

# 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 serves as not only a foundation of constructive mathematics but also a dependently typed programming language. Dependent types are types depend on values of other types. Type Theory is based on the Curry-Howard isomorphism which relates computer programs with mathematical proofs such that we can do computer-aided formal reasoning and write certified programs in programming languages like Agda, Epigram etc. Martin Löf proposed two kinds of variants of Type Theory which are differentiated on the treatment of equalities. In Intensional Type Theory, propositional equality induced by identity types does not imply definitional equality, and the 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 usually more popular, however there are some important extensional concepts missing, such as functional extensionality and quotient types etc.

This thesis is about quotient types. A quotient type is a new type whose equality is redefined by an given equivalence relation. However in the usual formulation of Intensional Type Theory, there is no such type former to do it. We will also lose canonicity if we add quotient types into Intensional Type Theory as axioms. In this thesis, we first investigate what are the syntax of quotient types we expect to have, and explain the syntax with categorical counterparts. 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 one 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 it is only a compromise approach since it does not apply to all quotients. It seems that we can not 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  $\omega$ -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 investigate a new extension of Martin-Löf type theory called Homotopy Type Theory where types are interpreted as weak  $\omega$ -groupoids. It can be seen as a generalisation of groupoid model and the extensional concepts including quotient types are available. We also introduce a syntactic encoding of weak  $\omega$ -groupoids which is considered as a first step to build a weak  $\omega$ -groupoids model in Intensional Type Theory. All these implementations have been done 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 was later developed into this project. He has always been patient in explaining my questions and taught me a lot on research. I would also like to thank him for his guidance and feedbacks on this thesis. I would also thank my second supervisor Thomas Anberree who introduced me functional programming and he has given me a lot precious advices on my project. I would also like to thank Venanzio Capretta who examined my first and second year reports and provided me helpful feedbacks.

I would like to thank Ambrus Kaposi and Nicolai Kraus for their great help in my thesis writing and comments on my drafts. My friends Nicolai Kraus, Ambrus Kaposi, Christian Sattler, Paulo Caprotti, Florent Balestrieri, Gabe Dijkstra, Ivan Perez, Neil Sculthorpe and all other PhD students in our lab has helped me a lot in my project by either teaching me mathematics, discussing on research topics and playing badminton to exercise our bodies. The Functional programming lab is like a lively family and I really enjoy 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 discussion topics some of which are related to part of this work, especially Guillaume Brunerie whose proposal made it possible. I would also thank Nordvall Forsberg who provided important discussions on 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 have always been very supportive of me, even though I am 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 helps me a lot. She has given my many advices and has sent mails to me. Without their support I would never achieve my goals.

Finally, I would like to thank the School of Computer Science and international office in the University of Nottingham who financially support this project by providing important studentship to me.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Quotient types . . . . .	2
1.2	Overview . . . . .	6
<b>2</b>	<b>Type Theory</b>	<b>9</b>
2.1	A brief history of Type Theory . . . . .	10
2.2	The formal system of Type Theory . . . . .	15
2.3	An implementation of Type Theory : Agda . . . . .	20
2.4	Extensional concepts . . . . .	25
2.5	An Intensional Type Theory with <b>Prop</b> . . . . .	30
2.6	Homotopy Type Theory . . . . .	31
2.7	Summary . . . . .	36
<b>3</b>	<b>Quotient Types</b>	<b>39</b>
3.1	Quotients in Type Theory . . . . .	39
3.2	Quotients are coequalizers . . . . .	45
3.3	Quotients as an adjunction . . . . .	48
3.4	Quotients in Homotopy Type Theory . . . . .	51
3.5	Related work . . . . .	60
3.6	Summary . . . . .	64
<b>4</b>	<b>Definable Quotients</b>	<b>65</b>
4.1	Algebraic structures of quotients . . . . .	66
4.2	Integers . . . . .	70
4.3	Rational numbers . . . . .	73
4.4	The application of definable quotients . . . . .	78
4.5	Related work . . . . .	85
4.6	Summary . . . . .	86
<b>5</b>	<b>Undefinable Quotients</b>	<b>87</b>
5.1	Definability via normalisation . . . . .	87

5.2	Real numbers as Cauchy sequences . . . . .	89
5.3	$\mathbb{R}_0/\sim$ is undefinable via normalisation . . . . .	91
5.4	Other examples . . . . .	98
5.5	Related work . . . . .	102
5.6	Summary . . . . .	102
<b>6</b>	<b>The Setoid Model</b>	<b>105</b>
6.1	Introduction . . . . .	106
6.2	Metatheory . . . . .	107
6.3	Categories with families . . . . .	112
6.4	Related work . . . . .	121
6.5	Summary . . . . .	122
<b>7</b>	<b>Syntactic <math>\omega</math>-groupoids</b>	<b>125</b>
7.1	Syntax of weak $\omega$ -groupoids . . . . .	127
7.2	Some Important Derivable Constructions . . . . .	134
7.3	Semantics . . . . .	144
7.4	Related work . . . . .	148
7.5	Summary . . . . .	148
<b>8</b>	<b>Conclusion and Future Work</b>	<b>151</b>
<b>A</b>	<b>Definable quotient structures</b>	<b>155</b>
A.1	Rational numbers . . . . .	169
<b>B</b>	<b>Category with families of setoids</b>	<b>173</b>
B.1	Metatheory . . . . .	173
B.2	Categories with families . . . . .	175
<b>C</b>	<b>syntactic weak <math>\omega</math>-groupoids</b>	<b>191</b>
C.1	Syntax of $\mathcal{T}_{\infty\text{-groupoid}}$ . . . . .	191
C.2	Some Important Derivable Constructions . . . . .	203
C.3	Semantics . . . . .	220
	<b>Bibliography</b>	<b>223</b>



# 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 a proposition with a type such that a proof of it is a term of that type. As a programming language, it means that we can express a specification as a type of the programs satisfying it. Moreover, one can write programs and reason about them in it, thus write certified programs. There are a number of implementations of it, such as NuPRL, LEGO, Coq, Agda, Epigram, Pi-Sigma etc.

As a foundation of constructive mathematics, there have been a lot of mathematics formalised in it, for example a formal proof of the four-colour theorem by Georges Gonthier [40]<sup>1</sup>. The formalisation of mathematics in programming languages not only provides mathematicians a powerful tool to constructively prove theorems with computerised verification, but also helps in program specification and reasoning.

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*

---

<sup>1</sup>More formalised mathematics can be found in [78]

*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 the other side, we have *identity types* which are types as propositions expressing the equality between two terms of the same type, so it is called propositional equality. 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, the propositional equality is intensional. The extensional propositional equality is not inhabited, for example, equality between two point-wise equal functions, two logically equivalent propositions, and two “equivalence classes” of a quotient  $[a], [b]$  where  $a \sim b$ . In fact there is a list of extensional concepts (see Section 2.4) which are useful, justifiable but not available in ITT. Nevertheless ITT is still preferable to ETT as the basis for programming languages, because its type checking is decidable so that it has good computational behaviours. 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 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:  $\forall a, b \in A, a \sim b \rightarrow b \sim a$
- transitive:  $\forall a, b, c \in A, a \sim b \rightarrow b \sim c \rightarrow a \sim c$ .

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:

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

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 [29]. There are some problems of it: we can recover the witness of the equality between two equal elements in quotient types [70]; we can define functions on quotient types which do not respect the equivalence relation (see Section 3.5).

Because of the good computational property of Intensional Type Theory, we would like to have quotient types in Intensional Type Theory as well. However in 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  $(A, \sim, eq_\sim)$  (usually written as just  $(A, \sim)$ ) consists of

1. a set (type)  $A : \mathbf{Set}$ ,
2. a binary relation  $\sim : A \rightarrow A \rightarrow \mathbf{Prop}$ , and
3. it is an equivalence, i.e. there are proofs that it is reflexive, symmetric and transitive.

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

A function  $f : A \rightarrow B$  is well-defined on a setoid  $(A, \sim)$  only if it respects  $\sim$  or it is *compatible* with  $\sim$ :

**Definition 1.2.** We say a function  $f : A \rightarrow B$  is *compatible* with  $\sim$  if

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

However using setoids to represents 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 as functions between setoids, equalities on setoids, products on setoids, etc. Moreover, considering a quotient  $A/\sim$  whose  $A$  is already defined as a setoid, it is essential to define generalised "setoids" on setoids. 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 can be represented as pairs of natural numbers  $\mathbb{N} \times \mathbb{N}$  which are equivalent if the subtraction of first number from the second are equal. 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 these quotients, the set definition can be seen as the normal form of "equivalence classes" which is usually described by a mapping from setoid representation to set representation called **normalisation function**. In this thesis we say that such quotients *definable via a normalisation (function)* (see Chapter 4).

However, there are also some quotients that are not definable via normalisation, for example, the set of real numbers represented by the Cauchy sequences of rational numbers, the finite multisets represented as lists quotiented by permutation equivalence (or bag equivalence [35]), the non-terminating programs represented by 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 the 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 some research has been done by Martin Hofmann [45, 46], 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.

It has been unclear for a long time what identity types are in Intensional Type Theory. Intuitively, the uniqueness of identity proof (UIP), namely two terms of a same identity type are always propositional 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 an extra eliminator  $K$  suggested by Thomas Streicher [76]. Hofmann and Streicher further [49] propose a groupoid interpretation of Intensional Type Theory where  $K$  is refuted and then UIP fails. The groupoid interpretation can be seen as a generalisation of setoid one, where the identity type is not a proposition but a set.

In fact, it is more accurate to interpret a type as an  $\infty$ -groupoid which is a generalisation of a groupoid. Since Grothendieck's homotopy hypothesis states that  $\infty$ -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 where types are interpreted as spaces (abstractly). Because the coherence conditions like the groupoid laws hold as propositional equality in Intensional Type Theory, it is better to interpret types as *weak*  $\omega$ -groupoids. However, describing weak  $\omega$ -groupoids using globular sets seems to be very involved due to coherence conditions. A more commonly used approach is therefore to define them in terms of Kan simplicial sets or cubical sets. Nevertheless, it is possible to build a syntactic type theory to describe weak  $\omega$ -groupoids in Intensional Type Theory (see Chapter 7).

In Homotopy Type Theory, the most important axiom is *univalence* which was suggested by Voevodsky [86]. Roughly speaking, univalence states that identity corresponds to equivalence. Many extensional concepts are derivable including functional extensionality, propositional extensionality, quotient types. For example, Voevodsky has proposed an impredicative encoding of quotient types (see Section 3.4.1). However the computational interpretation of univalence remains an open problem.

Quotient types can be applied in both formalisation of mathematics and program verification. As we mentioned, one of the fundamental mathematical notions *real numbers* can be defined by as a quotient where the base set is the Cauchy sequences of rational numbers. From programming perspective, it also provides more algebraic datatypes and enable us to reason about infinite types and semantics-based verification of concurrent programs as suggested by Hofmann [46].

## 1.2 Overview

In Chapter 2, we introduce Martin-Löf type theory as the background of our study. We provide a brief history of it and some basic rules of it. We also introduce the main tool we use – Agda, which is a dependently typed functional programming language based on intensional version of Martin-Löf type theory. Then we discuss the missing extensional concepts in Intensional Type Theory excluding quotient types. We also introduce the new interpretation of Homotopy Type Theory, where

we have univalence, and higher inductive types which allow constructors for internal equalities. Finally we discuss the extensional concepts in it and the potential computational interpretations of it in Intensional Type Theory.

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 because of the different interpretation of sets, the traditional quotient types are just quotient *sets*. We first introduce the Voevodsky's impredicative encoding of quotient sets together with a set of 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 achievements, the definable quotient algebraic structures. We observe that there are some quotients which are definable inductively, which means a new type former of quotient is not necessary for them. A definable quotient consists of a setoid representation  $(A, \sim)$ , a set representation  $Q$  and a normalisation function  $[\_] : A \rightarrow Q$  which gives us the normal form for each "equivalence class". As a example, integers can be encoded as the quotients 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 abstract the relation between two representations and provide a flexible way of conversions. In fact, it can be seen as a manual construction of the quotient types.

In Chapter 5, we discuss quotients that are undefinable as an inductive type with a normalisation function, such as the real numbers, finite multisets and partiality monads. 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 not necessarily inhabited in Martin-Löf type theory but holds meta-theoretically. We prove that  $R_0/\sim$  is *connected*, and it implies that all endofunctions that respect the equivalence relation of Cauchy sequences is constant. Therefore there is no definable normalisation endofunction for Cauchy sequences. We also similarly prove that non-terminating programs

encoded as partiality monad with weak bisimilarity as equivalence relation is also undefinable with a normalisation function. We can not define normalisation function for finite multisets represented by lists quotiented by permutation, unless the underlying set has a decidable total order so that a sorted list can be the canonical choice for each equivalence relation.

In Chapter 6, we discussed several models where quotient types are available. We present an implementation on the setoid model approach to encode extensional concepts. The work is mainly extending the setoid model done by Altenkirch in [3] to quotient types. Some other models including models of Homotopy Type Theory are also mentioned.

In Chapter 7, we present a new formalisation of the syntax of weak  $\omega$ -groupoids in Agda using heterogeneous equality. We show how to recover basic constructions on  $\omega$ -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 shown our Agda code corresponding to the work in Chapter 4, Chapter 6 and Chapter 7.



# Chapter 2

## Type Theory

A type theory usually refers to a formal system in which every term always has 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 languages in which we can write certified programs. There are a variety of type theories, like Russell's theory of types, simply typed  $\lambda$ -calculus, Gödel's System T [39] 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 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 also introduce the main 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, George 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 the 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 and Frege found that they have to make a distinction between objects, predicates, predicates of predicates, etc. Then Russell proposed the theory of types [75] 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 state 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 term are usually constructed using a list of constructors binding to its type. Hence a term of integer  $2 : \mathbb{Z}$  is different to  $2 : \mathbb{N}$  in type theory, while it is not the case in set theory.

Following the idea of theories of types, various type theories have been developed since then. Simply typed lambda calculus (or Church's theory of types) is the first type theory to introduce functions as primitive objects [32]. It was originally introduced by Alonzo Church in 1940 to avoid the Kleene-Rosser paradox [54] 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:*

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

It is applied to various fields including computer science. For instance, Haskell was originally based on one of the variants of lambda calculus called System F<sup>1</sup>.

In 1970s, Per Martin-Löf [62, 66] developed his profound intuitionistic type theory (also called Martin-Löf type theory). In this thesis, we will use *Type Theory* specially for it if unambiguous. It serves as a foundation of constructive mathematics [64] and also can be used as a functional programming language [79] in which the evaluation of a well-typed program always terminates [71].

From early type theories like Russell’s and Church’s 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 extension and discovery 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 mathematical proofs or generally 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 a 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 unit type, false is 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.

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 the means for defining families of types, for example, a family of lists with

---

<sup>1</sup>It has evolved into System FC recently

explicit length called *Vector*, for example  $\text{Vec } \mathbb{N} \ 3$  stands for a three element list of  $\mathbb{N}$ . Since there is more information in the type, 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 multiplication of matrix with dependent types.

The 1971 version of Martin-Löf type theory [62] was impredicative and turned out to be inconsistent due to Girard's paradox [51]. 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 [63] to describe the type of all types and usually denoted as  $\mathbf{U}$ . An impredicative universe  $\mathbf{U}$  has an axiom  $\mathbf{U} : \mathbf{U}$ . Starting from the 1972 version [65], a predicative hierarchy of universes is adopted and they are more widely used. Briefly speaking, it starts from 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 [73].

**Equality** is always one of the most contentious topics in Type Theory. In regular mathematics the notion of equality is usually used to describe sameness and taken as a given. But in Type Theory, we have different notions of equality or equivalence of the terms. First of all, **definitional equality** is a judgement-level equality, e.g.  $a \equiv b$ , which holds when two terms have the same normal forms [71]. It is also called **judgemental equality** [64]. Usually it already includes **computational equality** which is the congruence on terms generated from reduction rules like  $\beta$ -reduction and  $\eta$ -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 [63], 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  $\text{Id}_{\mathbb{N}}(a, b)$  or  $a =_{\mathbb{N}} b$  (see subsection 2.2.1).

In Intensional Type Theory (ITT or  $\text{TT}_I$  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  $\text{TT}_E$  for short), like the 1980 version [64] or NuPRL, propositional equality is undistinguished from definitional equality, in other words, two propositional equal objects are judgementally equal. This is achieved by the **equality reflection rule**:

$$\frac{a = b}{a \equiv b} \text{ ID-DEFEQ} \quad (2.1)$$

and the **uniqueness of identity proofs**:

$$\frac{p : a = b}{p \equiv \text{refl}} \text{ ID-UNI} \quad (2.2)$$

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, the 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 by programming languages such as Coq, Agda and Epigram, because its definitional equality is decidable, hence its type checking is decidable and programs written in it is terminating.

However in Intensional Type Theory, **extensional equality** is not available. For example equality of functions, equality of different proofs for the same proposition, and more extensional concepts like quotient types are desirable and reasonable to acquire. However simply axiomatising them can result in non-canonical objects e.g. a term of  $\mathbb{N}$  which are not reducible to numerals (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 1990s, there are some models of Type Theory 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 [91] interprets types as strict  $\omega$ -groupoids.

In recent years, Voevodsky proposes a new interpretation of intensional Martin-Löf type theory by homotopy-theoretical notions [53, 87] called Homotopy Type Theory (see Section 2.6). It is a univalent foundation 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 bases, we only treat them purely homotopically. The equality is internalised into types such that the types have infinite level of higher structures as weak  $\omega$ -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 by 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 [62] is impredicative and turned out to be inconsistent due to Girard’s paradox.
2. The 1972 version which is published in 1996 [65] 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 [63] introduced inductively defined identity type to internalising equality as a type.
4. The 1980 version which is summarised by Giovanni Sambin in 1984 [64] is extensional. It adopts equality reflection, namely an inhabitant of an identity type implies definitionally equality.
5. In the homotopic version [80], 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 written as derivation of these judgements. Usually we have following judgements in this thesis:

$$\begin{array}{ll}
 \Gamma \vdash & \Gamma \text{ is a well formed context} \\
 \Gamma \vdash A & A \text{ is a well formed type} \\
 \Gamma \vdash a : A & a \text{ is a well typed term of type } A \text{ in context } \Gamma \\
 \delta : \Gamma \Rightarrow \Delta & \delta \text{ is a substitution from context } \Gamma \text{ to } \Delta
 \end{array}$$

There are also equality judgements for contexts, types, terms and substitution. For instance,

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

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 usually use the following notational conventions:

- $\Gamma, \Delta$  for contexts
- $\gamma, \sigma$  for substitutions
- $A, B, C$  for types

- $a, b, c, t, x$  for terms
- $:\equiv$  for definitions
- **Set** or **Set**<sub>0</sub> for universe of small types, **Set**<sub>1</sub>, **Set**<sub>2</sub> ... for higher universes

### 2.2.1 Rules for types

The rules usually describe how we can define types with judgements above. They are syntactic rules but the semantic meaning may be revealed from the construction. Rules for types are usually classified as formation rule, introduction rule, elimination rule, computation rule ( $\beta$ ) and uniqueness rule ( $\eta$ ). Here we will only show the rules for the most important types. The substitution rules are not discussed here but a good reference is [46]).

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

$$\frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma, A \vdash} \quad (\text{COMPREHENSION})$$

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

**$\Pi$ -types** (dependent function type)

$$\begin{array}{c} \frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash \Pi (x : A) B} \quad (\Pi\text{-FORM}) \quad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \Pi (x : A) B} \quad (\Pi\text{-INTRO}) \\[10pt] \frac{\Gamma \vdash f : \Pi (x : A) B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]} \quad (\Pi\text{-ELIM}) \end{array}$$

In the expressions like  $\lambda(x : A).b$ ,  $\lambda$  binds the free occurrences of  $x$  in  $b$ . In the expressions like  $B[a/x]$  or  $b[a/x]$  we do an *standard substitution* in type  $B$  or term  $b$  that replaces free occurrences 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  $\Pi$ -types, for example  $(x : A) \rightarrow B$ , and their terms  $\lambda(x : A) \rightarrow b$ .

computation rule

uniqueness rule

$$(\lambda(x : A) \rightarrow b)(a) \equiv b[a]$$

$$f \equiv \lambda x \rightarrow f(x)$$

**$\Sigma$ -types** (dependent product type)

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash \Sigma A B} (\Sigma\text{-FORM}) \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : \Sigma A B} (\Sigma\text{-INTRO})$$

There are two ways to eliminate a term of  $\Sigma$ -types,

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

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

Identity types is a notion of intensional propositional equality given by the following rules:

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

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

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

Its computation rule is,

$$J(t, a, a, r(a)) \equiv t(a)$$

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

Let

$$C(x, y, p) \equiv B(x) \rightarrow B(y)$$

$$t(x) \equiv \text{id}$$

Thus,

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

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

**Unit type**

$$\begin{array}{c}
\frac{}{\vdash \top} \quad (\top\text{-FORM}) \qquad \frac{}{\vdash \mathbf{tt} : \top} \quad (\top\text{-INTRO}) \\
\\
\frac{\Gamma, x : \top \vdash A \quad \Gamma \vdash t : A[*]}{\vdash t : A} \quad (\top\text{-ELIM})
\end{array}$$

### Empty type

$$\frac{}{\vdash \perp} \quad (\perp\text{-FORM}) \qquad \frac{\Gamma \vdash A \quad e : \perp}{\Gamma \vdash \text{abort}(e) : A} \quad (\perp\text{-ELIM})$$

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

### Universe types

$$\frac{}{\Gamma \vdash \mathbf{U}} \quad (\mathbf{U}\text{-FORM}) \qquad \frac{\Gamma \vdash \hat{A} : \mathbf{U}}{\Gamma \vdash \mathbf{El}(\hat{A})} \quad (\mathbf{U}\text{-EL})$$

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

### Inductive types

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

The formation and introduction rules of a type is enough to build a type inductively. Examples like natural numbers  $\mathbb{N} : \mathbf{Set}$  can be defined as follows:

- $0 : \mathbb{N}$
- $\text{suc} : \mathbb{N} \rightarrow \mathbb{N}$

The terms are freely generated by a finite list of these constructors, for instance,  $\text{suc}(\text{suc } 0)$  stands for natural number 2. They are similar to data structures in programming languages, so most systems of Type Theory used as programming languages have inductive types along with structural recursion to eliminate or use them.

## 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 [92].

- *Functional programming language.* As the name indicates that, functional programming languages emphasise the application of functions rather than changing data in the 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.* It is a variant of Intensional Type Theory which is based on the Curry-Howard isomorphisms [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. [71] 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 clearly distinct. It not only helps us encode quantifiers but also allows writing very expressive types as program specifications such that programs become are 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  $\text{Matrix } m \ n \rightarrow \text{Matrix } n \ p \rightarrow \text{Matrix } m \ p$  where  $m, n, p : \mathbb{N}$ .

### 2.3.1 Features

Some features of being a functional programming language makes 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, by pattern matching on a term of identity type, the two sides on the equation automatically becomes the same in the context so that we can simply prove the symmetry of identity as,

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

compared to the case of using the eliminator J.

```
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
```

Function on inductive types are usually recursively defined using pattern matching. For example the addition of natural numbers are usually 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, it 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 [94].

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

- *Type checker.* Type checker is an essential part of Agda. It will detect type mismatch problems when some codes are 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 functions must terminate in Agda [72]. As a theorem prover, the type checker hence ensures that the proof is complete and not defined by itself.
- *Interactive interface.* It 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-terminating and yellow for not inferable. In the interactive Emacs, 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 documentations.
- *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. However in Agda, it provides unicode inputs and reads unicode symbols like  $\beta$ ,  $\forall$  and  $\exists$ .

It also uses a flexible mixfix notation where the positions of arguments are indicated by underscores. E.g.  $\_ \Rightarrow \_$  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,  $\mathbb{N}$  and  $\equiv$  are unicode characters,  $+$  and  $\equiv$  are mixfix operators.

$$\text{comm} : \forall (a\ b : \mathbb{N}) \rightarrow a + b \equiv b + a$$

Note that in Agda  $\equiv$  is used for identity types. See discussion in Section 2.3.2.

The unicode symbols and mixfix notations 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,

$$\begin{aligned} \text{id} &: \{A : \text{Set}\} \rightarrow A \rightarrow A \\ \text{id } a &= a \end{aligned}$$

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 which are typically infinite data structures. Coinductive occurrences must be labelled with  $\infty$  and coinductive types do not need to terminate but have to be productive. It is often used in conjunction with lazy evaluation. [93]

For coinductive types and more generally mixed inductive/coinductive types [36], we adopt a set of operators which are defined in module **Coinduction**. Infinite lists (or streams) can be defined as:

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

The delay operator  $\infty$  denotes a coinductive argument. Given  $a : A$ , the expression with a delay function  $\sharp a$  is an element of type  $\infty A$ .  $\flat x$  will force computation in  $x : \infty A$ .

- *Ring solver.* Compared to Coq, Agda has no tactics including automated proving 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 make confusions,

- The meaning of  $=$  is swapped with the one of  $\equiv$ . The symbol  $"="$  is reserved for function definition as the convention in programming languages. The congruence symbol  $"\equiv"$  is used for identity type. This is inconsistent with our conventional choice of symbols in articles.
- $:$  is used for typing judgement, for example  $a:A$ , while double colons  $::$  is the *cons* constructor for list. It is different from the usual notational conventions in Haskell.
- The universe of small types is **Set**<sub>0</sub> or **Set** instead of **Type**, even though it is not a set in set-theoretical sense.
- The universe of propositions **Prop** (**Prop**  $\subset$  **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  $\Pi$ -types.  $\Pi$ -types are written in a generalized arrow notation  $(x : A) \rightarrow B$  for  $\Pi x : A. B$ . Together with implicit arguments, it is valid to write a type signature as  $\forall \{A : \mathbf{Set}\} (x : A) \rightarrow \{y : A\} \rightarrow x \equiv y$ .
- $\Sigma$ -types are defined in Agda standard library. There is also a generalised  $\Sigma$ -types called *dependent record type* which can be defined by keyword **record**.
- In Agda, we use 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 which 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 same the *observable* behaviours, in other words they can be substituted by the other in any contexts without changing the meaning of a 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* has been summarised and comprehensively studied by Martin Hofmann [46] a list of them are given as follows:

- **Functional extensionality**

$$\frac{\Gamma \vdash A \quad \Gamma, x : A \vdash B \quad \Gamma \vdash f, g : (x : A) \rightarrow B(x) \quad \Gamma, a : A \vdash p : f(a) = g(a)}{\Gamma \vdash \text{ext}(p) : f = g} \text{ (FUN-EXT)}$$

If two (dependent) functions point-wise propositionally equal, they are (extensional) propositionally equal. This is called functional extensionality which is not inhabited in traditional formulation Intensional Type Theory [3]. For example, two functions of type  $\mathbb{N} \rightarrow \mathbb{N}$ ,  $\lambda n \rightarrow n$  and  $\lambda n \rightarrow n + 0$  are point-wise propositional equal, but the intensional propositional equality of them is not inhabited due to the fact that  $n + 0$  does not reduce to  $n$  (assume  $\_ + \_$  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  $\xi$ -rule, we know that  $\Gamma \vdash \lambda a. f a \equiv \lambda a. g a$ . From the  $\eta$ -rule of  $\Pi$ -types and the transitivity of  $\equiv$ , we know that  $\Gamma \vdash f \equiv g$ . Finally we can conclude that  $\Gamma \vdash \text{refl}(f) : f = g$ .  $\square$

In Intensional Type Theory, since propositional equality is not identified with definitional equality, it is not inhabited. If we postulate it, the  $\mathbb{N}$ -canonicity property (see Definition 2.1.9 in [46]) of Intensional Type Theory is lost, or we can say the theory is no longer *adequate* [3].

**Definition 2.3.** *A type theory is **adequate** if every term of  $\mathbb{N}$  is definitionally equal to a numeral, i.e. either 0 or in the form of  $\text{suc}(\dots)$ .*

**Theorem 2.4.** *If we introduce functional extensionality into Intensional Type Theory, the theory is not adequate.*

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

$$\text{id} := \lambda x \rightarrow x \text{ and } \text{id}' := \lambda x \rightarrow x + 0$$

where  $+$  is defined recursively as

$$0 + n := n \quad (\text{suc } m) + n := \text{suc } (m + n)$$

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

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

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

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

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  $\text{refl}$ , this closed term of natural number is not reduced to either 0 or in the form of  $\text{suc}(\dots)$ .

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)**

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

UIP is not a consequence of the fundamental eliminator for identity type  $\mathbf{J}$  as shown in Hofmann and Streicher's groupoid interpretation of Type Theory [49]. It holds if we add another eliminator  $\mathbf{K}$  introduced by Streicher in [76] as follows:

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

computation rule:

$$\mathbf{K}(t, \text{refl}(a)) \equiv t$$

In programming languages such as Agda and Epigram, UIP and  $\mathbf{K}$  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 [43]. A construction of the proof can be found in [34].

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

### • Proof irrelevance

In traditional Intensional Type Theory, there is no universe of propositions  $\mathbf{Prop}$  which has proof irrelevance:

$$\frac{\Gamma \vdash P : \mathbf{Prop} \quad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \quad (\text{PROOF-IRR})$$

We usually use  $\mathbf{Set}$  instead which does not automatically gives us a proof that  $(p, q : P) \rightarrow p = q$ .

An example of Intensional Type Theory extended with  $\mathbf{Prop}$  is the metatheory of Altenkirch’s setoid model (see Section 6.1).

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

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

It is different to a universe of propositions because not every set 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 type for sets as  $x = y : \mathbf{Prop}$  and UIP is provable.

- **Propositional extensionality**

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

Propositional equality between two propositions is given by logical equivalence. Similarly it only make senses if there is **Prop**.

- **Quotient types**

A quotient type is a type formed by redefining equality on a underlying type with 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  $(A = B) \rightarrow (A \simeq B)$  is an equivalence.

Equivalence can be thought of a refinement of isomorphism in higher categories. The notions of Homotopy Type Theory is discussed in Section 2.6.

### 2.4.1 Conservativity of $\mathbf{TT}_E$ over $\mathbf{TT}_I$ 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  $\eta$ -rule for  $\Pi$ -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 the type checking is undecidable

which has been formally proved by Hofmann in [46]. This makes Intensional Type Theory more favourable, so adding extensional principles into Intensional Type Theory is one of the most important topic in Type Theory. It is preferable if the decidability of type-checking and canonicity are not sacrificed. It is also valid and justifiable to extend  $\text{TT}_I$  with them.

The following theorem proved by Hofmann in [47] states that  $\text{TT}_E$  is conservative over  $\text{TT}_I$  with functional extensionality and uniqueness of identity added.  $\|_\_$  is an interpretation of  $\text{TT}_I$  into  $\text{TT}_E$  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  $\text{TT}_I$ , for example categories with families (see Definition 6.2) in the sense of Dybjer which is also a model of  $\text{TT}_E$  due to the mapping  $\|_\_$  discussed above. The interpretation of the term  $a$  in this model gives a term of type  $A$  by fullness in  $\text{TT}_I$ , hence  $a'$ . The detailed proof can be found in [47]. In the model  $\mathbf{Q}$ , types and contexts are propositionally equal if they are isomorphic, which becomes definitional equal in  $\text{TT}_E$ . The proof is also applied to quotient types which has been shown in [46]. However, the proof is non-constructive so that 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  $\eta$ -rules for  $\Pi$ -types and  $\Sigma$ -types [3]. It is used as a metatheory for his setoid model (see Chapter 6).

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

$$\frac{\Gamma \vdash P : \mathbf{Prop} \quad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \quad (\text{PROOF-IRR})$$

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

Notice that it is not a definition of types, which means that we cannot conclude a type is of type  $\mathbf{Prop}$  if we have a proof that all inhabitants of it are definitionally equal.

The propositional universe is closed under  $\Pi$ -types and  $\Sigma$ -types:

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

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

We may also use shorthand notation for  $\Pi$ -types, for example  $(x : A) \rightarrow P$  or  $\forall(x : A) \rightarrow P$ ,  $P \wedge Q$  if  $Q$  does not depend on  $P$ .

The metatheory is then proved to be:

- Decidable. The definitional equality is decidable, hence type checking is decidable.
- Consistent. Not all types inhabited and not all well typed definitional equality holds.
- Adequate. All terms of type  $\mathbb{N}$  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 Homotopic 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** for which 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 [86] develops 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  $f : A \rightarrow B$  is a continuous map between spaces  $A$  and  $B$ .

- Types are interpreted as spaces.  $a : A$  can be stated as  $a$  is a point of space  $A$ .
- Terms are continuous functions, for example,  $f : A \rightarrow B$  is a continuous function between spaces and it is equivalent to say that  $a$  is a point of the space or  $a : 1 \rightarrow A$  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  $p : [0, 1] \rightarrow X$ ).
- There are also 3-homotopies and 4-homotopies and higher levels which forms an infinite structure called  $\omega$ -groupoids in higher category theory.

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



## 2.6.2 Weak $\omega$ -groupoids

We can also interpret types as  **$\omega$ -groupoids**. An  $\omega$ -groupoid (or weak  $\infty$ -groupoids) is an  $\omega$ -category where all  $k$ -morphisms for all  $k \in \mathbb{N}$  are equivalences.  $k$ -morphisms between  $(k - 1)$ -morphisms, for all  $k \in \mathbb{N}$ . An equivalence is a morphism which is invertible up to all higher equivalences.

The notion of  $\omega$ -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.

The notion of equivalence can be seen as a refinement of isomorphism without UIP [7]. In the higher-categorical setting, equivalence can be thought of as arising from isomorphisms by systematically replacing equalities by higher cells. For example, an equivalence between two objects  $A$  and  $B$  in a 2-category is a morphism  $f : A \rightarrow B$  which has a corresponding inverse morphism  $g : B \rightarrow A$ , 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  $\omega$ -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 say just equivalence. In fact the  $\omega$ -groupoids used to model the identity types are also weak, so we should call them **weak  $\omega$ -groupoids**.

To distinguish these structured objects interpretation from usual set-like types, we also call them **homotopy types**. Although we interpret all types as  $\omega$ -groupoids, most types behave internally like  $k$ -groupoid for some  $k \in \mathbb{N}$ , which is called **truncation**. An  $n$ -truncated  $\omega$ -groupoid is an  $n$ -groupoid.

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 0-groupoid. Thus a set is called 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. Following this relation, a  $(-1)$ -type is a proposition (**mere proposition** or **h-proposition** in Homotopy Type Theory) and  $(-2)$ -type is a unit type or contractible type.

### 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 [85].

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

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

derived from induction on identity types. The univalence axiom just claims that this mapping is an equivalence.

It can be viewed as a strong extensionality 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 are automatically transported between them, and it actually makes reasoning 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 are not expressive enough.

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 `base` :  $\mathbb{S}^1$ , and
- A path `loop` : `base`  $=_{\mathbb{S}^1}$  `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. For example, to define a function  $f : \mathbb{S}^1 \rightarrow B$ , assume  $f(\text{base}) = b$ , we have to map `loop` to an identity path  $l : b = b$ , i.e.  $ap_f(\text{loop}) = l$ .

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 it 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 [80]. In this thesis, “*the HoTT book*” usually refers to it.

### 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  $\omega$ -groupoids, one main problems 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  $\omega$ -groupoids. Nevertheless there are some attempt of encoding weak  $\omega$ -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 [53] is based on Kan simplicial sets. There is a concise introduction written by Streicher [77]. However the simplicial set model is not constructive as Coquand showed that it requires classical logic in an essential way [31]. To avoid the use of classical logic, types can be interpreted as *semi-simplicial sets*. We have not successfully implemented the notion of semi-simplicial sets in an Intensional Type Theory like Agda. Some relevant discussion of it can be found online [90].

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 more plausible model for obtaining a computational interpretation of univalence.

## 2.7 Summary

The theory of types was originally invented to resolve an inconsistency in set theory in 1900s. After that, mathematicians developed it by adding more properties, for example functions as primitive types, Curry-Howard isomorphism. Type theory is closely related to type systems in programming languages, and some type theories like the simply-typed lambda calculus, Per Martin L  f’s intuitionistic type theory and the calculus of constructions, are themselves programming languages.

Martin-L  f 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 Curry-Howard isomorphism and dependent types, we can implement intuitionistic logic in Type Theory. It means that we can do constructive reasoning by program constructions. From a mathematician’s point of view, it provides computer-aided formal reasoning in languages like Agda and Epigram. From a programmer’s point of view, it provides program verification in itself and a more expressive way to write specifications for programs.

The intensional version of Martin-L  f type theory has decidable type checking which is a popular choice in programming languages. Agda is one of these languages and it has many good features supporting mathematical constructions and reasoning. It is used a lot 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 many extensional concepts like functional extensionality and quotient types. Much research has been done to add them into Type Theory without losing the computational property of Type Theory. 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  $\omega$ -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 type 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 [52]. Quotient types in our definition are essentially quotient *sets*, if quotients are not interpreted as sets, for example in Homotopy Type Theory. We present 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

The quotient types are defined by the following rules as described in [45, 52].

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

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

The equivalence properties include

$$\begin{array}{c}
\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{ref}_\sim(a) : a \sim a} \quad (\sim\text{-refl}) \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash p : a \sim b}{\Gamma \vdash \text{sym}_\sim(p) : b \sim a} \quad (\sim\text{-sym}) \\
\\
\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash c : A \quad \Gamma \vdash p : a \sim b \quad \Gamma \vdash q : b \sim c}{\Gamma \vdash \text{trn}_\sim(p, q) : a \sim c} \quad (\sim\text{-trans})
\end{array}$$

*Remark 3.1.* Notice that the formation rule is different to Hofmann’s version [45] where  $\sim$  is not required to be only equivalence relation. In fact his version is just more general which accepts non equivalence relation  $R : A \rightarrow A \rightarrow \mathbf{Prop}$ , but  $A/R$  has to be understood as the quotient of  $A$  by the equivalence closure of  $R$ .

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

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  $[a]$  can be confused with the substitution notation like  $B[a]$  or  $B[[a]/x]$ . So for a  $\Pi$ -type  $B : (x : A) \rightarrow \mathbf{Set}$  and  $a : A$ , we write  $B(a) : \mathbf{Set}$  for  $B[[a]/x]$  where order of the arguments in brackets matches its definition.

In Hofmann’s [45] definition, it comes with an eliminator (also called *lifting*) with its computation rule ( $\beta$ -rule) and an induction principle (equivalent to a  $\eta$ -rule):

1

$$\frac{\Gamma \vdash B \quad \Gamma \vdash f : A \rightarrow B \quad \Gamma, a : A, b : A, p : a \sim b \vdash f^\sim(a, b, p) : f(a) =_B f(b) \quad \Gamma \vdash q : A/\sim}{\Gamma \vdash \hat{f}(q) : B} \quad (\text{Q-elim})$$

---

<sup>1</sup>We use shorthand notation  $\hat{\phantom{x}}$  for lifting here



$$\begin{array}{c}
\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{Qcomp}(a) : \hat{f}([a]) \equiv f(a)} \quad (\text{Q-comp}) \\
\\
\frac{\Gamma, x : A/\sim \vdash P : \mathbf{Prop} \quad \Gamma, a : A \vdash h(a) : P([a]) \quad \Gamma \vdash q : A/\sim}{\Gamma \vdash \text{Qind}(h, q) : P(q)} \quad (\text{Q-ind})
\end{array}$$

Given a function  $f : A \rightarrow B$  which respects  $\sim$ , we can lift it to be a function on  $A/\sim$  as  $\hat{f} : A/\sim \rightarrow B$  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  $P : A/\sim \rightarrow \mathbf{Prop}$ . 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.  $[a]$ .

Alternatively, a *dependent* eliminator (dependent lifting) serves the same purpose:

$$\begin{array}{c}
\frac{\Gamma, x : A/\sim \vdash B \quad \Gamma \vdash f : (a : A) \rightarrow B([a]) \quad \Gamma, a : A, b : A, p : a \sim b \vdash f^=(a, b, p) : f(a) \stackrel{p}{=} f(b) \quad \Gamma \vdash q : A/\sim}{\Gamma \vdash \hat{f}(q) : B(q)} \quad (\text{Q-dep-elim}) \\
\\
\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{Qdcomp}(a) : \hat{f}([a]) \equiv f(a)} \quad (\text{Q-dep-comp})
\end{array}$$

Notice that  $\stackrel{p}{=}$  is an abbreviation for propositional equality which requires substitution in the type of the left hand side by  $\text{Qax}(p)$  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  $(a : A) \rightarrow B([a])$  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 is of type  $A \rightarrow B'$  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  $P : A/\sim \rightarrow \mathbf{Prop}$  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 find out that the non-dependent eliminator and induction principle are just special cases of dependent eliminator.

A formalised version of this proof in Agda can be found in [Appendix A](#).  $\square$

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

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

In fact propositional extensionality implies that all quotients defined with *equivalence* relation are effective.

**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  $P_a : A \rightarrow \mathbf{Prop}$  as

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

$P_a$  is compatible with  $\sim$  since

$$x \sim y$$

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

$$\equiv P_a(x) \iff P_a(y) \text{ (propositional extensionality)}$$

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

Therefore we can lift  $P_a$ <sup>2</sup> 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

---

<sup>2</sup>The elimination rule applies to large types

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

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

$$a \sim b$$

.

□

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

**Theorem 3.4.** *If we postulate the rules of quotient types, the type theory is not adequate.*

*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  $B : A/\sim \rightarrow \mathbf{Set}$  as

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

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

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

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

## 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 category **Set**. Let us recall the definition.

**Definition 3.5. Coequalizer.** *Given two objects  $X$  and  $Y$  and two parallel morphisms  $f, g : X \rightarrow Y$ , a coequalizer is an object  $Q$  with a morphism  $q : Y \rightarrow Q$  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  $q' = u \circ q$ :*

$$\begin{array}{ccccc}
 X & \xrightarrow{\quad f \quad} & Y & \xrightarrow{\quad q \quad} & Q \\
 & \xrightarrow{\quad g \quad} & & & \vdots \\
 & & & & u \\
 & & & & \downarrow \\
 & & & & Q'
 \end{array}$$

$q'$

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

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

and the two projections are two parallel morphisms  $\pi_1, \pi_2 : R \rightarrow A$ ,

A quotient corresponds to the coequalizer  $(A/\sim, [\_])$ :

$$\begin{array}{ccccc}
 R & \xrightarrow[\pi_2]{\pi_1} & A & \xrightarrow{[\_]} & A/\sim \\
 & & \searrow f & & \downarrow \hat{f} \\
 & & & & B
 \end{array}$$

The factorisation  $\hat{\_}$  is just the eliminator, the computation rule and induction principle corresponds 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  $g : A/\sim \rightarrow B$  fulfils the same property as  $\hat{f}$ , applying induction principle on

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

we can deduce that

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

hence  $g = \hat{f}$ .

The other way is more difficult:

Given  $P : A \rightarrow \mathbf{Prop}$ ,  $h : (x : A) \rightarrow P(x)$  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 universal property, there is a unique  $\hat{h}'$  s.t.

$$\hat{h}' \circ [\_] = h' \quad (3.3)$$

By replacing 3.3 in 3.2

$$\pi_1 \circ \hat{h}' \circ [\_] = [\_] \quad (3.4)$$

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 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, a dependent eliminator is also derivable.  $\square$

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

$$\begin{array}{ccc} R & \xrightarrow{\pi_1} & A \\ \pi_2 \downarrow & \lrcorner & \downarrow [\_] \\ A & \xrightarrow{[\_]} & Q \end{array}$$

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

From the pullback property, there is a unique point  $r : \mathbf{1} \rightarrow R$  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.

Categorically speaking, a *prequotient* is just a *fork*

$$R \begin{array}{c} \xrightarrow{\pi_0} \\ \xrightarrow{\pi_1} \end{array} A \xrightarrow{[\_]} Q$$

and a *definable quotient* corresponds to a *split coequalizer* which is a *fork* with two morphisms  $\text{emb} : Q \rightarrow A$  and  $t : A \rightarrow R$  such that

- $[\_] \circ \text{emb} = 1_Q$
- $\text{emb} \circ [\_] = \pi_2 \circ t$  and
- $\pi_1 \circ t = 1_A$

In fact,  $t$  corresponds to the "complete" property of definable quotients as  $t(a) = (\text{emb}[a], a)$ .

### 3.3 Quotients as an adjunction

As Jacobs [52] suggests, quotient can be described as a left adjoint to an equality functor.

Let us recall the definition first.



**Definition 3.7. Adjunction.** *Given two categories  $A$   $B$ , a functor  $F : A \rightarrow B$  is left adjoint to  $G : B \rightarrow A$  if we have a natural isomorphism  $\Phi : \text{hom}_B(F \_, \_) \rightarrow \text{hom}_A(\_, G \_)$*

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

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

where the morphism part is trivial embedding.

Quotients can be seen as a functor  $\mathbf{Q} : \mathbf{Setoid} \rightarrow \mathbf{Set}$  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

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

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

$$\frac{Y/\sim \rightarrow X}{(Y, \sim) \rightarrow (X, =_X)}$$

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

Given an identity morphism  $\text{id} : A/\sim \rightarrow A/\sim$ ,

$$\Phi(\text{id}) : (A, \sim) \rightarrow (A/\sim, =_{A/\sim})$$

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

Given a morphism  $f : (A, \sim) \rightarrow (B, =_B)$  which is a function compatible with  $\sim$ ,

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

which corresponds to the elimination rule.

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

$$\begin{array}{ccc} (A, \sim) & & \\ \eta_{(A, \sim)} \downarrow & \searrow f & \\ (A/\sim, =_{A/\sim}) & \xrightarrow{\nabla(\Phi^{-1}(f))} & (B, =_B) \end{array}$$

which is commutative because

$$\begin{aligned} & \nabla(\Phi^{-1}(f)) \circ \eta_{(A, \sim)} \\ &= \Phi(\Phi^{-1}(f)) \text{ by adjunction law } G(f) \circ \eta_Y = \Phi(f) \\ &= f \end{aligned}$$

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

For simplicity, we use "sets" for h-sets and "propositions" for h-propositions here. Also **Prop** is *not* the built-in universe of propositions in Coq, but the universe of h-propositions.

#### 3.4.1 An impredicative encoding of quotient sets

Vladimir Voevodsky has introduced an impredicative definition of quotients which has been encoded in Coq [84].

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

**Definition 3.8.** *An **equivalence class** is a predicate  $P : A \rightarrow \mathbf{Prop}$  such that it is inhabited  $\exists(a : A) P(a)$*

*For all  $x, y : A$ ,*

$$P(x) \rightarrow P(y) \rightarrow x \sim y$$

$$P(x) \rightarrow x \sim y \rightarrow P(y)$$

*The properties can be encoded as*

$$EqClass(P) \equiv (\exists(a : A) P(a)) \wedge (\forall(x, y : A) \rightarrow P(x) \rightarrow (x \sim y \iff P(y)))$$

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

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

$A/\sim$  is a set because  $A \rightarrow \mathbf{Prop}$  is a set and  $\text{EqClass}(P)$  is a proposition.  $\wedge$  is the non-dependent  $\Sigma$ -type for propositions and  $\forall$  is the  $\Pi$ -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 of  $P$  is inhabited, and  $q$  contains the proofs of the logical equivalence.

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

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

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

$$|x| \equiv \lambda P f \rightarrow f(x)$$

We can simply recover the elimination rule for truncation: given any function  $f : X \rightarrow P$  where  $P$  is a proposition, we can define a function of type  $\|X\| \rightarrow P$  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 [88, 89],  $\|X\|$  is moved to the universe of  $\mathbf{Set}$ . We can apply the resizing rule for proposition because  $\|X\|$  behaves like a proposition. It also has to be noticed that it is impossible to extract an element of  $A$  from an proof of  $\text{EqClass}(P)$  because of the impredicative encoding.

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

$$[a] := (\lambda x \rightarrow a \sim x, |a, \text{ref}(a)|, \lambda x \ y \ p \rightarrow (\lambda q \rightarrow \text{trn}(p, q), \lambda q \rightarrow \text{trn}(\text{sym}(p), q)))$$

which is compatible with  $\sim$ . 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 an unique representation of an equivalence class.

**Lemma 3.11.** *Given  $(P, p, q) : A/\sim$  and  $a : A$  such that  $ex : P(a)$  holds,  $[a] = A/\sim(P, p, q)$ , namely*

$$\Sigma(a : A) \ P(a) \rightarrow [a] = A/\sim(P, p, q)$$

*is inhabited.*

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

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

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

By functional extensionality, we only need to prove that

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

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

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

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

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

□

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

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

*Proof.* Assume we have an element  $(P, p, q) : A/\sim$ ,

we can define a function  $f_P : (\Sigma(x : A) P(x)) \rightarrow B$  simply by

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

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

$$\bar{f}_P : \|(\Sigma(x : A) P(x))\| \rightarrow B$$

from 3.13 if  $f_P$  is a constant function:

For any two elements of  $\Sigma(x : A) P(x)$ ,  $(x_1, p_1)$  and  $(x_2, p_2)$ , by applying one of the property contained in  $q$

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

to  $p_1 : P(x_1)$  and  $p_2 : P(x_2)$ , we prove that

$$x_1 \sim x_2$$

Then because  $f$  is compatible with  $\sim$ ,

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

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:

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

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

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

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

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

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

from 3.11. Thus we can substitute in  $h(a) : Q([a])$  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.  $\square$

The following lemma is suggested by Nicolai Kraus and can be found in [56].

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

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

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

*Proof.* Define the subset

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

Intuitively it contains only 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

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

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

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

From the fact that 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  $g' : X \rightarrow Y'$  from  $g$  as

$$g'(x) := (g(x), \lambda Q \ f \rightarrow f(x, \text{refl}_=(g(x))))$$



Because  $Y'$  is propositional, it is possible to lift  $g'$  to a function  $\tilde{g}' : \|X\| \rightarrow Y'$  which is technically 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, we can prove that the impredicative quotients are effective from Theorem 3.3.

**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 by using higher inductive types.

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

- $[\_] : A \rightarrow A/\sim$

- $eqv : (a, b : A) \rightarrow a \sim b \rightarrow [a] = [b]$
- $isSet : (x, y : A/\sim) \rightarrow (p_1, p_2 : x = y) \rightarrow p_1 = p_2$

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

*Remark 3.15.* Categorically, there is no difference between a set and a setoid because a skeleton of a setoid is just a discrete category which is equivalent to a set.

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

One example is the definition of real numbers  $\mathbb{R}$  which will be discussed in Chapter 5. 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 [80]) 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 tree) which are trees connected to a multiset of subtrees, hence there is no ordering on subtrees.

Firstly we define ordered trees as:

- A leaf  $l : \text{Tree}$ , or
- An ordered list of subtrees indexed by  $I$ ,  $st : (I \rightarrow \text{Tree}) \rightarrow \text{Tree}$ ,

With the following equivalence relation:

- $l_{eq} : l \sim l$ ,
- $st_{eq} : (f, g : I \rightarrow \text{Tree}) \rightarrow f \sim_p g \rightarrow sp(f) \sim sp(g)$

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

If we define unordered trees as a quotient type  $\text{Tree}^\sim := \text{Tree}/\sim$ , 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  $\text{BTree} \rightarrow \text{BTree} \rightarrow \text{BTree}$ . 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 constructors for types so that:

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

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

Similarly the cumulative hierarchy of all sets introduced in [80] (see section 10.5) also suggests that quotient types have some weaknesses compared to quotient inductive types.

A cumulative hierarchy can be given by constructors,

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

along with a subset relation,

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

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

Then we can easily define the equivalence relation on "sets" using set-theoretical definition

$$A \sim B \equiv \forall m : M_0, m \in A \iff m \in B$$

Similar to unordered trees, we can not obtain the constructor  $\widehat{\{\_ \}}$  because the index 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 [80]). It is interesting to investigate *real quotient types* in Homotopy Type Theory, but it is beyond 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 [28], Mendler et al. have first 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 the an equivalence relation  $E$ , the quotient is denoted as  $(x, y) : A//E$ . 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 using a simple example:

When we want to define a function  $f : (x, y) : A/E \rightarrow \mathbf{2}$ , 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 inconsistency since  $E(a, b)$  implies  $a$  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 is contradicted with the equality reflection rule.

Therefore a function is well-defined [28] on the new type only if it respects the equivalence relation  $E$ , namely

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

After the introduction of quotient types, Mendler further investigates this topic from a categorical perspective in [69]. 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 Nögin [70].

- Nögin [70] considers a modular approach to axiomatizing quotient types in NuPRL as well. He discusses some problems about quotient types. For example, since the equality is extensional, we cannot recover the witness of the equality. He suggests including more axioms to conceptualise quotients. He decomposes the formalisation of quotient type into several smaller primitives such that they can be handled much simpler.
- Jacobs [52] introduces a syntax for quotient types based on a predicate logic within simple type theory. He discusses quotient types from a categorical theoretical 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 [46]. 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 groupoid model which supports most features required but it is not definable in Intensional Type Theory. He also proposes a third model to 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 [47].

- Altenkirch [3] also provides a different setoid model which is built in an Intensional Type Theory extended with proof-irrelevant universes of propositions and  $\eta$ -rules for  $\Pi$ -types and  $\Sigma$ -types. It is decidable, adequate and permits large eliminations. We have implemented this setoid model and interpret quotient types in it (see Chapter 6).
- Homeier [50] also 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 such that users are able to define new types in HOL. Next he defines the normalisation functions and proves several properties of these. Finally he discussed the issues when quotienting on the aggregate types such as lists and pairs.
- Courtieu [33] shows an extension of 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] also propose a new notion called *congruence types*, which is also a special class of quotient types, in which the base type are

inductively defined and with a set of reduction rules called the term-rewriting system. The idea behind it is the  $\beta$ -equivalence is replaced by a set of  $\beta$ -conversion rules. Congruence types can be treated as an alternative to the pattern matching introduced in [30]. 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 term-rewriting system to link a term of base type to a unique term of congruence type as its normal form. However this approach has some problems in termination criteria and interaction between rewriting systems [33].

- Barthe and Capretta [16] compare different ways to setoids in Type Theory. The setoid is classified as partial setoid or total setoid depending on whether the equality relation is reflexive or not. They also consider obtaining quotients with different kinds of setoids, especially the ones from 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 *defined* elements, the others are undefined. Therefore if we simply define a quotient by replacing 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 [84] implements quotients in Coq based on a set of axioms of Homotopy Type Theory. He firstly implement equivalence class and use it to implement quotients which is an analogy to the construction of quotient sets in set theory. The details has been given in Section 3.4.1.

## 3.6 Summary

We have given the syntax of quotient types in this chapter. The relation is required to be an equivalence in our definition which is different to [45]. In fact, the equivalence condition does not affect the construction of quotient types. Jacobs [52] has shown that, for arbitrary relation  $R$ , the same construction can be interpreted as set theoretical quotient sets of  $A/R^\equiv$ , where  $R^\equiv$  is the equivalence closure of  $R$ . However the equivalent condition does allow us to prove that

Two approaches of defining elimination rules has been given, one is a combination of non-dependent eliminator with an induction principle as in Hofmann's definition and another is a dependent eliminator. We have also shown they are equivalent.

We have also shown that propositional extensionality actually implies the effectiveness of quotients. The characterisation of quotients in category theory has been given. It does not only corresponds to coequalizers but also can be generated from a left adjoint functor to the equality functor  $\nabla : \mathbf{Set} \rightarrow \mathbf{Setoid}$ . We have also briefly reviewed the research of quotient types by other authors.



## 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  $[\_] : \mathbb{Z}_0 \rightarrow \mathbb{Z}$ .

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 only be reduced to a unique one called a *reduced fraction*. Therefore, the set  $\mathbb{Q}$  can also be defined as  $\Sigma$ -type consists of a fraction together with a 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 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 fractions 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, it requires us to redefine all operations on sets again on setoids, for example  $\text{List}_{\text{setoid}}(A, \sim)$ . 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 such 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 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 corresponds 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  $[\_] : A \rightarrow Q$ ,

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

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

Roughly speaking, 1 corresponds to the formation rule, 2 corresponds to the introduction rule and 3 corresponds to  $Q\text{-}\mathbf{Ax}$ . 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  $B : Q \rightarrow \mathbf{Set}$ , an eliminator

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

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

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

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

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

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

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

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

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  $(Q, [\_], \text{sound})$  with

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

It is exactly the specification of  $[\_]$  as a normalisation function with respect to  $\text{emb}$  (see [5]). It is also related to the choice operator for quotient types in Martin Hofmann's definition[45].

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

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

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

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

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

By completeness we get

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

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

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

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

By stability, 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

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

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

However, from the following equation

$$n_1 - n_2 = n_3 - n_4$$

it is easy to find 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 to transform the equation as

$$n_1 + n_4 = n_3 + n_2$$

This gives rise to an equivalence relation:

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

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

Thereby the setoid of integers can be formed as:

```

 $\mathbb{Z}$ -Setoid : Setoid
 $\mathbb{Z}$ -Setoid = record
  { Carrier    =  $\mathbb{Z}_0$ 
  ;  $\_ \approx \_$     =  $\_ \sim \_$ 
  ; isEquivalence =  $\_ \sim \_$ 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:

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

If we define  $\mathbb{Z}$  in this way, 0 have 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:

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

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

```

data  $\mathbb{Z}$  : Set where
  +\_ :  $\mathbb{N} \rightarrow \mathbb{Z}$ 
  -suc\_ :  $\mathbb{N} \rightarrow \mathbb{Z}$ 

```

Although it is 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:

$$\begin{aligned} [\_] & : \mathbb{Z}_0 \rightarrow \mathbb{Z} \\ [m, 0] & = +m \\ [0, \text{suc } n] & = -\text{suc } n \\ [\text{suc } m, \text{suc } n] & = [m, n] \end{aligned}$$

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

It is plausible to define the embedding function written as  $\ulcorner \_ \urcorner$ . In fact the first two cases in definition of  $[\_]$  already gives us the answer:

$$\begin{aligned} \ulcorner \_ \urcorner & : \mathbb{Z} \rightarrow \mathbb{Z}_0 \\ \ulcorner +n \urcorner & = n, 0 \\ \ulcorner -\text{suc } n \urcorner & = 0, \text{suc } n \end{aligned}$$

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

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

Apply the lemma  $\text{sound}'$ , we get



$$[a] = [b]$$

hence  $[\_]$  is *sound* (compatible with  $\sim$ ).

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 simplest,

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

the sign of rational number is contained in numerator and it is easy to exclude 0 by viewing  $n$  as a denominator  $n + 1$ , therefore we encode it as follows:

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

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

Different fractions can represent the same rational numbers.

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

However 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:

$$\begin{aligned} \_ \sim \_ &: \mathbb{Q}_0 \rightarrow \mathbb{Q}_0 \rightarrow \text{Set} \\ n1 \text{ /suc } d1 \sim n2 \text{ /suc } d2 &= n1 \mathbb{Z}^* (+ \text{ suc } d2) \equiv n2 \mathbb{Z}^* (+ \text{ suc } d1) \end{aligned}$$

### 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. It is equivalent to say that their absolute values are coprime, thus we can define a predicate of  $\mathbb{Q}_0$  as

$$\begin{aligned} \text{IsReduced} &: \mathbb{Q}_0 \rightarrow \text{Set} \\ \text{IsReduced } (n \text{ /suc } d) &= \text{True } (\text{coprime?} \mid n \mid (\text{suc } d)) \end{aligned}$$

which decides whether they are coprime or not, if it is the case, it becomes  $\top$  otherwise it becomes  $\perp$ . Therefore it is a propositional set because 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:

```

 $\mathbb{Q} : \text{Set}$ 
 $\mathbb{Q} = \Sigma[ q : \mathbb{Q}_0 ] \text{IsReduced } q$ 

```

It 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 in unreduced forms. In fact someone complained about this problem<sup>1</sup> 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 proving 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 calculate 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 :  $\forall (x\ y : \mathbb{N}) \rightarrow y \neq 0 \rightarrow \mathbb{Q}$ 

```

---

<sup>1</sup>Discussion on Agda mailing list <http://comments.gmane.org/gmane.comp.lang.agda/6372>

```

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)

```

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  $\Sigma$ -types.

```

⌈ _ ⌋ : ℚ → ℚ₀
⌈ _ ⌋ = proj₁

```

To complete the definable quotient, we have to prove all essential properties 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 calculated and it becomes  $q_1 * r_2 = r_1 * q_2$  where  $(q_1, q_2)$  and  $(r_1, r_2)$  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  $\frac{x}{y}$ , whether the reduced one is equivalent to it. This is also easy to verify because in the reducing 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 coprime property which also reduce the complexity. Thus we can define operators and prove properties on setoid representation. Then we can easily lift them by two ways of conversions.

**Operators** For a unary operator  $f$ , we can lift them by

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

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

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

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 definition for operators on of  $\mathbb{Z}_0$  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 is compatible with  $\sim$ .

$$\begin{aligned} \_ + \_ &: \mathbb{Z}_0 \rightarrow \mathbb{Z}_0 \rightarrow \mathbb{Z}_0 \\ (\text{x+} \_, \text{x-}) + (\text{y+} \_, \text{y-}) &= (\text{x+} \text{N+} \text{y+}) \_, (\text{x-} \text{N+} \text{y-}) \end{aligned}$$

It has only one case which means that we usually do not need to do case analysis when proving properties about additions. In fact, it is also the case 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  $f : A \rightarrow A$  such that  $\forall(a :$

$A) \rightarrow f(f(a)) \sim a$ , we can prove that  $\forall(q : Q) \rightarrow \text{liftop1}(f)(\text{liftop1}(f)(q)) = q$  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 assumption  $\forall(a : A) \rightarrow f(f(a)) \sim a$  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  $f$  is compatible with  $\sim$ ,

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

By transitivity of  $\sim$

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

Because  $[\_]$  is compatible with  $\sim$ ,

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

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 such that properties can be proved with less case distinctions. Another advantage is that usually there are functions and properties available for the setoid form which are very useful.

In [57], 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 extremely 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 standard library which calculates signs and absolute values separately:

$$\begin{aligned} \_Z^* \_ &: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \\ i \_Z^* j &= \text{sign } i \ S^* \ \text{sign } j \triangleleft |i| \ N^* \ |j| \end{aligned}$$

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

When all of them are non-negative integers, we can apply the left distributivity law of natural numbers which are available. In fact it can be applied to all the cases that  $y$  and  $z$  have the same sign, because signs can be moved out.

Thus we can write some parts of the proof (Note: `DistributesOverl` means that the distributivity of the first operators 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. In experience, it is very complicated and inefficient because we can not use meaningful proved theorems to build new proofs.

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

In fact with the help of *ring solver*, it can be proved automatically. A *ring solver* is an automatic equation checker for rings e.g. the ring of integers. It is implemented based on the theory described in [41].

```

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 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 quote the type of the current goal so that the application of ring solver can be automated. There is already some work done by van der Walt [83]. 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 ring solvers. However ring solver has to calculate the proof which takes very long time in type checking from our experience. As these basic laws are used a lot in complicated theorems, pragmatically speaking, it is better not to prove them using 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 is 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  $\mathbb{Z}$  from scratch like in the proving 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-lem](#)<sup>*l*</sup> a b d c f e))

It only needs 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 quotient structure in the integer case provides us a general approach to define functions and prove theorems when the base types are simpler to deal with. To form the field of rational numbers, we can also benefit from the setoid representation. There is no extra properties to prove. After embedding the natural numbers into integers, any theorems using only the operations in the field  $\mathbb{Q}$ , are turned out to be equations of integers which can be automatically solved by ring solver.

## 4.5 Related work

Courtieu [33] considers an extension of calculus of inductive constructions (CIC) which is an intensional type theory by *normalized types* which can be seen as a type former 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 [27] also defines a quotient structure in Coq, which consists of  $Q$  as a quotient type,  $T$  as base type, two mapping  $\text{pi} : T \rightarrow Q$  and  $\text{repr} : Q \rightarrow T$  and a proof that  $\text{pi}$  is a left inverse of  $\text{repr}$ . It is similar to our algebraic structure of definable quotients without an equivalence relation involved,  $\text{pi}$  corresponds to  $[_]$ ,  $\text{repr}$  corresponds to  $\text{emb}$ , and the equivalence relation can be recovered simply: if for any two  $s, t : T$  such that  $\text{pi}(s) = \text{pi}(t)$ , then they are equivalent.

## 4.6 Summary

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

We introduce 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 has a dependent eliminator and the other given by Hofmann added a non-dependent eliminator and an induction principle. A *definable quotient* includes an embedding function selecting canonical choice for each equivalence class such that  $[\_]$  is correctly specified as a normalisation function. It is very useful in practice. It provides us flexible conversions between setoid representations and set representations. We can usually benefit from the convenience of 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 illustrate the advantages of definable quotients in the comparison between  $Z$  and  $Z_0$  in proving 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], finite multisets represented by lists etc. Intuitively speaking there is no definable normalisation function which returns a canonical choice for each equivalence class. For the Cauchy sequences of rational numbers, Nicolai Kraus [55] has shown that all definable endofunctions respect 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 [81].

### 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 inductively defined so that we can talk about normalisation function as  $[\_] : A \rightarrow Q$ . 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:*

- $[\_]_0 : A \rightarrow A$
- $\text{sound} : \forall(a, b : A) \rightarrow a \sim b \rightarrow [a]_0 = [b]_0$
- $\text{complete} : \forall(a : A) \rightarrow [a]_0 \sim a$

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

- The quotient set can be defined as

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

- The "normalisation function" is

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

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

- The embedding function is just first projection

$$\text{emb} \equiv \pi_1$$

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

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

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

We can prove it by J

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

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



- Completeness: given  $a : A$ , we need to prove  $\text{emb}[a] \sim a$  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,

•

$$[\_]_0 :\equiv \text{emb} \circ [\_]$$

- Soundness: given  $a, b : A$  such that  $a \sim b$ , because  $[\_]_0$  is sound, we know

$$[a]_0 = [b]_0$$

By congruence rule,

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

hence  $[\_]_0$  is sound as well.

- Completeness: given  $a : A$ ,  $\text{emb}[a] \sim a$  is just the completeness property of the definable quotient.

## 5.2 Real numbers as Cauchy sequences

The real numbers can be defined as  $(\mathbb{R}_0, \sim)$ , where  $\mathbb{R}_0$  is the set of Cauchy sequences, and two Cauchy sequences are equivalent if and only if their point-wise difference converges to 0.

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

$$\text{isCauchy}(f) := \forall(\varepsilon : \mathbb{Q}^+) \rightarrow \exists(m : \mathbb{N}) \forall(i : \mathbb{N}) \rightarrow i > m \rightarrow |f_i - f_m| < \varepsilon \quad (5.1)$$

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

$$\mathbb{R}_0 := \Sigma(f : \mathbb{N} \rightarrow \mathbb{Q}) \text{ isCauchy}(f)$$

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

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

To implement this definition, the existential quantifier is usually encoded by  $\Sigma$ -types so that we can guess the real number from the explicit witness  $m$ . However, usually we would like to keep the proof propositional so that the Cauchy sequences are proof-irrelevant.

To avoid this problem, we can use an alternative equivalent definition of the property, but the type of  $f$  has to be  $\mathbb{N}^+ \rightarrow \mathbb{Q}$ :

$$\text{isCauchy}(f) := \forall(k : \mathbb{N}^+), \forall(m, n > k) \rightarrow |f_m - f_n| < \frac{1}{k} \quad (5.2)$$

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.

We can slightly modify the definition which is still equivalent,

$$\text{isCauchy}(f) := \forall(n, m : \mathbb{N}^+), n > m \rightarrow |f_n - f_m| < \frac{1}{n} \quad (5.3)$$

### 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  $[\_]_0 : \mathbb{R}_0 \rightarrow \mathbb{R}_0$  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 claim that two  $a, b : A$  are *separable*, if there exists a definable test  $P : A \rightarrow \mathbf{2}$  such that  $P(a) \neq P(b)$ . Then we claim a definable set  $A$  is *discrete* if  $a \neq b$  always imply  $a$  and  $b$  are separable. However it has some problems as Martín Escardó pointed out. He provides a counterexample that for two distinguishable terms i.e.  $a \neq b$  there can be no definable test [38]. We sketch the proof here:

*Proof.* In the proof, he uses  $\mathbb{N}_\infty := \mathbb{N} \rightarrow \mathbf{2}$  which is a decreasing sequence of  $\mathbf{2}$  called *generic convergent sequence*. Intuitively speaking, 11000... represents 2 and the sequence of 1, namely 1111... represents  $\infty$ , therefore, it can be seen as the disjoint union of natural numbers and infinity i.e.  $\mathbb{N} + \infty$ . For simplicity, we write  $s_k$  for the sequence whose first  $k$  digits are 1 and the rest digits are 0.

From continuity, we know that:

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

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

$s_k^0 := (s_k, \lambda r \rightarrow 0)$  and

$s_k^1 := (s_k, \lambda r \rightarrow 1)$ ,

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

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

To prove it, assume  $f(\infty_0) \neq f(\infty_1)$ . We can prove that for all  $k : \mathbb{N}$  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 show 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  $d : M \times M \rightarrow \mathbb{R}$  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.  $\mathbf{2}$ ,  $\mathbb{N}$ ,  $\mathbb{Q}$ , 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  $f_1, f_2 : \mathbb{N}^+ \rightarrow \mathbb{Q}$  can be defined as

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

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  $f : X \rightarrow Y$  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  $f : (\mathbb{N}^+ \rightarrow \mathbb{Q}) \rightarrow \mathbf{2}$ , it only inspects finite many terms of the input sequences to compute the result.

We define a generalised condition of **isCauchy**:

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

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

The usual condition  $\text{isCauchy}$  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

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

which respects  $\sim$ , is constant.

*Proof.* Assume we have a  $f$  which respects  $\sim$ .

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

By abuse of notation, we write  $\llbracket \mathbb{R}_0 \rrbracket$  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  $\llbracket \mathbb{R}_0 \rrbracket / \llbracket \sim \rrbracket$ . 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  $\bar{\cdot} : \llbracket \mathbb{R}_0 \rrbracket \rightarrow \mathbb{R}$ , thus given a Cauchy sequence  $r : \mathbb{R}_0$ , the real numbers it represents can be written as  $\overline{\llbracket r \rrbracket} \in \mathbb{R}$ .

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

$$\llbracket f \rrbracket(c_1) \neq \llbracket f \rrbracket(c_2)$$

Define

$$m_1 \equiv \sup\{\bar{d} \in \mathbb{R} \mid d \in \llbracket \mathbb{R}_0 \rrbracket, \bar{d} \leq \max(\bar{c}_1, \bar{c}_2), \llbracket f \rrbracket(d) = \llbracket 1_2 \rrbracket\} \quad (5.6)$$

$$m_2 \equiv \sup\{\bar{d} \in \mathbb{R} \mid d \in \llbracket \mathbb{R}_0 \rrbracket, \bar{d} \leq \max(\bar{c}_1, \bar{c}_2), \llbracket f \rrbracket(d) = \llbracket 0_2 \rrbracket\} \quad (5.7)$$

(note that one of these two necessarily has to be  $\bar{c}_1$  or  $\bar{c}_2$ , whichever is bigger).

Set  $m := \min(m_1, m_2)$ . 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  $x = \bar{e}$  (for some  $e$ ) with  $\llbracket f \rrbracket(e) \neq \llbracket f \rrbracket(t)$ .

Let  $c \in \llbracket \mathbb{R}_0 \rrbracket$  be a Cauchy sequence such that  $\bar{c} = m$ . We may assume that  $c$  satisfies the condition  $\llbracket \text{isCauchy}_5 \rrbracket$ .

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

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

then

$$h(\llbracket f \rrbracket(c), \llbracket f \rrbracket(c')) < \epsilon < 1$$

hence  $\llbracket f \rrbracket(c') = \llbracket f \rrbracket(c)$ .

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

Let  $x \in I$ , there is a Cauchy sequence  $t : \llbracket \mathbb{R}_0 \rrbracket$  such that  $\bar{t} = x$  and we may assume that  $t$  satisfies the condition  $\llbracket \text{isCauchy}_{5n_0} \rrbracket$ .

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

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

Observe that  $g$  is also a Cauchy sequence, i.e.  $\llbracket \text{isCauchy} \rrbracket(g)$ . 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

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

Calculate

$$|g(n) - g(m)| \quad (5.10)$$

$$= |c(n) - t(m - n_0)| \quad (5.11)$$

$$= |c(n) - \bar{c} + \bar{c} - \bar{t} + \bar{t} - t(m - n_0)| \quad (5.12)$$

$$\leq |c(n) - \bar{c}| + |\bar{c} - \bar{t}| + |\bar{t} - t(m - n_0)| \quad (5.13)$$

$$\leq \frac{1}{5n} + \frac{1}{2n_0} + \frac{1}{5n_0 \cdot (m - n_0)} \quad (5.14)$$

$$\leq \frac{1}{5n} + \frac{1}{2n} + \frac{1}{5n_0} \quad (5.15)$$

$$< \frac{1}{n} \quad (5.16)$$

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

By the definition of  $g$ , it converges to the same real number as  $t$ , i.e.  $\bar{g} = \bar{t}$ . It is equivalent to say  $g \llbracket \sim \rrbracket t$  and by the condition  $\llbracket f \rrbracket$  respects  $\llbracket \sim \rrbracket$ , we can prove that  $\llbracket f \rrbracket(t) = \llbracket f \rrbracket(g) = \llbracket f \rrbracket(c)$  and therefore  $x = \bar{t} \in \bar{U}$ . Now we can conclude that  $I \subset \bar{U}$  which is equivalent to say  $\bar{U}$  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  $x = \bar{e}$  (for some  $e$ ) with  $\llbracket f \rrbracket(e) \neq \llbracket f \rrbracket(c)$ .  $\square$

This approach also is 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 endofunction  $f$  on  $\mathbb{R}_0$  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  $\llbracket f \rrbracket : \llbracket \mathbb{R}_0 \rrbracket \rightarrow \llbracket \mathbb{R}_0 \rrbracket$ , omitting the proof part of  $f$ .

Given a positive natural number  $n : \llbracket \mathbb{N}^+ \rrbracket$ ,  $\pi_n : \llbracket \mathbb{R}_0 \rrbracket \rightarrow \llbracket \mathbb{Q} \rrbracket$  is the projection function. Define a function  $h_n : \llbracket \mathbb{R}_0 \rrbracket \rightarrow \llbracket \mathbb{Q} \rrbracket$  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  $\Pi$ ,  $\Sigma$ ,  $\mathbf{W}$ ,  $=$ ,  $\mathbb{N}$ , if  $T$  does have two distinguishable elements, then it is not connected. Because  $\mathbb{R}_0/\sim$  is connected, this conjecture implies that the 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 [59]. 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  $\pi$ . It has been proved by Robert S. Lubarsky in [59]. If we add the axiom of Countable Choice ( $AC_\omega$ ) 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  $AC_\omega$  is a classical result which is stronger than the premise "in classical logic".

In the HoTT book [80] (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  $s : \mathbb{Q}^+ \rightarrow \mathbb{R}_C$  we have  $\lim(s) : \mathbb{R}_C$  as a limit of Cauchy sequence of real numbers, hence it is Cauchy complete. Higher inductive type allows 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

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

Here the condition  $a \neq b$  is not required, so the relation is can be reflexive.

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  $\leq : A \rightarrow A \rightarrow \mathbf{Prop}$  equipped with

$$\min, \max : A \rightarrow A \rightarrow A$$

calculating the binary minimum and maximum for that order. This allows us to define  $[\_]_0 : A \times A \rightarrow A \times A$  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  $(\text{List } A, \sim)$  where two lists are (bag) equivalent [35] 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 –  $\text{Vec}$  here.

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

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

where  $\text{Fin} : \mathbb{N} \rightarrow \mathbf{Set}$  represents finite sets and  $\text{Bijection} : (\text{Fin } n \rightarrow \text{Fin } n) \rightarrow \mathbf{Prop}$  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  $\text{sort} : \text{Vec } A \ n \rightarrow \text{Vec } A \ n$ . 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  $A_\perp$  (or  $(\text{Delay } A)$  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 : ∞ ((b x) ~ (b 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 : ∞ ((b x) ≈ (b y))) → (later x) ≈ (later y)
```

where  $x \downarrow y$  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  $a = \text{later} (\text{later} (\text{now } x))$ , then by abuse of notations,  $a_1 = \text{later}$ ,  $a_2 = \text{later}$ , and  $a_3 = \text{now } x$  (the rest  $a_n$  for  $n > 3$  can be filled by **later**). Then a standard metric space for  $A_\perp$  can be given by

$$g(a, b) = 2^{-\inf\{k \in \mathbb{N} \mid a(k) \neq b(k)\}} \quad (5.17)$$

Similar to the proof in Proposition 5.4, we can prove  $A_\perp / \approx$  is connected, i.e. any definable (continuous) function  $f : A_\perp \rightarrow \mathbf{2}$  which respects  $\approx$  is constant.

We assume  $\llbracket f \rrbracket$  is non-constant, i.e. there are  $x, y : \llbracket A_\perp \rrbracket$  such that  $\llbracket f \rrbracket(x) \neq \llbracket f \rrbracket(y)$ .

We can also assume  $\llbracket f \rrbracket(\llbracket \text{never} \rrbracket) = 1$ , because  $\llbracket f \rrbracket$  is continuous, there exists  $n_0 \in \mathbb{N}$  such that for all  $a \in \llbracket A_\perp \rrbracket$ , if the first  $n_0$  "elements" of  $a$  are **laters** (namely they coincide with those of **never**), then  $\llbracket f \rrbracket(a) = \llbracket f \rrbracket(\llbracket \text{never} \rrbracket) = 1$ .

Since  $\llbracket f \rrbracket(x) \neq \llbracket f \rrbracket(y)$ , one of them must have  $k < n_0$  **laters** before **now**, assume it is  $x$  then  $\llbracket f \rrbracket(x) = 0$  and  $\llbracket f \rrbracket(y) = 1$ . By adding  $n_0 - k$  **laters**, we obtain  $x'$  such that  $\llbracket f \rrbracket(x') = \llbracket f \rrbracket(x) = 0$  because  $\llbracket f \rrbracket$  respects  $\llbracket \approx \rrbracket$ . However,  $x'$  has  $n_0$  **laters** such that  $\llbracket f \rrbracket(x') = \llbracket f \rrbracket(\llbracket \text{never} \rrbracket) = 1$ , contradicts to the just established statement.

Similarly, utilising the sequence interpretation of  $A_\perp$ , we can show that any endofunction  $f : A_\perp \rightarrow A_\perp$  that respects  $\approx$  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.  $\square$

## 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 [80]). Both Dedekind reals and Cauchy reals have been considered. They provide a higher inductive definition of Cauchy reals using Cauchy approximations so that it is Cauchy complete.

Finite multisets as bag equivalent lists have been considered by Danielsson in [35]. 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  $\mathbb{R}_0 \rightarrow \mathbf{2}$  which respects  $\sim$  is constant. It 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 arise 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 general, but we have not yet prove it formally.





# Chapter 6

## The Setoid Model

To introduce extensional concepts into Intensional Type Theory, one can simply postulate them as axioms but we lose 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 does not allow us to enforce equalities on types. Therefore it is essential to interpret types with equalities, for example, setoids, groupoids, or  $\omega$ -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 irrelevance universe **Prop** in Agda, but the current version of Agda supports some proof-irrelevance features [1], for example proof-irrelevant field in record types, proof-irrelevant argument 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  $\Pi$ -types, natural numbers and the simply typed universe. We also extend it to  $\Sigma$ -types and quotient types which 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 [46]. However a naïve version of setoid model does not satisfy all definitional equalities. A simple model for quotient types introduced in [45] 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  $\eta$ -rules for  $\Pi$ -types and  $\Sigma$ -types hold.

$$\frac{\Gamma \vdash P : \mathbf{Prop} \quad \Gamma \vdash p, q : P}{\Gamma \vdash p \equiv q : P} \quad (\text{PROOF-IRR})$$

**Prop** only contains "propositional" sets which has 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  $\Pi$ -types and  $\Sigma$ -types:

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

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

The metatheory has been proved [3] 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 holds.

- *Adequate.* All terms of type  $\mathbb{N}$  are reducible to numerals.

And then Altenkirch constructs an intensional setoid model within this metatheory using categories with families as introduced by Dybjer [37] and Hofmann [48]. It is also decidable and adequate, 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 [44]. 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 it **E-setoids**. All morphisms of **E-setoids** gives rise to types and they are Cartesian closed, namely the category is locally Cartesian closed.

However not all morphisms in our category of setoids **Std** give rise to types which means that it is not a LCCC. Every LCCC can serve as a model for categories with families but not every category with families has to be a LCCC. Altenkirch and Kraus [6] have shown that both **Gpd** and **Std** are Cartesian closed but not locally Cartesian closed by giving a morphism, whose pullback functor does not have a right adjoint, as a counterexample.

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 put 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 used in dependent function types, dependent products (record types). For example, we can define "subset" of  $A$  with respect to a predicate  $B : A \rightarrow \text{Set}$  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
```

Thus, the proposition that the term fulfils the predicate is irrelevant.

We can also declare a function itself is irrelevant.

```
.g : A → B
g a = b
```

There are several restrictions of the 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.

There is a small problem of irrelevant fields of record types as we will see later: we can not use a irrelevant value to construct a 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 = ?
```

Because the result type cannot declared as irrelevant, although the function (or field) is irrelevant. The problem can be temporarily fixed by using 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 a small bit of our code, e.g. the construction of natural numbers and universes in setoid model.

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  $\sim: A \rightarrow A \rightarrow \mathbf{Prop}$  for a propositional equivalence relation in original metatheory. However in our implementation, we write  $\sim: A \rightarrow A \rightarrow \mathbf{Set}$ , 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  $\Sigma$ -types, but is not "closed" under  $\Pi$ -types, because we cannot declare it result type as irrelevant. Instead, we have to declare a  $\Pi$ -type itself is irrelevant.

This metatheory is still decidable, consistent and should be adequate because the only axiom is irrelevant which can not be used to construct non-canonical terms of  $\mathbb{N}$ .

### 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  $S : \text{Setoid}$ , one has to write  $\_ \approx \_ A a b$  which is not readable. By renaming, we can write  $[A] a \approx b$  for a better style. We will also rename some fields for other records types later, but we may omit the code which is unnecessary.

Functions between setoids consists of a function on 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 which has a unique homomorphism for any object in this category:

```

● : Setoid
● = record {
  Carrier = T;
  _≈_ = λ _ _ → T;
  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 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  $\mathbf{C}$  with a terminal object where objects are contexts and morphisms are substitutions.
- A functor  $F : \mathbf{C}^{op} \rightarrow \mathbf{Fam}$  where  $\mathbf{Fam}$  is a category of families whose objects are dependent products and morphisms are naturally pairs of functions. The types and terms are projections of this functor: given an object  $\Gamma : \mathbf{C}$ , we usually write  $F(\Gamma) \equiv \Sigma(A : \mathbf{Ty} \ \Gamma) \ \mathbf{Tm} \ \Gamma \ A$ , and the substitution of types and terms are just contained in the morphism part of this functor.
- A comprehension of  $\Gamma$  and  $A : \mathbf{Ty} \ \Gamma$  written as  $\Gamma, A$  (or  $\Gamma \& A$ ) is a construction of a new object in  $\mathbf{C}$  which expresses the extension of contexts.

In the setoid model, the category of contexts is just **Std**,

**Con** = **Setoid**

And the semantic types  $\mathbf{Ty}$  and semantic terms  $\mathbf{Tm}$  can be interpreted as the functor.

Given a context  $\Gamma$ , types over it can be defined as functors from  $\Gamma$  to **Std** because since a context is a setoid which are special cases of groupoids. However setoids are not implemented as categories, so we build a semantic type  $A : \mathbf{Ty} \ \Gamma$  (a functor) as follows:

```
record Ty (Γ : Con) : Set1 where
  field
    fm : | Γ | → Con
```



$$\begin{aligned}
&\text{substT} : \{x\ y : \mid \Gamma\} \rightarrow \\
&\quad .([ \Gamma ]\ x \approx y) \rightarrow \\
&\quad \mid \text{fm } x \mid \rightarrow \\
&\quad \mid \text{fm } y \mid \\
&\text{.subst}^* : \forall \{x\ y : \mid \Gamma\} \\
&\quad (p : ([ \Gamma ]\ x \approx y)) \\
&\quad \{a\ b : \mid \text{fm } x \mid\} \rightarrow \\
&\quad .([ \text{fm } x ]\ a \approx b) \rightarrow \\
&\quad ([ \text{fm } y ]\ \text{substT } p\ a \approx \text{substT } p\ b) \\
\\
&\text{.refl}^* : \forall \{x : \mid \Gamma\} \{a : \mid \text{fm } x \mid\} \rightarrow \\
&\quad [ \text{fm } x ]\ \text{substT } ([ \Gamma ]\ \text{refl})\ a \approx a \\
&\text{.trans}^* : \forall \{x\ y\ z : \mid \Gamma\} \\
&\quad \{p : [ \Gamma ]\ x \approx y\} \\
&\quad \{q : [ \Gamma ]\ y \approx z\} \\
&\quad (a : \mid \text{fm } x \mid) \rightarrow \\
&\quad [ \text{fm } z ]\ \text{substT } q\ (\text{substT } p\ a) \\
&\quad \approx \text{substT } ([ \Gamma ]\ \text{trans } p\ q)\ a \\
\\
&\text{.tr}^* : \forall \{x\ y : \mid \Gamma\} \\
&\quad \{p : [ \Gamma ]\ y \approx x\} \\
&\quad \{q : [ \Gamma ]\ x \approx y\} \\
&\quad \{a : \mid \text{fm } x \mid\} \rightarrow \\
&\quad [ \text{fm } x ]\ \text{substT } p\ (\text{substT } q\ a) \approx a \\
&\text{tr}^* = [ \text{fm } \_ ]\ \text{trans } (\text{trans}^* \_) \text{ refl}^* \\
\\
&\text{substT-inv} : \{x\ y : \mid \Gamma\} \rightarrow \\
&\quad .([ \Gamma ]\ x \approx y) \rightarrow \\
&\quad \mid \text{fm } y \mid \rightarrow \\
&\quad \mid \text{fm } x \mid \\
&\text{substT-inv } p\ y = \text{substT } ([ \Gamma ]\ \text{sym } p)\ y
\end{aligned}$$

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  $f : x \sim x$  and  $g : y \sim x$ , 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 it has to respect 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 _ $\Rightarrow$ _ f

```

`refl*` and `trans*` can also be verified easily because of proof irrelevance. We simplify our definition by open special record types, i.e. type  $A$  and morphism  $f$ , which is not ambiguous in the scope.

The substitution of terms is similar:

```

_ $\llbracket$ _  $\rrbracket$  m :  $\forall \{ \Gamma \Delta : \text{Con} \} \{ A : \text{Ty } \Delta \} \rightarrow \text{Tm } A$ 
   $\rightarrow (f : \Gamma \Rightarrow \Delta) \rightarrow \text{Tm } (A \llbracket f \rrbracket T)$ 
_ $\llbracket$ _  $\rrbracket$  m t f = record
  { tm =  $\llbracket t \rrbracket$  tm  $\circ$   $\llbracket f \rrbracket$  fn
    ; respt =  $\llbracket t \rrbracket$  respt  $\circ$   $\llbracket f \rrbracket$  resp
  }

```

Empty context is just terminal object of **Std** as we seen before.

Given a context  $\Gamma$  and a type  $A : \text{Ty } \Gamma$ , 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 by  $\Sigma$ -types.

```

_ $\&$ _ : ( $\Gamma : \text{Setoid}$ )  $\rightarrow \text{Ty } \Gamma \rightarrow \text{Setoid}$ 
 $\Gamma \& A =$ 
  record { Carrier =  $\Sigma [x : \Gamma] \mid \text{fm } x \mid$ 
    ;  $\_ \approx \_ = \lambda \{ (x, a) (y, b) \rightarrow$ 
       $\Sigma [p : x \approx y] \llbracket \text{fm } y \rrbracket (\text{substT } p \ a) \approx b \}$ 
    ; refl = refl , refl*
    ; sym =  $\lambda \{ (p, q) \rightarrow (\text{sym } p) ,$ 
       $\llbracket \text{fm } \_ \rrbracket \text{trans } (\text{subst}^* \_ (\llbracket \text{fm } \_ \rrbracket \text{sym } q)) \text{tr}^* \}$ 
    ; trans =  $\lambda \{ (p, q) (m, n) \rightarrow \text{trans } p \ m ,$ 

```

```

[ fm _ ]trans ([ fm _ ]trans
([ fm _ ]sym (trans* _)) (subst* _ q)) n}
}
where
  open Setoid  $\Gamma$ 
  open Ty A

```

The new relation is also an equivalence following from the properties of  $\Gamma$  as a setoid and the properties of  $A$ . Since the context  $\Gamma$  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  $\Pi$ -types and dependent product types  $\Sigma$ -types are essential in a dependent type theory. Intuitively, they are just  $\Pi$ -types and  $\Sigma$ -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, but due to the complication of the codes, we keep them in appendix (see Appendix B).

We have also built some basic types including natural numbers, a simply typed universe and equality types. For natural numbers, equality can be recursively defined as

```

_  $\approx_{\text{nat}}$  _ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$ 
zero  $\approx_{\text{nat}}$  zero =  $\top$ 
zero  $\approx_{\text{nat}}$  suc n =  $\perp$ 
suc m  $\approx_{\text{nat}}$  zero =  $\perp$ 
suc m  $\approx_{\text{nat}}$  suc n = m  $\approx_{\text{nat}}$  n

```

The problem of irrelevancy annotation arises here: to prove the *irrelevant* field `subst*`, we can not simply use its *irrelevant* argument because the result type is not declared irrelevant. Therefore we use the *irrelevant* axiom as we mentioned before to fix the problem.

```
subst* = λ _ x → irrelevant x
```

### 6.3.2 Quotient types

We build our quotient types in an Agda module. Given a context  $\Gamma$ , and a type  $A : \text{Ty } \Gamma$ ,

```
module Q (Γ : Con)(A : Ty Γ)
```

we can build a quotient type if we have an equivalence relation  $R$  defined on  $A$  which have to respect the underlying equivalence of  $A$ . In principle, the type of  $R$  should be  $\mathbf{Tm} (\Pi A \Pi A^+ \mathbf{Prop})$  where  $A^+ :\equiv A \text{ [fst]}$  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:

$$\begin{aligned} &.(R_{\text{respt}} : \forall \{\gamma \ \gamma' : |\Gamma|\} \\ &\quad (p : [\Gamma] \ \gamma \approx \gamma') \\ &\quad (a \ b : |[A]_{\text{fm}} \ \gamma|) \rightarrow \\ &\quad .(R \ \gamma \ a \ b) \rightarrow \\ &\quad R \ \gamma' \ ([A]_{\text{subst}} \ p \ a) \ ([A]_{\text{subst}} \ p \ b)) \end{aligned}$$

Of course, because it is defined on type  $A$ , it has to respect equality (equivalence) of  $A$ .

$$.(R_{\text{rsp}} : \forall \{\gamma \ a \ b\} \rightarrow .([A]_{\text{fm}} \ \gamma) \ a \approx b \rightarrow R \ \gamma \ a \ b)$$

It is an equivalence relation, so we have reflexivity, symmetry and transitivity.

$$\begin{aligned} &.(R_{\text{ref}} : \forall \{\gamma \ a\} \rightarrow R \ \gamma \ a \ a) \\ &.(R_{\text{sym}} : (\forall \{\gamma \ a \ b\} \rightarrow .(R \ \gamma \ a \ b) \rightarrow R \ \gamma \ b \ a)) \\ &.(R_{\text{trn}} : (\forall \{\gamma \ a \ b \ c\} \rightarrow .(R \ \gamma \ a \ b) \\ &\quad \rightarrow .(R \ \gamma \ b \ c) \rightarrow R \ \gamma \ a \ c)) \end{aligned}$$

The quotient type  $Q$  shares the same underlying set with  $A$ , but the internal equality is replaced by  $R$ .

$$[[Q]]_0 : |\Gamma| \rightarrow \text{Setoid}$$

```

[[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  $f : A \rightarrow B$  which respects  $R$ , we can lift it as a function of type  $Q \rightarrow B$  whose underlying function is the same as  $f$ . Because it respects  $R$ , the lifted function is a 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  $P : Q \rightarrow \mathbf{Set}$  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 [74] has also shown a formalisation of constructive set theory in terms of setoids in Intensional Type Theory. They has considered it as a solution to the problem that the uniqueness of identity proofs for sets are not derivable from  $\mathbf{J}$  eliminator.

The categories with families (CwFs) were introduced by Dybjer [37] as a model of dependent types which can be defined in Intensional Type Theory. Hofmann [48] has also explained the categorical semantics of dependent types provided by CwFs. Clairambault [26] has shown that categories with families are locally Cartesian closed after some additional structures are added such as  $\Pi$ -types,  $\Sigma$ -types, identity types etc.

In [44] 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 [45, 46] 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  $\text{Prf}(P)$  for each  $P : \mathbf{Prop}$ . He has provided interpretations of both propositions and quotient types with a choice operator. He has also proposed a groupoid model [46, 49] 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  $\mathbf{K}$  eliminator. However the groupoid model uses Extensional Type Theory as meta-theory.

## 6.5 Summary

In this chapter we have seen an implementation of the Altenkirch's setoid model with a slightly difference in the metatheory. We have used irrelevance feature of Agda to imitate the proof-irrelevance 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 replacing the internal equivalence of a type  $A$  as a setoid with a given equivalence relation on it.

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’ *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 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 behaves like sets and it helps us avoid the heavy use of **subst** which complicates formalisation and reasoning, for example, in Section 7.1.2 we use it for syntactic terms.



# Chapter 7

## Syntactic $\omega$ -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  $\omega$ -groupoid* (see Section 2.6.2) and was proposed by Grothendieck 1983 in a famous manuscript *Pursuing Stacks* [42]. Maltsiniotis continued his work and suggested a simplification of the original definition which can be found in [61]. Later Ara also presents a slight variation of the simplification of weak  $\omega$ -groupoids in [12]. Categorically speaking an  $\omega$ -groupoid is an  $\omega$ -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  $\omega$ -groupoid (also called  $\infty$ -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  $\omega$ -groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak  $\omega$ -groupoids in Type Theory, for instance, Altenkirch and Rypáček [7], and Brunerie’s notes [24].

In this chapter, our implementation of weak  $\omega$ -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  $\omega$ -groupoid by first defining a type theory called  $\mathcal{T}_{\infty\text{-groupoid}}$  to describe the internal language of Grothendieck weak  $\omega$ -groupoids, then interpret it with a globular set and a dependent function to it. All coherence laws of weak  $\omega$ -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 [58].

## 7.1 Syntax of weak $\omega$ -groupoids

We develop the type theory of  $\omega$ -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  $\omega$ -groupoid. There are no definitional equalities, this corresponds to the fact that we consider *weak*  $\omega$ -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  $\omega$ -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 [68]. 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  $\mathcal{T}_{\infty\text{-groupoid}}$  namely the internal language of weak  $\omega$ -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 ( $\Gamma$  : Con) : Set
data Tm      : { $\Gamma$  : Con}(A : Ty  $\Gamma$ )  $\rightarrow$  Set
data Var      : { $\Gamma$  : Con}(A : Ty  $\Gamma$ )  $\rightarrow$  Set
data  $\_ \Rightarrow \_$  : Con  $\rightarrow$  Con  $\rightarrow$  Set
data isContr  : Con  $\rightarrow$  Set

```

Contexts are inductively defined as either an empty context or a context with a type in it.

```

data Con where
   $\epsilon$       : Con
   $\_, \_$     : ( $\Gamma$  : Con)(A : Ty  $\Gamma$ )  $\rightarrow$  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.

```

data Ty  $\Gamma$  where
   $*$       : Ty  $\Gamma$ 
   $\_ =_h \_$  : {A : Ty  $\Gamma$ }(a b : Tm A)  $\rightarrow$  Ty  $\Gamma$ 

```

### 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  $\omega$ -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 [68] used to resolve this issue is to define equality for terms of different types, but its inhabitants only for terms of definitionally equal types. However, the corresponding elimination principle relies on UIP. In Intensional Type Theory, UIP is not provable in general, namely not all types are h-sets (homotopy 0-types). However every type with decidable equality is an h-set. 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 [43] that any type with decidable equality 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 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.

```
data _≅_ {Γ : Con}{A : Ty Γ} :
  {B : Ty Γ} → Tm A → Tm B → Set where
  refl : (b : Tm A) → b ≅ b
```

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.

$$\begin{aligned}
\_ \llbracket \_ \rrbracket & : \{ \Gamma : \mathbf{Con} \} \{ A B : \mathbf{Ty} \Gamma \} (a : \mathbf{Tm} B) \\
& \rightarrow A \equiv B \rightarrow \mathbf{Tm} A \\
a \llbracket \mathbf{refl} \rrbracket & = a \\
\mathbf{cohOp} & : \{ \Gamma : \mathbf{Con} \} \{ A B : \mathbf{Ty} \Gamma \} \{ a : \mathbf{Tm} B \} (p : A \equiv B) \\
& \rightarrow a \llbracket p \rrbracket \cong a \\
\mathbf{cohOp} \mathbf{refl} & = \mathbf{refl} \_
\end{aligned}$$

### 7.1.3 Substitutions

In this chapter we usually define a set of functions together and we name a function  $x$  as  $x_C$  for contexts,  $x_T$  for types,  $x_V$  for variables  $x_{tm}$  for terms and  $x_S$  for context morphisms (substitutions) as conventions. For example the substitutions are declared as follows:

$$\begin{aligned}
\_ \llbracket \_ \rrbracket^T & : \forall \{ \Gamma \Delta \} \rightarrow \mathbf{Ty} \Delta \rightarrow \Gamma \Rightarrow \Delta \rightarrow \mathbf{Ty} \Gamma \\
\_ \llbracket \_ \rrbracket^V & : \forall \{ \Gamma \Delta A \} \rightarrow \mathbf{Var} A \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \mathbf{Tm} (A \llbracket \delta \rrbracket^T) \\
\_ \llbracket \_ \rrbracket^{tm} & : \forall \{ \Gamma \Delta A \} \rightarrow \mathbf{Tm} A \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \mathbf{Tm} (A \llbracket \delta \rrbracket^T)
\end{aligned}$$

Indeed, compositions of context morphisms can be understood as substitutions for context morphisms as well.

$$\_ \odot \_ : \forall \{ \Gamma \Delta \Theta \} \rightarrow \Delta \Rightarrow \Theta \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \Gamma \Rightarrow \Theta$$

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  $\_ \Rightarrow \_$  where
  •  $\_$  :  $\forall \{\Gamma\} \rightarrow \Gamma \Rightarrow \epsilon$ 
   $\_', \_$  :  $\forall \{\Gamma \Delta\} (\delta : \Gamma \Rightarrow \Delta) \{A : \text{Ty } \Delta\} (a : \text{Tm } (A \text{ [ } \delta \text{ ] } T))$ 
     $\rightarrow \Gamma \Rightarrow (\Delta , A)$ 

```

### 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.

```

 $\_ + T \_$  :  $\forall \{\Gamma\} (A : \text{Ty } \Gamma) (B : \text{Ty } \Gamma) \rightarrow \text{Ty } (\Gamma , B)$ 
 $\_ + \text{tm } \_$  :  $\forall \{\Gamma A\} (a : \text{Tm } A) (B : \text{Ty } \Gamma) \rightarrow \text{Tm } (A + T B)$ 
 $\_ + S \_$  :  $\forall \{\Gamma \Delta\} (\delta : \Gamma \Rightarrow \Delta) (B : \text{Ty } \Gamma) \rightarrow (\Gamma , B) \Rightarrow \Delta$ 

```

### 7.1.5 Terms

A term can be either a variable or a coherence constant ( $\text{coh}$ ).

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  $\text{vS}$ .

```

data Var where
  v0 :  $\forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow \text{Var } (A + T A)$ 
  vS :  $\forall \{\Gamma\} \{A B : \text{Ty } \Gamma\} (x : \text{Var } A) \rightarrow \text{Var } (A + T B)$ 

```

The coherence constants are the most crucial topic in *wog*. 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  :  $\forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow \text{Var } A \rightarrow \text{Tm } A$ 
  coh  :  $\forall \{\Gamma \Delta\} \rightarrow \text{isContr } \Delta \rightarrow (\delta : \Gamma \Rightarrow \Delta)$ 
         $\rightarrow (A : \text{Ty } \Delta) \rightarrow \text{Tm } (A [\delta ] T)$ 
```

### 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 ( $\epsilon$  ,  $*$ )
  ext  :  $\forall \{\Gamma\} \rightarrow \text{isContr } \Gamma \rightarrow \{A : \text{Ty } \Gamma\} (x : \text{Var } A)$ 
         $\rightarrow \text{isContr } (\Gamma , A , (\text{var } (vS \ x) =_h \text{var } v0))$ 
```

### 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:

$$\begin{aligned}
[\odot]T & : \forall\{\Gamma \Delta \Theta A\}\{\theta : \Delta \Rightarrow \Theta\}\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow A \ [\theta \odot \delta]T \equiv (A \ [\theta]T) [\delta]T \\
[\odot]v & : \forall\{\Gamma \Delta \Theta A\}(x : \text{Var } A)\{\theta : \Delta \Rightarrow \Theta\}\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow x \ [\theta \odot \delta]V \cong (x \ [\theta]V) \ [\delta]tm \\
[\odot]tm & : \forall\{\Gamma \Delta \Theta A\}(a : \text{Tm } A)\{\theta : \Delta \Rightarrow \Theta\}\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow a \ [\theta \odot \delta]tm \cong (a \ [\theta]tm) \ [\delta]tm \\
\odot\text{assoc} & : \forall\{\Gamma \Delta \Theta \Omega\}(\gamma : \Theta \Rightarrow \Omega)\{\theta : \Delta \Rightarrow \Theta\}\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow (\gamma \odot \theta) \odot \delta \equiv \gamma \odot (\theta \odot \delta)
\end{aligned}$$

The second set states that weakening inside substitution is equivalent to weakening outside:

$$\begin{aligned}
[+S]T & : \forall\{\Gamma \Delta A B\}\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow A \ [\delta +S B]T \equiv (A \ [\delta]T) +T B \\
[+S]tm & : \forall\{\Gamma \Delta A B\}(a : \text{Tm } A)\{\delta : \Gamma \Rightarrow \Delta\} \\
& \rightarrow a \ [\delta +S B]tm \cong (a \ [\delta]tm) +tm B \\
[+S]S & : \forall\{\Gamma \Delta \Theta B\}\{\delta : \Delta \Rightarrow \Theta\}\{\gamma : \Gamma \Rightarrow \Delta\} \\
& \rightarrow \delta \odot (\gamma +S B) \equiv (\delta \odot \gamma) +S B
\end{aligned}$$

We can cancel the last term in the substitution for weakened objects since weakening doesn't introduce new variables in types and terms.

$$\begin{aligned} +T[, ]T & : \forall \{ \Gamma \Delta A B \} \{ \delta : \Gamma \Rightarrow \Delta \} \{ b : Tm (B [ \delta ] T) \} \\ & \rightarrow (A +T B) [ \delta , b ] T \equiv A [ \delta ] T \end{aligned}$$

$$\begin{aligned} +tm[, ]tm & : \forall \{ \Gamma \Delta A B \} \{ \delta : \Gamma \Rightarrow \Delta \} \{ c : Tm (B [ \delta ] T) \} \\ & \rightarrow (a : Tm A) \\ & \rightarrow (a +tm B) [ \delta , c ] tm \cong a [ \delta ] tm \end{aligned}$$

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.

$$\begin{aligned} var\ x & [ \delta ] tm = x [ \delta ] V \\ coh\ c\Delta\ \gamma\ A & [ \delta ] tm = coh\ c\Delta\ (\gamma \odot \delta)\ A\ [\![\ sym\ [\odot] T ]\!] \end{aligned}$$

## 7.2 Some Important Derivable Constructions

In this section we show how to reconstruct the structure of a (weak)  $\omega$ -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  $level\ A \equiv n$ , where

$$\begin{aligned} level & : \forall \{ \Gamma \} \rightarrow Ty\ \Gamma \rightarrow \mathbb{N} \\ level\ * & = 0 \\ level\ (\_ =h\_ \{A\} \_ \_) & = suc\ (level\ A) \end{aligned}$$

In any  $\omega$ -category, any  $n$ -cell  $a$  has a domain (source),  $s_m^n a$ , and a codomain (target),  $t_m^n a$ , for each  $m \leq n$ . These are, of course,  $(n-m)$ -cells. For each pair

of  $n$ -cells such that for some  $m$ ,  $s_m^n a \equiv t_m^n b$ , there must exist their composition  $a \circ_m^n b$  which is an  $n$ -cell. Composition is (weakly) associative. Moreover for any  $(n-m)$ -cell  $x$  there exists an  $n$ -cell  $\text{id}_m^n x$  which behaves like a (weak) identity with respect to  $\circ_m^n$ . 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  $s_1^2 a$  and target  $t_1^2 a$ , which is also the source and target, of the horizontal source and target, respectively, of  $a$ . There is horizontal composition of 1-cells  $\circ_1^1: x \rightarrow^f y \rightarrow^g z$ , and also horizontal composition of 2-cells  $\circ_1^2$ , and vertical composition of 2-cells  $\circ_2^2$ . There is a horizontal identity on  $a$ ,  $\text{id}_1^1 a$ , and vertical identity on  $a$ ,  $\text{id}_1^2 a = \text{id}_2^2 \text{id}_1^1 a$ .

Thus each  $\omega$ -groupoid construction is defined with respect to a *level*,  $m$ , and depth  $n-m$  and the structure of an  $\omega$ -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  $\mathbf{Tm}, \mathbf{coh}$ , which tells us that to construct a new term of type  $\Gamma \vdash A$ , we need a contractible context,  $\Delta$ , a type  $\Delta \vdash T$  and a context morphism  $\delta : \Gamma \Rightarrow \Delta$  such that

$$\mathbf{T}[\delta]\mathbf{T} \equiv A$$

Because in a contractible context all types are inhabited we may in a way work freely in  $\Delta$  and then pull back all terms to  $A$  using  $\delta$ . To show this formally, we must first define identity context morphisms which complete the definition of a *category* of contexts and context morphisms:

$$\text{IdS} : \forall\{\Gamma\} \rightarrow \Gamma \Rightarrow \Gamma$$

It satisfies the following property:

$$\text{IC-T} : \forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow A \llbracket \text{IdS} \rrbracket T \equiv A$$

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:

$$\begin{aligned} \text{Coh-Contr} & : \forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow \text{isContr } \Gamma \rightarrow \text{Tm } A \\ \text{Coh-Contr isC} & = \text{coh isC IdS } \_ \llbracket \text{sym IC-T} \rrbracket \end{aligned}$$

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  $\Gamma$  of level  $n$  one can define a context with  $2n$  variables, called the *stalk* of  $A$ . Moreover one can define a morphism from  $\Gamma$  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:  $\varepsilon$  (the empty context) for  $n = 0$ ;  $(x_0 : *, x_1 : *)$  for  $n = 1$ ;  $(x_0 : *, x_1 : *, x_2 : x_0 =_h x_1, x_3 : x_0 =_h x_1)$  for  $n = 2$ , etc.

				6 7
			4 5	4 5
		2 3	2 3	2 3
	0 1	0 1	0 1	0 1
$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$



This is the  $\Delta = \varepsilon$  case of a more general construction where in we *suspend* an arbitrary context  $\Delta$  by adding  $2n$  variables to the beginning of it, and weakening the rest of the variables appropriately so that type  $*$  becomes  $x_{2n-2} =_h x_{2n-1}$ . 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.  $\Sigma C$  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.

$$\begin{aligned} \Sigma C &: \mathbf{Con} \rightarrow \mathbf{Con} \\ \Sigma T &: \forall \{\Gamma\} \rightarrow \mathbf{Ty} \Gamma \rightarrow \mathbf{Ty} (\Sigma C \Gamma) \\ \Sigma C \varepsilon &= \varepsilon, *, * \\ \Sigma C (\Gamma, A) &= \Sigma C \Gamma, \Sigma T A \end{aligned}$$

The rest of the definitions is straightforward by structural recursion. In particular we suspend variables, terms and context morphisms:

$$\begin{aligned} \Sigma v &: \forall \{\Gamma\} \{A : \mathbf{Ty} \Gamma\} \rightarrow \mathbf{Var} A \rightarrow \mathbf{Var} (\Sigma T A) \\ \Sigma tm &: \forall \{\Gamma\} \{A : \mathbf{Ty} \Gamma\} \rightarrow \mathbf{Tm} A \rightarrow \mathbf{Tm} (\Sigma T A) \\ \Sigma s &: \forall \{\Gamma \Delta\} \rightarrow \Gamma \Rightarrow \Delta \rightarrow \Sigma C \Gamma \Rightarrow \Sigma C \Delta \end{aligned}$$

The following lemma establishes preservation of contractibility by one-step suspension:

$$\Sigma C\text{-Contr} : \forall \Delta \rightarrow \mathbf{isContr} \Delta \rightarrow \mathbf{isContr} (\Sigma C \Delta)$$

It is also essential that suspension respects weakening and substitution:

$$\begin{aligned} \Sigma T[+T] & : \forall \{\Gamma\} (A : Ty \ \Gamma) \\ & \rightarrow \Sigma T (A +T B) \equiv \Sigma T A +T \Sigma T B \end{aligned}$$

$$\begin{aligned} \Sigma tm[+tm] & : \forall \{\Gamma \ A\} (a : Tm \ A) (B : Ty \ \Gamma) \\ & \rightarrow \Sigma tm (a +tm B) \cong \Sigma tm a +tm \Sigma T B \end{aligned}$$

$$\begin{aligned} \Sigma T[\Sigma s]T & : \forall \{\Gamma \ \Delta\} (A : Ty \ \Delta) (\delta : \Gamma \Rightarrow \Delta) \\ & \rightarrow (\Sigma T A) [\Sigma s \ \delta]T \equiv \Sigma T (A [\delta]T) \end{aligned}$$

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  $\Gamma$ , while they depend only on its level.

$$\Sigma C\text{-it} : \forall \{\Gamma\} (A : Ty \ \Gamma) \rightarrow Con \rightarrow Con$$

$$\Sigma T\text{-it} : \forall \{\Gamma \ \Delta\} (A : Ty \ \Gamma) \rightarrow Ty \ \Delta \rightarrow Ty \ (\Sigma C\text{-it} A \ \Delta)$$

$$\begin{aligned} \Sigma tm\text{-it} & : \forall \{\Gamma \ \Delta\} (A : Ty \ \Gamma) \{B : Ty \ \Delta\} \rightarrow Tm \ B \\ & \rightarrow Tm \ (\Sigma T\text{-it} A \ B) \end{aligned}$$

Finally, it is clear that iterated suspension preserves contractibility.

$$\begin{aligned} \Sigma C\text{-it-Contr} & : \forall \{\Gamma \ \Delta\} (A : Ty \ \Gamma) \rightarrow isContr \ \Delta \\ & \rightarrow isContr \ (\Sigma C\text{-it} A \ \Delta) \end{aligned}$$

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  $n = 0$ ;  $(x_0 : *, x_1 : *, x_2 : x_0 =_h x_1)$  for  $n = 1$ ;  $(x_0 : *, x_1 : *, x_2 : x_0 =_h x_1, x_3 : x_0 =_h x_1, x_4 : x_2 =_h x_3)$  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  $A$ .

				8
			6	6 7
		4	4 5	4 5
	2	2 3	2 3	2 3
0	0 1	0 1	0 1	0 1
$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$

### 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  $\Gamma$ , the context of  $A$ . The new context contains two parts, the first is the same as  $\Gamma$ , and the second is the suspended  $\Delta$  substituted by [filter](#). However, we also have to drop the stalk of  $A$  because it already exists in  $\Gamma$ .

This operation is called *replacement* because we can interpret it as replacing  $*$  in  $\Delta$  by  $A$ .

As always, we define replacement for contexts, types and terms simultaneously:

$$\begin{aligned}
 \text{rpl-C} & : \forall \{\Gamma\} (A : \text{Ty } \Gamma) \rightarrow \text{Con} \rightarrow \text{Con} \\
 \text{rpl-T} & : \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) \rightarrow \text{Ty } \Delta \rightarrow \text{Ty } (\text{rpl-C } A \Delta) \\
 \text{rpl-tm} & : \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) \{B : \text{Ty } \Delta\} \rightarrow \text{Tm } B \\
 & \rightarrow \text{Tm } (\text{rpl-T } A B)
 \end{aligned}$$

Replacement for contexts, [rpl-C](#), defines for a type  $A$  in  $\Gamma$  and another context  $\Delta$  a context which begins as  $\Gamma$  and follows by each type of  $\Delta$  with  $*$  replaced with (pasted onto)  $A$ .

$$\text{rpl-C } \{\Gamma\} A \varepsilon = \Gamma$$

$$\text{rpl-C } A (\Delta, B) = \text{rpl-C } A \Delta, \text{rpl-T } A B$$

To this end we must define the substitution `filter` which pulls back each type from suspended  $\Delta$  to the new context.

$$\begin{aligned} \text{filter} &: \forall \{\Gamma\} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \\ &\rightarrow \text{rpl-C } A \Delta \Rightarrow \Sigma\text{C-it } A \Delta \end{aligned}$$

$$\text{rpl-T } A B = \Sigma\text{T-it } A B [\text{filter } \_ A]^\top$$

### 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  $\Delta$  and use lifting and substitution – replacement – to pull the term built by `coh` in  $\Delta$  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.

$$\begin{aligned} \text{Coh-rpl} &: \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) (B : \text{Ty } \Delta) \rightarrow \text{isContr } \Delta \\ &\rightarrow \text{Tm } (\text{rpl-T } A B) \\ \text{Coh-rpl } \{\_ \} \{\Delta\} A \_ \text{isC} &= \text{coh } (\Sigma\text{C-it-}\epsilon\text{-Contr } A \text{ isC}) \_ \_ \end{aligned}$$

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:\*, $\alpha$ :x=y** indicate their definitions which have been hidden. Agda treats all sequences of characters uninterrupted by whitespace as identifiers. For instance **x:\*,y:\*, $\alpha$ :x=y** is a name of a context for which we are assuming the definition: **x:\*,y:\*, $\alpha$ :x=y** =  $\epsilon$  , \* , \* , (var (vS v0) =h var v0).

```
sym*-Ty : Ty x:*,y:*, $\alpha$ :x=y
sym*-Ty = vY =h vX

sym*-Tm : Tm {x:*,y:*, $\alpha$ :x=y} sym*-Ty
sym*-Tm = Coh-Contr (ext c* v0)
```

**Transitivity** (composition)

```
trans*-Ty : Ty x:*,y:*, $\alpha$ :x=y,z:*, $\beta$ :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      : { $\Gamma$  : Con}(A : Ty  $\Gamma$ )
```

$$\begin{aligned} & \rightarrow \text{Tm} (\text{rpl-T} \{ \Delta = x:* \} A (\text{var } v0 =_h \text{var } v0)) \\ \text{refl-Tm } A &= \text{rpl-tm } A \text{ refl*-Tm} \end{aligned}$$

$$\begin{aligned} \text{sym-Tm} &: \forall \{ \Gamma \} (A : \text{Ty } \Gamma) \rightarrow \text{Tm} (\text{rpl-T } A \text{ sym*-Ty}) \\ \text{sym-Tm } A &= \text{rpl-tm } A \text{ sym*-Tm} \end{aligned}$$

$$\begin{aligned} \text{trans-Tm} &: \forall \{ \Gamma \} (A : \text{Ty } \Gamma) \rightarrow \text{Tm} (\text{rpl-T } A \text{ trans*-Ty}) \\ \text{trans-Tm } A &= \text{rpl-tm } A \text{ trans*-Tm} \end{aligned}$$

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.

$$\begin{aligned} & \text{Tm-right-identity*} : \\ & \text{Tm } \{ x:*, y:*, \alpha: x=y \} (\text{trans*-Tm } [ \text{IdS} , vY , \text{reflY} ] \text{tm} \\ & \quad =_h v\alpha) \\ & \text{Tm-right-identity*} = \text{Coh-Contr } (\text{ext } c^* v0) \end{aligned}$$

$$\begin{aligned} & \text{Tm-left-identity*} : \\ & \text{Tm } \{ x:*, y:*, \alpha: x=y \} (\text{trans*-Tm } [ ((\text{IdS} \odot \text{pr1} \odot \text{pr1}) , vX) , \\ & \quad \text{reflX} , vY , v\alpha ] \text{tm} =_h v\alpha) \\ & \text{Tm-left-identity*} = \text{Coh-Contr } (\text{ext } c^* v0) \end{aligned}$$

$$\begin{aligned} & \text{Tm-right-inverse*} : \\ & \text{Tm } \{ x:*, y:*, \alpha: x=y \} (\text{trans*-Tm } [ (\text{IdS} , vX) , \text{sym*-Tm} ] \text{tm} \\ & \quad =_h \text{reflX}) \\ & \text{Tm-right-inverse*} = \text{Coh-Contr } (\text{ext } c^* v0) \end{aligned}$$

$$\begin{aligned} & \text{Tm-left-inverse*} : \\ & \text{Tm } \{ x:*, y:*, \alpha: x=y \} (\text{trans*-Tm } [ ((\bullet , vY) , vX , \text{sym*-Tm} , \\ & \quad vY) , v\alpha ] \text{tm} =_h \text{reflY}) \end{aligned}$$

$$\text{Tm-left-inverse}^* = \text{Coh-Contr} (\text{ext } c^* v0)$$

$$\text{Tm-G-assoc}^* : \text{Tm Ty-G-assoc}^*$$

$$\begin{aligned} \text{Tm-G-assoc}^* &= \text{Coh-Contr} (\text{ext} (\text{ext} (\text{ext } c^* v0) (vS v0)) \\ &\quad (vS v0)) \end{aligned}$$

Their general versions are defined using replacement. For instance, for associativity, we define:

$$\begin{aligned} \text{Tm-G-assoc} &: \forall \{\Gamma\} (A : \text{Ty } \Gamma) \\ &\quad \rightarrow \text{Tm} (\text{rpl-T } A \text{ Ty-G-assoc}^*) \\ \text{Tm-G-assoc } A &= \text{rpl-tm } A \text{ Tm-G-assoc}^* \end{aligned}$$

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  $S_0 \Rightarrow S_1 \Rightarrow \dots \Rightarrow S_n$ . 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 <sup>1</sup>. 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 `ldw`.

---

<sup>1</sup>The Agda `Set` stands for an arbitrary type, not a set in the sense of Homotopy Type Theory.



$\text{Id}\omega : (A : \text{Set}) \rightarrow \text{Glob}$   
 $\text{Id}\omega A = A \parallel (\lambda a b \rightarrow \# \text{Id}\omega (a \equiv b))$

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
    π : ∀{Γ A} → Var A → (γ : [[ Γ ]]C)
      → | [[ A ]]T γ |

```

$\pi$  provides the projection of the semantic variable out of a semantic context.

Following are the computation laws for the interpretations of contexts and types.

$[[\_]]C\text{-}\beta 1 : [[ \varepsilon ]]C \equiv \top$   
 $[[\_]]C\text{-}\beta 2 : \forall \{ \Gamma A \} \rightarrow [[ \Gamma , A ]]C \equiv$   
 $\Sigma [[ \Gamma ]]C (\lambda \gamma \rightarrow | [[ A ]]T \gamma |)$   
 $[[\_]]T\text{-}\beta 1 : \forall \{ \Gamma \} \{ \gamma : [[ \Gamma ]]C \} \rightarrow [[ * ]]T \gamma \equiv G$   
 $[[\_]]T\text{-}\beta 2 : \forall \{ \Gamma A u v \} \{ \gamma : [[ \Gamma ]]C \}$   
 $\rightarrow [[ u =_h v ]]T \gamma \equiv$   
 $\flat (\text{hom } ([[ A ]]T \gamma) ([[ u ]]tm \gamma) ([[ v ]]tm \gamma))$

Semantic substitution and semantic weakening laws are also required. The semantic substitution properties are essential for dealing with substitutions inside interpretation,

$$\begin{aligned} \text{semSb-T} & : \forall \{\Gamma \Delta\} \{A : \text{Ty } \Delta\} (\delta : \Gamma \Rightarrow \Delta) (\gamma : \llbracket \Gamma \rrbracket C) \\ & \rightarrow \llbracket A \llbracket \delta \rrbracket T \rrbracket T \gamma \equiv \llbracket A \rrbracket T (\llbracket \delta \rrbracket S \gamma) \end{aligned}$$

$$\begin{aligned} \text{semSb-tm} & : \forall \{\Gamma \Delta\} \{A : \text{Ty } \Delta\} (a : \text{Tm } A) (\delta : \Gamma \Rightarrow \Delta) \\ & (\gamma : \llbracket \Gamma \rrbracket C) \rightarrow \text{subst } \llbracket \_ \rrbracket (\text{semSb-T } A \delta \gamma) \\ & (\llbracket a \llbracket \delta \rrbracket \text{tm} \rrbracket \text{tm } \gamma) \equiv \llbracket a \rrbracket \text{tm} (\llbracket \delta \rrbracket S \gamma) \end{aligned}$$

$$\begin{aligned} \text{semSb-S} & : \forall \{\Gamma \Delta \Theta\} (\gamma : \llbracket \Gamma \rrbracket C) (\delta : \Gamma \Rightarrow \Delta) \\ & (\theta : \Delta \Rightarrow \Theta) \rightarrow \llbracket \theta \odot \delta \rrbracket S \gamma \equiv \\ & \llbracket \theta \rrbracket S (\llbracket \delta \rrbracket S \gamma) \end{aligned}$$

Since the computation laws for the interpretations of terms and context morphisms are well typed up to these properties.

$$\begin{aligned} \llbracket \_ \rrbracket \text{tm-}\beta 1 & : \forall \{\Gamma A\} \{x : \text{Var } A\} \{\gamma : \llbracket \Gamma \rrbracket C\} \\ & \rightarrow \llbracket \text{var } x \rrbracket \text{tm } \gamma \equiv \pi x \gamma \end{aligned}$$

$$\begin{aligned} \llbracket \_ \rrbracket \text{S-}\beta 1 & : \forall \{\Gamma\} \{\gamma : \llbracket \Gamma \rrbracket C\} \\ & \rightarrow \llbracket \bullet \rrbracket S \gamma \equiv \text{coerce } \llbracket \_ \rrbracket \text{C-}\beta 1 \text{ tt} \end{aligned}$$

$$\begin{aligned} \llbracket \_ \rrbracket \text{S-}\beta 2 & : \forall \{\Gamma \Delta\} \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \{\gamma : \llbracket \Gamma \rrbracket C\} \\ & \{a : \text{Tm } (A \llbracket \delta \rrbracket T)\} \rightarrow \llbracket \delta , a \rrbracket S \gamma \\ & \equiv \text{coerce } \llbracket \_ \rrbracket \text{C-}\beta 2 ((\llbracket \delta \rrbracket S \gamma) , \\ & \text{subst } \llbracket \_ \rrbracket (\text{semSb-T } A \delta \gamma) (\llbracket a \rrbracket \text{tm } \gamma)) \end{aligned}$$

The semantic weakening properties should actually be derivable since weakening is equivalent to projection substitution.

$$\begin{aligned}
\text{semWk-T} & : \forall \{ \Gamma A B \} (\gamma : \llbracket \Gamma \rrbracket C) (v : | \llbracket B \rrbracket T \gamma |) \\
& \rightarrow \llbracket A +T B \rrbracket T (\text{coerce } \llbracket \_ \rrbracket C\text{-}\beta 2 (\gamma , v)) \equiv \\
& \llbracket A \rrbracket T \gamma
\end{aligned}$$

$$\begin{aligned}
\text{semWk-S} & : \forall \{ \Gamma \Delta B \} \{ \gamma : \llbracket \Gamma \rrbracket C \} \{ v : | \llbracket B \rrbracket T \gamma | \} \\
& \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \llbracket \delta +S B \rrbracket S \\
& (\text{coerce } \llbracket \_ \rrbracket C\text{-}\beta 2 (\gamma , v)) \equiv \llbracket \delta \rrbracket S \gamma
\end{aligned}$$

$$\begin{aligned}
\text{semWk-tm} & : \forall \{ \Gamma A B \} (\gamma : \llbracket \Gamma \rrbracket C) (v : | \llbracket B \rrbracket T \gamma |) \\
& \rightarrow (a : Tm A) \rightarrow \text{subst } | \_ | (\text{semWk-T } \gamma v) \\
& (\llbracket a +tm B \rrbracket tm (\text{coerce } \llbracket \_ \rrbracket C\text{-}\beta 2 (\gamma , v))) \\
& \equiv (\llbracket a \rrbracket tm \gamma)
\end{aligned}$$

Here we declare them as properties because they are essential for the computation laws of function  $\pi$ .

$$\begin{aligned}
\pi\text{-}\beta 1 & : \forall \{ \Gamma A \} (\gamma : \llbracket \Gamma \rrbracket C) (v : | \llbracket A \rrbracket T \gamma |) \\
& \rightarrow \text{subst } | \_ | (\text{semWk-T } \gamma v) \\
& (\pi \text{ v0 } (\text{coerce } \llbracket \_ \rrbracket C\text{-}\beta 2 (\gamma , v))) \equiv v
\end{aligned}$$

$$\begin{aligned}
\pi\text{-}\beta 2 & : \forall \{ \Gamma A B \} (x : Var A) (\gamma : \llbracket \Gamma \rrbracket C) (v : | \llbracket B \rrbracket T \gamma |) \\
& \rightarrow \text{subst } | \_ | (\text{semWk-T } \gamma v) (\pi (\text{vS } \{ \Gamma \} \{ A \} \{ B \} x) \\
& (\text{coerce } \llbracket \_ \rrbracket C\text{-}\beta 2 (\gamma , v))) \equiv \pi x \gamma
\end{aligned}$$

The only part of the semantics where we have any freedom is the interpretation of the coherence constants:

$$\begin{aligned}
\llbracket \text{coh} \rrbracket & : \forall \{ \Theta \} \rightarrow \text{isContr } \Theta \rightarrow (A : Ty \Theta) \\
& \rightarrow (\theta : \llbracket \Theta \rrbracket C) \rightarrow | \llbracket A \rrbracket T \theta |
\end{aligned}$$

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  $\mathbf{Id}\omega$  is not an instance anymore. 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  $\omega$ -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  $\omega$ -groupoids.

## 7.4 Related work

The groupoid interpretation of Martin-Löf type theory was first proposed to Hofmann and Streicher [49]. Sozeau and Tabareau [67] have formalised it in Coq. They have also considered to generalise their definitions to  $\omega$ -groupoids in the future. Warren [91] has shown an interpretation of Type Theory using *strict*  $\omega$ -groupoids. Lumsdaine [60], van den Berg and Garner [82] have shown that  $\mathbf{J}$  eliminator gives rise to a weak  $\omega$ -groupoid, van den Berg and Garner have proved that that every type is a weak  $\omega$ -groupoid. Altenkirch and Rypáček [7] have proposed a syntactic formalisation of weak  $\omega$ -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  $\omega$ -groupoids following Brunerie's suggestion. Briefly speaking, we defined the syntax of the type

theory  $\mathcal{T}_{\infty\text{-groupoid}}$ , then a weak  $\omega$ -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 of that  $\mathbf{Id}\omega$  is a weak  $\omega$ -groupoid. As Altenkirch suggests, we can potentially solve the problem that our definition of  $\mathbf{Id}\omega$  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  $\mathbf{Id}\omega$  as a globular set in this universe. Finally the most challenge task would be to model Type Theory with weak  $\omega$ -groupoids and to eliminate the univalence axiom.



## Chapter 8

# Conclusion and Future Work

We have presented the evolution of theories of types especially Martin-Löf type theory (Type Theory) and discussed different variants of it. We have 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 gives us these extensional concepts but its type checking is undecidable due to the identification of propositional equality and definitional equality.

The notion of quotient types is one of the important extensional concepts which can facilitate mathematical and programming constructions. Because of the good computational properties such as decidable type checking, ITT is usually preferable to be implemented as a programming language compared to ETT and we would like to have quotient types in it. We have presented a definition of quotient types in a type theory with a proof-irrelevant universe, and shown that simply add it into Intensional Type Theory as an axiom makes the theory inadequate, i.e. canonicity is lost. We have also made clear its correspondence to coequalizers in **Set** and a left adjoint functor in category theory.

We have 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

have proposed an algebraic structure to bridge the setoid representations and set definitions. We have shown that the application of definable quotient structure can improve the constructions by keeping good properties of both representations by providing some applications. Because it can be seen as a simulation of quotient types, we can also expect similar benefits from the applications of quotient types.

For the definable quotient structure, a potential future project would be to complete the implementation of numbers in Agda with the help of definable quotients. There are also other definable quotients implementable in our algebraic quotient structures. It can enrich the library of Agda for more potential mathematical proofs. We can also extend Agda with normalised types [33], namely to build 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 unnecessary for definable quotients, it seems indispensable for some other quotients whose normalisation functions are not definable. With the assumption of Brouwer's continuity holds in meta-theory, we have shown a proof that there is no definable normalisation function for Cauchy reals  $\mathbb{R}_0/\sim$ . There are also other examples like partiality monad, 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.

The solution to introduce quotient types in Intensional Type Theory without losing good computational properties is to build models where types are interpreted as sets with equality internally defined, such as setoids, groupoids or weak  $\omega$ -groupoids. We have developed an implementation of Altenkirch's setoid model in Agda, and explained our construction of quotient types inside of it.

For the setoid model, there are more details to work out. For example the verification of properties, to define a type for propositions such that we can write the type of equivalence relation using  $\Pi$ -types. We can also simplify the setoid model by using heterogeneous equality as we discussed in Chapter 6. We have also considered to use h-propositions in place of the universe of propositions in the metatheory. However it requires functional extensionality to prove the  $\Pi$ -closure



of h-propositions. It would still 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 have also investigated the new extension of Martin-Löf type theory– Homotopy Type Theory. In Homotopy Type Theory, types are interpreted as weak  $\omega$ -groupoids which is a generalization of a groupoid. We have 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 computation interpretation of Homotopy Type Theory can also been seen as a solution to quotient types in Intensional Type Theory.

We have shown a syntactic construction of weak  $\omega$ -groupoids in Agda as a first step to build a weak  $\omega$ -groupoid model of Type Theory. We have defined a type theory  $\mathcal{T}_{\infty\text{-groupoid}}$  to describe the coherence conditions for a globular set to become a weak  $\omega$ -groupoid. We have shown some constructions of the coherences laws, for example groupoid laws inside this theory by using suspensions and replacement techniques. We also use heterogeneous equality for terms to overcome technical issues in implementation.

There are still a lot work to do in the syntactic framework. For instance, we would like to investigate the relation between the  $\mathcal{T}_{\infty\text{-groupoid}}$  and a type theory with equality types and J eliminator which is called  $\mathcal{T}_{eq}$ . One direction is to simulate the J eliminator syntactically in  $\mathcal{T}_{\infty\text{-groupoid}}$  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  $\omega$ -groupoid. The syntax could be simplified by adopting categories with families. An alternative may be to use higher inductive types directly to formalize the syntax of type theory.

We would like to formalise a proof of that `ldw` is a weak  $\omega$ -groupoid, but the base set in a globular set is an h-set which is incompatible with `ldw`. Perhaps as Altenkirch suggests [4], we can solve the problem by using a universe with extensional equality, and Agda's propositional equality as strict equality so that

we can define  $\mathbf{ld}\omega$  as a globular set in this universe. Finally to model Type Theory with weak  $\omega$ -groupoids and to eliminate 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 _ ~ _ =  $\forall \{a \ a'\} \rightarrow a \sim a' \rightarrow f \ a \equiv f \ a'$ 
```

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')

```

$$\begin{aligned}
&\rightarrow \text{subst } B \text{ [ p ]}^= (f \ a) \equiv f \ a') \\
&\rightarrow (q : Q) \rightarrow B \ q \\
\text{qelim-}\beta &: \forall \{B \ a \ f\} \\
&(\text{resp} : (\forall \{a \ a'\} \rightarrow (p : a \sim a') \\
&\rightarrow \text{subst } B \text{ [ p ]}^= (f \ a) \equiv f \ a')) \\
&\rightarrow \text{qelim } \{B\} \ f \ \text{resp} \text{ [ a ]} \equiv f \ a
\end{aligned}$$

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** :  $\forall q \rightarrow [ \text{emb } q ] \equiv q$

**Proof** : Definable quotients are exact.

**exact** :  $\forall \{a\} \{b\} \rightarrow [ a ] \equiv [ b ] \rightarrow a \sim b$   
**exact** {a} {b} p =  
 ~-trans (~-sym (complete a))  
 (~-trans (subst ( $\lambda x \rightarrow$   
 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**  $\rightarrow$  **Quotient** :  $\{S : \text{Setoid}\} \{PQ : \text{pre-Quotient } S\} \rightarrow$   
 (Hof-Quotient PQ)  $\rightarrow$  (Quotient PQ)  
**Hof-Quotient**  $\rightarrow$  **Quotient** {S} {PQ} QuH =  
 record  
 { **qelim** =  $\lambda \{B\} f \text{ resp}$   
 $\rightarrow \text{proj}_1 (\text{qelim}' f \text{ resp})$   
 ; **qelim- $\beta$**  =  $\lambda \{B\} \{a\} \{f\} \text{ resp}$   
 $\rightarrow \text{proj}_2 (\text{qelim}' f \text{ resp})$   
 }  
 where  
 open pre-Quotient PQ  
 open Hof-Quotient QuH  
  
**qelim'** :  $\{B : Q \rightarrow \text{Set}\}$   
 $\rightarrow (f : (a : A) \rightarrow B [ a ])$   
 $\rightarrow (\forall \{a\} \{a'\} \rightarrow (p : a \sim a'))$

$$\begin{aligned}
&\rightarrow \text{subst } B \text{ [ p ]} = (f \ a) \equiv f \ a') \\
&\rightarrow \Sigma [ f^\wedge : ((q : Q) \rightarrow B \ q) ] \\
&\quad (\forall \{a\} \rightarrow f^\wedge \text{ [ a ]} \equiv f \ a) \\
&\text{qelim}' \{B\} f \text{ resp} = f^\wedge, f^\wedge\text{-}\beta
\end{aligned}$$

where

$$\begin{aligned}
f_0 &: A \rightarrow \Sigma Q \ B \\
f_0 \ a &= \text{[ a ]}, f \ a
\end{aligned}$$

$$\begin{aligned}
\text{resp}_0 &: f_0 \text{ respects } \_ \sim \_ \\
\text{resp}_0 \ p &= \Sigma \text{eq} \text{ [ p ]} = (\text{resp } p)
\end{aligned}$$

$$\begin{aligned}
f' &: Q \rightarrow \Sigma Q \ B \\
f' &= \text{lift } f_0 \ \text{resp}_0
\end{aligned}$$

$$\begin{aligned}
\text{id}' &: Q \rightarrow Q \\
\text{id}' &= \text{proj}_1 \circ f'
\end{aligned}$$

$$\begin{aligned}
P &: Q \rightarrow \text{Set} \\
P \ q &= \text{id}' \ q \equiv q
\end{aligned}$$

$$\begin{aligned}
f'\text{-}\beta &: \{a : A\} \rightarrow f' \text{ [ a ]} \equiv \text{[ a ]}, f \ a \\
f'\text{-}\beta &= \text{lift-}\beta \ \_
\end{aligned}$$

$$\begin{aligned}
\text{islda} &: \forall \{a\} \rightarrow \text{id}' \text{ [ a ]} \equiv \text{[ a ]} \\
\text{islda} &= \text{cong } \text{proj}_1 \ f'\text{-}\beta
\end{aligned}$$

$$\begin{aligned}
\text{isldq} &: \forall \{q\} \rightarrow \text{id}' \ q \equiv q \\
\text{isldq} \ \{q\} &= \text{qind } P \equiv \text{prop} \ (\lambda \_ \rightarrow \text{islda}) \ q
\end{aligned}$$

$$\begin{aligned}
f^\wedge &: (q : Q) \rightarrow B \ q \\
f^\wedge \ q &= \text{subst } B \ \text{isldq} \ (\text{proj}_2 \ (f' \ q))
\end{aligned}$$



$$\begin{aligned}
& \text{f'-sound2} : \forall \{a\} \rightarrow \\
& \quad \text{subst } B \text{ isIdea } (\text{proj}_2 (f' [a])) \equiv f a \\
& \text{f'-sound2} = \text{cong-proj}_2 \_ \_ f'\text{-}\beta \\
& f'\text{-}\beta : \forall \{a\} \rightarrow f^{\wedge} [a] \equiv f a \\
& f'\text{-}\beta \{a\} = \text{trans } (\text{subIrr isIdq isIdea}) f'\text{-sound2}
\end{aligned}$$

Quotient→Hof-Quotient :

$$\begin{aligned}
& \{S : \text{Setoid}\} \{PQ : \text{pre-Quotient } S\} \\
& \rightarrow (\text{Quotient } PQ) \\
& \rightarrow (\text{Hof-Quotient } PQ) \\
\text{Quotient}\rightarrow\text{Hof-Quotient } \{S\} \{PQ\} \text{QU} = \\
& \text{record} \\
& \{ \text{lift} = \lambda f \text{ resp} \\
& \quad \rightarrow \text{qelim } f (\text{resp}' \text{ resp}) \\
& ; \text{lift-}\beta = \lambda \text{ resp} \\
& \quad \rightarrow \text{qelim-}\beta (\text{resp}' \text{ resp}) \\
& ; \text{qind} = \lambda P \text{ isP } f \\
& \quad \rightarrow \text{qelim } \{P\} f (\lambda \_ \rightarrow \text{isP } \_ \_) \\
& \} \\
& \text{where} \\
& \text{open pre-Quotient } PQ \\
& \text{open Quotient } QU \\
& \text{resp}' : \{B : \text{Set}\} \{a a' : A\} \\
& \quad \{f : A \rightarrow B\} \\
& \quad (\text{resp} : f \text{ respects } \_ \sim \_) \\
& \quad (p : a \sim a') \\
& \quad \rightarrow \text{subst } (\lambda \_ \rightarrow B) [p] = (f a) \\
& \quad \equiv f a' \\
& \text{resp}' \text{ resp } p = \\
& \quad \text{trans } (\text{subIrr2 } [p] =)
\end{aligned}$$

(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

$$\begin{aligned} \ulcorner \_ \urcorner & : \mathbb{Z} \rightarrow \mathbb{Z}_0 \\ \ulcorner + n \urcorner & = n , 0 \\ \ulcorner -\text{suc } n \urcorner & = 0 , \mathbb{N}.\text{suc } n \end{aligned}$$

Stability

$$\begin{aligned} \text{stable} & : \forall \{n\} \rightarrow [\ulcorner n \urcorner] \equiv n \\ \text{stable } \{+ n\} & = \text{refl} \\ \text{stable } \{-\text{suc } n\} & = \text{refl} \end{aligned}$$

Completeness

$$\begin{aligned} \text{compl} & : \forall n \rightarrow \ulcorner [n] \urcorner \sim n \\ \text{compl } (x , 0) & = \text{refl} \\ \text{compl } (0 , \text{suc } y) & = \text{refl} \\ \text{compl } (\text{suc } x , \text{suc } y) & = \sim\text{trans } (\text{compl } (x , y)) \\ & \quad (\text{sym } (\text{sm} + n \equiv m + \text{sn } x)) \end{aligned}$$

$$\begin{aligned} \text{sound}' & : \forall \{i j\} \rightarrow \ulcorner i \urcorner \sim \ulcorner j \urcorner \rightarrow i \equiv j \\ \text{sound}' \{+ i\} \{+ j\} \text{eqt} & = +\_ \star (+\text{r-cancel } 0 \text{ eqt}) \\ \text{sound}' \{+ i\} \{-\text{suc } j\} \text{eqt} & \text{with } i + \text{suc } j \neq 0 \text{ eqt} \\ \dots & \mid () \\ \text{sound}' \{-\text{suc } i\} \{+ j\} \text{eqt} & \text{with } j + \text{suc } i \neq 0 \langle \text{eqt} \rangle \\ \dots & \mid () \end{aligned}$$

$\text{sound}' \{ -\text{suc } i \} \{ -\text{suc } j \} \text{eqt} = -\text{suc } \_ \star \text{pred} \star \langle \text{eqt} \rangle$

Soundness

$\text{sound} : \forall \{x\ y\} \rightarrow x \sim y \rightarrow [x] \equiv [y]$   
 $\text{sound} \{x\} \{y\} x \sim y = \text{sound}' (\sim\text{trans} (\text{compl } \_))$   
 $(\sim\text{trans} (x \sim y) (\sim\text{sym} (\text{compl } \_))))$

The quotient definitions for  $\mathbb{Z}$

$\mathbb{Z}\text{-PreQu} : \text{pre-Quotient } \mathbb{Z}\text{-Setoid}$

$\mathbb{Z}\text{-PreQu} = \text{record}$

$\{ \text{Q} = \mathbb{Z}$   
 $; [\_] = [\_]$   
 $; [\_]^\# = \text{sound}$   
 $\}$

$\mathbb{Z}\text{-QuD} : \text{def-Quotient } \mathbb{Z}\text{-PreQu}$

$\mathbb{Z}\text{-QuD} = \text{record}$

$\{ \text{emb} = \ulcorner \_ \urcorner$   
 $; \text{complete} = \lambda z \rightarrow \text{compl } \_$   
 $; \text{stable} = \lambda z \rightarrow \text{stable}$   
 $\}$

$\mathbb{Z}\text{-Qu} = \text{def-Quotient} \rightarrow \text{Quotient } \mathbb{Z}\text{-QuD}$



## 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 coprime to the denominator

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

The Definition of  $\mathbb{Q}$  which is equivalent to the one in standard library

$\mathbb{Q} : \text{Set}$   
 $\mathbb{Q} = \Sigma[ q : \mathbb{Q}_0 ] \text{IsReduced } q$

**Normalisation function:**

1. Calculate a reduced fraction for  $\frac{x}{y}$  with a condition that  $y$  is not zero.

```

calQ :  $\forall (x \ y : \mathbb{N}) \rightarrow y \neq 0 \rightarrow \mathbb{Q}$ 
calQ x y neo with gcd' x y
calQ .(q1  $\mathbb{N}^*$  di) .(q2  $\mathbb{N}^*$  di) neo
  | di , gcd-* q1 q2 c = (numr /suc pred q2) , iscopime
  where
    numr = + q1
    deno = suc (pred q2)

lzero :  $\forall x \ y \rightarrow x \equiv 0 \rightarrow x \mathbb{N}^* y \equiv 0$ 
lzero .0 y refl = refl

q2≠0 :  $q_2 \neq 0$ 
q2≠0 qe = neo (lzero q2 di qe)

invsuc :  $\forall n \rightarrow n \neq 0 \rightarrow n \equiv \text{suc } (\text{pred } n)$ 
invsuc zero nz with nz refl
... | ()
invsuc (suc n) nz = refl

deno≡q2 :  $q_2 \equiv \text{deno}$ 
deno≡q2 = invsuc q2 q2≠0

copnd : Coprime q1 deno
copnd = subst ( $\lambda x \rightarrow \text{Coprime } q_1 \ x$ ) deno≡q2 c

```

```

witProp :  $\forall a b \rightarrow \text{GCD } a b$  1
         $\rightarrow \text{True } (\text{coprime? } a b)$ 
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

```

- _ :  $\mathbb{Q} \rightarrow \mathbb{Q}$ 
- _ ((n /suc d) , isC) = (( $\mathbb{Z}$ - n) /suc d) ,
  subst ( $\lambda x \rightarrow \text{True } (\text{coprime? } x (\text{suc } d))$ )
  (forgetSign n) isC
where
forgetSign :  $\forall x \rightarrow |x| \equiv | \mathbb{Z}\text{- } x |$ 
forgetSign (-suc n) = refl
forgetSign (+ zero) = refl
forgetSign (+ (suc n)) = refl

```

## 3.Normalisation function

```

[ ] :  $\mathbb{Q}_0 \rightarrow \mathbb{Q}$ 
[ (+ n) /suc d ] = calQ n (suc d) ( $\lambda ()$ )
[ (-suc n) /suc d ] = - calQ (suc n) (suc d) ( $\lambda ()$ )

```

Embedding function

$$\begin{aligned} \lceil \_ \rceil : \mathbb{Q} &\rightarrow \mathbb{Q}_0 \\ \lceil \_ \rceil &= \text{proj}_1 \end{aligned}$$

# 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    :  $\forall \{x\} \rightarrow x \approx x$ 
.sym     :  $\forall \{x\ y\} \rightarrow x \approx y \rightarrow y \approx x$ 
.trans   :  $\forall \{x\ y\ z\} \rightarrow x \approx y \rightarrow y \approx z \rightarrow x \approx z$ 
open Setoid public renaming
  (Carrier to |_| ;  $_ \approx _$  to [|_|  $\approx$  |_] ; refl to [|_|refl];
   trans to [|_|trans]; sym to [|_|sym])

```

### Morphisms between Setoids (Functors)

```

infix 5  $_ \Rightarrow _$ 

record  $_ \Rightarrow _$  (A B : Setoid) : Set where
  constructor fn :  $_ \text{resp} _$ 
  field
    fn : | A |  $\rightarrow$  | B |
    .resp : {x y : | A |}  $\rightarrow$ 
      ([ A ] x  $\approx$  y)  $\rightarrow$ 
      [ B ] fn x  $\approx$  fn y
open  $_ \Rightarrow _$  public renaming (fn to [|_|fn ; resp to [|_|resp])

```

### Terminal object

```

● : Setoid
● = record {
  Carrier =  $\top$ ;
   $_ \approx _$  =  $\lambda \_ \_ \rightarrow \top$ ;
  refl = tt;
  sym =  $\lambda \_ \rightarrow \text{tt}$ ;
  trans =  $\lambda \_ \_ \rightarrow \text{tt}$  }

★ : {Δ : Setoid}  $\rightarrow \Delta \Rightarrow \bullet$ 

```

```

★ = 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 _ $\Rightarrow$ _ f

```

### Semantic Terms

```

record Tm {Γ : Con}(A : Ty Γ) : Set where
  constructor tm: _resp: _
  field
    tm    : (x : | Γ |) → | [ A ]fm x |
    .resp : ∀ {x y : | Γ |} →
      (p : [ Γ ] x  $\approx$  y) →
      [ [ A ]fm y ] [ A ]subst p (tm x)  $\approx$  tm y

open Tm public renaming (tm to [_]tm ; resp to [_]resp)

```

### Term substitution

```

_[]m : ∀ {Γ Δ : Con}{A : Ty Δ} → Tm A
      → (f : Γ  $\Rightarrow$  Δ) → Tm (A [ f ]T)
_[]m t f = record
  { tm = [ t ]tm ∘ [ f ]fn
  ; resp = [ t ]resp ∘ [ 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 >

```

$$\text{; resp} = \langle [f]_{\text{resp}}, [t]_{\text{respt}} \rangle$$

$$\}$$

### Projections

$$\text{fst} : \{\Gamma \Delta : \text{Con}\} \{A : \text{Ty } \Delta\} \rightarrow \Gamma \rightrightarrows (\Delta \& A) \rightarrow \Gamma \rightrightarrows \Delta$$

$$\text{fst } f = \text{record}$$

$$\{ \text{fn} = \text{proj}_1 \circ [f]_{\text{fn}}$$

$$\text{; resp} = \text{proj}_1 \circ [f]_{\text{resp}}$$

$$\}$$

$$\text{snd} : \{\Gamma \Delta : \text{Con}\} \{A : \text{Ty } \Delta\} \rightarrow (f : \Gamma \rightrightarrows (\Delta \& A))$$

$$\rightarrow \text{Tm } (A \text{ [fst } \{A = A\} f] \text{T})$$

$$\text{snd } f = \text{record}$$

$$\{ \text{tm} = \text{proj}_2 \circ [f]_{\text{fn}}$$

$$\text{; respt} = \text{proj}_2 \circ [f]_{\text{resp}}$$

$$\}$$

$$\_ \wedge \_ : \{\Gamma \Delta : \text{Con}\} (f : \Gamma \rightrightarrows \Delta) (A : \text{Ty } \Delta)$$

$$\rightarrow \Gamma \& A \text{ [f]T} \rightrightarrows \Delta \& A$$

$$f \wedge A = \text{record}$$

$$\{ \text{fn} = \langle [f]_{\text{fn}} \circ \text{proj}_1, \text{proj}_2 \rangle$$

$$\text{; resp} = \langle [f]_{\text{resp}} \circ \text{proj}_1, \text{proj}_2 \rangle$$

$$\}$$

### $\Pi$ -types (object level)

$$\Pi : \{\Gamma : \text{Setoid}\} (A : \text{Ty } \Gamma) (B : \text{Ty } (\Gamma \& A)) \rightarrow \text{Ty } \Gamma$$

$$\Pi \{\Gamma\} A B = \text{record}$$

$$\{ \text{fm} = \lambda x \rightarrow \text{let } Ax = [A]_{\text{fm}} x \text{ in}$$

$$\text{let } Bx = \lambda a \rightarrow [B]_{\text{fm}} (x, a) \text{ in}$$

```

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))
; resp = 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}
; resp = 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)
    }

```

$\Sigma$ -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

$\llbracket Q \rrbracket_0 : | \Gamma | \rightarrow \text{Setoid}$

$\llbracket Q \rrbracket_0 \gamma = \text{record}$   
 $\{ \text{Carrier} = | [ A ]_{\text{fm}} \gamma |$   
 $; \_ \approx \_ = R \gamma$   
 $; \text{refl} = \text{Rref}$   
 $; \text{sym} = \text{Rsym}$   
 $; \text{trans} = \text{Rtrn}$   
 $\}$

$\llbracket Q \rrbracket : \text{Ty } \Gamma$

$\llbracket Q \rrbracket = \text{record}$   
 $\{ \text{fm} = \llbracket Q \rrbracket_0$   
 $; \text{substT} = [ A ]_{\text{subst}}$   
 $; \text{subst}^* = \lambda p q \rightarrow \text{Rrespt } p \_ \_ q$   
 $; \text{refl}^* = \text{Rrsp } [ A ]_{\text{refl}}^*$   
 $; \text{trans}^* = \lambda a \rightarrow \text{Rrsp } ([ A ]_{\text{trans}}^* \_)$   
 $\}$

$\llbracket [ \_ ] \rrbracket : \text{Tm } A \rightarrow \text{Tm } \llbracket Q \rrbracket$

$\llbracket [ x ] \rrbracket = \text{record}$   
 $\{ \text{tm} = [ x ]_{\text{tm}}$   
 $; \text{respt} = \lambda p \rightarrow \text{Rrsp } ([ x ]_{\text{respt}} p)$   
 $\}$

$\llbracket [ \_ ] \rrbracket' : \text{Tm } (A \Rightarrow \llbracket Q \rrbracket)$

$\llbracket [ \_ ] \rrbracket' = \text{record}$   
 $\{ \text{tm} = \lambda x \rightarrow (\lambda a \rightarrow a) ,$   
 $(\lambda a b p \rightarrow$   
 $\text{Rrsp } ([ [ A ]_{\text{fm}} \_ ]_{\text{trans}} [ A ]_{\text{refl}}^* p))$   
 $; \text{respt} = \lambda p a \rightarrow \text{Rrsp } [ A ]_{\text{tr}}^*$   
 $\}$

.Q-Ax :  $\forall \gamma \ a \ b \rightarrow [ [ A ]_{\text{fm}} \gamma ] \ a \approx b \rightarrow [ [ [Q] ]_{\text{fm}} \_ ] \ a \approx b$   
 Q-Ax  $\gamma \ a \ b = \text{Rrsp}$

Q-elim :  $(B : \text{Ty } \Gamma)(f : \text{Tm } (A \Rightarrow B))$   
 (frespR :  $\forall \gamma \ a \ b \rightarrow (R \ \gamma \ a \ b)$ )  
 $\rightarrow [ [ B ]_{\text{fm}} \gamma ] \text{prj}_1 ([ f ]_{\text{tm}} \gamma) \ a$   
 $\approx \text{prj}_1 ([ f ]_{\text{tm}} \gamma) \ b$   
 $\rightarrow \text{Tm } ([Q] \Rightarrow B)$   
 Q-elim B f frespR = record  
 { tm =  $\lambda \gamma \rightarrow \text{prj}_1 ([ f ]_{\text{tm}} \gamma)$  ,  $(\lambda \ a \ b \ p \rightarrow$   
 $[ [ B ]_{\text{fm}} \_ ]_{\text{trans}} [ B ]_{\text{refl}^*} (\text{frespR } \_ \_ \_ p))$   
 ; respt =  $\lambda \{ \gamma \} \{ \gamma' \} \ p \ a \rightarrow [ f ]_{\text{respt}} \ p \ a$   
 }

substQ :  $(\Gamma \ \& \ A) \Rightarrow (\Gamma \ \& \ [Q])$   
 substQ = record  
 { fn =  $\lambda \{ (x \ , \ a) \rightarrow x \ , \ a \}$   
 ; resp =  $\lambda \{ (p \ , \ q) \rightarrow p \ , \ (R_{\text{rsp}} \ q) \}$   
 }

Q-ind :  $(P : \text{Ty } (\Gamma \ \& \ [Q]))$   
 $\rightarrow (\text{isProp} : \forall \{x \ a\} \ (r \ s : [ [ P ]_{\text{fm}} (x \ , \ a) ] \rightarrow$   
 $[ [ P ]_{\text{fm}} (x \ , \ a) ] \ r \approx s)$   
 $\rightarrow (h : \text{Tm } (\prod A \ (P [ \text{substQ } T])))$   
 $\rightarrow \text{Tm } (\prod [Q] \ P)$   
 Q-ind P isProp h = record  
 { tm =  $\lambda x \rightarrow (\text{prj}_1 ([ h ]_{\text{tm}} x))$  ,  
 $(\lambda \ a \ b \ p \rightarrow \text{isProp } \{x\} \ \{b\} \_ \_)$   
 ; respt =  $[ h ]_{\text{respt}}$

---

 $\}$



# Appendix C

## syntactic weak $\omega$ -groupoids

### C.1 Syntax of $\mathcal{T}_{\infty\text{-groupoid}}$

```
data Con          : Set
data Ty ( $\Gamma$  : Con) : Set
data Tm           : { $\Gamma$  : Con}( $A$  : Ty  $\Gamma$ )  $\rightarrow$  Set
data Var          : { $\Gamma$  : Con}( $A$  : Ty  $\Gamma$ )  $\rightarrow$  Set
data  $\_ \Rightarrow \_$  : Con  $\rightarrow$  Con  $\rightarrow$  Set
data isContr      : Con  $\rightarrow$  Set
```

#### Contexts

```
data Con where
   $\epsilon$       : Con
   $\_ , \_$       : ( $\Gamma$  : Con)( $A$  : Ty  $\Gamma$ )  $\rightarrow$  Con
```

#### Types

```

data Ty  $\Gamma$  where
  *      : Ty  $\Gamma$ 
  _=h_   : {A : Ty  $\Gamma$ } (a b : Tm A)  $\rightarrow$  Ty  $\Gamma$ 

```

### Heterogeneous Equality for Terms

```

data _ $\cong$ _ { $\Gamma$  : Con} {A : Ty  $\Gamma$ } :
  {B : Ty  $\Gamma$ }  $\rightarrow$  Tm A  $\rightarrow$  Tm B  $\rightarrow$  Set where
  refl : (b : Tm A)  $\rightarrow$  b  $\cong$  b

```

```

_ $^{-1}$       :  $\forall$  { $\Gamma$  : Con} {A B : Ty  $\Gamma$ }
  {a : Tm A} {b : Tm B}  $\rightarrow$  a  $\cong$  b  $\rightarrow$  b  $\cong$  a
(refl _)  $^{-1}$  = refl _

```

```

infixr 4 _ $\sim$ _

```

```

_ $\sim$ _ : { $\Gamma$  : Con}
  {A B C : Ty  $\Gamma$ }
  {a : Tm A} {b : Tm B} {c : Tm C}  $\rightarrow$ 
  a  $\cong$  b  $\rightarrow$ 
  b  $\cong$  c
 $\rightarrow$  a  $\cong$  c
_ $\sim$ _ {c = c} (refl .c) (refl .c) = refl c

```

```

_ $\llbracket$ _  $\gg$       : { $\Gamma$  : Con} {A B : Ty  $\Gamma$ } (a : Tm B)
 $\rightarrow$  A  $\equiv$  B  $\rightarrow$  Tm A
a  $\llbracket$  refl  $\gg$  = a

```

```

cohOp      : { $\Gamma$  : Con} {A B : Ty  $\Gamma$ } {a : Tm B} (p : A  $\equiv$  B)
 $\rightarrow$  a  $\llbracket$  p  $\gg$   $\cong$  a

```



$\text{cohOp refl} = \text{refl } \_$

$\text{cohOp-eq} : \{\Gamma : \text{Con}\}\{A B : \text{Ty } \Gamma\}\{a b : \text{Tm } B\}$

$\{p : A \equiv B\} \rightarrow (a \cong b)$

$\rightarrow (a \llbracket p \rrbracket \cong b \llbracket p \rrbracket)$

$\text{cohOp-eq } \{\Gamma\} \{.B\} \{B\} \{a\} \{b\} \{\text{refl}\} r = r$

$\text{cohOp-hom} : \{\Gamma : \text{Con}\}\{A B : \text{Ty } \Gamma\}\{a b : \text{Tm } B\}(p : A \equiv B) \rightarrow$

$(a \llbracket p \rrbracket =_h b \llbracket p \rrbracket) \equiv (a =_h b)$

$\text{cohOp-hom refl} = \text{refl}$

$\text{cong}\cong : \{\Gamma \Delta : \text{Con}\}\{A B : \text{Ty } \Gamma\}\{a : \text{Tm } A\}\{b : \text{Tm } B\}$

$\{D : \text{Ty } \Gamma \rightarrow \text{Ty } \Delta\} \rightarrow (f : \{C : \text{Ty } \Gamma\} \rightarrow \text{Tm } C \rightarrow \text{Tm } (D C)) \rightarrow$

$a \cong b \rightarrow f a \cong f b$

$\text{cong}\cong f (\text{refl } \_) = \text{refl } \_$

## Substitutions

$\_[\_]T : \forall\{\Gamma \Delta\} \rightarrow \text{Ty } \Delta \rightarrow \Gamma \Rightarrow \Delta \rightarrow \text{Ty } \Gamma$

$\_[\_]V : \forall\{\Gamma \Delta A\} \rightarrow \text{Var } A \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \text{Tm } (A [\delta]T)$

$\_[\_]tm : \forall\{\Gamma \Delta A\} \rightarrow \text{Tm } A \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \text{Tm } (A [\delta]T)$

$\_ \odot \_ : \forall\{\Gamma \Delta \Theta\} \rightarrow \Delta \Rightarrow \Theta \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \Gamma \Rightarrow \Theta$

## Contexts morphisms

$\text{data } \_ \Rightarrow \_ \text{ where}$

$\bullet : \forall\{\Gamma\} \rightarrow \Gamma \Rightarrow \epsilon$

$\_,\_ : \forall\{\Gamma \Delta\}(\delta : \Gamma \Rightarrow \Delta)\{A : \text{Ty } \Delta\}(a : \text{Tm } (A [\delta]T))$

$$\rightarrow \Gamma \Rightarrow (\Delta, A)$$

### Weakening

$$\begin{aligned} \_+T\_ &: \forall\{\Gamma\}(A : \text{Ty } \Gamma)(B : \text{Ty } \Gamma) \rightarrow \text{Ty } (\Gamma, B) \\ \_+tm\_ &: \forall\{\Gamma A\}(a : \text{Tm } A)(B : \text{Ty } \Gamma) \rightarrow \text{Tm } (A +T B) \\ \_+S\_ &: \forall\{\Gamma \Delta\}(\delta : \Gamma \Rightarrow \Delta)(B : \text{Ty } \Gamma) \rightarrow (\Gamma, B) \Rightarrow \Delta \end{aligned}$$

$$\begin{aligned} * \quad \_+T B &= * \\ (a =_h b) \_+T B &= a \_+tm B =_h b \_+tm B \end{aligned}$$

$$\begin{aligned} * \quad [\delta]T &= * \\ (a =_h b) [\delta]T &= a [\delta]tm =_h b [\delta]tm \end{aligned}$$

### Variables and terms

$$\begin{aligned} \text{data Var where} \\ v0 &: \forall\{\Gamma\}\{A : \text{Ty } \Gamma\} \rightarrow \text{Var } (A +T A) \\ vS &: \forall\{\Gamma\}\{A B : \text{Ty } \Gamma\}(x : \text{Var } A) \rightarrow \text{Var } (A +T B) \end{aligned}$$

$$\begin{aligned} \text{data Tm where} \\ var &: \forall\{\Gamma\}\{A : \text{Ty } \Gamma\} \rightarrow \text{Var } A \rightarrow \text{Tm } A \\ coh &: \forall\{\Gamma \Delta\} \rightarrow \text{isContr } \Delta \rightarrow (\delta : \Gamma \Rightarrow \Delta) \\ &\rightarrow (A : \text{Ty } \Delta) \rightarrow \text{Tm } (A [\delta]T) \end{aligned}$$

$$\text{cohOpV} : \{\Gamma : \text{Con}\}\{A B : \text{Ty } \Gamma\}\{x : \text{Var } A\}(p : A \equiv B) \rightarrow$$

$$\begin{aligned}
& \text{var } (\text{subst } \text{Var } p \ x) \cong \text{var } x \\
& \text{cohOpV } \{x = x\} \text{ refl} = \text{refl } (\text{var } x) \\
\\
& \text{cohOpVs} : \{\Gamma : \text{Con}\} \{A \ B \ C : \text{Ty } \Gamma\} \{x : \text{Var } A\} (p : A \equiv B) \rightarrow \\
& \quad \text{var } (\text{vS } \{B = C\} (\text{subst } \text{Var } p \ x)) \cong \text{var } (\text{vS } x) \\
& \text{cohOpVs } \{x = x\} \text{ refl} = \text{refl } (\text{var } (\text{vS } x)) \\
\\
& \text{coh-eq} : \{\Gamma \ \Delta : \text{Con}\} \{\text{isc} : \text{isContr } \Delta\} \{\gamma \ \delta : \Gamma \Rightarrow \Delta\} \\
& \quad \{A : \text{Ty } \Delta\} \rightarrow \gamma \equiv \delta \rightarrow \text{coh } \text{isc } \gamma \ A \cong \text{coh } \text{isc } \delta \ A \\
& \text{coh-eq } \text{refl} = \text{refl } \_
\end{aligned}$$

### Contractible contexts

$$\begin{aligned}
& \text{data isContr where} \\
& \quad \text{c*} : \text{isContr } (\epsilon, *) \\
& \quad \text{ext} : \forall \{\Gamma\} \rightarrow \text{isContr } \Gamma \rightarrow \{A : \text{Ty } \Gamma\} (x : \text{Var } A) \\
& \quad \rightarrow \text{isContr } (\Gamma, A, (\text{var } (\text{vS } x) =_h \text{var } v0)) \\
\\
& \text{hom} \equiv : \{\Gamma : \text{Con}\} \{A \ A' : \text{Ty } \Gamma\} \\
& \quad \{a : \text{Tm } A\} \{a' : \text{Tm } A'\} (q : a \cong a') \\
& \quad \{b : \text{Tm } A\} \{b' : \text{Tm } A'\} (r : b \cong b') \\
& \quad \rightarrow (a =_h b) \equiv (a' =_h b') \\
& \text{hom} \equiv \{\Gamma\} \{A'\} \{A'\} \{a'\} \{a'\} (\text{refl } .a') \{b'\} \{b'\} (\text{refl } .b') = \text{refl} \\
\\
& \text{S-eq} : \{\Gamma \ \Delta : \text{Con}\} \{\gamma \ \delta : \Gamma \Rightarrow \Delta\} \{A : \text{Ty } \Delta\} \\
& \quad \{a : \text{Tm } (A \ [\gamma] T)\} \{a' : \text{Tm } (A \ [\delta] T)\} \\
& \quad \rightarrow \gamma \equiv \delta \rightarrow a \cong a' \\
& \quad \rightarrow \_ \equiv \_ \ \{\_ \} \ \{\Gamma \Rightarrow (\Delta, A)\} \ (\gamma, a) \ (\delta, a')
\end{aligned}$$

$$\text{S-eq refl (refl \_)} = \text{refl}$$

### Some lemmas

$$\begin{aligned} [\odot]T & : \forall \{\Gamma \Delta \Theta A\} \{\theta : \Delta \Rightarrow \Theta\} \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow A \ [ \theta \odot \delta ]T \equiv (A \ [ \theta ]T) [ \delta ]T \end{aligned}$$

$$\begin{aligned} [\odot]V & : \forall \{\Gamma \Delta \Theta A\} (x : \text{Var } A) \{\theta : \Delta \Rightarrow \Theta\} \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow x \ [ \theta \odot \delta ]V \cong (x \ [ \theta ]V) [ \delta ]tm \end{aligned}$$

$$\begin{aligned} [\odot]tm & : \forall \{\Gamma \Delta \Theta A\} (a : \text{Tm } A) \{\theta : \Delta \Rightarrow \Theta\} \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow a \ [ \theta \odot \delta ]tm \cong (a \ [ \theta ]tm) [ \delta ]tm \end{aligned}$$

$$\begin{aligned} \odot\text{assoc} & : \forall \{\Gamma \Delta \Theta \Omega\} (\gamma : \Theta \Rightarrow \Omega) \{\theta : \Delta \Rightarrow \Theta\} \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow (\gamma \odot \theta) \odot \delta \equiv \gamma \odot (\theta \odot \delta) \end{aligned}$$

$$\begin{aligned} & \bullet \quad \odot \delta = \bullet \\ (\delta \ , \ a) \odot \delta' & = (\delta \odot \delta') \ , \ a \ [ \delta' ]tm \llbracket [\odot]T \rrbracket \end{aligned}$$

$$\begin{aligned} [+S]T & : \forall \{\Gamma \Delta A B\} \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow A \ [ \delta +S B ]T \equiv (A \ [ \delta ]T) +T B \end{aligned}$$

$$\begin{aligned} [+S]tm & : \forall \{\Gamma \Delta A B\} (a : \text{Tm } A) \{\delta : \Gamma \Rightarrow \Delta\} \\ & \rightarrow a \ [ \delta +S B ]tm \cong (a \ [ \delta ]tm) +tm B \end{aligned}$$

$$\begin{aligned} [+S]S & : \forall \{\Gamma \Delta \Theta B\} \{\delta : \Delta \Rightarrow \Theta\} \{\gamma : \Gamma \Rightarrow \Delta\} \\ & \rightarrow \delta \odot (\gamma +S B) \equiv (\delta \odot \gamma) +S B \end{aligned}$$

$$\begin{aligned} \text{wk-tm+} & : \{\Gamma \Delta : \text{Con}\} \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} (B : \text{Ty } \Gamma) \\ & \rightarrow \text{Tm } (A \ [ \delta ]T +T B) \rightarrow \text{Tm } (A \ [ \delta +S B ]T) \end{aligned}$$

$$\text{wk-tm}+ B \, t = t \llbracket [+S]T \gg$$

$$\bullet \quad +S B = \bullet$$

$$(\delta, a) +S B = (\delta +S B), \text{wk-tm}+ B (a +tm B)$$

$$[+S]T \{A = *\} = \text{refl}$$

$$[+S]T \{A = a =h b\} = \text{hom} \equiv ([+S]tm \, a) ([+S]tm \, b)$$

$$+T[, ]T : \forall \{\Gamma \Delta A B\} \{\delta : \Gamma \Rightarrow \Delta\} \{b : Tm (B [ \delta ]T)\}$$

$$\rightarrow (A +T B) [ \delta, b ]T \equiv A [ \delta ]T$$

$$+tm[, ]tm : \forall \{\Gamma \Delta A B\} \{\delta : \Gamma \Rightarrow \Delta\} \{c : Tm (B [ \delta ]T)\}$$

$$\rightarrow (a : Tm A)$$

$$\rightarrow (a +tm B) [ \delta, c ]tm \cong a [ \delta ]tm$$

$$(\text{var } x) \quad +tm B = \text{var } (vS \, x)$$

$$(\text{coh } c\Delta \, \delta A) +tm B = \text{coh } c\Delta (\delta +S B) A \llbracket \text{sym } [+S]T \gg$$

$$\text{cong+tm} : \{\Gamma : \text{Con}\} \{A B C : Ty \, \Gamma\} \{a : Tm A\} \{b : Tm B\} \rightarrow$$

$$a \cong b$$

$$\rightarrow a +tm C \cong b +tm C$$

$$\text{cong+tm } (\text{refl } \_) = \text{refl } \_$$

$$\text{cong+tm2} : \{\Gamma : \text{Con}\} \{A B C : Ty \, \Gamma\}$$

$$\{a : Tm B\} (p : A \equiv B)$$

$$\rightarrow a +tm C \cong a \llbracket p \gg +tm C$$

$$\text{cong+tm2 } \text{refl} = \text{refl } \_$$

$$\text{wk-T} : \{\Delta : \text{Con}\}$$

$$\{A B C : Ty \, \Delta\}$$

$$\rightarrow A \equiv B \rightarrow A +T C \equiv B +T C$$

$$\text{wk-T } \text{refl} = \text{refl}$$

$$\text{wk-tm} : \{\Gamma \Delta : \text{Con}\}$$

$$\begin{aligned}
& \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Delta\} \{b : \text{Tm } (B \text{ [ } \delta \text{ ]T})\} \\
& \rightarrow \text{Tm } (A \text{ [ } \delta \text{ ]T}) \rightarrow \text{Tm } ((A +^{\text{T}} B) \text{ [ } \delta , b \text{ ]T}) \\
& \text{wk-tm } t = t \llbracket +^{\text{T}}[\cdot, \cdot] \text{T} \rrbracket
\end{aligned}$$

$$\begin{aligned}
& \text{v0 [ } \delta , a \text{ ]V} = \text{wk-tm } a \\
& \text{vS } x \text{ [ } \delta , a \text{ ]V} = \text{wk-tm } (x \text{ [ } \delta \text{ ]V})
\end{aligned}$$

$$\begin{aligned}
& \text{wk-coh} : \{\Gamma \Delta : \text{Con}\} \\
& \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Delta\} \{b : \text{Tm } (B \text{ [ } \delta \text{ ]T})\} \\
& \{t : \text{Tm } (A \text{ [ } \delta \text{ ]T})\} \\
& \rightarrow \text{wk-tm } \{B = B\} \{b = b\} t \cong t \\
& \text{wk-coh} = \text{cohOp } +^{\text{T}}[\cdot, \cdot] \text{T}
\end{aligned}$$

$$\begin{aligned}
& \text{wk-coh+} : \{\Gamma \Delta : \text{Con}\} \\
& \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Gamma\} \\
& \{x : \text{Tm } (A \text{ [ } \delta \text{ ]T } +^{\text{T}} B)\} \\
& \rightarrow \text{wk-tm+ } B \ x \cong x \\
& \text{wk-coh+} = \text{cohOp } [+^{\text{S}}] \text{T}
\end{aligned}$$

$$\begin{aligned}
& \text{wk-hom} : \{\Gamma \Delta : \text{Con}\} \\
& \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Delta\} \{b : \text{Tm } (B \text{ [ } \delta \text{ ]T})\} \\
& \{x \ y : \text{Tm } (A \text{ [ } \delta \text{ ]T})\} \\
& \rightarrow (\text{wk-tm } \{B = B\} \{b = b\} x \text{ =h wk-tm } \\
& \quad \{B = B\} \{b = b\} y) \equiv (x \text{ =h } y) \\
& \text{wk-hom} = \text{hom} \equiv \text{wk-coh } \text{wk-coh}
\end{aligned}$$

$$\begin{aligned}
& \text{wk-hom+} : \{\Gamma \Delta : \text{Con}\} \\
& \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Gamma\}
\end{aligned}$$

$$\begin{aligned}
& \{x\ y : \mathsf{Tm}\ (A\ [\delta]\mathsf{T} + \mathsf{T}\ B)\} \\
& \rightarrow (\mathsf{wk-tm} + B\ x\ \mathsf{=h}\ \mathsf{wk-tm} + B\ y) \equiv (x\ \mathsf{=h}\ y) \\
& \mathsf{wk-hom} + = \mathsf{hom} \equiv \mathsf{wk-coh} + \mathsf{wk-coh} +
\end{aligned}$$

$$\begin{aligned}
& \mathsf{wk-}\odot : \{\Gamma\ \Delta\ \Theta : \mathsf{Con}\} \\
& \{\theta : \Delta \Rightarrow \Theta\}\{\delta : \Gamma \Rightarrow \Delta\}\{A : \mathsf{Ty}\ \Theta\} \\
& \rightarrow \mathsf{Tm}\ ((A\ [\theta]\mathsf{T})[\delta]\mathsf{T}) \rightarrow \mathsf{Tm}\ (A\ [\theta\ \odot\ \delta]\mathsf{T}) \\
& \mathsf{wk-}\odot\ t = t\ \llbracket [\odot]\mathsf{T} \rrbracket
\end{aligned}$$

$$\begin{aligned}
& [+S]S\ \{\delta = \bullet\} = \mathsf{refl} \\
& [+S]S\ \{\delta = \delta\ ,\ a\} = \mathsf{S-eq}\ [+S]S\ (\mathsf{cohOp}\ [\odot]\mathsf{T} \sim \\
& \quad ([+S]\mathsf{tm}\ a \sim \mathsf{cong+tm2}\ [\odot]\mathsf{T}) \sim \mathsf{wk-coh} +^{-1})
\end{aligned}$$

$$\begin{aligned}
& \mathsf{wk+S+T} : \forall\{\Gamma\ \Delta : \mathsf{Con}\}\{A : \mathsf{Ty}\ \Gamma\}\{B : \mathsf{Ty}\ \Delta\} \\
& \quad \{\Upsilon\}\{C\} \rightarrow \\
& \quad A\ [\Upsilon]\mathsf{T} \equiv C \\
& \rightarrow A\ [\Upsilon\ +S\ B]\mathsf{T} \equiv C\ +\mathsf{T}\ B \\
& \mathsf{wk+S+T}\ \mathsf{eq} = \mathsf{trans}\ [+S]\mathsf{T}\ (\mathsf{wk-T}\ \mathsf{eq})
\end{aligned}$$

$$\begin{aligned}
& \mathsf{wk+S+tm} : \{\Gamma\ \Delta : \mathsf{Con}\}\{A : \mathsf{Ty}\ \Gamma\}\{B : \mathsf{Ty}\ \Delta\} \\
& \quad (a : \mathsf{Tm}\ A)\{C : \mathsf{Ty}\ \Delta\}\{\Upsilon : \Delta \Rightarrow \Gamma\}\{c : \mathsf{Tm}\ C\} \rightarrow \\
& \quad a\ [\Upsilon]\mathsf{tm} \cong c \\
& \rightarrow a\ [\Upsilon\ +S\ B]\mathsf{tm} \cong c\ +\mathsf{tm}\ B \\
& \mathsf{wk+S+tm}\ \_ \mathsf{eq} = [+S]\mathsf{tm}\ \_ \sim \mathsf{cong+tm}\ \mathsf{eq}
\end{aligned}$$

$$\begin{aligned}
& \mathsf{wk+S+S} : \forall\{\Gamma\ \Delta\ \Delta_1 : \mathsf{Con}\}\{\delta : \Delta \Rightarrow \Delta_1\}\{\Upsilon : \Gamma \Rightarrow \Delta\} \\
& \quad \{\omega : \Gamma \Rightarrow \Delta_1\}\{B : \mathsf{Ty}\ \Gamma\} \\
& \rightarrow \delta\ \odot\ \Upsilon \equiv \omega \\
& \rightarrow \delta\ \odot\ (\Upsilon\ +S\ B) \equiv \omega\ +S\ B \\
& \mathsf{wk+S+S}\ \mathsf{eq} = \mathsf{trans}\ [+S]S\ (\mathsf{cong}\ (\lambda\ x \rightarrow x\ +S\ \_) \mathsf{eq})
\end{aligned}$$

$$[\odot]T \{A = *\} = \text{refl}$$

$$[\odot]T \{A = \_ = h \_ \{A\} \ a \ b\} = \text{hom} \equiv ([\odot]tm \ \_) ([\odot]tm \ \_)$$

$$+T[, ]T \{A = *\} = \text{refl}$$

$$+T[, ]T \{A = \_ = h \_ \{A\} \ a \ b\} = \text{hom} \equiv (+tm[, ]tm \ \_) (+tm[, ]tm \ \_)$$

$$\text{var } x \quad [\delta]tm = x \ [\delta]V$$

$$\text{coh } c\Delta \ \gamma \ A \quad [\delta]tm = \text{coh } c\Delta \ (\gamma \odot \delta) \ A \ll \text{sym } [\odot]T \gg$$

$$\text{congT} : \forall \{ \Gamma \ \Delta : \text{Con} \} \{ A \ B : \text{Ty } \Delta \} \{ \gamma : \Gamma \Rightarrow \Delta \} \rightarrow A \equiv B \rightarrow A \ [\gamma]T \equiv B \ [\gamma]T$$

$$\text{congT refl} = \text{refl}$$

$$\text{congT2} : \forall \{ \Gamma \ \Delta \} \rightarrow \{ \delta \ \gamma : \Delta \Rightarrow \Gamma \} \{ A : \text{Ty } \Gamma \} \rightarrow \delta \equiv \gamma \rightarrow A \ [\delta]T \equiv A \ [\gamma]T$$

$$\text{congT2 refl} = \text{refl}$$

$$\text{congV} : \{ \Gamma \ \Delta : \text{Con} \} \{ A \ B : \text{Ty } \Delta \} \{ a : \text{Var } A \} \{ b : \text{Var } B \} \rightarrow$$

$$\text{var } a \cong \text{var } b \rightarrow$$

$$\{ \delta : \Gamma \Rightarrow \Delta \}$$

$$\rightarrow a \ [\delta]V \cong b \ [\delta]V$$

$$\text{congV } \{ \Gamma \} \{ \Delta \} \{ .B \} \{ B \} \{ .b \} \{ b \} \ (\text{refl } .(\text{var } b)) = \text{refl } \_$$

$$\text{congtm} : \{ \Gamma \ \Delta : \text{Con} \} \{ A \ B : \text{Ty } \Gamma \} \{ a : \text{Tm } A \} \{ b : \text{Tm } B \}$$

$$(p : a \cong b) \rightarrow$$

$$\{ \delta : \Delta \Rightarrow \Gamma \}$$

$$\rightarrow a \ [\delta]tm \cong b \ [\delta]tm$$

$$\text{congtm } (\text{refl } \_) = \text{refl } \_$$

$$\text{congtm2} : \{ \Gamma \ \Delta : \text{Con} \} \{ A : \text{Ty } \Gamma \} \{ a : \text{Tm } A \}$$

$$\{ \delta \ \gamma : \Delta \Rightarrow \Gamma \} \rightarrow$$



$$\begin{aligned}
& (p : \delta \equiv \gamma) \\
& \rightarrow a \text{ [ } \delta \text{ ]tm} \cong a \text{ [ } \gamma \text{ ]tm} \\
& \text{congtm2 refl} = \text{refl } \_
\end{aligned}$$

$$\begin{aligned}
& \odot \text{assoc} \bullet = \text{refl} \\
& \odot \text{assoc} (\_, \_ \gamma \{A\} a) = \text{S-eq} (\odot \text{assoc } \gamma) \\
& \quad (\text{cohOp } [\odot]T \\
& \quad \sim (\text{congtm} (\text{cohOp } [\odot]T) \\
& \quad \sim ((\text{cohOp } [\odot]T \\
& \quad \sim [\odot]tm a)^{-1}))
\end{aligned}$$

$$\begin{aligned}
& [\odot]v (v0 \{ \Gamma_1 \} \{A\}) \{ \theta , a \} = \text{wk-coh} \sim \text{cohOp} \\
& \quad [\odot]T \sim \text{congtm} (\text{cohOp} + T[.,]T^{-1}) \\
& [\odot]v (vS \{ \Gamma_1 \} \{A\} \{B\} x) \{ \theta , a \} = \\
& \quad \text{wk-coh} \sim ([\odot]v x \sim (\text{congtm} (\text{cohOp} + T[.,]T)^{-1}))
\end{aligned}$$

$$\begin{aligned}
& [\odot]tm (\text{var } x) = [\odot]v x \\
& [\odot]tm (\text{coh } c \gamma A) = \text{cohOp} (\text{sym } [\odot]T) \sim (\text{coh-eq} (\text{sym} (\odot \text{assoc } \gamma)) \\
& \quad \sim \text{cohOp} (\text{sym } [\odot]T)^{-1}) \sim \text{congtm} (\text{cohOp} (\text{sym } [\odot]T)^{-1})
\end{aligned}$$

$$\begin{aligned}
& \odot \text{wk} : \forall \{ \Gamma \Delta \Delta_1 \} \{ B : \text{Ty } \Delta \} (\gamma : \Delta \Rightarrow \Delta_1) \{ \delta : \Gamma \Rightarrow \Delta \} \\
& \quad \{ c : \text{Tm} (B \text{ [ } \delta \text{ ]T}) \} \rightarrow (\gamma +S B) \odot (\delta , c) \equiv \gamma \odot \delta
\end{aligned}$$

$$\begin{aligned}
& \odot \text{wk} \bullet = \text{refl} \\
& \odot \text{wk} (\_, \_ \gamma \{A\} a) = \text{S-eq} (\odot \text{wk } \gamma) (\text{cohOp } [\odot]T \sim \\
& \quad (\text{congtm} (\text{cohOp } [+S]T) \sim +tm[.,]tm a) \sim \text{cohOp } [\odot]T^{-1})
\end{aligned}$$

$$\begin{aligned}
& +tm[.,]tm (\text{var } x) = \text{cohOp} + T[.,]T \\
& +tm[.,]tm (\text{coh } x \gamma A) = \text{congtm} (\text{cohOp} (\text{sym } [+S]T)) \sim \\
& \quad \text{cohOp} (\text{sym } [\odot]T) \sim \text{coh-eq} (\odot \text{wk } \gamma) \sim \text{cohOp} (\text{sym } [\odot]T)^{-1}
\end{aligned}$$

$$\begin{aligned}
[+S]V &: \{\Gamma \Delta : \text{Con}\} \{A : \text{Ty } \Delta\} \\
& (x : \text{Var } A) \{\delta : \Gamma \Rightarrow \Delta\} \\
& \{B : \text{Ty } \Gamma\} \\
& \rightarrow x \text{ [ } \delta \text{ +S B ]} V \cong (x \text{ [ } \delta \text{ ]} V) \text{ +tm B} \\
[+S]V \text{ v0 } \{ \_, \_ \delta \{A\} a \} &= \text{wk-coh} \sim \text{wk-coh+} \sim \text{cong+tm2 +T[,]T} \\
[+S]V (\text{vS } x) \{ \delta , a \} &= \text{wk-coh} \sim [+S]V x \sim \text{cong+tm2 +T[,]T}
\end{aligned}$$

$$\begin{aligned}
[+S]\text{tm } (\text{var } x) &= [+S]V x \\
[+S]\text{tm } (\text{coh } x \delta A) &= \text{cohOp } (\text{sym } [\odot]T) \sim \text{coh-eq } [+S]S \sim \\
&\text{cohOp } (\text{sym } [+S]T)^{-1} \sim \text{cong+tm2 } (\text{sym } [\odot]T)
\end{aligned}$$

### Some simple contexts

$$\begin{aligned}
x.^* &: \text{Con} \\
x.^* &= \varepsilon , ^* \\
x.^*, y.^*, \alpha: x=y &: \text{Con} \\
x.^*, y.^*, \alpha: x=y &= x.^* , ^* , (\text{var } (\text{vS } v0) =_h \text{var } v0) \\
vX &: \text{Tm } \{x.^*, y.^*, \alpha: x=y\} ^* \\
vX &= \text{var } (\text{vS } (\text{vS } v0)) \\
vY &: \text{Tm } \{x.^*, y.^*, \alpha: x=y\} ^* \\
vY &= \text{var } (\text{vS } v0) \\
v\alpha &: \text{Tm } \{x.^*, y.^*, \alpha: x=y\} (vX =_h vY) \\
v\alpha &= \text{var } v0 \\
x.^*, y.^*, \alpha: x=y, z.^*, \beta: y=z &: \text{Con}
\end{aligned}$$

$$x:*, y:*, \alpha:x=y, z:*, \beta:y=z = x:*, y:*, \alpha:x=y, *, *, \\ (\text{var } (vS \ (vS \ v0)) \Rightarrow_h \text{var } v0)$$

$$vZ : \text{Tm } \{x:*, y:*, \alpha:x=y, z:*, \beta:y=z\} \ * \\ vZ = \text{var } (vS \ v0)$$

$$v\beta : \text{Tm } \{x:*, y:*, \alpha:x=y, z:*, \beta:y=z\} \ (vY \ +_{tm} \_ \ +_{tm} \_ \Rightarrow_h \ vZ) \\ v\beta = \text{var } v0$$

## C.2 Some Important Derivable Constructions

### Identity morphism

$$\text{IdS} : \forall \{\Gamma\} \rightarrow \Gamma \Rightarrow \Gamma \\ \text{IC-T} : \forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow A \ [ \text{IdS} ] \text{T} \equiv A \\ \text{IC-v} : \forall \{\Gamma : \text{Con}\} \{A : \text{Ty } \Gamma\} (x : \text{Var } A) \rightarrow x \ [ \text{IdS} ] \text{V} \cong \text{var } x \\ \text{IC-S} : \forall \{\Gamma \Delta : \text{Con}\} (\delta : \Gamma \Rightarrow \Delta) \rightarrow \delta \odot \text{IdS} \equiv \delta \\ \text{IC-tm} : \forall \{\Gamma : \text{Con}\} \{A : \text{Ty } \Gamma\} (a : \text{Tm } A) \rightarrow a \ [ \text{IdS} ] \text{tm} \cong a$$

$$\text{Coh-Contr} : \forall \{\Gamma\} \{A : \text{Ty } \Gamma\} \rightarrow \text{isContr } \Gamma \rightarrow \text{Tm } A \\ \text{Coh-Contr isC} = \text{coh isC IdS } \_ \llbracket \text{sym IC-T} \rrbracket$$

$$\text{IdS } \{\epsilon\} = \bullet \\ \text{IdS } \{\Gamma, A\} = \text{IdS } +S \_, \text{var } v0 \llbracket \text{wk+S+T IC-T} \rrbracket$$

$$\text{IC-T } \{\Gamma\} \{*\} = \text{refl} \\ \text{IC-T } \{\Gamma\} \{a \Rightarrow_h b\} = \text{hom} \equiv (\text{IC-tm } a) (\text{IC-tm } b)$$

$$\text{IC-v } \{.(\Gamma, A)\} \{.(A +T A)\} (v0 \{\Gamma\} \{A\}) = \text{wk-coh} \sim \text{cohOp } (\text{wk+S+T IC-T}) \\ \text{IC-v } \{.(\Gamma, B)\} \{.(A +T B)\} (vS \{\Gamma\} \{A\} \{B\} x) = \text{wk-coh} \sim \\ \text{wk+S+tm } (\text{var } x) (\text{IC-v } \_)$$

$$\text{IC-S } \bullet = \text{refl}$$

$$\text{IC-S } (\delta, a) = \text{S-eq } (\text{IC-S } \delta) (\text{cohOp } [\odot]T \sim \text{IC-tm } a)$$

$$\text{IC-tm } (\text{var } x) = \text{IC-v } x$$

$$\text{IC-tm } (\text{coh } x \delta A) = \text{cohOp } (\text{sym } [\odot]T) \sim \text{coh-eq } (\text{IC-S } \delta)$$

### Some auxiliary functions

$$\text{1-1S-same} : \{\Gamma : \text{Con}\} \{A B : \text{Ty } \Gamma\} \rightarrow$$

$$B \equiv A \rightarrow (\Gamma, A) \Rightarrow (\Gamma, B)$$

$$\text{1-1S-same eq} = \text{pr1}, \text{pr2 } \llbracket \text{congT eq} \rrbracket$$

$$\text{1-1S-same-T} : \{\Gamma : \text{Con}\} \{A B : \text{Ty } \Gamma\} \rightarrow$$

$$(\text{eq} : B \equiv A) \rightarrow (A +T B) \llbracket \text{1-1S-same eq} \rrbracket T \equiv A +T A$$

$$\text{1-1S-same-T eq} = \text{trans } +T[\cdot]T (\text{trans } [+S]T (\text{wk-T IC-T}))$$

$$\text{1-1S-same-tm} : \forall \{\Gamma : \text{Con}\} \{A : \text{Ty } \Gamma\} \{B : \text{Ty } \Gamma\} \rightarrow$$

$$(\text{eq} : B \equiv A) (a : \text{Tm } A) \rightarrow$$

$$(a +tm B) \llbracket \text{1-1S-same eq} \rrbracket tm \cong (a +tm A)$$

$$\text{1-1S-same-tm eq } a = +tm[\cdot]tm a \sim [+S]tm a \sim \text{cong+tm } (\text{IC-tm } a)$$

$$\text{1-1S-same-v0} : \forall \{\Gamma : \text{Con}\} \{A B : \text{Ty } \Gamma\} \rightarrow$$

$$(\text{eq} : B \equiv A) \rightarrow \text{var v0} \llbracket \text{1-1S-same eq} \rrbracket tm \cong \text{var v0}$$

$$\text{1-1S-same-v0 eq} = \text{wk-coh} \sim \text{cohOp } (\text{congT eq}) \sim \text{pr2-v0}$$

$$\_++\_ : \text{Con} \rightarrow \text{Con} \rightarrow \text{Con}$$

$$\text{cor} : \{\Gamma : \text{Con}\} (\Delta : \text{Con}) \rightarrow (\Gamma ++ \Delta) \Rightarrow \Delta$$

$$\text{repeat-p1} : \{\Gamma : \text{Con}\} (\Delta : \text{Con}) \rightarrow (\Gamma ++ \Delta) \Rightarrow \Gamma$$

$$\begin{aligned}
& \Gamma \text{ ++ } \varepsilon = \Gamma \\
& \Gamma \text{ ++ } (\Delta , A) = \Gamma \text{ ++ } \Delta , A \text{ [ cor } \Delta \text{ ]T} \\
\\
& \text{repeat-p1 } \varepsilon = \text{ldS} \\
& \text{repeat-p1 } (\Delta , A) = \text{repeat-p1 } \Delta \odot \text{pr1} \\
\\
& \text{cor } \varepsilon = \bullet \\
& \text{cor } (\Delta , A) = (\text{cor } \Delta \text{ ++S } \_ ) , \text{var v0 [ [+S]T } \gg \\
\\
& \_ \text{ ++S } \_ : \forall \{ \Gamma \Delta \Theta \} \rightarrow \Gamma \Rightarrow \Delta \rightarrow \Gamma \Rightarrow \Theta \rightarrow \Gamma \Rightarrow (\Delta \text{ ++ } \Theta) \\
& \text{cor-inv} : \forall \{ \Gamma \Delta \Theta \} \rightarrow \{ \gamma : \Gamma \Rightarrow \Delta \} (\delta : \Gamma \Rightarrow \Theta) \rightarrow \text{cor } \Theta \odot (\gamma \text{ ++S } \delta) \equiv \delta \\
\\
& \gamma \text{ ++S } \bullet = \gamma \\
& \gamma \text{ ++S } (\delta , a) = \gamma \text{ ++S } \delta , a \text{ [ trans (sym [}\odot\text{]T) (congT2 (cor-inv } \_)) \gg} \\
\\
& \text{cor-inv } \bullet = \text{refl} \\
& \text{cor-inv } (\delta , a) = \text{S-eq (trans (}\odot\text{wk } \_ ) (\text{cor-inv } \delta)) \\
& \quad (\text{cohOp [}\odot\text{]T} \sim \text{congtm (cohOp [+S]T)} \\
& \quad \sim \text{cohOp ++T[,]T} \\
& \quad \sim \text{cohOp (trans (sym [}\odot\text{]T) (congT2 (cor-inv } \_)))) \\
\\
& \text{id-S++} : \{ \Gamma : \text{Con} \} (\Delta \Theta : \text{Con}) \rightarrow (\Delta \Rightarrow \Theta) \rightarrow (\Gamma \text{ ++ } \Delta) \Rightarrow (\Gamma \text{ ++ } \Theta) \\
& \text{id-S++ } \Delta \Theta \gamma = \text{repeat-p1 } \Delta \text{ ++S } (\gamma \odot \text{cor } \_)
\end{aligned}$$

### C.2.1 Suspension and Replacement

#### One-step suspension

$$\begin{aligned}
& \Sigma C : \text{Con} \rightarrow \text{Con} \\
& \Sigma T : \forall \{ \Gamma \} \rightarrow \text{Ty } \Gamma \rightarrow \text{Ty } (\Sigma C \Gamma) \\
\\
& \Sigma C \varepsilon = \varepsilon , * , *
\end{aligned}$$

$$\Sigma C (\Gamma, A) = \Sigma C \Gamma, \Sigma T A$$

$$\Sigma v : \forall \{\Gamma\} \{A : Ty \Gamma\} \rightarrow Var A \rightarrow Var (\Sigma T A)$$

$$\Sigma tm : \forall \{\Gamma\} \{A : Ty \Gamma\} \rightarrow Tm A \rightarrow Tm (\Sigma T A)$$

$$\Sigma s : \forall \{\Gamma \Delta\} \rightarrow \Gamma \Rightarrow \Delta \rightarrow \Sigma C \Gamma \Rightarrow \Sigma C \Delta$$

$$*' : \{\Gamma : Con\} \rightarrow Ty (\Sigma C \Gamma)$$

$$*' \{\epsilon\} = var (vS v0) =_h var v0$$

$$*' \{\Gamma, A\} = '*' \{\Gamma\} + T \_$$

$$\Sigma T \{\Gamma\} * = '*' \{\Gamma\}$$

$$\Sigma T (a =_h b) = \Sigma tm a =_h \Sigma tm b$$

$$\Sigma s\bullet : (\Gamma : Con) \rightarrow \Sigma C \Gamma \Rightarrow \Sigma C \epsilon$$

$$\Sigma s\bullet \epsilon = IdS$$

$$\Sigma s\bullet (\Gamma, A) = \Sigma s\bullet \Gamma + S \_$$

$$\Sigma C\text{-Contr} : \forall \Delta \rightarrow isContr \Delta \rightarrow isContr (\Sigma C \Delta)$$

$$\Sigma T[+T] : \forall \{\Gamma\} (A B : Ty \Gamma)$$

$$\rightarrow \Sigma T (A + T B) \equiv \Sigma T A + T \Sigma T B$$

$$\Sigma tm[+tm] : \forall \{\Gamma A\} (a : Tm A) (B : Ty \Gamma)$$

$$\rightarrow \Sigma tm (a + tm B) \cong \Sigma tm a + tm \Sigma T B$$

$$\Sigma T[\Sigma s]T : \forall \{\Gamma \Delta\} (A : Ty \Delta) (\delta : \Gamma \Rightarrow \Delta)$$

$$\rightarrow (\Sigma T A) [\Sigma s \delta] T \equiv \Sigma T (A [\delta] T)$$

$$\Sigma T[+T] * B = refl$$

$$\Sigma T[+T] (\_ =_h \_ \{A\} a b) B = hom \equiv (\Sigma tm[+tm] a B) (\Sigma tm[+tm] b B)$$

$$\Sigma v \{.( \Gamma, A) \} \{.(A + T A) \} (v0 \{ \Gamma \} \{A\}) = subst Var (sym (\Sigma T[+T] A A)) v0$$

$$\Sigma v \{.( \Gamma, B) \} \{.(A + T B) \} (vS \{ \Gamma \} \{A\} \{B\} x) = subst Var (sym (\Sigma T[+T] \{ \_ \} A B)) (vS (\Sigma v x))$$

$$\begin{aligned}
\Sigma_{\text{tm}} (\text{var } x) &= \text{var } (\Sigma_{\text{v}} x) \\
\Sigma_{\text{tm}} (\text{coh } x \delta A) &= \text{coh } (\Sigma_{\text{C-Contr}} \_ x) (\Sigma_{\text{s}} \delta) (\Sigma_{\text{T}} A) \llbracket \text{sym } (\Sigma_{\text{T}}[\Sigma_{\text{s}}]_{\text{T}} A \delta) \rrbracket \\
\\
\Sigma_{\text{tm-p1}} : \{\Gamma : \text{Con}\} (A : \text{Ty } \Gamma) \rightarrow \Sigma_{\text{tm}} \{\Gamma, A\} (\text{var } v0) \cong \text{var } v0 \\
\Sigma_{\text{tm-p1}} A &= \text{cohOpV } (\text{sym } (\Sigma_{\text{T}}[+_{\text{T}}] A A)) \\
\\
\Sigma_{\text{tm-p2}} : \{\Gamma : \text{Con}\} (A B : \text{Ty } \Gamma) (x : \text{Var } A) \rightarrow \text{var } (\Sigma_{\text{v}} (vS \{B = B\} x)) \cong \\
&\quad \text{var } (vS (\Sigma_{\text{v}} x)) \\
\Sigma_{\text{tm-p2}} \{\Gamma\} A B x &= \text{cohOpV } (\text{sym } (\Sigma_{\text{T}}[+_{\text{T}}] A B)) \\
\\
\Sigma_{\text{tm-p2-sp}} : \{\Gamma : \text{Con}\} (A : \text{Ty } \Gamma) (B : \text{Ty } (\Gamma, A)) \rightarrow \Sigma_{\text{tm}} \{\Gamma, A, B\} \\
&\quad (\text{var } (vS v0)) \cong (\text{var } v0) +_{\text{tm}} \_ \\
\Sigma_{\text{tm-p2-sp}} A B &= \Sigma_{\text{tm-p2}} (A +_{\text{T}} A) B v0 \sim \text{cong} +_{\text{tm}} (\Sigma_{\text{tm-p1}} A) \\
\\
\Sigma_{\text{s}} \{\Gamma\} \{\Delta, A\} (\Upsilon, a) &= (\Sigma_{\text{s}} \Upsilon, \Sigma_{\text{tm}} a \llbracket \Sigma_{\text{T}}[\Sigma_{\text{s}}]_{\text{T}} A \Upsilon \rrbracket) \\
\Sigma_{\text{s}} \{\Gamma\} \bullet &= \Sigma_{\text{s}} \bullet \Gamma \\
\\
\text{cong} \Sigma_{\text{tm}} : \{\Gamma : \text{Con}\} \{A B : \text{Ty } \Gamma\} \{a : \text{Tm } A\} \{b : \text{Tm } B\} \rightarrow a \cong b \rightarrow \\
&\quad \Sigma_{\text{tm}} a \cong \Sigma_{\text{tm}} b \\
\text{cong} \Sigma_{\text{tm}} (\text{refl } \_) &= \text{refl } \_ \\
\\
\text{cohOp} \Sigma_{\text{tm}} : \forall \{\Delta : \text{Con}\} \{A B : \text{Ty } \Delta\} (t : \text{Tm } B) (p : A \equiv B) \rightarrow \\
&\quad \Sigma_{\text{tm}} (t \llbracket p \rrbracket) \cong \Sigma_{\text{tm}} t \\
\text{cohOp} \Sigma_{\text{tm}} t p &= \text{cong} \Sigma_{\text{tm}} (\text{cohOp } p) \\
\\
\Sigma_{\text{s}} \odot : \forall \{\Delta \Delta_1 \Gamma\} (\delta : \Delta \Rightarrow \Delta_1) (\Upsilon : \Gamma \Rightarrow \Delta) \rightarrow \Sigma_{\text{s}} (\delta \odot \Upsilon) \equiv \Sigma_{\text{s}} \delta \odot \Sigma_{\text{s}} \Upsilon \\
\\
\Sigma_{\text{v}}[\Sigma_{\text{s}}]_{\text{v}} : \forall \{\Gamma \Delta : \text{Con}\} \{A : \text{Ty } \Delta\} (x : \text{Var } A) (\delta : \Gamma \Rightarrow \Delta) \rightarrow \\
&\quad \Sigma_{\text{v}} x \llbracket \Sigma_{\text{s}} \delta \rrbracket_{\text{V}} \cong \Sigma_{\text{tm}} (x \llbracket \delta \rrbracket_{\text{V}}) \\
\Sigma_{\text{v}}[\Sigma_{\text{s}}]_{\text{v}} (v0 \{\Gamma\} \{A\}) (\delta, a) &= \text{cong} \text{tm} (\Sigma_{\text{tm-p1}} A) \sim \text{wk-coh} \sim \\
&\quad \text{cohOp } (\Sigma_{\text{T}}[\Sigma_{\text{s}}]_{\text{T}} A \delta) \sim \text{cohOp} \Sigma_{\text{tm}} a +_{\text{T}} [\_,]_{\text{T}}^{-1} \\
\Sigma_{\text{v}}[\Sigma_{\text{s}}]_{\text{v}} (vS \{\Gamma\} \{A\} \{B\} x) (\delta, a) &= \text{cong} \text{tm} (\Sigma_{\text{tm-p2}} A B x) \sim \\
&\quad +_{\text{tm}} [\_,]_{\text{tm}} (\Sigma_{\text{tm}} (\text{var } x)) \sim
\end{aligned}$$

$$\Sigma v[\Sigma s]v \times \delta \sim \text{cohOp}\Sigma\text{tm} (x [\delta]V) + T[,]T^{-1}$$

$$\Sigma\text{tm}[\Sigma s]\text{tm} : \forall \{\Gamma \Delta : \text{Con}\}\{A : \text{Ty } \Delta\}(a : \text{Tm } A)(\delta : \Gamma \Rightarrow \Delta) \rightarrow \\ (\Sigma\text{tm } a) [\Sigma s \delta] \text{tm} \cong \Sigma\text{tm} (a [\delta] \text{tm})$$

$$\Sigma\text{tm}[\Sigma s]\text{tm} (\text{var } x) \delta = \Sigma v[\Sigma s]v \times \delta$$

$$\Sigma\text{tm}[\Sigma s]\text{tm} \{\Gamma\} \{\Delta\} (\text{coh } \{\Delta = \Delta_1\} \times \delta A) \delta_1 = \text{congtm} (\text{cohOp} \\ (\text{sym} (\Sigma T[\Sigma s]T A \delta))) \\ \sim \text{cohOp} (\text{sym} [\odot]T) \\ \sim \text{coh-eq} (\text{sym} (\Sigma s \odot \delta \delta_1)) \\ \sim (\text{cohOp}\Sigma\text{tm} (\text{coh } x (\delta \odot \delta_1) A) (\text{sym} [\odot]T) \\ \sim \text{cohOp} (\text{sym} (\Sigma T[\Sigma s]T A (\delta \odot \delta_1))))^{-1}$$

$$\Sigma s \bullet \text{-left-id} : \forall \{\Gamma \Delta : \text{Con}\}(\gamma : \Gamma \Rightarrow \Delta) \rightarrow \Sigma s \{\Gamma\} \bullet \equiv \Sigma s \{\Delta\} \bullet \odot \Sigma s \gamma$$

$$\Sigma s \bullet \text{-left-id} \{\varepsilon\} \{\varepsilon\} \bullet = \text{refl}$$

$$\Sigma s \bullet \text{-left-id} \{\varepsilon\} \{\Delta, A\} (\gamma, a) = \text{trans} (\Sigma s \bullet \text{-left-id } \gamma) \\ (\text{sym} (\odot \text{wk} (\Sigma s \bullet \Delta)))$$

$$\Sigma s \bullet \text{-left-id} \{\Gamma, A\} \{\varepsilon\} \bullet = \text{trans} (\text{cong} (\lambda x \rightarrow x + S \Sigma T A) \\ (\Sigma s \bullet \text{-left-id} \{\Gamma\} \{\varepsilon\} \bullet)) (S\text{-eq } (S\text{-eq refl } ([+S]V (vS v0) \{\Sigma s \bullet \Gamma\}^{-1})) \\ ([+S]V v0 \{\Sigma s \bullet \Gamma\}^{-1}))$$

$$\Sigma s \bullet \text{-left-id} \{\Gamma, A\} \{\Delta, A_1\} (\gamma, a) = \text{trans} (\Sigma s \bullet \text{-left-id } \gamma) \\ (\text{sym} (\odot \text{wk} (\Sigma s \bullet \Delta)))$$

$$\Sigma s \odot \bullet \gamma = \Sigma s \bullet \text{-left-id } \gamma$$

$$\Sigma s \odot \{\Delta\} (\_, \_ \delta \{A\} a) \gamma = S\text{-eq} (\Sigma s \odot \delta \gamma) (\text{cohOp} (\Sigma T[\Sigma s]T A (\delta \odot \gamma)) \\ \sim \text{cohOp}\Sigma\text{tm} (a [\gamma] \text{tm}) [\odot]T \sim (\text{cohOp} [\odot]T \sim \text{congtm} \\ (\text{cohOp} (\Sigma T[\Sigma s]T A \delta)) \sim \Sigma\text{tm}[\Sigma s]\text{tm } a \gamma)^{-1})$$

$$\Sigma T[+S]T : \forall \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Delta)(\delta : \Gamma \Rightarrow \Delta)(B : \text{Ty } \Gamma) \rightarrow \\ \Sigma T A [\Sigma s \delta + S \Sigma T B]T \equiv \Sigma T (A [\delta]T) + T \Sigma T B \\ \Sigma T[+S]T A \delta B = \text{trans } [+S]T (\text{wk-T} (\Sigma T[\Sigma s]T A \delta))$$

$$\Sigma s \text{Dis} : \forall \{\Gamma \Delta : \text{Con}\}\{A : \text{Ty } \Delta\}(\delta : \Gamma \Rightarrow \Delta)(a : \text{Tm } (A [\delta]T)) \\ (B : \text{Ty } \Gamma) \rightarrow (\Sigma s \{\Gamma\} \{\Delta, A\} (\delta, a)) + S \Sigma T B \equiv$$



$$\begin{aligned}
& \Sigma_s \delta +S \Sigma T B, ((\Sigma_{tm} a) +tm \Sigma T B) \llbracket \Sigma T[+S]T A \delta B \rrbracket \\
& \Sigma_s Dis \{ \Gamma \} \{ \Delta \} \{ A \} \delta a B = S\text{-eq refl } (wk\text{-coh+} \sim (cohOp (trans [+S]T \\
& \quad (wk\text{-T } (\Sigma T[\Sigma_s]T A \delta))) \sim cong+tm2 (\Sigma T[\Sigma_s]T A \delta))^{-1}) \\
\\
& \Sigma_s \Sigma T : \forall \{ \Gamma \Delta : Con \} (\delta : \Gamma \Rightarrow \Delta) (B : Ty \Gamma) \rightarrow \Sigma_s (\delta +S B) \equiv \Sigma_s \delta +S \Sigma T B \\
& \Sigma_s \Sigma T \bullet \_ = refl \\
& \Sigma_s \Sigma T (\_, \_ \delta \{ A \} a) B = S\text{-eq } (\Sigma_s \Sigma T \delta B) (cohOp (\Sigma T[\Sigma_s]T A (\delta +S B)) \sim \\
& \quad cohOp \Sigma_{tm} (a +tm B) [+S]T \sim \Sigma_{tm}[+tm] a B \sim cong+tm2 (\Sigma T[\Sigma_s]T A \delta) \sim \\
& \quad wk\text{-coh+}^{-1}) \\
\\
& *'[\Sigma_s]T : \{ \Gamma \Delta : Con \} \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow *' \{ \Delta \} [ \Sigma_s \delta ]T \equiv *' \{ \Gamma \} \\
& *'[\Sigma_s]T \{ \epsilon \} \bullet = refl \\
& *'[\Sigma_s]T \{ \Gamma, A \} \bullet = trans ([+S]T \{ A = *' \{ \epsilon \} \} \{ \delta = \Sigma_s \{ \Gamma \} \bullet \}) \\
& \quad (wk\text{-T } (*'[\Sigma_s]T \{ \Gamma \} \bullet)) \\
& *'[\Sigma_s]T \{ \Gamma \} \{ \Delta, A \} (\gamma, a) = trans +T[, ]T (*'[\Sigma_s]T \gamma) \\
\\
& \Sigma T[\Sigma_s]T * \delta = *'[\Sigma_s]T \delta \\
& \Sigma T[\Sigma_s]T (\_ =h\_ \{ A \} a b) \delta = hom \equiv (\Sigma_{tm}[\Sigma_s]tm a \delta) (\Sigma_{tm}[\Sigma_s]tm b \delta) \\
\\
& \Sigma_{tm}[+tm] \{ A = A \} (var x) B = cohOpV (sym (\Sigma T[+T] A B)) \\
& \Sigma_{tm}[+tm] \{ \Gamma \} (coh \{ \Delta = \Delta \} x \delta A) B = cohOp \Sigma_{tm} (coh x (\delta +S B) A) \\
& \quad (sym [+S]T) \sim cohOp (sym (\Sigma T[\Sigma_s]T A (\delta +S B))) \sim coh\text{-eq } (\Sigma_s \Sigma T \delta B) \sim \\
& \quad cohOp (sym [+S]T)^{-1} \sim cong+tm2 (sym (\Sigma T[\Sigma_s]T A \delta)) \\
\\
& \Sigma C\text{-Contr} .(\epsilon, *) c* = ext c* v0 \\
& \Sigma C\text{-Contr} .(\Gamma, A, (var (vS x) =h var v0)) (ext \{ \Gamma \} r \{ A \} x) = \\
& \quad subst (\lambda y \rightarrow isContr (\Sigma C \Gamma, \Sigma T A, y)) \\
& \quad (hom \equiv (cohOpV (sym (\Sigma T[+T] A A))^{-1}) \\
& \quad (cohOpV (sym (\Sigma T[+T] A A))^{-1}) \\
& \quad (ext (\Sigma C\text{-Contr } \Gamma r) \{ \Sigma T A \} (\Sigma v x)))
\end{aligned}$$

**General suspension**

$$\Sigma C\text{-it} : \forall \{\Gamma\} (A : \text{Ty } \Gamma) \rightarrow \text{Con} \rightarrow \text{Con}$$

$$\Sigma T\text{-it} : \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) \rightarrow \text{Ty } \Delta \rightarrow \text{Ty } (\Sigma C\text{-it } A \Delta)$$

$$\begin{aligned} \Sigma \text{tm-it} &: \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) \{B : \text{Ty } \Delta\} \rightarrow \text{Tm } B \\ &\rightarrow \text{Tm } (\Sigma T\text{-it } A B) \end{aligned}$$

$$\begin{aligned} \text{suspend-S} &: \{\Gamma \Delta \Theta : \text{Con}\} (A : \text{Ty } \Gamma) \rightarrow \Theta \Rightarrow \Delta \rightarrow \\ &(\Sigma C\text{-it } A \Theta) \Rightarrow (\Sigma C\text{-it } A \Delta) \end{aligned}$$

$$\Sigma C\text{-it} * \Delta = \Delta$$

$$\Sigma C\text{-it } (\_ =_{\text{h}} \_ \{A\} \text{ a b}) \Delta = \Sigma C (\Sigma C\text{-it } A \Delta)$$

$$\Sigma T\text{-it} * B = B$$

$$\Sigma T\text{-it } (\_ =_{\text{h}} \_ \{A\} \text{ a b}) B = \Sigma T (\Sigma T\text{-it } A B)$$

$$\Sigma \text{tm-it} * t = t$$

$$\Sigma \text{tm-it } (\_ =_{\text{h}} \_ \{A\} \text{ a b}) t = \Sigma \text{tm} (\Sigma \text{tm-it } A t)$$

$$\text{suspend-S} * \gamma = \gamma$$

$$\text{suspend-S } (\_ =_{\text{h}} \_ \{A\} \text{ a b}) \gamma = \Sigma s (\text{suspend-S } A \gamma)$$

$$\text{minimum-S} : \forall \{\Gamma : \text{Con}\} (A : \text{Ty } \Gamma) \rightarrow \Gamma \Rightarrow \Sigma C\text{-it } A \epsilon$$

$$\Sigma C\text{-p1} : \{\Gamma : \text{Con}\} (A : \text{Ty } \Gamma) \rightarrow \Sigma C (\Gamma, A) \equiv \Sigma C \Gamma, \Sigma T A$$

$$\Sigma C\text{-p1} * = \text{refl}$$

$$\Sigma C\text{-p1 } (a =_{\text{h}} b) = \text{refl}$$

$$\begin{aligned} \Sigma C\text{-it-p1} &: \{\Gamma \Delta : \text{Con}\} (A : \text{Ty } \Gamma) (B : \text{Ty } \Delta) \rightarrow \Sigma C\text{-it } A (\Delta, B) \equiv \\ &(\Sigma C\text{-it } A \Delta, \Sigma T\text{-it } A B) \end{aligned}$$

$$\Sigma C\text{-it-p1} * B = \text{refl}$$

$$\Sigma C\text{-it-p1 } (\_ =_{\text{h}} \_ \{A\} \text{ a b}) B = \text{cong } \Sigma C (\Sigma C\text{-it-p1 } A B)$$

$$\begin{aligned}
\Sigma C\text{-it-S-spl}' &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)(B : \text{Ty } \Delta) \rightarrow \\
&(\Sigma C\text{-it } A \Delta, \Sigma T\text{-it } A B) \equiv \Sigma C\text{-it } A (\Delta, B) \\
\Sigma C\text{-it-S-spl}' * B &= \text{refl} \\
\Sigma C\text{-it-S-spl}' (\_ =h\_ \{A\} a b) B &= \text{cong } \Sigma C (\Sigma C\text{-it-S-spl}' A B)
\end{aligned}$$

$$\begin{aligned}
\Sigma C\text{-it-S-spl} &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)(B : \text{Ty } \Delta) \rightarrow \\
&(\Sigma C\text{-it } A \Delta, \Sigma T\text{-it } A B) \Rightarrow \Sigma C\text{-it } A (\Delta, B) \\
\Sigma C\text{-it-S-spl} * B &= \text{IdS} \\
\Sigma C\text{-it-S-spl} (\_ =h\_ \{A\} a b) B &= \Sigma s (\Sigma C\text{-it-S-spl } A B)
\end{aligned}$$

$$\begin{aligned}
\Sigma C\text{-it-S-spl}^{-1} &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)(B : \text{Ty } \Delta) \rightarrow \\
&\Sigma C\text{-it } A (\Delta, B) \Rightarrow (\Sigma C\text{-it } A \Delta, \Sigma T\text{-it } A B) \\
\Sigma C\text{-it-S-spl}^{-1} * B &= \text{IdS} \\
\Sigma C\text{-it-S-spl}^{-1} (\_ =h\_ \{A\} a b) B &= \Sigma s (\Sigma C\text{-it-S-spl}^{-1} A B)
\end{aligned}$$

$$\begin{aligned}
\Sigma C\text{-it-S-spl2} &: \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma) \\
&\rightarrow (\Sigma C\text{-it } A \epsilon, \Sigma T\text{-it } A *, \Sigma T\text{-it } A * +T \_) \Rightarrow \\
&\Sigma C (\Sigma C\text{-it } A \epsilon) \\
\Sigma C\text{-it-S-spl2} * &= \text{IdS} \\
\Sigma C\text{-it-S-spl2} (\_ =h\_ \{A\} a b) &= \Sigma s (\Sigma C\text{-it-S-spl2 } A) \odot 1\text{-1S-same} \\
&(\Sigma T[+T] (\Sigma T\text{-it } A *) (\Sigma T\text{-it } A *))
\end{aligned}$$

$$\begin{aligned}
\Sigma T\text{-it-wk} &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)(B : \text{Ty } \Delta) \rightarrow \\
&(\Sigma T\text{-it } A *) [\Sigma C\text{-it-S-spl } A B]T \equiv \Sigma T\text{-it } A * +T \_ \\
\Sigma T\text{-it-wk} * B &= \text{refl} \\
\Sigma T\text{-it-wk} (\_ =h\_ \{A\} a b) B &= \text{trans } (\Sigma T[\Sigma s]T (\Sigma T\text{-it } A *) \\
&(\Sigma C\text{-it-S-spl } A B)) (\text{trans } (\text{cong } \Sigma T (\Sigma T\text{-it-wk } A B)) \\
&(\Sigma T[+T] (\Sigma T\text{-it } A *) (\Sigma T\text{-it } A B)))
\end{aligned}$$

$$\Sigma T\text{-it-p1} : \forall \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma) \rightarrow \Sigma T\text{-it } A * [\text{minimum-S } A]T \equiv A$$

$$\begin{aligned}
\Sigma T\text{-it-p2} &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)\{B : \text{Ty } \Delta\}\{a b : \text{Tm } B\} \rightarrow \\
&\quad \Sigma T\text{-it } A (a =_h b) \equiv (\Sigma \text{tm-it } A a =_h \Sigma \text{tm-it } A b) \\
\Sigma T\text{-it-p2 } * &= \text{refl} \\
\Sigma T\text{-it-p2 } (\_ =_h \_ \{A\} \_ \_) &= \text{cong } \Sigma T (\Sigma T\text{-it-p2 } A)
\end{aligned}$$

$$\begin{aligned}
\Sigma T\text{-it-p3} &: \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma)\{B C : \text{Ty } \Delta\} \rightarrow \\
&\quad \Sigma T\text{-it } A (C + T B) [\Sigma C\text{-it-S-spl } A B ]T \equiv \Sigma T\text{-it } A C + T \_ \\
\Sigma T\text{-it-p3 } * &= \text{trans } +T[.]T (\text{wk}+S+T \text{ IC-T}) \\
\Sigma T\text{-it-p3 } (\_ =_h \_ \{A\} a b) \{B\} \{C\} &= \text{trans } (\Sigma T[\Sigma s]T (\Sigma T\text{-it } A (C + T B)) \\
&\quad (\Sigma C\text{-it-S-spl } A B)) (\text{trans } (\text{cong } \Sigma T (\Sigma T\text{-it-p3 } A)) (\Sigma T[+T] \\
&\quad (\Sigma T\text{-it } A C) (\Sigma T\text{-it } A B)))
\end{aligned}$$

$$\begin{aligned}
\text{minimum-S } * &= \bullet \\
\text{minimum-S } \{\Gamma\} (\_ =_h \_ \{A\} a b) &= \Sigma C\text{-it-S-spl2 } A \odot ((\text{minimum-S } A , \\
&\quad (a \llbracket \Sigma T\text{-it-p1 } A \rrbracket)) , (\text{wk-tm } (b \llbracket \Sigma T\text{-it-p1 } A \rrbracket)))
\end{aligned}$$

$$\begin{aligned}
\Sigma C\text{-it-}\epsilon\text{-Contr} &: \forall \{\Gamma \Delta : \text{Con}\}(A : \text{Ty } \Gamma) \rightarrow \text{isContr } \Delta \rightarrow \\
&\quad \text{isContr } (\Sigma C\text{-it } A \Delta) \\
\Sigma C\text{-it-}\epsilon\text{-Contr } * &= \text{isC} \\
\Sigma C\text{-it-}\epsilon\text{-Contr } (\_ =_h \_ \{A\} a b) \text{ isC} &= \Sigma C\text{-Contr } \_ (\Sigma C\text{-it-}\epsilon\text{-Contr } A \text{ isC})
\end{aligned}$$

$$\begin{aligned}
\text{wk-susp} &: \forall \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma)(a : \text{Tm } A) \rightarrow a \llbracket \Sigma T\text{-it-p1 } A \rrbracket \cong a \\
\text{wk-susp } A a &= \text{cohOp } (\Sigma T\text{-it-p1 } A)
\end{aligned}$$

$$\begin{aligned}
\text{fci-l1} &: \forall \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma) \rightarrow \Sigma T (\Sigma T\text{-it } A *) [\Sigma C\text{-it-S-spl2 } A ]T \equiv \\
&\quad (\text{var } (vS v0) =_h \text{var } v0) \\
\text{fci-l1 } * &= \text{refl} \\
\text{fci-l1 } \{\Gamma\} (\_ =_h \_ \{A\} a b) &= \text{trans } [\odot]T (\text{trans } (\text{congT } (\text{trans } (\Sigma T[\Sigma s]T \\
&\quad (\Sigma T (\Sigma T\text{-it } A *)) (\Sigma C\text{-it-S-spl2 } A)) (\text{cong } \Sigma T (\text{fci-l1 } A)))) (\text{hom} \equiv
\end{aligned}$$

$$\begin{aligned}
& (\text{congtm } (\Sigma\text{tm-p2-sp } (\Sigma\text{T-it } A \text{ *}) (\Sigma\text{T-it } A \text{ * } +\text{T } \Sigma\text{T-it } A \text{ *})) \sim \\
& 1\text{-1S-same-tm } (\Sigma\text{T}[+\text{T}] (\Sigma\text{T-it } A \text{ *}) (\Sigma\text{T-it } A \text{ *})) (\text{var } v0)) \\
& (\text{congtm } (\Sigma\text{tm-p1 } (\Sigma\text{T-it } A \text{ * } +\text{T } \Sigma\text{T-it } A \text{ *})) \sim 1\text{-1S-same-v0 } (\Sigma\text{T}[+\text{T}] \\
& (\Sigma\text{T-it } A \text{ *}) (\Sigma\text{T-it } A \text{ *}))) )
\end{aligned}$$

$$\Sigma\text{T-it-p1 } * = \text{refl}$$

$$\begin{aligned}
\Sigma\text{T-it-p1 } (\_ =\text{h\_ } \{A\} a b) &= \text{trans } [\odot]\text{T } (\text{trans } (\text{congT } (\text{fci-l1 } A)) \\
& (\text{hom}\equiv (\text{prf } a) (\text{prf } b)))
\end{aligned}$$

where

$$\begin{aligned}
\text{prf} : (a : \text{Tm } A) &\rightarrow ((a \llbracket \Sigma\text{T-it-p1 } A \rrbracket \gg) \llbracket +\text{T}[,\text{T}] \gg) \llbracket +\text{T}[,\text{T}] \gg \cong a \\
\text{prf } a &= \text{wk-coh} \sim \text{wk-coh} \sim \text{wk-susp } A a
\end{aligned}$$

$$\begin{aligned}
\Sigma\text{tm-it-p1} : \{ \Gamma \Delta : \text{Con} \} (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} &\rightarrow \Sigma\text{tm-it } A (\text{var } v0) \\
[ \Sigma\text{C-it-S-spl } A B ]\text{tm} &\cong \text{var } v0
\end{aligned}$$

$$\Sigma\text{tm-it-p1 } * \{B\} = \text{wk-coh} \sim \text{cohOp } (\text{wk}+\text{S}+\text{T } \text{IC-T})$$

$$\begin{aligned}
\Sigma\text{tm-it-p1 } (\_ =\text{h\_ } \{A\} a b) \{B\} &= \Sigma\text{tm}[\Sigma\text{s}]\text{tm } (\Sigma\text{tm-it } A (\text{var } v0)) \\
(\Sigma\text{C-it-S-spl } A B) &\sim \text{cong}\Sigma\text{tm } (\Sigma\text{tm-it-p1 } A) \sim \text{cohOpV } (\text{sym } (\Sigma\text{T}[+\text{T}] \\
& (\Sigma\text{T-it } A B) (\Sigma\text{T-it } A B)))
\end{aligned}$$

$$\Sigma\text{tm-it-p2} : \{ \Gamma \Delta : \text{Con} \} (A : \text{Ty } \Gamma) \{ B C : \text{Ty } \Delta \} (x : \text{Var } B) \rightarrow$$

$$(\Sigma\text{tm-it } A (\text{var } (vS \text{ } x))) [ \Sigma\text{C-it-S-spl } A C ]\text{tm} \cong$$

$$\Sigma\text{tm-it } A (\text{var } x) +\text{tm } \_$$

$$\Sigma\text{tm-it-p2 } * x = \text{wk-coh} \sim [+S]V x \sim \text{cong}+\text{tm } (\text{IC-v } x)$$

$$\begin{aligned}
\Sigma\text{tm-it-p2 } \{ \Gamma \} \{ \Delta \} (\_ =\text{h\_ } \{A\} a b) \{ B \} \{ C \} x &= \Sigma\text{tm}[\Sigma\text{s}]\text{tm } (\Sigma\text{tm-it } A \\
& (\text{var } (vS \text{ } x))) (\Sigma\text{C-it-S-spl } A C) \sim \text{cong}\Sigma\text{tm } (\Sigma\text{tm-it-p2 } \{ \Gamma \} \{ \Delta \} A \{ B \} x) \\
& \sim \Sigma\text{tm}[+\text{tm}] (\Sigma\text{tm-it } A (\text{var } x)) (\Sigma\text{T-it } A C)
\end{aligned}$$

$$\Sigma\text{C-it-Contr} : \forall \{ \Gamma \Delta \} (A : \text{Ty } \Gamma) \rightarrow \text{isContr } \Delta$$

$$\rightarrow \text{isContr } (\Sigma\text{C-it } A \Delta)$$

$$\Sigma\text{C-it-Contr } * x = x$$

$$\Sigma\text{C-it-Contr } \{ \Gamma \} \{ \Delta \} (\_ =\text{h\_ } \{A\} a b) x = \Sigma\text{C-Contr } (\Sigma\text{C-it } A \Delta) (\Sigma\text{C-it-Contr } A x)$$

## Replacement

$$\begin{aligned}
\text{rpl-C} & : \forall\{\Gamma\}(A : \text{Ty } \Gamma) \rightarrow \text{Con} \rightarrow \text{Con} \\
\text{rpl-T} & : \forall\{\Gamma \Delta\}(A : \text{Ty } \Gamma) \rightarrow \text{Ty } \Delta \rightarrow \text{Ty } (\text{rpl-C } A \Delta) \\
\text{rpl-tm} & : \forall\{\Gamma \Delta\}(A : \text{Ty } \Gamma)\{B : \text{Ty } \Delta\} \rightarrow \text{Tm } B \\
& \rightarrow \text{Tm } (\text{rpl-T } A B)
\end{aligned}$$

$$\begin{aligned}
\text{rpl-C } \{\Gamma\} A \epsilon & = \Gamma \\
\text{rpl-C } A (\Delta , B) & = \text{rpl-C } A \Delta , \text{rpl-T } A B
\end{aligned}$$

$$\begin{aligned}
\text{filter} & : \forall\{\Gamma\}(\Delta : \text{Con})(A : \text{Ty } \Gamma) \\
& \rightarrow \text{rpl-C } A \Delta \Rightarrow \Sigma\text{C-it } A \Delta
\end{aligned}$$

$$\text{rpl-T } A B = \Sigma\text{T-it } A B [ \text{filter } \_ A ] \text{T}$$

$$\text{rpl-pr1} : \{\Gamma : \text{Con}\}(\Delta : \text{Con})(A : \text{Ty } \Gamma) \rightarrow \text{rpl-C } A \Delta \Rightarrow \Gamma$$

$$\begin{aligned}
\text{rpl-pr1 } \epsilon A & = \text{IdS} \\
\text{rpl-pr1 } (\Delta , A) A_1 & = \text{rpl-pr1 } \Delta A_1 +S \_
\end{aligned}$$

$$\begin{aligned}
\text{filter } \epsilon A & = \text{minimum-S } A \\
\text{filter } (\Delta , A) A_1 & = \Sigma\text{C-it-S-spl } A_1 A \odot ((\text{filter } \Delta A_1 +S \_ , \\
& \text{var v0 } \llbracket [+S] \text{T} \rrbracket))
\end{aligned}$$

$$\begin{aligned}
\text{rpl-T-p1} & : \{\Gamma : \text{Con}\}(\Delta : \text{Con})(A : \text{Ty } \Gamma) \rightarrow \text{rpl-T } A * \equiv A [ \text{rpl-pr1 } \Delta A ] \text{T} \\
\text{rpl-T-p1 } \epsilon A & = \text{trans } (\Sigma\text{T-it-p1 } A) (\text{sym IC-T}) \\
\text{rpl-T-p1 } (\Delta , A) A_1 & = \text{trans } [\odot] \text{T} (\text{trans } (\text{congT } (\Sigma\text{T-it-wk } A_1 A)) \\
& (\text{trans } +\text{T}[ , ] \text{T} (\text{trans } [+S] \text{T} (\text{trans } (\text{wk-T } (\text{rpl-T-p1 } \Delta A_1)) \\
& (\text{sym } [+S] \text{T}))))))
\end{aligned}$$

$$\text{rpl-tm } A a = \Sigma\text{tm-it } A a [ \text{filter } \_ A ] \text{tm}$$

$$\begin{aligned}
\text{rpl-tm-id} & : \{\Gamma : \text{Con}\}\{A : \text{Ty } \Gamma\} \rightarrow \text{Tm } A \rightarrow \text{Tm } (\text{rpl-T } \{\Delta = \epsilon\} A *) \\
\text{rpl-tm-id } x & = x \llbracket \Sigma\text{T-it-p1 } \_ \rrbracket
\end{aligned}$$

$$\begin{aligned} \text{rpl-T-p2} : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} \{ a \ b : \text{Tm } B \} \rightarrow \\ \text{rpl-T } A \ (a \text{ =h } b) \equiv (\text{rpl-tm } A \ a \text{ =h } \text{rpl-tm } A \ b) \\ \text{rpl-T-p2 } \Delta \ A = \text{congT } (\Sigma\text{T-it-p2 } A) \end{aligned}$$

$$\begin{aligned} \text{rpl-T-p3} : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} \{ C : \text{Ty } \Delta \} \\ \rightarrow \text{rpl-T } A \ (C \text{ +T } B) \equiv \text{rpl-T } A \ C \text{ +T } \_ \\ \text{rpl-T-p3 } \_ \ A = \text{trans } [\odot]T \ (\text{trans } (\text{congT } (\Sigma\text{T-it-p3 } A)) \\ (\text{trans } \text{+T}[.,]T \text{ [+S]T})) \end{aligned}$$

$$\begin{aligned} \text{rpl-T-p3-wk} : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} \{ C : \text{Ty } \Delta \} \\ \{ \gamma : \Gamma \Rightarrow \text{rpl-C } A \ \Delta \} \{ b : \text{Tm } ((\Sigma\text{T-it } A \ B \text{ [ filter } \Delta \ A \text{ ]T}) \text{ [ } \gamma \text{ ]T}) \} \\ \rightarrow \text{rpl-T } A \ (C \text{ +T } B) \text{ [ } \gamma \text{ , } b \text{ ]T} \equiv \text{rpl-T } A \ C \text{ [ } \gamma \text{ ]T} \\ \text{rpl-T-p3-wk } \Delta \ A = \text{trans } (\text{congT } (\text{rpl-T-p3 } \Delta \ A)) \text{ +T}[.,]T \end{aligned}$$

$$\begin{aligned} \text{rpl-tm-v0}' : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} \\ \rightarrow \text{rpl-tm } \{ \Delta = \Delta \text{ , } B \} \ A \ (\text{var } v0) \cong \text{var } v0 \\ \text{rpl-tm-v0}' \ \Delta \ A = [\odot]\text{tm} \ (\Sigma\text{tm-it } A \ (\text{var } v0)) \sim \text{congtm} \ (\Sigma\text{tm-it-p1 } A) \sim \\ \text{wk-coh} \sim \text{wk-coh+} \end{aligned}$$

$$\begin{aligned} \text{rpl-tm-v0} : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B : \text{Ty } \Delta \} \{ \gamma : \Gamma \Rightarrow \text{rpl-C } A \ \Delta \} \\ \{ b : \text{Tm } A \} \{ b' : \text{Tm } ((\Sigma\text{T-it } A \ B \text{ [ filter } \Delta \ A \text{ ]T}) \text{ [ } \gamma \text{ ]T}) \} \\ \rightarrow (\text{prf} : b' \cong b) \\ \rightarrow \text{rpl-tm } \{ \Delta = \Delta \text{ , } B \} \ A \ (\text{var } v0) \text{ [ } \gamma \text{ , } b' \text{ ]tm} \cong b \\ \text{rpl-tm-v0 } \Delta \ A \ \text{prf} = \text{congtm} \ (\text{rpl-tm-v0}' \ \Delta \ A) \sim \text{wk-coh} \sim \text{prf} \end{aligned}$$

$$\begin{aligned} \text{rpl-tm-vS} : \{ \Gamma : \text{Con} \} (\Delta : \text{Con}) (A : \text{Ty } \Gamma) \{ B \ C : \text{Ty } \Delta \} \{ \gamma : \Gamma \Rightarrow \text{rpl-C } A \ \Delta \} \\ \{ b : \text{Tm } (\text{rpl-T } A \ B \text{ [ } \gamma \text{ ]T}) \} \{ x : \text{Var } C \} \rightarrow \\ \text{rpl-tm } \{ \Delta = \Delta \text{ , } B \} \ A \ (\text{var } (vS \ x)) \text{ [ } \gamma \text{ , } b \text{ ]tm} \cong \\ \text{rpl-tm } A \ (\text{var } x) \text{ [ } \gamma \text{ ]tm} \\ \text{rpl-tm-vS } \Delta \ A \ \{ x = x \} = \text{congtm} \ ([\odot]\text{tm} \ (\Sigma\text{tm-it } A \ (\text{var } (vS \ x)))) \sim \\ (\text{congtm} \ (\Sigma\text{tm-it-p2 } A \ x)) \sim \text{+tm}[.,]\text{tm} \ (\Sigma\text{tm-it } A \ (\text{var } x)) \sim \\ ([\text{+S}]\text{tm} \ (\Sigma\text{tm-it } A \ (\text{var } x)))) \sim \text{+tm}[.,]\text{tm} \ (\Sigma\text{tm-it } A \ (\text{var } x) \\ \text{[ filter } \_ \ A \text{ ]tm}) \end{aligned}$$

### Basic examples of replacement

$\text{base-1} : \{\Gamma : \text{Con}\}\{A : \text{Ty } \Gamma\} \rightarrow \text{rpl-C } A \ (\epsilon, *) \equiv (\Gamma, A)$   
 $\text{base-1} = \text{cong } (\lambda x \rightarrow \_, x) \ (\Sigma\text{T-it-p1 } \_)$

$\text{map-1} : \{\Gamma : \text{Con}\}\{A : \text{Ty } \Gamma\} \rightarrow (\Gamma, A) \Rightarrow \text{rpl-C } A \ (\epsilon, *)$   
 $\text{map-1} = \text{1-1S-same } (\Sigma\text{T-it-p1 } \_)$

### Lemmas about replacement

$\text{rpl}^*\text{-A} : \{\Gamma : \text{Con}\}\{A : \text{Ty } \Gamma\} \rightarrow \text{rpl-T } \{\Delta = \epsilon\} A \ * \ [\text{IdS}]T \equiv A$   
 $\text{rpl}^*\text{-A} = \text{trans IC-T } (\Sigma\text{T-it-p1 } \_)$

$\text{rpl}^*\text{-a} : \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma)\{a : \text{Tm } A\} \rightarrow \text{rpl-tm } \{\Delta = \epsilon, *\} A$   
 $(\text{var } v0) \ [\text{IdS}, a \llbracket \text{rpl}^*\text{-A} \rrbracket]_{\text{tm}} \cong a$   
 $\text{rpl}^*\text{-a } A = \text{rpl-tm-v0 } \epsilon \ A \ (\text{cohOp } (\text{rpl}^*\text{-A } \{A = A\}))$

$\text{rpl}^*\text{-A2} : \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma)\{a : \text{Tm } (\text{rpl-T } A \ (* \ \{\epsilon\}) \ [\text{IdS}]T)\}$   
 $\rightarrow \text{rpl-T } A \ (* \ \{\epsilon, *\}) \ [\text{IdS}, a]T \equiv A$   
 $\text{rpl}^*\text{-A2 } A = \text{trans } (\text{rpl-T-p3-wk } \epsilon \ A) \ \text{rpl}^*\text{-A}$

$\text{rpl-xy} : \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma)(a \ b : \text{Tm } A)$   
 $\rightarrow \text{rpl-T } \{\Delta = \epsilon, *, *\} A \ (\text{var } (vS \ v0) =_h \text{var } v0)$   
 $[\text{IdS}, a \llbracket \text{rpl}^*\text{-A} \rrbracket, b \llbracket \text{rpl}^*\text{-A2 } A \rrbracket]T \equiv (a =_h b)$   
 $\text{rpl-xy } A \ a \ b = \text{trans } (\text{congT } (\text{rpl-T-p2 } (\epsilon, *, *) \ A))$   
 $(\text{hom} \equiv ((\text{rpl-tm-vS } (\epsilon, *) \ A) \sim \text{rpl}^*\text{-a } A)$   
 $(\text{rpl-tm-v0 } (\epsilon, *) \ A \ (\text{cohOp } (\text{rpl}^*\text{-A2 } A))))$

$\text{rpl-sub} : (\Gamma : \text{Con})(A : \text{Ty } \Gamma)(a \ b : \text{Tm } A) \rightarrow \text{Tm } (a =_h b)$   
 $\rightarrow \Gamma \Rightarrow \text{rpl-C } A \ (\epsilon, *, *, (\text{var } (vS \ v0) =_h \text{var } v0))$   
 $\text{rpl-sub } \Gamma \ A \ a \ b \ t = \text{IdS}, a \llbracket \text{rpl}^*\text{-A} \rrbracket, b \llbracket \text{rpl}^*\text{-A2 } A \rrbracket, t \llbracket \text{rpl-xy } A \ a \ b \rrbracket$



### C.2.2 First-level Groupoid Structure

$$\begin{aligned}
\text{Coh-rpl} & : \forall \{\Gamma \Delta\} (A : \text{Ty } \Gamma) (B : \text{Ty } \Delta) \rightarrow \text{isContr } \Delta \\
& \rightarrow \text{Tm } (\text{rpl-T } A B) \\
\text{Coh-rpl } \{\_ \} \{\Delta\} A \_ \text{isC} & = \text{coh } (\Sigma \text{C-it-}\epsilon\text{-Contr } A \text{ isC}) \_ \_
\end{aligned}$$

#### Reflexivity

$$\begin{aligned}
\text{refl}^*\text{-Tm} & : \text{Tm } \{x : *\} (\text{var } v0 =_h \text{var } v0) \\
\text{refl}^*\text{-Tm} & = \text{Coh-Contr } c^*
\end{aligned}$$

#### Symmetry

$$\begin{aligned}
\text{sym}^*\text{-Ty} & : \text{Ty } x : *, y : *, \alpha : x = y \\
\text{sym}^*\text{-Ty} & = vY =_h vX \\
\text{sym}^*\text{-Tm} & : \text{Tm } \{x : *, y : *, \alpha : x = y\} \text{sym}^*\text{-Ty} \\
\text{sym}^*\text{-Tm} & = \text{Coh-Contr } (\text{ext } c^* v0)
\end{aligned}$$

#### Transitivity (composition)

$$\begin{aligned}
\text{trans}^*\text{-Ty} & : \text{Ty } x : *, y : *, \alpha : x = y, z : *, \beta : y = z \\
\text{trans}^*\text{-Ty} & = (vX +_{\text{tm}} \_ +_{\text{tm}} \_) =_h vZ \\
\text{trans}^*\text{-Tm} & : \text{Tm } \text{trans}^*\text{-Ty} \\
\text{trans}^*\text{-Tm} & = \text{Coh-Contr } (\text{ext } (\text{ext } c^* v0) (vS v0))
\end{aligned}$$

$$\begin{aligned} \text{refl-Tm} & : \{\Gamma : \text{Con}\}(A : \text{Ty } \Gamma) \\ & \rightarrow \text{Tm } (\text{rpl-T } \{\Delta = x:*\} A (\text{var } v0 =_h \text{var } v0)) \\ \text{refl-Tm } A & = \text{rpl-tm } A \text{ refl*-Tm} \end{aligned}$$

$$\begin{aligned} \text{sym-Tm} & : \forall \{\Gamma\}(A : \text{Ty } \Gamma) \rightarrow \text{Tm } (\text{rpl-T } A \text{ sym*-Ty}) \\ \text{sym-Tm } A & = \text{rpl-tm } A \text{ sym*-Tm} \end{aligned}$$

$$\begin{aligned} \text{trans-Tm} & : \forall \{\Gamma\}(A : \text{Ty } \Gamma) \rightarrow \text{Tm } (\text{rpl-T } A \text{ trans*-Ty}) \\ \text{trans-Tm } A & = \text{rpl-tm } A \text{ trans*-Tm} \end{aligned}$$

$$\begin{aligned} \text{reflX} & : \text{Tm } (vX =_h vX) \\ \text{reflX} & = \text{refl-Tm } * +_{\text{tm}} \_ +_{\text{tm}} \_ \end{aligned}$$

$$\begin{aligned} \text{reflY} & : \text{Tm } (vY =_h vY) \\ \text{reflY} & = \text{refl-Tm } * +_{\text{tm}} \_ \end{aligned}$$

$$\begin{aligned} m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q & : \text{Con} \\ m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q = x:*,y:*,\alpha:x=y,z:*,\beta:y=z, *, & \\ (\text{var } (vS (vS v0))) =_h \text{var } v0 & \end{aligned}$$

$$\begin{aligned} vM & : \text{Tm } \{m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q\} * \\ vM & = \text{var } (vS (vS (vS (vS (vS (vS v0)))))) \end{aligned}$$

$$\begin{aligned} vN & : \text{Tm } \{m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q\} * \\ vN & = \text{var } (vS (vS (vS (vS (vS v0)))))) \end{aligned}$$

$$\begin{aligned} vMN & : \text{Tm } \{m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q\} (vM =_h vN) \\ vMN & = \text{var } (vS (vS (vS (vS v0)))) \end{aligned}$$

$$vP : \text{Tm } \{m:*,n:*,\alpha:m=n,p:*,\beta:n=p,q:*,\gamma:p=q\} *$$

$$vP = \text{var } (vS \ (vS \ (vS \ v0)))$$

$$\begin{aligned} vNP &: \text{Tm } \{m:*, n:*, \alpha:m=n, p:*, \beta:n=p, q:*, \gamma:p=q\} \ (vN =_h vP) \\ vNP &= \text{var } (vS \ (vS \ v0)) \end{aligned}$$

$$\begin{aligned} vQ &: \text{Tm } \{m:*, n:*, \alpha:m=n, p:*, \beta:n=p, q:*, \gamma:p=q\} \ * \\ vQ &= \text{var } (vS \ v0) \end{aligned}$$

$$\begin{aligned} vPQ &: \text{Tm } \{m:*, n:*, \alpha:m=n, p:*, \beta:n=p, q:*, \gamma:p=q\} \ (vP =_h vQ) \\ vPQ &= \text{var } v0 \end{aligned}$$

$$\begin{aligned} \text{Ty-G-assoc}^* &: \text{Ty } m:*, n:*, \alpha:m=n, p:*, \beta:n=p, q:*, \gamma:p=q \\ \text{Ty-G-assoc}^* &= (\text{trans}^* \text{-Tm } [ (((\bullet, vM), vP), \\ &\quad (\text{trans}^* \text{-Tm } [ pr1 \odot pr1 ]tm)), vQ), vPQ ]tm =_h \\ &\quad \text{trans}^* \text{-Tm } [ (pr1 \odot pr1 \odot pr1 \odot pr1, vQ), \\ &\quad (\text{trans}^* \text{-Tm } [ (((\bullet, vN), vP), vNP), vQ), \\ &\quad vPQ ]tm) ]tm) \end{aligned}$$

$$\begin{aligned} \text{Tm-right-identity}^* &: \\ &\quad \text{Tm } \{x:*, y:*, \alpha:x=y\} \ (\text{trans}^* \text{-Tm } [ \text{IdS}, vY, \text{reflY} ]tm \\ &\quad =_h v\alpha) \\ \text{Tm-right-identity}^* &= \text{Coh-Contr } (\text{ext } c^* \ v0) \end{aligned}$$

$$\begin{aligned} \text{Tm-left-identity}^* &: \\ &\quad \text{Tm } \{x:*, y:*, \alpha:x=y\} \ (\text{trans}^* \text{-Tm } [ ((\text{IdS} \odot pr1 \odot pr1), vX), \\ &\quad \text{reflX}, vY, v\alpha ]tm =_h v\alpha) \\ \text{Tm-left-identity}^* &= \text{Coh-Contr } (\text{ext } c^* \ v0) \end{aligned}$$

$$\begin{aligned} \text{Tm-right-inverse}^* &: \\ &\quad \text{Tm } \{x:*, y:*, \alpha:x=y\} \ (\text{trans}^* \text{-Tm } [ (\text{IdS}, vX), \text{sym}^* \text{-Tm } ]tm \\ &\quad =_h \text{reflX}) \\ \text{Tm-right-inverse}^* &= \text{Coh-Contr } (\text{ext } c^* \ v0) \end{aligned}$$

$\text{Tm-left-inverse}^*$  :

$\text{Tm } \{x:*, y:*, \alpha: x=y\} (\text{trans}^* \text{-Tm } [ ((\bullet, vY), vX, \text{sym}^* \text{-Tm } ,$   
 $vY), v\alpha ]_{\text{tm}} =_{\text{h}} \text{refl} Y)$

$\text{Tm-left-inverse}^* = \text{Coh-Contr } (\text{ext } c^* v0)$

$\text{Tm-G-assoc}^* : \text{Tm Ty-G-assoc}^*$

$\text{Tm-G-assoc}^* = \text{Coh-Contr } (\text{ext } (\text{ext } (\text{ext } c^* v0) (vS v0))$   
 $(vS v0))$

$\text{Tm-G-assoc} : \forall \{\Gamma\} (A : \text{Ty } \Gamma)$   
 $\rightarrow \text{Tm } (\text{rpl-T } A \text{ Ty-G-assoc}^*)$

$\text{Tm-G-assoc } A = \text{rpl-tm } A \text{ Tm-G-assoc}^*$

## C.3 Semantics

### Globular types

$\text{record Glob} : \text{Set}_1 \text{ where}$   
 $\text{constructor } \_ || \_$   
 $\text{field}$   
 $| \_ | : \text{Set}$   
 $\text{hom} : | \_ | \rightarrow | \_ | \rightarrow \infty \text{ Glob}$

$\text{open Glob public}$

$\text{Id}\omega : (A : \text{Set}) \rightarrow \text{Glob}$

$\text{Id}\omega A = A || (\lambda a b \rightarrow \# \text{Id}\omega (a \equiv b))$

## 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 ≡
    b (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

```

$$\begin{aligned}
\llbracket \_ \rrbracket^{\text{S-}\beta 1} & : \forall \{\Gamma\} \{\gamma : \llbracket \Gamma \rrbracket C\} \\
& \rightarrow \llbracket \bullet \rrbracket S \gamma \equiv \text{coerce } \llbracket \_ \rrbracket^{C-\beta 1} \text{ tt} \\
\\
\llbracket \_ \rrbracket^{\text{S-}\beta 2} & : \forall \{\Gamma \Delta\} \{A : \text{Ty } \Delta\} \{\delta : \Gamma \Rightarrow \Delta\} \{\gamma : \llbracket \Gamma \rrbracket C\} \\
& \{a : \text{Tm } (A \llbracket \delta \rrbracket T)\} \rightarrow \llbracket \delta , a \rrbracket S \gamma \\
& \equiv \text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} ((\llbracket \delta \rrbracket S \gamma) , \\
& \text{subst } \llbracket \_ \rrbracket (\text{semSb-T } A \delta \gamma) (\llbracket a \rrbracket^{\text{tm}} \gamma)) \\
\\
\text{semWk-T} & : \forall \{\Gamma A B\} (\gamma : \llbracket \Gamma \rrbracket C) (v : \llbracket B \rrbracket T \gamma) \\
& \rightarrow \llbracket A + T B \rrbracket T (\text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} (\gamma , v)) \equiv \\
& \llbracket A \rrbracket T \gamma \\
\\
\text{semWk-S} & : \forall \{\Gamma \Delta B\} \{\gamma : \llbracket \Gamma \rrbracket C\} \{v : \llbracket B \rrbracket T \gamma\} \\
& \rightarrow (\delta : \Gamma \Rightarrow \Delta) \rightarrow \llbracket \delta + S B \rrbracket S \\
& (\text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} (\gamma , v)) \equiv \llbracket \delta \rrbracket S \gamma \\
\\
\text{semWk-tm} & : \forall \{\Gamma A B\} (\gamma : \llbracket \Gamma \rrbracket C) (v : \llbracket B \rrbracket T \gamma) \\
& \rightarrow (a : \text{Tm } A) \rightarrow \text{subst } \llbracket \_ \rrbracket (\text{semWk-T } \gamma v) \\
& (\llbracket a + \text{tm } B \rrbracket^{\text{tm}} (\text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} (\gamma , v))) \\
& \equiv (\llbracket a \rrbracket^{\text{tm}} \gamma) \\
\\
\pi\text{-}\beta 1 & : \forall \{\Gamma A\} (\gamma : \llbracket \Gamma \rrbracket C) (v : \llbracket A \rrbracket T \gamma) \\
& \rightarrow \text{subst } \llbracket \_ \rrbracket (\text{semWk-T } \gamma v) \\
& (\pi \text{ v0 } (\text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} (\gamma , v))) \equiv v \\
\\
\pi\text{-}\beta 2 & : \forall \{\Gamma A B\} (x : \text{Var } A) (\gamma : \llbracket \Gamma \rrbracket C) (v : \llbracket B \rrbracket T \gamma) \\
& \rightarrow \text{subst } \llbracket \_ \rrbracket (\text{semWk-T } \gamma v) (\pi (\text{vS } \{\Gamma\} \{A\} \{B\} x) \\
& (\text{coerce } \llbracket \_ \rrbracket^{C-\beta 2} (\gamma , v))) \equiv \pi x \gamma \\
\\
\llbracket \text{coh} \rrbracket & : \forall \{\Theta\} \rightarrow \text{isContr } \Theta \rightarrow (A : \text{Ty } \Theta) \\
& \rightarrow (\theta : \llbracket \Theta \rrbracket C) \rightarrow \llbracket A \rrbracket T \theta
\end{aligned}$$

# Bibliography

- [1] Irrelevance – agda wiki. URL <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Irrelevance>.
- [2] Michael Abbott, 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/S09567968090007278. URL <http://dx.doi.org/10.1017/S09567968090007278>.
- [6] Thorsten Altenkirch and Nicolai Kraus. Setoids are not an LCCC, 2012.
- [7] Thorsten Altenkirch and Ondřej Rypáček. A syntactical Approach to Weak  $\omega$ -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 Löb, and Nicolas Oury. Pisma: Dependent Types without the Sugar. submitted for publication, November 2009.
- [10] Thorsten Altenkirch, Thomas Anberree, and Nuo Li. Definable Quotients in Type Theory. 2011.
- [11] Thorsten Altenkirch, Nuo Li, and Ondřej Rypáček. Some constructions on  $\omega$ -groupoids. In *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, LFMTTP '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  $\infty$ -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](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](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-3WXSM9-6/2/c1225a60fbeld641e225bdf749181845>.
- [24] Guillaume Brunerie. Syntactic Grothendieck weak  $\infty$ -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](http://dx.doi.org/10.2168/LMCS-1(2:1)2005).
- [26] Pierre Clairambault. From categories with families to locally cartesian closed categories. *Project Report, ENS Lyon*, 2006.
- [27] Cyril Cohen. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving*, pages 213–228, 2013.

- [28] 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.
- [29] 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>.
- [30] Thierry Coquand. Pattern Matching with Dependent Types. In *Types for Proofs and Programs*, 1992.
- [31] Thierry Coquand. Types as Kan Simplicial Sets. Accessed: 2014-09-10, 12 2012. URL <http://www.cse.chalmers.se/~coquand/stockholm.pdf>.
- [32] 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/>.
- [33] Pierre Courtieu. **Normalized types**. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] Peter Dybjer. Internal Type Theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.
- [38] Martín Escardó. Counterexample on local continuity. URL <http://www.cs.bham.ac.uk/~mhe/agda/FailureOfTotalSeparatedness.html>. Accessed: 2014-08-20.
- [39] Kurt Gödel. *On formally undecidable propositions of principia mathematica and related systems*. Dover, 1992.
- [40] 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>.
- [41] 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.
- [42] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.
- [43] 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>.
- [44] 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.
- [45] 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>.

- [46] Martin Hofmann. *Extensional concepts in Intensional Type Theory*. PhD thesis, School of Informatics., 1995.
- [47] 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>.
- [48] Martin Hofmann. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.
- [49] 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.
- [50] Peter V. Homeier. Quotient Types. In *In TPHOLs 2001: Supplemental Proceedings*, page 0046, 2001.
- [51] Antonius JC Hurkens. A simplification of Girard’s paradox. In *Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.
- [52] Bart Jacobs. Quotients in Simple Type Theory. *Manuscript, Math. Inst*, 1994.
- [53] Krzysztof Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The Simplicial Model of Univalent Foundations. arXiv:1211.2851, 2012.
- [54] Stephen C Kleene and J Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.
- [55] Nicolai Kraus. Non-Normalizability of Cauchy Sequences. 2014. URL <http://www.cs.nott.ac.uk/~ngk/normalizability.pdf>.
- [56] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of Anonymous Existence in Martin-Löf Type Theory. 2014.
- [57] Nuo Li. Representing numbers in Agda. Technical report, School of Computer Science, University of Nottingham, 2010. Final year dissertation.

- [58] Nuo Li. Some constructions on  $\omega$ -groupoids: codes, 2014.
- [59] Robert S. Lubarsky. On the Cauchy Completeness of the Constructive Cauchy Reals. *Electr. Notes Theor. Comput. Sci.*, 167:225–254, 2007.
- [60] Peter LeFanu Lumsdaine. Weak  $\omega$ -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](http://dx.doi.org/10.2168/LMCS-6(3:24)2010).
- [61] G. Maltsiniotis. Grothendieck  $\infty$ -groupoids, and still another definition of  $\infty$ -categories. *ArXiv e-prints*, September 2010.
- [62] Per Martin-Löf. A Theory of Types. Technical Report 71–3, University of Stockholm, 1971.
- [63] 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.
- [64] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984. ISBN 88-7088-105-9.
- [65] 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.
- [66] 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.
- [67] Nicolas Tabareau Matthieu Sozeau. Internalization of the Groupoid Interpretation of Type Theory. *Types 2014*, 2014.
- [68] 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.

- [69] N.P. Mendler. Quotient types via coequalizers in Martin-Löf type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.
- [70] Aleksey Nogin. Quotient types: A Modular Approach. In *ITU-T Recommendation H.324*, pages 263–280. Springer-Verlag, 2002.
- [71] 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.
- [72] Ulf Norell. Dependently typed programming in Agda. Available at: <http://www.cse.chalmers.se/~ulfn/papers/afp08tutorial.pdf>, 2008.
- [73] 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.
- [74] Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. Preprint, June 2014.
- [75] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, Cambridge, 1903.
- [76] Thomas Streicher. *Investigations into intensional type theory*. PhD thesis, Habilitation thesis, Ludwig-Maximilians-University Munich, 1993.
- [77] Thomas Streicher. A model of type theory in simplicial sets: A brief introduction to Voevodsky’s homotopy type theory. pages 45–49, 2014.
- [78] Laurent Théry. A selected bibliography on formalised mathematics. URL <http://www-sop.inria.fr/marelle/personnel/Laurent.Thery/math.html>. Accessed: 2014-09-20.
- [79] A. S. Troelstra. From Constructivism to Computer Science. pages 233–252, 1999.
- [80] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <http://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.

- [81] 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>.
- [82] Benno van den Berg and Richard Garner. Types are weak  $\omega$ -groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.
- [83] Paul van der Walt. *Reflection in Agda*. PhD thesis, Master’s thesis, Universiteit Utrecht, 2012.
- [84] Vladimir Voevodsky. Generalities on hSet - Coq library hSet, . URL [http://www.math.ias.edu/~vladimir/Foundations\\_library/hSet.html](http://www.math.ias.edu/~vladimir/Foundations_library/hSet.html).
- [85] Vladimir Voevodsky. Univalent Foundations Project. . URL [http://www.math.ias.edu/~vladimir/Site3/Univalent\\_Foundations\\_files/univalent\\_foundations\\_project.pdf](http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf).
- [86] Vladimir Voevodsky. A very short note on the homotopy  $\lambda$ -calculus. 2006. URL [http://www.math.ias.edu/~vladimir/Site3/Univalent\\_Foundations\\_files/Hlambda\\_short\\_current.pdf](http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf).
- [87] Vladimir Voevodsky. A very short note on homotopy  $\lambda$ -calculus. Available at: [http://math.ucr.edu/home/baez/Voevodsky\\_note.ps](http://math.ucr.edu/home/baez/Voevodsky_note.ps), 2006.
- [88] 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](http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_Bergen.pdf).
- [89] Vladimir Voevodsky. A universe polymorphic type system. 2012. URL <http://uf-ias-2012.wikispaces.com/file/view/Universe+polymorphic+type+sytem.pdf>.
- [90] 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.
- [91] Michael Warren. The strict  $\omega$ -groupoid interpretation of type theory. In *Models, Logics and Higher-Dimensional Categories*, CRM Proc. Lecture Notes 53, pages 291–340. Amer. Math. Soc., 2011.

- [92] The Agda Wiki. Main page, 2014. URL <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage>. [Online; accessed 15-June-2014].
- [93] Wikipedia. Coinduction — Wikipedia, the free encyclopedia, 2010. URL <http://en.wikipedia.org/wiki/Coinduction>. [Online; accessed 20-April-2010].
- [94] Wikipedia. Lazy evaluation — Wikipedia, the free encyclopedia, 2010. URL [http://en.wikipedia.org/wiki/Lazy\\_evaluation](http://en.wikipedia.org/wiki/Lazy_evaluation). [Online; accessed 20-April-2010].