Li, Nuo (2015) Quotient types in type theory. PhD thesis, University of Nottingham.

Quotient Types in Type Theory

Nuo Li, BSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

September 2014

Abstract

Martin-Löf's intuitionistic type theory (Type Theory) is a formal system that serves not only as a foundation of constructive mathematics but also as a dependently typed programming language. Dependent types are types that depend on values of other types. Type Theory is based on the Curry-Howard isomorphism which relates computer programs with mathematical proofs so that we can do computer-aided formal reasoning and write certified programs in programming languages like Agda, Epigram etc. Martin Löf proposed two variants of Type Theory which are differentiated by the treatment of equality. In Intensional Type Theory, propositional equality defined by identity types does not imply definitional equality, and type checking is decidable. In Extensional Type Theory, propositional equality is identified with definitional equality which makes type checking undecidable. Because of the good computational properties, Intensional Type Theory is more popular, however it lacks some important extensional concepts such as functional extensionality and quotient types.

This thesis is about quotient types. A quotient type is a new type whose equality is redefined by a given equivalence relation. However, in the usual formulation of Intensional Type Theory, there is no type former to create a quotient. We also lose canonicity if we add quotient types into Intensional Type Theory as axioms. In this thesis, we first investigate the expected syntax of quotient types and explain it with categorical notions. For quotients which can be represented as a setoid as well as defined as a set without a quotient type former, we propose to define an algebraic structure of quotients called *definable quotients*. It relates the setoid interpretation and the set definition via a normalisation function which returns a normal form (canonical choice) for each equivalence class. It can be seen as a simulation of quotient types and it helps theorem proving because we can benefit from both representations. However this approach cannot be used for all quotients. It seems that we cannot define a normalisation function for some quotients in Type Theory, e.g. Cauchy reals and finite multisets. Quotient types are indeed essential for formalisation of mathematics and reasoning of programs. Then we consider some models of Type Theory where types are interpreted as structured objects such as setoids, groupoids or weak ω-groupoids. In these models equalities are internalised into types which means that it is possible to redefine equalities. We present an implementation of Altenkirch's [3] setoid model and show that quotient types can be defined within this model. We also describe a new extension of Martin-Löf type theory called Homotopy Type Theory where types are interpreted as weak ω-groupoids. It can be seen as a generalisation of the groupoid model which makes extensional concepts including quotient types available. We also introduce a syntactic encoding of weak ω-groupoids which can be seen as a first step towards building a weak ω-groupoids model in Intensional Type Theory. All of these implementations were performed in the dependently typed programming language Agda which is based on intensional Martin-Löf type theory.

Acknowledgements

The first person I would like to thank is my supervisor Thorsten Altenkirch. He offered me a great internship opportunity about encoding numbers in Agda, which later developed into my PhD project. He was always patient in answering questions and taught me a lot about research. I would also like to thank him for his guidance and feedback on this thesis. I would also thank my second supervisor Thomas Anberrée who introduced me to functional programming and gave me precious advice on my project. I would also like to thank Venanzio Capretta who examined my first and second year reports and provided me with helpful feedback.

I would like to thank Ambrus Kaposi and Nicolai Kraus for their great help in writing my thesis and comments on my drafts, and correcting my grammatical errors for the final version. My friends Nicolai Kraus, Ambrus Kaposi, Christian Sattler, Paolo Capriotti, Florent Balestrieri, Gabe Dijkstra, Ivan Perez, Neil Sculthorpe and all the other PhD students in our lab helped me a lot by either teaching me mathematics, discussing research topics or playing badminton to exercise our bodies. The Functional Programming Lab is like a lively family and I really enjoyed the time here. I would like to thank everyone here. Without their kind help, the thesis would not have been finished.

I would also like to thank the organizers and other participants of the special year on Homotopy Type Theory at the Institute for Advanced Study where we had many interesting discussions some of which were related to parts of this work, especially Guillaume Brunerie whose proposal made it possible. I would also like to thank F. Nordvall Forsberg with whom we had important discussions related to our work.

I would also like to thank my parents and other family members. During the years in Nottingham, my mother Guangfei Lv and my father Youyuan Li were always very supportive of me, even though I was living far from them. They talked to me a lot and sent my favourite foods to me many times. My aunt Guangshu Lv who is living in Germany also helped me a lot. She gave me many advice and has sent mails to me. Without their support I would have never achieved my goals.

Finally, I would like to thank the School of Computer Science and the International Office at the University of Nottingham who financially supported this project by providing a scholarship.

Contents

Chapter 1

Introduction

Martin-Löf type theory (or just Type Theory) is a type theory which serves as a foundation of constructive mathematics and is also a dependently typed programming language. Different from other foundations like set theory, it is not based on predicate logic but internalises the BHK interpretation of intuitionistic logic through the Curry-Howard isomorphism. It identifies propositions with types such that proofs of a proposition become terms of the corresponding type. Viewed as a programming language, this means that we can express a specification as a type, and a program of that type will satisfy the specification. Moreover, one can write programs and reason about them in the same language resulting in certified programs. Implementations of Type Theory include NuPRL, LEGO, Coq, Agda, Epigram, Pi-Sigma.

Viewed as a foundation for mathematics, Type Theory is a powerful tool for constructively proving theorems with computerised verification. An example is the formal proof of the four-colour theorem by Georges Gonthier [42] [1].

There are two versions of Martin-Löf type theory, the *intensional* version (Intensional Type Theory or ITT) and the *extensional* version (Extensional Type Theory or ETT). They differ in the treatment of two notions of equality, *propositional equality* and *definitional equality*. In ITT, if two expressions can be computed to the same object then we make the judgement that they are definitionally equal. On

[1] More formalised mathematics can be found in [80].

the other hand, we have the *identity type* or propositional equality which is a type expressing the equality of two terms. Definitional equality implies propositional equality, but not the other way around which is usually called equality reflection. In ETT, they are identified, which makes definitional equality and thereby type checking undecidable.

In Intensional Type Theory, propositional equality is intensional. Some extensional equality types such as the equality of two point-wise equal functions, the equality of two logically equivalent propositions, and equality of two "equivalence classes" of a quotient $[a]$, $[b]$ where $a \sim b$, are not inhabited. There are several extensional concepts (see Section 2.4) which are useful and justifiable but not available in ITT. Nevertheless ITT is still preferable to ETT as the basis for programming languages, because of its good computational properties. Therefore, we would like to extend ITT with these extensional concepts, and the notion of quotient types is one of them.

1.1 Quotient types

Quotient is a primitive notion in mathematics. In arithmetic, quotient refers to the result of a division:

$$8 \div 4 = 2 \quad \text{or} \quad 8/4 = 2.$$

The notion is generalised in more abstract branches of mathematics, such as set theory, group theory, topology etc. For example in set theory, given a set A and an equivalence relation $\sim$, the set of all equivalence classes of $\sim$ is called the *quotient set* of A by $\sim$.

An **equivalence relation** is a binary relation which is

- reflexive: $\forall a \in A, a \sim a$,
- symmetric: $\forall a\ b \in A, a \sim b \to b \sim a$

- transitive: $\forall a\ b\ c \in A, a \sim b \to b \sim c \to a \sim c$.

The **equivalence class** of an element a is a subset of A which contains all elements equivalent to a:

$$[a] = \{x \in A \mid a \sim x\}$$

The **quotient set** of A by $\sim$ is just the set of equivalence classes:

$$A/\!\sim\, = \{[a] \mid a : A\}$$

Similarly, we can also "divide" a group, space, category or another algebraic structure by a given structure-preserving equivalence relation on it.

Naturally one would also expect **quotient types** in Type Theory. Intuitively speaking, a *quotient type* $A/\!\sim$ is a type A whose equality is redefined by an equivalence relation on it. In Extensional Type Theory, it is possible to redefine the equalities of types. For example, in NuPRL which is an implementation of Extensional Type Theory, there is a quotient operator which builds a new type from a given type and an equivalence relation on it [30]. However it is not possible to recover the witness of the equality between two equal elements in quotient types [71].

Because of the good computational properties, we would like to have quotient types in Intensional Type Theory as well. However in the traditional formulation of Intensional Type Theory, such a type former does not exist because there is no attached equivalence on each type except definitional equality which can not be changed. Instead **setoids** are usually used to represent quotients:

Definition 1.1. ***Setoid****.* *A setoid* $(A, \sim, eqv_{\sim})$ *(usually written as just* $(A, \sim)$*) consists of*

1. *a set (type)* $A : \mathbf{Set}$*,*

2. *a binary relation* $\sim: A \to A \to \mathbf{Prop}$*, and*

3. *a proof that it is an equivalence, i.e. reflexive, symmetric and transitive.*

Notice that this notion is also called a *total setoid.* If the relation of the setoid is not required to be reflexive it is called a *partial setoid.* In this thesis, the word "setoid" refers to a total setoid.

A function $f : A \to B$ is well-defined on a setoid $(A, \sim)$ only if it respects $\sim$:

Definition 1.2. *We say a function $f : A \to B$* ***respects*** *$\sim$ if*

$$\forall (x, y : A) \to x \sim y \to f(x) =_B f(y)$$

However using setoids to represent quotients is not an ideal solution. Since it is an alternative representation of sets, everything defined on **Set** has to be redefined on **Setoid** again. Examples are functions between setoids, equalities on setoids, products on setoids, etc. In fact, in other branches of mathematics, the quotient object is essentially the same kind of object as the base one. Therefore, it is better to have a representation of the quotient $A/\sim$ which is in the same sort as A is.

In fact not all quotients have to be defined using a quotient type former. For example integers are usually represented as pairs of natural numbers $\mathbb{N} \times \mathbb{N}$ which are equivalent if subtracting first number from the second gives the same result. This gives rise to a quotient. However the set of integers can also be defined inductively from the observation that $\mathbb{Z} \simeq \mathbb{N} + \mathbb{N}$. For such quotients, the set definition can be seen as a normal form of the equivalence classes. There is a mapping from the setoid representation to the set representation called the **normalisation function**. In this thesis we say that such quotients are *definable via a normalisation (function)* (see Chapter 4).

Some quotients are not definable via normalisation, for example the set of real numbers represented by Cauchy sequences of rational numbers, the finite multisets represented as lists quotiented by permutation equivalence (or bag equivalence [36]), the non-terminating programs represented by the partiality monad quotiented by weak bisimilarity and so on. In these cases, a general schema to define quotient types is essential.

If we simply introduce quotient types as axioms in Intensional Type Theory, we lose the *canonicity* property, in other words, we can construct non-canonical terms of $\mathbb{N}$ which can not be reduced to numerals (see Theorem 3.4). In fact, similar issues arise when adding other extensional concepts as axioms e.g. functional extensionality. Therefore it is essential to find a computational interpretation of these extensional concepts including quotient types.

To achieve these goals, we have to "refine" our interpretation of types. Usually a type is treated as a set without attached equality. If a type is interpreted as a *setoid*, in other words internalising propositional equality, quotient types can be defined simply by replacing “internal” equality. This is called *setoid interpretation* which is inspired by Bishop’s [20] definition of sets and has been studied by Martin Hofmann [47, 48] and Thorsten Altenkirch [3, 8]. Based on this interpretation, we can build a setoid model in Intensional Type Theory which gives us the computational interpretation of quotient types.

For a long time, the nature of identity types was mysterious in Intensional Type Theory. Intuitively, the uniqueness of identity proofs (UIP), stating that two terms of the same identity type are always propositionally equal, is valid because there is at most one canonical element expressing the equality between two objects. However UIP is not derivable from the eliminator for identity type J (see Section 2.2.1) but needs an extra eliminator K suggested by Thomas Streicher [78]. Furthermore, Hofmann and Streicher [51] propose a groupoid interpretation of Intensional Type Theory where K is refuted, hence UIP fails. The groupoid interpretation can be seen as a generalisation of the setoid one, where the identity type is not a proposition but a set. It means that there can be several proofs of the same identity which are not equal.

In fact, the groupoid interpretation of types can be extended to ω-groupoids which are generalisations of groupoids. Roughly speaking, an ω-groupoid consists of objects, morphisms between objects, morphisms between morphisms and so on, having infinite levels of morphisms. All of these morphisms are isomorphisms which hold up to higher isomorphisms. These isomorphisms are called equivalences. An introduction to ω-groupoids is given in Section 2.6.2. Since Grothendieck’s homotopy hypothesis states that ω-groupoids are spaces [14], we can interpret types as spaces indeed. In recent years, such an interpretation has been developed into a new field called Homotopy Type Theory. In Homotopy Type Theory, types are interpreted as spaces (abstractly) or as *weak* ω-groupoids. However, it is very difficult to describe all levels of coherence conditions of *weak* ω-groupoid such as groupoid laws. A more commonly used approach is therefore to define them in terms of Kan simplicial sets or cubical sets (See Section 2.6.5). Nevertheless, it is possible to build a syntactic type theory to describe weak ω-groupoids in Intensional Type Theory (see Chapter 7).

In Homotopy Type Theory, the most important axiom is *univalence* which was suggested by Voevodsky [88]. Roughly speaking, univalence states that identity of types corresponds to equivalence. Many extensional concepts are derivable from this axiom, including functional extensionality, propositional extensionality, quotient types. For example, Voevodsky has proposed an impredicative encoding of quotient types (see Section 3.4.1). The computational interpretation of univalence remains an open problem, but it is likely to be solved by a recently proposed model called *cubical sets model* (Bezem, Coquand and Huber [18]).

Quotient types can be applied in the formalisation of mathematics and in program verification. As we mentioned before, one of the fundamental mathematical notions, *real numbers* can be defined as a quotient where the base set is the set of Cauchy sequences of rational numbers. From a programming perspective, they provide more algebraic datatypes and enables us to reason about infinite types and semantics-based verification of concurrent programs as suggested by Hofmann [48].

1.2 Structure of the thesis

In Chapter 2, we introduce Martin-Löf type theory as the basis of our study. We briefly describe its history and present its basic rules. We also introduce our main technical tool – Agda, a dependently typed functional programming language based on the intensional version of Martin-Löf type theory. Then we discuss the missing extensional concepts in Intensional Type Theory excluding quotient types. We also describe Homotopy Type Theory which is an extension of Martin-Löf type theory by the univalence axiom and higher inductive types which allow constructors for internal equalities. We discuss how this theory gives rise to extensional concepts.

In Chapter 3, we provide the syntactic rules of quotient types together with a discussion of effectiveness. Categorically speaking, a quotient type is a *coequalizer*. We also explain the rules of quotient types given by an adjunction. In Homotopy Type Theory, our quotient types become quotient *sets*. We first introduce Voevodsky's impredicative encoding of quotient sets together with proofs that all essential rules are derivable. We also introduce quotient inductive types (QITs) i.e. quotient sets defined using higher inductive types.

In Chapter 4, we introduce one of our original developments, the definable quotient structure. We observe that there are some quotients which are definable inductively in Martin-Löf type theory without adding a new quotient type formation rule. A definable quotient consists of a setoid representation $(A, \sim)$, a set representation Q and a normalisation function $[\_] : A \to Q$ which gives the normal form for each "equivalence class". As an example, integers can be encoded as the quotient types of paired natural numbers over the equivalence relation that two pairs are equal if they share the same result of subtraction. Integers can also be defined inductively as a set. The definable quotients structure is an abstraction of the relation between the two representations and provides a flexible way of conversing between them. In fact, it can be seen as a manual construction of quotient types.

In Chapter 5, we discuss quotients that are not definable as an inductive type with a normalisation function, such as the real numbers, finite multisets and the partiality monad. We present a proof of the undefinability of real numbers as Cauchy sequences $(R_0/\sim)$ with a normalisation function. The proof was mainly conducted by Nicolai Kraus. The proof is based on Brouwer's continuity principle – all definable functions are continuous, which is inconsistent if we have it within Martin-Löf type theory as shown by Escardo and Xu [40] but holds meta-theoretically. We prove that $R_0/\sim$ is *connected*, and it implies that all functions $R_0 \to R_0/\sim$ that respect the equivalence relation of Cauchy sequences are constant. Therefore there is no definable normalisation endofunction for Cauchy sequences. Similarly we also prove that non-terminating programs encoded using the partiality monad quotiented by weak bisimilarity, are also undefinable with a normalisation function. For unordered tuples such as unordered pairs and finite multisets represented by lists quotiented by permutation, it is also impossible for to find a canonical normalisation function unless the underlying set has a decidable total order.

In Chapter 6, we discuss several models of Type Theory where quotient types are available. We present an implementation of the setoid model encoding extensional concepts. The work is an extension of the setoid model by Altenkirch [3] with quotient types. Some other models including models of Homotopy Type Theory are also discussed.

In Chapter 7, we present a new formalisation of the syntax of weak ω-groupoids in Agda using heterogeneous equality. We show how to recover basic constructions on ω-groupoids using suspension and replacement. In particular we show that any type forms a groupoid and we outline how to derive higher dimensional composition. We present a possible semantics using globular sets and discuss the issues which arise when using globular types instead. The work in the chapter has been published in [11] together with Thorsten Altenkirch and Ondřej Rypáček.

In the Appendices, we show our Agda code corresponding to the work in Chapter 4, Chapter 6 and Chapter 7.

Chapter 2

Type Theory

Type theory usually refers to a formal system in which terms always have a type. It was initially invented as a foundation of mathematics as an alternative to set theory, but it also works well in computer science as a programming language in which we can write certified programs. There are a variety of type theories, like Russell's theory of types, simply typed λ-calculus, Gödel's System T [41] etc. In this thesis we mainly focus on Per Martin-Löf's intuitionistic type theory. There are also different versions of Martin-Löf type theory and the intensional version (Intensional Type Theory for short) has better computational behaviour and is widely used in programming languages like Agda, Epigram etc. However, several desirable extensional concepts such as functional extensionality and quotient types are not available in Intensional Type Theory. Much research has been done to extend Type Theory with these concepts and new interpretations of type theory are popular and reasonable solutions. Homotopy Type Theory is one of them and is also a variant of Martin-Löf type theory and connected to homotopy theory.

In this chapter we will first briefly introduce the original motivation and evolution of type theory. Then we explain important notions in Martin-Löf type theory, and a list of extensional concepts will be presented. Finally we will describe the programming language Agda which is an implementation of the intensional version of Martin-Löf type theory.

2.1 A brief history of Type Theory

Type theory was first introduced as a refinement of set theory. In the 1870s, Georg Cantor and Richard Dedekind founded set theory as a branch of mathematical logic and started to use set theory as a language to describe definitions of various mathematical objects. In the 1900s, Bertrand Russell discovered a paradox in this system. In naïve set theory, there was no distinction between small sets like the set of natural numbers and "larger" sets like the set of all sets.

Example 2.1 (Russell's Paradox)**.** *Let R be the set of all sets which do not contain themselves $R = \{x \mid x \notin x\}$. Then we get a contradiction $R \in R \iff R \notin R$.*

To avoid this paradox, Russell found that we have to make a distinction between objects, predicates, predicates of predicates, etc. Then Russell proposed the theory of types [76] where the distinction is internalised by types. In this simple type theory, each mathematical object is assigned a type. This is done in a hierarchical structure such that "larger" sets and small sets reside in different levels. The "set" of all sets is no longer a small set, hence the paradox disappears.

In type theory, The elementary notion *type* plays a similar role to set in set theory, but differs fundamentally. Every term comes with its unique type while in set theory, an element can belong to multiple sets. For example to introduce a term of natural number 2, we have to use a typing judgement $2 : \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. The terms are usually constructed using a list of constructors belonging to a type. Hence an integer term $2 : \mathbb{Z}$ is constructively different to $2 : \mathbb{N}$ in type theory.

Following the idea of theories of types, various type theories have been developed. Simply typed lambda calculus (or Church's theory of types) is the first type theory to introduce functions as primitive objects [33]. It was originally introduced by Alonzo Church in 1940 to avoid the Kleene-Rosser paradox [56] in his untyped lambda calculus.

Example 2.2 (Kleene-Rosser paradox)**.** *Suppose we have a function $f = \lambda x.\neg(x\ x)$, then we can deduce a contradiction by applying it to itself:*

$$ff = (\lambda x.\neg(x\ x))f = \neg(f\ f)$$

Type theory is applied in various fields including computer science. For instance, Haskell was originally based on one of the variants of lambda calculus called System F[1].

In 1970s, Per Martin-Löf [64, 68] developed his profound intuitionistic type theory (also called Martin-Löf type theory). In this thesis, we will refer to this system when using the term *Type Theory*. Type Theory serves as a foundation for constructive mathematics [66] and can also be used as a functional programming language [81] in which the evaluation of a well-typed program always terminates [72].

From early type theories like that of Russell and Church to modern type theories like de Bruijn's Automath, Martin-Löf type theory and Coquand's Calculus of Constructions (CoC), one of the most important extensions and discoveries is the correspondence between mathematical proofs and computer programs (terms). Different to set theory whose axioms are based on first-order logic, in modern type theories, intuitionistic logic concepts can be encoded as types through the **Curry-Howard isomorphism (correspondence)**. The American mathematician Haskell Curry and logician William Alvin Howard first discovered a correspondence between logic and computation. They found that propositions can be encoded as types and proofs can be given by constructing terms (programs). The idea also relates to the Brouwer–Heyting–Kolmogorov (BHK) interpretation of intuitionistic logic. For example, a proof of $P \wedge Q$ can be encoded as the product type $P \times Q$ which contains a proof of $p : P$ and a proof of $q : Q$. Computationally, implications are function types, conjunctions are product types, true is the unit type, false is the empty type etc. With dependent types (introduced below), the correspondence extends to predicate logic: the universal and existential quantification correspond to dependent functions and dependent sums. This feature turns Type Theory into a programming language where we can formalise proofs as computer programs. We can do computer-aided reasoning about mathematics as well as programs. From a programmer's perspective, it provides a programming language where we can write certified programs.

[1] It has evolved into System FC recently.

Another central concept in Martin-Löf type theory is **Dependent types**. A dependent type is a type which depends on values of other types [21]. It provides us with the means for defining families of types, for example the family of lists with explicit length called *Vector*, for example Vec $\mathbb{N}$ 3 stands for a three element list of type $\mathbb{N}$. Since the type carries more information, the program specifications can be expressed more accurately. In the example of vectors, we can write a look-up function without "index out of range" problems. It is much simpler to write matrix multiplication with dependent types.

The 1971 version of Martin-Löf type theory [64] was impredicative and turned out to be inconsistent due to Girard's paradox [53]. It is impredicative in the sense that the universe of types is impredicative. The notion of a **universe of types** was first used by Martin-Löf [65] to describe the type of all types and usually denoted as U. An impredicative universe U has an axiom $\mathsf{U} : \mathsf{U}$. Starting from the 1972 version [67], a predicative hierarchy of universes was adopted. Briefly speaking, we start with a universe of small types called U_0 and for each $n : \mathbb{N}$ we have $U_n : U_{n+1}$ which forms a cumulative hierarchy of universe. There is a more detailed introduction to the notion of universe written by Erik Palmgren [74].

Equality is one of the most contentious topics in Type Theory. In everyday mathematics the notion of equality is used to describe sameness and taken as granted. But in Type Theory, we have different notions of equality or equivalence of the terms. First of all **definitional equality** (or **judgemental equality** [66]) denoted $a \equiv b$ is a meta-theoretic equality, which holds when two terms have the same normal forms [72]. Usually it already includes **computational equality** which is the congruence on terms generated from reduction rules like β-reduction and η-expansion.

Since equalities are also propositions, they can be encoded as types. In the 1972 version of Martin-Löf type theory, there is a type for the equality of natural numbers. It is defined by pattern matching on the two numbers and eventually reduces to unit type or empty type.

In the 1973 version [65], Martin Löf introduced an equality type which works for every type, not only for natural numbers. It is called **identity type** or **intensional propositional equality** or **intensional equality**. It is denoted e.g. for natural numbers by $\mathrm{Id}_{\mathbb{N}}(a, b)$ or $a =_{\mathbb{N}} b$ (see subsection 2.2.1).

In Intensional Type Theory (ITT or TT_I for short), like the 1973 version or Agda, propositional equality is different from definitional equality. The definitional equality is always decidable hence type checking that depends on definitional equality is decidable as well [3].

In Extensional Type Theory (ETT or TT_E for short), like the 1980 version [66] or NuPRL, propositional equality is reflected in definitional equality, in other words, two propositionally equal objects are judgementally equal. This is achieved by the **equality reflection rule**:

$$\frac{a = b}{a \equiv b}\ \text{ID-DEFEQ} \tag{2.1}$$

and the **uniqueness of identity proofs**:

$$\frac{p : a = b}{p \equiv \mathrm{refl}}\ \text{ID-UNI} \tag{2.2}$$

Notice that this version of UIP type checks only if we have equality reflection. In some versions of Intensional Type Theory, UIP also holds in other forms, see Section 2.4.

Due to the addition of equality reflection, type checking becomes undecidable because it has to respect propositional equality which is not decidable in general. For example, the equality reflection rule implies functional extensionality which is not decidable.

Intensional Type Theory is more widely used as a programming language (examples are Coq, Agda, Epigram), because its definitional equality is decidable, hence its type checking is decidable and programs written in it are terminating.

However in Intensional Type Theory, **extensional concepts** are not available. For example extensional equality of functions, equality of different proofs for the same proposition, and quotient types. Simply adding these concepts as axioms can result in non-canonical objects e.g. a term of $\mathbb{N}$ which does not reduce to a numeral (see Theorem 2.4).

To add these extensional concepts into Intensional Type Theory without losing decidable type checking and canonicity, it seems that types have to be interpreted with more complicated structures than sets. In the 1990s, some models of Type Theory were proposed such as Hofmann's setoid model, Altenkirch's setoid model, Hofmann and Streicher's groupoid model etc. The idea of viewing types as groupoids later inspired other mathematicians. For example, Warren [93] interprets types as strict ω-groupoids.

Recently, Voevodsky proposed a new interpretation of intensional Martin-Löf type theory by homotopy-theoretic notions [55, 89] called Homotopy Type Theory (see Section 2.6), or univalent foundations of mathematics. Type are treated as *spaces* or *higher groupoids*, and terms are *points* of this space, and more generally, functions between types are *continuous maps*. Identity types are *paths*, identity types of identity types are *homotopies*. Although these notions are originally defined with topological notions, in Type Theory they are treated purely homotopically. Equality is internalised as a type so that types have infinite levels of higher structures as weak ω-groupoids.

The new interpretation clarifies the nature of equality in Type Theory. The central idea of Homotopy Type Theory is univalence which can be understood as the property that isomorphic types are equal. In regular mathematics we usually do abstract reasoning on structures which applies to all isomorphic structures, because they can not be distinguished from other objects, hence isomorphic structures can be identified. Univalence can be seen as a formal acceptance of this idea in Type Theory such that we can do abstract reasoning about types. Moreover, many extensional concepts arise from it automatically. The interpretation also helps mathematicians to reason about homotopy theory in programming languages.

To summarise, we present a list of different versions of Martin-Löf type theory:

1. The 1971 version [64] has an impredicative universe, i.e. $\mathsf{U} : \mathsf{U}$, and it turned out to be inconsistent by Girard's paradox.

2. The 1972 version which was published in 1996 [67] abandons the impredicative universe and all later versions are predicative. It does not have an inductive identity type but recursively defines equality for given types e.g. $\mathbb{N}$.

3. The 1973 version [65] introduced the inductively defined identity type internalising equality as a type.

4. The 1980 version which is summarised by Giovanni Sambin in 1984 [66] is extensional. It adopts equality reflection, namely an inhabitant of an identity type implies definitionally equality.

5. In the homotopic version [82], Vladimir Voevodsky extends it with univalence axiom and provides a homotopic interpretation of it.

2.2 The formal system of Type Theory

The formal type system of Type Theory is given by a list of judgements and a sequence of rules deriving such judgements. We will use the following judgements in this thesis:

$\Gamma \vdash$	Γ is a well formed context
$\Gamma \vdash A$	A is a well formed type
$\Gamma \vdash a : A$	a is a well typed term of type A in context Γ
$\delta : \Gamma \Rightarrow \Delta$	δ is a substitution from context Γ to Δ

We also have equality judgements for contexts, types, terms and substitution. For instance,

$\Gamma \vdash a \equiv a' : A$ $\quad$ a and a' are definitionally equal terms of type A in context Γ

In Intensional Type Theory the judgemental equality $\equiv$ is the same as definitional equality, while propositional equality is usually expressed by an inhabitant of the identity type $\Gamma \vdash p : a =_A a'$.

Throughout the thesis, we use the following notational conventions:

- Γ, Δ for contexts
- γ, σ for substitutions
- A, B, C for types
- a, b, c, t, x for terms
- $:\equiv$ for definitions
- **Set** or $\mathbf{Set}_0$ for the universe of small types, $\mathbf{Set}_1$, $\mathbf{Set}_2$, ... for higher universes

2.2.1 Rules for types

The rules describe how one can derive the judgements above. They are syntactic rules but the semantic meaning may be revealed from the construction. The rules for each type former are usually classified as a formation rule, introduction rule, elimination rule, computation rule (β) and uniqueness rule (η). Here we will only show the rules for the most important types. The substitution rules are not discussed here but a good reference is [48]).

First of all, a **context** is either empty (denoted as ()) or extended by context comprehension:

$$\frac{\Gamma \vdash \qquad \Gamma \vdash A}{\Gamma, x : A \vdash} \quad (\textsc{comprehension})$$

In practice, the empty context is usually not written, for example $\vdash \mathbb{N}$.

Π-types (dependent function type)

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Pi\ (x : A)\ B} \ (\Pi\textsc{-form}) \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \Pi\ (x : A)\ B} \ (\Pi\textsc{-intro})$$

$$\frac{\Gamma \vdash f : \Pi\ (x : A)\ B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]} \quad (\Pi\textsc{-elim})$$

In the expressions like $\lambda(x : A).b$, λ binds the free occurrences of x in b. In the expressions like $B[a/x]$ or $b[a/x]$ we do a *standard substitution* in type B or term b that replaces free occurences of x by a. We will use a shorthand notation for substitution later, for example, $C[a, b]$ for $C[a/x, b/y]$ where the order of arguments corresponds to the order in the typing rule.

In this thesis, we also adopt a generalised arrow notation to write Π-types, for example $(x : A) \to B$, and their terms $\lambda(x : A) \to b$.

computation rule uniqueness rule

$$(\lambda(x : A) \to b)(a) \equiv b[a] \qquad\qquad f \equiv \lambda x \to f(x)$$

Σ-types (dependent product type)

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Sigma\ A\ B}\ (\Sigma\text{-}\textsc{form}) \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : \Sigma\ A\ B}\ (\Sigma\text{-}\textsc{intro})$$

There are two ways to eliminate a term of a Σ-type:

$$\frac{\Gamma \vdash t : \Sigma\ A\ B}{\pi_1(t) : A}\ (\Sigma\text{-}\textsc{proj}_1) \qquad \frac{\Gamma \vdash t : \Sigma\ A\ B}{\pi_2(t) : B[\pi_1(t)]}\ (\Sigma\text{-}\textsc{proj}_2)$$

The computation rules are

$$\pi_1\ (a, b) \equiv a \text{ and } \pi_2\ (a, b) \equiv b$$

and the uniqueness rule is

$$t \equiv (\pi_1\ t, \pi_2\ t).$$

Identity type

The identity type is a notion of intensional propositional equality given by the following rules:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash a : A, \qquad \Gamma \vdash a' : A}{\Gamma \vdash a =_A a'} \ (=\text{-FORM}) \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}(a) : a =_A a} \ (=\text{-INTRO})$$

We use $a =_A a'$ instead of $\mathrm{Id}_A(a, a')$ to denote the identity type, or simply $a = a'$.

$$\frac{\Gamma, x : A, y : A, p : x =_A y \vdash C \qquad \Gamma, x : A \vdash t(x) : C[x, x, refl(x)] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a' : A \qquad \Gamma \vdash p : a =_A a'}{\Gamma \vdash \mathsf{J}(t, a, a', p) : C[a', a', p]} \ (\mathsf{J})$$

Its computation rule is

$$\mathsf{J}(t, a, a, r(a)) \equiv t(a).$$

The *uniqueness of identity proofs* (UIP) is not a consequence of J but another eliminator called K (see Section 2.4).

Definition 2.1. ***"subst" function.***

Given a type family $B : A \to \mathbf{Set}$, and $p : a =_A a'$, we can easily define a function of type $B(a) \to B(a')$ by applying J:

Let

$$C(x, y, p) :\equiv B(x) \to B(y)$$

$$t(x) :\equiv id$$

Thus,

$$\mathsf{subst}(B, p) :\equiv: \mathsf{J}(t, a, a', p) : B(a) \to B(a')$$

For simplicity, if we have a term $b : B(a)$, we write the result of subst *as* $\mathsf{subst}(B, p, b) : B(a')$.

Unit type

$$\frac{}{\vdash \top}\ (\top\text{-}\textsc{form}) \qquad\qquad \frac{}{\vdash \mathsf{tt} : \top}\ (\top\text{-}\textsc{intro})$$

$$\frac{\Gamma, x : \top \vdash A \qquad \Gamma \vdash t : A[\mathsf{tt}]}{\vdash t : A}\ (\top\text{-}\textsc{elim})$$

Empty type

$$\frac{}{\vdash \bot}\ (\bot\text{-}\textsc{form}) \qquad\qquad \frac{\Gamma \vdash A \qquad e : \bot}{\Gamma \vdash \mathrm{abort}(e) : A}\ (\bot\text{-}\textsc{elim})$$

There is no term of the empty type so there is no introduction rule.

Universe types

$$\frac{}{\Gamma \vdash \mathsf{U}}\ (\mathsf{U}\text{-}\textsc{form}) \qquad\qquad \frac{\Gamma \vdash \hat{A} : \mathsf{U}}{\Gamma \vdash \mathsf{El}(\hat{A})}\ (\mathsf{U}\text{-}\textsc{El})$$

$$\frac{}{\Gamma \vdash nat : \mathsf{U}}\ (\mathsf{U}\text{-}\textsc{intro-nat}) \qquad\qquad \frac{\Gamma \vdash \hat{A}, \hat{B} : \mathsf{U}}{\Gamma \vdash arr(\hat{A}, \hat{B}) : \mathsf{U}}\ (\mathsf{U}\text{-}\textsc{intro-arr})$$

The computation rules are

$$\begin{aligned} \mathsf{El}(nat) &\equiv \mathbb{N} \\ \mathsf{El}(arr(\hat{A}, \hat{B})) &\equiv \mathsf{El}(\hat{A}) \to \mathsf{El}(\hat{B}) \end{aligned}$$

The notation of $\hat{A}$ indicates that it is a code for a type (a term of U) rather than a type.

Inductive types

Inductive types are a self-referential schema to define new types by specifying a collection of *constructors* which can be constants or functions.

The formation and introduction rules are enough to build a type inductively. Natural numbers $\mathbb{N}$: **Set** can be defined as follows:

- $0 : \mathbb{N}$
- $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$

The terms are freely generated by a finite list of these constructors, for instance, suc (suc 0) stands for natural number 2. They are similar to data structures in programming languages, and most implementations of Type Theory have inductive types along with structural recursion to eliminate from them.

Coinductive types

Coinductive types can be seen as infinitary extensions of inductive types [26]. A typical example of an infinite data structure is the type of streams (or infinite lists). A stream of type A has one constructor:

- $\mathrm{cons} : A \to \mathrm{Stream}\ A \to \mathrm{Stream}\ A$

An object of it can be destructed into an element of A and again a stream of A, in other words, it can continuously produce terms of type A. To manipulate coinductive types, we usually use corecursion which can be non-terminating but has to be productive. For example a stream of 0s can be constructed by:

$$\mathrm{zeros} = \mathrm{cons}(0, \mathrm{zeros})$$

Note that the manner of using coinductive types varies in different languages. For further reference, one can read [26].

2.3 An implementation of Type Theory: Agda

Agda is a dependently typed functional programming language which is based on the intensional version of Martin-Löf type theory [94].

- *Functional programming language.* As the name indicates, functional programming languages emphasise the application of functions rather than changing data in imperative style like C++ and Java. The basis of functional programming is the lambda calculus. There are several generations of functional programming languages, for example Lisp, Erlang, Haskell, SML etc. Agda is a pure functional programming language which offers lazy evaluation (see subsection 2.3.1) like Haskell. In a pure language, side effects are eliminated which means we ensure that the result will be the same no matter how many times we input the same data.

- *Implementing Per Martin-Löf Type Theory.* Agda is based on the Curry-Howard isomorphism [22]. It means that we can reason about mathematics and programs by constructing proofs as programs. In many languages the correctness of programs has to be verified on the meta-level. However in Agda we verify programs within the same language, and express specifications and programs at the same time, as Nordström et al. [72] pointed out.

- *Dependent types.* As a feature of Martin-Löf intuitionistic Type Theory, types in Agda can depend on values of other types [21], which is different from Haskell and other Hindley-Milner style languages where types and values are distinct. It not only helps encoding quantifiers but also allows writing very expressive types which can be seen as program specifications resulting in programs being less error-prone. For example, in Agda the type of matrices comes with accurate size e.g. Matrix 3 4. Thus we can specify the multiplication of matrices as a function of type $\mathrm{Matrix}\ m\ n \to \mathrm{Matrix}\ n\ p \to \mathrm{Matrix}\ m\ p$ where $m, n, p : \mathbb{N}$.

2.3.1 Features

Some features of being a functional programming language make theorem proving easier,

- *Pattern matching.* The mechanism for dependently typed pattern matching is very powerful [9]. Pattern matching is a more intuitive way to use terms than eliminators. For example, to prove symmetry of identity by pattern matching on a term of identity type, the only possible case refl exists when a and b are identical, hence the result type becomes a≡a.

```
symm : {A : Set}{a b : A} → a ≡ b → b ≡ a
symm refl = refl
```

Using the eliminator J is more tedious:

```
symm' : {A : Set}{a b : A} → a ≡ b → b ≡ a
symm' = J (λ a b _ → b ≡ a) (λ _ → refl) _ _
```

- *Inductive & Recursive definition.* In Agda, types are often defined inductively, for example, natural numbers are defined as:

  ```
  data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ
  ```

  Functions on inductive types can be defined recursively using pattern matching. For example addition on natural numbers is defined as:

  ```
  _+_ : ℕ → ℕ → ℕ
  zero   + n = n
  suc m + n = suc (m + n)
  ```

  It also enables programmers to prove propositions in the same manner as mathematical induction and case analysis.

- *Lazy evaluation.* As a pure functional programming language, Agda offers lazy evaluation which eliminates unnecessary operation to delay a computation until we need its result. It is often used to handle infinite data structures [95].

Compared to other programming languages like Haskell, there is an interactive Emacs interface which provides a few important functions.

- *Type checker.* The type checker is an essential part of Agda. It will detect type mismatch problems when some code is loaded into Agda. It also includes a coverage checker and a termination checker. The *coverage checker* ensures that the patterns cover all possible cases so that programs do not crash [22]. The *termination checker* ensures that all Agda functions terminate [73]. As a theorem prover, the type checker ensures that the proof is complete and not defined by itself.

- *Interactive interface.* Agda has an Emacs-based interface for interactively writing and verifying proofs. As long as code is loaded, namely type checked, the code will be highlighted and problematic code is coloured by red for non-termination and yellow for not inferable implicit arguments. In the interactive Emacs interface, there are a few convenient short-cut keys, for example showing the context, refining the goal with a partial program, navigating to definitions of some functions or types. The refinement function helps us incrementally build programs with explicit context information. Thus type signatures are usually essential for accurate information. The code navigation alleviates a great deal of work of programmers to look up the documentation.

- *Unicode and mixfix support.* In Haskell and Coq, unicode support is not an essential part. The name of operations can be very complicated without enough symbols. Agda handles unicode characters and is able to handle unicode symbols like β, ∀ and ∃.

  It also uses a flexible mixfix notation where the positions of arguments are indicated by underscores. E.g. _ ⇒ _ is one identifier which can be applied to two arguments as in $A \Rightarrow B$.

  In the following type signature of the commutativity theorem for addition of natural numbers, ℕ and ≡ are unicode characters, + and ≡ are mixfix operators.

  ```
  comm : ∀ (a b : ℕ) → a + b ≡ b + a
  ```

Note that in Agda ≡ is used for the identity type. See discussion in Section 2.3.2.

Unicode symbols and the mixfix notation improves the readability and provides familiar symbols used in mathematics. Interestingly we could use some characters of other languages to define functions such as Chinese characters.

- *Implicit arguments and wildcards.* Sometimes it is unnecessary to state an argument. If an argument can be inferred from other arguments we can mark it as implicit with curly brackets. For example, whenever we feed an argument a to function id, the implicit type A is inferable:

  ```
  id : {A : Set} → A → A
  id a = a
  ```

  If an explicit argument can be automatically inferred or not used in the program definition, we can replace it with underscores as wildcards (see the code on symm' above in Section 2.3.1).

  In practice, the use of implicit arguments and wildcards makes the code more readable.

- *Module system.* The mechanism of parametrised modules makes it possible to define generic operations and prove a whole set of generic properties.

- *Coinduction.* We can define coinductive types such as streams in Agda:

  ```
  data Stream (A : Set) : Set where
    _::_ : A → ∞ (Stream A) → Stream A
  ```

  The coinductive occurrences in the definition are labelled with the delay operator ∞. To manipulate coinductive types and more generally mixed inductive/coinductive types [37], we use the delay operation ♯ and the force operation ♭ defined in module **Coinduction**:

$$\sharp : \forall\{A : \mathbf{Set}\} \to A \to \infty\ A$$
$$\flat : \forall\{A : \mathbf{Set}\} \to \infty\ A \to A$$

As an example, to add one to every object of a stream of natural numbers, we define the function using corecursion as follows:

```
plus1 : Stream ℕ → Stream ℕ
plus1 (n :: ns) = suc n :: ♯ plus1 (♭ ns)
```

- *Ring solver.* Compared to Coq, Agda has no tactics providing automated proof generation although it has a ring solver which plays a similar role to the tactic *ring*. It is easy to use for people who are familiar with constructive mathematics.

2.3.2 Agda conventions

The syntax of Agda has some similarities to Haskell or Martin-Löf type theory, but there are some important differences which may cause confusion:

- The meaning of $=$ is swapped with the one of $\equiv$. The symbol "$=$" is reserved for function definition following the convention in programming languages. The congruence symbol "$\equiv$” is used for the identity type. This is inconsistent with our conventional choice of symbols in articles.

- : is used for typing judgement, for example a:A, while double colon :: is the *cons* constructor for list. It is different from the usual notational conventions in Haskell.

- The universe of small types is $\mathbf{Set}_0$ or **Set** instead of **Type**, even though it is not a set in set-theoretical sense.

- The universe of propositions **Prop** ($\mathbf{Prop} \subset \mathbf{Set}$) is not available. Propositions are also in the universe **Set**. If necessary, we will postulate the proof-irrelevance property for a given proposition $P : \mathbf{Set}$.

- Agda has a more liberal way to define Π-types. Π-types are written in a generalized arrow notation $(x : A) \to B$ for $\Pi x : A.B$. Together with implicit arguments, it is valid to write a type signature as $\forall \{A : \mathbf{Set}\}(x : A) \to \{y : A\} \to x \equiv y$.

- Σ-types are defined in Agda standard library. There are also generalised Σ-types called *dependent record types* which can be defined with keyword **record**.
- In Agda, we use the Paulin-Mohring style identity type:

  ```
  data _≡_ {A : Set} (x : A) : A → Set where
    refl : x ≡ x
  ```

  It is parametrised by the left side of the identity and is equivalent to the original version.

2.4 Extensional concepts

In Intensional Type Theory, extensional (propositional) equality is not captured by the identity type which is intensional.

However, the identity type in intensional type theory is not powerful enough for formalisation of mathematics and program development. Notably, it does not identify pointwise equal functions (functional extensionality) and provides no means of redefining equality on a type as a given relation, i.e. quotient types. We call such capabilities extensional concepts.

Objects are extensionally equal if they have the same *observable* behaviour. In other words, they can be substituted by one another in any context without changing the output of the program. For example point-wise equal functions, different proofs of the same proposition etc. Extensional (propositional) equality is not captured by the identity type which is intensional. Thus in the traditional formulation of Intensional Type Theory, extensionality and some other related features of propositional equality like quotient types are not available. These *extensional concepts* have been summarised and comprehensively studied by Martin Hofmann [48]; a list of them are given as follows:

- **Functional extensionality**

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B \qquad \Gamma \vdash f, g : (x : A) \to B(x) \qquad \Gamma, a : A \vdash p : f(a) = g(a)}{\Gamma \vdash \mathrm{ext}(a, p) : f = g} \ (\textsc{fun-ext})$$

If two (dependent) functions are point-wise propositionally equal, they are (extensionally) propositionally equal. This is called functional extensionality which is not inhabited in the traditional formulation of Intensional Type Theory [3]. For example, two functions of type $\mathbb{N} \to \mathbb{N}$, $\lambda n \to n$ and $\lambda n \to n + 0$ are point-wise propositionally equal, but the intensional propositional equality of them is not inhabited due to the fact that $n + 0$ does not reduce to n (assuming that $\_ + \_$ is defined as the one in Section 2.3.1).

In Extensional Type Theory, functional extensionality is inhabited:

Theorem 2.2. *Functional extensionality is derivable from the equality reflection rule.*

Proof. Suppose $\Gamma, a : A \vdash p : f\,a = g\,a$, with the reflection rule we have $\Gamma, a : A \vdash f\,a \equiv g\,a$. Then using ξ-rule, we know that $\Gamma \vdash \lambda a.f\,a \equiv \lambda a.g\,a$. From the η-rule of Π-types and the transitivity of $\equiv$, we know that $\Gamma \vdash f \equiv g$. Finally we can conclude that $\Gamma \vdash \mathrm{refl}(f) : f = g$. □

In Intensional Type Theory, since propositional equality is not identified with definitional equality, it is not inhabited. If we postulate FUN-EXT, the $\mathbb{N}$-canonicity property by Hofmann (see Definition 2.1.9 in [48]) of Intensional Type Theory is lost, or we can say the theory in no longer *adequate* [3].

Definition 2.3. *A type theory has the* $\mathbb{N}$*-canonicity property if every closed term of* $\mathbb{N}$ *is definitionally equal to a numeral, i.e. either* 0 *or in the form of* $suc(\ldots)$.

Theorem 2.4. *If we introduce functional extensionality into Intensional Type Theory, the* $\mathbb{N}$*-canonicity property is lost.*

Proof. Suppose we define two functions of type $\mathbb{N} \to \mathbb{N}$

$$\mathrm{id} :\equiv \lambda x \to x \text{ and } \mathrm{id}' :\equiv \lambda x \to x + 0$$

where + is defined recursively as

$$\begin{aligned} 0 + n &:\equiv n \\ (\text{suc } m) + n &:\equiv \text{suc } (m + n) \end{aligned}$$

The propositional equality $p : \forall(x : \mathbb{N}) \rightarrow \text{id}(x) = \text{id}'(x)$ is provable by induction on x. By *functional extensionality*, these two functions are propositionally equal

$$\text{ext}(p) : \text{id} = \text{id}'$$

Assume $B : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \textbf{Set}$ which is defined as

$B(f) :\equiv \mathbb{N}$

It is easy to see that 0 is an element for $B(\text{id})$. By applying subst function (see Definition 2.1), we can construct an element of $B(\text{id}')$ as

$$\mathsf{subst}(B, (\text{ext}(p), 0) : B(\text{id}')$$

which is also a term of $\mathbb{N}$ by definition of B. Because the proof $\text{ext}(p)$ is not canonical, namely it can not be reduced to refl, this closed term of natural number is not reduced to either 0 or in the form of $\text{suc}(\ldots)$.

In fact, with this term, we can construct irreducible terms of arbitrary type A by a mapping $f : \mathbb{N} \rightarrow A$. □

- **Uniqueness of Identity Proof (UIP)**

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash x, y : A \qquad \Gamma \vdash p, q : x = y}{\Gamma \vdash \text{uip}(p, q) : p = q} \quad (\text{UIP})$$

UIP is not a consequence of the eliminator for the identity type J as shown in Hofmann and Streicher's groupoid interpretation of Type Theory [51]. It holds if we add another eliminator K introduced by Streicher in [78] as follows:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : a = a \vdash C(x) \qquad \Gamma \vdash t : C(\mathrm{refl}(a)) \qquad \Gamma \vdash p : a = a}{\Gamma \vdash \mathsf{K}(t, p) : C(p)} \quad (\mathsf{K})$$

Computation rule:

$$\mathsf{K}(t, \mathrm{refl}(a)) \equiv t$$

In programming languages such as Agda and Epigram, UIP and K are provable using dependent pattern matching. We can add an Agda flag "–without-K" to deny pattern matching on $a = a$ if we do not accept UIP in general. Although UIP for arbitrary types is not derivable, types equipped with decidable equality have the property UIP as shown by Michael Hedberg [45]. A construction of the proof can be found in [35].

In Homotopy Type Theory, an *h-set* is a type which has UIP e.g. $\mathbb{N}$ (See Section 2.6).

- **Proof irrelevance**

  In traditional Intensional Type Theory, there is no universe of propositions **Prop** which has proof irrelevance:

$$\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \quad (\textsc{proof-irr})$$

  We usually use **Set** instead which does not automatically give us a proof that $(p, q : P) \to p = q$.

  An example of Intensional Type Theory extended with **Prop** is the metatheory of Altenkirch's setoid model (see Section 6.1).

  In Homotopy Type Theory, **Prop** is usually treated as the universe of h-propositions which are types of h-level 1 (see Section 2.6.1). One can think of h-propositions as the sets which have the proof-irrelevance property, hence

$$\mathbf{HProp} = \Sigma(A : \mathbf{Set})\ ((a, b : A) \to a = b)$$

  .

It is different from a universe of propositions because not every set that behaves like a proposition must be in **Prop**, while it is the case for **HProp**.

If we have proof irrelevance, we can simply define identity types for sets as $x = y : \mathbf{Prop}$ and UIP is provable.

- **Propositional extensionality**

  $$\forall P, Q : \mathbf{Prop} \to (P \iff Q) \to (P = Q) \tag{2.3}$$

  Propositional equality between two propositions is given by logical equivalence. Note that this only make senses if there is a universe **Prop**.

- **Quotient types**

  A quotient type is a type formed by redefining its equality by a given equivalence relation on it. It is the main topic of this thesis and is discussed in detail in Chapter 3.

- **Univalence**

  Univalence is an extensional principle from homotopy theory which is an axiom in Homotopy Type Theory. It states:

  Given any two types A, B, the canonical mapping $(A = B) \to (A \simeq B)$ is an equivalence.

  Equivalence can be thought of a refinement of isomorphism in higher categories. The notions of Homotopy Type Theory are discussed in Section 2.6. Propositional extensionality is just the univalence for propositions.

2.4.1 Conservativity of TT_E over TT_I with extensional concepts

In Extensional Type Theory where we accept equality reflection and UIP, many extensional concepts are derivable, for example functional extensionality is derivable from equality reflection with η-rule for Π-types, see Theorem 2.2. Compared to Intensional Type Theory it seems to be more appealing to mathematicians who are more familiar with Set Theory. However type checking is undecidable which has been formally proved by Hofmann in [48]. This makes Intensional Type Theory more favourable, so adding extensional principles into Intensional Type Theory is one of the most important topics in Type Theory. It is preferable if the decidability of type-checking and canonicity are not sacrificed.

The following theorem proved by Hofmann in [49] states that TT_E is conservative over TT_I with functional extensionality and uniqueness of identity proofs added. $\|\_\|$ is an interpretation of TT_I into TT_E and the judgements are differentiated by the subscript of $\vdash$.

Theorem 2.5. *If* $\Gamma \vdash_I A : \mathbf{Set}$ *and* $\|\Gamma\| \vdash_E a : \|A\|$ *for some* a *then there exists* a' *such that* $\Gamma \vdash_I a' : A$

Briefly speaking it is proved by using a model $\mathbf{Q}$ of TT_I, for example categories with families (see Definition 6.2) in the sense of Dybjer which is also a model of TT_E due to the mapping $\|\_\|$ discussed above. The interpretation of the term a in this model gives a term of type A by fullness in TT_I, hence a'. The detailed proof can be found in [49]. In the model $\mathbf{Q}$, types and contexts are propositionally equal if they are isomorphic, which becomes definitional equal in TT_E. The proof is also applied to quotient types which has been shown in [48]. However, the proof is non-constructive i.e. it does not provide an algorithm to compute the term a'.

2.5 An Intensional Type Theory with Prop

Altenkirch has introduced an extension of Intensional Type Theory by a universe of proof-irrelevant propositions and η-rules for Π-types and Σ-types [3]. It is used as a metatheory for his setoid model (see Chapter 6).

The proof-irrelevant universe of propositions **Prop** is a subuniverse of **Set** i.e. $p : \mathbf{Prop}$ implies $p : \mathbf{Set}$. It only contains sets with at most one inhabitant:

$$\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \quad (\textsc{proof-irr})$$

We also introduce $\top, \bot : \mathbf{Prop}$ as basic propositions which are similar to the unit types and empty types, namely we have $tt : \top$, and $\mathrm{abort}(e) : A$ for any type A and any $e : \bot$.

Notice that it is not a definition of types, which means that given a proof that all inhabitants of it are definitionally equal we cannot conclude that a type is of type **Prop**.

The propositional universe is closed under Π-types and Σ-types:

$$\frac{\Gamma \vdash A : \mathbf{Set} \qquad \Gamma, x : A \vdash P : \mathbf{Prop}}{\Gamma \vdash \Pi\ (x : A)\ P : \mathbf{Prop}} \quad (\Pi\text{-}\textsc{Prop})$$

$$\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma, x : P \vdash Q : \mathbf{Prop}}{\Gamma \vdash \Sigma\ (x : P)\ Q : \mathbf{Prop}} \quad (\Sigma\text{-}\textsc{Prop})$$

The metatheory is then proved to be:

- Decidable. The definitional equality is decidable, hence type checking is decidable.
- Consistent. Not all types are inhabited and not all well typed definitional equalities hold.
- ℕ-canonical. All terms of type ℕ are reducible to numerals.

The proof can be found in [3].

2.6 Homotopy Type Theory

Homotopy Type Theory (HoTT) refers to a new interpretation of intensional Martin-Löf type theory into *abstract* homotopy theory. It accepts Vladimir Voevodsky's **univalence axiom** and a new schema to define types called *higher inductive types*, which make many extensional concepts derivable including quotient types.

2.6.1 Homotopical interpretation

Types are usually interpreted as sets in Martin-Löf type theory, but the identity type of types enforces a more sophisticated structure on types compared to the one on sets due to the missing Axiom K that asserts that all inhabitants are equal to the only constructor refl.

Inspired by the groupoid model of (intensional) Martin-Löf type theory due to Hofmann and Streicher, Awodey, Warren [13] and Voevodsky [88] developed Homotopy Type Theory which is a homotopic interpretation of Martin-Löf type theory.

In Homotopy Type Theory, types are regarded as spaces (or higher groupoids) instead of sets, terms are "points" of types. A function $f : A \to B$ is a continuous map between spaces A and B.

- Types are interpreted as spaces. $a : A$ can be viewed as a being a point of space A.
- Terms are continuous functions, for example, $f : A \to B$ is a continuous function between spaces and it is equivalent to say that a is a point of the space or $a : 1 \to A$ is a continuous function.
- Identity types are path spaces.
- Identity types of identity types are homotopies (if a path is considered as a continuous function $p : [0, 1] \to X$).
- Identity types of identity types of identity types and more iterated identity types are 3-homotopies, 4-homotopies etc. They form an infinite structure called ω-groupoids in higher category theory.

Remark 2.6. It has to be emphasised that notions like space are purely homotopical, in other words, there are no topological notions like open sets in Homotopy Type Theory.

2.6.2 Types as weak ω-groupoids

We can also interpret types as **weak ω-groupoids**. The notion of ω-groupoid is a generalisation of groupoid which has infinite levels of "isomorphisms" corresponding to the infinite tower of iterated identity types, i.e. the identity type of identity type, the identity type of identity type of identity type etc.

Formally speaking, a weak ω-groupoid (or weak ∞-groupoid) is a weak ω-category where all k-morphisms between $(k-1)$-morphisms for all $k \in \mathbb{N}$ are equivalences.

An ordinary category only has objects and morphisms. A 2-category includes 2-morphisms between the 1-morphisms and equalities in ordinary category are replaced by explicit arrows. We can continue this generalisation up to n-morphisms between $(n-1)$-morphisms which gives an n-category. An ω-category is an infinite generalisation of this. Objects are also called 0-cells, morphisms between objects are called 1-cells, and morphisms between n-cells are called $(n+1)$-cells.

An equivalence is a morphism which is invertible up to all higher equivalences. The notion of equivalence can be seen as a refinement of isomorphism in a setting without UIP [7]. In the higher-categorical setting, equivalence can be thought of as arising from isomorphisms by systematically replacing equalities by higher cells (morphisms). For example, an equivalence between two objects A and B in a 2-category is a morphism $f : A \to B$ which has a corresponding inverse morphism $g : B \to A$, but instead of the equalities $f \circ g = 1_B$ and $g \circ f = 1_A$ we have 2-cell isomorphisms $f \circ g \cong 1_B$ and $g \circ f \cong 1_A$. In an ω-category, these later isomorphisms are equivalences again. These equivalences are *weak* in the sense that they only hold up to higher equivalences. As all equivalences here are weak equivalences, from now on we just say equivalence.

In fact the ω-groupoids used to model the identity types are also weak, which means that the equalities such as associativity of compositions in the ω-groupoid do not hold strictly. Therefore we should call them **weak ω-groupoids**.

There are several versions of algebraic definitions of weak ω-groupoids (and also weak ω-categories), one of them is the Grothendieck-Maltsiniotis ω-groupoid which has been formalised in [63].

In Homotopy Type Theory the notion of **homotopy n-types** are analogous to n-groupoids in higher category theory. A set can be seen as a discrete space which is a 0-groupoid. Thus a set is called a homotopy 0-type or **h-set** which is of **homotopy level** (or h-level) 2. It is a fact that the identity type of an $(n + 1)$-type is an n-type, for example, the identity type of a groupoid is a set. It can be extended to lower levels: a (-1)-type is a proposition (**mere proposition** or **h-proposition** in Homotopy Type Theory) and a (-2)-type is a contractible type. Because the identity type of a (-2)-type is also a (-2)-type, the hierarchy does not extend further.

2.6.3 Univalence Axiom

Voevodsky recognised that the homotopic interpretation is *univalent* which means isomorphic types are equal, which does not usually hold in Intensional Type Theory. It is one of the fundamental axioms of Homotopy Type Theory and is central to the Voevodsky's proposal of Univalent Foundation Project [87].

For any two types A, B, there is a canonical mapping

$$f : X = Y \to X \simeq Y$$

derived by induction on the identity type. The univalence axiom just claims that this mapping is an equivalence.

It can be viewed as a strong extensionality principle which does imply functional extensionality (a Coq proof of this can be found in [17]). Since isomorphic types are considered the same, all constructions and proofs can be transported between them, and it actually makes reasoning more abstract.

2.6.4 Higher inductive types

In Intensional Type Theory, types are treated as sets and we use *inductive types* to define sets which have only "points". However, in Homotopy Type Theory, due to the enriched structures of types, inductive types can be generalised.

A more general schema to define types including higher paths is required which is higher inductive types (HITs). Higher inductive types allow constructors not only for points of the type being defined, but also for elements of its iterated identity types. One commonly used example is the circle $\mathbb{S}^1$ (1-sphere) which can be *inductively* defined as:

- A point $\mathsf{base} : \mathbb{S}^1$, and
- A path $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$.

It is also essential to provide the elimination rule for the paths as well. Categorically speaking, it means that the functions have to be functorial on paths. That is to say, to define a function $f : \mathbb{S}^1 \to B$, assuming $f(base) = b$, we have to map loop to an identity path $l : b = b$, namely we have an operation $\mathrm{ap}_f : (x =_{\mathbb{S}^1} y) \to (f(x) =_B f(y))$ satisfying $\mathrm{ap}_f(\mathsf{loop}) = l$.

In Homotopy Type Theory, many extensional concepts are derivable. As we have seen, functional and propositional extensionality and are both implied by univalence, UIP for h-sets, proof irrelevance for h-propositions are also available.

Quotient types or more precisely quotient sets (because of the different interpretation of types) are also available. We will discuss them in detail in Section 3.4.

For further explanation of Homotopy Type Theory, a well-written text book elaborated by a group of mathematicians and computer scientists is available online [82]. In this thesis, we refer to it by "*the* HoTT book".

2.6.5 Towards a computational interpretation of HoTT

One of the most important challenges in Homotopy Type Theory is to build a constructive model which would give us a computational interpretation of univalence, so that the good computational properties of Type Theory are preserved [18].

To interpret types as weak ω-groupoids, one main problem is the complexity of its definition. The coherence conditions are very difficult to specify so that people usually choose to use Kan simplicial sets, cubical sets to specify weak ω-groupoids. Nevertheless there are some attempts of encoding weak ω-groupoids in Type Theory. A syntactic approach has been implemented in Agda by the author, Altenkirch and Rypáček (see Chapter 7).

It is much simpler to interpret types as *Kan simplicial sets*. Voevodsky's univalent model [55] is based on Kan simplicial sets. There is a concise introduction written by Streicher [79]. However the simplicial set model is not constructive as Coquand showed that it requires classical logic in an essential way [32]. To avoid the use of classical logic, types can be interpreted as *semi-simplicial sets*. We have not yet implemented the notion of semi-simplicial sets in an Intensional Type Theory like Agda. Some relevant discussion of it can be found online [92].

Recently, Bezem, Coquand and Huber [18] proposed another model of dependent type theory in *cubical sets*. It is expressed in a constructive metalogic which makes it a candidate for obtaining a computational interpretation of univalence. The model seems plausible but some details still need to be verified.

2.7 Summary

The theory of types was originally invented to resolve an inconsistency in set theory in the 1900s. After that, mathematicians developed it by adding more properties, for example functions as primitive types, dependent sum and product types. Type theory is related to type systems in programming languages through the Curry-Howard isomorphisms, and some type theories like the simply-typed lambda calculus, Per Martin Löf's intuitionistic type theory and the calculus of constructions are used as cores of programming languages.

Martin-Löf type theory is one of the most modern type theories which is closely related to constructive mathematics and computer science. It is a formal system given by a sequence of rules written as derivations of judgements. Because of the Curry-Howard isomorphism and dependent types, it is also a system for intuitionistic logic. This means that we can do constructive reasoning by constructing programs. From a mathematician's point of view, this provides computer-aided formal reasoning. From a a programmer's point of view, this provides program verification in itself and a more expressive way to write specifications for programs. Programming languages like Agda, Coq or Epigram exploit these properties.

The intensional version of Martin-Löf type theory has decidable type checking which is essential for a programming language. Agda is a language based on this theory providing numerous features supporting mathematical constructions and reasoning. It is widely used in academia by theoretical computer scientists and mathematicians, for example the Homotopy Type Theory community.

Despite the good properties of Intensional Type Theory, it lacks some extensional concepts like functional extensionality and quotient types. Much research has been done to add them into Type Theory without losing the computational properties. This thesis is one attempt in this direction.

Finally we discussed Homotopy Type Theory where many extensional concepts including quotient types (see Section 3.4) are available. We briefly compared different models of Homotopy Type Theory where types are interpreted as different forms of weak ω-groupoids. However only constructive models can possibly provide computational interpretations of univalence. It is still an open problem to find such a computational interpretation, but a potential solution could be the cubical set model.

Chapter 3

Quotient Types

In this chapter, we present a definition of quotient types in an Intensional Type Theory extended with a proof-irrelevant universe of propositions in the sense of Section 2.5. We prove that, given propositional extensionality, all quotients are effective. We also explain the rules of quotient types categorically. A quotient is essentially a coequalizer or given by an adjunction with equality predicate functor [54]. Quotient types in our definition are essentially quotient *sets*. In Homotopy Type Theory, where types are not interpreted as sets, we discuss Voevodsky's impredicative encoding of quotient sets with all essential rules, and also quotient sets defined using higher inductive types.

3.1 Quotients in Type Theory

3.1.1 Rules for quotients

Quotient types can be defined by the following rules as described in [47, 54].

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A, y : A \vdash x \sim y : \mathbf{Prop} \qquad \sim \text{ is an equivalence}}{\Gamma \vdash A/{\sim}} \; (\text{Q-}\mathbf{Form})$$

Given a type A with a binary equivalence relation $\sim$ on A, we can form the quotient $A/{\sim}$. Here, we use infix notation for readability.

The equivalence properties are

- **Reflexivity** $\mathrm{ref}_{\sim} : \forall (a : A) \to a \sim a$
- **Symmetry** $\mathrm{sym}_{\sim} : \forall (a, b : A) \to a \sim b \to b \sim a$
- **Transitivity** $\mathrm{trn}_{\sim} : \forall (a, b, c : A) \to a \sim b \to\to b \sim c \to a \sim c$

Remark 3.1. Notice that the formation rule is different to Hofmann's version [47] where $\sim$ is not required to be an equivalence relation. In fact his version is just more general which accepts non-equivalence relations $R : A \to A \to \mathbf{Prop}$, but A/R has to be understood as the quotient of A by the equivalence closure of R.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : A/{\sim}} \;(\text{Q-}\mathbf{Intro}) \qquad \frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash p : a \sim b}{\Gamma \vdash \mathrm{Qax}(p) : [a] =_{A/\sim} [b]} \;(\text{Q-}\mathbf{Ax})$$

We introduce an "equivalence class" for each element of A. It is usually denoted as $[a]$, or $[a]_{\sim}$ for $\sim$ if it is unclear which relation it refers to. Qax states that the "equivalence classes" of two terms which are related by $\sim$ are (propositionally) equal.

Notice that the notation of terms $[a]$ should not be confused with notation for substitution such as $B[a]$ or $B[a/x]$. For a Π-type $B : (x : A) \to \mathbf{Set}$ and $a : A$, we therefore write $B(a) : \mathbf{Set}$ for $B[a/x]$ where order of the arguments in brackets matches its definition.

In Hofmann's [47] definition, it comes with an eliminator (also called *lifting*) with a computation rule (β-rule) and an induction principle (equivalent to a η-rule): [1]

$$\frac{\begin{array}{c}\Gamma \vdash B \qquad \Gamma \vdash f : A \to B \\ \Gamma, a : A, b : A, p : a \sim b \vdash f^{\sim}(a, b, p) : f(a) =_B f(b) \qquad \Gamma \vdash q : A/{\sim}\end{array}}{\Gamma \vdash \hat{f}(q) : B} \;(\text{Q-}\mathbf{elim})$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{Qcomp}(a) : \hat{f}([a]) = f(a)} \qquad (\text{Q-}\mathbf{comp})$$

[1] We use shorthand notation ˆ for lifting here

$$\frac{\Gamma, x : A/{\sim} \vdash P : \mathbf{Prop} \qquad \Gamma, a : A \vdash h(a) : P([a]) \qquad \Gamma \vdash q : A/{\sim}}{\Gamma \vdash \mathrm{Qind}(h, q) : P(q)} \ (\text{Q-}\mathbf{ind})$$

Given a function $f : A \to B$ which respects $\sim$, we can lift it to be a function on $A/{\sim}$ as $\hat{f} : A/{\sim} \to B$ such that for any element $a : A$, $\hat{f}([a])$ computes to the same value as $f(a)$. It allows us to define functions on quotient types by functions on base types (representatives). Notice that we omit $f^{=}$ since the computation rule already implies that it is proof-irrelevant.

The induction principle states that for any proposition $P : A/{\sim} \to \mathbf{Prop}$, it is enough to just consider cases $P([a])$ for all $a : A$. In other words, $A/{\sim}$ only consists of "equivalence classes" i.e. $[a]$.

An alternative definition in Hofmann's thesis [48] includes a *dependent* eliminator (dependent lifting) serves the same purpose:

$$\frac{\begin{array}{c}\Gamma, x : A/{\sim} \vdash B \qquad \Gamma \vdash f : (a : A) \to B([a]) \\ \Gamma, a : A, b : A, p : a \sim b \vdash f^{=}(a, b, p) : f(a) \overset{p}{=} f(b) \qquad \Gamma \vdash q : A/{\sim}\end{array}}{\Gamma \vdash \hat{f}(q) : B(q)} \ (\text{Q-}\mathbf{dep\text{-}elim})$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{Qdcomp}(a) : \hat{f}([a]) = f(a)} \ (\text{Q-}\mathbf{dep\text{-}comp})$$

Notice that $\overset{p}{=}$ is an abbreviation for propositional equality which requires substitution in the type of the left hand side by $\mathrm{Qax}(p)$ so that both sides have the same type. We use the same notation for the two versions of eliminators because they are in fact equivalent.

Proposition 3.2. *The* non-dependent eliminator *with the* induction principle *is equivalent to the* dependent eliminator.

Proof. 1. Assume we have the non-dependent eliminator and the induction principle, B is a dependent type on $A/{\sim}$, f is a dependent function of type $(a : A) \to B([a])$ and it respects $\sim$ under substitution (i.e. $f^{=}$), q is an element of $A/{\sim}$.

Set B' as a dependent product $\Sigma(r : A/{\sim})\ B(r)$, Then a non-dependent version of f which has type $A \to B'$ can be defined as

$$f'(a) :\equiv [a], f(a)$$

Given $p : a \sim b$, we can conclude that $f'(a) =_{B'} f'(b)$ is inhabited from Qax and $f^{=}$.

It allows us to lift the non-dependent function f' as $\hat{f}'$ such that

$$\hat{f}'([a]) \equiv [a], f(a) \tag{3.1}$$

Applying first projection on both sides of 3.1, the following propositional equality is inhabited:

$$\pi_1(\hat{f}'([a])) = [a]$$

By induction principle, the predicate $P : A/\!\sim \to \mathbf{Prop}$ defined as

$$P(q) :\equiv \pi_1\ (\hat{f}'(q)) =_{A/\sim}\ q$$

is inhabited for all $q : A/\!\sim$.

Finally, to complete the dependent eliminator, we can construct an element of type $B(q)$ by

$$\pi_2\ (\hat{f}'(q))$$

which has the correct type because $P(q)$ holds. The computation rule is simply derivable from 3.1.

2. It is easy to check that the non-dependent eliminator and induction principle are just special cases of the dependent eliminator.

A formalised version of this proof in Agda can be found in Appendix A. □

Additionally, a quotient is effective (or exact) if an "equivalence class" only contains terms that are related by $\sim$.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad p : [a] =_{A/\sim} [b]}{\text{eff}(p) : a \sim b} \quad (\text{Q-}\textbf{effective})$$

In fact all quotients defined with *equivalence* relations are effective if we have propositional extensionality. This has been proved by Hofmann (See Section 5.1.6.4 in [48]).

Theorem 3.3. *With propositional extensionality, we can prove that all quotient types are effective.*

Proof. Suppose we have a quotient type $A/\!\sim$, two elements $a, b : A$ and $[a] = [b]$

Set a predicate $P_a : A \to \mathbf{Prop}$ as

$$P_a(x) :\equiv a \sim x$$

P_a respects $\sim$ since

$x \sim y$

$\Rightarrow a \sim x \iff a \sim y$ (symmetry and transitivity)

$\equiv P_a(x) \iff P_a(y)$ (propositional extensionality)

$\Rightarrow P_a(x) = P_a(y)$

Therefore we can lift P_a[2] such that for any $x : A$

$$\hat{P}([x]) \equiv a \sim x$$

We can simply deduce $\hat{P}([a]) = \hat{P}([b])$ from assumption $[a] = [b]$ which by definition is just

[2] The elimination rule applies to large types

$$a \sim a = a \sim b$$

Finally, with the eliminator J and $\text{refl}(a) : a \sim a$, we can easily prove

$$a \sim b.$$

□

Similar to other extensional concepts like functional extensionality, simply adding quotient types to Intensional Type Theory as axioms can also result in *non-canonical constructions.*

Theorem 3.4. *If we postulate the rules of quotient types, the* $\mathbb{N}$*-canonicity property is lost.*

Proof. Given a type A and an equivalence relation $\sim$, we postulate $A/\sim$ exists with all the rules above.

Suppose we have two elements $a, b : A$ such that $p : a \sim b$, we have

$$\text{Qax}(p) : [a] = [b]$$

Define $B : A/\sim \to \textbf{Set}$ as

$$B(q) :\equiv \mathbb{N}$$

We can observe that $0 : B([a])$, thus by using subst function (see Definition 2.1), we can obtain a term of $B([b])$:

$$\mathsf{subst}(B, \text{Qax}(p), 0) : B([b])$$

which is also a term of $\mathbb{N}$ by definition of B. This term is irreducible to any numeral because $\mathrm{Qax}(p)$ can not be reduced to the canonical term of the identity type (i.e. refl). Moreover, one can not postulate propositional equality $\mathrm{Qax}(p) = \mathrm{refl}_{[a]}$ or $\mathrm{Qax}(p) = \mathrm{refl}_{[b]}$ because their types are not definitionally equal. □

3.2 Quotients are coequalizers

The rules of quotient types can be characterised in a category-theoretical way.

Categorically speaking, a quotient is a **coequalizer** in the category **Set**. Let us recall the definition.

Definition 3.5. ***Coequalizer***. *Given two objects X and Y and two parallel morphisms $f, g : X \to Y$, a coequalizer is an object Q with a morphism $q : Y \to Q$ such that $q \circ f = q \circ g$ and it is universal: any pair (Q', q') satisfying $q' \circ f = q' \circ g$ has a unique factorisation u such that $q' = u \circ q$:*

$$\begin{tikzcd}
X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & Y \arrow[r, "q"] \arrow[rd, "q'"'] & Q \arrow[d, dotted, "u"] \\
 & & Q'
\end{tikzcd}$$

Now, we show that in **Set**, assuming

$$R :\equiv \Sigma(a_1, a_2 : A)\ a_1 \sim a_2$$

with the two projections being two parallel morphisms $\pi_1, \pi_2 : R \to A$, a quotient corresponds to the coequalizer $(A/\!\sim, [\_])$:

$$\begin{tikzcd}
R \arrow[r, shift left, "\pi_1"] \arrow[r, shift right, "\pi_2"'] & A \arrow[r, "{[\_]}"] \arrow[dr, "f"'] & A/\!\sim \arrow[d, dotted, "\hat{f}"] \\
 & & B
\end{tikzcd}$$

The factorisation $\hat{\_}$ is just the eliminator, the computation rule and induction principle correspond to the universal property of it.

Proposition 3.6. *The induction principle implies uniqueness, and is also derivable from the definition of coequalizer.*

Proof. It is easy to see that induction principle implies the uniqueness of $\hat{f}$:

Given any $g : A/\!\sim \to B$ fulfils the same property as $\hat{f}$, applying induction principle on

$$\forall (a : A) \to g([a]) = \hat{f}([a])$$

we can deduce that

$$\forall (q : A/\!\sim) \to g(q) = \hat{f}(q)$$

hence $g = \hat{f}$.

The other way is more difficult:

Given $P : A \to \textbf{Prop}$, $h : (x : A) \to P(x)$ define

$$P' :\equiv \Sigma(x : A)\ P(x) \text{ and } h'(x) = ([x], h(x))$$

we can observe that

$$\pi_1 \circ h' = [\_] \tag{3.2}$$

By the universal property, there is a unique $\hat{h'}$ s.t.

$$\hat{h'} \circ [\_] = h' \tag{3.3}$$

By replacing 3.3 in 3.2

$$\pi_1 \circ \hat{h'} \circ [\_] = [\_] \tag{3.4}$$

From *uniqueness* we can easily prove that $[\_]$ is an *epimorphism.*

Thus from 3.4, we prove that

$$\pi_1 \circ \hat{h'} = id$$

which implies that for any $q : A/{\sim}$, the type of $\pi_2(\hat{h'}(q))$ is

$P(\pi_1(\hat{h'}(q))) = P(q)$

as expected, hence we derive the induction principle. In fact, following the same procedure, the dependent eliminator is also derivable. □

The coequalizer (quotient) is effective if the following diagram is a *pullback*

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\pi_2"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "{[\_]}"] \\
A \arrow[r, "{[\_]}"'] & Q
\end{tikzcd}$$

Proof. Assume we have two points $a, b : \mathbf{1} \to A$ satisfying $[a] = [b]$.

From the pullback property, there is a unique point $r : \mathbf{1} \to R$ such that

$$\pi_1(r) = a$$

and

$$\pi_2(r) = b$$

Hence (a, b) is an element of R, by definition it means

$$a \sim b$$

□

In Chapter 4, we also introduce two other notions: prequotient and definable quotient.

Cateogorically speaking, a *prequotient* is just a *fork* which is just a morphism $[\_]$ such that the following diagram commutes:

$$\begin{tikzcd} R \arrow[r, shift left, "\pi_0"] \arrow[r, shift right, "\pi_1"'] & A \arrow[r, "{[\_]}"] & Q \end{tikzcd}$$

and a *definable quotient* corresponds to a *split coequalizer* which is a *fork* with two morphisms $\mathrm{emb} : Q \to A$ and $t : A \to R$ such that emb chooses a representative in every equivalence class:

- $[\_] \circ \mathrm{emb} = 1_Q$
- $\mathrm{emb} \circ [\_] = \pi_0 \circ t$ and
- $\pi_1 \circ t = 1_A$

Further, we can deduce that $t(a) = (\mathrm{emb}[a], a)$, which gives the proof that each element is related to the representative of its class, namely the "completeness" property of definable quotients.

3.3 Quotients as an adjunction

As Jacobs [54] suggests, quotients can be described as a left adjoint to an equality functor.

Let us recall the definition first.

Definition 3.7. ***Adjunction***. *Given two categories A and B, a functor $F : A \to B$ is left adjoint to $G : B \to A$ if we have a natural isomorphism $\Phi : hom_B(F\ \_, \_) \to hom_A(\_, G\ \_)$*

Given the category of setoids **Setoid** and category of sets **Set**, there is an equality functor $\nabla : \mathbf{Set} \to \mathbf{Setoid}$ defined as

$$\nabla A :\equiv (A, =_A)$$

where the morphism part is trivial embedding.

Quotients can be seen as a functor $\mathbf{Q} : \mathbf{Setoid} \to \mathbf{Set}$ which is left-adjoint to a equality functor $\nabla A :\equiv (A, =_A)$

The object part of this functor corresponds to the formation rule of quotients, hence we can use $B/\!\sim$ to represent $\mathbf{Q}\ (B, \sim)$.

The adjunction can be described by a natural isomorphism

$\Phi : \mathrm{hom}_{\mathrm{Set}}(\mathbf{Q}\ \_, \_) \to \mathrm{hom}_{\mathrm{Setoid}}(\_, \nabla\ \_)$

or a diagram for each $(Y, \sim) : \mathbf{Setoid}$ and $X : \mathbf{Set}$:

$$\frac{Y/\!\sim\ \to X}{\overline{(Y, \sim) \to (X, =_X)}}$$

which consists of $\Phi_{(Y,\sim),X}$ and its inverse $\Phi^{-1}_{(Y,\sim),X}$ (the subscripts are omitted later).

Given an identity morphism $\mathrm{id} : A/\!\sim\ \to A/\!\sim$,

$$\Phi(\mathrm{id}) : (A, \sim) \to (A/\!\sim, =_{A/\sim})$$

is just the introduction rule $[\_] : A \to A/\!\sim$ with the property that it respects $\sim$. It is also called *unit* written as $\eta_{(A,\sim)}$.

Given a morphism $f : (A, \sim) \to (B, =_B)$ which is a function that respects $\sim$,

$$\Phi^{-1}(f) : A/{\sim} \to B$$

which corresponds to the elimination rule.

The computation rule $\hat{f} \circ [\_] \equiv f$ corresponds to the following digram in the category of setoids:

$$\begin{tikzcd}
(A, \sim) \arrow[d, "\eta_{(A,\sim)}"'] \arrow[dr, "f"] & \\
(A/{\sim}, =_{A/\sim}) \arrow[r, "\nabla(\Phi^{-1}(f))"'] & (B, =_B)
\end{tikzcd}$$

which is commutative because

$\nabla(\Phi^{-1}(f)) \circ \eta_{(A,\sim)}$

$= \Phi(\Phi^{-1}(f))$ by adjunction law $G(f) \circ \eta_Y = \Phi(f)$

$= f$

We can also recover the adjunction from the definition of quotients. Define

$$\mathbf{Q}\ (Y, \sim) :\equiv Y/{\sim}$$

The adjunction is given by

$$\Phi(f) :\equiv f \circ [\_] \text{ and } \Phi^{-1}(g, g^{\sim}) :\equiv \hat{g}$$

The computation rule and induction principle just express that these two mapping are each other inverses.

3.4 Quotients in Homotopy Type Theory

As we mentioned before, quotient types (in the sense of 3.1.1) are available in Homotopy Type Theory. Because of the different interpretations of types, it makes less confusion to call them *quotients* or *set quotients* here.

First, let us recall that

- an h-proposition (hProp) is a type A which has the property $\forall (a, b : A) \to a =_A b$, and
- an hSet is a type S such that forall $x, y : S$, $x =_S y$ are h-propositions.

For simplicity, we use the term "set" for h-sets and "proposition" for h-propositions. Note that **Prop** is *not* the built-in universe of propositions in Coq, but the internally defined universe of h-propositions.

3.4.1 An impredicative encoding of quotient sets

Vladimir Voevodsky introduced an impredicative definition of quotients which was formalised in Coq [86].

Assume we have a set A and an equivalence relation $\sim : A \to A \to \mathbf{Prop}$.

Definition 3.8. *An* ***equivalence class*** *is a predicate* $P : A \to \mathbf{Prop}$ *such that*

it is inhabited: $\exists (a : A)\ P(a)$,

and for all $x, y : A$,

$P(x) \to P(y) \to x \sim y$ *and*

$P(x) \to x \sim y \to P(y)$.

These properties can be encoded as

$$\mathit{EqClass}(P) :\equiv (\exists (a : A)\ P(a)) \wedge (\forall (x, y : A) \to P(x) \to (x \sim y \iff P(y))).$$

Definition 3.9. *We define the* ***set quotient*** *as*

$$A/{\sim} :\equiv \Sigma(P : A \to \mathbf{Prop})\ \mathit{EqClass}(P)$$

$A/{\sim}$ is a set because $A \to \mathbf{Prop}$ is a set and $\mathrm{EqClass}(P)$ is a proposition. $\wedge$ is the non-dependent Σ-type for propositions and $\forall$ is the Π-type for propositions. Because it is in fact a triple, we use $(P, p, q) : A/{\sim}$ to represent an element of it for convenience, where P is the predicate, p is the truncated witness that P is inhabited, and q contains the proofs of the logical equivalence.

The encoding of $\exists(a : A)\ P(a)$ is given by a truncated Σ-type: $\|\Sigma(a : A)\ P(a)\|$. The (-1)-truncation $\| - \|$ is defined *impredicatively* as

$$\|X\| :\equiv \forall(P : \mathbf{Prop}) \to (X \to P) \to P$$

with a trivial embedding function $|\_| : X \to \|X\|$:

$$|x| :\equiv \lambda P\ f \to f(x)$$

We can simply recover the elimination rule for truncation: given any function $f : X \to P$ where P is a proposition, we can define a function of type $\|X\| \to P$ as

$$\tilde{f}(x) :\equiv x(P, f)$$

and $\tilde{f}(|x|) \equiv f(x)$ automatically holds.

Remark 3.10. Note that $\|X\|$ is in the universe of $\mathbf{Set}_1$, but with **resizing rules** proposed by Voevodsky [90, 91], $\|X\|$ is moved to the universe $\mathbf{Set}$. We can apply the resizing rule for propositions because $\|X\|$ behaves like a proposition. It also has to be noticed that it is impossible to extract an element of A from a proof of $\mathrm{EqClass}(P)$ because of the truncation.

There is a canonical function $[\_] : A \to A/{\sim}$ corresponding to the **introduction rule**:

$$[a] :\equiv (\lambda x \to a \sim x, |a, \mathrm{ref}(a)|, \lambda x\ y\ p \to (\lambda q \to \mathrm{trn}(p, q), \lambda q \to \mathrm{trn}(\mathrm{sym}(p), q)))$$

which respects $\sim$. The verification of compatibility requires propositional extensionality and functional extensionality which are available in Homotopy Type Theory. In fact, we can prove that $[a]$ is a unique representation of an equivalence class.

Lemma 3.11. *Given any $(P, p, q) : A/\!\sim$, it is the unique representation of an equivalence class, namely*

$$\forall (a : A) \to P(a) \to [a] =_{A/\sim} (P, p, q)$$

is inhabited.

Proof. Because $A/\!\sim$ is a Σ-type whose second component $\mathrm{EqClass}(P)$ is a proposition depends on the first component, if the first components are equal, i.e.

$$\lambda b \to a \sim b = P$$

then their second components are also equal because of proof-irrelevance.

By functional extensionality, we only need to prove that

$$\forall (b : A) \to a \sim b = P(b) \tag{3.5}$$

Recall that the type of q is $\forall (x, y : A) \to P(x) \to (x \sim y \iff (P(y)))$, from assumption $ex : P(a)$, we can prove that

$$\forall (b : A) \to a \sim b \iff P(b)$$

Then we can simply prove 3.5 by applying propositional extensionality. Therefore

$$[a] = (P, p, q)$$

□

A lifting function (non-dependent eliminator) for functions respecting $\sim$ is also expected. Since we cannot extract a element of A, it has to be defined in a more complicated way.

Lemma 3.12. *Given a function $f : A \to B$ into a* set *B which respects $\sim$, there exists a unique function $\hat{f} = A/\!\sim \to B$ such that $\hat{f}([a]) \equiv f(a)$.*

Proof. Assuming we have an element $(P, p, q) : A/\!\sim$, we can define a function $f_P : (\Sigma(x : A)\ P(x)) \to B$ simply by

$$f_P :\equiv f \circ \pi_1$$

but our witness $p : \|\Sigma(x : A)\ P(x)\|$ is truncated which cannot be applied to f_P. However we can generate a function

$$\bar{f_P} : \|(\Sigma(x : A)\ P(x))\| \to B$$

applying lemma 3.13, which needs that f_P is a constant function:

for any two elements (x_1, p_1) and (x_2, p_2) of type $\Sigma(x : A)\ P(x)$, by applying the property

$$\forall(x, y : A) \to P(x) \to P(y) \to x \sim y$$

contained in q to $p_1 : P(x_1)$ and $p_2 : P(x_2)$, we have that

$$x_1 \sim x_2.$$

Then because f respects $\sim$,

$$f(x_1) = f(x_2).$$

By definition of f_P,

$$f_P(x_1, p_1) \equiv f(x_1) = f(x_2) \equiv f_P(x_2, p_2),$$

hence f_P is a constant function.

To summarise, the lifting function can be defined as

$$\hat{f}(P, p, q) :\equiv \bar{f}_P(p)$$

The **computational rule** can be verified easily:

$$\hat{f}([a]) \equiv \bar{f}_{\lambda x \to a \sim x}(|a|) \equiv f(a)$$

The **induction principle** can be generated as follows:

suppose we have $Q : A/\sim \to \mathbf{Prop}$, $h : (a : A) \to Q([a])$ and $(P, p, q) : A/\sim$, we expect the *proposition* $Q(P, p, q)$ to hold. Since $p : \|\Sigma(a : A)\ P(a)\|$, from the elimination rule for truncation, we only need to construct a function of type

$$\Sigma(a : A)\ P(a) \to Q(P, p, q).$$

Given $(a, ex) : \Sigma(a : A)\ P(a)$, we know

$$[a] = (P, p, q)$$

from 3.11. Thus we can substitute into $h(a) : Q([a])$ to generate a term of type $Q(P, p, q)$. Therefore we have the induction principle. The uniqueness of $\hat{f}$ is simply implied by the induction principle. □

The following lemma is suggested by Nicolai Kraus and can be found in [58].

Lemma 3.13. *Given a constant function $g : X \to Y$ where Y is a set, i.e. it satisfies*

$$\forall (x, y : X) \to g(x) = g(y),$$

there exists a function $\bar{g} : \|X\| \to Y$ such that $\bar{g}(|x|) \equiv g(x)$.

Proof. Define the subset

$$Y' :\equiv \Sigma(y : Y)\ \|\Sigma(x : X)\ g(x) = y\|$$

Intuitively, Y' only contains the image of the constant function i.e. Y' **is propositional**:

For any $(y_1, p_1) : Y'$ and $(y_2, p_2) : Y'$,

we can first generate the proofs

$p_1((g(x) = y_1), \pi_2) : g(x) = y_1$ and

$p_2((g(x) = y_2), \pi_2) : g(x) = y_2$.

By symmetry and transitivity we can prove that $y_1 = y_2$.

From the fact that a truncated type is always propositional, we can also deduce that $p_1 = p_2$, then $(y_1, p_1) = (y_2, p_2)$. Hence we can conclude that Y' is propositional.

We can simply define a function $g' : X \to Y'$ using g as

$$g'(x) :\equiv (g(x), \lambda Q\ f \to f(x, \mathrm{refl}_=(g(x)))).$$

Because Y' is propositional, it is possible to lift g' to a function $\tilde{g'} : \|X\| \to Y'$ which is defined as

$$\tilde{g'}(x) :\equiv x(Y', g').$$

Finally we define

$$\bar{g} :\equiv \pi_1 \circ \tilde{g'}$$

which fulfils the computation rule

$$\bar{g}(|x|) \equiv \pi_1(|x|(Y', g')) \equiv \pi_1(g'(x)) \equiv g(x).$$

□

Furthermore, since propositional extensionality is a special case of univalence, by Theorem 3.3, we can prove that the impredicative quotients are effective.

Theorem 3.14. *In Homotopy Type Theory, the impredicative encoding of quotient sets gives rise to all the rules of quotients in the sense of 3.1.1 including effectiveness.*

3.4.2 Quotient inductive types

An alternative way to define quotients in Homotopy Type Theory is using higher inductive types.

Assume that A is a set and $\_ \sim \_ : A \to A \to \mathbf{Prop}$ is an equivalence relation. To build a quotient, we can simply impose level-1 morphisms in the structure of the given set according to the equivalence relation. Thus, a quotient $A/\sim$ can be defined as a higher inductive type with the following contstructors:

- $[\_] : A \to A/\sim$
- $\mathit{eqv} : (a, b : A) \to a \sim b \to [a] = [b]$
- $\mathit{isSet} : (x, y : A/\sim\,) \to (p_1, p_2 : x = y) \to p_1 = p_2$

It is also a set so we call it **set-quotient** or **quotient inductive types** (QITs).

Some examples suggest that QITs are more powerful than quotient types.

One of the examples is the definition of real numbers $\mathbb{R}$ which will be discussed in Chapter 5. Briefly speaking, our construction of reals by Cauchy sequences of rational numbers is not Cauchy complete because not all equivalence classes have a limit. However, the Cauchy approximation approach (see Subsection 11.3.1 in [82]) using quotient inductive types is Cauchy complete due to the fact that the equivalence relation and limits are included in its definition.

Another example is unordered trees (rooted trees) which are trees connected to a multiset of subtrees, hence there is no ordering on subtrees.

First we define ordered trees as:

- a leaf $l : \mathsf{Tree}$, or

- an indexed family of subtrees indexed by a set I, $st : (I \to \mathsf{Tree}) \to \mathsf{Tree}$

with the following equivalence relation:

- $l_{eq} : l \sim l$,
- $st_{eq} : (f, g : I \to \mathsf{Tree}) \to f \sim_p g \to st(f) \sim st(g)$,

where $f \sim_p g$ stands for f is a permutation of g. The permutation can be defined using a bijective map $p : I \to I$ which relates equivalent subtrees recursively.

If we define unordered trees as a quotient type $\mathsf{Tree}^{\sim} := \mathsf{Tree}/\!\sim$, it is problematic to lift the constructor st, i.e. to define $\hat{st}$. For trees with finite subtrees such as *binary trees* where $I :\equiv \mathbf{2}$, it can be lifted by nesting lifting functions,

$$\hat{st}(a, b) = \widehat{\hat{st}(a)}(b)$$

because its type is isomorphic to $\mathsf{BTree} \to \mathsf{BTree} \to \mathsf{BTree}$. Intuitively this approach can be applied to trees with finite subtrees. However it fails if have infinite subtrees, for example when $I :\equiv \mathbb{N}$.

However if we use QITs to define unordered trees, we can define the equivalence relation simultaneously with the constructors by the higher inductive type having the following constructors:

- $l : \mathsf{Tree}$,
- $st : (I \to \mathsf{Tree}) \to \mathsf{Tree}$,

  and a set of paths relating two permuted trees:
- $l_{eq} : l =_{\mathsf{Tree}} l$,
- $st_{eq} : \forall (f, g : I \to \mathsf{Tree}) \to f \sim_p g \to st(f) =_{\mathsf{Tree}} st(g)$.

Thus we avoid the problem of lifting st because the equivalence relation has become the internal equality of this type.

Similarly the cumulative hierarchy of all sets introduced in [82] (see section 10.5) suggests that quotient types have some weaknesses compared to quotient inductive types.

A cumulative hierarchy can be given by constructors

$$\{\_\} : (I : \mathbf{Set}) \to (I \to M_0) \to M_0$$

along with a subset relation

$$\_ \in \_ : M_0 \to M_0 \to \mathbf{Prop}$$

which is inhabited if $f(i) \in \{I, f\}$.

Then we can easily define the equivalence relation on "sets" using the set-theoretical definition $A \sim B :\equiv \forall m : M_0, m \in A \iff m \in B$.

Similarly to unordered trees, we cannot obtain the constructor $\widehat{\{\_\}}$ because the indexing set I can be infinite.

To summarise, it seems that quotient inductive types are more powerful than quotient types due to the ability of defining term constructors and equivalence relations simultaneously. However, quotient inductive types are not available in type theories other than Homotopy Type Theory and the computational interpretation of them is still an open problem. Moreover, there can be more general quotients in Homotopy Type Theory, for example a quotient of a type by a 1-groupoid (See section 9.9 in [82]). It is interesting to investigate *real quotient types* in Homotopy Type Theory, but it is beyond the scope of this thesis.

3.5 Related work

The introduction of quotient types in Type Theory has been studied by several authors in different versions of Martin-Löf type theory and using various approaches.

- In [29], Mendler et al. considered building new types from a given type using a quotient operator $//$. Their work is done in an implementation of Extensional Type Theory, NuPRL.

  In NuPRL, given the base type A and an equivalence relation E, the quotient is denoted as $A//E$. Since every type comes with its own equality relation in NuPRL, the quotient operator can be seen as a way of redefining equality for a type.

  They also discuss problems that arise from defining functions on the new type which can be illustrated by a simple example:

  when we want to define a function $f : (x, y) : A//E \to \mathbf{2}$, it is in fact defining a function on A. Assume $a, b : A$ such that $E(a, b)$ but $f(a) \neq f(b)$. This will lead to an inconsistency since $E(a, b)$ implies that a converts to b in Extensional Type Theory, hence the left hand side $f(a)$ can be converted to $f(b)$, namely we get $f(b) \neq f(b)$ which contradicts the equality reflection rule.

  Therefore a function is well-defined [29] on the new type only if it respects the equivalence relation E, namely

$$\forall (a, b : A) \to E(a, b) \to f(a) = f(b)$$

  After the introduction of quotient types, Mendler further investigates this topic from a categorical perspective in [70]. He uses the correspondence between quotient types in Martin-Löf type theory and coequalizers in a category of types to define a notion called *squash types*, which is further discussed by Nogin [71].

- Nogin [71] considers a modular approach to axiomatizing quotient types in NuPRL. He discusses some problems with quotient types. For example, since equality is extensional, we cannot recover the witness of equality. He suggests including more axioms to conceptualise quotients. He decomposes the formalisation of a quotient type into several smaller primitives which are easier to manipulate.

- Jacobs [54] introduces a syntax for quotient types based on predicate logic within simple type theory. He discusses quotient types from a categorical perspective. In fact the syntax of quotient types arises from an adjunction as we mentioned before.

- To add quotient types to Martin-Löf type theory, Hofmann proposes three models for quotient types in [48]. The first one is a setoid model for quotient types. In this model all types are attached with partial equivalence relations, namely all types are partial setoids rather than sets. It does not provide dependency at the level of types but only at the level of the relations. The second one is the groupoid model which supports most features required but it is not definable in Intensional Type Theory. He also proposes a third model as an attempt to overcome problems in the previous two models. More type dependency is provided and quotient types are believed to be definable in this model, however it also has some disadvantages. He also shows that Extensional Type Theory is conservative over Intensional Type Theory extended with quotient types [49].

- Altenkirch [3] also provides a different setoid model which is built in an Intensional Type Theory extended with a proof-irrelevant universe of propositions and η-rules for Π-types and Σ-types. It is decidable, $\mathbb{N}$-canonical and permits large eliminations. We implemented this setoid model and interpreted quotient types in it (see Chapter 6).

- Homeier [52] axiomatises quotient types in Higher Order Logic (HOL) which is also a theorem prover. He creates a tool package to construct quotient types as a conservative extension of HOL so that users are able to define new types in HOL. Then he defines the normalisation functions and proves several properties of them. Finally he discusses the issues arising when quotienting on aggregate types such as lists and pairs.

- Courtieu [34] extends of the Calculus of Inductive Constructions with *Normalised Types* which are similar to quotient types, but equivalence relations are replaced by normalisation functions which select a canonical element for each equivalence class. In fact normalised types can be seen as a proper subset of quotient types. We can easily recover a quotient type from a normalised type as below

$$a \sim b :\equiv [a] = [b]$$

However not all quotient types have normal forms, for example, the set of real numbers (see Chapter 5). The notion *definable quotients* we proposed in Chapter 4 is also similar to it, but does not provide a new type automatically.

- Barthe and Geuvers [15] propose a new notion called *congruence types*, which is also a special class of quotient types in which the base type is inductively defined and comes with a set of reduction rules called the term-rewriting system. The idea is that β-equivalence is replaced by a set of β-conversion rules. Congruence types can be treated as an alternative to pattern matching introduced in [31]. The main purpose of introducing congruence types is to solve problems in term rewriting systems rather than to implement quotient types. Congruence types are not inductive but have good computational behaviour because we can use the term-rewriting system to link a term of the base type with a unique term of the congruence type which is its normal form. However this approach has some problems in termination criteria and interaction between rewriting systems [34].

- Barthe, Capretta and Pons [16] compare different ways of defining setoids in Type Theory. Setoids are classified as partial setoids or total setoids depending on whether the equality relation is reflexive or not. They also consider obtaining quotients for different kinds of setoids, especially for partial setoids. In their framework of partial setoids, suppose we have a partial setoid $(A, \sim)$, an element $x : A$ such that $x \sim x$ is called a *defined* element, the others are undefined. In this case if we simply define a quotient by replacing the underlying partial equivalence relation with a new one R, undefined elements in the base setoid may be incorrectly introduced in the quotient. The reason is that there possibly exist some undefined elements $x : A$ satisfying $R(x, y)$. They solve the problem by defining a restricted version of R which only relates defined elements.

- Abbott, Altenkirch et al. [2] provides the basis for programming with quotient datatypes polymorphically based on their works on containers which are datatypes whose instances are collections of objects, such as arrays, trees and so on. Generalising the notion of container, they define quotient containers as the containers quotiented by a collection of isomorphisms on the positions within the containers.

- Voevodsky [86] implements quotients in Coq based on a set of axioms of Homotopy Type Theory. He first implements the notion of equivalence class and uses it to implement quotients which is analogous to the construction of quotient sets in set theory. The details are given in Section 3.4.1.

3.6 Summary

We gave the syntax of quotient types in this chapter. The underlying relation is required to be an equivalence in our definition which is different from [47]. In fact, the equivalence condition does not affect the construction of quotient types. Jacobs [54] has shown that, for an arbitrary relation R, the same constructions can be interpreted as set theoretical quotient sets of $A/R^{\equiv}$, where $R^{\equiv}$ is the equivalence closure of R.

Two approaches of defining elimination rules were given, one having a combination of non-dependent eliminator with an induction principle as in Hofmann's definition and another having a dependent eliminator. We also showed that they are equivalent.

We showed that propositional extensionality implies the effectiveness of quotients. We characterised quotients in category theory. They do not only correspond to coequalizers but also can be generated from a left adjoint functor to the equality functor $\nabla : \mathbf{Set} \to \mathbf{Setoid}$. We concluded with a literature review about quotient types.

Chapter 4

Definable Quotients

In Intensional Type Theory, the quotient type former is not necessary to define all quotients as sets. One of the most basic examples is the set of integers $\mathbb{Z}$. On one hand it can be interpreted as a quotient set $\mathbb{Z}_0 :\equiv \mathbb{N} \times \mathbb{N}/\sim$ in which we use a pair of natural numbers (a, b) to represent the integer as the result of subtraction $a - b$. On the other hand, from the usual notation of integers, $\mathbb{Z}$ can be inductively defined as natural numbers together with a sign. Given any element $(a, b) : \mathbb{N} \times \mathbb{N}$, there must be an element of $c : \mathbb{Z}$ which can be seen as the name of the equivalence class or **normal form** of (a, b), thereby we can define a **normalisation function** denoted as $[\_] : \mathbb{Z}_0 \to \mathbb{Z}$.

Another example is the set of rational numbers $\mathbb{Q}$. Usually, rational numbers are represented as fractions, e.g. $\frac{1}{2}$. However different fractions can refer to the same rational numbers, e.g. $\frac{1}{2} = \frac{2}{4}$. It naturally gives us a quotient definition of rational numbers as *fractions* (or unreduced fractions). As we know, for one rational number, different fractions for it can always be reduced to a unique one called *reduced fraction*. Therefore, the set $\mathbb{Q}$ can also be defined as a Σ-type consisting of a fraction together with a proof of the property that it is reduced. Thus, a normalisation function in this case is just an implementation of the reduction process.

For these quotients which are definable as a set without being treated as quotients, it seems unnecessary to interpret them as setoids. However in practice, the setoid definitions have some advantages compared to the set definition. For example, we can define operations on $\mathbb{Z}$ like addition and multiplication and prove algebraic properties, such as verifying that the structure is a ring. However, this is quite complicated and uses many unnecessary case distinctions due to the cases in the set definition. E.g. the proving of distributivity within this setting is not satisfactory since too many cases have to be proven from scratch. In the setoid definition $\mathbb{Z}_0$, there is only one case and the algebraic properties are direct consequences of the semiring structure of the natural numbers. For rational numbers, it is also conceivable that operations on unreduced functions are simpler to define because there is no need to make sure the result is reduced in every step.

Although the setoid definitions have some nice features in these cases, they require us to redefine all operations on sets again on setoids, for example $\mathrm{List}(A, \sim)$. Hence, we propose to use both the setoid and the associated set, but to use the setoid structure to define operations on the quotient set and to reason about it. The setoid definition and set definition can be related by the normalisation function so that we can lift operations and properties in the same manner as quotient types.

In this chapter we introduce the formal framework to do this, i.e. we provide the definition of quotients as algebraic structures specifying the normalisation function with necessary properties. Indeed, it can be seen as a “manual construction” of quotient types, in other words, instead of automatically creating a type given a setoid, we prove another given type *is* the quotient. It provides us with conversions between two representations and so combines the nice features of both representations.

4.1 Algebraic structures of quotients

We first define several algebraic structures for quotients corresponding to the rules of quotient types (see Section 3.1.1).

Definition 4.1. ***Prequotient****.* *Given a setoid* $(A, \sim)$*, a* prequotient *over that setoid consists of*

1. *a set* Q*,*

2. *a function* $[\_] : A \to Q$*,*

3. *a proof* sound *that the function* $[\_]$ *respects the relation* $\sim$*, that is*

$$\text{sound}\colon (a, b : A) \to a \sim b \to [a] = [b],$$

Roughly speaking, 1 corresponds to the formation rule, 2 corresponds to the introduction rule and 3 corresponds to Q-**Ax**. The function $[\_]$ is intended to be the *normalisation* function with respect to the equivalence relation, however it is not enough to determine it now.

To complete a *quotient*, we also need the elimination rule and the computation rule.

Definition 4.2. ***Quotient***. *A prequotient* $(Q, [\_], \text{sound})$ *is a quotient if we also have*

4. *for any* $B : Q \to \mathbf{Set}$*, an eliminator*

$$\begin{aligned} \text{qelim}_B \ &: \ (f\colon (a : A) \to B\,[a]) \\ &\to ((p : a \sim b) \to f(a) \simeq_{\text{sound}(p)} f(b)) \\ &\to ((q : Q) \to B(q)) \end{aligned}$$

such that $\text{qelim-}\beta : \text{qelim}_B(f, p, [a]) = f(a)$*.*

This definition has a dependent eliminator. An alternative equivalent definition given by Martin Hofmann has a **non-dependent** eliminator and an induction principle.

Definition 4.3. ***Quotient (Hofmann's)***. *A prequotient* $(Q, [\_], \text{sound})$ *is a quotient (Hofmann's) if we also have*

$$\text{lift} : (f : A \to B) \to (\forall a, b \to a \sim b \to f(a) = f(b)) \to (Q \to B)$$

together with an induction principle. Suppose B *is a predicate, i.e.* $B : Q \to \mathbf{Prop}$*,*

$$\text{qind}\colon ((a : A) \to B([a])) \to ((q : Q) \to B(q))$$

Definition 4.4. ***Effective quotient.*** *A quotient is* effective *(or exact) if we have the property that*

$$\mathit{effective} : (\forall a, b : A) \to [a] = [b] \to a \sim b$$

We now consider a specific group of quotients which have a canonical choice in each equivalence class.

Definition 4.5. ***Definable quotient.***

Given a setoid $(A, \sim)$*, a* definable quotient *is a prequotient* $(Q, [\_], \mathrm{sound})$ *with*

$$\begin{aligned} \mathrm{emb} &: Q \to A \\ \mathrm{complete} &: (a : A) \to \mathrm{emb}[a] \sim a \\ \mathrm{stable} &: (q : Q) \to [\mathrm{emb}(q)] = q. \end{aligned}$$

It is exactly the specification of $[\_]$ as a normalisation function with respect to emb (see [5]). It is also related to the choice operator for quotient types in Martin Hofmann's definition[47].

Proposition 4.6. *All definable quotients are effective quotients.*

Proof. Assume B : **Set**, given any function $f : A \to B$ such that $p : a \sim b \to f(a) = f(b)$, define

$$\mathrm{lift}_B(f, p, q) :\equiv f(\mathrm{emb}(q))$$

To verify the computation rule, assume $a : A$,

$$\mathrm{lift}_B(f, p, [a]) \equiv f(\mathrm{emb}([a]))$$

By completeness, we get

$$\text{emb}([a]) \sim a$$

Then by $p : a \sim b \to f(a) = f(b)$, we can prove that

$$f(\text{emb}([a])) = f(a)$$

For induction principle, suppose $B : Q \to \mathbf{Prop}$, let $f : (a : A) \to B([a])$ and $q : Q$.

By stabiliy, we get

$$[\text{emb}(q)] = q$$

Thereby from

$$f(\text{emb}(q)) : B([\text{emb}(q)])$$

we can derive a proof of $B(q)$.

It follows from Proposition 3.2 that this also gives rise to a quotient.

Finally, assume $[a] = [b]$ for given $a, b : A$, by completeness property, we obtain that

$$a \sim \text{emb}[a] = \text{emb}[b] \sim b$$

and hence $a \sim b$, i.e. the quotient is effective.

□

However, a quotient is not enough to build a definable quotient because we can not extract a canonical choice for each equivalence class $q : Q$.

The definitions of these algebraic structures and proofs about the relations between them have been encoded in Agda (see Appendix A).

Let us investigate some examples of definable quotients.

4.2 Integers

4.2.1 The setoid definition $(\mathbb{Z}_0, \sim)$

Negative whole numbers can be understood as the results of subtraction of a larger natural number from a smaller one. In fact, any integer can be seen as a result of subtraction of a natural number from another. It implies that integers can be represented by pairs of natural numbers

$$\mathbb{Z}_0 :\equiv \mathbb{N} \times \mathbb{N}$$

for example, from the equation $1 - 4 = -3$, we learn that -3 can be represented by $(1, 4)$.

However, from the equation

$$n_1 - n_2 = n_3 - n_4,$$

it is easy to see that one integer can be represented by different pairs.

The equivalence relation can not be simply defined as this because subtraction is not closed on natural numbers. We only need to transform the equation as

$$n_1 + n_4 = n_3 + n_2.$$

This gives rise to an equivalence relation:

$$(n_1, n_2) \sim (n_3, n_4) :\equiv n_1 + n_4 = n_3 + n_2.$$

We can easily verify that it is reflexive, symmetric and transitive by equation transformations, so the proof is omitted here.

Thereby the setoid of integers can be formed as:

```
ℤ-Setoid : Setoid
ℤ-Setoid = record
  { Carrier       = ℤ₀
  ; _≈_           = _∼_
  ; isEquivalence = _∼_isEquivalence
  }
```

4.2.2 The set definition $\mathbb{Z}$

The usual notation for an integer is a natural number with a positive or negative sign in front:

- $+\_ : \mathbb{N} \to \mathbb{Z}$
- $-\_ : \mathbb{N} \to \mathbb{Z}$

If we define $\mathbb{Z}$ in this way, 0 has two intensionally different representations, which is considered harmful because we lose canonicity and it will result in unnecessary troubles. We can fix this problem by giving a special constructor for 0:

- $+\mathrm{suc}\_ : \mathbb{N} \to \mathbb{Z}$
- $\mathrm{zero} : \mathbb{Z}$
- $-\mathrm{suc}\_ : \mathbb{N} \to \mathbb{Z}$

In principle, it is preferable to use fewer constructors, because there will be fewer cases to analyse when doing pattern matching. Taking into account the embedding of natural numbers into integers, it makes sense to combine the positive integers with 0:

```
data ℤ : Set where
  +_    : ℕ → ℤ
  -suc_ : ℕ → ℤ
```

Although it is a not symmetric, we achieve both canonicity and simplicity.

4.2.3 The definable quotient of integers

The basic ingredients for the definable quotient of integers have been given. One essential component of the quotient structure which relates the base type and quotient type is a normalisation function which can be recursively defined as follows:

```
[_]               : ℤ₀ → ℤ
[ m , 0 ]         = + m
[ 0 , suc n ]     = -suc n
[ suc m , suc n ] = [ m , n ]
```

The soundness property of [_] can be proved by case analysis, but it turns out to be too complicated.

It is plausible define the embedding function written as ⌜_⌝. In fact the first two cases in definition of [_] already gives us the answer:

```
⌜_⌝         : ℤ → ℤ₀
⌜ + n ⌝     = n , 0
⌜ -suc n ⌝ = 0 , suc n
```

To complete the definition of definable quotient, there are several properties to prove. The stability and completeness can simply be proved by recursion. To prove soundness, we can first prove an equivalent lemma:

$$\text{sound}' : \forall (a, b : \mathbb{Z}) \rightarrow \ulcorner a \urcorner \sim \ulcorner b \urcorner \rightarrow a = b$$

Given $a \sim b$, by transitivity and symmetry of $\sim$ and completeness, we can prove that

$$\ulcorner [a] \urcorner \sim a \sim b \sim \ulcorner [b] \urcorner$$

Applying the lemma sound′, we get

$$[a] = [b]$$

hence [_] is *sound* (it respects $\sim$).

These properties have been verified in Agda (see Appendix A), we omit the detailed proofs here.

4.3 Rational numbers

4.3.1 Setoid: fractions

In Type Theory, we usually choose fractions to represent rational numbers because the decimal expansion of a rational number can be infinite. Any rational number can be expressed as a fraction $\frac{m}{n}$ consists of an integer m called *numerator* and a non-zero integer n called *denominator*.

There are different ways to interpret a fraction: two natural numbers together with a sign; two integers with a condition that the denominator is non-zero; an integer for numerator and a natural number for denominator. It is clear that the last one is the simplest,

$$\mathbb{Q}_0 = \mathbb{Z} \times \mathbb{N},$$

where the sign of rational number is contained in numerator and it is easy to exclude 0 by viewing n as a denominator $n + 1$. This means that we encode rational numbers as follows:

```
data ℚ₀ : Set where
  _/suc_ : (n : ℤ) → (d : ℕ) → ℚ₀
```

such that 2/suc 2 stands for $\frac{2}{3}$.

Different fractions can represent the same rational numbers.

$$\frac{a}{b} = \frac{c}{d}$$

However since integers are not closed under division, we have to transform the equation into

$$a \times d = c \times b$$

in order to encode it as an equivalence relation as follows:

```
_∼_   : ℚ₀ → ℚ₀ → Set
n1 /suc d1 ∼ n2 /suc d2 =   n1 ℤ* (+ suc d2) ≡ n2 ℤ* (+ suc d1)
```

4.3.2 Set: reduced fractions

A fraction $\frac{a}{b}$ is reduced if and only if a and b are coprime which means if their greatest common divisor is 1. Equivalently, we can say that their absolute values are coprime, thus we can define a predicate of $\mathbb{Q}_0$ as

```
IsReduced : ℚ₀ → Set
IsReduced (n /suc d) = True (coprime? ∣ n ∣ (suc d))
```

which decides whether they are coprime or not, if it is the case, it becomes $\top$, otherwise it becomes $\bot$. Therefore, it is a propositional set (there is at most one inhabitant).

The reduced fractions are canonical representations of rational numbers. It is a subset of fractions, so we only need to add the property above to it:

```
ℚ : Set
ℚ = Σ[ q : ℚ₀ ] IsReduced q
```

This is equivalent to the definition of $\mathbb{Q}$ in Agda standard library which uses record types.

4.3.3 The definable quotient of rational numbers

The set definition $\mathbb{Q}$ ensures the canonicity of representations, but it complicates the manipulation of rational numbers.

To calculate rational numbers using $\mathbb{Q}$, we have to reduce fractions in every step which is unnecessary from our usual experience because operations can be carried out correctly on unreduced forms. In fact someone complained about this problem[1] in practical use of unreduced fractions in Agda standard library.

Therefore a definable quotient of rational numbers consisting of both $\mathbb{Q}_0$ and $\mathbb{Q}$ and conversions between them is very useful. We can carry out calculations and prove properties using $\mathbb{Q}_0$ and reduce fraction when a canonical form is required. We believe that it can also improve the computational efficiency, even though some people claim that the unreduced numbers can be too large to make it efficient.

We only need to implement the reduction process to be the normalisation function.

We first define an auxiliary function calℚ. It calculates a reduced fraction for a positive rational represented by a pair of natural numbers $x, y : \mathbb{N}$ with a condition that y is not zero. It uses a library function gcd′ which computes the greatest common divisor di, the the new numerator q_1, the new denominator q_2 such that $q_1 * di = x$, $q_2 * di = y$ and q_1 and q_2 are coprime.

```
calℚ : ∀(x y : ℕ) → y ≢ 0 → ℚ
calℚ x y neo with gcd′ x y
calℚ .(q₁ ℕ* di) .(q₂ ℕ* di) neo
  | di , gcd-* q₁ q₂ c = (numr /suc pred q₂) , iscoprime
    where
      numr = + q₁
      deno = suc (pred q₂)

      lzero : ∀ x y → x ≡ 0 → x ℕ* y ≡ 0
      lzero .0 y refl = refl

      q2≢0 : q₂ ≢ 0
      q2≢0 qe = neo (lzero q₂ di qe)

      invsuc : ∀ n → n ≢ 0 → n ≡ suc (pred n)
      invsuc zero nz with nz refl
      ... | ()
```

[1] Discussion on the Agda mailing list: http://comments.gmane.org/gmane.comp.lang.agda/6372

```
invsuc (suc n) nz = refl

deno≡q2 : q₂ ≡ deno
deno≡q2 = invsuc q₂ q2≢0

copnd : Coprime q₁ deno
copnd = subst (λ x → Coprime q₁ x) deno≡q2 c

witProp : ∀ a b → GCD a b 1
  → True (coprime? a b)
witProp a b gcd1 with gcd a b
witProp a b gcd1 | zero , y with GCD.unique gcd1 y
witProp a b gcd1 | zero , y | ()
witProp a b gcd1 | suc zero , y = tt
witProp a b gcd1 | suc (suc n) , y
                                   with GCD.unique gcd1 y
witProp a b gcd1 | suc (suc n) , y | ()

iscoprime : True (coprime? | numr | deno)
iscoprime = witProp _ _ (coprime-gcd copnd)
```

To apply this function to negative rational numbers, we only need to define the negation as

```
-_ : ℚ → ℚ
-_ ((n /suc d) , isC) = ((ℤ- n) /suc d) ,
   subst (λ x → True (coprime? x (suc d)))
   (forgetSign n) isC
   where
   forgetSign : ∀ x → | x | ≡ |  ℤ- x |
   forgetSign (-suc n) = refl
   forgetSign (+ zero) = refl
```

```
    forgetSign (+ (suc n)) = refl
```

Then it is natural to define the normalisation function as

```
[_] : ℚ₀ → ℚ
[ (+ n) /suc d ] = calℚ n (suc d) (λ ())
[ (-suc n) /suc d ] = - calℚ (suc n) (suc d) (λ ())
```

Because $\mathbb{Q}$ is just a subset of $\mathbb{Q}_0$, the embedding function is just the first projection of the Σ-types.

```
⌜_⌝ : ℚ → ℚ₀
⌜_⌝ = proj₁
```

To complete the definable quotient, we have to prove all essential properties. Because of the complicated definitions, we only sketch proofs here:

- The soundness can be understood as the uniqueness of reduced forms which can be proved from the unique prime factorization of integers. Given the equation $a_1 * b_2 = b_1 * a_2$, we can cancel the two greatest common divisors, and the equation becomes $q_1 * r_2 = r_1 * q_2$ where (q_1, q_2) and (r_1, r_2) are the reduced pairs of (a_1, a_2) and (b_1, b_2), and both pairs are coprime. The coprime property implies that there are no common prime factors, thus we can deduce that the set of prime factors of q_1 is a subset of r_1 and vice versa, hence $q_1 = r_1$ and $q_2 = r_2$ for the same reason. In fact this has been implemented in Agda standard library for rational numbers.

- The stability means that given a reduced fraction, if we reduce it again, it stays the same. It is the case because from the coprime property we can deduce that their greatest common divisor is 1, thus the new numerator and denominator are the same as the old ones.

- The completeness means that if we reduce a fraction $\frac{x}{y}$, the reduced one is equivalent to it. This is also easy to verify because in the reduction process we have the explicit proofs of $q_1 * di = x$, $q_2 * di = y$, to prove $q_1 * (q_2 * di) = (q_1 * di) * q_2$ we can simply cancel the greatest common divisor di. We ignore the sign of the fractions because it can be cancelled in those equations.

4.4 The application of definable quotients

Usually the definable quotient structure is useful when the base type (or carrier) is easier to use. For example, compared to $\mathbb{Z}$, $\mathbb{Z}_0$ has only one pattern which leads to less case distinctions. In the case of rational numbers, $\mathbb{Q}_0$ does not have the coprime property, which also reduces complexity. Thus we can define operators and prove properties on setoid representation. Then, we can easily lift them by two ways of conversions.

Operators We can lift a unary operator f by

$$\text{liftop1}(f) :\equiv [\_] \circ f \circ \ulcorner \_ \urcorner$$

This approach can be generalised to n-ary operators. An operator respects $\sim$ if

$$a \sim b \to f(a) \sim f(b)$$

It has to be noticed that this property is not required to verify before lifting. It allows unsafe lifting but it is simpler. We can verify the properties separately.

For integers, most of the definitions for operators on $\mathbb{Z}_0$ can be induced from mathematical equations. Because we can only do valid operations on natural numbers ($+$ or $*$) except $-$ which is replaced by pairing operation. For instance, to define the addition operator

$$(a_1 - b_1) + (a_2 - b_2) = (a_1 + a_2) - (b_1 + b_2)$$

provides a clear way to define it, which respects $\sim$.

```
_+_ : ℤ₀ → ℤ₀ → ℤ₀
(x+ , x-) + (y+ , y-) = (x+ ℕ+ y+) , (x- ℕ+ y-)
```

There is only one case, which means that we usually do not need to do case analysis when proving properties about additions. In fact, the same is true for other operators.

Properties Properties about setoids and operators defined on setoids can be lifted by using soundness of $[\_]$ and operators, completeness, stability and equivalence properties. For example, given a unary operator $f : A \to A$ such that $\forall (a : A) \to f(f(a)) \sim a$, we can prove that $\forall (q : Q) \to \mathrm{liftop1}(f)(\mathrm{liftop1}(f)(q)) = q$ as follows:

Proof. By definitional expansion, the property can be rewritten as:

$$([\_] \circ f \circ \ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) = q$$

Applying the assumption $\forall (a : A) \to f(f(a)) \sim a$ on $\ulcorner q \urcorner$, we get

$$(f \circ f \circ \ulcorner \_ \urcorner)(q) \sim \ulcorner q \urcorner$$

By completeness on $(f \circ \ulcorner \_ \urcorner)(q)$, we can prove that

$$(\ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) \sim (f \circ \ulcorner \_ \urcorner)(q)$$

Because f respects $\sim$,

$$(f \circ \ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) \sim (f \circ f \circ \ulcorner \_ \urcorner)(q)$$

By transitivity of $\sim$

$$(f \circ \ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) \sim \ulcorner q \urcorner$$

Because $[\_]$ respects $\sim$,

$$([\_] \circ f \circ \ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) = ([\_] \circ \ulcorner \_ \urcorner)(q)$$

Finally, by applying stability on the right hand side, we prove that

$$([\_] \circ f \circ \ulcorner \_ \urcorner \circ [\_] \circ f \circ \ulcorner \_ \urcorner)(q) = q$$

□

As we mentioned, one of the important motivations of definable quotients is that the setoid form is simpler and therefore properties can be proved with less case distinctions. Another advantage is that usually there are functions and properties available for the setoid form that are very useful.

In [59], the author has proved all necessary properties to form a commutative ring of integers in Agda. In practice, for the set definition of integers, most of the basic operations and simple theorems are not unbearably complicated. However, the number of cases grows exponentially when case analysis is unavoidable. Although it is possible to prove lemmas which cover several cases, it is still very inefficient in general. We have experienced extreme difficulty in proving the distributivity law within the ring of integers.

Case: distributivity proof As an example we only discuss the left distributivity

$$x \times (y + z) = x \times y + x \times z$$

We use the multiplication defined in the standard library which calculates signs and absolute values separately:

```
_ℤ*_ : ℤ → ℤ → ℤ
i ℤ* j = sign i S* sign j ◃ ∣ i ∣ ℕ* ∣ j ∣
```

If we split all cases, we will have $2*2*2$ cases in total, which is rather complicated and inconvenient. Therefore, we decide to combine several cases.

When all of them are non-negative integers, we can apply apply the left distributivity law of natural numbers which we assume is available. In fact, it can be applied in all cases in which y and z have the same sign, because signs can be moved out. Thus we can write some parts of the proof (Note: DistributesOver^{l} means that the first operator distributes over the second one):

```
distˡ   : _ℤ*_ DistributesOverˡ _ℤ+_
distˡ x y z with sign y S≟ sign z
distˡ x y z    | yes p
  rewrite p
    | lem1 y z p
    | lem2 y z p =
  trans (cong (λ n → sign x S* sign z ◃ n)
  (ℕdistˡ ∣ x ∣ ∣ y ∣ (∣ z ∣)))
  (lem3 (∣ x ∣ ℕ* ∣ y ∣) (∣ x ∣ ℕ* ∣ z ∣) _)
distˡ x y z | no ¬p = ...
```

To prove these simpler cases we need three lemmas,

```
lem1 : ∀ x y → sign x ≡ sign y → | x ℤ+ y | ≡ | x | ℕ+ | y |
lem1 (-suc x) (-suc y) e = cong suc (sym (m+1+n≡1+m+n x y))
lem1 (-suc x) (+ y) ()
lem1 (+ x) (-suc y) ()
lem1 (+ x) (+ y) e = refl

lem2 : ∀ x y → sign x ≡ sign y → sign (x ℤ+ y) ≡ sign y
lem2 (-suc x) ( -suc y) e = refl
lem2 (-suc x) (+ y) ()
lem2 (+ x) (-suc y) ()
lem2 (+ x) (+ y) e = refl


lem3 : ∀ x y s → s ◁ (x ℕ+ y) ≡ (s ◁ x) ℤ+ (s ◁ y)
lem3 0 0 s = refl
lem3 0 (suc y) s = sym (ℤ-id-l _)
lem3 (suc x) y s = trans (h s (x ℕ+ y)) (
    trans (cong (λ n → (s ◁ suc 0) ℤ+ n) (lem3 x y s)) (
    trans (sym (ℤ-+-assoc (s ◁ suc 0) (s ◁ x) (s ◁ y))) (
    cong (λ n → n ℤ+ (s ◁ y)) (sym (h s x)))))
  where
  h : ∀ s y → s ◁ suc y ≡ (s ◁ (suc 0)) ℤ+ (s ◁ y)
  h s 0 = sym (ℤ-id-r _)
  h Sign.- (suc y) = refl
  h Sign.+ (suc y) = refl
```

However, intuitively speaking, if y and z have different signs, it is impossible to apply the left distributivity law for natural numbers. There is no rule to turn $x * (y - z)$ into an expression which only contains natural numbers. The case analysis is unavoidable here, and we have to prove it from scratch. From the author's experience, this is very complicated and inefficient because we can not refer to proved theorems in a meaningful way.

It is much simpler to prove distributivity for Z_0. As we have mentioned, the definitions of these operators only involve operators for natural numbers. Therefore all these properties which only involve plus, minus and multiplication, are intensional equations about natural numbers with the operators which forms a commutative semiring of natural numbers. We can use these laws to prove distributivity easily.

In fact with the help of the *ring solver*, it can be proved automatically. The *ring solver* is an automatic equation checker for rings, e.g. the ring of integers. It is implemented based on the theory described in [43].

```
distˡ :  _*_ DistributesOverˡ _+_
distˡ (a , b) (c , d) (e , f) = solve 6
  (λ a b c d e f → a :* (c :+ e) :+ b :* (d :+ f) :+
    (a :* d :+ b :* c :+ (a :* f :+ b :* e))
    :=
    a :* c :+ b :* d :+ (a :* e :+ b :* f) :+
    (a :* (d :+ f) :+ b :* (c :+ e))) refl a b c d e f
```

It is not the simplest way to use the ring solver since we have to feed the type (i.e. the equation) to the solver. In fact Agda has a feature called "reflection" which helps us to quote the type of the current goal so that the application of the ring solver can be automated. There is already some work done by van der Walt [85]. It can be seen as an analogy of the "ring" tactic from Coq.

To form the commutative ring of integers, we can prove all properties using the ring solver. However, the ring solver has to calculate the proof which takes a very long time to type check from our experience. As these basic laws are used a lot in complicated theorems, pragmatically speaking, it is better not to prove them using the ring solver. Instead, we can manually construct the proof terms to improve efficiency of library code, sacrificing some conveniences.

Luckily, it is still much simpler than the ones for the set of integers $\mathbb{Z}$. First, there is only one case of integer and as we know the equations are indeed equations of natural numbers which can be proved using only the properties in the commutative semiring of natural numbers. There is no need to prove some properties for ℤ from scratch like in the proof of distributivity.

```
dist-lemˡ : ∀ a b c d e f →
  a ℕ* (c ℕ+ e) ℕ+ b ℕ* (d ℕ+ f) ≡
  (a ℕ* c ℕ+ b ℕ* d) ℕ+ (a ℕ* e ℕ+ b ℕ* f)
dist-lemˡ a b c d e f = trans
  (cong₂ _ℕ+_ (ℕdistˡ a c e) (ℕdistˡ b d f))
  (swap23 (a ℕ* c) (a ℕ* e) (b ℕ* d) (b ℕ* f))

distˡ : _ℤ₀*_ DistributesOverˡ _ℤ₀+_
distˡ (a , b) (c , d) (e , f) =
  cong₂ _ℕ+_ (dist-lemˡ a b c d e f)
  (sym (dist-lemˡ a b d c f e))
```

We only need one special lemma which can be proved by applying distributivity laws for natural numbers. The swap23 is a commonly used equation rewriting lemma

$(m + n) + (p + q) = (m + p) + (n + q)$

After all, the application of the quotient structure in the integer case provides a general approach to defining functions and prove theorems when the base types are simpler to deal with. When working with the field of rational numbers, we can benefit from the setoid representation. We use ℤ and ℕ for the definition of ℚ, and ℤ itself uses only ℕ. Therefore, any equation of rational numbers amounts to an equation of natural numbers, allowing us to apply the ring solver.

4.5 Related work

Courtieu [34] considers an extension of the calculus of inductive constructions (CIC), an intensional type theory, by *normalized types*. Those can be seen as type formers for definable quotients in our sense, namely quotients which have a normalisation function. Therefore, to form a normalised type, a normalisation function is required instead of an equivalence relation. He also provides an example of integers, where the base type has three constructors 0, S for successors and P for predecessors.

Cohen [28] also defines a quotient structure in Coq, which consists of Q as a quotient type, T as base type, two mapping $\mathsf{pi} : \mathsf{T} \rightarrow \mathsf{Q}$ and $\mathsf{repr} : \mathsf{Q} \rightarrow \mathsf{T}$ and a proof that pi is a left inverse of repr. It is similar to our algebraic structure of definable quotients without an equivalence relation involved, pi corresponds to $[\_]$, repr corresponds to emb, and the equivalence relation can be recovered simply: if for any two $s, t : \mathsf{T}$ such that $\mathsf{pi}(s) = \mathsf{pi}(t)$, then they are equivalent.

4.6 Summary

In this chapter, we have shown that, although we work in a theory in which quotient types are unavailable, there are some quotients that are themselves definable together with a normalisation function without using quotient types.

We introduced several algebraic structures for quotients which can be seen as "manual construction" of quotient types. A *prequotient* gives the basic ingredients for later constructions. We give two equivalent definitions of *quotients*, one of which has a dependent eliminator, while the other (as given by Hofmann) adds a non-dependent eliminator and an induction principle. A *definable quotient* includes an embedding function selecting a canonical choice for each equivalence class such that $[\_]$ is correctly specified as a normalisation function. This is very useful in practice. It provides us with a flexible conversion between setoid representations and set representations. We can usually benefit from the convenience of the simple setoid form and auxiliary functions without losing canonicity of set representation, hence it is not necessary to redefine all kinds of functions and types on sets e.g. lists, on setoids again.

To show the application of definable quotients, we used two examples, the set of integers and the set of rational numbers. Some concrete cases have been given to show how to lift operations and theorems from setoid representations. We illustrated the advantages of definable quotients in the comparison between Z and Z_0, using the proof of distributivity for the commutative ring of integers.

Chapter 5

Undefinable Quotients

In this chapter, we will discuss some other quotients which are not definable via normalisation, for example the set of real numbers as Cauchy sequences of rational numbers [19] and finite multisets represented by lists. We say that a quotient is undefinable if there is no definable normalisation function which returns a canonical choice for each equivalence class. For the Cauchy sequences of rational numbers, Nicolai Kraus [57] has shown that all definable endofunctions respecting the equivalence relation have to be constant, hence it is impossible to define a normalisation function. We reproduce the proof here and extend it to other cases especially, the partiality monad. It has to be noticed that the proof is conducted in basic Martin-Löf type theory and can be generalised to any extension as long as it admits the Brouwer's continuity principle, i.e. definable functions are continuous [83].

5.1 Definability via normalisation

Although we have provided the definition of *definable quotients* (see Definition 4.5), it is not always the case that the quotient set can be defined inductively and we are able to talk about a normalisation function as $[\_] : A \to Q$. Therefore, we provide a different characterisation of the property which only talks about a setoid $(A, \sim)$.

Definition 5.1 (Definable via normalisation). *Given a setoid $(A, \sim)$, the quotient $A/\sim$ is definable via normalisation if there is an endofunction $[\_]_0$ which is a* normalisation function*:*

- $[\_]_0 : A \to A$
- sound : $\forall (a, b : A) \to a \sim b \to [a]_0 = [b]_0$
- complete : $\forall (a : A) \to [a]_0 \sim a$

It is actually equivalent to say the quotient is definable: First, given $[\_]_0 : A \to A$ specified as above,

- The quotient set can be defined as

  $$Q :\equiv \Sigma(a : A), [a]_0 = a$$

- The "normalisation function" is

  $$[a] :\equiv ([a]_0, \mathrm{refl})$$

  which is also sound because $[\_]_0$ is sound.

- The embedding function is just first projection

  $$\mathrm{emb} :\equiv \pi_1$$

- Stability: given $(a, p) : \Sigma(a : A), [a]_0 = a$

  $$[\mathrm{emb}(a, p)] \equiv ([a]_0, \mathrm{refl})$$

  Hence we need to prove $([a]_0, \mathrm{refl}) = (a, p)$.

  We can prove it by J,

  $$\mathsf{J}(t, [a]_0, a, p) : ([a]_0, \mathrm{refl}) = (a, p)$$

  where $t(x) :\equiv \mathrm{refl} : (x, \mathrm{refl}) = (x, \mathrm{refl})$

- Completeness: given $a : A$, we need to prove $\mathrm{emb}[a] \sim a$ which turns out to be

  $$[a]_0 \sim a$$

  This is exactly the completeness property in the specification of $[\_]_0$.

In the other direction, given a definable quotient,

- $$[\_]_0 :\equiv \mathrm{emb} \circ [\_]$$
- Soundness: given $a, b : A$ such that $a \sim b$, because $[\_]$ is sound, we know

  $$[a] = [b]$$

  By the congruence rule,

  $$\mathrm{emb}[a] = \mathrm{emb}[b]$$

  hence $[\_]_0$ is sound as well.
- Completeness: given $a : A$, $\mathrm{emb}[a] \sim a$ is just the completeness property of the definable quotient.

5.2 Real numbers as Cauchy sequences

One attempt to define the real numbers is via the set $\mathbb{R}_0$ of Cauchy sequences. We can define an equivalence relation $\sim$ on $\mathbb{R}_0 \times \mathbb{R}_0$, where two Cauchy sequences are equivalent if and only if their point-wise differences converges to 0. This defines a setoid $(\mathbb{R}_0, \sim)$. We give the definitions in detail below:

Definition 5.2. *A function $f : \mathbb{N} \to \mathbb{Q}$ is called a **Cauchy sequence** if*

$$\mathsf{isCauchy}(f) :\equiv \forall(\varepsilon : \mathbb{Q}^+) \to \exists(m : \mathbb{N})\ \forall(i : \mathbb{N}) \to i > m \to |f_i - f_m| < \varepsilon \tag{5.1}$$

Hence we can define $\mathbb{R}_0$ *as*

$$\mathbb{R}_0 :\equiv \Sigma(f : \mathbb{N} \to \mathbb{Q})\ \mathsf{isCauchy}(f)$$

Two Cauchy sequences are equivalent if and only if their point-wise difference converges to 0:

$$r \sim s :\equiv \forall(\varepsilon : \mathbb{Q}^+) \to \exists(m : \mathbb{N})\ \forall(i : \mathbb{N}) \to i > m \to |r_i - s_i| < \varepsilon$$

To implement this definition, the existential quantifier is usually encoded as a Σ-type so that we can guess the real number from the explicit witness m. However, we would like to keep the proof propositional so that the property of being a Cauchy sequence is proof-irrelevant.

To combine these two things, we can use an alternative equivalent definition of the property, where we change the type of f to be $\mathbb{N}^+ \to \mathbb{Q}$ so that we can write:

$$\mathsf{isCauchy}(f) :\equiv \forall(k : \mathbb{N}^+), \forall(m, n > k) \to |f_m - f_n| < \frac{1}{k} \tag{5.2}$$

The rate of convergence is fixed so that we can guess the number while the condition is also propositional. Note that we use some shorthand notations in these definitions.

A slight modification of the definition which is still equivalent is

$$\mathsf{isCauchy}(f) :\equiv \forall(n, m : \mathbb{N}^+), n < m \to |f_n - f_m| < \frac{1}{n} \tag{5.3}$$

5.3 $\mathbb{R}_0/\sim$ is undefinable via normalisation

In Intensional Type Theory without quotient types, we can define a setoid $(\mathbb{R}_0, \sim)$ to represent the set of real numbers. However we can show that there is no definable normalisation function $[\_]_0 : \mathbb{R}_0 \to \mathbb{R}_0$ in the sense of 5.1.

We have made an attempt to prove that the set of reals is undefinable in the presence of local continuity (see Section. 5 in [10]). We say that two $a, b : A$ are *separable*, if there exists a definable test $P : A \to \mathbf{2}$ such that $P(a) \neq P(b)$. Then, we claim that a definable set A is *discrete* in the sense that $a \neq b$ always implies that a and b are separable. However, this is not the case, as Martín Escardó pointed out. He provides a counterexample in which he shows that, for two distinguishable terms (i.e. $a \neq b$), there is no definable test [39]. We sketch the proof here:

Proof. In the proof, he uses $\mathbb{N}_\infty :\equiv \mathbb{N} \to \mathbf{2}$ which is a decreasing sequence of $\mathbf{2}$ called *generic convergent sequence.* Intuitively speaking, $11000\ldots$ represents 2 and the sequence of 1, namely $1111\ldots$ represents ∞. For simplicity, we write s_k for the sequence whose first k digits are 1 and whose remaining digits are 0.

From continuity, we know that:

given any definable function $f : \mathbb{N}_\infty \to \mathbf{2}$, there exists $n : \mathbb{N}$ such that for all $s_k : \mathbb{N}_\infty$ $(k \geq n)$ whose first n digits coincide with ∞, $f(s_k) = f(\infty)$.

Set $X :\equiv \Sigma u : \mathbb{N}_\infty, u = \infty \to \mathbf{2}$,

$s_k^0 :\equiv (s_k, \lambda r \to 0)$ and

$s_k^1 :\equiv (s_k, \lambda r \to 1)$,

there are two unequal terms of X, $\infty_0 = s_\infty^0$ and $\infty_1 = s_\infty^1$,

such that for all definable function $f : X \to \mathbf{2}$, $f(\infty_0) = f(\infty_1)$.

To prove it, assume $f(\infty_0) \neq f(\infty_1)$. We can prove that for all $k : N$ such that $(s_k \neq \infty)$,

$$f(s_k^0) = f(s_k^1)$$

because the second part is always the same due to the fact that $s_k \neq \infty$. From continuity, we can deduce that

$$f(\infty_0) = f(s_k^0) = f(s_k^1) = f(\infty_1)$$

which contradicts our premise. □

Here we present a meta-level proof to show that all definable endofunctions are constant, hence no normalisation function is definable.

5.3.1 Preliminaries

We use some topological notions.

Recall that a **metric space** is a set where a notion of distance (called a metric) between elements of the set is defined. It is an ordered pair (M, d) where M is a set and d is a metric on M:

1. M is a set,
2. and $d : M \times M \to \mathbb{R}^*$ s.t.
3. $d(x, y) = 0 \iff x = y$
4. $d(x, y) = d(y, x)$
5. $d(x, y) + d(y, z) \geq d(x, z)$

We usually give a standard topological structure for types.

For example for types with a decidable equality which are called **discrete types**, e.g. **2**, $\mathbb{N}$, $\mathbb{Q}$, we can give metric spaces as

- $(\mathbf{2}, h)$ where $h(m, n) = \begin{cases} 0 & \text{if } m = n \\ 1 & \text{if } m \neq n \end{cases}$

- $(\mathbb{N}, d)$ where $d(m,n) = \begin{cases} 0 & \text{if } m = n \\ 1 & \text{if } m \neq n \end{cases}$
- $(\mathbb{Q}, e)$ where $e(m,n) = \begin{cases} 0 & \text{if } m = n \\ 1 & \text{if } m \neq n \end{cases}$

For sequences over a discrete type, especially the sequences over $\mathbb{Q}$, the *distance* between two functions $f_1, f_2 : \mathbb{N}^+ \to \mathbb{Q}$ can be defined as

$$d(f_1, f_2) = 2^{-\mathsf{inf}\{k \in \mathbb{N}^+ \mid f_1(k) \neq f_2(k)\}} \tag{5.4}$$

which makes up a metric space if we use 5.3 as the definition of Cauchy sequences. If we define $\mathbb{R}_0$ using 5.2, there would be two different proof terms for the same sequence, hence $d(x,y) = 0 \iff x = y$ is violated and it is not a metric space.

Given two metric spaces (X, d) and (Y, e), a function $f : X \to Y$ is *continuous* if for every $x : X$ and $\epsilon > 0$ there exists a $\delta > 0$ such that

$$\forall y : X, d(x,y) < \delta \Rightarrow e(f(x), f(y)) < \epsilon$$

With the standard topological structures, we say that definable functions are *continuous* which is usually called Brouwer's continuity principle. It may not hold in Intensional Type Theory, but it holds meta-theoretically. Intuitively speaking, for a function $f : (\mathbb{N}^+ \to \mathbb{Q}) \to \mathbf{2}$, it only inspects finite many terms of the input sequences to compute the result.

We define a generalised condition of $\mathsf{isCauchy}$:

Definition 5.3. *For a sequence $f : \mathbb{N}^+ \to \mathbb{Q}$, we say that f is* Cauchy with factor *k, written as $\mathsf{isCauchy}_k$, for some $k \in \mathbb{Q}^+$, if*

$$\mathsf{isCauchy}_k(f) \;:\equiv\; \forall (n, m : \mathbb{N}^+) \to n < m \to |f_n - f_m| < \frac{1}{k \cdot n}. \tag{5.5}$$

The usual condition $\mathsf{isCauchy}$ is just "Cauchy with factor 1".

The main proposition we make is:

Proposition 5.4. $\mathbb{R}_0/\sim$ ***is connected.*** *In Type Theory, it means that any definable (continuous) function*

$$f : \mathbb{R}_0 \to \mathbf{2}$$

which respects $\sim$*, is* constant.

Proof. Assume f which respects $\sim$.

Consider the "naive" set model (with "classical standard mathematics" as meta-theory). It works for a minimalistic type theory with Π, Σ, W, $=$, $\mathbb{N}$. The general idea is to interpret our definitions in the set model using function $[\![\_]\!]$, and we prove that $[\![f]\!] : [\![\mathbb{R}_0]\!] \to [\![\mathbf{2}]\!]$ is constant in the model, which implies it is also constant in the theory.

By abuse of notation, we write $[\![\mathbb{R}_0]\!]$ for the set of Cauchy sequences without proof terms which is justifiable. For simplicity, we write $\mathbb{R}$ for the field of real numbers which can be defined as the quotient set $[\![\mathbb{R}_0]\!] \;/\; [\![\sim]\!]$. It does not make confusion because $\mathbb{R}$ is not defined in the theory. We also just write $=$ for equality and 3 for natural numbers in both the theory and the model.

In the model, we have a limit function $\bar{\cdot} : [\![\mathbb{R}_0]\!] \to \mathbb{R}$, thus given a Cauchy sequence $r : \mathbb{R}_0$, the real numbers it represents can be written as $\overline{[\![r]\!]} \in \mathbb{R}$.

We assume $[\![f]\!]$ is non-constant, hence there are two $c_1, c_2 : [\![\mathbb{R}_0]\!]$ such that

$$[\![f]\!](c_1) \neq [\![f]\!](c_2)$$

Define

$$m_1 :\equiv \mathsf{sup}\{\overline{d} \in \mathbb{R} \mid d \in [\![\mathbb{R}_0]\!], \overline{d} \leq \mathsf{max}(\overline{c_1}, \overline{c_2}), [\![f]\!](d) = [\![1_{\mathbf{2}}]\!]\} \tag{5.6}$$

$$m_2 :\equiv \mathsf{sup}\{\overline{d} \in \mathbb{R} \mid d \in [\![\mathbb{R}_0]\!], \overline{d} \leq \mathsf{max}(\overline{c_1}, \overline{c_2}), [\![f]\!](d) = [\![0_{\mathbf{2}}]\!]\} \tag{5.7}$$

(note that one of these two necessarily has to be $\overline{c_1}$ or $\overline{c_2}$, whichever is bigger).

Set $m :\equiv \mathsf{min}(m_1, m_2)$. Because m is a supremum, we can observe that in *every* neighbourhood U of m, given any t, we can always find another point $x \in U$ such that $x = \overline{e}$ (for some e) with $[\![f]\!](e) \neq [\![f]\!](t)$.

Let $c \in [\![\mathbb{R}_0]\!]$ be a Cauchy sequence such that $\overline{c} = m$. We may assume that c satisfies the condition $[\![\mathsf{isCauchy}_5]\!]$.

From the assumption we know f is continuous, hence $[\![f]\!]$ is also continuous. It means that for an arbitrary $\epsilon < 1$, there exists $n_0 \in [\![\mathbb{N}]\!]$ such that for any Cauchy sequence $c' \in [\![\mathbb{R}_0]\!]$, if the first n_0 sequence elements of c' coincide with those of c, namely the distance

$$g(c, c') = 2^{-\mathsf{inf}\{k \in \mathbb{N} \mid c(k) \neq c'(k)\}} < 2^{-n_0}$$

then

$$h([\![f]\!](c), [\![f]\!](c')) < \epsilon < 1$$

hence $[\![f]\!](c') = [\![f]\!](c)$.

Write $U \subset [\![\mathbb{R}_0]\!]$ for the set of Cauchy sequences which fulfil this property, and $\overline{U} :\equiv \{\overline{d} \mid d \in U\}$ for the set of reals that U corresponds to. We claim that $\overline{U}$ is a neighbourhood of m by proving an open interval $I :\equiv (m - \frac{1}{2n_0}, m + \frac{1}{2n_0})$ is contained in $\overline{U}$, i.e. $I \subset \overline{U}$.

Let $x \in I$, there is a Cauchy sequence $t : [\![\mathbb{R}_0]\!]$ such that $\overline{t} = x$ and we may assume that t satisfies the condition $[\![\mathsf{isCauchy}_{5n_0}]\!]$.

We can concatenate the first n_0 elements of the sequence c with t, hence define a function $g : [\![\mathbb{N}^+ \to \mathbb{Q}]\!]$ as

$$g(n) = \begin{cases} c(n) & \text{if } n \leq n_0 \\ t(n - n_0) & \text{else.} \end{cases} \tag{5.8}$$

Observe that g is also a Cauchy sequence, i.e. $[\![\mathsf{isCauchy}]\!](g)$. To verify it, the only thing that needs to be checked is whether the two "parts" of g work well together, i.e. let $0 < n \leq n_0$ and $m > n_0$ be two natural numbers. We need to show that

$$|g(n) - g(m)| < \frac{1}{n}. \tag{5.9}$$

Calculate

$$\begin{aligned}
& |g(n) - g(m)| && (5.10)\\
= \;& |c(n) - t(m - n_0)| && (5.11)\\
= \;& |c(n) - \overline{c} + \overline{c} - \overline{t} + \overline{t} - t(m - n_0)| && (5.12)\\
\leq \;& |c(n) - \overline{c}| + |\overline{c} - \overline{t}| + |\overline{t} - t(m - n_0)| && (5.13)\\
\leq \;& \frac{1}{5n} + \frac{1}{2n_0} + \frac{1}{5n_0 \cdot (m - n_0)} && (5.14)\\
\leq \;& \frac{1}{5n} + \frac{1}{2n} + \frac{1}{5n_0} && (5.15)\\
< \;& \frac{1}{n} && (5.16)
\end{aligned}$$

Because the first n_0 sequence elements of g coincide with those of c, we know that $[\![f]\!](g) = [\![f]\!](c)$.

By the definition of g, it converges to the same real number as t, i.e. $\overline{g} = \overline{t}$. It is equivalent to say $g[\![\sim]\!]t$ and by the condition $[\![f]\!]$ respects $[\![\sim]\!]$, we can prove that $[\![f]\!](t) = [\![f]\!](g) = [\![f]\!](c)$ and therefore $x = \overline{t} \in \overline{U}$. Now we can conclude that $I \subset \overline{U}$ which is equivalent to say $\overline{U}$ is a neighbourhood of m.

However it contradicts to the definition of m: in *every* neighbourhood of m, and thus in particular in $(m - \frac{1}{2n_0}, m + \frac{1}{2n_0})$, we can always find an x such that $x = \overline{e}$ (for some e) with $[\![f]\!](e) \neq [\![f]\!](c)$. □

This approach is also applicable to other discrete types.

Corollary 5.5. *Any continuous function from $\mathbb{R}_0$ to any discrete type that respects $\sim$ is constant.*

Theorem 5.6. *Any continuous function $f : \mathbb{R}_0 \to \mathbb{R}_0$ that respects $\sim$ is constant.*

Proof. Assume we have f as required.

To prove f is constant, it is enough to show that the sequence part is constant because the proof part is propositional, so by slight abuse of notation, we write $[\![f]\!] : [\![\mathbb{R}_0]\!] \to [\![\mathbb{R}_0]\!]$, omitting the proof part of f.

Given a positive natural number $n : [\![\mathbb{N}^+]\!]$, $\pi_n : [\![\mathbb{R}_0]\!] \to [\![\mathbb{Q}]\!]$ is the projection function. Define a function $h_n : [\![\mathbb{R}_0]\!] \to [\![\mathbb{Q}]\!]$ as

$$h_n :\equiv \pi_n \circ f$$

By Corollary 5.5, h_n has to be constant. Thereby f is constant everywhere, it is enough to show that f is constant.

□

Corollary 5.7. *There is no definable normalisation function on $\mathbb{R}_0$ in the sense of Definition 5.1, namely $\mathbb{R}_0/\sim$ is not definable via normalisation.*

Even though there is no definable endofunctions, it does not imply that we cannot define the set of real numbers, although we believe it is the case. In fact, Kraus has made a conjecture that for a definable type T in minimalistic type theory with Π, Σ, W, $=$, $\mathbb{N}$, if T does have two distinguishable elements, then it is not connected. Because $\mathbb{R}_0/\sim$ is connected, this conjecture implies that the the set of real numbers are not definable.

Remark 5.8 ($\mathbb{R}_0$ **is not Cauchy complete**). Is our definition $\mathbb{R}_0$ Cauchy complete? In other words, is there a representative Cauchy sequence as a limit for every equivalence class (i.e. real number)? The answer is no.

Recall that if for every Cauchy sequence of **real** numbers there is a real number as its limit, then we say it is Cauchy complete.

In classical logic, the Cauchy reals are Cauchy complete because the limit can be built via a kind of diagonalization [61]. Also classically Cauchy reals are equivalent to another definition called Dedekind Reals. However, in Type Theory both of them are not representable. We cannot find a canonical representative for each equivalence class. Intuitively speaking it is easy to find a canonical choice for any rational number but it is impossible to find one for any irrational number like π. It has been proved by Robert S. Lubarsky in [61]. If we add the axiom of Countable Choice (AC_ω) to Type Theory, Cauchy reals become Cauchy complete because it provides us a choice function for equivalence classes which helps us find a canonical choice. The AC_ω is a classical result which is stronger than the premise "in classical logic".

In the HoTT book [82] (see Section 11.3), there is a higher inductive definition of Cauchy reals $\mathbb{R}_C$ using **Cauchy approximation**. Briefly speaking, it first embeds rational numbers, and then for each $s : \mathbb{Q}^+ \to \mathbb{R}_C$ we have $lim(s) : \mathbb{R}_C$ as a limit of Cauchy sequence of real numbers, hence it is Cauchy complete. Higher inductive types allow us to define *equality* of terms as constructors in inductive definitions, see Section 2.6.4.

5.4 Other examples

5.4.1 Unordered pairs

In Type Theory, given a set A, $(a, b) : A \times A$ is an *ordered* pair. Unordered pair can be interpreted as the setoid $(A \times A, \sim)$, where $\sim$ is generated by

$$(a, b) \sim (b, a)$$

Intuitively speaking, for an arbitrary order pair (a, b), we can not decide whether (a, b) or (b, a) should be the normal form of the unordered pair they represent. In general, we can not define a normalisation function for $(A \times A, \sim)$, unless the set A has a decidable total order $\leq : A \to A \to \mathbf{Prop}$ equipped with

$$\min, \max : A \to A \to A$$

calculating the binary minimum and maximum for that order. This allows us to define $[\_]_0 : A \times A \to A \times A$ as

$$[(a, b)]_0 :\equiv (\min(a, b), \max(a, b))$$

Soundness and completeness can be easily verified by the properties of min and max.

5.4.2 Finite multisets

In Type Theory, a multiset (bag) can be seen as a generalisation of unordered pairs. Given a set A, the finite multisets of elements in A can be interpreted as the setoid (List $A, \sim$) where two lists are (bag) equivalent [36] if they are equal up to reordering. For example, $[1, 2, 2, 5, 1]$ is equivalent to $[2, 2, 1, 1, 5]$ since they are permutation of each other. We can observe that two such lists always have the same length so we use length-explicit lists – Vec here.

Given two lists $p, q : \text{Vec } A\ n$ of length n

$$p \sim q :\equiv \Sigma(\phi : \text{Fin } n \to \text{Fin } n) \text{ Bijection } \phi \ \wedge \forall(x : \text{Fin } n) \to p_x = q_{\phi(x)}$$

where $\text{Fin} : \mathbb{N} \to \mathbf{Set}$ represents finite sets and $\text{Bijection} : (\text{Fin } n \to \text{Fin } n) \to \mathbf{Prop}$ is the predicate that a mapping between finite sets is bijective.

Because finite multisets can be seen as unordered n-tuples, therefore, it is also not definable via normalisation unless A has a decidable total order which gives us a sorting function $\text{sort} : \text{Vec } A\ n \to \text{Vec } A\ n$. It allows us to define

$$[vs]_0 :\equiv \text{sort}(vs)$$

which is sound and complete by the properties of the sorting function.

5.4.3 Partiality monad

Given a set A, the set of partial/non-terminating computations over A can be represented by the partiality (delay) monad $A_{\bot}$ (or (Delay A) introduced by Capretta [25]. In Agda, the partiality (delay) monad can be coinductively defined as:

```
data Delay (A : Set) : Set where
  now : A → Delay A
  later : ∞ (Delay A) → Delay A
```

A non-terminating program can be defined by postponing computations forever:

```
never : {A : Set} → Delay A
never = later (♯ never)
```

Two computations are *strongly* bisimilar if they are the same after the same number of steps delay (there can be infinite steps):

```
data _∼_ {A : Set} : Delay A → Delay A → Set where
  now    : ∀ {x} → (now x) ∼ (now x)
  later  : ∀ {x y} (x∼y : ∞ ((♭ x) ∼ (♭ y))) → (later x) ∼ (later y)
```

If we ignore the number of steps a computation is postponed, two computations are *weakly* bisimilar if they terminate with the same value:

```
data _≈_ {A : Set} : Delay A → Delay A → Set where
  now    : ∀ {x y a} → x ↓ a → y ↓ a → x ≈ y
  later  : ∀ {x y} (x∼y : ∞ ((♭ x) ≈ (♭ y))) → (later x) ≈ (later y)
```

where x ↓ y means "x terminates with y":

```
data _↓_ {A : Set} : Delay A → A → Set where
  nowT   : ∀{a} → (now a) ↓ a
  laterT : ∀{d a} → d ↓ a → (later (♯ d)) ↓ a
```

Thus $A_\perp$ together with $\approx$ gives rise a quotient $A_\perp / \approx$ which stands for the set of partial computations.

Theorem 5.9. *There is no definable normalisation function on $A_\perp$ in the sense of Definition 5.1.*

Proof. Because there can be infinitely many later, we can not decide whether an element $a : A_\perp$ is equal to never or not.

We can interpret an element of $a : A_\perp$ as a sequence, for instance, suppose $a =$ later (later (now x)), then by abuse of notations, $a_1 =$ later, $a_2 =$ later, and $a_3 =$ now x (the rest a_n for $n > 3$ can be filled by later). Then a standard metric space for $A_\perp$ can be given by

$$g(a, b) = 2^{-\mathsf{inf}\{k \in \mathbb{N} \mid a(k) \neq b(k)\}} \tag{5.17}$$

Similar to the proof in Proposition 5.4, we can prove $A_\perp / \approx$ is connected, i.e. any definable (continuous) function $f : A_\perp \to \mathbf{2}$ which respects $\approx$ is constant.

We assume $[\![f]\!]$ is non-constant, i.e. there are $x, y : [\![A_\perp]\!]$ such that $[\![f]\!](x) \neq [\![f]\!](y)$.

We can also assume $[\![f]\!]([\![\mathsf{never}]\!]) = 1$, because $[\![f]\!]$ is continuous, there exists $n_0 \in \mathbb{N}$ such that for all $a \in [\![A_\perp]\!]$, if the first n_0 "elements" of a are laters (namely they coincide with those of never), then $[\![f]\!](a) = [\![f]\!]([\![\mathsf{never}]\!]) = 1$.

Since $[\![f]\!](x) \neq [\![f]\!](y)$, one of them must have $k < n_0$ laters before now, assume it is x then $[\![f]\!](x) = 0$ and $[\![f]\!](y) = 1$. By adding $n_0 - k$ laters, we obtain x' such that $[\![f]\!](x') = [\![f]\!](x) = 0$ because $[\![f]\!]$ respects $[\![\approx]\!]$. However, x' has n_0 laters such that $[\![f]\!](x') = [\![f]\!](\mathsf{never}) = 1$, contradicts to the just established statement.

Similarly, utilising the sequence interpretation of $A_\perp$, we can show that any endofunction $f : A_\perp \to A_\perp$ that repsects $\approx$ has to be constant on every choice of later or now, hence f is constant, therefore, there is no definable normalisation function on $A_\perp$ in the sense of Definition 5.1. □

5.5 Related work

Geuvers and Niqui have shown a construction of the real numbers using Cauchy sequences of the rational numbers based on a set of axioms in Coq. They have also the choice of different ways to define Cauchy properties. They have shown there is a model of these axioms and any two models are isomorphic. They have also discussed the equivalence between their axioms with the ones introduced by Bridges [23].

The formalisation of real numbers in Homotopy Type Theory has been discussed in the HoTT book (see Chapter 11 in [82]). Both Dedekind reals and Cauchy reals have been considered. They define the Cauchy reals via a higher inductive type, which makes them Cauchy complete.

Finite multisets as bag equivalent lists have been considered by Danielsson in [36]. He has mainly discussed bag equivalence for lists and has also generalised it to arbitrary containers. He has also provided a set equivalence which means that we can represent (finite) sets using the setoid arises from it.

5.6 Summary

To summarize, we have shown some quotients which are not definable via normalisation. In particular, we show that the set of real numbers as $\mathbb{R}_0/\!\sim$ is connected which means that any definable (continuous) function on $\mathbb{R}_0 \to \mathbf{2}$ which respects $\sim$ is constant. This implies that any definable endofunction on $\mathbb{R}_0$ is constant, hence there is no definable normalisation function for the setoid $(\mathbb{R}_0, \sim)$ that can be lifted. We similarly proved that the partiality computations which are represented by partiality monad quotiented by weak bisimilarity is also not definable via normalisation. For quotients arising from permutations, such as unordered pairs and finite multisets, a normalisation function can be defined if we have a decidable total order. In addition, we believe that these quotients are not definable (in the sense that there is a carrier Q with the properties stated in Definition 4.5), but we have not yet proved it formally.

Chapter 6

The Setoid Model

To introduce extensional concepts into Intensional Type Theory, one can simply postulate them as axioms, but this destroys the good computational properties of Type Theory. It is crucial to construct an intensional model where these extensional concepts like functional extensionality, quotient types are automatically derivable. In the usual set model, types are sets which do not have internal equalities. Therefore it is essential to enrich the structure of types, hence we can interpret types as setoids, groupoids, or ω-groupoids.

In this chapter, we mainly introduce an implementation of Altenkirch's setoid model [3] where types are interpreted as setoids. We define the model as categories with families in Agda. There is no proof irrelevant universe **Prop** in Agda, but the current version of Agda supports some proof-irrelevance features [1], for example proof-irrelevant fields in record types, proof-irrelevant arguments in function types, etc. It has been shown by Altenkirch [3] that functional extensionality is inhabited in this model. More importantly, because types are interpreted as setoids, quotient types can be defined simply by replacing equality in a given setoid. We build some basic types from [3] including Π-types, natural numbers and the simply typed universe. We also extend it to Σ-types and quotient types which are not discussed in Altenkirch's original construction.

6.1 Introduction

A setoid model of Intensional Type Theory is a model where types are interpreted as setoids i.e. every closed type comes with an equivalence relation. It is usually used to introduce extensional concepts, for example, Martin Hofmann has defined a setoid model in [48]. However a naïve version of the setoid model does not satisfy all definitional equalities. A simple model for quotient types introduced in [47] is a solution to the problem using a modified interpretation of families, but it does not allow *large eliminations*.

Altenkirch [3] proposes a different approach based on the setoid model. He uses an extension of Intensional Type Theory by a universe of propositions **Prop** as metatheory, and the η-rules for Π-types and Σ-types hold.

$$\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \tag{PROOF-IRR}$$

Prop only contains "propositional” sets which have at most one inhabitant. Notice that it is not a definition of types, which means that we cannot conclude a type is of type **Prop** if we have a proof that all inhabitants of it are definitionally equal.

The propositional universe is closed under Π-types and Σ-types:

$$\frac{\Gamma \vdash A : \mathbf{Set} \qquad \Gamma, x : A \vdash P : \mathbf{Prop}}{\Gamma \vdash \Pi\ (x : A) \to P : \mathbf{Prop}} \tag{Π-Prop}$$

$$\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma, x : P \vdash Q : \mathbf{Prop}}{\Gamma \vdash \Sigma\ (x : P)\ Q : \mathbf{Prop}} \tag{Σ-Prop}$$

The metatheory has been proved [3] to be:

- *Decidable.* Definitional equality is decidable, hence type checking is decidable.
- *Consistent.* Not all types are inhabited and not all well-typed definitional equalities hold.

- $\mathbb{N}$-*canonical.* All terms of type $\mathbb{N}$ are reducible to numerals.

Altenkirch further constructs an intensional setoid model within this metatheory using categories with families as introduced by Dybjer [38] and Hofmann [50]. It is also decidable and $\mathbb{N}$-canonical, functional extensionality is inhabited and it permits large elimination. It is decidable because its definitional equalities are interpreted by definitional equality in the metatheory which is decidable.

Remark 6.1 (The category of setoids is not LCCC). This model is the category of setoids **Std** which is a full subcategory of **Gpd** (the category of small groupoids). Every object of **Gpd** whose all homsets contain at most one morphism are in this subcategory.

It is different from a setoid model as an E-category, for instance the one introduced by Hofmann [46]. An E-category is a category equipped with an equivalence relation for homsets. The E-category of setoids in Martin-Löf type theory forms a locally Cartesian closed category (LCCC) which we call **E-setoids**. All morphisms of **E-setoids** give rise to types and they are Cartesian closed, i.e. the category is locally Cartesian closed.

Every LCCC can serve as a model for categories with families but not every category with families has to be an LCCC. In our category of setoids **Std**, not all morphisms give rise to types and it is not an LCCC. Altenkirch and Kraus have written a short note that explains why **Gpd** and **Std** are Cartesian closed but not locally Cartesian closed. As a counterexample, they give a morphism the pullback functor of which does not have a right adjoint (see [6]).

We will introduce the model along with our implementation of it in Agda. For readability, we will omit some unnecessary code. The complete code can be found in Appendix B.

6.2 Metatheory

Agda does not fulfil all requirements of the metatheory, in particular, there is no proof-irrelevant universe of propositions **Prop**. Instead Agda has irrelevancy annotations [1]. For example we can declare an argument of type A is proof-irrelevant by putting a small dot in front of it:

```
f : .A → B
f a = b
```

It implies that f does not depend computationally on this argument, hence $f(a) \equiv f(b)$ for any $a, b : A$. It can also be used in dependent function types, dependent products (record types). For example, we can define "subset" of A with respect to a predicate $B : A \to Set$ as follows

```
record Subset {a b} (A : Set a)
    (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    prj₁ : A
    .prj₂ : B prj₁
open Subset public
```

(In the code above, the variables a, b denote the levels of types.)

Thus, the proposition that the term fulfils the predicate is proof-irrelevant.

We can also declare that a function itself is proof-irrelevant

```
.g : A → B
g a = b
```

which creates a proof-irrelevant term of the result type B.

There are several restrictions of this annotation.

- One cannot declare the result type of a function as irrelevant.
- The irrelevant values cannot be used in non-irrelevant contexts.
- We cannot pattern match on irrelevant terms.

In most occasions, it replaces propositions. However there is a small problem of irrelevant fields of record types as we will see later: we can not use an irrelevant value to construct an irrelevant field or irrelevant function. For example, we can not simply write p in the place of ? in the following function

```
.ideq : ∀{A : Set}{a b : A} → .(a ≡ b) → a ≡ b
ideq p = ?
```

The reason is that the result type cannot be declared as irrelevant, although the function (or field) is proof-irrelevant which means the result is expected to be proof-irrelevant. The problem can be temporarily fixed by adding an axiom:

```
postulate
  .irrelevant : {A : Set} → .A → A
```

This issue is also discussed in [1], and hopefully can be fixed in the future. Fortunately, it only affects small bits of our code, e.g. the construction of natural numbers and universes in setoid model. Moreover, the axiom itself is proof-irrelevant so that it will not affect the $\mathbb{N}$-canonicity property.

Compared to **Prop** in the original metatheory, we have to make more efforts to imitate it using this annotations. For example, we can simply write $\sim : A \to A \to$ **Prop** for a propositional equivalence relation in the original metatheory. However, in our implementation, we write $\sim : A \to A \to$ **Set**, but in every occurrence of it we use the irrelevancy annotation, such that it behaves like a term of **Prop**.

We can easily observe that it is "closed" under Σ-types, but is not “closed” under Π-types, because we cannot declare its result type as irrelevant. Instead, we have to declare a Π-type itself is irrelevant.

This metatheory is still decidable, consistent and should be ℕ-canonical because the only axiom is irrelevance which can not be used to construct non-canonical terms of ℕ.

6.2.1 Category of Setoids: Std

We can define a setoid as usual, but declare the equivalence properties as irrelevant:

```
record Setoid : Set₁ where
  infix 4 _≈_
  field
    Carrier : Set
    _≈_     : Carrier → Carrier → Set
    .refl   : ∀{x} → x ≈ x
    .sym    : ∀{x y} → x ≈ y → y ≈ x
    .trans  : ∀{x y z} → x ≈ y → y ≈ z → x ≈ z
open Setoid public renaming
      (Carrier to |_| ; _≈_ to [_]_≈_ ; refl to [_]refl;
      trans to [_]trans; sym to [_]sym)
```

Notice that we rename our fields for readability of the code. Usually, to project out the equivalence relation for a setoid S : Setoid, one has to write _≈_ $A\ a\ b$ which is not readable. By renaming, we can write $[A]\ a \approx b$ for better style. We will also rename some fields for other records types later, but we may omit code in case it is not necessary for the understanding.

A functions between setoids consists of a function between the underlying sets and a property that it respects the equivalence relation:

```
infix 5 _⇉_

record _⇉_ (A B : Setoid) : Set where
  constructor fn:_resp:_
  field
    fn   : | A | → | B |
    .resp : {x y : | A |} →
      ([ A ] x ≈ y) →
      [ B ] fn x ≈ fn y
open _⇉_ public renaming (fn to [_]fn ; resp to [_]resp)
```

The category **Std** has a terminal object, that is, a setoid which receives a unique setoid homomorphism from any setoid:

```
● : Setoid
●   = record {
  Carrier = ⊤;
  _≈_   = λ _ _ → ⊤;
  refl     = tt;
  sym     = λ _ → tt;
  trans   = λ _ _ → tt }

⋆ : {Δ : Setoid} → Δ ⇉ ●
⋆ = record
  { fn = λ _ → tt
  ; resp = λ _ → tt }

uniqueHom : ∀ (Δ : Setoid)
      → (f : Δ ⇉ ●) → f ≡ ⋆
uniqueHom Δ f = PE.refl
```

Because we do not use a categorical construction to build the "categories with families", we do not verify that it forms a setoid here.

6.3 Categories with families

The setoid model is essentially a category with families:

Definition 6.2. ***Categories with families.***

- *A category* C *with a terminal object.*
- *A functor* $F : \mathsf{C}^{op} \to Fam$*. Fam is a category of families whose objects are pairs* (A, A') *where* A *is a set and* A' *is a family of sets indexed over* A*. Morphisms are pairs of functions* (f, f') *such that, for any* $a : A$ *and* $a' : A'(a)$*, we have* $f(a) : B$ *and* $f'(a') : B'(f(a))$*.*
- *A comprehension of* Γ *and* $A : Ty\ \Gamma$*, written as* Γ, A *(or* $\Gamma \& A$*), is a construction of a new object in* C *which expresses the extension of contexts.*

Usually we think of the objects of the category C as contexts and morphisms as substitutions. The types and terms are projections of the functor F : given an object (context) $\Gamma : \mathsf{C}$, we usually write $F(\Gamma) :\equiv \Sigma(A : \mathrm{Ty}\ \Gamma)\ \mathrm{Tm}\ \Gamma\ A$, and the substitution of types and terms are just contained in the morphism part of this functor.

In the setoid model, the category of contexts is just **Std**,

```
Con = Setoid
```

Given a context Γ, types over it Ty Γ can be defined as functors from Γ to **Std** because types are interpreted as setoids and morphisms between setoids are functors. However setoids here are not implemented as categories, so we build a semantic type $A : \mathsf{Ty}\ \Gamma$ (a functor) as follows:

```
record Ty (Γ : Con) : Set₁ where
  field
    fm    : | Γ | → Con
    substT : {x y : | Γ |} →
      .([ Γ ] x ≈ y) →
      | fm x | →
      | fm y |
    .subst* : ∀{x y : | Γ |}
      (p : ([ Γ ] x ≈ y))
      {a b : | fm x |} →
      .([ fm x ] a ≈ b) →
      ([ fm y ] substT p a ≈ substT p b)

    .refl*   : ∀{x : | Γ |}{a : | fm x |} →
      [ fm x ] substT ([ Γ ]refl) a ≈ a
    .trans* : ∀{x y z : | Γ |}
      {p : [ Γ ] x ≈ y}
      {q : [ Γ ] y ≈ z}
      (a : | fm x |) →
      [ fm z ] substT q (substT p a)
          ≈ substT ([ Γ ]trans p q) a

  .tr* : ∀{x y : | Γ |}
    {p : [ Γ ] y ≈ x}
    {q : [ Γ ] x ≈ y}
    {a : | fm x |} →
    [ fm x ] substT p (substT q a) ≈ a
  tr* = [ fm _ ]trans (trans* _) refl*

  substT-inv : {x y : | Γ |} →
      .([ Γ ] x ≈ y) →
      | fm y | →
      | fm x |
    substT-inv p y = substT ([ Γ ]sym p) y
```

fm is the object part of this functor, substT is the morphism part which stands for substitution via an equivalence $x \sim y$ for $x, y : \Gamma$. subst∗ states that the functions between setoids preserve the equivalence relation. refl∗ and trans∗ are functor laws up to the equivalence relation. We also prove a lemma tr∗ which can be understood as the property that given arbitrary morphisms $p : y \sim x$ and $q : x \sim y$, the composition of them always equal to the identity morphism. substT-inv just gives the inverse of substT.

Notice that we mark all occurrences of $\sim$ irrelevant. We also omit some unnecessary syntactic renaming of the fields.

Then, terms follow naturally as families of elements in the underlying set of types indexed by $x : \Gamma$, and they have to respects the equivalent relation as well:

```
record Tm {Γ : Con}(A : Ty Γ) : Set where
  constructor tm:_resp:_
  field
    tm    : (x : | Γ |) → | [ A ]fm x |
    .respt : ∀ {x y : | Γ |} →
           (p : [ Γ ] x ≈ y) →
           [ [ A ]fm y ] [ A ]subst p (tm x) ≈ tm y
```

The substitution of types can be defined simply by composing the underlying objects of types and context morphisms:

```
_[_]T : ∀ {Γ Δ : Setoid} → Ty Δ → Γ ⇉ Δ → Ty Γ
_[_]T {Γ} {Δ} A f
  = record
  { fm     = λ x → fm (fn x)
  ; substT = λ p → substT _
  ; subst* = λ p → subst* (resp p)
  ; refl*   = refl*
  ; trans* = trans*
  }
  where
    open Ty A
    open _⇉_ f
```

refl∗ and trans∗ can also be verified easily because of proof irrelevance. We simplify our definition by opening two record types which are not ambiguous in the scope.

The substitution of terms is similar:

```
_[_]m : ∀ {Γ Δ : Con}{A : Ty Δ} → Tm A
  → (f : Γ ⇉ Δ) → Tm (A [ f ]T)
_[_]m t f = record
  { tm = [ t ]tm ∘ [ f ]fn
  ; respt = [ t ]respt ∘ [ f ]resp
  }
```

The empty context is just the terminal object of **Std** as we have seen before.

Given a context Γ and a type A : Ty Γ, we can form a new context $\Gamma \& A$ which is usually called **context comprehension**. Syntactically, it corresponds to introducing a new variable of type A. We can simply construct it with a Σ-type.

```
_&_ : (Γ : Setoid) → Ty Γ → Setoid
Γ & A =
  record { Carrier = Σ[ x : | Γ | ] | fm x |
  ; _≈_ = λ{(x , a) (y , b) →
    Σ[ p : x ≈ y ] [ fm y ] (substT p a) ≈ b }
  ; refl = refl , refl*
```

```
    ; sym =   λ {(p , q) → (sym p) ,
      [ fm _ ]trans (subst* _ ([ fm _ ]sym q)) tr* }
    ; trans = λ {(p , q) (m , n) → trans p m ,
    [ fm _ ]trans ([ fm _ ]trans
    ([ fm _ ]sym (trans* _)) (subst* _ q)) n}
    }
    where
      open Setoid Γ
      open Ty A
```

The new relation is also an equivalence which follows from the properties of Γ as a setoid and the properties of A. Since the context Γ and type A as record types are opened in the scope, we can unambiguously use fields such as fm and $subst*$.

We have also defined a few common operations as usual, e.g. projections and pairing. The code of them can be found in Appendix B.

6.3.1 Type construction in the setoid model

Dependent function types (i.e. Π-types) and dependent product types (i.e. Σ-types) are essential in a dependent type theory. Intuitively, they are just Π-types and Σ-types in the metatheory together with the proofs that the setoid equivalence is respected. We have implemented them according to the original construction and reasoning in [3] with minor adaptation. For example, given a type A in Γ and a type B in $\Gamma\&A$, we define Π A B as a type in Γ. The elements of Π-types are dependent functions which respect the equivalence relation.

```
Π : {Γ : Setoid}(A : Ty Γ)(B : Ty (Γ & A)) → Ty Γ
Π {Γ} A B = record
  { fm = λ x → let Ax = [ A ]fm x in
  let Bx = λ a → [ B ]fm (x , a) in
  record
```

```
{ Carrier = Subset ((a : | Ax |) → | Bx a |) (λ fn →
    (a b : | Ax |)
    (p : [ Ax ] a ≈ b) →
    [ Bx b ] [ B ]subst ([ Γ ]refl ,
    [ Ax ]trans [ A ]refl* p) (fn a) ≈ fn b)
```

The associated equality is pointwise equality of functions. To prove that it is an equivalence relation, we can simply exploit the corresponding rules of the equivalence relation within the type B.

```
; _≈_    = λ{(f , _) (g , _) → ∀ a → [ Bx a ] f a ≈ g a }
; refl   = λ a → [ Bx _ ]refl
; sym    = λ f a → [ Bx _ ]sym (f a)
; trans  = λ f g a → [ Bx _ ]trans (f a) (g a)
}
```

For the rest of the construction we just follow Altenkirch's work in [3] and keep them in the appendix (see Appendix B).

We also construct some basic types that appeared in Altenkirch's work, e.g. a simply typed universe and equality types. Since they have been discussed in [3], we just omit them here and focus on the more important one – the construction of quotient types.

6.3.2 Quotient types

We build our quotient types in an Agda module. Given a context Γ, and a type $A : \mathsf{Ty}\ \Gamma$,

```
module Q (Γ : Con)(A : Ty Γ)
```

we can build a quotient type if we have an equivalence relation R defined on A which has to respect the underlying equivalence of A. In principle, the type of R should be Tm $(\Pi\ (a : A\ \Pi\ A^+\ \textbf{Prop})$ where $A^+ :\equiv A\ [\mathrm{fst}]$ and fst corresponds to weakening. However we can not define an object-level **Prop** because our definition of setoids does not allow universes as underlying sets, and there is no universe **Prop** in meta-theory as well.

We declare the object part and properties of the relation explicitly. As long as we can define R properly, we can extract objects and properties of R so that this definition of quotient types still works.

The object part of R is a family of binary relation,

```
(R : (γ : | Γ |) → | [ A ]fm γ | → | [ A ]fm γ | → Set)
```

which should be proof-irrelevant. Therefore, the internal equality of the result type should be logical equivalence, hence the respT property can be interpreted as: for any $(\gamma, \gamma' : |\Gamma|)$ such that $(p : \gamma \approx_\Gamma \gamma')$, and $(a, b : |A_{fm}(\gamma)|)$, we have a logical equivalence

$$R((A_{subst}(p,a)),(A_{subst}(p,b))) \iff R_\gamma(a,b)$$

Here we only use one direction of this equivalence:

```
.(Rrespt : ∀{γ γ' : | Γ |}
  (p : [ Γ ] γ ≈ γ')
  (a b : | [ A ]fm γ |) →
  .(R γ a b) →
  R γ' ([ A ]subst p a) ([ A ]subst p b))
```

Of course, because it is defined on the type A, it has to respect equality (equivalence) of A.

```
.(Rrsp : ∀ {γ a b} → .([ [ A ]fm γ ] a ≈ b) → R γ a b)
```

It is an equivalence relation, so we have reflexivity, symmetry and transitivity.

```
.(Rref : ∀ {γ a} → R γ a a)
.(Rsym : (∀ {γ a b} → .(R γ a b) → R γ b a))
.(Rtrn :  (∀ {γ a b c} → .(R γ a b)
  →   .(R γ b c) → R γ a c))
```

The quotient type Q shares the same underlying set with A, but the internal equality is replaced by R.

```
⟦Q⟧₀ : | Γ | → Setoid
⟦Q⟧₀ γ = record
  { Carrier = | [ A ]fm γ |
  ; _≈_ = R γ
  ; refl = Rref
  ; sym = Rsym
  ; trans = Rtrn
  }
```

The underlying substitution is the same and we can easily verify the properties of R.

```
⟦Q⟧ : Ty Γ
⟦Q⟧ = record
  { fm = ⟦Q⟧₀
  ; substT = [ A ]subst
```

```
    ; subst* = λ p q → Rrespt p _ _ q
    ; refl* = Rrsp [ A ]refl*
    ; trans* = λ a → Rrsp ([ A ]trans* _)
    }
```

Given a term of A, we can introduce a term of Q.

```
⟦[_]⟧ : Tm A → Tm ⟦Q⟧
⟦[ x ]⟧ = record
    { tm = [ x ]tm
    ; respt = λ p → Rrsp ([ x ]respt p)
    }
```

We can also define a function between type A and Q inside the model.

```
⟦[_]⟧' : Tm (A ⇒ ⟦Q⟧)
⟦[_]⟧' = record
    { tm = λ x → (λ a → a) ,
       (λ a b p →
       Rrsp ([ [ A ]fm _ ]trans [ A ]refl* p))
    ; respt = λ p a → Rrsp [ A ]tr*
    }
```

Q-**Ax** can be simply proved because the new equivalence R respects the old one in A:

```
.Q-Ax : ∀ γ a b → [ [ A ]fm γ ] a ≈ b → [ [ ⟦Q⟧ ]fm _ ] a ≈ b
Q-Ax γ a b = Rrsp
```

The elimination rule and induction principle for quotient types are also straightforward. Given a function $f : A \to B$ which respects R, we can lift it as a function of type $Q \to B$ whose underlying function is the same as f. Because it respects R, the lifted function is well-typed. Since we still use the same substitution of A in the definition of Q, the respt property automatically holds.

```
Q-elim : (B : Ty Γ)(f : Tm (A ⇒ B))
   (frespR : ∀ γ a b → (R γ a b)
      → [ [ B ]fm γ ] prj₁ ([ f ]tm γ) a
         ≈   prj₁ ([ f ]tm γ) b)
→ Tm (⟦Q⟧ ⇒ B)
Q-elim B f frespR = record
   { tm = λ γ → prj₁ ([ f ]tm γ) , (λ a b p →
      [ [ B ]fm _ ]trans [ B ]refl* (frespR _ _ _ p))
   ; respt = λ {γ} {γ'} p a → [ f ]respt p a
   }
```

To prove the inductive principle, first we have to define a substitution which allows us to apply a variable to a predicate $P : Q \to \textbf{Set}$ in the form of $P([a])$:

```
substQ : (Γ & A) ⇉ (Γ & ⟦Q⟧)
substQ = record
   { fn = λ {(x , a) → x , a}
   ; resp = λ{ (p , q) → p , (Rrsp q)}
   }
```

Given P as a predicate on Q, we assume the result type of P is propositional, i.e. all terms of the underlying set is equivalent. h is a dependent function, or we can say it is a proof that for all $a : A$, $P([a])$ holds. Similar to elimination rule, we still use the same function h in the lifted version. The assumption we made about P helps us to prove that h is well-typed. The respect property is also inherited.

```
Q-ind : (P : Ty (Γ & ⟦Q⟧))
→ (isProp : ∀ {x a} (r s : | [ P ]fm (x , a) |) →
   [ [ P ]fm (x , a) ] r ≈ s )
→ (h : Tm (Π A (P [ substQ ]T)))
→ Tm (Π ⟦Q⟧ P)
Q-ind P isProp h = record
{ tm = λ x → (prj₁ ([ h ]tm x)) ,
   (λ a b p → isProp {x} {b} _ _)
; respt = [ h ]respt
}
```

6.4 Related work

Barthe, Capretta and Pons [16] have considered different definitions of setoids, and possible mathematical construction using setoids. The definition of a setoid we used is called a total setoid, while if the internal relation is not required to be reflexive, it is called a partial setoid. They have discussed quotients realisation using different approaches of setoids. Palmgren and Wilander [75] have also shown a formalisation of constructive set theory in terms of setoids in Intensional Type Theory. They have considered it as a solution to the problem that the uniqueness of identity proofs for sets are not derivable from J eliminator.

The categories with families (CwFs) ware introduced by Dybjer [38] as a model of dependent types which can be defined in Intensional Type Theory. Hofmann [50] has also explained the categorical semantics of dependent types provided by CwFs. Clairambault [27] has shown that categories with families are locally Cartesian closed after some additional structures are added such as Π-types, Σ-types, identity types etc.

In [46] Hofmann has discussed building a model of dependent type theory as categories with attributes from a locally Cartesian closed category (LCCC), for example the E-category of setoids (see Theorem 6.1). In that interpretation every morphism gives rise to a function. The E-category of setoids is different to the one used in our model which is not lccc. Hofmann [47, 48] has also proposed a setoid model where types are interpreted as partial setoids. It is built in Intensional Type Theory with a type of propositions **Prop** and a type Prf(P) for each P : **Prop**. He has provided interpretations of both propositions and quotient types with a choice operator. He has also proposed a groupoid model [48, 51] to interpret type dependency which does not exist in his setoid model. It can be seen as a setoid whose relation $\sim$ becomes proof-relevant, or more precisely $a \sim b$ is a set for each $a, b : A$, hence we lose UIP and K eliminator. However the groupoid model uses Extensional Type Theory as meta-theory.

6.5 Summary

In this chapter we have seen an implementation of Altenkirch's setoid model with a slight difference in the metatheory. We have used Agda's irrelevance feature to imitate the proof-irrelevant universe of propositions **Prop**. As we have seen it has a problem which has to be fixed by a postulate. It does not affect most of the implementation and we do not lose canonicity because the postulate is irrelevant so that we cannot construct natural numbers using it. We have implemented the model as a category with families and have introduced various types in it. Most importantly, we have shown that to define quotient types in this model, we can simply replace the internal equivalence of a type A as a setoid with a given equivalence relation on it. The original constributions of this work are the implementation of Altenkirch's setoid model in Agda and the extension with quotient types.

We can further simplify the construction of the setoid model by adopting McBride's heterogeneous approach to equality as discussed in Altenkirch, McBride and Swierstra's *Observational Type Theory* [8]. They identify values up to observation rather than construction which is called **observational equality**. It is the propositional equality induced by the setoid model. In general we have a heterogeneous equality which allows us to compare terms of different types. It can only be inhabited if the types are equal. In Agda, it can be defined as

```
data _≅_ {A : Set} (x : A) : ∀{B : Set} → B → Set where
  refl : x ≅ x
```

However, by defining equality irrelevant with the actual proof of the equality between types, we silently claim that the types are essentially sets which have UIP. Therefore if we do not accept K or UIP, we cannot use it in general. However we can use heterogeneous equality for types which actually *are* sets, which helps us avoid the heavy use of subst. This is fortunate, as subst complicates formalisation and reasoning. For example, we have used this in Section 7.1.2 for syntactic terms.

Chapter 7

Syntactic ω-groupoids

As we have seen in Chapter 6, a type can be interpreted as a setoid and its equivalence proofs, i.e. reflexivity, symmetry and transitivity, are unique. However in Homotopy Type Theory, we reject the principle of uniqueness of identity proofs (UIP). Instead we accept the *univalence axiom* proposed by Voevodsky (see Section 2.6.3) which says that equality of types is weakly equivalent to *weak equivalence* (see Section 2.6.2). It can be viewed as a strong extensionality axiom and it does imply functional extensionality. However, adding univalence as an axiom destroys canonicity, i.e. that every closed term of type $\mathbb{N}$ is reducible to a numeral. In the special case of extensionality and assuming a strong version of UIP Altenkirch and McBride were able to eliminate this issue [3, 8] using setoids. However, it is not clear how to generalize this in the absence of UIP to univalence which is incompatible with UIP. To solve the problem we should generalise the notion of setoids, namely to enrich the structure of the identity proofs.

The generalised notion is called *weak ω-groupoid* (see Section 2.6.2) and was proposed by Grothendieck 1983 in a famous manuscript *Pursuing Stacks* [44]. Maltsiniotis continued his work and suggested a simplification of the original definition which can be found in [63]. Later Ara also presents a slight variation of the simplification of weak ω-groupoids in [12]. Categorically speaking an ω-groupoid is an ω-category in which morphisms on all levels are equivalences. As we know that a set can be seen as a discrete category, a setoid is a category where every morphism between any two objects is unique. A groupoid is more generalised, every morphism is an isomorphism but the proof of isomorphism is unique, namely the composition of a morphism with its inverse is equal to the identity. Similarly, an n-groupoid is an n-category in which morphisms on all levels are equivalences. weak ω-groupoid (also called ∞-groupoid) is an infinite version of n-groupoid.

To model Type Theory without UIP we also allow the equalities to be non-strict, in other words, they are propositional but not necessarily definitional equalities. Finally we should use weak ω-groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak ω-groupoids in Type Theory, for instance, Altenkirch and Rypáček [7], and Brunerie's notes [24].

In this chapter, our implementation of weak ω-groupoids builds on the syntactic approach of [7] but simplifies it greatly following Brunerie's proposal [24] by replacing the distinct constants for each of the higher coherence cells by a single constant coh. In more detail, we specify when a globular set is a weak ω-groupoid by first defining a type theory called $\mathcal{T}_{\infty-groupoid}$ to describe the internal language of Grothendieck weak ω-groupoids, then interpret it with a globular set and a dependent function to it. All coherence laws of weak ω-groupoids are derivable from the syntax, we will present some basic ones, for example reflexivity. Everything is formalised in Agda. This is the first attempt to formalise this approach in a dependently typed language like Agda or Coq. Most of the work has been published in [11] by the author, Altenkirch and Rypáček.

One of our main contributions is to use heterogeneous equality for terms to overcome difficult problems encountered when using the usual homogeneous one. We present the formalisation but omit some complicated and less important programs, namely the proofs of some lemmas or definitions of some auxiliary functions. For the reader who is interested in the details, you can find the complete code in Appendix C and also online [60].

7.1 Syntax of weak ω-groupoids

We develop the type theory of ω-groupoids formally, following [24]. This is a type theory with only one type former which we can view as equality type and interpret as the homset of the ω-groupoid. There are no definitional equalities, this corresponds to the fact that we consider *weak* ω-groupoids. None of the groupoid laws on any levels are strict (i.e. definitional) but all are witnessed by terms. Compared to [7] the definition is greatly simplified by the observation that all laws of a weak ω-groupoid follow from the existence of coherence constants for any contractible context.

In our formalisation we exploit the more liberal way to do mutual definitions in Agda, which was implemented following up a suggestion by the Altenkirch. It allows us to first introduce a type former but give its definition later.

Since we are avoiding definitional equalities, we have to define a syntactic substitution operation which we need for the general statement of the coherence constants. However, defining these constants requires us to prove a number of substitution laws which with the usual definition of identity types take a very complex mutually recursive form (see [7]). We address this issue by using heterogeneous equality [69]. Although it exploits UIP, our approach is sound because UIP holds for the syntax. See Section 7.1.2 for more details.

7.1.1 Basic Objects

We first declare the syntax of our type theory which is called $\mathcal{T}_{\infty-groupoid}$ namely the internal language of weak ω-groupoids. Since the definitions of syntactic objects involve each other, it is essential to define them in an inductive-inductive way. Agda allows us to state the types and constructors separately for involved inductive-inductive definitions. The following declarations in order are contexts as sets, types are sets dependent on contexts, terms and variables are sets dependent on types, context morphisms and contractible contexts.

```
data Con          : Set
data Ty (Γ : Con)   : Set
data Tm             : {Γ : Con}(A : Ty Γ) → Set
data Var            : {Γ : Con}(A : Ty Γ) → Set
data _⇒_            : Con → Con → Set
data isContr        : Con → Set
```

Contexts are inductively defined. The base case is an empty context ϵ, and given a type A in a context Γ we can extend Γ with A written as Γ, A:

```
data Con where
  ε     : Con
  _,_   : (Γ : Con)(A : Ty Γ) → Con
```

Types are defined as either $*$ which we call 0-cells, or a equality type between two terms of some type A. If the type A is an n-cell then we call its equality type an $(n+1)$-cell. For example, for a set $\mathbb{N}$, $*$ is just the same as $\mathbb{N}$ and there are no higher cells because none of any two elements in $\mathbb{N}$ are equal.

```
data Ty Γ where
  *       : Ty Γ
  _=h_    : {A : Ty Γ}(a b : Tm A) → Ty Γ
```

7.1.2 Heterogeneous Equality for Terms

One of the big challenges we encountered was the difficulty to formalise and reason about the equalities of terms, which is essential when defining substitution. When the usual homogeneous identity types are used one has to use substitution to unify the types on both sides of equality types. This results in *subst* to appear in terms, about which one has to state substitution lemmas. This further pollutes syntax requiring lemmas about lemmas, lemmas about lemmas about lemmas, etc. For example, we have to prove that using *subst* consecutively with two equalities of types is propositionally equal to using *subst* with the composition of these two equalities. As the complexity of the proofs grows more lemmas are needed. The resulting recurrence pattern has been identified and implemented in [7] for the special cases of coherence cells for associativity, units and interchange. However it is not clear how that approach could be adapted to the present, much more economical formulation of weak ω-groupoids. Moreover, the complexity brings the Agda type checker to its limits and correctness into question.

The idea of heterogeneous equality (or JM equality) due to McBride [69] used to resolve this issue is to define equality for terms of different types which are supposed to be propositionally equal.

```
data _≅_ {Γ : Con}{A : Ty Γ} :
    {B : Ty Γ} → Tm A → Tm B → Set where
  refl : (b : Tm A) → b ≅ b
```

Notice that it only inhabits if A and B are computationally equal. It is actually proof-irrelevant on the equality $A = B$, namely the elimination rule of it relies on UIP. As we know in Intensional Type Theory, UIP is not provable in general, namely not all types are h-sets (homotopy 0-types) and indeed we did not assume UIP for all types by adding the special case of heterogeneous equality. It only requires that $Ty\ \Gamma$ to be an h-set. In Intensional Type Theory, It is a folklore that inductive types with finitary constructors have decidable equality. In our case, the types which stand for syntactic objects (contexts, types, terms) are all inductive-inductive types with finitary constructors. It follows by Hedberg's Theorem [45] that any type with decidable equality is an h-set, satisfies UIP and it therefore follows that the syntax satisfies UIP. Because, the equality of syntactic types is unique, it is safe to use heterogeneous equality for terms and proceed without using substitution lemmas which would otherwise be necessary to match terms of different types. From a computational perspective, it means that every equality of types can be reduced to *refl* and using *subst* to construct terms is proof-irrelevant, which is expressed in the following definition of heterogeneous equality for terms.

Once we have heterogeneous equality for terms, we can define a proof-irrelevant substitution which we call *coercion* since it gives us a term of type A if we have a term of type B and the two types are equal. We can also prove that the coerced term is heterogeneously equal to the original term. Combining these definitions, it is much more convenient to formalise and reason about term equations.

```
_⟦_⟫         : {Γ : Con}{A B : Ty Γ}(a : Tm B)
              → A ≡ B → Tm A
a ⟦ refl ⟫    = a

cohOp         : {Γ : Con}{A B : Ty Γ}{a : Tm B}(p : A ≡ B)
              → a ⟦ p ⟫ ≅ a
cohOp refl    = refl _
```

7.1.3 Substitutions

In this chapter we usually define a set of functions together and we name a function x as xC for contexts, xT for types, xV for variables xtm for terms and xS for context morphisms (substitutions) as conventions. For example the substitutions are declared as follows:

```
_[_]T    : ∀{Γ Δ} → Ty Δ → Γ ⇒ Δ → Ty Γ
_[_]V    : ∀{Γ Δ A} → Var A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_[_]tm   : ∀{Γ Δ A} → Tm A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
```

Indeed, compositions of context morphisms can be understood as substitutions for context morphisms as well.

```
_⊚_ : ∀{Γ Δ Θ} → Δ ⇒ Θ → (δ : Γ ⇒ Δ) → Γ ⇒ Θ
```

Context morphisms are defined inductively similarly to contexts. A context morphism is a list of terms corresponding to the list of types in the context on the right hand side of the morphism.

```
data _⇒_ where
  •      : ∀{Γ} → Γ ⇒ ε
  _,_    : ∀{Γ Δ}(δ : Γ ⇒ Δ){A : Ty Δ}(a : Tm (A [ δ ]T))
           → Γ ⇒ (Δ , A)
```

7.1.4 Weakening

We can freely add types to the contexts of any given type judgements, term judgements or context morphisms. These are the weakening rules.

```
_+T_    : ∀{Γ}(A : Ty Γ)(B : Ty Γ) → Ty (Γ , B)
_+tm_   : ∀{Γ A}(a : Tm A)(B : Ty Γ) → Tm (A +T B)
_+S_    : ∀{Γ Δ}(δ : Γ ⇒ Δ)(B : Ty Γ) → (Γ , B) ⇒ Δ
```

7.1.5 Terms

A term can be either a variable or a coherence constant (coh).

We first define variables separately using the weakening rules. We use typed de Bruijn indices to define variables as either the rightmost variable of the context, or some variable in the context which can be found by cancelling the rightmost variable along with each vS.

```
data Var where
  v0 : ∀{Γ}{A : Ty Γ}   → Var (A +T A)
```

```
    vS : ∀{Γ}{A B : Ty Γ}(x : Var A) → Var (A +T B)
```

The coherence constants are the most important and contentious issue of weak ω-groupoids. In this syntactic approach, they are primitive terms of the primitive types in *contractible contexts* which will be introduced below. Indeed it encodes the fact that any type in a contractible context is inhabited, and so are the types generated by substituting into a contractible context.

```
  data Tm where
    var   : ∀{Γ}{A : Ty Γ} → Var A → Tm A
    coh   : ∀{Γ Δ} → isContr Δ → (δ : Γ ⇒ Δ)
          → (A : Ty Δ) → Tm (A [ δ ]T)
```

7.1.6 Contractible contexts

With variables defined, it is possible to formalise another core part of the syntactic framework, *contractible contexts*. Intuitively speaking, a context is contractible if its geometric realization is contractible to a point. It either contains one variable of the type $*$ which is the base case, or we can extend a contractible context with a variable of an existing type and an n-cell, namely a morphism, between the new variable and some existing variable. Contractibility of contexts is defined as follows:

```
  data isContr where
    c*    : isContr (ε , *)
    ext   : ∀{Γ} → isContr Γ → {A : Ty Γ}(x : Var A)
          → isContr (Γ , A , (var (vS x) =h var v0))
```

Notice that ϵ is not contractible, otherwise $*$ is inhabited (all types in contractible context are inhabited) which is not true in all cases.

7.1.7 Lemmas

Since contexts, types, variables and terms are all mutually defined, most of their properties have to be proved simultaneously as well. Note that we are free to define all the types first and all the definitions (not shown) later.

The following lemmas are essential for the constructions and theorem proving later. The first set of lemmas states that to substitute a type, a variable, a term, or a context morphism with two context morphisms consecutively, is equivalent to substitute with the composition of the two context morphisms:

```
[⊚]T     : ∀{Γ Δ Θ A}{θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
         → A [ θ ⊚ δ ]T ≡ (A [ θ ]T)[ δ ]T

[⊚]v     : ∀{Γ Δ Θ A}(x : Var A){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
         → x [ θ ⊚ δ ]V ≅ (x [ θ ]V) [ δ ]tm

[⊚]tm    : ∀{Γ Δ Θ A}(a : Tm A){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
         → a [ θ ⊚ δ ]tm ≅ (a [ θ ]tm) [ δ ]tm

⊚assoc   : ∀{Γ Δ Θ Ω}(γ : Θ ⇒ Ω){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
         → (γ ⊚ θ) ⊚ δ ≡ γ ⊚ (θ ⊚ δ)
```

The second set states that weakening inside substitution is equivalent to weakening outside:

```
[+S]T    : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}
         → A [ δ +S B ]T ≡ (A [ δ ]T) +T B
```

```
[+S]tm   : ∀{Γ Δ A B}(a : Tm A){δ : Γ ⇒ Δ}
          → a [ δ +S B ]tm ≅ (a [ δ ]tm) +tm B

[+S]S    : ∀{Γ Δ Θ B}{δ : Δ ⇒ Θ}{γ : Γ ⇒ Δ}
          → δ ⊚ (γ +S B) ≡ (δ ⊚ γ) +S B
```

We can cancel the last term in the substitution for weakened objects since weakening doesn't introduce new variables in types and terms.

```
+T[,]T      : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{b : Tm (B [ δ ]T)}
             → (A +T B) [ δ , b ]T ≡ A [ δ ]T

+tm[,]tm    : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{c : Tm (B [ δ ]T)}
             → (a : Tm A)
             → (a +tm B) [ δ , c ]tm ≅ a [ δ ]tm
```

Most of the substitutions are defined as usual, except the one for coherence constants. In this case, we substitute in the context morphism part and one of the lemmas declared above is used.

```
var x         [ δ ]tm = x [ δ ]V
coh cΔ γ A    [ δ ]tm = coh cΔ (γ ⊚ δ) A ⟦ sym [⊚]T ⟫
```

7.2 Some Important Derivable Constructions

In this section we show how to reconstruct the structure of a (weak) ω-groupoid from the syntactical framework presented in Section 7.1 in the more explicit style of [7]. To this end, let us call a term a : Tm A an n-cell if level A ≡ n, where

```
level                  : ∀ {Γ} → Ty Γ → ℕ
level *                = 0
level (_=h_ {A} _ _)   = suc (level A)
```

In any ω-category, any n-cell a has a domain (source), $s^n_m\, a$, and a codomain (target), $t^n_m\, a$, for each $m \leq n$. These are, of course, (n-m)-cells. For each pair of n-cells such that for some m, $s^n_m a \equiv t^n_m b$, there must exist their composition $a \circ^n_m b$ which is an n-cell. Composition is (weakly) associative. Moreover for any (n-m)-cell x there exists an n-cell $\mathsf{id}^n_m\, \mathsf{x}$ which behaves like a (weak) identity with respect to $\circ^n_m$. For the time being we discuss only the construction of cells and omit the question of coherence.

For instance, in the simple case of bicategories, each 2-cell a has a horizontal source $s^1_1\, a$ and target $t^1_1\, a$, and also a vertical source $s^2_1\, a$ and target $t^2_1 a$, which is also the source and target, of the horizontal source and target, respectively, of a. There is horizontal composition of 1-cells $\circ^1_1$: $x \to^f y \to^g z$, and also horizontal composition of 2-cells $\circ^2_1$, and vertical composition of 2-cells $\circ^2_2$. There is a horizontal identity on a, $\mathsf{id}^1_1\, a$, and vertical identity on a, $\mathsf{id}^2_1\, a = \mathsf{id}^2_2 \mathsf{id}^1_1\, a$.

Thus each ω-groupoid construction is defined with respect to a *level*, m, and depth n-m and the structure of an ω-groupoid is repeated on each level. As we are working purely syntactically we may make use of this fact and define all groupoid structure only at level $m = 1$ and provide a so-called *replacement operation* which allows us to lift any cell to an arbitrary type A. It is called 'replacement' because we are syntactically replacing the base type $*$ with an arbitrary type, A.

An important general mechanism we rely on throughout the development follows directly from the type of the only non-trivial constructor of Tm, coh, which tells us that to construct a new term of type $\Gamma \vdash A$, we need a contractible context, Δ, a type $\Delta \vdash T$ and a context morphism $\delta : \Gamma \Rightarrow \Delta$ such that

$$\mathsf{T}\,[\,\delta\,]\mathsf{T} \equiv \mathsf{A}$$

Because in a contractible context all types are inhabited we may in a way work freely in Δ and then pull back all terms to A using δ. To show this formally, we must first define identity context morphisms which complete the definition of a *category* of contexts and context morphisms:

```
IdS : ∀{Γ} → Γ ⇒ Γ
```

It satisfies the following property:

```
IC-T : ∀{Γ}{A : Ty Γ} → A [ IdS ]T ≡ A
```

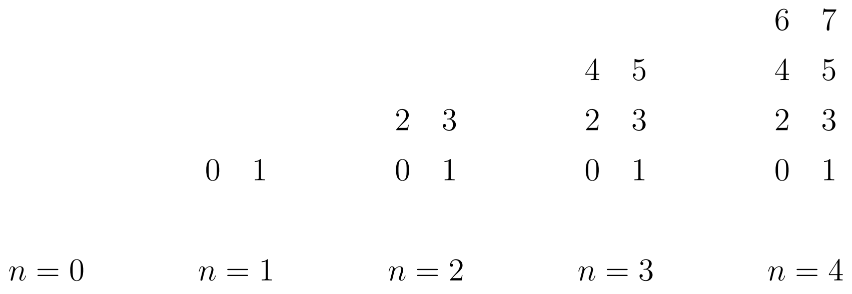
The definition proceeds by structural recursion and therefore extends to terms, variables and context morphisms with analogous properties. It allows us to define at once:

```
Coh-Contr       : ∀{Γ}{A : Ty Γ} → isContr Γ → Tm A
Coh-Contr isC   = coh isC IdS _ ⟦ sym IC-T ⟫
```

We use Coh-Contr as follows: for each kind of cell we want to define, we construct a minimal contractible context built out of variables together with a context morphism that populates the context with terms and a lemma that states an equality between the substitution and the original type.

7.2.1 Suspension and Replacement

For an arbitrary type A in Γ of level n one can define a context with $2n$ variables, called the *stalk* of A. Moreover one can define a morphism from Γ to the stalk of A such that its substitution into the maximal type in the stalk of A gives back A. The stalk of A depends only on the level of A, the terms in A define the substitution. Here is an example of stalks of small levels: ε (the empty context) for $n = 0$; $(x_0 : *, x_1 : *)$ for $n = 1$; $(x_0 : *, x_1 : *, x_2 : x_0 =_{\mathsf{h}} x_1, x_3 : x_0 =_{\mathsf{h}} x_1)$ for $n = 2$, etc.



This is the $\Delta = \varepsilon$ case of a more general construction where in we *suspend* an arbitrary context Δ by adding $2n$ variables to the beginning of it, and weakening the rest of the variables appropriately so that type $*$ becomes $x_{2n-2} =_{\mathsf{h}} x_{2n-1}$. A crucial property of suspension is that it preserves contractibility.

7.2.1.1 Suspension

Suspension is defined by iteration level-A-times the following operation of one-level suspension. ΣC takes a context and gives a context with two new variables of type $*$ added at the beginning, and with all remaining types in the context suspended by one level.

```
ΣC : Con → Con
ΣT : ∀{Γ} → Ty Γ → Ty (ΣC Γ)

ΣC ε          = ε , * , *
ΣC (Γ , A)    = ΣC Γ , ΣT A
```

The rest of the definitions are straightforward by structural recursion. In particular we suspend variables, terms and context morphisms:

```
Σv    : ∀{Γ}{A : Ty Γ} → Var A → Var (ΣT A)
```

```
Σtm   : ∀{Γ}{A : Ty Γ} → Tm A → Tm (ΣT A)
Σs    : ∀{Γ Δ} → Γ ⇒ Δ → ΣC Γ ⇒ ΣC Δ
```

The following lemma establishes preservation of contractibility by one-step suspension:

```
ΣC-Contr : ∀ Δ → isContr Δ → isContr (ΣC Δ)
```

It is also essential that suspension respects weakening and substitution:

```
ΣT[+T]    : ∀{Γ}(A B : Ty Γ)
            → ΣT (A +T B) ≡ ΣT A +T ΣT B

Σtm[+tm] : ∀{Γ A}(a : Tm A)(B : Ty Γ)
            → Σtm (a +tm B) ≅ Σtm a +tm ΣT B

ΣT[Σs]T   : ∀{Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)
            → (ΣT A) [ Σs δ ]T ≡ ΣT (A [ δ ]T)
```

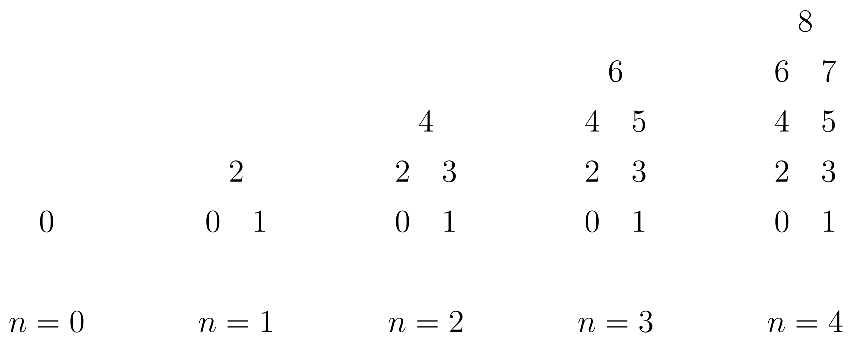
General suspension to the level of a type A is defined by iteration of one-level suspension. For symmetry and ease of reading the following suspension functions take as a parameter a type A in Γ, while they depend only on its level.

```
ΣC-it    : ∀{Γ}(A : Ty Γ) → Con → Con

ΣT-it    : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (ΣC-it A Δ)

Σtm-it   : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
         → Tm (ΣT-it A B)
```

Finally, it is clear that iterated suspension preserves contractibility.

```
ΣC-it-Contr   : ∀ {Γ Δ}(A : Ty Γ) → isContr Δ
              → isContr (ΣC-it A Δ)
```

By suspending the minimal contractible context, *, we obtain a so-called *span*. They are stalks with a top variable added. For example $(x_0 : *)$ (the one-variable context) for $n = 0$; $(x_0 : *, x_1 : *, x_2 : x_0 =_{\mathsf{h}} x_1)$ for $n = 1$; $(x_0 : *, x_1 : *, x_2 : x_0 =_{\mathsf{h}} x_1, x_3 : x_0 =_{\mathsf{h}} x_1, x_4 : x_2 =_{\mathsf{h}} x_3)$ for $n = 2$, etc. Spans play an important role later in the definition of composition. Following is a picture of the first few spans for increasing levels n of A.



7.2.1.2 Replacement

After we have suspended a context by inserting an appropriate number of variables, we may proceed to a substitution which, so to speak, fills the stalk for A with A. The context morphism representing this substitution is called filter. In the final step we combine it with Γ, the context of A. The new context contains two parts, the first is the same as Γ, and the second is the suspended Δ substituted by filter. However, we also have to drop the stalk of A because it already exists in Γ.

This operation is called *replacement* because we can interpret it as replacing $*$ in Δ by A.

As always, we define replacement for contexts, types and terms simultaneously:

```
rpl-C   : ∀{Γ}(A : Ty Γ) → Con → Con
rpl-T   : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (rpl-C A Δ)
rpl-tm  : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
          → Tm (rpl-T A B)
```

Replacement for contexts, rpl-C, defines for a type A in Γ and another context Δ a context which begins as Γ and follows by each type of Δ with $*$ replaced with (pasted onto) A.

```
rpl-C {Γ} A ε       = Γ
rpl-C A (Δ , B)     = rpl-C A Δ , rpl-T A B
```

To this end we must define the substitution filter which pulls back each type from suspended Δ to the new context.

```
filter  : ∀{Γ}(Δ : Con)(A : Ty Γ)
        → rpl-C A Δ ⇒ ΣC-it A Δ

rpl-T A B = ΣT-it A B [ filter _ A ]T
```

7.2.2 First-level Groupoid Structure

We can proceed to the definition of the groupoid structure of the syntax. We start with the base case: 1-cells. Replacement defined above allows us to lift this structure to an arbitrary level n (we leave most of the routine details out). This shows that the syntax is a 1-groupoid on each level. In the next section we show how also the higher-groupoid structure can be defined.

We start by an essential lemma which formalises the discussion at the beginning of this section: to construct a term in a type A in an arbitrary context, we first restrict attention to a suitable contractible context Δ and use lifting and substitution – replacement – to pull the term built by coh in Δ back. This relies on the fact that a lifted contractible context is also contractible, and therefore any type lifted from a contractible context is also inhabited.

```
Coh-rpl  : ∀{Γ Δ}(A : Ty Γ)(B : Ty Δ) → isContr Δ
           → Tm (rpl-T A B)
Coh-rpl {_} {Δ} A _ isC = coh (ΣC-it-ε-Contr A isC) _ _
```

Next we define the reflexivity, symmetry and transitivity terms of any type. Let us start from some base cases. Each of the base cases is derivable in a different contractible context with Coh-Contr which gives you a coherence constant for any type in any contractible context.

Reflexivity (identity) It only requires a one-object context.

```
refl*-Tm : Tm {x:*} (var v0 =h var v0)
refl*-Tm = Coh-Contr c*
```

Symmetry (inverse) It is defined similarly. Note that the intricate names of contexts, as in Ty x:*,y:*,α:x=y indicate their definitions which have been hidden. Agda treats all sequences of characters uninterrupted by whitespace as identifiers. For instance x:*,y:*,α:x=y is a name of a context for which we are assuming the definition: x:*,y:*,α:x=y = ε , * , * , (var (vS v0) =h var v0).

```
sym*-Ty : Ty x:*,y:*,α:x=y
sym*-Ty = vY =h vX

sym*-Tm : Tm {x:*,y:*,α:x=y} sym*-Ty
sym*-Tm = Coh-Contr (ext c* v0)
```

Transitivity (composition)

```
trans*-Ty : Ty x:*,y:*,α:x=y,z:*,β:y=z
trans*-Ty = (vX +tm _ +tm _) =h vZ

trans*-Tm : Tm trans*-Ty
trans*-Tm = Coh-Contr (ext (ext c* v0) (vS v0))
```

To obtain these terms for any given type in any give context, we use replacement.

```
refl-Tm     : {Γ : Con}(A : Ty Γ)
            → Tm (rpl-T {Δ = x:*} A (var v0 =h var v0))
refl-Tm A   = rpl-tm A refl*-Tm

sym-Tm      : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A sym*-Ty)
sym-Tm A    = rpl-tm A sym*-Tm

trans-Tm    : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A trans*-Ty)
trans-Tm A  = rpl-tm A trans*-Tm
```

For each of reflexivity, symmetry and transitivity we can construct appropriate coherence 2-cells witnessing the groupoid laws. The base case for variable contexts is proved simply using contractibility as well. However the types of these laws are not as trivial as the proving parts. We use substitution to define the application of the three basic terms we have defined above.

```
Tm-right-identity* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ IdS , vY , reflY ]tm
  =h vα)
Tm-right-identity* = Coh-Contr (ext c* v0)
```

```
Tm-left-identity* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ ((IdS ⊚ pr1 ⊚ pr1) , vX) ,
  reflX , vY , vα ]tm =h vα)
Tm-left-identity* = Coh-Contr (ext c* v0)

Tm-right-inverse* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ (IdS , vX) , sym*-Tm ]tm
  =h reflX)
Tm-right-inverse* = Coh-Contr (ext c* v0)

Tm-left-inverse* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ ((• , vY) , vX , sym*-Tm ,
  vY) , vα ]tm =h reflY)
Tm-left-inverse* = Coh-Contr (ext c* v0)

Tm-G-assoc*   : Tm Ty-G-assoc*
Tm-G-assoc*   = Coh-Contr (ext (ext (ext c* v0) (vS v0))
                (vS v0))
```

Their general versions are defined using replacement. For instance, for associativity, we define:

```
Tm-G-assoc      : ∀{Γ}(A : Ty Γ)
                → Tm (rpl-T A Ty-G-assoc*)
Tm-G-assoc A    = rpl-tm A Tm-G-assoc*
```

Following the same pattern, the n-level groupoid laws can be obtained as the coherence constants as well.

7.2.3 Higher Structure

In the previous text we have shown how to define 1-groupoid structure on an arbitrary level. Here we indicate how all levels also bear the structure of n-groupoid for arbitrary n. The rough idea amounts to redefining telescopes of [7] in terms of appropriate contexts, which are contractible, and the different constructors for terms used in [7] in terms of coh.

To illustrate this we consider the simpler example of higher identities. Note that the domain and codomain of $n{+}1$-iterated identity are n-iterated identities. Hence we proceed by induction on n. Denote a span of depth n S_n. Then there is a chain of context morphisms $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$. Each S_{n+1} has one additional variable standing for the identity iterated $n{+}1$-times. Because S_{n+1} is contractible, one can define a morphism $S_n \Rightarrow S_{n+1}$ using coh to fill the last variable and variable terms on the first n levels. By composition of the context morphisms one defines n new terms in the basic one variable context $*$ – the iterated identities. Finally, using suspension one can lift the identities to an arbitrary level.

Each n-cell has n-compositions. In the case of 2-categories, 1-cells have one composition, 2-cells have vertical and horizontal composition. Two 2-cells are horizontally composable only if their 1-cell top and bottom boundaries are composable. The boundary of the composition is the composition of the boundaries. Thus for arbitrary n we proceed using a chain of V-shaped contractible contexts. That is contexts that are two spans conjoined at the base level at a common middle variable. Each successive composition is defined using contractibility and coh.

To fully imitate the development in [7], one would also have to define all higher coherence laws. But the sole purpose of giving an alternative type theory in this chapter is to avoid that.

7.3 Semantics

7.3.1 Globular Types

To interpret the syntax, we need globular types [1] . Globular types are defined coinductively as follows:

```
record Glob : Set₁ where
  constructor _||_
  field
    |_|   : Set
    hom    : |_| → |_| → ∞ Glob
```

If all the object types (|_|) are indeed sets, i.e. UIP holds for them, we call this a globular set.

As an example, we could construct the identity globular type called Idω.

```
Idω     : (A : Set) → Glob
Idω A   = A || (λ a b → ♯ Idω (a ≡ b))
```

Given a globular type G, we can interpret the syntactic objects.

```
record Semantic (G : Glob) : Set₁ where
  field
    ⟦_⟧C    : Con → Set
    ⟦_⟧T    : ∀{Γ} → Ty Γ → ⟦ Γ ⟧C → Glob
    ⟦_⟧tm   : ∀{Γ A} → Tm A → (γ : ⟦ Γ ⟧C)
            → | ⟦ A ⟧T γ |
    ⟦_⟧S    : ∀{Γ Δ} → Γ ⇒ Δ → ⟦ Γ ⟧C → ⟦ Δ ⟧C
```

[1] The Agda Set stands for an arbitrary type, not a set in the sense of Homotopy Type Theory.

```
π          : ∀{Γ A} → Var A → (γ : ⟦ Γ ⟧C)
           → ∣ ⟦ A ⟧T γ ∣
```

π provides the projection of the semantic variable out of a semantic context.

Following are the computation laws for the interpretations of contexts and types.

```
⟦_⟧C-β1    : ⟦ ε ⟧C ≡ ⊤
⟦_⟧C-β2    : ∀ {Γ A} → ⟦ Γ , A ⟧C ≡
           Σ ⟦ Γ ⟧C (λ γ  → ∣ ⟦ A ⟧T γ ∣)

⟦_⟧T-β1    : ∀{Γ}{γ : ⟦ Γ ⟧C} → ⟦ * ⟧T γ ≡ G
⟦_⟧T-β2    : ∀{Γ A u v}{γ : ⟦ Γ ⟧C}
           → ⟦ u =h v ⟧T γ ≡
           ♭ (hom (⟦ A ⟧T γ) (⟦ u ⟧tm γ) (⟦ v ⟧tm γ))
```

Semantic substitution and semantic weakening laws are also required. The semantic substitution properties are essential for dealing with substitutions inside interpretation,

```
semSb-T     : ∀ {Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)(γ : ⟦ Γ ⟧C)
            → ⟦ A [ δ ]T ⟧T γ ≡ ⟦ A ⟧T (⟦ δ ⟧S γ)

semSb-tm    : ∀{Γ Δ}{A : Ty Δ}(a : Tm A)(δ : Γ ⇒ Δ)
            (γ : ⟦ Γ ⟧C) → subst ∣_∣ (semSb-T A δ γ)
            (⟦ a [ δ ]tm ⟧tm γ) ≡ ⟦ a ⟧tm (⟦ δ ⟧S γ)

semSb-S     : ∀ {Γ Δ Θ}(γ : ⟦ Γ ⟧C)(δ : Γ ⇒ Δ)
            (θ : Δ ⇒ Θ) → ⟦ θ ⊚ δ ⟧S γ ≡
            ⟦ θ ⟧S (⟦ δ ⟧S γ)
```

Since the computation laws for the interpretations of terms and context morphisms are well typed up to these properties.

```
⟦_⟧tm-β1  : ∀{Γ A}{x : Var A}{γ : ⟦ Γ ⟧C}
            → ⟦ var x ⟧tm γ ≡ π x γ

⟦_⟧S-β1  : ∀{Γ}{γ : ⟦ Γ ⟧C}
           → ⟦ • ⟧S γ ≡ coerce ⟦_⟧C-β1 tt

⟦_⟧S-β2  : ∀{Γ Δ}{A : Ty Δ}{δ : Γ ⇒ Δ}{γ : ⟦ Γ ⟧C}
           {a : Tm (A [ δ ]T)} → ⟦ δ , a ⟧S γ
           ≡ coerce ⟦_⟧C-β2 ((⟦ δ ⟧S γ) ,
           subst ∣_∣ (semSb-T A δ γ) (⟦ a ⟧tm γ))
```

The semantic weakening properties should actually be derivable since weakening is equivalent to projection substitution.

```
semWk-T   : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
            → ⟦ A +T B ⟧T (coerce ⟦_⟧C-β2 (γ , v)) ≡
            ⟦ A ⟧T γ


semWk-S   : ∀ {Γ Δ B}{γ : ⟦ Γ ⟧C}{v : ∣ ⟦ B ⟧T γ ∣}
            → (δ : Γ ⇒ Δ) → ⟦ δ +S B ⟧S
            (coerce ⟦_⟧C-β2 (γ , v)) ≡ ⟦ δ ⟧S γ

semWk-tm : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
            → (a : Tm A) → subst ∣_∣ (semWk-T γ v)
               (⟦ a +tm B ⟧tm (coerce ⟦_⟧C-β2 (γ , v)))
                 ≡ (⟦ a ⟧tm γ)
```

Here we declare them as properties because they are essential for the computation laws of function π.

```
π-β1  : ∀{Γ A}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ A ⟧T γ ∣)
      → subst ∣_∣ (semWk-T γ v)
        (π v0 (coerce ⟦_⟧C-β2 (γ , v))) ≡ v

π-β2  : ∀{Γ A B}(x : Var A)(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
      → subst ∣_∣ (semWk-T γ v) (π (vS {Γ} {A} {B} x)
        (coerce ⟦_⟧C-β2 (γ , v))) ≡ π x γ
```

The only part of the semantics where we have any freedom is the interpretation of the coherence constants:

```
⟦coh⟧  : ∀{Θ} → isContr Θ → (A : Ty Θ)
       → (θ : ⟦ Θ ⟧C) → ∣ ⟦ A ⟧T θ ∣
```

However, we also need to require that the coherence constants are well behaved with respect to substitution which in turn relies on the interpretation of all terms. To address this we state the required properties in a redundant form because the correctness for any other part of the syntax follows from the defining equations we have already stated. There seems to be no way to avoid this.

If the underlying globular type is not a globular set, we need to add coherence laws, which is not very well understood. On the other hand, restricting ourselves to globular sets means that our prime example Idω is not an instance anymore because the definition of our Idω do not have the conditions that every level is a set. We should still be able to construct non-trivial globular sets, e.g. by encoding basic topological notions and defining higher homotopies as in a classical framework. However, we do not currently know a simple definition of a globular set which is a weak ω-groupoid. One possibility would be to use the syntax of type theory with equality types. Indeed we believe that this would be an alternative way to formalize weak ω-groupoids.

Altenkirch also suggests a potential solution to fix the problem that our definition of Idω is not a globular set by using the approach discussed in [4]. we can define a universe with extensional equality, and use Agda's propositional equality as strict equality so that we can define Idω as a globular set in this universe.

7.4 Related work

The groupoid interpretation of Martin-Löf type theory was first proposed to Hofmann and Streicher [51]. Sozeau and Tabareau [77] have formalised it in Coq. They have also considered to generalise their definitions to ω-groupoids in the future. Warren [93] has shown an interpretation of Type Theory using *strict* ω-groupoids. Lumsdaine [62], van den Berg and Garner [84] have shown that J eliminator gives rise to a weak ω-groupoid, van den Berg and Garner have proved that that every type is a weak ω-groupoid. Altenkirch and Rypáček [7] have proposed a syntactic formalisation of weak ω-groupoids in Type Theory and a simplification of it has been suggested by Brunerie [24].

7.5 Summary

In this chapter, we have introduced an implementation of weak ω-groupoids following Brunerie's suggestion. Briefly speaking, we defined the syntax of the type theory $\mathcal{T}_{\infty-groupoid}$, then a weak ω-groupoid is a globular set with the interpretation of the syntax. To overcome some technical problems, we used heterogeneous equality for terms, some auxiliary functions and loop context in all implementation. We constructed the identity morphisms and verified some groupoid laws in the syntactic framework. The suspensions for all sorts of objects were also defined for other later constructions. In the future, we would like to formalise a proof that Idω is a weak ω-groupoid. As Altenkirch suggests, we can potentially solve the problem that our definition of Idω is not a globular set by using the approach discussed in [4]. Briefly speaking, we can define a universe with extensional equality, and use Agda's propositional equality as strict equality so that we can define Idω as a globular set in this universe. Finally the most challenging task would be to model Type Theory with weak ω-groupoids and to eliminate the univalence axiom.

Chapter 8

Conclusion and Future Work

We presented the evolution of type theories focusing on Martin-Löf type theory (Type Theory) and discussed different variants. We compared two versions of Type Theory: Extensional Type Theory (ETT) and Intensional Type Theory (ITT). ITT has decidable type checking but lacks some extensional concepts such as functional extensionality and quotient types. On the other hand, ETT has equality reflection which provides these extensional concepts but makes type checking undecidable due to the identification of propositional and definitional equalities.

The notion of quotient types is one of the important extensional concepts which facilitates mathematical and programming constructions. An interesting question is whether ITT could be extended with quotient types. We presented a definition of quotient types in a type theory with a proof-irrelevant universe, and we showed that simply adding the rules of quotient types to Intensional Type Theory as axioms results in the loss of the $\mathbb{N}$-canonicity property. We also clarified the correspondence with coequalizers in **Set** and a left adjoint functor in category theory.

We discussed the definability of a normalisation function for a given quotient represented as a setoid. For quotients which can be defined inductively with a normalisation function e.g. the set of integers and the set of rational numbers, we proposed an algebraic structure to bridge the gap between the setoid representations and the set definitions. We showed that the application of a definable quotient structure can improve the constructions by keeping good properties of both representations. As definable quotients can be seen as a simulation of quotient types, we expect similar benefits from using quotient types.

An interesting future project is the further development of the implementation of numbers in Agda using the definable quotient structure. It could be extended to other definable quotients implementable in our algebraic quotient structures. This would make the Agda standard library more convenient to use for mathematical applications. Another possibility is the extension of Agda with normalised types [34], that is, building a special case of quotient types with respect to a normalisation function in the sense of Definition 5.1.

Although a quotient type former is not necessary for definable quotients, it seems indispensable for some other quotients which don't have a definable normalisation function. With the assumption that Brouwer's continuity holds in the meta-theory, we proved that there is no definable normalisation function for Cauchy reals $\mathbb{R}_0/{\sim}$. Other examples include the partiality monad and finite multisets. In the future, we would like to investigate the definability of quotients in general, and in particular, we would like to find out whether the non-existence of a normalisation function for a quotient implies that it is not definable as a set in general.

A way of introducing quotient types in Intensional Type Theory without losing good computational properties is building models where types are interpreted as sets with an internally defined equality, such as setoids, groupoids or weak ω-groupoids. We have developed an implementation of Altenkirch's setoid model in Agda, and explained our construction of quotient types inside it.

There are more open research questions regarding the setoid model, for example the verification of certain properties or the definition of a type of propositions for which we can write the type of equivalence relations using Π-types. A simplification would be the usage of heterogeneous equality as discussed in Chapter 6. One could also consider the usage of h-propositions instead of a universe of propositions in the metatheory. However Π-closure of h-propositions needs functional extensionality. It would be interesting to compare this approach with the one we have presented. It is also worthwhile to extend the setoid model with examples of quotients like the set of real numbers and finite multisets which are not definable via normalisation. Other extensional concepts and coinductive types can also be considered in the setoid model.

We also investigated another extension of Martin-Löf type theory– Homotopy Type Theory. In Homotopy Type Theory, types are interpreted as weak ω-groupoids which are generalizations of groupoids. We discussed quotients in Homotopy Type Theory. With univalence, quotients can be defined impredicatively. We can also define quotients using higher inductive types (HITs), and in fact HITs can be seen as "generalized quotient types". Therefore a computational interpretation of Homotopy Type Theory can also be seen as a way of adding quotient types to Intensional Type Theory.

We showed a syntactic construction of weak ω-groupoids in Agda as a first step towards building a weak ω-groupoid model of Type Theory. We defined the type theory $\mathcal{T}_{\infty-groupoid}$ which describes the coherence conditions of a weak ω-groupoid required for a globular set. Inside this theory, we showed how to reconstruct some coherences laws, for example the groupoid laws using suspensions and replacement techniques. Here we also used heterogeneous equality for terms to ease implementation.

There are further interesting questions regarding our syntactic framework. For instance, we would like to investiage the relation between the $\mathcal{T}_{\infty-groupoid}$ and a type theory with equality types and the J eliminator which is called $\mathcal{T}_{eq}$. One direction is to simulate the J eliminator syntactically in $\mathcal{T}_{\infty-groupoid}$ as we mentioned before, the other direction is to derive J using coh if we can prove that the $\mathcal{T}_{eq}$ is a weak ω-groupoid. The syntax could be simplified by adopting categories with families. An alternative way may be the usage of higher inductive types to formalize the syntax of type theory.

When attempting to prove that Idω is a weak ω-groupoid, we encountered the problem that the base set in a globular set is an h-set which is incompatible with Idω. Altenkirch suggests [4] a solution using a universe with extensional equality, and Agda's propositional equality as strict equality so that we can define Idω as a globular set in this universe. Finally, modelling Type Theory with weak ω-groupoids and thus eliminating the univalence axiom would be the most challenging task to do in the future.

It would also be interesting to consider quotient *types* in Homotopy Type Theory. The notion of quotient types we considered in this thesis refers to the quotients with a *propositional* equivalence relation. However in a type theory with higher dimensions, like Homotopy Type Theory, the notion of quotient types can be more general and we would like to consider non-propositional quotients, for example, the quotient of a set by a groupoid.

Appendix A

Definable quotient structures

```
record Setoid : Set₁ where
  infix 4 _~_
  field
    Carrier       : Set
    _~_           : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _~_

  open IsEquivalence isEquivalence public
```

We first define the relation that "f respects $\sim$" (f is compatible with $\sim$)

```
_respects_ : {A : Set}{B : Set}(f : A → B)
  → (_~_ : A → A → Set) → Set
f respects _~_ = ∀ {a a'} → a ~ a' → f a ≡ f a'
```

Prequotient

```
record pre-Quotient (S : Setoid) : Set₁ where
  open Setoid S renaming (Carrier to A)
  field
    Q   : Set
    [_] : A → Q
    [_]⁼ : [_] respects _~_
```

We can assume UIP which will only be applied on quotient sets

```
≡prop : {A : Set}{a b : A} → (p q : a ≡ b) → p ≡ q
≡prop {A} {a} {.a} refl refl = refl

sublrr : {S : Set}{A : S → Set}{a b : S}(p q : a ≡ b){m : A a}
  → subst A p m ≡ subst A q m
sublrr p q with ≡prop p q
sublrr p .p | refl = refl

sublrr2 : {S : Set}{A : Set}{a b : S}(p : a ≡ b){m : A}
  → subst (λ _ → A) p m ≡ m
sublrr2 refl = refl
```

Quotient with dependent eliminator

```
record Quotient {S : Setoid}
  (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    qelim   : {B : Q → Set}
            → (f : (a : A) → B [ a ])
            → (∀ {a a'} → (p : a ~ a')
          → subst B [ p ]⁼ (f a) ≡ f a')
          → (q : Q) → B q
    qelim-β : ∀ {B a f}
          (resp : (∀ {a a'} → (p : a ∼ a')
          → subst B [ p ]⁼ (f a) ≡ f a'))
          → qelim {B} f resp [ a ] ≡ f a
```

Quotient (Hofmann's)

```
record Hof-Quotient {S : Setoid}
  (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    lift    : {B : Set}
            → (f : A → B)
            → f respects _∼_
            → Q → B

    lift-β : ∀ {B a f}(resp : f respects _∼_)
            → lift {B} f resp [ a ] ≡ f a

    qind    : ∀ (P : Q → Set)
            → (∀{x} → (p q : P x) → p ≡ q)
            → (∀ a → P [ a ])
            → (∀ x → P x)
```

```
record Hof-Quotient' {S : Setoid}
  (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
```

```
    lift   : {B : Set}
           → (f : A → B)
           → f respects _∼_
           → Q → B

    lift-β : ∀ {B a f}(resp : f respects _∼_)
           → lift {B} f resp [ a ] ≡ f a

    qind   : ∀ (P : Q → Set)
           → (∀{x} → (p q : P x) → p ≡ q)
           → (∀ a → P [ a ])
           → (∀ x → P x)
```

Exact quotient

```
record exact-Quotient {S : Setoid}
  (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    Qu    : Quotient PQ
    exact : ∀ {a b : A} → [ a ] ≡ [ b ] → a ∼ b
```

Definable quotient

```
record def-Quotient {S : Setoid}
      (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    emb      : Q → A
    complete : ∀ a → emb [ a ] ∼ a
```

```
        stable   : ∀ q → [ emb q ] ≡ q
```

Proof : Definable quotients are exact.

```
exact : ∀{a b} → [ a ] ≡ [ b ] → a ~ b
exact {a} {b} p =
  ~-trans (~-sym (complete a))
  (~-trans (subst (λ x →
  emb [ a ] ~ emb x)
  p ~-refl) (complete b))
```

Equivalences and conversions among the quotient structures

Proof : Hofmann's definition of quotient is equivalent to Quotient.

```
Hof-Quotient→Quotient : {S : Setoid}{PQ : pre-Quotient S} →
  (Hof-Quotient PQ) → (Quotient PQ)
Hof-Quotient→Quotient {S} {PQ} QuH =
  record
    { qelim   = λ {B} f resp
    → proj₁ (qelim' f resp)
    ; qelim-β = λ {B} {a} {f} resp
    → proj₂ (qelim' f resp)
    }
  where
    open pre-Quotient PQ
    open Hof-Quotient QuH

    qelim' : {B : Q → Set}
      → (f : (a : A) → B [ a ])
      → (∀ {a a'} → (p : a ~ a')
    → subst B [ p ]⁼ (f a) ≡ f a')
    → Σ[ f^ : ((q : Q) → B q) ]
          (∀ {a} → f^ [ a ] ≡ f a)
  qelim' {B} f resp =   f^ , f^-β
where

    f₀ : A → Σ Q B
    f₀ a = [ a ] , f a

    resp₀ : f₀ respects _~_
    resp₀ p = Σeq [ p ]⁼ (resp p)


    f' : Q → Σ Q B
    f' = lift f₀ resp₀

    id' : Q → Q
    id' = proj₁ ∘ f'

    P : Q → Set
    P q = id' q ≡ q

    f'-β : {a : A} → f' [ a ] ≡ [ a ] , f a
    f'-β = lift-β _

    isIda : ∀ {a} → id' [ a ] ≡ [ a ]
    isIda = cong proj₁ f'-β

    isIdq : ∀ {q} → id' q ≡ q
    isIdq {q} = qind P ≡prop (λ _ → isIda) q

    f^ : (q : Q) → B q
    f^ q = subst B isIdq (proj₂ (f' q))
```

```
      f'-sound2 : ∀ {a} →
        subst B isIda (proj₂ (f' [ a ])) ≡ f a
      f'-sound2 = cong-proj₂ _ _ f'-β

      f^-β : ∀ {a} → f^ [ a ] ≡ f a
      f^-β {a} = trans (subIrr isIdq isIda) f'-sound2



Quotient→Hof-Quotient :
  {S : Setoid}{PQ : pre-Quotient S}
  → (Quotient PQ)
  → (Hof-Quotient PQ)
Quotient→Hof-Quotient {S} {PQ} QU =
  record
  { lift   = λ f resp
    → qelim f (resp' resp)
  ; lift-β = λ resp
    → qelim-β (resp' resp)
  ; qind = λ P isP f
        → qelim {P} f (λ _ → isP _ _)
  }
  where
    open pre-Quotient PQ
    open Quotient QU

    resp' : {B : Set}{a a' : A}
      {f : A → B}
      (resp : f respects _~_)
      (p : a ~ a')
      → subst (λ _ → B) [ p ]⁼ (f a)
      ≡ f a'
    resp' resp p =
      trans (subIrr2 [ p ]⁼)
```

```
              (resp p)
```

Proof : A definable quotient gives rise to a *quotient*.

```
def-Quotient→Quotient :
  {S : Setoid}{PQ : pre-Quotient S}
  → (def-Quotient PQ) → (Quotient PQ)
def-Quotient→Quotient {S} {PQ} QuD =
  record { qelim =
    λ {B} f resp q → subst B (stable q) (f (emb q))
    ; qelim-β =
    λ {B} {a} {f} resp →
    trans (sublrr (stable [ a ])
    [ complete a ]⁼) (resp (complete a))

  }
  where
  open pre-Quotient PQ
  open def-Quotient QuD
```

Proof : A definable quotients gives rise to an *exact (effective) quotient*.

```
def-Quotient→exact-Quotient :
  {S : Setoid}{PQ : pre-Quotient S}
  → def-Quotient PQ → exact-Quotient PQ
def-Quotient→exact-Quotient {S} {PQ} QuD =
  record { Qu = def-Quotient→Quotient QuD
    ; exact = exact
    }
  where
  open pre-Quotient PQ
```

```
    open def-Quotient QuD



def-Quotient→Hof-Quotient
  : {S : Setoid}
  → {PQ : pre-Quotient S}
  → (def-Quotient PQ)
  → (Hof-Quotient PQ)
def-Quotient→Hof-Quotient {S} {PQ} QuD =
  record
  { lift    = λ f _ → f ∘ emb
  ; lift-β = λ resp → resp (complete _)
  ; qind    = λ P _ f _ →
              subst P (stable _) (f (emb _))
  }
  where
    open pre-Quotient PQ
    open def-Quotient QuD



def-Quotient→Hof-Quotient' :
  {S : Setoid}{PQ : pre-Quotient S}
  → (def-Quotient PQ) → (Hof-Quotient PQ)
def-Quotient→Hof-Quotient' =
  Quotient→Hof-Quotient ∘ def-Quotient→Quotient
```

Proof : The propositional univalence (propositional extensionality) implies that a quotient is always exact.

Assume we have the propositional univalence (the other direction trivial holds)

```
(PropUni₁ : ∀ {p q : Set} → (p ⇔ q) → p ≡ q)
{S : Setoid}
{PQ : pre-Quotient S}
{Qu : Hof-Quotient PQ}
  where
open pre-Quotient PQ
open Hof-Quotient Qu

coerce : {A B : Set} → A ≡ B → A → B
coerce refl m = m

exact : ∀ a a' → [ a ] ≡ [ a' ] → a ~ a'
exact a a' p = coerce P^-β (~-refl {a})
  where
    P : A → Set
    P x = a ~ x

    isEqClass : ∀ {a b} → a ~ b → P a ⇔ P b
    isEqClass p = (λ q → ~-trans q p) ,
      (λ q → ~-trans q (~-sym p))

    P-resp : P respects _~_
    P-resp p = PropUni₁ (isEqClass p)

    P^ : Q → Set
    P^ = lift P P-resp

    P^-β : P a ≡ P a'
    P^-β = trans (sym (lift-β _))
      (trans (cong P^ p) (lift-β _))
```

Setoid Integer

Base set

```
infix 4 _,_

data ℤ₀ : Set where
  _,_ : ℕ → ℕ → ℤ₀
```

Equivalence relation

```
infixl 2 _∼_

_∼_ : ℤ₀ → ℤ₀ → Set
(x+ , x-) ∼ (y+ , y-) = (x+ + y-) ≡ (y+ + x-)
```

Equivalence properties

```
∼refl : ∀ {a} → a ∼ a
∼refl {x+ , x-} = refl

∼sym : ∀ {a b} → a ∼ b → b ∼ a
∼sym {x+ , x-} {y+ , y-} = sym

∼trans :   ∀ {a b c} → a ∼ b → b ∼ c → a ∼ c
∼trans {x+ , x-} {y+ , y-} {z+ , z-} x=y y=z =
    cancel-+-left (y+ + y-)
    (swap24 y+ y- x+ z-
    >≡< ((y=z += x=y) >≡< swap13 z+ y- y+ x-))

_∼_isEquivalence : IsEquivalence _∼_
_∼_isEquivalence = record
```

```
  { refl   = ∼refl
  ; sym    = ∼sym
  ; trans = ∼trans
  }
```

$(\mathbb{Z}_0, \sim)$ is a setoid

```
ℤ-Setoid : Setoid
ℤ-Setoid = record
  { Carrier   = ℤ₀
  ; _~_       = _∼_
  ; isEquivalence = _∼_isEquivalence
  }
```

Definition of $\mathbb{Z}$

```
data ℤ : Set where
  +_   : (n : ℕ) → ℤ
  -suc_ : (n : ℕ) → ℤ
```

Normalisation function

```
[_]          : ℤ₀ → ℤ
[ m , 0 ]    = + m
[ 0 , suc n ]   = -suc n
[ suc m , suc n ] = [ m , n ]
```

Embedding function

```
⌜_⌝        : ℤ → ℤ₀
⌜ + n ⌝    = n , 0
⌜ -suc n ⌝ = 0 , ℕ.suc n
```

Stability

```
stable            : ∀ {n} → [ ⌜ n ⌝ ] ≡ n
stable {+ n}      = refl
stable { -suc n } = refl
```

Completeness

```
compl : ∀ n → ⌜ [ n ] ⌝ ∼ n
compl (x , 0)         = refl
compl (0 , suc y)     = refl
compl (suc x , suc y) = ∼trans (compl (x , y))
  (sym (sm+n≡m+sn x))
```

```
sound' : ∀ {i j} → ⌜ i ⌝ ∼ ⌜ j ⌝   → i ≡ j
sound'   {+ i} {+ j} eqt   = +_ ⋆ (+r-cancel 0 eqt)
sound'   {+ i} { -suc j } eqt with i +suc j ≢0 eqt
... | ()
sound'   { -suc i } { + j } eqt with j +suc i ≢0 ⟨ eqt ⟩
... | ()
```

```
sound'  { -suc i } { -suc j } eqt = -suc_ ⋆ pred ⋆ ⟨ eqt ⟩
```

Soundness

```
sound : ∀ {x y} → x ∼ y → [ x ] ≡ [ y ]
sound { x } { y } x∼y = sound' (∼trans (compl _)
   (∼trans (x∼y) (∼sym (compl _))))
```

The quotient definitions for ℤ

```
ℤ-PreQu : pre-Quotient ℤ-Setoid
ℤ-PreQu = record
   { Q    = ℤ
   ; [_]  =   [_]
   ; [_]=   = sound
   }

ℤ-QuD : def-Quotient ℤ-PreQu
ℤ-QuD = record
   { emb        = ⌜_⌝
   ; complete   = λ z → compl _
   ; stable     = λ z → stable
   }

ℤ-Qu = def-Quotient→Quotient ℤ-QuD
```

A.1 Rational numbers

```
data ℚ₀ : Set where
  _/suc_ : (n : ℤ) → (d : ℕ) → ℚ₀
```

Extractions

```
num : ℚ₀ → ℤ
num (n /suc _) = n

den : ℚ₀ → ℕ
den (_ /suc d) = suc d
```

Equivalence relation

```
infixl 2 _∼_

_∼_  : ℚ₀ → ℚ₀ → Set
n1 /suc d1 ∼ n2 /suc d2 =  n1 ℤ* (+ suc d2) ≡ n2 ℤ* (+ suc d1)
```

Property: a fraction is reduced

i.e. the absolute value of the numerator is comprime to the denominator

```
IsReduced : ℚ₀ → Set
IsReduced (n /suc d) = True (coprime? ∣ n ∣ (suc d))
```

The Definition of ℚ which is equivalent to the one in standard library

```
ℚ : Set
ℚ = Σ[ q : ℚ₀ ] IsReduced q
```

Normalisation function:

1. Calculate a reduced fraction for $\frac{x}{y}$ with a condition that y is not zero.

```
calℚ : ∀(x y : ℕ) → y ≢ 0 → ℚ
calℚ x y neo with gcd′ x y
calℚ .(q₁ ℕ* di) .(q₂ ℕ* di) neo
  | di , gcd-* q₁ q₂ c = (numr /suc pred q₂) , iscoprime
    where
      numr = + q₁
      deno = suc (pred q₂)

      lzero : ∀ x y → x ≡ 0 → x ℕ* y ≡ 0
      lzero .0 y refl = refl

      q2≢0 : q₂ ≢ 0
      q2≢0 qe = neo (lzero q₂ di qe)

      invsuc : ∀ n → n ≢ 0 → n ≡ suc (pred n)
      invsuc zero nz with nz refl
      ... | ()
      invsuc (suc n) nz = refl

      deno≡q2 : q₂ ≡ deno
      deno≡q2 = invsuc q₂ q2≢0

      copnd : Coprime q₁ deno
      copnd = subst (λ x → Coprime q₁ x) deno≡q2 c
```

```
witProp : ∀ a b → GCD a b 1
   → True (coprime? a b)
witProp a b gcd1 with gcd a b
witProp a b gcd1 | zero , y with GCD.unique gcd1 y
witProp a b gcd1 | zero , y | ()
witProp a b gcd1 | suc zero , y = tt
witProp a b gcd1 | suc (suc n) , y
                                   with GCD.unique gcd1 y
witProp a b gcd1 | suc (suc n) , y | ()

iscoprime : True (coprime? | numr | deno)
iscoprime = witProp _ _ (coprime-gcd copnd)
```

2.Negation

```
-_ : ℚ → ℚ
-_ ((n /suc d) , isC) = ((ℤ- n) /suc d) ,
   subst (λ x → True (coprime? x (suc d)))
   (forgetSign n) isC
   where
   forgetSign : ∀ x → | x | ≡ |  ℤ- x |
   forgetSign (-suc n) = refl
   forgetSign (+ zero) = refl
   forgetSign (+ (suc n)) = refl
```

3.Normalisation function

```
[_] : ℚ₀ → ℚ
[ (+ n) /suc d ] = calℚ n (suc d) (λ ())
[ (-suc n) /suc d ] = - calℚ (suc n) (suc d) (λ ())
```

Embedding function

$$\ulcorner\_\urcorner : \mathbb{Q} \to \mathbb{Q}_0$$
$$\ulcorner\_\urcorner = \mathsf{proj}_1$$

Appendix B

Category with families of setoids

B.1 Metatheory

Subset defined by a predicate B

```
record Subset {a b} (A : Set a)
    (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    prj₁ : A
    .prj₂ : B prj₁
open Subset public
```

Setoids

```
record Setoid : Set₁ where
  infix 4 _≈_
  field
    Carrier : Set
    _≈_  : Carrier → Carrier → Set
    .refl   : ∀{x} → x ≈ x
    .sym    : ∀{x y} → x ≈ y → y ≈ x
    .trans  : ∀{x y z} → x ≈ y → y ≈ z → x ≈ z
open Setoid public renaming
      (Carrier to |_| ; _≈_ to [_]_≈_ ; refl to [_]refl;
      trans to [_]trans; sym to [_]sym)
```

Morphisms between Setoids (Functors)

```
infix 5 _⇉_

record _⇉_ (A B : Setoid) : Set where
  constructor fn:_resp:_
  field
    fn    : | A | → | B |
    .resp : {x y : | A |} →
      ([ A ] x ≈ y) →
      [ B ] fn x ≈ fn y
open _⇉_ public renaming (fn to [_]fn ; resp to [_]resp)
```

Terminal object

```
● : Setoid
●    = record {
  Carrier = ⊤;
  _≈_    = λ _ _ → ⊤;
  refl     = tt;
  sym     = λ _ → tt;
  trans    = λ _ _ → tt }

⋆ : {Δ : Setoid} → Δ ⇉ ●
⋆ = record
  { fn = λ _ → tt
  ; resp = λ _ → tt }

uniqueHom : ∀ (Δ : Setoid)
    → (f : Δ ⇉ ●) → f ≡ ⋆
uniqueHom Δ f = PE.refl
```

B.2 Categories with families

Context are interpreted as setoids

```
Con = Setoid
```

Semantic Types

```
record Ty (Γ : Setoid) : Set₁ where
  field
    fm     : | Γ | → Setoid
    substT : {x y : | Γ |} →
      .([ Γ ] x ≈ y) →
      | fm x | →
      | fm y |
    .subst* : ∀{x y : | Γ |}
      (p : ([ Γ ] x ≈ y))
      {a b : | fm x |} →
      .([ fm x ] a ≈ b) →
      ([ fm y ] substT p a ≈ substT p b)

    .refl*   : ∀{x : | Γ |}{a : | fm x |} →
```

```
          [ fm x ] substT ([ Γ ]refl) a ≈ a
        .trans* : ∀{x y z : | Γ |}
          {p : [ Γ ] x ≈ y}
          {q : [ Γ ] y ≈ z}
          (a : | fm x |) →
          [ fm z ] substT q (substT p a)
                ≈ substT ([ Γ ]trans p q) a

    .tr* : ∀{x y : | Γ |}
       {p : [ Γ ] y ≈ x}
       {q : [ Γ ] x ≈ y}
       {a : | fm x |} →
       [ fm x ] substT p (substT q a) ≈ a
    tr* = [ fm _ ]trans (trans* _) refl*

    substT-inv : {x y : | Γ |} →
          .([ Γ ] x ≈ y) →
          | fm y | →
          | fm x |
    substT-inv p y = substT ([ Γ ]sym p) y
```

Type substitution

```
  _[_]T : ∀ {Γ Δ : Setoid} → Ty Δ → Γ ⇉ Δ → Ty Γ
  _[_]T {Γ} {Δ} A f
    = record
    { fm     = λ x → fm (fn x)
    ; substT = λ p → substT _
    ; subst* = λ p → subst* (resp p)
    ; refl*    = refl*
    ; trans* = trans*
    }
  where
    open Ty A
    open _⇉_ f
```

Semantic Terms

```
record Tm {Γ : Con}(A : Ty Γ) : Set where
  constructor tm:_resp:_
  field
    tm   : (x : | Γ |) → | [ A ]fm x |
    .respt : ∀ {x y : | Γ |} →
          (p : [ Γ ] x ≈ y) →
          [ [ A ]fm y ] [ A ]subst p (tm x) ≈ tm y

open Tm public renaming (tm to [_]tm ; respt to [_]respt)
```

Term substitution

```
_[_]m : ∀ {Γ Δ : Con}{A : Ty Δ} → Tm A
  → (f : Γ ⇉ Δ) → Tm (A [ f ]T)
_[_]m t f = record
  { tm = [ t ]tm ∘ [ f ]fn
  ; respt = [ t ]respt ∘ [ f ]resp
  }
```

Context comprehension

```
_&_ : (Γ : Setoid) → Ty Γ → Setoid
Γ & A =
  record { Carrier = Σ[ x : | Γ | ] | fm x |
        ; _≈_ = λ{(x , a) (y , b) →
          Σ[ p : x ≈ y ] [ fm y ] (substT p a) ≈ b }
        ; refl = refl , refl*
        ; sym =   λ {(p , q) → (sym p) ,
          [ fm _ ]trans (subst* _ ([ fm _ ]sym q)) tr* }
        ; trans = λ {(p , q) (m , n) → trans p m ,
  [ fm _ ]trans ([ fm _ ]trans
  ([ fm _ ]sym (trans* _)) (subst* _ q)) n}
        }
  where
    open Setoid Γ
    open Ty A


infixl 5 _&_

fst& : {Γ : Con}{A : Ty Γ} → Γ & A ⇉ Γ
fst& = record
        { fn = proj₁
        ; resp = proj₁
        }
```

Pairing operation

```
_,,_ : {Γ Δ : Con}{A : Ty Δ}(f : Γ ⇉ Δ)
  → (Tm (A [ f ]T)) → Γ ⇉ (Δ & A)
f ,, t = record
  { fn = < [ f ]fn , [ t ]tm >
  ; resp = < [ f ]resp , [ t ]respt >
  }
```

Projections

```
fst : {Γ Δ : Con}{A : Ty Δ} → Γ ⇉ (Δ & A) → Γ ⇉ Δ
fst f = record
  { fn = proj₁ ∘ [ f ]fn
  ; resp = proj₁ ∘ [ f ]resp
  }

snd : {Γ Δ : Con}{A : Ty Δ} → (f : Γ ⇉ (Δ & A))
  → Tm (A [ fst {A = A} f ]T)
snd f = record
  { tm = proj₂ ∘ [ f ]fn
  ; respt = proj₂ ∘ [ f ]resp
  }

_^_ : {Γ Δ : Con}(f : Γ ⇉ Δ)(A : Ty Δ)
  → Γ & A [ f ]T ⇉ Δ & A
f ^ A = record
  { fn = < [ f ]fn ∘ proj₁ , proj₂ >
  ; resp = < [ f ]resp ∘ proj₁ , proj₂ >
  }
```

Π-types (object level)

```
Π : {Γ : Setoid}(A : Ty Γ)(B : Ty (Γ & A)) → Ty Γ
Π {Γ} A B = record
  { fm = λ x → let Ax = [ A ]fm x in
  let Bx = λ a → [ B ]fm (x , a) in
```

```
record
{ Carrier = Subset ((a : | Ax |) → | Bx a |) (λ fn →
  (a b : | Ax |)
  (p : [ Ax ] a ≈ b) →
  [ Bx b ] [ B ]subst ([ Γ ]refl ,
  [ Ax ]trans [ A ]refl* p) (fn a) ≈ fn b)


; _≈_    = λ{(f , _) (g , _) → ∀ a → [ Bx a ] f a ≈ g a }
; refl                = λ a → [ Bx _ ]refl
; sym                 = λ f a → [ Bx _ ]sym (f a)
; trans               = λ f g a → [ Bx _ ]trans (f a) (g a)
}

; substT = λ {x} {y} p → λ {(f , rsp) →
                      let y2x = λ a → [ A ]subst ([ Γ ]sym p) a in
                      let x2y = λ a → [ A ]subst p a in
(λ a → [ B ]subst (p , [ A ]tr*)
(f (y2x a))) ,
(λ a b q →
  let a' = y2x a in
  let b' = y2x b in
  let q' = [ A ]subst* ([ Γ ]sym p) q in
  let H = rsp a' b' ([ A ]subst* ([ Γ ]sym p) q) in
  let r : [ Γ & A ] (x , b') ≈ (y , b) r = (p , [ A ]tr*) in
  let pre = [ B ]subst* r
     (rsp a' b' ([ A ]subst* ([ Γ ]sym p) q)) in
  [ [ B ]fm (y , b) ]trans
  ([ B ]trans* _)
  ([ [ B ]fm (y , b) ]trans
  ([ [ B ]fm (y , b) ]sym ([ B ]trans* _))
  pre))}
  ; subst* = λ _ q _ → [ B ]subst* _ (q _)
  ; refl* = λ {x} {a} ax
    → let rsp = prj₂ a in (rsp _ _ [ A ]refl*)
  ; trans* =   λ {(f , rsp) a →
  [ [ B ]fm _ ]trans
  ([ [ B ]fm _ ]trans
  ([ B ]trans* _)
  ([ [ B ]fm _ ]sym ([ B ]trans* _)))
  ([ B ]subst* _ (rsp _ _ ([ A ]trans* _) )) }
  }


lam : {Γ : Con}{A : Ty Γ}{B : Ty (Γ & A)} → Tm B → Tm (Π A B)
lam {Γ} {A} (tm: tm resp: respt) =
  record { tm = λ x → (λ a → tm (x , a))
    , (λ a b p → respt ([ Γ ]refl ,
    [ [ A ]fm x ]trans [ A ]refl* p))
  ; respt = λ p _ → respt (p , [ A ]tr*)
  }


app : {Γ : Con}{A : Ty Γ}{B : Ty (Γ & A)} → Tm (Π A B) → Tm B
app {Γ} {A} {B} (tm: tm resp: respt) =
  record { tm = λ {(x , a) → prj₁ (tm x) a}
  ; respt = λ {x} {y} → λ {(p , tr) →
    let fresp = prj₂ (tm (proj₁ x)) in
    [ [ B ]fm _ ]trans
    ([ B ]subst* (p , tr)
    ([ [ B ]fm _ ]sym [ B ]refl*))
    ([ [ B ]fm _ ]trans
    ([ B ]trans* {p = ([ Γ ]refl , [ A ]refl*)} _)

    ([ [ B ]fm _ ]trans
    ([ [ B ]fm _ ]sym
        ([ B ]trans* {q = (p , [ A ]tr*)} _))
        ([ [ B ]fm _ ]trans
        ([ B ]subst* _ (fresp _ _
                              ([ [ A ]fm _ ]trans ([ [ A ]fm _ ]sym [ A ]tr*)
                              ([ A ]subst* ([ Γ ]sym p) tr))))
        (respt p _)))) }
      }

    _⇒_ : {Γ : Con}(A B : Ty Γ) → Ty Γ
    A ⇒ B = Π A (B [ fst& {A = A} ]T)

    infixr 6 _⇒_
```

Simpler definition for functions

```
    [_,_]_⇒fm_ : (Γ : Con)(x : | Γ |)
      → Setoid → Setoid → Setoid
    [ Γ , x ] Ax ⇒fm Bx
      = record
      { Carrier = Σ[ fn : (| Ax | → | Bx |) ] ((a b : | Ax |)
        (p : [ Ax ] a ≈ b) → [ Bx ] fn a ≈ fn b)
      ; _≈_   = λ{(f , _) (g , _) → ∀ a → [ Bx ] f a ≈ g a }
      ; refl   = λ _ → [ Bx ]refl
      ; sym    = λ f a → [ Bx ]sym (f a)
      ; trans  = λ f g a → [ Bx ]trans (f a) (g a)
      }
```

Σ-types (object level)

```
    Σ' : {Γ : Con}(A : Ty Γ)(B : Ty (Γ & A)) → Ty Γ
```

```
Σ' {Γ} A B = record
  { fm = λ x → let Ax = [ A ]fm x in
    let Bx = λ a → [ B ]fm (x , a) in
  record
  { Carrier = Σ[ a : | Ax | ] | Bx a |

  ; _≈_            = λ{(a₁ , b₁) (a₂ , b₂) →
    Subset ([ Ax ] a₁ ≈ a₂)
    (λ eq₁ → [ Bx _ ] [ B ]subst
      ([ Γ ]refl , [ [ A ]fm x ]trans
      [ A ]refl* eq₁) b₁ ≈ b₂)
  }

  ; refl    = λ {t} → [ Ax ]refl , [ B ]refl*

  ; sym     = λ {(p , q) → ([ Ax ]sym p) ,
      [ Bx _ ]trans ([ B ]subst* _
      ([ Bx _ ]sym q)) [ B ]tr*}

  ; trans   = λ {(p , q) (r , s) → ([ Ax ]trans p r) ,
      [ Bx _ ]trans ([ Bx _ ]trans
      ([ Bx _ ]sym ([ B ]trans* _))
      ([ B ]subst* _ q)) s}
  }

  ; substT = λ x≈y → λ {(p , q) →
    ([ A ]subst x≈y p) , [ B ]subst (x≈y ,
    [ [ A ]fm _ ]refl) q}

  ; subst* = λ x≈y → λ {(p , q) → [ A ]subst* x≈y p ,
    [ [ B ]fm _ ]trans ([ [ B ]fm _ ]trans
    ([ B ]trans* _)
    ([ [ B ]fm _ ]sym ([ B ]trans* _)))
    ([ B ]subst* (x≈y , [ [ A ]fm _ ]refl) q) }
    ; refl* = λ {x} {a} →
    let (p , q) = a in [ A ]refl* , [ B ]tr*
    ; trans* =            λ {(p , q)   → ([ A ]trans* _) ,
                          ([ [ B ]fm _ ]trans
                          ([ B ]trans* _) ([ B ]trans* _)) }
    }
```

Binary relation

```
  Rel : {Γ : Con} → Ty Γ → Set₁
  Rel {Γ} A = Ty (Γ & A & A [ fst& {A = A} ]T)
```

Natural numbers

Axiom: irrelevant:

```
  postulate
    .irrelevant : {A : Set} → .A → A
```

```
  module Natural (Γ : Con) where

    _≈nat_ : ℕ → ℕ → Set
    zero ≈nat zero = ⊤
    zero ≈nat suc n = ⊥
    suc m ≈nat zero = ⊥
    suc m ≈nat suc n = m ≈nat n

    reflNat : {x : ℕ} → x ≈nat x
```

```
reflNat {zero} = tt
reflNat {suc n} = reflNat {n}

symNat : {x y : ℕ} → x ≈nat y → y ≈nat x
symNat {zero} {zero} eq = tt
symNat {zero} {suc _} eq = eq
symNat {suc _} {zero} eq = eq
symNat {suc x} {suc y} eq = symNat {x} {y} eq

transNat : {x y z : ℕ}
  → x ≈nat y → y ≈nat z → x ≈nat z
transNat {zero} {zero} xy yz = yz
transNat {zero} {suc _} () yz
transNat {suc _} {zero} () yz
transNat {suc _} {suc _} {zero} xy yz = yz
transNat {suc x} {suc y} {suc z} xy yz =
  transNat {x} {y} {z} xy yz


⟦Nat⟧ : Ty Γ
⟦Nat⟧ = record
  { fm = λ γ → record
    { Carrier = ℕ
    ; _≈_ = _≈nat_
    ; refl = λ {n} → reflNat {n}
    ; sym = λ {x} {y} → symNat {x} {y}
    ; trans = λ {x} {y} {z} → transNat {x} {y} {z}
    }
  ; substT = λ _ n → n
  ; subst* = λ _ x → irrelevant x
  ; refl* = λ {x} {a} → reflNat {a}
  ; trans* = λ a → reflNat {a}
  }
```

```
⟦0⟧ : Tm ⟦Nat⟧
⟦0⟧ = record
  { tm = λ _ → 0
  ; respt = λ p → tt
  }

⟦s⟧ : Tm ⟦Nat⟧ → Tm ⟦Nat⟧
⟦s⟧ (tm: t resp: respt)
  = record
  { tm = suc ∘ t
  ; respt = respt
  }
```

Simply typed universe

Quotient types

```
module Q (Γ : Con)(A : Ty Γ)

    (R : (γ : | Γ |) → | [ A ]fm γ | → | [ A ]fm γ | → Set)

    .(Rrespt : ∀{γ γ' : | Γ |}
              (p : [ Γ ] γ ≈ γ')
              (a b : | [ A ]fm γ |) →
              .(R γ a b) →
              R γ' ([ A ]subst p a) ([ A ]subst p b))

    .(Rrsp : ∀ {γ a b} → .([ [ A ]fm γ ] a ≈ b) → R γ a b)

    .(Rref : ∀ {γ a} → R γ a a)
    .(Rsym : (∀ {γ a b} → .(R γ a b) → R γ b a))
    .(Rtrn :   (∀ {γ a b c} → .(R γ a b)
      →   .(R γ b c) → R γ a c))
    where

  ⟦Q⟧₀ : | Γ | → Setoid
  ⟦Q⟧₀ γ = record
    { Carrier = | [ A ]fm γ |
    ; _≈_ = R γ
    ; refl = Rref
    ; sym = Rsym
    ; trans = Rtrn
    }


  ⟦Q⟧ : Ty Γ
  ⟦Q⟧ = record
    { fm = ⟦Q⟧₀
    ; substT = [ A ]subst
    ; subst* = λ p q → Rrespt p _ _ q
    ; refl* = Rrsp [ A ]refl*
    ; trans* = λ a → Rrsp ([ A ]trans* _)
    }

  ⟦[_]⟧ : Tm A → Tm ⟦Q⟧
  ⟦[ x ]⟧ = record
    { tm = [ x ]tm
    ; respt = λ p → Rrsp ([ x ]respt p)
    }

  ⟦[_]⟧' : Tm (A ⇒ ⟦Q⟧)
  ⟦[_]⟧' = record
    { tm = λ x → (λ a → a) ,
        (λ a b p →
        Rrsp ([ [ A ]fm _ ]trans [ A ]refl* p))
    ; respt = λ p a → Rrsp [ A ]tr*
    }
```

```
.Q-Ax : ∀ γ a b → [ [ A ]fm γ ] a ≈ b → [ [ ⟦Q⟧ ]fm _ ] a ≈ b
Q-Ax γ a b = Rrsp

Q-elim : (B : Ty Γ)(f : Tm (A ⇒ B))
  (frespR : ∀ γ a b → (R γ a b)
    → [ [ B ]fm γ ] prj₁ ([ f ]tm γ) a
      ≈   prj₁ ([ f ]tm γ) b)
→ Tm (⟦Q⟧ ⇒ B)
Q-elim B f frespR = record
  { tm = λ γ → prj₁ ([ f ]tm γ) , (λ a b p →
    [ [ B ]fm _ ]trans [ B ]refl* (frespR _ _ _ p))
  ; respt = λ {γ} {γ'} p a → [ f ]respt p a
  }


substQ : (Γ & A) ⇉ (Γ & ⟦Q⟧)
substQ = record
  { fn = λ {(x , a) → x , a}
  ; resp = λ{ (p , q) → p , (Rrsp q)}
  }

Q-ind : (P : Ty (Γ & ⟦Q⟧))
→ (isProp : ∀ {x a} (r s : | [ P ]fm (x , a) |) →
  [ [ P ]fm (x , a) ] r ≈ s )
→ (h : Tm (Π A (P [ substQ ]T)))
→ Tm (Π ⟦Q⟧ P)
Q-ind P isProp h = record
  { tm = λ x → (prj₁ ([ h ]tm x)) ,
    (λ a b p → isProp {x} {b} _ _)
  ; respt = [ h ]respt
```

```
            }
```

Appendix C

syntactic weak ω-groupoids

C.1 Syntax of $\mathcal{T}_{\infty-groupoid}$

```
data Con           : Set
data Ty (Γ : Con)  : Set
data Tm            : {Γ : Con}(A : Ty Γ) → Set
data Var           : {Γ : Con}(A : Ty Γ) → Set
data _⇒_           : Con → Con → Set
data isContr       : Con → Set
```

Contexts

```
data Con where
  ε     : Con
  _,_   : (Γ : Con)(A : Ty Γ) → Con
```

Types

```
data Ty Γ where
  *        : Ty Γ
  _=h_     : {A : Ty Γ}(a b : Tm A) → Ty Γ
```

Heterogeneous Equality for Terms

```
data _≅_ {Γ : Con}{A : Ty Γ} :
    {B : Ty Γ} → Tm A → Tm B → Set where
  refl : (b : Tm A) → b ≅ b

_-¹          : ∀{Γ : Con}{A B : Ty Γ}
  {a : Tm A}{b : Tm B} → a ≅ b → b ≅ a
(refl _) -¹   = refl _

infixr 4 _∼_

_∼_ : {Γ : Con}
  {A B C : Ty Γ}
  {a : Tm A}{b : Tm B}{c : Tm C} →
  a ≅ b →
  b ≅ c
  → a ≅ c
_∼_ {c = c} (refl .c) (refl .c) = refl c

_⟦_⟫         : {Γ : Con}{A B : Ty Γ}(a : Tm B)
             → A ≡ B → Tm A
a ⟦ refl ⟫   = a

cohOp        : {Γ : Con}{A B : Ty Γ}{a : Tm B}(p : A ≡ B)
             → a ⟦ p ⟫ ≅ a
```

```
cohOp refl   = refl _
```

```
cohOp-eq : {Γ : Con}{A B : Ty Γ}{a b : Tm B}
  {p : A ≡ B} → (a ≅ b)
  → (a ⟦ p ⟫ ≅ b ⟦ p ⟫)
cohOp-eq {Γ} {.B} {B} {a} {b} {refl} r = r

cohOp-hom : {Γ : Con}{A B : Ty Γ}{a b : Tm B}(p : A ≡ B) →
    (a ⟦ p ⟫ =h b ⟦ p ⟫) ≡ (a =h b)
cohOp-hom refl = refl

cong≅ : {Γ Δ : Con}{A B : Ty Γ}{a : Tm A}{b : Tm B}
  {D : Ty Γ → Ty Δ} → (f : {C : Ty Γ} → Tm C → Tm (D C))→
  a ≅ b   → f a ≅ f b
cong≅ f (refl _) = refl _
```

Substitutions

```
_[_]T    : ∀{Γ Δ} → Ty Δ → Γ ⇒ Δ → Ty Γ
_[_]V    : ∀{Γ Δ A} → Var A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_[_]tm   : ∀{Γ Δ A} → Tm A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_⊚_ : ∀{Γ Δ Θ} → Δ ⇒ Θ → (δ : Γ ⇒ Δ) → Γ ⇒ Θ
```

Contexts morphisms

```
data _⇒_ where
  •      : ∀{Γ} → Γ ⇒ ε
  _,_    : ∀{Γ Δ}(δ : Γ ⇒ Δ){A : Ty Δ}(a : Tm (A [ δ ]T))
```

```
          → Γ ⇒ (Δ , A)
```

Weakening

```
_+T_    : ∀{Γ}(A : Ty Γ)(B : Ty Γ) → Ty (Γ , B)
_+tm_   : ∀{Γ A}(a : Tm A)(B : Ty Γ) → Tm (A +T B)
_+S_    : ∀{Γ Δ}(δ : Γ ⇒ Δ)(B : Ty Γ) → (Γ , B) ⇒ Δ
```

```
*   +T B = *
(a =h b) +T B = a +tm B =h b +tm B
```

```
*   [ δ ]T = *
(a =h b) [ δ ]T = a [ δ ]tm =h b [ δ ]tm
```

Variables and terms

```
data Var where
  v0 : ∀{Γ}{A : Ty Γ}    → Var (A +T A)
  vS : ∀{Γ}{A B : Ty Γ}(x : Var A) → Var (A +T B)

data Tm where
  var   : ∀{Γ}{A : Ty Γ} → Var A → Tm A
  coh   : ∀{Γ Δ} → isContr Δ → (δ : Γ ⇒ Δ)
        → (A : Ty Δ) → Tm (A [ δ ]T)

cohOpV : {Γ : Con}{A B : Ty Γ}{x : Var A}(p : A ≡ B) →
```

```
               var (subst Var p x) ≅ var x
cohOpV {x = x} refl = refl (var x)

cohOpVs : {Γ : Con}{A B C : Ty Γ}{x : Var A}(p : A ≡ B) →
                var (vS {B = C} (subst Var p x)) ≅ var (vS x)
cohOpVs {x = x} refl = refl (var (vS x))

coh-eq : {Γ Δ : Con}{isc : isContr Δ}{γ δ : Γ ⇒ Δ}
               {A : Ty Δ} → γ ≡ δ → coh isc γ A ≅ coh isc δ A
coh-eq refl = refl _
```

Contractible contexts

```
data isContr where
  c*    : isContr (ε , *)
  ext   : ∀{Γ} → isContr Γ → {A : Ty Γ}(x : Var A)
        → isContr (Γ , A , (var (vS x) =h var v0))
```

```
hom≡ : {Γ : Con}{A A' : Ty Γ}
  {a : Tm A}{a' : Tm A'}(q : a ≅ a')
  {b : Tm A}{b' : Tm A'}(r : b ≅ b')
  → (a =h b) ≡ (a' =h b')
hom≡ {Γ} {.A'} {A'} {.a'} {a'} (refl .a') {.b'} {b'} (refl .b') = refl
```

```
S-eq : {Γ Δ : Con}{γ δ : Γ ⇒ Δ}{A : Ty Δ}
  {a : Tm (A [ γ ]T)}{a' : Tm (A [ δ ]T)}
  → γ ≡ δ → a ≅ a'
  → _≡_ {_} {Γ ⇒ (Δ , A)} (γ , a) (δ , a')
```

```
S-eq refl (refl _) = refl
```

Some lemmas

```
[⊚]T      : ∀{Γ Δ Θ A}{θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → A [ θ ⊚ δ ]T ≡ (A [ θ ]T)[ δ ]T

[⊚]v      : ∀{Γ Δ Θ A}(x : Var A){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → x [ θ ⊚ δ ]V ≅ (x [ θ ]V) [ δ ]tm

[⊚]tm     : ∀{Γ Δ Θ A}(a : Tm A){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → a [ θ ⊚ δ ]tm ≅ (a [ θ ]tm) [ δ ]tm

⊚assoc    : ∀{Γ Δ Θ Ω}(γ : Θ ⇒ Ω){θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → (γ ⊚ θ) ⊚ δ ≡ γ ⊚ (θ ⊚ δ)



•         ⊚ δ = •
(δ , a) ⊚ δ' = (δ ⊚ δ') , a [ δ' ]tm ⟦ [⊚]T ⟫

[+S]T     : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}
          → A [ δ +S B ]T ≡ (A [ δ ]T) +T B

[+S]tm    : ∀{Γ Δ A B}(a : Tm A){δ : Γ ⇒ Δ}
          → a [ δ +S B ]tm ≅ (a [ δ ]tm) +tm B

[+S]S     : ∀{Γ Δ Θ B}{δ : Δ ⇒ Θ}{γ : Γ ⇒ Δ}
          → δ ⊚ (γ +S B) ≡ (δ ⊚ γ) +S B

wk-tm+       : {Γ Δ : Con}{A : Ty Δ}{δ : Γ ⇒ Δ}(B : Ty Γ)
             → Tm (A [ δ ]T +T B) → Tm (A [ δ +S B ]T)
```

```
wk-tm+ B t   = t ⟦ [+S]T ⟫

•          +S B = •
(δ , a) +S B = (δ +S B) , wk-tm+ B (a +tm B)

[+S]T {A = *}      = refl
[+S]T {A = a =h b} = hom≡ ([+S]tm a) ([+S]tm b)

+T[,]T     : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{b : Tm (B [ δ ]T)}
           → (A +T B) [ δ , b ]T ≡ A [ δ ]T

+tm[,]tm   : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{c : Tm (B [ δ ]T)}
           → (a : Tm A)
           → (a +tm B) [ δ , c ]tm ≅ a [ δ ]tm

(var x)         +tm B = var (vS x)
(coh cΔ δ A) +tm B = coh cΔ (δ +S B) A ⟦ sym [+S]T ⟫

cong+tm : {Γ : Con}{A B C : Ty Γ}{a : Tm A}{b : Tm B} →
          a ≅ b
        → a +tm C ≅ b +tm C
cong+tm (refl _) = refl _

cong+tm2 : {Γ : Con}{A B C : Ty Γ}
  {a : Tm B}(p : A ≡ B)
    → a +tm C ≅ a ⟦ p ⟫ +tm C
cong+tm2 refl = refl _

wk-T : {Δ : Con}
  {A B C : Ty Δ}
  → A ≡ B → A +T C ≡ B +T C
wk-T refl = refl

wk-tm : {Γ Δ : Con}
```

```
    {A : Ty Δ}{δ : Γ ⇒ Δ}
    {B : Ty Δ}{b : Tm (B [ δ ]T)}
    → Tm (A [ δ ]T) → Tm ((A +T B) [ δ , b ]T)
wk-tm t = t ⟦ +T[,]T ⟫

v0   [ δ , a ]V = wk-tm a
vS x [ δ , a ]V = wk-tm (x [ δ ]V)

wk-coh : {Γ Δ : Con}
    {A : Ty Δ}{δ : Γ ⇒ Δ}
    {B : Ty Δ}{b : Tm (B [ δ ]T)}
    {t : Tm (A [ δ ]T)}
    → wk-tm {B = B} {b = b} t ≅ t
wk-coh = cohOp +T[,]T

wk-coh+ : {Γ Δ : Con}
    {A : Ty Δ}{δ : Γ ⇒ Δ}
    {B : Ty Γ}
    {x : Tm (A [ δ ]T +T B)}
          → wk-tm+ B x ≅ x
wk-coh+ = cohOp [+S]T

wk-hom : {Γ Δ : Con}
    {A : Ty Δ}{δ : Γ ⇒ Δ}
    {B : Ty Δ}{b : Tm (B [ δ ]T)}
    {x y : Tm (A [ δ ]T)}
    → (wk-tm {B = B} {b = b} x =h wk-tm
    {B = B} {b = b} y) ≡ (x =h y)
wk-hom = hom≡ wk-coh wk-coh


wk-hom+ : {Γ Δ : Con}
    {A : Ty Δ}{δ : Γ ⇒ Δ}
    {B : Ty Γ}
    {x y : Tm (A [ δ ]T +T B)}
    → (wk-tm+ B x =h wk-tm+ B y) ≡ (x =h y)
wk-hom+ = hom≡ wk-coh+ wk-coh+



wk-⊚ : {Γ Δ Θ : Con}
  {θ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}{A : Ty Θ}
  → Tm ((A [ θ ]T)[ δ ]T) → Tm (A [ θ ⊚ δ ]T)
wk-⊚ t = t ⟦ [⊚]T ⟫

[+S]S {δ = •} = refl
[+S]S {δ = δ , a} = S-eq [+S]S (cohOp [⊚]T ∼
  ([+S]tm a ∼ cong+tm2 [⊚]T) ∼ wk-coh+ -¹)



wk+S+T : ∀{Γ Δ : Con}{A : Ty Γ}{B : Ty Δ}
         {γ}{C} →
         A [ γ ]T ≡ C
  → A [ γ +S B ]T ≡ C +T B
wk+S+T eq = trans [+S]T (wk-T eq)

wk+S+tm : {Γ Δ : Con}{A : Ty Γ}{B : Ty Δ}
          (a : Tm A){C : Ty Δ}{γ : Δ ⇒ Γ}{c : Tm C} →
          a [ γ ]tm ≅ c
        → a [ γ +S B ]tm ≅ c +tm B
wk+S+tm _ eq = [+S]tm _ ∼ cong+tm eq



wk+S+S : ∀{Γ Δ Δ₁ : Con}{δ : Δ ⇒ Δ₁}{γ : Γ ⇒ Δ}
    {ω : Γ ⇒ Δ₁}{B : Ty Γ}
  → δ ⊚ γ ≡ ω
  → δ ⊚ (γ +S B) ≡ ω +S B
wk+S+S eq = trans [+S]S (cong (λ x → x +S _) eq)
```

```
[⊚]T {A = *} = refl
[⊚]T {A = _=h_ {A} a b} = hom≡ ([⊚]tm _) ([⊚]tm _)

+T[,]T {A = *} = refl
+T[,]T {A = _=h_ {A} a b} = hom≡  (+tm[,]tm _) (+tm[,]tm _)

var x         [ δ ]tm = x [ δ ]V
coh cΔ γ A    [ δ ]tm = coh cΔ (γ ⊚ δ) A ⟦ sym [⊚]T ⟫




congT : ∀ {Γ Δ : Con}{A B : Ty Δ}{γ : Γ ⇒ Δ} → A ≡ B → A [ γ ]T ≡ B [ γ ]T
congT refl = refl

congT2 : ∀ {Γ Δ} → {δ γ : Δ ⇒ Γ}{A : Ty Γ} → δ ≡ γ → A [ δ ]T ≡ A [ γ ]T
congT2 refl = refl

congV : {Γ Δ : Con}{A B : Ty Δ}{a : Var A}{b : Var B} →
  var a ≅ var b →
  {δ : Γ ⇒ Δ}
  → a [ δ ]V ≅ b [ δ ]V
congV {Γ} {Δ} {.B} {B} {.b} {b} (refl .(var b)) = refl _

congtm : {Γ Δ : Con}{A B : Ty Γ}{a : Tm A}{b : Tm B}
     (p : a ≅ b) →
     {δ : Δ ⇒ Γ}
     → a [ δ ]tm ≅ b [ δ ]tm
congtm (refl _) = refl _

congtm2 : {Γ Δ : Con}{A : Ty Γ}{a : Tm A}
        {δ γ : Δ ⇒ Γ} →
        (p : δ ≡ γ)
  → a [ δ ]tm ≅ a [ γ ]tm
congtm2 refl = refl _

⊚assoc • = refl
⊚assoc (_,_ γ {A} a) = S-eq (⊚assoc γ)
  (cohOp [⊚]T
  ∼ (congtm (cohOp [⊚]T)
  ∼ ((cohOp [⊚]T
  ∼ [⊚]tm a) -¹)))


[⊚]v (v0 {Γ₁} {A}) {θ , a}   = wk-coh ∼ cohOp
  [⊚]T ∼ congtm (cohOp +T[,]T -¹)
[⊚]v (vS {Γ₁} {A} {B} x) {θ , a} =
  wk-coh ∼ ([⊚]v x ∼ (congtm (cohOp +T[,]T) -¹))



[⊚]tm (var x) = [⊚]v x
[⊚]tm (coh c γ A) = cohOp (sym [⊚]T) ∼ (coh-eq (sym (⊚assoc γ))
    ∼ cohOp (sym [⊚]T) -¹) ∼ congtm (cohOp (sym [⊚]T) -¹)


⊚wk : ∀{Γ Δ Δ₁}{B : Ty Δ}(γ : Δ ⇒ Δ₁){δ : Γ ⇒ Δ}
    {c : Tm (B [ δ ]T)} → (γ +S B) ⊚ (δ , c) ≡ γ ⊚ δ
⊚wk • = refl
⊚wk (_,_ γ {A} a) = S-eq (⊚wk γ) (cohOp [⊚]T ∼
  (congtm (cohOp [+S]T) ∼ +tm[,]tm a) ∼ cohOp [⊚]T -¹)

+tm[,]tm (var x) = cohOp +T[,]T
+tm[,]tm (coh x γ A) = congtm (cohOp (sym [+S]T)) ∼
  cohOp (sym [⊚]T) ∼ coh-eq (⊚wk γ) ∼ cohOp (sym [⊚]T) -¹
```

```
[+S]V : {Γ Δ : Con}{A : Ty Δ}
  (x : Var A){δ : Γ ⇒ Δ}
  {B : Ty Γ}
  → x [ δ +S B ]V ≅ (x [ δ ]V) +tm B
[+S]V v0 {_,_ δ {A} a} = wk-coh ∼ wk-coh+ ∼ cong+tm2 +T[,]T
[+S]V (vS x) {δ , a} = wk-coh ∼ [+S]V x ∼ cong+tm2 +T[,]T
```

```
[+S]tm (var x) = [+S]V x
[+S]tm (coh x δ A) = cohOp (sym [⊚]T) ∼ coh-eq [+S]S ∼
  cohOp (sym [+S]T) -¹ ∼ cong+tm2 (sym [⊚]T)
```

Some simple contexts

```
x:* : Con
x:* = ε , *

x:*,y:*,α:x=y : Con
x:*,y:*,α:x=y = x:* , * , (var (vS v0) =h var v0)

vX : Tm {x:*,y:*,α:x=y} *
vX = var (vS (vS v0))

vY : Tm {x:*,y:*,α:x=y} *
vY = var (vS v0)

vα : Tm {x:*,y:*,α:x=y} (vX =h vY)
vα = var v0

x:*,y:*,α:x=y,z:*,β:y=z : Con
```

```
x:*,y:*,α:x=y,z:*,β:y=z = x:*,y:*,α:x=y , * ,
  (var (vS (vS v0)) =h var v0)

vZ : Tm {x:*,y:*,α:x=y,z:*,β:y=z} *
vZ = var (vS v0)

vβ : Tm {x:*,y:*,α:x=y,z:*,β:y=z} (vY +tm _ +tm _ =h vZ)
vβ = var v0
```

C.2 Some Important Derivable Constructions

Identity morphism

```
IdS : ∀{Γ} → Γ ⇒ Γ
IC-T : ∀{Γ}{A : Ty Γ} → A [ IdS ]T ≡ A
IC-v   : ∀{Γ : Con}{A : Ty Γ}(x : Var A) → x [ IdS ]V ≅ var x
IC-S   : ∀{Γ Δ : Con}(δ : Γ ⇒ Δ)   → δ ⊚ IdS ≡ δ
IC-tm : ∀{Γ : Con}{A : Ty Γ}(a : Tm A) → a [ IdS ]tm ≅ a

Coh-Contr        : ∀{Γ}{A : Ty Γ} → isContr Γ → Tm A
Coh-Contr isC    = coh isC IdS _ ⟦ sym IC-T ⟫

IdS {ε}    = •
IdS {Γ , A} = IdS +S _ , var v0 ⟦ wk+S+T IC-T ⟫

IC-T {Γ} {*} = refl
IC-T {Γ} {a =h b} = hom≡ (IC-tm a) (IC-tm b)

IC-v {.(Γ , A)} {.(A +T A)} (v0 {Γ} {A}) = wk-coh ∼ cohOp (wk+S+T IC-T)
IC-v {.(Γ , B)} {.(A +T B)} (vS {Γ} {A} {B} x) = wk-coh ∼
  wk+S+tm (var x) (IC-v _)
```

```
IC-S • = refl
IC-S (δ , a) = S-eq (IC-S δ) (cohOp [⊚]T ∼ IC-tm a)

IC-tm (var x) = IC-v x
IC-tm (coh x δ A) = cohOp (sym [⊚]T) ∼ coh-eq (IC-S δ)
```

Some auxiliary functions

```
1-1S-same : {Γ : Con}{A B : Ty Γ} →
  B ≡ A → (Γ , A) ⇒ (Γ , B)
1-1S-same eq = pr1 , pr2 ⟦ congT eq ⟫

1-1S-same-T : {Γ : Con}{A B : Ty Γ} →
    (eq : B ≡ A) → (A +T B) [ 1-1S-same eq ]T ≡ A +T A
1-1S-same-T eq = trans +T[,]T (trans [+S]T (wk-T IC-T))

1-1S-same-tm : ∀ {Γ : Con}{A : Ty Γ}{B : Ty Γ} →
    (eq : B ≡ A)(a : Tm A) →
    (a +tm B) [ 1-1S-same eq ]tm ≅ (a +tm A)
1-1S-same-tm eq a = +tm[,]tm a ∼ [+S]tm a ∼ cong+tm (IC-tm a)

1-1S-same-v0 : ∀ {Γ : Con}{A B : Ty Γ} →
    (eq : B ≡ A) → var v0 [ 1-1S-same eq ]tm ≅ var v0
1-1S-same-v0 eq = wk-coh ∼ cohOp (congT eq) ∼ pr2-v0

_++_ : Con → Con → Con

cor : {Γ : Con}(Δ : Con) → (Γ ++ Δ) ⇒ Δ

repeat-p1 : {Γ : Con}(Δ : Con) → (Γ ++ Δ) ⇒ Γ
```

```
Γ ++ ε = Γ
Γ ++ (Δ , A) = Γ ++ Δ , A [ cor Δ ]T

repeat-p1 ε = IdS
repeat-p1 (Δ , A) = repeat-p1 Δ ⊚ pr1

cor ε = •
cor (Δ , A) = (cor Δ +S _) , var v0 ⟦ [+S]T ⟫

_++S_ : ∀ {Γ Δ Θ} → Γ ⇒ Δ → Γ ⇒ Θ → Γ ⇒ (Δ ++ Θ)
cor-inv : ∀ {Γ Δ Θ} → {γ : Γ ⇒ Δ}(δ : Γ ⇒ Θ) → cor Θ ⊚ (γ ++S δ) ≡ δ

γ ++S • = γ
γ ++S (δ , a) = γ ++S δ , a ⟦ trans (sym [⊚]T) (congT2 (cor-inv _)) ⟫

cor-inv • = refl
cor-inv (δ , a) = S-eq (trans (⊚wk _) (cor-inv δ))
  (cohOp [⊚]T ∼ congtm (cohOp [+S]T)
  ∼ cohOp +T[,]T
  ∼ cohOp (trans (sym [⊚]T) (congT2 (cor-inv _))))

id-S++ : {Γ : Con}(Δ Θ : Con) → (Δ ⇒ Θ) → (Γ ++ Δ) ⇒ (Γ ++ Θ)
id-S++ Δ Θ γ = repeat-p1 Δ ++S (γ ⊚ cor _)
```

C.2.1 Suspension and Replacement

One-step suspension

```
ΣC : Con → Con
ΣT : ∀{Γ} → Ty Γ → Ty (ΣC Γ)

ΣC ε          = ε , * , *
```

```
ΣC (Γ , A)   = ΣC Γ , ΣT A

Σv    : ∀{Γ}{A : Ty Γ} → Var A → Var (ΣT A)
Σtm   : ∀{Γ}{A : Ty Γ} → Tm A → Tm (ΣT A)
Σs    : ∀{Γ Δ} → Γ ⇒ Δ → ΣC Γ ⇒ ΣC Δ

*' : {Γ : Con} → Ty (ΣC Γ)
*' {ε} = var (vS v0) =h var v0
*' {Γ , A} = *' {Γ} +T _

ΣT {Γ} * = *' {Γ}
ΣT (a =h b) = Σtm a =h Σtm b

Σs• : (Γ : Con) → ΣC Γ ⇒ ΣC ε
Σs• ε = IdS
Σs• (Γ , A) = Σs• Γ +S _

ΣC-Contr : ∀ Δ → isContr Δ → isContr (ΣC Δ)

ΣT[+T]    : ∀{Γ}(A B : Ty Γ)
            → ΣT (A +T B) ≡ ΣT A +T ΣT B

Σtm[+tm] : ∀{Γ A}(a : Tm A)(B : Ty Γ)
            → Σtm (a +tm B) ≅ Σtm a +tm ΣT B

ΣT[Σs]T   : ∀{Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)
            → (ΣT A) [ Σs δ ]T ≡ ΣT (A [ δ ]T)

ΣT[+T] * B = refl
ΣT[+T] (_=h_ {A} a b) B = hom≡ (Σtm[+tm] a B) (Σtm[+tm] b B)

Σv {.(Γ , A)} {.(A +T A)} (v0 {Γ} {A}) = subst Var (sym (ΣT[+T] A A)) v0
Σv {.(Γ , B)} {.(A +T B)} (vS {Γ} {A} {B} x) = subst Var (sym (ΣT[+T]
   {_} A B)) (vS (Σv x))
```

```
Σtm (var x) = var (Σv x)
Σtm (coh x δ A) = coh (ΣC-Contr _ x) (Σs δ) (ΣT A) ⟦ sym (ΣT[Σs]T A δ) ⟫

Σtm-p1 : {Γ : Con}(A : Ty Γ) → Σtm {Γ , A} (var v0) ≅ var v0
Σtm-p1 A = cohOpV (sym (ΣT[+T] A A))

Σtm-p2 : {Γ : Con}(A B : Ty Γ)(x : Var A) → var (Σv (vS {B = B} x)) ≅
  var (vS (Σv x))
Σtm-p2 {Γ} A B x = cohOpV (sym (ΣT[+T] A B))

Σtm-p2-sp : {Γ : Con}(A : Ty Γ)(B : Ty (Γ , A)) → Σtm {Γ , A , B}
            (var (vS v0)) ≅ (var v0) +tm _
Σtm-p2-sp A B = Σtm-p2 (A +T A) B v0 ∼  cong+tm (Σtm-p1 A)

Σs {Γ} {Δ , A} (γ , a) = (Σs γ) , Σtm a ⟦ ΣT[Σs]T A γ ⟫
Σs {Γ} • = Σs• Γ

congΣtm : {Γ : Con}{A B : Ty Γ}{a : Tm A}{b : Tm B} → a ≅ b →
  Σtm a ≅ Σtm b
congΣtm (refl _) = refl _

cohOpΣtm : ∀ {Δ : Con}{A B : Ty Δ}(t : Tm B)(p : A ≡ B) →
           Σtm (t ⟦ p ⟫) ≅ Σtm t
cohOpΣtm t p =  congΣtm (cohOp p)

Σs⊚ : ∀ {Δ Δ₁ Γ}(δ : Δ ⇒ Δ₁)(γ : Γ ⇒ Δ) → Σs (δ ⊚ γ) ≡ Σs δ ⊚ Σs γ

Σv[Σs]v : ∀ {Γ Δ : Con}{A : Ty Δ}(x : Var A)(δ : Γ ⇒ Δ) →
          Σv x [ Σs δ ]V ≅ Σtm (x [ δ ]V)
Σv[Σs]v (v0 {Γ} {A}) (δ , a) = congtm (Σtm-p1 A) ∼ wk-coh ∼
  cohOp (ΣT[Σs]T A δ) ∼ cohOpΣtm a +T[,]T -¹
Σv[Σs]v (vS {Γ} {A} {B} x) (δ , a) = congtm (Σtm-p2 A B x) ∼
  +tm[,]tm (Σtm (var x)) ∼
  Σv[Σs]v x δ ∼ cohOpΣtm (x [ δ ]V) +T[,]T ⁻¹

Σtm[Σs]tm : ∀ {Γ Δ : Con}{A : Ty Δ}(a : Tm A)(δ : Γ ⇒ Δ) →
  (Σtm a) [ Σs δ ]tm ≅ Σtm (a [ δ ]tm)
Σtm[Σs]tm (var x) δ = Σv[Σs]v x δ
Σtm[Σs]tm {Γ} {Δ} (coh {Δ = Δ₁} x δ A) δ₁ = congtm (cohOp
  (sym (ΣT[Σs]T A δ)))
                        ∼ cohOp (sym [⊚]T)
                        ∼ coh-eq (sym (Σs⊚ δ δ₁))
                        ∼ (cohOpΣtm (coh x (δ ⊚ δ₁) A) (sym [⊚]T)
                        ∼ cohOp (sym (ΣT[Σs]T A (δ ⊚ δ₁)))) ⁻¹

Σs•-left-id : ∀{Γ Δ : Con}(γ : Γ ⇒ Δ) → Σs {Γ} • ≡ Σs {Δ} • ⊚ Σs γ
Σs•-left-id {ε} {ε} • = refl
Σs•-left-id {ε} {Δ , A} (γ , a) = trans (Σs•-left-id γ)
  (sym (⊚wk (Σs• Δ)))
Σs•-left-id {Γ , A} {ε} • = trans (cong (λ x → x +S ΣT A)
  (Σs•-left-id {Γ} {ε} •)) (S-eq (S-eq refl ([+S]V (vS v0) {Σs• Γ} ⁻¹))
  ([+S]V v0 {Σs• Γ} ⁻¹))
Σs•-left-id {Γ , A} {Δ , A₁} (γ , a) = trans (Σs•-left-id γ)
  (sym (⊚wk (Σs• Δ)))

Σs⊚ • γ = Σs•-left-id γ
Σs⊚ {Δ} (_,_ δ {A} a) γ = S-eq (Σs⊚ δ γ) (cohOp (ΣT[Σs]T A (δ ⊚ γ))
  ∼ cohOpΣtm (a [ γ ]tm) [⊚]T ∼ (cohOp [⊚]T ∼ congtm
  (cohOp (ΣT[Σs]T A δ)) ∼ Σtm[Σs]tm a γ) ⁻¹)



ΣT[+S]T : ∀{Γ Δ : Con}(A : Ty Δ)(δ : Γ ⇒ Δ)(B : Ty Γ) →
             ΣT A [ Σs δ +S ΣT B ]T ≡ ΣT (A [ δ ]T) +T ΣT B
ΣT[+S]T A δ B = trans [+S]T (wk-T (ΣT[Σs]T A δ))

ΣsDis : ∀{Γ Δ : Con}{A : Ty Δ}(δ : Γ ⇒ Δ)(a : Tm (A [ δ ]T))
  (B : Ty Γ) → (Σs {Γ} {Δ , A} (δ , a)) +S ΣT B ≡
```

```
   Σs δ +S ΣT B , ((Σtm a) +tm ΣT B) ⟦ ΣT[+S]T A δ B ⟫
ΣsDis {Γ} {Δ} {A} δ a B = S-eq refl (wk-coh+ ∼ (cohOp (trans [+S]T
   (wk-T (ΣT[Σs]T A δ))) ∼ cong+tm2 (ΣT[Σs]T A δ)) -¹)

ΣsΣT : ∀ {Γ Δ : Con}(δ : Γ ⇒ Δ)(B : Ty Γ) → Σs (δ +S B) ≡ Σs δ +S ΣT B
ΣsΣT • _ = refl
ΣsΣT (_,_ δ {A} a) B = S-eq (ΣsΣT δ B) (cohOp (ΣT[Σs]T A (δ +S B)) ∼
       cohOpΣtm (a +tm B) [+S]T ∼ Σtm[+tm] a B ∼ cong+tm2 (ΣT[Σs]T A δ) ∼
       wk-coh+ -¹)

*'[Σs]T : {Γ Δ : Con} → (δ : Γ ⇒ Δ) → *' {Δ} [ Σs δ ]T ≡ *' {Γ}
*'[Σs]T {ε} • = refl
*'[Σs]T {Γ , A} • = trans ([+S]T {A = *' {ε}} {δ = Σs {Γ} •})
   (wk-T (*'[Σs]T {Γ} •))
*'[Σs]T {Γ} {Δ , A} (γ , a) = trans +T[,]T (*'[Σs]T γ)

ΣT[Σs]T * δ = *'[Σs]T δ
ΣT[Σs]T (_=h_ {A} a b) δ = hom≡ (Σtm[Σs]tm a δ) (Σtm[Σs]tm b δ)

Σtm[+tm] {A = A} (var x) B = cohOpV (sym (ΣT[+T] A B))
Σtm[+tm] {Γ} (coh {Δ = Δ} x δ A) B = cohOpΣtm (coh x (δ +S B) A)
   (sym [+S]T) ∼ cohOp (sym (ΣT[Σs]T A (δ +S B))) ∼ coh-eq (ΣsΣT δ B) ∼
   cohOp (sym [+S]T) -¹ ∼ cong+tm2 (sym (ΣT[Σs]T A δ))

ΣC-Contr .(ε , *) c* = ext c* v0
ΣC-Contr .(Γ , A , (var (vS x) =h var v0)) (ext {Γ} r {A} x) =
   subst (λ y → isContr (ΣC Γ , ΣT A , y))
   (hom≡ (cohOpV (sym (ΣT[+T] A A)) -¹)
   (cohOpV (sym (ΣT[+T] A A)) -¹))
   (ext (ΣC-Contr Γ r) {ΣT A} (Σv x))
```

General suspension

```
ΣC-it    : ∀{Γ}(A : Ty Γ) → Con → Con

ΣT-it    : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (ΣC-it A Δ)

Σtm-it   : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
           → Tm (ΣT-it A B)

suspend-S : {Γ Δ Θ : Con}(A : Ty Γ) → Θ ⇒ Δ →
   (ΣC-it A Θ) ⇒ (ΣC-it A Δ)

ΣC-it * Δ = Δ
ΣC-it (_=h_ {A} a b) Δ = ΣC (ΣC-it A Δ)

ΣT-it * B = B
ΣT-it (_=h_ {A} a b) B = ΣT (ΣT-it A B)

Σtm-it * t = t
Σtm-it (_=h_ {A} a b) t = Σtm (Σtm-it A t)

suspend-S * γ = γ
suspend-S (_=h_ {A} a b) γ = Σs (suspend-S A γ)

minimum-S : ∀ {Γ : Con}(A : Ty Γ) → Γ ⇒ ΣC-it A ε

ΣC-p1 :{Γ : Con}(A : Ty Γ) → ΣC (Γ , A) ≡ ΣC Γ , ΣT A
ΣC-p1 * = refl
ΣC-p1 (a =h b) = refl


ΣC-it-p1 : {Γ Δ : Con}(A : Ty Γ)(B : Ty Δ) → ΣC-it A (Δ , B) ≡
   (ΣC-it A Δ , ΣT-it A B)
ΣC-it-p1 * B = refl
ΣC-it-p1 (_=h_ {A} a b) B = cong ΣC (ΣC-it-p1 A B)
```

```
ΣC-it-S-spl' : {Γ Δ : Con}(A : Ty Γ)(B : Ty Δ) →
      (ΣC-it A Δ , ΣT-it A B) ≡ ΣC-it A (Δ , B)
ΣC-it-S-spl' * B = refl
ΣC-it-S-spl' (_=h_ {A} a b) B = cong ΣC (ΣC-it-S-spl' A B)

ΣC-it-S-spl : {Γ Δ : Con}(A : Ty Γ)(B : Ty Δ) →
      (ΣC-it A Δ , ΣT-it A B) ⇒ ΣC-it A (Δ , B)
ΣC-it-S-spl * B = IdS
ΣC-it-S-spl (_=h_ {A} a b) B = Σs (ΣC-it-S-spl A B)


ΣC-it-S-spl-¹ : {Γ Δ : Con}(A : Ty Γ)(B : Ty Δ) →
         ΣC-it A (Δ , B) ⇒ (ΣC-it A Δ , ΣT-it A B)
ΣC-it-S-spl-¹ * B = IdS
ΣC-it-S-spl-¹ (_=h_ {A} a b) B = Σs (ΣC-it-S-spl-¹ A B)


ΣC-it-S-spl2 : {Γ : Con}(A : Ty Γ)
   → (ΣC-it A ε , ΣT-it A * ,   ΣT-it A * +T _) ⇒
              ΣC (ΣC-it A ε)
ΣC-it-S-spl2 * = IdS
ΣC-it-S-spl2 (_=h_ {A} a b) = Σs (ΣC-it-S-spl2 A) ⊚ 1-1S-same
   (ΣT[+T] (ΣT-it A *) (ΣT-it A *))


ΣT-it-wk : {Γ Δ : Con}(A : Ty Γ)(B : Ty Δ) →
   (ΣT-it A *) [ ΣC-it-S-spl A B ]T ≡ ΣT-it A * +T _
ΣT-it-wk * B = refl
ΣT-it-wk (_=h_ {A} a b) B = trans (ΣT[Σs]T (ΣT-it A *)
   (ΣC-it-S-spl A B)) (trans (cong ΣT (ΣT-it-wk A B))
   (ΣT[+T] (ΣT-it A *) (ΣT-it A B)))

ΣT-it-p1 : ∀ {Γ : Con}(A : Ty Γ) → ΣT-it A * [ minimum-S A ]T ≡ A
```

```
ΣT-it-p2 : {Γ Δ : Con}(A : Ty Γ){B : Ty Δ}{a b : Tm B} →
  ΣT-it A (a =h b) ≡ (Σtm-it A a =h Σtm-it A b)
ΣT-it-p2 * = refl
ΣT-it-p2 (_=h_ {A} _ _) = cong ΣT (ΣT-it-p2 A)


ΣT-it-p3 : {Γ Δ : Con}(A : Ty Γ){B C : Ty Δ} →
  ΣT-it A (C +T B) [ ΣC-it-S-spl A B ]T ≡ ΣT-it A C +T _
ΣT-it-p3 * = trans +T[,]T (wk+S+T IC-T)
ΣT-it-p3 (_=h_ {A} a b) {B} {C} = trans (ΣT[Σs]T (ΣT-it A (C +T B))
  (ΣC-it-S-spl A B)) (trans (cong ΣT (ΣT-it-p3 A)) (ΣT[+T]
  (ΣT-it A C) (ΣT-it A B)))


minimum-S * = •
minimum-S {Γ} (_=h_ {A} a b) = ΣC-it-S-spl2 A ⊚ ((minimum-S A ,
  (a ⟦ ΣT-it-p1 A ⟫)) , (wk-tm (b ⟦ ΣT-it-p1 A ⟫)))


ΣC-it-ε-Contr : ∀{Γ Δ : Con}(A : Ty Γ) → isContr Δ →
  isContr (ΣC-it A Δ)
ΣC-it-ε-Contr * isC = isC
ΣC-it-ε-Contr (_=h_ {A} a b) isC = ΣC-Contr _ (ΣC-it-ε-Contr A isC)


wk-susp : ∀ {Γ : Con}(A : Ty Γ)(a : Tm A) → a ⟦ ΣT-it-p1 A ⟫ ≅ a
wk-susp A a = cohOp (ΣT-it-p1 A)

fci-l1 : ∀ {Γ : Con}(A : Ty Γ) → ΣT (ΣT-it A *) [ ΣC-it-S-spl2 A ]T ≡
  (var (vS v0) =h var v0)
fci-l1 * = refl
fci-l1 {Γ} (_=h_ {A} a b) = trans [⊚]T (trans (congT  (trans (ΣT[Σs]T
  (ΣT (ΣT-it A *)) (ΣC-it-S-spl2 A)) (cong ΣT (fci-l1 A)))) (hom≡
  (congtm (Σtm-p2-sp (ΣT-it A *) (ΣT-it A * +T ΣT-it A *)) ~
  1-1S-same-tm (ΣT[+T] (ΣT-it A *) (ΣT-it A *)) (var v0))
  (congtm (Σtm-p1 (ΣT-it A * +T ΣT-it A *)) ~ 1-1S-same-v0 (ΣT[+T]
  (ΣT-it A *) (ΣT-it A *)))) )

ΣT-it-p1   * = refl
ΣT-it-p1 (_=h_ {A} a b) = trans [⊚]T (trans (congT (fci-l1 A))
  (hom≡ (prf a) (prf b)))
  where
    prf : (a : Tm A) → ((a ⟦ ΣT-it-p1 A ⟫) ⟦ +T[,]T ⟫) ⟦ +T[,]T ⟫ ≅ a
    prf a = wk-coh ~ wk-coh ~ wk-susp A a

Σtm-it-p1 : {Γ Δ : Con}(A : Ty Γ){B : Ty Δ} → Σtm-it A (var v0)
  [ ΣC-it-S-spl A B ]tm ≅ var v0
Σtm-it-p1 * {B} = wk-coh ~ cohOp (wk+S+T IC-T)
Σtm-it-p1 (_=h_ {A} a b) {B} = Σtm[Σs]tm (Σtm-it A (var v0))
  (ΣC-it-S-spl A B) ~ congΣtm (Σtm-it-p1 A) ~ cohOpV (sym (ΣT[+T]
  (ΣT-it A B) (ΣT-it A B)))

Σtm-it-p2 : {Γ Δ : Con}(A : Ty Γ){B C : Ty Δ}(x : Var B) →
  (Σtm-it A (var (vS x))) [ ΣC-it-S-spl A C ]tm ≅
  Σtm-it A (var x) +tm _
Σtm-it-p2 * x = wk-coh ~ [+S]V x ~ cong+tm (IC-v x)
Σtm-it-p2 {Γ} {Δ} (_=h_ {A} a b) {B} {C} x = Σtm[Σs]tm (Σtm-it A
  (var (vS x))) (ΣC-it-S-spl A C) ~ congΣtm (Σtm-it-p2 {Γ} {Δ} A {B} x)
  ~ Σtm[+tm] (Σtm-it A (var x)) (ΣT-it A C)

ΣC-it-Contr   : ∀ {Γ Δ}(A : Ty Γ) → isContr Δ
                → isContr (ΣC-it A Δ)
ΣC-it-Contr * x = x
ΣC-it-Contr {Γ}{Δ}(_=h_ {A} a b) x = ΣC-Contr (ΣC-it A Δ) (ΣC-it-Contr A x)
```

Replacement

```
rpl-C    : ∀{Γ}(A : Ty Γ) → Con → Con
rpl-T    : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (rpl-C A Δ)
rpl-tm   : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
           → Tm (rpl-T A B)

rpl-C {Γ} A ε      = Γ
rpl-C A (Δ , B)    = rpl-C A Δ , rpl-T A B

filter   : ∀{Γ}(Δ : Con)(A : Ty Γ)
           → rpl-C A Δ ⇒ ΣC-it A Δ

rpl-T A B = ΣT-it A B [ filter _ A ]T

rpl-pr1   : {Γ : Con}(Δ : Con)(A : Ty Γ) → rpl-C A Δ ⇒ Γ

rpl-pr1 ε A = IdS
rpl-pr1 (Δ , A) A₁ = rpl-pr1 Δ A₁ +S _

filter ε A = minimum-S A
filter (Δ , A) A₁ =  ΣC-it-S-spl A₁ A ⊚ ((filter Δ A₁ +S _) ,
  var v0 ⟦ [+S]T ⟫)

rpl-T-p1 : {Γ : Con}(Δ : Con)(A : Ty Γ) → rpl-T A * ≡ A [ rpl-pr1 Δ A ]T
rpl-T-p1 ε A = trans (ΣT-it-p1 A) (sym IC-T)
rpl-T-p1 (Δ , A) A₁ = trans [⊚]T (trans (congT (ΣT-it-wk A₁ A))
  (trans +T[,]T (trans [+S]T (trans (wk-T (rpl-T-p1 Δ A₁))
  (sym [+S]T)))))

rpl-tm A a = Σtm-it A a [ filter _ A ]tm

rpl-tm-id : {Γ : Con}{A : Ty Γ} → Tm A → Tm (rpl-T {Δ = ε} A *)
rpl-tm-id x =   x ⟦ ΣT-it-p1 _ ⟫
```

```
rpl-T-p2 : {Γ : Con}(Δ : Con)(A : Ty Γ){B : Ty Δ}{a b : Tm B} →
  rpl-T A (a =h b) ≡ (rpl-tm A a =h rpl-tm A b)
rpl-T-p2 Δ A = congT (ΣT-it-p2 A)

rpl-T-p3 : {Γ : Con}(Δ : Con)(A : Ty Γ){B : Ty Δ}{C : Ty Δ}
    → rpl-T A (C +T B) ≡ rpl-T A C +T _
rpl-T-p3 _ A = trans [⊚]T (trans (congT (ΣT-it-p3 A))
  (trans +T[,]T [+S]T))

rpl-T-p3-wk : {Γ : Con}(Δ : Con)(A : Ty Γ){B : Ty Δ}{C : Ty Δ}
  {γ : Γ ⇒ rpl-C A Δ}{b : Tm ((ΣT-it A B [ filter Δ A ]T) [ γ ]T)}
    → rpl-T A (C +T B) [ γ , b ]T ≡ rpl-T A C [ γ ]T
rpl-T-p3-wk Δ A = trans (congT (rpl-T-p3 Δ A)) +T[,]T

rpl-tm-v0' : {Γ : Con}(Δ : Con)(A : Ty Γ){B : Ty Δ}
      → rpl-tm {Δ = Δ , B} A (var v0) ≅ var v0
rpl-tm-v0' Δ A = [⊚]tm (Σtm-it A (var v0)) ∼ congtm (Σtm-it-p1 A) ∼
  wk-coh ∼ wk-coh+

rpl-tm-v0 : {Γ : Con}(Δ : Con)(A : Ty Γ){B : Ty Δ}{γ : Γ ⇒ rpl-C A Δ}
      {b : Tm A}{b' : Tm ((ΣT-it A B [ filter Δ A ]T) [ γ ]T)}
    → (prf : b' ≅ b)
    → rpl-tm {Δ = Δ , B} A (var v0) [ γ , b' ]tm ≅ b
rpl-tm-v0 Δ A prf = congtm (rpl-tm-v0' Δ A) ∼ wk-coh ∼ prf

rpl-tm-vS : {Γ : Con}(Δ : Con)(A : Ty Γ){B C : Ty Δ}{γ : Γ ⇒ rpl-C A Δ}
      {b : Tm (rpl-T A B [ γ ]T)}{x : Var C} →
      rpl-tm {Δ = Δ , B} A (var (vS x)) [ γ , b ]tm ≅
      rpl-tm A (var x) [ γ ]tm
rpl-tm-vS Δ A {x = x} = congtm ([⊚]tm (Σtm-it A (var (vS x))) ∼
  (congtm (Σtm-it-p2 A x))   ∼ +tm[,]tm (Σtm-it A (var x)) ∼
  ([+S]tm (Σtm-it A (var x)))) ∼ +tm[,]tm (Σtm-it A (var x)
  [ filter _ A ]tm)
```

Basic examples of replacement

```
base-1 : {Γ : Con}{A : Ty Γ} → rpl-C A (ε , *) ≡ (Γ , A)
base-1 = cong (λ x → _ , x) (ΣT-it-p1 _)

map-1 : {Γ : Con}{A : Ty Γ} → (Γ , A) ⇒ rpl-C A (ε , *)
map-1 = 1-1S-same (ΣT-it-p1 _)
```

Lemmas about replacement

```
rpl*-A : {Γ : Con}{A : Ty Γ} → rpl-T {Δ = ε} A * [ IdS ]T ≡ A
rpl*-A = trans IC-T (ΣT-it-p1 _)

rpl*-a : {Γ : Con}(A : Ty Γ){a : Tm A} → rpl-tm {Δ = ε , *} A
  (var v0) [ IdS , a ⟦ rpl*-A ⟫ ]tm ≅ a
rpl*-a A = rpl-tm-v0 ε A   (cohOp (rpl*-A {A = A}))

rpl*-A2 : {Γ : Con}(A : Ty Γ){a : Tm (rpl-T A (* {ε}) [ IdS ]T)}
      → rpl-T A (* {ε , *}) [ IdS , a ]T ≡ A
rpl*-A2 A = trans (rpl-T-p3-wk ε A) rpl*-A

rpl-xy :   {Γ : Con}(A : Ty Γ)(a b : Tm A)
  → rpl-T {Δ = ε , * , *} A (var (vS v0) =h var v0)
  [ IdS , a ⟦ rpl*-A ⟫ , b ⟦ rpl*-A2 A ⟫ ]T ≡ (a =h b)
rpl-xy A a b =   trans (congT (rpl-T-p2 (ε , * , *) A))
      (hom≡ ((rpl-tm-vS (ε , *) A)   ∼ rpl*-a A)
        (rpl-tm-v0 (ε , *) A (cohOp (rpl*-A2 A))))

rpl-sub : (Γ : Con)(A : Ty Γ)(a b : Tm A) → Tm (a =h b)
      → Γ ⇒ rpl-C A (ε , * , * , (var (vS v0) =h var v0))
rpl-sub Γ A a b t = IdS , a ⟦ rpl*-A ⟫ , b ⟦ rpl*-A2 A ⟫ , t ⟦ rpl-xy A a b ⟫
```

C.2.2 First-level Groupoid Structure

```
Coh-rpl  : ∀{Γ Δ}(A : Ty Γ)(B : Ty Δ) → isContr Δ
           → Tm (rpl-T A B)
Coh-rpl {_} {Δ} A _ isC = coh (ΣC-it-ε-Contr A isC) _ _
```

Reflexivity

```
refl*-Tm : Tm {x:*} (var v0 =h var v0)
refl*-Tm = Coh-Contr c*
```

Symmetry

```
sym*-Ty : Ty x:*,y:*,α:x=y
sym*-Ty = vY =h vX

sym*-Tm : Tm {x:*,y:*,α:x=y} sym*-Ty
sym*-Tm = Coh-Contr (ext c* v0)
```

Transitivity (composition)

```
trans*-Ty : Ty x:*,y:*,α:x=y,z:*,β:y=z
trans*-Ty = (vX +tm _ +tm _) =h vZ

trans*-Tm : Tm trans*-Ty
trans*-Tm = Coh-Contr (ext (ext c* v0) (vS v0))
```

```
refl-Tm      : {Γ : Con}(A : Ty Γ)
             → Tm (rpl-T {Δ = x:*} A (var v0 =h var v0))
refl-Tm A    = rpl-tm A refl*-Tm

sym-Tm       : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A sym*-Ty)
sym-Tm A     = rpl-tm A sym*-Tm

trans-Tm     : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A trans*-Ty)
trans-Tm A   = rpl-tm A trans*-Tm




reflX : Tm (vX =h vX)
reflX = refl-Tm * +tm _ +tm _

reflY : Tm (vY =h vY)
reflY = refl-Tm * +tm _

m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q : Con
m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q = x:*,y:*,α:x=y,z:*,β:y=z , * ,
   (var (vS (vS v0)) =h var v0)

vM : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} *
vM = var (vS (vS (vS (vS (vS (vS v0))))))

vN : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} *
vN = var (vS (vS (vS (vS (vS v0)))))

vMN : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} (vM =h vN)
vMN = var (vS (vS (vS (vS v0))))

vP : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} *
vP = var (vS (vS (vS v0)))

vNP : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} (vN =h vP)
vNP = var (vS (vS v0))

vQ : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} *
vQ = var (vS v0)

vPQ : Tm {m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q} (vP =h vQ)
vPQ = var v0

Ty-G-assoc* : Ty m:*,n:*,α:m=n,p:*,β:n=p,q:*,γ:p=q
Ty-G-assoc* = (trans*-Tm [ ((((• , vM) , vP) ,
  (trans*-Tm [ pr1 ⊚ pr1 ]tm)) , vQ) , vPQ ]tm =h
  trans*-Tm [ (pr1 ⊚ pr1 ⊚ pr1 ⊚ pr1 , vQ) ,
  (trans*-Tm [ ((((• , vN) , vP) , vNP) , vQ) ,
  vPQ ]tm) ]tm)




Tm-right-identity* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ IdS , vY , reflY ]tm
  =h vα)
Tm-right-identity* = Coh-Contr (ext c* v0)

Tm-left-identity* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ ((IdS ⊚ pr1 ⊚ pr1) , vX) ,
  reflX , vY , vα ]tm =h vα)
Tm-left-identity* = Coh-Contr (ext c* v0)

Tm-right-inverse* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ (IdS , vX) , sym*-Tm ]tm
  =h reflX)
Tm-right-inverse* = Coh-Contr (ext c* v0)
```

```
Tm-left-inverse* :
  Tm {x:*,y:*,α:x=y} (trans*-Tm [ ((• , vY) , vX , sym*-Tm ,
  vY) , vα ]tm =h reflY)
Tm-left-inverse* = Coh-Contr (ext c* v0)

Tm-G-assoc*   : Tm Ty-G-assoc*
Tm-G-assoc*   = Coh-Contr (ext (ext (ext c* v0) (vS v0))
                (vS v0))

Tm-G-assoc      : ∀{Γ}(A : Ty Γ)
                → Tm (rpl-T A Ty-G-assoc*)
Tm-G-assoc A    = rpl-tm A Tm-G-assoc*
```

C.3 Sematics

Globular types

```
record Glob : Set₁ where
  constructor _||_
  field
    |_|   : Set
    hom   : |_| → |_| → ∞ Glob

open Glob public




Idω      : (A : Set) → Glob
Idω A    = A || (λ a b → ♯ Idω (a ≡ b))
```

Semantic interpretation

```
record Semantic (G : Glob) : Set₁ where
  field
    ⟦_⟧C      : Con → Set
    ⟦_⟧T      : ∀{Γ} → Ty Γ → ⟦ Γ ⟧C → Glob
    ⟦_⟧tm     : ∀{Γ A} → Tm A → (γ : ⟦ Γ ⟧C)
              → | ⟦ A ⟧T γ |
    ⟦_⟧S      : ∀{Γ Δ} → Γ ⇒ Δ → ⟦ Γ ⟧C → ⟦ Δ ⟧C
    π         : ∀{Γ A} → Var A → (γ : ⟦ Γ ⟧C)
              → | ⟦ A ⟧T γ |
    ⟦_⟧C-β1     : ⟦ ε ⟧C ≡ ⊤
    ⟦_⟧C-β2     : ∀ {Γ A} → ⟦ Γ , A ⟧C ≡
                Σ ⟦ Γ ⟧C (λ γ   → | ⟦ A ⟧T γ |)

    ⟦_⟧T-β1     : ∀{Γ}{γ : ⟦ Γ ⟧C} → ⟦ * ⟧T γ ≡ G
    ⟦_⟧T-β2     : ∀{Γ A u v}{γ : ⟦ Γ ⟧C}
                → ⟦ u =h v ⟧T γ ≡
                ♭ (hom (⟦ A ⟧T γ) (⟦ u ⟧tm γ) (⟦ v ⟧tm γ))

    semSb-T     : ∀ {Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)(γ : ⟦ Γ ⟧C)
                → ⟦ A [ δ ]T ⟧T γ ≡ ⟦ A ⟧T (⟦ δ ⟧S γ)

    semSb-tm    : ∀{Γ Δ}{A : Ty Δ}(a : Tm A)(δ : Γ ⇒ Δ)
                (γ : ⟦ Γ ⟧C) → subst |_| (semSb-T A δ γ)
                (⟦ a [ δ ]tm ⟧tm γ) ≡ ⟦ a ⟧tm (⟦ δ ⟧S γ)

    semSb-S     : ∀ {Γ Δ Θ}(γ : ⟦ Γ ⟧C)(δ : Γ ⇒ Δ)
                (θ : Δ ⇒ Θ) → ⟦ θ ⊚ δ ⟧S γ ≡
                ⟦ θ ⟧S (⟦ δ ⟧S γ)

    ⟦_⟧tm-β1    : ∀{Γ A}{x : Var A}{γ : ⟦ Γ ⟧C}
                → ⟦ var x ⟧tm γ ≡ π x γ
```

```
⟦_⟧S-β1   : ∀{Γ}{γ : ⟦ Γ ⟧C}
          → ⟦ • ⟧S γ ≡ coerce ⟦_⟧C-β1 tt

⟦_⟧S-β2   : ∀{Γ Δ}{A : Ty Δ}{δ : Γ ⇒ Δ}{γ : ⟦ Γ ⟧C}
          {a : Tm (A [ δ ]T)} → ⟦ δ , a ⟧S γ
          ≡ coerce ⟦_⟧C-β2 ((⟦ δ ⟧S γ) ,
          subst ∣_∣ (semSb-T A δ γ) (⟦ a ⟧tm γ))

semWk-T   : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
          → ⟦ A +T B ⟧T (coerce ⟦_⟧C-β2 (γ , v)) ≡
          ⟦ A ⟧T γ


semWk-S   : ∀ {Γ Δ B}{γ : ⟦ Γ ⟧C}{v : ∣ ⟦ B ⟧T γ ∣}
          → (δ : Γ ⇒ Δ) → ⟦ δ +S B ⟧S
          (coerce ⟦_⟧C-β2 (γ , v)) ≡ ⟦ δ ⟧S γ

semWk-tm : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
          → (a : Tm A) → subst ∣_∣ (semWk-T γ v)
            (⟦ a +tm B ⟧tm (coerce ⟦_⟧C-β2 (γ , v)))
              ≡ (⟦ a ⟧tm γ)

π-β1    : ∀{Γ A}(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ A ⟧T γ ∣)
        → subst ∣_∣ (semWk-T γ v)
          (π v0 (coerce ⟦_⟧C-β2 (γ , v))) ≡ v

π-β2    : ∀{Γ A B}(x : Var A)(γ : ⟦ Γ ⟧C)(v : ∣ ⟦ B ⟧T γ ∣)
        → subst ∣_∣ (semWk-T γ v) (π (vS {Γ} {A} {B} x)
          (coerce ⟦_⟧C-β2 (γ , v))) ≡ π x γ

⟦coh⟧     : ∀{Θ} → isContr Θ → (A : Ty Θ)
          → (θ : ⟦ Θ ⟧C) → ∣ ⟦ A ⟧T θ ∣
```

Bibliography

[1] Irrelevance – agda wiki. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Irrelevance.

[2] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with Quotient Types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.

[3] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Annual IEEE Symposium on Logic in Computer Science*, pages 412–420. IEEE Computer Society, 1999. ISBN 0-7695-0158-3.

[4] Thorsten Altenkirch. The Coherence Problem in HoTT. 2014.

[5] Thorsten Altenkirch and James Chapman. Big-step normalisation. *J. Funct. Program.*, 19(3-4):311–333, 2009. doi: 10.1017/S0956796809007278. URL http://dx.doi.org/10.1017/S0956796809007278.

[6] Thorsten Altenkirch and Nicolai Kraus. Setoids are not an LCCC, 2012.

[7] Thorsten Altenkirch and Ondřej Rypáček. A syntactical Approach to Weak ω-Groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, volume 16 of *LIPIcs*, pages 16–30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. ISBN 978-3-939897-42-2.

[8] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007*, pages 57–68. ACM, 2007. ISBN 978-1-59593-677-6.

[9] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. Pisigma: Dependent Types without the Sugar. submitted for publication, November 2009.

[10] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable Quotients in Type Theory. 2011.

[11] Thorsten Altenkirch, Nuo Li, and Ondřej Rypáček. Some constructions on ω-groupoids. In *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, LFMTP '14, pages 4:1–4:8, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2817-3. doi: 10.1145/2631172.2631176. URL http://doi.acm.org/10.1145/2631172.2631176.

[12] Dimitri Ara. On the homotopy theory of grothendieck ∞-groupoids. *Journal of Pure and Applied Algebra*, 217(7):1237–1278, 2013.

[13] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Math. Proc. Cambridge Philos. Soc.*, 146(1):45–55, 2009. ISSN 0305-0041. doi: 10.1017/S0305004108001783. URL http://dx.doi.org/10.1017/S0305004108001783.

[14] John Baez. The Homotopy Hypothesis, 2007. URL http://math.ucr.edu/home/baez/homotopy/homotopy.pdf.

[15] Gilles Barthe and Herman Geuvers. Congruence Types. In *Proceedings of CSL'95*, pages 36–51. Springer-Verlag, 1996.

[16] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[17] Andrej Bauer and Peter LeFanu Lumsdaine. A Coq proof that Univalence Axioms implies Functional Extensionality. 2013.

[18] Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs*

(TYPES 2013), volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-72-9. doi: http://dx.doi.org/10.4230/LIPIcs.TYPES.2013.107. URL http://drops.dagstuhl.de/opus/volltexte/2014/4628.

[19] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, New York, 1985. ISBN 0-387-15066-8.

[20] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, Berlin, Heidelberg, 1985.

[21] Ana Bove and Peter Dybjer. *Dependent Types at Work*. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6. doi: http://dx.doi.org/10.1007/978-3-642-03153-3_2.

[22] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda — a Functional Language with Dependent Types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: http://dx.doi.org/10.1007/978-3-642-03359-9_6.

[23] Douglas S. Bridges. Constructive mathematics: a foundation for computable analysis. *Theoretical Computer Science*, 219(1-2):95 – 109, 1999. ISSN 0304-3975. doi: DOI:10.1016/S0304-3975(98)00285-0. URL http://www.sciencedirect.com/science/article/B6V1G-3WXWSM9-6/2/c1225a60fbe1d641e225bdf749181845.

[24] Guillaume Brunerie. Syntactic Grothendieck weak ∞-groupoids, 2013.

[25] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005. doi: 10.2168/LMCS-1(2:1)2005. URL http://dx.doi.org/10.2168/LMCS-1(2:1)2005.

[26] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theor. Comput. Sci.*, 412(38):5006–5024, 2011. doi: 10.1016/j.tcs.2011.04.024. URL http://dx.doi.org/10.1016/j.tcs.2011.04.024.

[27] Pierre Clairambault. From categories with families to locally cartesian closed categories. *Project Report, ENS Lyon*, 2006.

[28] Cyril Cohen. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving*, pages 213–228, 2013.

[29] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing Mathematics with The Nuprl Proof Development System, 1986.

[30] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. ISBN 978-0-13-451832-9. URL http://dl.acm.org/citation.cfm?id=10510.

[31] Thierry Coquand. Pattern Matching with Dependent Types. In *Types for Proofs and Programs*, 1992.

[32] Thierry Coquand. Types as Kan Simplicial Sets. Accessed: 2014-09-10, 12 2012. URL http://www.cse.chalmers.se/~coquand/stockholm.pdf.

[33] Thierry Coquand. Type Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2014 edition, 2014. URL http://plato.stanford.edu/archives/sum2014/entries/type-theory/.

[34] Pierre Courtieu. **Normalized types**. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[35] Nils Anders Danielsson. Sets with decidable equality have unique identity proofs. URL http://www.cse.chalmers.se/~nad/listings/equality/Equality.Decidable-UIP.html. Accessed: 2014-09-20.

[36] Nils Anders Danielsson. Bag equivalence via a Proof-Relevant Membership Relation. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem*

Proving - Third International Conference, ITP 2012, volume 7406 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2012. ISBN 978-3-642-32346-1.

[37] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, Declaratively. In *Proceedings of Mathematics of Program Construction, 10th International Conference, MPC 2010*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2010. ISBN 978-3-642-13320-6.

[38] Peter Dybjer. Internal Type Theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[39] Martín Escardó. Counterexample on local continuity. URL http://www.cs.bham.ac.uk/~mhe/agda/FailureOfTotalSeparatedness.html. Accessed: 2014-08-20.

[40] Martín Escardó and Chuangjie Xu. The inconsistency of a brouwerian continuity principle with the Curry-Howard interpretation. 2014. URL http://www.cs.bham.ac.uk/~mhe/papers/escardo-xu-inconsistency-continuity.pdf.

[41] Kurt Godel. *On formally undecidable propositions of principia mathematica and related systems*. Dover, 1992.

[42] Georges Gonthier. Formal Proof the Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008. URL http://www.ams.org/notices/200811/tx081101382p.pdf.

[43] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Theorem Proving in Higher Order Logics*, pages 98–113. Springer, 2005.

[44] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.

[45] Michael Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4):413–436, 1998. URL http://journals.cambridge.org/action/displayAbstract?aid=44199.

[46] Martin Hofmann. On the Interpretation of Type Theory in Locally Cartesian Closed Categories. In *Computer Science Logic, 8th International Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1994. ISBN 3-540-60017-5.

[47] Martin Hofmann. A Simple Model for Quotient Types. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1995. ISBN 3-540-59048-X. doi: 10.1007/BFb0014055. URL http://dx.doi.org/10.1007/BFb0014055.

[48] Martin Hofmann. *Extensional concepts in Intensional Type Theory*. PhD thesis, School of Informatics., 1995.

[49] Martin Hofmann. Conservativity of Equality Reflection over Intensional Type Theory. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '95, pages 153–164, London, UK, 1996. Springer-Verlag. ISBN 3-540-61780-9. URL http://portal.acm.org/citation.cfm?id=646536.695865.

[50] Martin Hofmann. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[51] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, New York, 1998. Oxford University Press.

[52] Peter V. Homeier. Quotient Types. In *In TPHOLs 2001: Supplemental Proceedings*, page 0046, 2001.

[53] Antonius JC Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.

[54] Bart Jacobs. Quotients in Simple Type Theory. *Manuscript, Math. Inst*, 1994.

[55] Krzysztof Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The Simplicial Model of Univalent Foundations. arXiv:1211.2851, 2012.

[56] Stephen C Kleene and J Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.

[57] Nicolai Kraus. Non-Normalizability of Cauchy Sequences. 2014. URL http://www.cs.nott.ac.uk/~ngk/normalizability.pdf.

[58] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of Anonymous Existence in Martin-Löf Type Theory. 2014.

[59] Nuo Li. Representing numbers in Agda. Technical report, School of Computer Science, University of Nottingham, 2010. Final year dissertation.

[60] Nuo Li. Some constructions on ω-groupoids: codes, 2014.

[61] Robert S. Lubarsky. On the Cauchy Completeness of the Constructive Cauchy Reals. *Electr. Notes Theor. Comput. Sci.*, 167:225–254, 2007.

[62] Peter LeFanu Lumsdaine. Weak ω-categories from intensional type theory. *Logical Methods in Computer Science*, 6(3), 2010. doi: 10.2168/LMCS-6(3:24)2010. URL http://dx.doi.org/10.2168/LMCS-6(3:24)2010.

[63] G. Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. *ArXiv e-prints*, September 2010.

[64] Per Martin-Löf. A Theory of Types. Technical Report 71–3, University of Stockholm, 1971.

[65] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[66] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984. ISBN 88-7088-105-9.

[67] 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.

[68] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104, pages 153 – 175. Elsevier, 1982.

[69] Conor McBride. Elimination with a Motive. In *Types for Proofs and Programs, International Workshop, TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000. ISBN 3-540-43287-6.

[70] N.P. Mendler. Quotient types via coequalizers in Martin-Löf type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.

[71] Aleksey Nogin. Quotient types: A Modular Approach. In *ITU-T Recommendation H.324*, pages 263–280. Springer-Verlag, 2002.

[72] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction.* Clarendon Press, New York, NY, USA, 1990. ISBN 0-19-853814-6.

[73] Ulf Norell. Dependently typed programming in Agda. Available at: http://www.cse.chalmers.se/ ulfn/papers/afp08tutorial.pdf, 2008.

[74] Erik Palmgren. On universes in type theory. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, New York, 1998. Oxford University Press.

[75] Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. Preprint, June 2014.

[76] Bertrand Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1903.

[77] Matthieu Sozeau and Nicolas Tabareau. Internalization of the Groupoid Interpretation of Type Theory. *Types 2014*, 2014.

[78] Thomas Streicher. *Investigations into intensional type theory.* PhD thesis, Habilitation thesis, Ludwig-Maximilians-University Munich, 1993.

[79] Thomas Streicher. A model of type theory in simplicial sets: A brief introduction to Voevodsky's homotopy type theory. pages 45–49, 2014.

[80] Laurent Théry. A selected bibliography on formalised mathematics. URL http://www-sop.inria.fr/marelle/personnel/Laurent.Thery/math.html. Accessed: 2014-09-20.

[81] A. S. Troelstra. From Constructivism to Computer Science. pages 233–252, 1999.

[82] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[83] Mark van Atten and Dirk van Dalen. Arguments for the continuity principle. *Bulletin of Symbolic Logic*, 8(3):329–347, 2002. URL http://www.math.ucla.edu/~asl/bsl/0803/0803-001.ps.

[84] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[85] Paul van der Walt. *Reflection in Agda*. PhD thesis, Master's thesis, Universiteit Utrecht, 2012.

[86] Vladimir Voevodsky. Generalities on hSet - Coq library hSet, . URL http://www.math.ias.edu/~vladimir/Foundations_library/hSet.html.

[87] Vladimir Voevodsky. Univalent Foundations Project. . URL http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf.

[88] Vladimir Voevodsky. A very short note on the homotopy λ-calculus. 2006. URL http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf.

[89] Vladimir Voevodsky. A very short note on homotopy λ-calculus. Available at: http://math.ucr.edu/home/baez/Voevodsky_note.ps, 2006.

[90] Vladimir Voevodsky. Resizing Rules - their use and semantic justification, 9 2011. URL http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_Bergen.pdf.

[91] Vladimir Voevodsky. A universe polymorphic type system. 2012. URL http://uf-ias-2012.wikispaces.com/file/view/Universe+polymorphic+type+sytem.pdf.

[92] Vladimir Voevodsky, Thierry Coquand, and Benno van den Berg. Semi-simplicial types. URL https://uf-ias-2012.wikispaces.com/Semi-simplicial+types. Accessed: 2014-09-20.

[93] Michael Warren. The strict ω-groupoid interpretation of type theory. In *Models, Logics and Higher-Dimensional Categories*, CRM Proc. Lecture Notes 53, pages 291–340. Amer. Math. Soc., 2011.

[94] The Agda Wiki. Main page, 2014. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage. [Online; accessed 15-June-2014].

[95] Wikipedia. Lazy evaluation — Wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/wiki/Lazy_evaluation. [Online; accessed 20-April-2010].