List of Corrections

DOCTORAL THESIS

Quotient Types in Type Thoery

Author:
Nuo LI

Supervisor:
Dr. Thorsten ALTENKIRCH &
Dr. Thomas ANBERRÉE

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

in the

Functional Programming Lab
Department or Computer Science

March 2014

THE UNIVERSITY OF NOTTINGHAM

Abstract

Faculty Name
Department or Computer Science

Doctor of Philosophy

Quotient Types in Type Thoery

by Nuo Li

The Thesis Abstract is written here (and usually kept to just this page). This thesis mainly covered the quotient types in type theory

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Write: what we are going to write

1.1 Equality and extensional concepts

1.2 Quotient types

1.3 different models

1.4 Applications

mention the definable quotient is very useful

1.5 Overview

compact, comprehensive Overview:Add overview of each part, as much as you can, and compact

In chapter 2, we will discuss the backgroud of this research. Type theory is a popular topic in theoretical computer science. It is quite powerful not only a a theory but also as a programming language. We use a dependent functional programming language called Agda which is design based on Martin-Löf type theory. The related work will also be discussed in this chapter.

In chapter 3, we will discuss quotient types which is the topic of this thesis in detail. Quotient types can be understood as a interpretation of quotient set in set theory. It

is an extensional concept which is also related to other extensional concepts. It can be encoded in different ways. Categorically speaking it is a coequalizer, and a split quotient is a just a split coequalizer.

In chapter 4, we start introducing one of our achievements, the definable quotients. It is usually very unreadable, unorganised and complicated to write some programs without abstracting. It is also applied to quotient types. If we have some types that can be abstract as a quotient type of some common types, then it will be easily encoded and manipulated. 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 represent the same subtraction.

In chapter 5,

In chapter 6, we will talk about the setoid model approach to encode extensional concepts. The work is mainly extending the setoid model done by Altenkirch in [1] to quotient types.

In chapter 7, we will discuss the new area between mathematics and computer science – Homotopy Type Theory. We will talk about the higher inductive types and also the weak ω-groupoids-model which is used to interpret homotopy types in intensional type theory. Quotient types can be encoded Homotopy Type Theory simply.

Chapter 2

Background

Regular mathematics is based on classical logic and ZFC set theory. Set theory is a language to describe definitions of most mathematical objects, in other words, it serves as a foundational system for mathematics.

George Cantor and Richard Dedekind invented set theory in 1870s. However, in the 1900s, Bertrand Russell discovered a paradox in their system. In this naive set theory, there was no distinction between small sets like the set of natural numbers or the set of real numbers and "larger" sets like the set of all sets. This lead to the famous contradiction named after Russell. To avoid this paradox, he proposed the theories theory? of types [2] as an alternative to naïve set 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 on the same level as its elements and the paradox disappears.

2.1 Type Theory

The elementary notion of type theory is *type* which plays a similar role to set in set theory, but differs fundamentally. Every object in type theory comes with its unique type, while an object in set theory can appear in multiple sets and we can talk about an object without knowing which sets it belongs to. To explain the difference, we use the the number 2 as an example. In set theory, 2 is not an element of only one specific set, it belongs to the set of natural numbers $\mathbb{N}$ and also the set of integers $\mathbb{Z}$. While in type theory, it is impossible to avoid mentioning the type of object 2. Usually the term suc (suc zero) stands for 2 of type $\mathbb{N}$ and we have a different term of type $\mathbb{Z}$ constructed by constructors of $\mathbb{Z}$ which is a different object to the one of $\mathbb{N}$.

Since Russell's type theory, a variety of type theories have been developed by mathematicians and computer scientists, for example Gödel's System T [3]. There are two families of famous type theories building the bridges between mathematics and computer science, *lambda calculus* and *Martin-Löf type theory*.

2.1.1 Lambda Calculus

Alonzo Church introduced lambda calculus in the 1930s. He first introduced an untyped lambda calculus which turned out to be inconsistent due to the Kleene-Rosser paradox ?cite? . Then he refined it with adding types. This theory is also called Church's theory of types or simply typed lambda calculus is introduced as a foundation of mathematics. An important change is that functions become primitive objects which means functions are types defined inductively using the $\rightarrow$ type former. It is widely applied to various fields especially computer science. Some languages are extentions of lambda calculus, for example Haskell. Haskell belongs to one of the variants of lambda calculus called System F, although it has evovled into System FC recently. There are also other refinements of lambda calculus which is illustrated by the λ-cube [4].

2.1.2 Per Martin-Löf's Type Theories

In 1970s, Per Martin-Löf [5, 6] developed his profound intuitionistic type theory. His 1971's formulation which is impredicative was proved to be inconsistent with the Girard's paradox [7]. The impredicativty means that for a universe U, there is an axiom $\mathsf{U} \in \mathsf{U}$. The later version is predicative and is more widely used.

Like set theory, it can also serve as also a foundation of constructive mathematics [8]. Different to set theory whose axioms are based on first-order logic or intuitionistic logic, Martin-Löf type theory provides a means of implementing intuitionistic logic. This is achieved by the Curry-Howard isomorphism:

"propositions can be interpreted as types and their proofs are inhabitants of that types"

more explanation

Different to simply lambda calculus, dependent types are introduced.

Definition 2.1. *Dependent type.* Dependent types are types that depends on values of other types [9].

With dependent types, the quantifiers like $\forall$ and $\exists$ can be encoded. The Curry-Howard isomorphism is then extended to predicate logic. A predicate on X can be written as a dependent type Px where $x : X$.

There are also two variants of Martin-Löf type theory based on the treatment of equality. The notion of equality is one of the most profound topic in type theory. we have two kinds of equality, one is definitional equality, the other is propositional equality.

Definition 2.2. *definitional equality* definitional equality is a judgement-level equality, which holds when two objects have the same normal forms[10].

With dependent types, it is possible to write a type to encode the equality of objects.

Definition 2.3. *Propositional equality* Propositional equality is a type which represents a propostion that two objects of the same type are equal.

Intuitively if two objects are definitionally equal, they must be propositionally equal.

$$\frac{a \equiv b}{a = b}\ Id - intro$$

But how about the other way around? Are two propositional equal objects definitional equal?

In intensional type theory, the answer is no. Propositional equality (also called intensional equality [10]) is different to definitional equality. The definitional equality is always decidable hence type checking that depends on definitional equality is decidable as well [1]. Therefore intensional type theory has better computational behaviors. Types like $\mathsf{a} = \mathsf{b}$ which stands for a propostion that a equals b are propositional equalities. They are some types which we need to prove or disprove by construction. Each of them has an unique element refl which only exists if a and b are definitionally equal in all cases. However it is not enough for other extensional equalities, for example the equality of functions.

In extensional type theory, the propositional equality is extensional and is undistinguished with definitional equality, in other words, two propositional equal objects are judgementally equal. This is called reflecition rule.

$$\frac{a = b}{a \equiv b}\ Reflection \tag{2.1}$$

Objects of different normal forms, for example point-wise equal functions or different proofs of the same proposition, may be definitionally equal. This is called functional extensionality.

$$\frac{f\,g : A \to B, \forall a : A, f\,a = g\,a}{f = g}\ functional - extensionality \tag{2.2}$$

It is provable in extensional type theory.

Lemma 2.4. *We can prove 2.2 in extensional type theory where we have 2.1.*

Proof. Suppose $\Gamma \vdash f\,a = g\,a$, with reflection rule we have $\Gamma \vdash f\,a \equiv g\,a$. Then using ξ-rule, we know that $\Gamma \vdash \lambda a.f\,a \equiv \lambda a.g\,a$. From η-equivalence, we know that $\Gamma \vdash f \equiv g$. We can conclude that $\Gamma \vdash f = g$. □

In intensional type theory, this is not provable. If we add it as an axiom, we will lose the canonicity since we can construct a natural number with this extensionality and substitution for propositional equality. It means that the definitional equality is not decidable and type checking becomes undecidable as well. The type checker will not always terminate.

Usually we have to make a choice of them, either the better adpation of extensional equality or better computational behaviors. Agda chooses the intensional one while NuPRL chooses the extensional one. However Altenkirch and McBride introduced a variant of extensional type theory called *Observational Type Theory* [11] in which definitional equality is decidable and propositional equality is extensional.

Martin-Löf type theory can be encoded as programming languages in which the evaluation of a well-typed program always terminates [10]. There are various implementations based on different variants of it, such as NuPRL, LEGO, Coq, Epigram and Agda. Since we usually discuss in intensional type theory, we will use Agda throughout this thesis.

2.2 Agda

Agda is a dependently typed functional programming language which is designed based on intensional version of Martin-Löf type theory [12].

As we have seen, Martin-Löf type theory is based on the Curry-Howard isomorphism: types are identified with propositions and terms (or programs) are identified with proofs. It turns Agda is into a proof assistant like Coq, which allows users to do mathematical

reasoning. We can also reason about Agda programs inside itself. Usually to prove the correctness of programs, we need to state some theorems of programming languages on the meta-level, but in Agda we can prove and use these theorems alongwith writing programs.

There are more features of Agda as follows:

- *Dependent type.* As mentioned in 2.1, dependent types are types that depends on values of other types [9]. They enable us to write more expresive types as program speficication or propositions in order to reduce bugs. In Haskell and other Hindley-Milner style languages, types and values are clearly distinct [13], In Agda, we can define types depending on values which means the border between types and values is vague. To illustrate what this means, the most common example is VectorAn where we can length-explicit lists called vectors. It is a data type which represents a vector containing elements of type **A** and depends on a natural number **n** which is the length of the list. We can specify types with more constraints such that the we can express what programs we can better and leave the checking work to the type chekcer. For instance, to use the length-explicit vector, we will not encounter exceptions like out of bounds in Java, since it is impossible to define such functions before compiling.

- *Functional programming language.* As the name indicates that, functional programming languages emphasizes the application of functions rather than changing data in the imperative style like C++ and Java. The base of functional programming is lambda calculus. The key motivation to develop functional programmming language is to eliminating the side effects which means we can ensure the result will be the same no matter how many times we input the same data. There are several generations of functional programming languages, for example Lisp, Erlang, Haskell etc. Most of the applications of them are currently in the academic fields, however as the functional programming developed, more applications will be explored.

- *A proof assistant* Based on the Curry-Howard isomorphism, we have predicate logic available in Agda and we can prove mathematical theorems and theorems about the programs encoded in Agda itself.

  The general approach to do theorem proving in Agda is as follows: First we give the name of the proposition and encode it as the type. Then we can gradually refine the goal to formalise a type-correct program namely the proof. As long as we have the proof, it can be used as a lemma in other proofs or programs. Usually, there are no tactics like in Coq (it may be implemented in the future). But with

the gradually refinement mechanism, the process of building proofs is very similar to conceiving proofs in regular mathematics.

As a functional programming languages, Agda has some nice features for theorem proving,

- *Pattern matching.* The mechanism for dependently typed pattern matching is very powerful [14]. We could prove propositions case by case. In fact it is similar to the approach to prove propositions case by case in regular mathematics. Pattern match is a more intuitive way to use the eliminators of types.
- *Inductive & Recursive definition.* In Agda, types are often defined inductively, for example, natural numbers is defined as

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

  The function for inductive types are usually written in recursive style, for example, the double function for natural numbers,

  ```
  double : ℕ → ℕ
  double zero = zero
  double (suc n) = suc (suc (double n))
  ```

  The availability of recursive definition enables programmers to prove propositions in the same manner of mathematical induction.

- *Construction of functions.* One of the advantage of using a functional programming language as a theorem prover is the construction of functions which makes the proving more flexible.

  In functional programming languages, complicated programs are commonly built gradually using aunxiliary functions and frequently used functions in the library.

  Described as a proof assistant, complicated theorems are commonly proved gradually using lemmas and other theorems we have proved.

  This decreases the difficulty of interpreting proofs in mathematics into Agda.

- *Lazy evaluation.* Lazy evaluation could eliminate unecessary operation because Agda is lazy to delay a computation until we need its result. It is often used to handle infinite data structures. [15]

Agda also has some special functions in its interactive emacs interface beyond simple functional programming languages.

- *Type Checker.* Type checker is an essential part of Agda. You can use to to type check a file without compiling it. It is the type checker that detect type mismatch problem and for theorem proving, it means the proof is incorrect. It interactively shows the goals, assumptions and variables when buiding a proof.

  The coverage checker makes sure that the patterns cover all possible cases [16].

  The termination checker will warn possiblily non-terminated error. The missing cases error will be reported by type checker. The suspected non-terminated definition can not be used by other ones. All programs must terminate in Agda so that it will not crash [13]. The type checker then ensures that the proof is complete and not been proved by itself.

  In Agda, type signatures are essential due to the presence of type checker.

- *Interactive interface.* It has a Emacs-based interface for interactively writing and verifying proofs. With type checker we can refine our proofs step by step [16]. It also has some convenient functions and emacs means the potential to be extended.

- *Unicode support.* In Haskell and Coq, unicode support is not an essential part. However in Agda, to be a better theorem prover, it reads unicode symbols like: β, $\forall$ and $\exists$ and supports mixfix operators like: $+$ and $-$, which are very common for mathematics. It provides more meaningful names for types and lemmas and more flexible way to define operators. This also improve the readablity of the Agda proofs. For example, the commutativity of plus for natural numbers can be encoded as follows

  $$\mathsf{comm} : \forall\ (a\ b : \mathbb{N}) \to a + b \equiv b + a$$

  We can use symbols we are familiar in regular mathematics.

  Secondly we could use symbols to replace some common-used properties to simply the proofs a lot. The following code was simplied using several symbols,

  Finally, we could use some other languages characters to define functions such as Chinese characters.

- *Code navigation.* As long as a program is loaded, it provides shortcut keys to move to the original definitions of certain object and move back. In real life programming it alleviates a great deal of work of programmers to look up the library.

- *Implicit arguments.* Sometime it is unnessary to write an argument since it can be inferred from other arguments by the type checker. It can simplify the application of functions and make the programs more concise. For example, to define a polymorphic function id,

  $$\mathsf{id} : \{A : \mathsf{Set}\} \to A \to A$$
  $$\mathsf{id}\ a = a$$

  Whenever we give an argument a, its type A must be inferable.

- *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 like streams in Agda which are typically infinite data structures. Coinductive occurences must be labelled with ∞ and coninductive types do not need to terminate but has to be productive. It is often used in conjunction with lazy evaluation. [17]

With these helpful features, Agda is a very powerful proof assisstant. It does not magically prove theorems for people, but it really helps mathematicians and computer scientists to do formalised reasoning with verification by high-performance computers.

2.2.1 basic syntax

to help the reader understand the Agda code in text

To understand the code in this thesis, I will introduce some basic types and syntax of Agda.

First of all, like other languages, we use "=" for function definition rather than propositional equality type former which is "≡". This is inconsistent with our conventional choices of symbols in articles, but it follows the conventions in Haskell and other programming languages that "=" is used for definition.

Different to Haskell, we use single colon : for typing judgement, for example a:A means that a is of type A.

We have universe levels parameters in a lot of definitions which makes code looks unnecessarily cumbersome. We will follow the **typical ambiguity** in this thesis which says that we write A:Set for A:Set$a$ and Set:Set which stands for Set$i$:Set$(i+1)$.

The underscore marks the spaces for the explicit arguments in non-prefix operators.

2.2.2 Identity Type

Identity type is the type introduced by Martin-Löf to encode the propositional equality for definitionally equal terms [10]. For any two terms of $\mathsf{a}\,\mathsf{b} : \mathsf{A}$, we have the type $\mathsf{Id}(\mathsf{A}, \mathsf{a}, \mathsf{b})$ which is inhabitted when a and b are definitionally equal. Here we use an alternative equivalent version named after Paulin-Mohring which is parameterized with the left side of the identity.

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

In Agda eliminators are not automatically derived for the types defined. Instead we have pattern matching generally which is sometimes stronger than eliminators. As long as we pattern match on a variable of an identity type with the unique inhabitant refl, all occurences of both variables become the same. It is stronger and it provides the eliminator J.

```
J : (A : Set)(a : A) → (P : (b : A) → a ≡ b → Set)
  → P a refl
  → (b : A)(p : a ≡ b) → P b p
J A .b P m b refl = m
```

2.2.3 Extensionality

In regular mathematics, equality does not only exists between intensionally equal terms. Pointwise equal functions are usually identified and it is called functional extensionality as we mentioned 2.2. Therefore the identity type in intensional type theory is not powerful enough, we need extensionality in intensional type theory.

As Martin Hofmann summarises in [18], there are several related extensional concepts, *Functional extensionality*, *Uniqueness of identity*, *Proof-irrelevance*, *Subset types*, *Propositional extensionality*, *Quotient types*. These notions are not available in currect setting of intensional type theory, but they are worth interpreting to help both Mathematics and programs constructions. However, it only makes sense if the type-checking decidability and terms canonicity are not sacrificed.

Quotient types is one of the most interesting extensional concepts in type theory. It generally enables us to redefine equality on a type with a given equivalence relation so that we have more tools to formalise mathematical objects, like the real numbers.

2.3 Homotopy Type Theory

Homotopy Type Theory is a variant of intensional Martin-Löf type theory which is a new branch developed between theoretical computer science and mathematics. Vladimir Voevodsky found a surprising connection between homotopy theory and type theory [19]. He proposed the univalence axiom, which identifies isomorphic structures, as a univalent foundation for mathematics.

Definition 2.5. *univalence axiom.* for any two types X and Y,

$$X = Y \simeq X \simeq Y$$

holds.

In Homotopy Type Theory, there is an observation that notions in type theory can be interpreted by homotopy-theoretical terms. A type is regarded as a *space* and a term of this type is a *point* of this space. Functions between types are *continous maps* and identity types are usually considered as *paths*. Identity types of identity types are *homotopies*. Although these notions are originally defined with topological bases, we only employ them as homotopical notions on a higher level.

As univalence axiom states, equality is equivalent to equivalence. Acutally it can be seen as an formal acceptance of the ”common sense” in Mathematics that isomorphic structures can be identitfied. The higher structures of the equivalence also allows us to study the different ways of identification. Therefore it is more appropriate to interpret types as higher groupoids. People is trying to implement Homotopy Type Theory in intensional type theory and one possible way is to interpret weak ω-groupoids first in Agda. The author has done some work in this direction which can be found in Chapter 8.

Higher inductive types are another important ingredient of Homotopy Type Theory. It provides us an enriched way to define types, with the paths as constructors of the types as well. A circle $\mathsf{base} : \mathbb{S}^1$ can be *inductively* defined with two constructors,

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

The eliminator for this type has to take into account the path as well.

Homotopy Type Theory does not only help us model type theory with a focus on the equality, but also provides mathematicians type theoretical tools to study homotopy theory.

I will not explain this topic in detail here, I will!!

For further reference, a well-written text book on Homotopy Type Theory which is written by many brilliant mathematicians and computer scientists is available [20].

2.4 Some conventions

What conventions we will follow in this thesis and some commonly used symbols and function

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

2.5 a very short introduction to category theory

Category theory is a very useful tools to formalise mathematical notions, particularly focussing on morphisms which can be functions, relations and trasformations. A category has a collection of objects and one collection of arrows for each pair of objects. A simple finite category can be visualised as a directed diagram but there are also a set of conditions which are callled categorical laws to obey.

The most accessible example is the category of sets. Objects are sets, arrows are functions and all categorical conditions fulfilled. It is also very helpful to formulate type theoretical concepts in a categorical way. Type theory and category theory are closely related, especially Homotopy type theory [1].

[1] http://ncatlab.org/nlab/show/relation+between+type+theory+and+category+theory

Category theory abstracts a lot of similar concepts in different fields and provides a concise language for mathematics for mathematicians.

2.6 Extensional concepts in intensional type theory

2.6.1 Propositional extensionality

2.6.2 Functional extensionality

2.6.3 Hedberg Theorem

2.6.4 Quotient types

2.6.5 Why do we use type theory?

one page explanation of why type theory is useful

It is always debatable to choose set theory or type theory. First of all, it depends on whether do you think in a constructive way: does any proposition can be claimed true only if we have witness? It the answer is yes, then type theory is a better choice. From a computer scientist's point of view, it is more natural to think in a constructive way. Something without a type makes no sense to us because we are not sure what it stands for and how do we use it. The type definition describe the syntax so that some symbol makes sense and the semantic meaning may be revealed from the construction.

There is another question, whether mathematics is a collection of patterns and laws which is observed, or it is a system created and built by people to explain the patterns and laws in the world. I think people prefer the second answer usually accept the type theory more easily, although most people (probably 99.9 percent) prefer the first one. When we learn what is natural numbers, we learn it as "numbers like 1, 2, 3 ,4 and perhaps 0", the commutative law, associate law are axioms because there is no way to prove it if we introduce it in this manner. We are convinced by some examples like "2 + 3 = 3 + 2" and we find it works for most of the cases then we accept it by observations. It is some methods physicians used a lot – to conclude some laws from a number of facts. It is a proper method for physicians because what they research on is world can only be observed. However for mathematics, even though it is applied to the real world, it is a system completely created by people. People used their fingers to count, wrote symbols for results, even though it was very shallow it is obviously a aritificial system. People extend

Type theory is strongly connected with computation theory. Set

Type theory has fewer axioms, simpler model than set theory which has mutual foudations: logic and axioms.

Chapter 3

Quotient Types

should say something of computer science first rather than mathematics maybe, because it is a compu

Quotient types is an important part in dependently-typed programming languages. It makes some type much easier to define and for some other types possibly to define. Also there are a set of quotient properties such that we can lift functions from the base types. I will start to introduce the quotients from mathematical quotients to quotients in programmming languages to give you an overview what quotient types are.

3.1 Quotients in Mathematics

Quotient is a basic notion in mathematics. Usually the first quotient we learn is the result of division. $8 \div 4$ (or $8/4$) gives the result 2 which is called quotient.

At first most mathematicians are only interested in numbers. As long as they start working on other mathematical objects, some more abstract structures, many notions are extended. As a simple case, the product of numbers is extended to the product of vectors and the product of sets.

Similarly, quotient is also extended to other objects, for example the quotient in set theory.

3.1.1 Example : quotient sets

In set theory, we have a similar operation which turns some set into another set but the divisor is not the same kind of object as the dividend. We use equivalence relation to divide a set,

Definition 3.1. Equivalence relation. An equivalence relation is a binary relation which is refleixive, symmetric and transitive.

Intuitively, given any equivalence relation a set is partitioned into some cells, so that the elements equivalent to each other are in the same cell. The cells are called equivalence classes.

Definition 3.2. Equivalence class.

$$[a] = \{x : A \mid a \sim x\} \tag{3.1}$$

The set of these equivalence classes is called the quotient set.

Definition 3.3. Quotient set. Given a set A equipped with an equivalence relation $\sim$, a quotient set is denoted as $A/\sim$ which contains the set of equivalence classes.

$$A/\sim = \{[a] \mid a : A\} \tag{3.2}$$

Not only in set theory, the quotient of some algebraic structures is a common notions in other branches of mathematics. The notion of equivalence relation is extended to spaces, groups, categories and so does the quotient derived using the same construction.

Generally speaking, it describes the collection of equivalent classes of some equivalent relation on sets, spaces or other abstract structures. In type theory, following similar procedure, quotient type is also a conceivable notion.

3.1.2 Quotient types in extensional type theory

In extensional type theory like NuPrl, it is possible to redefine equality type of some types. However there is also some problems about it:

3.1.3 Quotient types in intensional type theory

Quotient types are not available in original intensional type theory. Alternatively we have *setoids* to simulate quotients.

Definition 3.4. *Setoid.* A setoid $(A, \sim) : \mathbf{Set}_1$ is a set [1] $A : \mathbf{Set}$ equipped with an equivalence relation $\sim : A \longrightarrow A \longrightarrow \mathbf{Prop}$.

[1] Setoid could be universe polymorphic.

It contains Carrier for an underlying set, _≈_ for a binary relation on Carrier and a proof that it is an equivalence relation.

In Agda, we define a setoid as

```
record Setoid : Set₁ where
  field
    Carrier    : Set
    _≈_        : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _≈_
```

We can use setoids to represent quotients, just like the quotient 4 can be represented as the pair $(8, 2)$. However several problems arise from this approach.

First of all, originally many operations and types are defined based on sets, which means that we have to redefine all these types and operations for setoids because they are different sorts. It is easy to see if we consider a question: how to represent a quotient if its base type A is already represented by a setoid.

Secondly, from a programming perspective, setoids are unsafe because we have access to the underlying sets [21]. The operations on setoids may not respect the equivalence relation and make no sense.

Therefore, ideally it is better that the object represent a quotient should also be of type **Set**, just as if we divide 8 by 2 we prefer 4 than $(8, 2)$ which makes more sense and can be manipulated uniformly. It is also the case in the other mathematical theories, the base object and the quotient object are of the same sort. So how quotient type should look like in intensional type theory?

Given a setoid $(A, \sim)$, a type $Q : \mathbf{Set}$ can represent the quotient type of this setoid, if it has several laws:

$$\frac{A : \mathbf{Set} \quad \sim : A \to A \to \mathbf{Prop}}{A/\sim : \mathbf{Set}}\ Q - \mathbf{Form}$$

$$\frac{a : A}{[a] : A/\sim}\ Q - \mathbf{Intro}$$

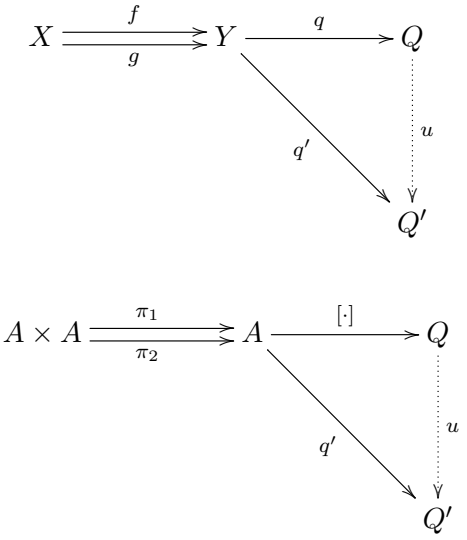
$$\frac{\begin{array}{l} B : A/{\sim} \to \mathbf{Set} \\ f : (a : A) \to B\,[a] \\ (a, b : A) \to (p : a \sim b) \to fa \stackrel{p}{=} fb \end{array}}{\hat{f} : (q : Q) \to B\,q}\; Q - \mathbf{elim}$$

3.2 Categorical intuition

3.2.1 split quotient/coequalizer

Categorically speaking, a quotient is a coequalizer.

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



a split quotient is just a split coequalizer, with an embedding function which finds a representative in each equivalence class.

3.2.2 Adjunction between Sets and Setoids

From a higher point of view, Quotient is a Functor which is left-adjoint to ∇ which is the trivial embedding functor from **Sets** to **Setoids**.

Definition 3.6. $\nabla A = (A, \equiv)$, $Q(B, \sim) = B/{\sim}$

We have following isomorphism for the adjunction of the Quotient Functor and ∇ functor.

$$\frac{B/\sim \ \to A}{(B,\sim) \to \nabla A}$$

3.3 Impredicative encoding of quotient types

In set theory, we have the subset relation such that we can construct equivalence classes and then quotient set. However in intensional type theory, the subset types are also unvailable and equivalence classes cannot be implemented.

The introduction of the univalence axiom for propositions, which is also called propositional extensionality changes this situation.

Definition 3.7. *propositional extensionality.*

$\forall P, Q : \mathbf{Prop}, (P \iff Q) \iff P = Q.$

Voevodsky firsrtly constructs quotients using the following impredicative approach in Homotopy Type Theory using Coq [22].

suppose we have $A : Set$, $\sim\colon A \to A \to Set$.

A quotient type $A/\sim$ is defined as a predicate and a proof that it gives rise to an equivalence class and it is non-empty.

$A/\sim = \Sigma P : A \to \mathbf{Prop}, EqClass(P) \times (\exists x : A, P(x))$

where the equivalence class proof is encoded as

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

The proof part on the right is the $(-1) - truncation$ such that different proofs will not gives different terms for the same equivalence class.

question of where shall we apply pe with the proposional extensionality we can prove that

Theorem 3.8. *Given $(P, prf) : A/\sim$, all proofs of $\exists x : A, P(x)$ are equal*

Proof. Given any two proofs of $\exists x : A, P(x)$ written as (x, px) and (y, py), apply the $EqClass(P)$ to (x, y, px, py) we know that $x \sim y$. Hence the truncation of □

Moreover we have another suprising theorem:

Theorem 3.9. *if we have propositional univalence, we can prove that all quotients are effective.*

Proof. Suppose we have a set A with equivalence relation $\sim$.

Given $a : A$, and a predicate $P : A \to Prop$ defined as

$$Px = a \sim x$$

With the lifting operator for quotient types, we have a lifted version of P such that

$$\hat{P}[x] = Px$$

Suppose we have a premise $[a] = [b]$, it is true that

$$[a] = [b]$$

and then

$$Pa = \hat{P}[a] = \hat{P}[b] = Pb$$

which is just

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

with propositional univalence, we know that they are logically equivalent

$$a \sim a \iff a \sim b$$

since $a \sim a$ is the reflexivity which is true trivially,

$$a \sim b$$

is also true.

Therefore we have a proof that $[a] = [b] -> a \sim b$ which means that the quotient is effective. □

3.3.1 Functional extensionality and quotient types

As we have mentioned before, in intensional type theory propositional equality $Id(A, a, b)$ is inhabited if and only if a and b are definitionally equal terms. The Agda definition could be written as

However the equality of functions are not only judged by definitions. Functions are usually viewed extensionally as black boxes. If two functions pointwise generate the same outputs for the same inputs, they are equivalent even though their definitions may differ. This is called functional extensionality which is not inhabited [1] in original intensional type theory and can be expressed as following,

given two types A and B, and two functions $f, g : A \longrightarrow B$,

$$Ext = \forall\, x\colon A, fx = gx \longrightarrow f = g$$

The problem seems easy to solve by just adding a constant $ext : Ext$ to intensional type theory as following codes in Agda

However, postulating something could lead to inconsistence. If we postulate Ext, then theory is no longer adequate, which means it is possible to define irreducible terms. It can be easily verified in Agda through formalising a non-canonical term for a natural number by an eliminator of intensional equality.

Using the eliminator |J| [2] of the |Id A a b| :

we can construct an irreducible term of natural number as

With this term, we can construct irreducible terms of any type A by a mapping $f : \mathbb{N} \to A$. This will destroy some good features of intensional type theory since it could leads to nonterminating programs.

Altenkirch investigates this issue and gives a solution in [1]. He proposes an extension of intensional type theory by a universe of propositions **Prop** in which all proofs of same propositions are definitionally equal, namely the theory is proof irrelevant. At the same time, a setoid model where types are interpreted by a type and an equivalence relation acts as the metatheory and η-rules for Π-types and Σ-types hold in the metatheory. The

[2] It is originally used by Martin-Löf [10] and a good explanation could be found in [23]

extended type theory generated from the metatheory is decidable and adequate, Ext is inhabited and it permits large elimination (defining a dependent type by recursion). Within this type theory, introduction of quotient types is straightforward. The set of functions are naturally quotient types, the hidden information is the definition of the functions and the equivalence relation is the functional extensionality.

There are more problems concerning quotient types and most of them are related to equality. One of the main problems is how to lift the functions for base types to the ones for quotient types. Only functions respecting the equivalence relation can be lifted. Even in extensional type theory, the implementation of quotient types does not stop at replacing equality of the types. We will discuss these in next section.

3.4 Example of Quotients

The introduction of quotient types is very helpful. Many types can be defined using quotient types, some of them can only be defined with quotient types, such as real numbers (the reason will be covered in).

quotient groups, quotient space, partiality monad.

3.5 literature review

In [24], Mendler et al. have firstly 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, every type comes with its own equality relation, so the quotient operator can be seen as a way of redefining equality in a type. But it is not all about building new types. They also discuss problems that arise from defining functions on the new type which can be illustrated using a simple example.

Assume the base type is A and the new equivalence relation is E, the new type can be formed as $A//E$.

When we want to define a function $f : A//E \longrightarrow Bool$, $f\,a \neq f\,b$ may exists for $a, b : A$ such that $E\,a\,b$. 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 said to be well-defined [24] on the new type only if it respects the equivalence relation E, namely

$$\forall\, a\, b : A, E\, a\, b \longrightarrow f\, a = f\, b$$

We call this *soundness* property in [21].

After the introduction of quotient types, Mendler further investigates this topic from a categorical perspective in [25]. He uses the correspondence between quotient types in Martin-Löf type theory and coequalizers in a category of types to define a notion called *squash types*, which is further discussed by Nogin [26].

To add quotient types to Martin-Löf type theory, Hofmann proposes three models for quotient types in his PhD thesis [18]. 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 setoids rather than sets. Types without a specific equivalence relation can be seen as setoids with the basic intensional equality. This is similar to extensional type theory in some sense. The second one is groupoid model which solves some problems but it is not definable in intensional type theory. He also proposes a third model to combine the advantages of the first two models, but it also has some disadvantages. Later in [27] he gives a simple model in which we have type dependency only at the propositional level, he also shows that extensional Type Theory is conservative over intensional type theory extended with quotient types and a universe [28].

Nogin [26] considers a modular approach to axiomatizing the same quotient types in NuPRL as well. Despite the ease of constructing new types from base types, he also 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.

Homeier [29] 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 [30] 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. However not all quotient types have normal forms. Normalised

types are proper subsets of quotient types, because we can easily recover a quotient type from a normalised type as below

Barthe and Geuvers [31] 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 β-equivalence is replaced by a set of β-conversion rules. Congruence types can be treated as an alternative to the pattern matching introduced in [32]. The main purpose of introducing congruence types is to solve problems in term rewriting systems rather than to implement quotient types.

Barthe and Capretta [33] 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 obtain quotients with different kinds of setoids, especially the ones from partial setoids are difficult to define because the lack of reflexivity.

Abbott, Altenkirch et al. [34] 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 [22] implements quotients in Coq based on a set of axioms of Homotopy Type Theory. It is based on the groupoid model for intensional type theory where isomorphisms are equalities. He firstly implement equivalence class and use it to implement quotients which is an analogy to the construction of quotient sets in set theory.

Chapter 4

Definable Quotients

1. Distributivity 2. Rational numbers 3. other definable quotients

Quotient types may be necessary for defining some types in intensional type theory, but it is not always the case. The set of integers and the set of rational numbers can be defined without quotients types, but there are more properties revealed if we view them as quotients. However there are some good properties if we relate them with the base types and equivalence relation, for example we can lift functions and their properties from base types to quotient types. Moreover, if the base types are simpler to manipulate, it is worthwhile using the base type to define functions and reasoning and then lifting them. We can achieve more convenience by manipulating base types and then lifting the operators and propositions according to the relation between quotient types and base types. The main things we are going to discuss in this chapter is called definable quotient structures which does not require quotient types to be added into the intensional type theory.

In this Chapter we will show this using one of the examples, the set of integers. Some of the work is is conducted by Thorsten Altenkirch, Thomas Anberrée and the author together, and summarised in [21] .

4.1 Integers

From the usual symbols to represent integers, we can easily figure out one inductive definition for integers,

```
data ℤ : Set where
```

```
+_ : ℕ → ℤ
zero : ℤ
-_ : ℕ → ℤ
```

However we face a trade-off: three different representation for zero or to use code $+0$ for number $+1$. Usually the principle is to not losing canonicity because it requires unnecessary checking for whether some functions respect the equivalence or not. Therefore, the second choice makes more sense and we refine it a bit as:

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

This is better, but in practice it is expected to have more cases if we use pattern matching. Every time we use pattern matching, a case will be split into three. This becomes worse and worse when we have mutiple integer arguments and we have to do case analysis on all of them. A simple refinement is combining the first two constructors:

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

This is the most proper version we decided to use for the set of integers. It is inductively defined and is readable because it is just an intepretation of the usual symbols for integers in regular mathematics.

Usually we believe that the reason of inventing integers is the lack of symbols to represent the results of subtraction between two natural numbers. Integers are used to represent these results, and vice versa, every integer can be represented as a pair of natural numbers and the choice is not unique. For example, from the equation $1 - 4 = -3$, it is clear that the integer -3 can be represented the pair $(1, 4)$. Therefore we can use the paired natural numbers as an alternative definition for integers.

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

However since there are different pairs for one integer, we have to quotient it with an equivalence relation. For any two pairs of natural numbers (n_1, n_2) and (n_3, n_4), we know they represent the same integer if

$$n_1 - n_2 = n_3 - n_4$$

Technically, this does not work because the subtraction defined for natural numbers only returns zero if the pair is for negative number. We only need to do some small modifications:

$$n_1 + n_4 = n_3 + n_2$$

This helps us define a relation but it is not enough. This is an equation in mathamtics, but in Type Theory we have to prove that it is an equivalence relation, namely, it is reflexive, symmetric and transitive.

Combining the carrier (the pair of natural numbers), the equivalence relation and its proof, we have a setoid.

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

Since the set of integers is definable as we discussed before, they can be seen as the normal forms of the equivalent classes. The normalisation function can be defined as follows:

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

The function should be proved well-defined on the setoid, namely it has to respect the equivalence relation. We call it soundness here. It is not trivial but easy to observe that the function is sound[1].

A setoid and a function respects this equivalence (not necessary to be a normalisation function) constitute a prequotient.

Definition 4.1. Prequotient.

Given a setoid $(A, \sim)$, a *prequotient* $(Q, [\_], \text{sound})$ over that setoid consists in

1. a set Q,
2. a function $[\_] : A \longrightarrow Q$,
3. a proof *sound* that the function $[\_]$ is compatible with the relation $\sim$, that is

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

Prequotient only includes the formalisation rules and introduction rules. To complete a *quotient*, we also need the elimination rule added into such a prequotient

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

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

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

This eliminator is also called dependent lifting function because it actually lifts a function which is well-defined on the setoid to a function defined on the quotient type. The result type is also dependent on the quotient type. There is an equivalent definition given by Martin Hofmann which has a non-dependent eliminator with an induction principle instead.

$$\text{lift} : (f : A \longrightarrow B) \longrightarrow (\forall a, b \cdot a \sim b \longrightarrow f\ a \equiv f\ b) \longrightarrow (Q \longrightarrow B)$$

[1] the formal proof can be found in appendix (we cheat a bit by defining embedding function to make it simpler)

Suppose B is a predicate,

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

However, it is oberservable that given any non-empty set Q, all constant functions fit in this definition. Any element of Q has to be mapped from at most one equivalence class of the setoid $(A, \sim)$. This property is called *exact* here

5. $exact : (\forall a, b : A) \longrightarrow\!\!\!\!\!\rightarrow [a] \equiv [b] \longrightarrow\!\!\!\!\!\rightarrow a \sim b$.

The quotient is exact if exactly one equivalence class corresponds to an element of Q.

We already know that the integer is definable and it is plausible to find a representative in each equivalence classes. Since we treat elements of Q as the name for the equivalence classes, the selection function can be defined as an embedding function from the quotient type Q to base type A. This is an alternative and more flexible way to eliminate the quotient type Q and if a prequotient $(Q, [\,\cdot\,], \text{sound})$ on a setoid $(A, \sim)$ has an embedding function which is specified as

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

Then it is a *definable quotient*. Composing the "normalisation" function $[\,\cdot\,]$ with the embedding function, we obtain the real normalisation function. A definable quotient is an *exact* quotient which is proved in [21].

Operations For a definable quotient, we can lift an operation by mixing the normalisation and embedding functions. For example, given an unary operator

$$\begin{aligned} &\mathsf{lift}_1 : (op : \mathbb{Z}_0 \to \mathbb{Z}_0) \to \mathbb{Z} \to \mathbb{Z} \\ &\mathsf{lift}_1\ op = [\_] \circ op \circ \ulcorner \_ \urcorner \end{aligned}$$

To lift binary or n-ary operators, we only need to apply the operator to the representative for each "equivalence class" and "normalise" the result so that it becomes a function defined on the set of "equivalence classes".

But there is a unavoidable problem: not all operations defined on $\mathbb{Z}_0$ is defind on the setoid, namely respect the equivalence relation. Therefore it is reasonable to verify if the function is well-defined on the setoid:

$$a \sim b \to op\, a \sim op\, b$$

We will show how to define the addition for the quotient integers. Given two numbers (a_1, b_1) and (a_2, b_2) We only need to add them pair-wisely together, and it can be verified easily because we know that

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

The verification is not necessary here but should be important in other cases.

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

Properties We can also define the ring of $\mathbb{Z}$. It contains a lot of properties to prove which are seemed as axioms in classic mathematics. In constructive mathematics, the only axioms for integers are the constructors and the elimination rules.

As what we have menetioned, even though the definition of integers only has two constructors, it gradually increase the difficulty of proving when doing case analyses on more and more integers. One example is the proving of distributivity.

An efficient utilization of quotient structure: the proving of distributivity One of the most important motivations of using setoid integers is that the setoid definition reduces the complexity of programs involving integers. We have shown it is simpler to define some operators, but it will be much more evident when proving properties of the ring of integers.

Most simple laws of the ring of integers are not as unbearably complicated as the distributivity laws. An attempt of the right distributivity for $\mathbb{Z}$ $((y+z) \times x = y \times x + z \times x)$ is shown as below. The multiplication is not defined with pattern matching, but in the arithmetic approach.

```
_ℤ*_ : ℤ → ℤ → ℤ
```

```
i ℤ* j = sign i S* sign j ◁ | i | ℕ* | j |
```

The first idea which I came up with is to use the right distributivity law of natural numbers. It is observable that if all three variables have the same signs, it is easy to apply the right distributivity law of natural numbers. We can relax the constraint a bit, only y and z have the same symbol, it is still plausible. To prove this part 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
```

The following is a partial definition with the first case that

```
dist^r : _ℤ*_ DistributesOver^r _ℤ+_
dist^r x y z with sign y S≟ sign z
dist^r x y z  | yes p
  rewrite lem1 y z p | p
                     | lem2 y z p =
  trans (cong (λ n → sign z S* sign x ◃ n)
  (distrib^r | x | | y | (| z |)))
  (lem3 (| y | ℕ* | x |) (| z | ℕ* | x |)
  (sign z S* sign x))
dist^r x y z | no ¬p = ...
```

If y and z have different signs, it is impossible to apply the right distributivity law for natural numbers. We have to prove it from scratch. Even though it seems that there are only two cases to prove, it is conceivable that how many lemmas we need as prerequisites. The proving is as complicated as using pattern matching on each variable. This is not the best solution we want. We should benefit more from the observation that any equation of integers can be turned into equation of natural numbers**??**.

However, if we prove the laws for quotient integers, it is much simpler since there is only one case to prove.

However, if we prove it for the quotient integers, it is much easier. In fact, it is in generally automatically provable. Since the equality of any two quotient integers is essentially the equality of two natural numbers after normalising. To prove the distributivity, the simpliest way is to use semiring solver for natural numbers. DistributesOver^l means that the distributivity of the first operators over the second one.

Remark 4.2. 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 "Proving Equalities in a Commutative Ring Done Right in Coq" by Grégoire and Mahboubi [35].

```
dist^l :  _*_ DistributesOver^l _+_
dist^l (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
```

The utilization of ring solver can be simplified even further by adopting "reflection". It helps us quote the type of the goal so that we can define a function that automatically do it without explicitly providing the equations. There is already some work done by van der Walt [36]. It can be seen as an analogy of the "ring" tactic from Coq.

The main drawback of this method is the type verification of the terms automatically generated requires more computations than the terms we manually construct. The optimization of Agda already shows a big improvement in this technical efficiency issue. However it will affect other functions that use this proof. It may heavily slow the type checking. Therefore although it is still very convenient to use the ring solver to prove any proposition for the quotient integers, we decide to prove commonly used properties for the commutative ring of the integers by hand. Luckily it is still much simpler than the ones for the set of integers $\mathbb{Z}$.

This is a good example of how the definable quotient structure helps simplifying definiting and proving about new types based on existing functions and theorems.

4.2 Rational numbers

The quotient of rational numbers is better known than the previous quotient of integers. We usually write two integers m and n (n is not zero) in fractional form $\frac{m}{n}$ to represent a rational number. Alternatively we can use an integer and a positive natural number such that it is simpler to exclude 0 in the denominator. Two fractions are equal if they are reduced to the same irreducible term. If the numerator and denominator of a fraction are coprime, it is said to be an irreducible fraction. Based on this observation, it is naturally to form a definable quotient, where the base type is

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

The integer stands for the *numerator* and the natural number is *denominator-1* (We use N for N^+ to avoid invalid fractions from construction rather than from zero test)

In Agda, to make the terms more meaningful we define it as

```
data ℚ₀ : Set where
```

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

In mathematics, to judge the equality of two fractions, it is easier to conduct the following conversion,

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

Therefore the equivalence relation can be defined as,

```
_*_ : ℤ → ℕ → ℤ
(+ x) * d = + (x ℕ* d)
(-suc x) * 0 = + 0
(-suc x) * suc d = -suc (x ℕ+ suc x ℕ* d)

_∼_ : Rel ℚ₀ _
(n1 /suc d1) ∼ (n2 /suc d2) = n1 * suc d2 ≡ n2 * suc d1
```

The normal form of rational numbers, namely the quotient type in this quotient is the set of irreducible fractions. We only need to add a restriction that the numerator and denominator is coprime,

$$\mathbb{Q} = \Sigma(n\colon \mathbb{Z}).\Sigma(d\colon \mathbb{N}).\mathsf{coprime}\, n\, (d+1)$$

It can be defined as follows which is available in standard library,

```
record ℚ : Set where
  field
    numerator   : ℤ
    denominator-1 : ℕ
    isCoprime   : True (C.coprime? | numerator | (suc denominator-1))

  denominator : ℤ
  denominator = + suc denominator-1

  coprime : Coprime numerator denominator
```

```
coprime = toWitness isCoprime
```

The normalisation function is an implementation of the reducing process. But first we need to the |gcd| function which calculates the greatest common divisor can help us reduce the fraction and give us the proof of coprime. First we need to define the conversion from the results of GCD to normal rational numbers (the full definition can be found in Appendix Appendix A),

```
GCD′→ℚ : ∀ x y di → y ≢ 0 → C.GCD′ x y di → ℚ
```

To normalise a fractional, we split it into 3 cases with respect to the numerator. The idea is to calculate the "gcd" and then use the above function to get the normalised rational number.

```
[_] : ℚ₀ → ℚ
[ (+ 0) /suc d ] = ℤ.+_ 0 ÷ 1
[ (+ (suc n)) /suc d ] with gcd (suc n) (suc d)
[ (+ suc n) /suc d ] | di , g = GCD′→ℚ (suc n) (suc d) di (λ ()) (C.gcd-gcd′ g)
```

The embedding function is simple. We only need to forget the coprime proof in the normal form,

```
⌜_⌝ : ℚ → ℚ₀
⌜ x ⌝ = (ℤcon (ℚ.numerator x)) /suc (ℚ.denominator-1 x)
```

Similarly, we are able to construct the setoid, the prequotient and then the definable quotient of rational numbers. We can benefit from the ease of defining operators and proving theorems on setoids while still using the normal form of rational numbers, the lifted operators and properties which are safer.

The same approach works here as well. Since we can easily embed the natural numbers into integers, the equations of the quotient rational numbers are degraded to equations of the integers. The commutative ring of integers also enable us to prove all properties of rational numbers automatically.

Chapter 5

Real numbers and other undefinable quotients

Martin Escardo's http://www.cs.bham.ac.uk/~mhe/agda/FailureOfTotalSeparatedness.html shou

The previous quotient types are all definable in intensional type theory, so we can construct the definable quotients for them. However, there are some types undefinable in intensional type theory. The set of real numbers $\mathbb{R}$ has been proved to be undefinable in [21].

5.1 Real numbers

We have several choices to represent real numbers. We choose Cauchy sequences of rational numbers to represent real numbers [37].

$$\mathbb{R}_0 = \{s : \mathbb{N} \longrightarrow \mathbb{Q} \mid \forall \varepsilon : \mathbb{Q}, \varepsilon > 0 \longrightarrow \exists m : \mathbb{N}, \forall i : \mathbb{N}, i > m \longrightarrow |s_i - s_m| < \varepsilon\}$$

It is implementable in Type Theory, but there is a problem of the choice of type of the second part of the cauchy sequence, namely the property that it is a cauchy sequence. Do we distinguish the same sequences with different proofs? Logically speaking we should not. It means that we need to truncate it to proposition but we will lose the important tool to guess what the real number is. To avoid this problem, we could use an alternative equivalent definition which is a subset of $\mathbb{R}_0$:

$$\mathbb{R}'_0 = \left\{ f : \mathbb{N} \longrightarrow \mathbb{Q} \mid \forall k : \mathbb{N}, \forall m, n > k, \longrightarrow |f_m - f_n| < \frac{1}{k} \right\}$$

With the definition, the condition part is propositional and we can guess the number by applying any number k to the sequence and we know the interval where it should be located.

Different cauchy sequences can represent the same number. Therefore an equivalence relation[1] is expected. In mathematics two Cauchy sequences $\mathbb{R}_0$ are said to be equal if their pointwise difference converges to zero,

$$r \sim s = \forall \varepsilon : \mathbb{Q}, \varepsilon > 0 \longrightarrow \exists m : \mathbb{N}, \forall i : \mathbb{N}, i > m \longrightarrow |r_i - s_i| < \varepsilon$$

5.1.1 Non-normalizability of Cauchy Sequences

To prove that it is impossible to give a full definable quotient structure of real numbers with the setoid of cauchy sequences, we could show it by proving that it is impossible to define a normalisation function for the cauchy sequences.

Definition 5.1. We say that a quotient structure $A/\sim$ is definable via a normalisation, if we have a normalisation function

$$nf : A \longrightarrow A \tag{5.1}$$

with the property that it respects $\sim$

$$p : \Pi_{c_1,c_2:A} c_1 \sim c_2 \longrightarrow nf(c_1) = nf(c_2). \tag{5.2}$$

such that

$$q : \Pi_{c:A} nf(c) \sim c. \tag{5.3}$$

It is equivalent to say that we have a definable quotient structure in the sense of [21], because we can form the set of equivalence classes as

$$Q :\equiv \Sigma_{c:\mathbb{R}_0} nf(c) = c$$

where the second part is propositional, and the "normalisation" function can be defined as

$$[c] :\equiv nf(c), p(q)$$

[1] The Agda version is in Appendix

and the embedding function is just the first projection. The properties can be verified easily.

In the other way around, the true normalisation function is just

$$n := emb \circ [\_]$$

and the properties hold as well.

We have made an attempt in the original version of our [21] draft, but there is something important problem pointed out by Martin Escardo. Laterly, Nicolai Kraus suggests to fix the proof by proving it as a meta-theoretical property. We will show an adaption of his proof here.

Some preliminaries In fact the proof is mainly conducted using topological tools. The following definitions are helpful for someone who are not so familiar with topological concepts.

Definition 5.2. Metric space. In mathematics, a metric space is a set where a notion of distance (called a metric) between elements of the set is defined.

A metric space is an ordered pair (M, d) where M is a set and d is a metric on M:

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

We usually define a standard topological structure for discrete types.

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

We use a slightly different definition of cauchy sequences of rational numbers here:

Definition 5.3. We call a function $f : \mathbb{N}^+ \longrightarrow \mathbb{Q}$[2] a *Cauchy Sequence* if it satisfies

$$\mathsf{isCauchy}(f) :\equiv \forall(n : \mathbb{N}^+).\, \forall(m : \mathbb{N}^+).\, m > n \longrightarrow |f(n) - f(m)| < \frac{1}{n}. \tag{5.4}$$

The type of Cauchy Sequences is thus

$$\mathbb{R}_0 :\equiv \Sigma_{f:\mathbb{N}^+ \longrightarrow \mathbb{Q}} \mathsf{isCauchy}(f).$$

And the standard metric space for the sequences $\mathbb{N}^+ \to \mathbb{Q}$ is defined by the distance function

$$g(f_1, f_2) = 2^{-\mathsf{inf}\{k \in \mathbb{N} \mid f_1(k) \neq f_2(k)\}} \tag{5.5}$$

Among all these standard metric spaces, It is a folklore that all definable functions are continous.

Theorem 5.4. *All definable functions are continous.*

Let us introduce the following auxiliary definition:

Definition 5.5. For a sequence $f : \mathbb{N} \longrightarrow \mathbb{Q}$, we say that f is *Cauchy with factor* k, written as $\mathsf{isCauchy}_k$, for some $k \in \mathbb{Q}$, $k > 0$, if

$$\mathsf{isCauchy}_k(f) \;:\equiv\; \forall(n\, m : \mathbb{N}).\, m > n \longrightarrow |f(n) - f(m)| < \frac{1}{k \cdot n}. \tag{5.6}$$

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

Remark 5.6. If we claim a function f is defined on $\mathbb{R}_0$ that respects $\sim$, it means that we have a proof

$$p : \Pi_{c_1, c_2 : \mathbb{R}_0} c_1 \sim c_2 \longrightarrow f(c_1) = f(c_2). \tag{5.7}$$

Now we have enough tools to prove the following proposition.

Proposition 5.7. ***"$\mathbb{R}_0/\sim$" is connected.*** *It means that any continuous function f*

$$f : \mathbb{R}_0 \longrightarrow \mathbf{2} \tag{5.8}$$

that respects $\sim$ is constant. We prove that it is impossible to find $c_1, c_2 : \mathbb{R}_0$ such that $f(c_1) \neq f(c_2)$ meta-theoretically, instead of deriving a proof term of this in type theory.

[2] we use $\mathbb{N}^+$ instead of $\mathbb{N}$ because n must be positive in $\frac{1}{n}$

The definability of function implies that it is a continous function 5.4 between the standard metric spaces for $\mathbb{R}_0$ *and* $\mathbf{2}$[3].

Proof. The general idea is to interpret our definition in classical mathematics, assume we have a non-constant function and deduce a contradiction.

Consider the "naive" set model (with "classical standard mathematics" as meta-theory). This clearly works if we are in a minimalisitic type theory with Π, Σ, W, $=$, $\mathbb{N}$; however, if we restrict ourselves to the types in the lowest universe of homotopy type theory (which is enough), it also works for HoTT. We use $[\![\cdot]\!]$ as an interpreation function; for example, we write $\mathbb{R}$ for the field of real numbers which can be defined as $[\![\mathbb{R}_0]\!]/[\![\sim]\!]$. By abuse of notation, we write $[\![\mathbb{R}_0]\!]$ for the set of Cauchy sequences in the model that fulfill the Cauchy condition, without the actual proof thereof. This is justified as this property is propositional.

For readability, we use the symbol $=$ for equality in the theory as well in the model, and we do not use the semantic brackets for natural numbers such as 2 or 4. In the model, we use $\overline{\cdot} : [\![\mathbb{R}_0]\!] \longrightarrow \mathbb{R}$ as the function that gives us the limit of a Cauchy sequence (Not all functions in the "naive" set model have to be continuous). Thus, for $r : \mathbb{R}_0$, we write $\overline{[\![r]\!]} \in \mathbb{R}$ for the real number it represents.

Assume f, p are given. We prove that $[\![f]\!] : [\![\mathbb{R}_0]\!] \longrightarrow [\![\mathbf{2}]\!]$ is constant in the model, which implies the statement of Proposition 5.7.

Thus, assume $[\![f]\!]$ is non-constant, there are $c_1, c_2 : [\![\mathbb{R}_0]\!]$ with $[\![f]\!](c_1) \neq [\![f]\!](c_2)$.

Define

$$m_1 := \mathsf{sup}\{\overline{d} \in \mathbb{R} \mid d \in [\![\mathbb{R}_0]\!], \overline{d} \leq \mathsf{max}(\overline{c_1}, \overline{c_2}), [\![f]\!](d) = [\![1_{\mathbf{2}}]\!]\} \tag{5.9}$$

$$m_2 := \mathsf{sup}\{\overline{d} \in \mathbb{R} \mid d \in [\![\mathbb{R}_0]\!], \overline{d} \leq \mathsf{max}(\overline{c_1}, \overline{c_2}), [\![f]\!](d) = [\![0_{\mathbf{2}}]\!]\} \tag{5.10}$$

(note that one of these two necessarily has to be $\overline{c_1}$ or $\overline{c_2}$, whichever is bigger). Set $m := \mathsf{min}(m_1, m_2)$, and we can observe that in *every* neighborhood U of m, given any t, we can always find another point $x \in U$ such that $x = \overline{e}$ (for some e) with $[\![f]\!](e) \neq [\![f]\!](t)$.

Let $c \in [\![\mathbb{R}_0]\!]$ be a Cauchy sequence such that $\overline{c}$ is equal to m. We may assume that c satisfies the condition $[\![\mathsf{isCauchy}_5]\!]$.[4]

As f (and thereby $[\![f]\!]$) is continuous (remember the metric spaces), there exists $n_0 \in [\![\mathbb{N}]\!]$ such that for any Cauchy sequence $c' \in [\![\mathbb{R}_0]\!]$, if the first n_0 sequence elements of c'

[3] The metric of $\mathbb{R}$ comes from the first component. Technically, if $\mathbb{R}_0$ is defined by **??**, this would not make it a metric space (as the distance between non-equal elements could be 0); however, this would not matter, and for our definition, there is no problem anyway.

[4] the factor 5 is chosen due to the need of a later proof.

coincide with those of c (namely the distance $g(c,c') = 2^{-\mathsf{inf}\{k\in\mathbb{N} \mid c(k)\neq c'(k)\}} \leq 2^{-n_0}$), then $[\![f]\!](c') = [\![f]\!](c)$. Write $U \subset [\![\mathbb{R}_0]\!]$ for the set of Cauchy sequences which fulfill this property, and $\overline{U} \coloneqq \{\overline{d} \mid d \in U\}$ for the set of reals that U corresponds to.

We claim that $\overline{U}$ is a neighborhood of m. More precisely, we prove: The interval $I \coloneqq (m-\frac{1}{2n_0}, m+\frac{1}{2n_0})$ is contained in $\overline{U}$. Let $x \in \mathbb{R}$ be in I. There is a sequence $t : [\![\mathbb{N}\longrightarrow\mathbb{Q}]\!]$ such that $[\![\mathsf{isCauchy}_{5n_0}]\!](t)$ and $\overline{t} = x$. Let us now "concatenate" the first n_0 elements of the sequence c with t, that is, define

$$g : [\![\mathbb{N} \longrightarrow \mathbb{Q}]\!] \tag{5.11}$$

$$g(n) = \begin{cases} c(n) & \text{if } n \leq n_0 \\ t(n-n_0) & \text{else.} \end{cases} \tag{5.12}$$

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

$$|g(n) - g(m)| < \frac{1}{n}. \tag{5.13}$$

Calculate

$$\begin{aligned}
& |g(n) - g(m)| && (5.14)\\
= \;& |c(n) - t(m-n_0)| && (5.15)\\
= \;& |c(n) - \overline{c} + \overline{c} - \overline{t} + \overline{t} - t(m-n_0)| && (5.16)\\
\leq \;& |c(n) - \overline{c}| + |\overline{c} - \overline{t}| + |\overline{t} - t(m-n_0)| && (5.17)\\
\leq \;& \frac{1}{5n} + \frac{1}{2n_0} + \frac{1}{5n_0 \cdot (m-n_0)} && (5.18)\\
\leq \;& \frac{1}{5n} + \frac{1}{2n} + \frac{1}{5n_0} && (5.19)\\
< \;& \frac{1}{n}. && (5.20)
\end{aligned}$$

From the continuity property of f and the definition of g we know that $[\![f]\!](g) = [\![f]\!](c)$. Clearly, $\overline{g} = \overline{t} = x \in I$. Therefore *all* $s \in [\![\mathbb{R}_0]\!]$ with $\overline{s} \in I$, we can use the "concatenation" approach to find a g satisfies $s\,[\![\sim]\!]\,g$ (namely $\overline{s} = \overline{g}$), and by the condition that f (and thereby $[\![f]\!]$) repects $\sim$, we can conclude that $[\![f]\!](s) = [\![f]\!](g) = [\![f]\!](c)$.

However, as we have seen, in *every* neighborhood of m, and thus in particular in $(m - \frac{1}{2n_0}, m + \frac{1}{2n_0})$, there is an x such that $x = \overline{e}$ (for some e) with $[\![f]\!](e) \neq [\![f]\!](c)$, in contradiction to the just established statement. □

The proposition that "$\mathbb{R}_0/\sim$ is connected" implies the following corollary:

Corollary 5.8. *Any continous function from $\mathbb{R}_0$ to any discrete type that respects $\sim$ is constant.*

Theorem 5.9. *Any continuous endofunction f on $\mathbb{R}_0$ that respects $\sim$ which means*

$$p : \Pi_{c_1,c_2:\mathbb{R}_0} c_1 \sim c_2 \longrightarrow f(c_1) = f(c_2). \tag{5.21}$$

is constant (in the sense of corollary 5.8).

Proof. Assume we have f, p as required.

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

Given a postive natural number $n : [\![\mathbb{N}^+]\!]$, we have a projection function $\pi_n : [\![\mathbb{R}_0]\!] \longrightarrow [\![\mathbb{Q}]\!]$. Define a function $h_n : [\![\mathbb{R}_0]\!] \longrightarrow [\![\mathbb{Q}]\!]$ as

$$h_n(c) :\equiv \pi_n \circ f$$

By corollary 5.8 we know that h_n is constant, hence f is constant everywhere, it is enough to show that f is constant.

□

Corollary 5.10. *There is no definable normalisation function on $\mathbb{R}_0$ in the sense of 5.1*

Corollary 5.11. *$\mathbb{R}_0/\sim$ is not definable in the sense of [21].*

However, it doesn't imply that we cannot define the set of real numbers in minimalisitic type theory with $\Pi, \Sigma, \mathsf{W}, =, \mathbb{N}$. The meaning of definability of real numbers is still not clear enough. To make it more precise, we define it as whether there is a type A in TT (minimalisitic type theory) such that its embedding $[\![A]\!]$ in $\mathsf{TT}+\mathsf{Q}$ (type theory extended with quotient types) is isomorphic to $[\![\mathbb{R}]\!]_0/[\![\sim]\!]$ (where it is a valid type). We conjecture that it is still not definable.

Proof. Assume the set of real numbers is definable, we have a type A and its embedding in $\mathsf{TT}+\mathsf{Q}$ is $[\![A]\!]$. It also gives us a normalisation function and a representative function between $[\![\mathbb{R}]\!]_0$ and $[\![A]\!]$.

P(T) -> Connected (T) -> Contractible(T), $\mathbb{R}/\sim$ is connected but not contractible?

□

5.1.2 Cauchy completeness of the Cauchy reals

expand Cauchy completeness of Cauchy reals: it should rely on the axiom of countable choice

Whether our definition of Cauchy sequence is Cauchy complete? In other words, is there a representative Cauchy sequence as a limit for every equivalence class? The answer is no.

In the HoTT book [20], an alternative definition is used instead which is called cauchy approximation. Because every approximation has been proven to have a limit, it is Cauchy complete. The definition uses the higher inductive types which will be discussed in later Chapter 7.

5.2 Multisets

Definition 5.12. Multiset. A multiset (or bag) is a set without the constraint that there is no repetitive elements.

A set is just a special case of multiset when the *multiplicity*(the number of the occurences) of every element is one. Multisets (or bags) are believed to be used in ancient times, but it is only studied by mathematicans from 20th century.

In set theory, a multiset is defined as a pair of a set A and a occurences counting function $m : A \to \mathbb{N}$.

However in type theory, the set-theoretical *"set"* is not a primitive notion and we need to define multisets in a type-theoretic way. In type theoretical language, a multiset can be seen as a unordered list. It can be encoded as lists which identifies permutations. To make it simpler we use length-explict list *Vector* ("Vector A 3" stands for a list of type A of length 3).

The equivalence relation required is as follows:

Given two lists of type A and has length n, $p\,q : Vector\,A\,n$, they are equivalent if we have an isomorphism between them.

$$p \sim q = \Sigma\phi : Fin\,n \to Fin\,n, \phi\,is\,a\,bijection \wedge \forall x, p_x = q_{\phi(x)}$$

5.3 Partiality monad

Partiality monad is a coninductive type which is available in the standard library of Agda. It is used to encode delayed computation.

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

A non-terminating program can be defined in a coinductive way.

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

We have two equality for Delay type: strong bisimilarity and weak bisimilarity. Two computation are strongly bisimilar if they are the same after the same number of steps delay (there can be infinite steps).

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

We inductively define an operator which states that "x terminates with y" if we write $x))\downarrow y$.

```
infix 4 _↓_

data _↓_ {A : Set} : Delay A → A → Set where
  nowT  : ∀{a} → (now a) ↓ a
  laterT : ∀{d a} → d ↓ a → (later (♯ d)) ↓ a
```

And two computation are weakly bisimilar if they terminates with the same value.

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

The quotient derived from these equivalent relations which represent the set of all computations can also be a good example of undefinable quotient.

Chapter 6

The Setoid Model

Quotient types are one of the extensional concepts in Type Theory [18]. There are several existing intensional models for extensional concepts. The first one we are going to work with is Altenkirch's setoid model. To introduce an extensional propositional equality in intensional type theory, Altenkirch [1] proposes an intensional setoid model with a proof-irrelevant universe of propositions **Prop**.

$$[proof - irr]\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma \vdash p, q : P}{\Gamma \vdash p = q : P} \tag{6.1}$$

It 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 are definitionally equal.

The propositional universe is closed under "Π" and "Σ", namely dependent functions and dependent products.

$$[\Pi - Prop]\frac{\Gamma \vdash A : \mathbf{Set} \qquad \Gamma, x : A \vdash P \in \mathbf{Prop}}{\Gamma \vdash \Pi\, x : A.P} \tag{6.2}$$

$$[\Sigma - Prop]\frac{\Gamma \vdash P : \mathbf{Prop} \qquad \Gamma, x : P \vdash Q \in \mathbf{Prop}}{\Gamma \vdash \Sigma\, x : P.Q} \tag{6.3}$$

It is called a setoid model since types are interpreted as setoids. The solution to introduce the extensional equality is an object type theory defined inside the setoid model which serves as the metatheory. He also proved that the extended type theory generated from

the metatheory is decidable and adequate, functional extensionality is inhabited and it permits large elimination (defining a dependent type by recursion). Within this type theory, introduction of quotient types is straightforward.

This model is different to a setoid model as an E-category, for instance the one introduced by Hofmann [38] . An E-category is a category equipped with an equivalence relation for homsets. To distinguish them, we call this category **E-setoids**. All morphisms of **E-setoids** gives rise to types and they are cartesian closed, namely it is a a locally cartesian closed category (LCCC). Not all morphisms in our category of setoids give rise to types and it is not an LCCC. Every LCCC can serve as a model for categories with families but not every category with families has to be an LCCC.

However this Setoid model is still a model for Type Theory just like the groupoid model which is a generalisation of it. To develop this model of type theory in Agda, we have implemented the categories with families of setoids. We build a category with families of setoids to accommodate the types theory described in [1] so that it is possible to define quotient types following Martin Hofmann's Paper [27]. Only necessary part for the Setoid model will be present here.

6.0.1 An implementation of categories with Families in Agda

Following the work in [**?**], we first define a proof-irrelevant universe of propositions. We name it as **hProp** since **Prop** is a reserved word which can't be used and **hProp** is a notion from Homotopy Type Theory which we will introduce later.

6.0.2 hProp

A proof-irrelvant universe only contains sets with at most one inhabitant.

```
record hProp : Set₁ where
  constructor hp
  field
    prf : Set
    Uni : {p q : prf} → p ≡ q

open hProp public renaming (prf to <_>)
```

We can extract the proof of any propostion $A : hProp$ by using <> and there is always a proof that all inhabitants of it are the same, in other words, if there is any proof of it, the proof is unique. This is not exactly the same as the *Prop* universe in Altenkirch's approach which is judgemental. It is just a judgement whether a set behaves like a *Proposition*. The *hProp* we define above is propositional since we can extract the proof of uniqueness.

We would like to have some basic propositions $\top$ and $\bot$. To distinguish them with the ones for non-proof irrelevant propositions which are already available in Agda library, we add a prime to all similar symbols.

```
⊤' : hProp
⊤' = hp ⊤ refl

⊥' : hProp
⊥' = hp ⊥ (λ {p} → ⊥-elim p)
```

We also want the universal and existential quantifier for *hProp*, namely it is closed under Π-types and Σ-types. The universal quantifier of *hProp* can be axiomitised but we decide to explicitly state that we require the functional extensionality to use this module. The reason is that functional extensionality is actually equivalent to the closure under Π-types.

```
∀' : (A : Set)(P : A → hProp) → hProp
∀' A P = hp ((x : A) → < P x >) (ext (λ x → Uni (P x)))
```

```
Σ' : (P : hProp)(Q : < P > → hProp) → hProp
Σ' P Q = hp (Σ < P > (λ x → < Q x >))
  (λ {p} {q} →
```

```
    sig-eq (Uni P) (Uni (Q (proj₁ q))))
```

Implication and conjuction which are independent ones of them follow simply.

```
_⇒_ : (P Q : hProp) → hProp
P ⇒ Q =  ∀' < P > (λ _ → Q)

_∧_ : (P Q : hProp) → hProp
P ∧ Q = Σ' P (λ _ → Q)
```

As long as we have implication and conjuction, more operators on proposition can be defined, for instances negation and logical equivalence.

```
¬ : hProp → hProp
¬ P = P ⇒ ⊥'

_↔_  : (P Q : hProp) → hProp
P ↔ Q = (P ⇒ Q) ∧ (Q ⇒ P)
```

6.0.3 Category

To define category of setoids we should define category first.

```
record Category : Set where
  constructor CatC
  field
    obj            : Set

    hom            : obj → obj → Set
```

```
id              : ∀ a
                → hom a a

[_⇒_]_∘_        : ∀ a {β} γ
                → hom β γ
                → hom a β
                → hom a γ

isCategory : IsCategory obj hom id [_⇒_]_∘_
```

isCategory contains all the laws for this structure to be a category, for instance the associativity laws for composition.

6.0.4 Category of setoids

Then we could define setoids using **hProp**. An equivalence relation has three properties reflexivity, symmetry and transitivity. Since we have $refl$ here, we call the reflexivity for propositional equality from the library with prefix as $PE.refl$.

```
record IshEquivalence {A : Set}(_≈h_ : A → A → hProp) : Set₁ where
  constructor _,_,_
  field
    refl    : {x : A} → < x ≈h x >
    sym     : {x y : A} → < x ≈h y > → < y ≈h x >
    trans   : {x y z : A} → < x ≈h y > → < y ≈h z > → < x ≈h z >
```

Here we use **hSetoid** as the name because **Setoid** is already used for non-proof-irrelvant setoids in the library. For each setoid, we have a carrier type and an equivalence relation.

```
record hSetoid : Set₁ where
```

```
  constructor _,_,_
  infix 4 _≈h_ _≈_
  field
    Carrier : Set
    _≈h_    : Carrier → Carrier → hProp
    isEquiv : ishEquivalence _≈h_
```

A morphism in this category is a function of the underlying sets which respects the equivalence relation. We don't identify the extensional equal functions in the homsets as in **E-setoids**.

```
record _⇉_ (A B : hSetoid) : Set₁ where
  constructor fn:_resp:_
  field
    fn   : | A | → | B |
    resp : {x y : | A |} →
      [ A ] x ≈ y →
      [ B ] fn x ≈ fn y
```

The definitions of identity morphism and composition are straightforward and the categorical laws hold trivially as follows.

```
id' : {Γ : hSetoid} → Γ ⇉ Γ
id' = record { fn = id; resp = id}

_∘c_ : ∀{Γ Δ Z} → Δ ⇉ Z → Γ ⇉ Δ → Γ ⇉ Z
yz ∘c xy = record
  { fn = [ yz ]fn ∘ [ xy ]fn
  ; resp = [ yz ]resp ∘ [ xy ]resp
  }
```

```
id₁ : ∀ Γ Δ (ch : Γ ⇉ Δ) → ch ∘c id' ≡ ch
id₁ _ _ ch = PE.refl

id₂ : ∀ Γ Δ (ch : Γ ⇉ Δ) → id' ∘c ch ≡ ch
id₂ _ _ ch = PE.refl

comp : ∀ Γ {Δ Φ} Ψ
  (f : Γ ⇉ Δ)
  (g : Δ ⇉ Φ)
  (h : Φ ⇉ Ψ)
  → h ∘c g ∘c f ≡ h ∘c (g ∘c f)
comp _ _ f g h = PE.refl
```

Combined all components we obtain the category of setoids.

```
setoid-Cat : Category
setoid-Cat = CatC hSetoid _⇉_ (λ _ → id') (λ _ _ → _∘c_)
  (IsCatC id₁ id₂ comp)
```

This category has a terminal object which is just the unit set with trivial equality. As a terminal object there is precisely one morphism from every object to it.

```
⊤-setoid : hSetoid
⊤-setoid = record {
  Carrier = ⊤;
  _≈h_    = λ _ _ → ⊤';
  isEquiv = record {
    refl    = tt;
    sym     = λ _ → tt;
    trans   = λ _ _ → tt } }

⋆ : {Δ : hSetoid} → Δ ⇉ ⊤-setoid
```

```
⋆ = record
  { fn = λ _ → tt
  ; resp = λ _ → tt }

unique⋆ : {Δ : hSetoid} → (f : Δ ⇉ ⊤-setoid) → f ≡ ⋆
unique⋆ f = PE.refl
```

6.0.5 categories with families of setoids

A Category with families consists of a base category and a functor [39]. We firstly define the categories with families of sets in Agda as a guidance for the one for setoids. We would present the setoid one here since it is relevant.

We would like to show two formalisation of category with families for setoids here. The first one is simple and short but not comprehensive. We have to extract all complicated components from the simple definition. However the second one gives these components one by one so that it more understandable and convenient.

The category with families works as a model for type theory. So we will introduce them from a type theoretical point of view.

The base category is the category for contexts. In the setoid version we interpret a context as a setoid as well.

To define the second component, namely the presheaf functor, it is necessary to construct the target category first. The objects of this category are families of setoids.The index setoids are the semantic types and the indexed families of setoids are terms. The morphisms are component-wise morphisms between setoids. All the categorical laws hold trivially.

```
inxSetoids : Set₁
inxSetoids = Σ[ I : hSetoid ] (| I | → hSetoid)

_⇉setoid_ : inxSetoids → inxSetoids → Set₁
(I , f) ⇉setoid (J , g) =
  Σ[ i-map : I ⇉ J ]
    ((i : | I |) → f i ⇉ g ( [ i-map ]fn i))
```

```
Fam-setoid : Category
Fam-setoid = CatC
        inxSetoids
        _⇉setoid_
        (λ _ → id' , (λ _ → id'))
        (λ { _ _ (fty , ftm) (gty , gtm) → fty ∘c gty ,
          (λ i → ftm ([ gty ]fn i) ∘c gtm i)})
        (IsCatC
          (λ α β f → PE.refl)
          (λ α β f → PE.refl)
          (λ α δ f g h → PE.refl))
```

Since we already specify the category of contexts, we only need the presheaf which is a contravariant functor from the category of contexts to the category we defined above. The definition of category with families of setoids could be as simple as follows.

```
record CWF-setoid : Set₁ where
  field
  T : Functor (Op setoid-Cat) Fam-setoid
```

All details of this definition are hidden including the functor laws. Therefore we will show the details as the second version.

The semantic contexts are setoids and the terminal object is just the empty context.

```
Con = hSetoid

emptyCon = ⊤-setoid

emptysub = ⋆
```

A semantic type has following components. fm is a setoid of all types. $substT$ is the substitution between types within the context. It should be a morphism between setoids so it has to preserve the equivalence relation. We also need to specify the computation rules for substitution.

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

  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
```

Some other lemmas on the proof irrelevance derived from these fields are not shown here since they are just auxiliary functions.

Then we have to define the substituting in a type given a context morphism and verify it preserves equivalence relation as well.

```
_[_]T : ∀ {Γ Δ : Con} → Ty Δ → Γ ⇉ Δ → Ty Γ
A [ f ]T
  = record
  { fm     = fm ∘ fn
  ; substT = substT ∘ resp
  ; subst* = subst* ∘ resp
  ; refl*  = λ _ _ → subst-pi'
  ; trans* = λ _ _ _ →
    [ fm (fn _) ]trans (trans* _ _ _) subst-pi
  }
  where
    open Ty A
    open _⇉_ f
```

The semantic terms are simpler. It should also preserve the equivalence relation on the elements of contexts.

```
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
```

Substitution for terms can be defined as

```
_[_]m : ∀ {Γ Δ : Con}{A : Ty Δ} →
  Tm A →
  (f : Γ ⇉ Δ)
  → Tm (A [ f ]T)
_[_]m t f = record
  { tm = [ t ]tm ∘ [ f ]fn
  ; respt = [ t ]respt ∘ [ f ]resp
  }
```

Syntactically we can form a new context by using a context Γ and a type $A : Ty\,\Gamma$. To introduce a term of it, we need a term of the semantic context Γ and a term of semantic type A. It is called context comprehension.

```
_&_ : (Γ : Con) → Ty Γ → Con
Γ & A = record
  { Carrier = Σ[ x : | Γ | ] | fm x |
  ; _≈h_    = λ{(x , a) (y , b) →
                Σ'[ p : x ≈h y ] [ fm y ] substT p a ≈h b}
  ; isEquiv =
  record
  { refl    = refl , (refl* _ _)
  ; sym     = λ {(p , q) → (sym p) ,
    [ fm _ ]trans
    (subst* _ ([ fm _ ]sym q))
    trans-refl }
  ; trans = λ {(p , q) (m , n) →
    trans p m ,
    [ fm _ ]trans
    ([ fm _ ]trans
    ([ fm _ ]sym (trans* _ _ _)) (subst* _ q)) n }
  }
  }
```

There are also some other morphisms come with it. Any morphism from a context Γ to a context $\Delta\&A$ consists of a morphism from Γ to Δ and a term of type A substituted. In other words, There is an isomorphism between $Hom(\Gamma, \Delta\&A)$ and $\Sigma\gamma : Hom(\Gamma, \Delta)A[\gamma]$.

fst projects the morphism and snd projects the term. Indeed the fst operation provides weakening for types, and the snd projection enables us to interpret variables. $fst\&$ defines a morphism for each type A which is a canonical projection of A. We need to use id' which are identity context morphisms to achieve these.

```
fst : {Γ Δ : Con}(A : Ty Δ) → Γ ⇉ (Δ & A) → Γ ⇉ Δ
fst A f = record
      { fn = proj₁ ∘ [ f ]fn
      ; resp = proj₁ ∘ [ f ]resp
      }

fst& : {Γ : Con}(A : Ty Γ) → Γ & A ⇉ Γ
fst& A = fst A id'

_+T_ : {Γ : Con} → Ty Γ → (A : Ty Γ) → Ty (Γ & A)
B +T A = B [ fst& A ]T

snd : {Γ Δ : Con}(A : Ty Δ) →
  (f : Γ ⇉ (Δ & A))
  → Tm (A [ fst A f ]T)
snd A f = record
      { tm = proj₂ ∘ [ f ]fn
      ; respt = proj₂ ∘ [ f ]resp
      }

v0 : {Γ : Con}(A : Ty Γ) → Tm (A +T A)
v0 A = snd A id'
```

Inversely we could define a pairing operation to combine a context morphism with a term. The η-law for the projection and pairing holds trivially.

```
_,,_ : {Γ Δ : Con}{A : Ty Δ}(f : Γ ⇉ Δ) →
  (Tm (A [ f ]T))
  → Γ ⇉ (Δ & A)
f ,, t = record
    { fn = ⟨ [ f ]fn , [ t ]tm ⟩
    ; resp = ⟨ [ f ]resp , [ t ]respt ⟩
    }

&-eta : {Γ Δ : Con}{A : Ty Δ}(f : Γ ⇉ (Δ & A))
  → _,,_ {A = A} (fst A f) (snd A f) ≡ f
&-eta f = PE.refl
```

Then a lifting operation could help us define Π-types.

```
lift : {Γ Δ : Con}(f : Γ ⇉ Δ)(A : Ty Δ) → Γ & A [ f ]T ⇉ Δ & A
lift f A = record
  { fn = ⟨ [ f ]fn ∘ proj₁ , proj₂ ⟩
  ; resp = ⟨ [ f ]resp ∘ proj₁ , proj₂ ⟩
  }

lift-eta : {Γ Δ : Con}
    (f : Γ ⇉ Δ)(A : Ty Δ)(x : | Γ |)
    (a : | [ A ]fm ([ f ]fn x) |)
    → [ lift f A ]fn (x , a) ≡ ( [ f ]fn x , a)
lift-eta f A x a = PE.refl
```

One of the most complicated part of this definition is the Π-types. Π-types is also called dependent function types. Semantically it is a function type on the underlying semantic types with a proof that the the functions respect the equivalence relation.

$$\Pi : \{\Gamma : \mathsf{Con}\}(A : \mathsf{Ty}\ \Gamma)(B : \mathsf{Ty}\ (\Gamma\ \&\ A)) \to \mathsf{Ty}\ \Gamma$$

It also comes with two necessary operation on the terms of Pi-types, λ-abstraction and application. There are $\beta-\eta$ laws to verfify for them so that we could form an isomorphism with these two operations. however technically it causes stack overflow. We may simplify these definition in the future so that we could verify them in Agda.

$$\mathsf{lam} : \{\Gamma : \mathsf{Con}\}\{A : \mathsf{Ty}\ \Gamma\}\{B : \mathsf{Ty}\ (\Gamma\ \&\ A)\} \to \mathsf{Tm}\ B \to \mathsf{Tm}\ (\Pi\ A\ B)$$

$$\mathsf{app} : \{\Gamma : \mathsf{Con}\}\{A : \mathsf{Ty}\ \Gamma\}\{B : \mathsf{Ty}\ (\Gamma\ \&\ A)\} \to \mathsf{Tm}\ (\Pi\ A\ B) \to \mathsf{Tm}\ B$$

Non-dependent version of Π-types namely function types can be defined with type weakening. Since the dependence disappears, it is possible to define it straightforwardly without using Π-types.

$$\_\Rightarrow'\_ : \{\Gamma : \mathsf{Con}\}(A\ B : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ \Gamma$$
$$A \Rightarrow' B = \Pi\ A\ (B +\mathsf{T}\ A)$$

6.1 What we can do in this model

6.1.1 Examples of types

We also implement some common types within the syntactic structure of this type theory. For example, natural numbers and the simply typed universe. I will only present the natural numbers here.

The semantic of natural numbers is just natural numbers and the equivalence relation is defined recursively with respect to case analysis on natural numbers. Other operations and properties of this type can be proved easily.

```
⟦Nat⟧ : {Γ : Con} → Ty Γ
⟦Nat⟧ = record
  { fm = λ γ → record
    { Carrier = ℕ
    ; _≈h_    = _≈nat_
    ; isEquiv = record
      { refl = λ {n} → reflNat {n}
      ; sym = λ {x} {y} → symNat {x} {y}
      ; trans = λ {x} {y} {z} → transNat {x} {y} {z}
      }
    }
  ; substT = λ _ → id
  ; subst* = λ _ → id
  ; refl* = λ x a → reflNat {a}
  ; trans* = λ p q a → reflNat {a}
  }
```

Zero and successor operator can be defined as follows.

```
⟦0⟧ : {Γ : Con} → Tm {Γ} ⟦Nat⟧
⟦0⟧ = record
  { tm = λ _ → 0
  ; respt = λ p → tt
  }

⟦s⟧ : {Γ : Con} → Tm {Γ}⟦Nat⟧ → Tm {Γ} ⟦Nat⟧
⟦s⟧ (tm: t resp: respt)
  = record
  { tm = suc ∘ t
  ; respt = respt
```

```
  }
```

The equality type is an essential part of a type theory. We could define it by using the equivalence relation from the setoid representation of type A. The equivalence relation is trivial since it is proof-irrelevant.

```
Rel : {Γ : Con} → Ty Γ → Set₁
Rel {Γ} A = Ty (Γ & A & A +T A)

⟦Id⟧ : {Γ : Con}(A : Ty Γ) → Rel A
⟦Id⟧ A
  = record
    { fm = λ {((x , a) , b) →   record
      { Carrier = [ [ A ]fm x ] a ≈ b
      ; _≈h_ = λ x₁ x₂ → ⊤'
      ; isEquiv = record
        { refl = λ {x₁} → tt
        ; sym = λ x₂ → tt
        ; trans = λ x₂ x₃ → tt
        }
      } }
    ; substT = λ {((x , a) , b) x0 →
      [ [ A ]fm _ ]trans
      ([ [ A ]fm _ ]sym a)
      ([ [ A ]fm _ ]trans
      ([ A ]subst* _ x0) b)
      }
    ; subst* = λ p x₁ → tt
    ; refl* = λ x a → tt
    ; trans* = λ p q a → tt }
```

The unique inhabitant $refl$ is defined as

```
cm-refl : {Γ : Con}(A : Ty Γ) → Γ & A ⇉ (Γ & A & A +T A)
cm-refl A = record { fn = λ x' → x' , proj₂ x'
      ; resp = λ x' → x' , proj₂ x' }

⟦refl⟧⁰ : {Γ : Con}(A : Ty Γ)
  → Tm {Γ & A} (⟦Id⟧ A
  [ cm-refl A ]T)
⟦refl⟧⁰ A = record
      { tm = λ {(x , a) → [ [ A ]fm x ]refl {a} }
      ; respt = λ p → tt
      }

⟦refl⟧ : {Γ : Con}(A : Ty Γ)
  → Tm {Γ} (Π A (⟦Id⟧ A
  [ cm-refl A ]T) )
⟦refl⟧ {Γ} A =   lam {Γ} {A} (⟦refl⟧⁰ A)
```

We have an abstracted $refl$ term as well. Using Π-types we could define the eliminator for Id, but it is more involved.

We have done the basics for category of families of setoids. There are more types can be interpreted in this model so that we could show that it is a valid model for Type Theory. We would like to interpret quotient types in this model by following Hofmann's method in [27] or by ourselves.

6.2 Quotient types in setoid model

6.3 Observational equality

Later in in [40], Altenkirch and McBride further simplifies the setoid model by adopting McBride's heterogeneous approach to equality. They identifies 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 compares terms of types which are different in construction. It only make sense when we can prove the types are the same. It helps us avoids the heavy use of *subst* which makes formalisation and reasoning involved. We could simplify the setoid model by adapting this approach and the implementation could be easier.

Chapter 7

Homotopy Type Theory and higher inductive types

Homotopy Type Theory is a new branch between theoretical computer science and mathematics and it is a variant of Martin-Löf type theory. We have Vladimir Voevodsky's univalence axiom which identifies isomorphic structures. I will not explain this topic in detail here, but a well-written text book on Homotopy Type Theory which is written by many brilliant mathematicians and computer scientists is available now [**?**].

With higher inductive types, it is possible to define the quotient types in Homotopy Type Theory. However the implementation of Homotopy Type Theory in intensional type theory is still n open problem. We work on defining semi-simplicial types and weak ω-groupoids to solve it. There is also the very new model using cubical sets proposed by Bezem, Coquand and Huber in [41].

In the next chapter we will present an syntactic implementation of weak ω-groupoids following Brunerie's approach.

Chapter 8

the ω-groupoids Model

Before 20th-Mar-2014

It is possible to define the quotient types in Homotopy Type Theory but to implement the Homotopy Type Theory in intensional type theory, it is still a difficult problem. We work on defining semi-simplicial types and weak ω-groupoids to solve it.

In this chapter we present an syntactic implementation of weak ω-groupoids following Brunerie's approach. We did some contributions like adapting using heterogeneous equality and syntactic construction of reflexivity.

8.1 An implementation of weak ω-groupoids

It is very interesting to investigate the approach to define quotient types in Homotopy Type Theory which is a variant of Martin-Löf type theory. In Homotopy Type Theory, we reject the principle of uniqueness of identity proofs (UIP) but instead we accept the univalence axiom which says that equality of types is weakly equivalent to weak equivalence. Weak equivalence can been seen as a refinement of isomorphism without UIP [42]. To make it more precise, a weak equivalence between two objects A and B in a 2-category is a morphism $f : A \longrightarrow B$ which has a corresponding inverse morphism $g : B \longrightarrow A$, but instead of the proofs of isomorphism $f \circ g = 1_B$ and $g \circ f = 1_A$ we have two 2-cell isomorphisms $f \circ g \cong 1_B$ and $g \circ f \cong 1_A$. Since the setoid interpretation of types in setoid model, as we mentioned before, relies on UIP, it has to be generalised so that we could formalise it in intensional type theory.

The generalised notion is called Grothendieck ω-groupoids. Grothendieck introduced the notion of ω-groupoids in 1983 in a famous Manuscript *Pursuing Stacks* [43]. Maltsiniotis

continued his work and suggested a simplification of the original definition which can be found in [44]. Later Ara also present a slight variation of the simplification of weak ω-groupoids in [45]. Categorically speaking an ω-groupoid is an ω-category in which morphisms on all levels are equivalences. As we know that a set can be seen as a discrete category, a setoid is a category where every morphism is unique between two objects. A groupoid is more generalised, every morphism is isomorphism but the proof of isomorphism is unique, namely the composition of a morphism with its inverse is equal to an identity morphism. Similarly, an n-groupoid is an n-category in which morphisms on all levels are equivalence. ω-groupoids which are also called ∞-groupoids is an infinite version of n-groupoids. To model Type Theory without UIP we also require the equalities to be non-strict, in other words, they are not definitionally equalities. Finally we should use weak ω-groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak ω-groupoids in Type Theory. For instance, Altenkirch and Rypacek's paper [42], and Brunerie's notes [46]. We work on an implementation of weak ω-groupoids following Brunerie's approach in Agda. The approach is to specify when a globular set is a weak ω-groupoid by first defining a type theory called $\mathcal{T}_{\infty-groupoid}$ to describe the internal language of Grothendieck weak ω-groupoids, then interpret it with a globular set and a dependent function. All coherence laws of the weak ω-groupoids should be derivable from the syntax, we will present some basic ones, for example reflexivity. One of the main contribution of our work is to use the heterogeneous equality for terms to overcome some very difficult problems when we used the normal homogeneous one. When introducing our implementation, we omit some complicated but less important programs, namely the proofs of some lemmas or the definitions of some auxiliary functions. it is still possible for the reader who is interested in the details to check the code online, in which there are only some minor differences.

8.1.1 Syntax

Since the definitions of contexts, types and terms involve each others, we adopt a more liberal way to do mutual definition in Agda which is a feature available since version 2.2.10. Something declared is free to use even it has not been completely defined.

8.2 Quotient types in Homotopy Type Theory

8.3 Syntax of weak ω-groupoids

8.4 Semantics

8.5 Higher inductive types

Chapter 9

Summary

9.0.1 Future work

Appendix A

Appendix A

```
GCD′→ℚ : ∀ x y di → y ≢ 0 → C.GCD′ x y di → ℚ
GCD′→ℚ .(q₁ ℕ* di) .(q₂ ℕ* di) di neo (C.gcd-* q₁ q₂ c) = record { numerator = numr
  ; denominator-1 = pred q₂
  ; isCoprime = iscoprime }
  where
  numr = ℤ.+_ q₁
  deno = suc (pred q₂)

  numr≡q1 : | numr | ≡ q₁
  numr≡q1 = refl

  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 → suc (pred n) ≡ n
  invsuc zero nz with nz refl
  ... | ()
  invsuc (suc n) nz = refl

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

```
transCop : ∀ {a b c d} → c ≡ a → d ≡ b → C.Coprime a b → C.Coprime c d
transCop refl refl c = c

copnd : C.Coprime | numr | deno
copnd = transCop numr≡q1 deno≡q2 c

witProp : ∀ a b → GCD a b 1 → True (C.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 (C.coprime? | numr | deno)
iscoprime = witProp | numr | deno (C.coprime-gcd copnd)
```

Bibliography

[1] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Bertrand Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1903.

[3] Kurt Godel. *On formally undecidable propositions of principia mathematica and related systems.* Dover, 1992.

[4] Henk Barendregt et al. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[5] Per Martin-Löf. A Theory of Types. Technical report, University of Stockholm, 1971.

[6] 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.

[7] Antonius JC Hurkens. A simplification of girard's paradox. In *Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.

[8] Per Martin-Löf. *Intuitionistic type theory*, volume 17.

[9] 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.

[10] 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.

[11] Thorsten Altenkirch. Should extensional type theory be considered harmful? Talk given at the Workshop on Trends in Constructive Mathematics, 2006.

[12] The Agda Wiki. Main page, 2010. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage. [Online; accessed 13-April-2010].

[13] Ulf Norell. Dependently typed programming in agda. Available at: http://www.cse.chalmers.se/ ulfn/papers/afp08tutorial.pdf, 2008.

[14] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. Pisigma: Dependent types without the sugar. submitted for publication, November 2009.

[15] Wikipedia. Lazy evaluation — Wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/wiki/Lazy_evaluation. [Online; accessed 20-April-2010].

[16] 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.

[17] Wikipedia. Coinduction — Wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/wiki/Coinduction. [Online; accessed 20-April-2010].

[18] Martin Hofmann. *Extensional concepts in Intensional Type Theory*. PhD thesis, School of Informatics., 1995.

[19] Vladimir Voevodsky. A very short note on homotopy λ-calculus. Available at: http://math.ucr.edu/home/baez/Voevodsky_note.ps, 2006.

[20] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[21] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable Quotients in Type Theory. 2011.

[22] Vladimir Voevodsky. Generalities on hSet - Coq library hSet. http://www.math.ias.edu/~vladimir/Foundations_library/hSet.html.

[23] Nicolai Kraus. Equality in the Dependently Typed Lambda Calculus: An Introduction to Homotopy Type Theory. http://www.cs.nott.ac.uk/~ngk/karlsruhe.pdf, October 2011.

[24] 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.

[25] N.P. Mendler. Quotient types via coequalizers in martin-löf type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.

[26] Aleksey Nogin. Quotient types: A Modular Approach. In *ITU-T Recommendation H.324*, pages 263–280. Springer-Verlag, 2002.

[27] Martin Hofmann. A Simple Model for Quotient Types. In *Proceedings of TLCA'95, volume 902 of Lecture Notes in Computer Science*, pages 216–234. Springer, 1995.

[28] 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.

[29] Peter V. Homeier. Quotient Types. In *In TPHOLs 2001: Supplemental Proceedings*, page 0046, 2001.

[30] Pierre Courtieu. **Normalized types**. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[31] Gilles Barthe and Herman Geuvers. Congruence Types. In *Proceedings of CSL'95*, pages 36–51. Springer-Verlag, 1996.

[32] Thierry Coquand. Pattern Matching with Dependent Types. In *Types for Proofs and Programs*, 1992.

[33] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[34] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with Quotient Types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.

[35] 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.

[36] Paul van der Walt. *Reflection in Agda*. PhD thesis, Master's thesis, Universiteit Utrecht, 2012.

[37] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, New York, 1985. ISBN 0-387-15066-8.

[38] Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, pages 427–441. Springer, 1995.

[39] Pierre Clairambault. From categories with families to locally cartesian closed categories. *Project Report, ENS Lyon*, 2005.

[40] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi: http://doi.acm.org/10.1145/1292597.1292608.

[41] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *Preprint., September*, 2013.

[42] Thorsten Altenkirch and Ondrej Rypacek. A syntactical Approach to Weak ω-groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, 2012.

[43] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.

[44] G. Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. *ArXiv e-prints*, September 2010.

[45] D. Ara. On the homotopy theory of Grothendieck ∞-groupoids. *ArXiv e-prints*, June 2012.

[46] Guillaume Brunerie. Syntactic Grothendieck weak ∞-groupoids. http://uf-ias-2012.wikispaces.com/file/view/SyntacticInfinityGroupoidsRawDefinition.pdf, 2013.