the foundations of mathematics,

vol. 80. Elsevier, pp. 73–118, 1975. doi: https://doi.org/10.1016/S0049-237X(08)71945-1.

[Cur34] H. B. Curry, "Functionality in Combinatory Logic," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, Nov. 1934, doi: 10.1073/pnas.20.11.584.

[How80] W. A. Howard, "The formulae-as-types notion of construction," *To H.B. curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, pp. 479–490, 1980.

[Bru83] N. G. de Bruijn, "AUTOMATH, a language for mathematics," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 159–200, 1983. doi: 10.1007/978-3-642-81955-1_11.

[Con+86] R. L. Constable *et al.*, *Implementing mathematics with the Nuprl proof development system*. USA: Prentice-Hall, Inc., 1986.

[Com20] T. m. Community, "The Lean mathematical library," Jan. 2020, pp. 367–381. doi: 10.1145/3372885.3373824.

[The24] The Agda Community, "Cubical Agda Library." Accessed: Jan. 09, 2025. [Online]. Available: https://github.com/agda/cubical

[GM05] B. Grégoire and A. Mahboubi, "Proving equalities in a commutative ring done right in coq," in *Proceedings of the 18th international conference on Theorem Proving in Higher Order Logics*, in TPHOLs'05. Berlin, Heidelberg: Springer-Verlag, Aug. 2005, pp. 98–113. doi: 10.1007/11541868_7.

[Hat04] A. Hatcher, "Spectral Sequences," *Algebraic Topology*. 2004. Accessed: Sep. 25, 2024. [Online]. Available: https://pi.math.cornell.edu/~hatcher/AT/SSpage.html

[McC01] J. McCleary, *A user's guide to spectral sequences*, 2nd ed., no. 58. in Cambridge studies in advanced mathematics. New York: Cambridge University Press, 2001.

[Kra15] N. Kraus, "The General Universal Property of the Propositional Truncation," *LIPIcs, Volume 39, TYPES 2014*, vol. 39, pp. 111–145, 2015, doi: 10.4230/LIPIcs.TYPES.2014.111.