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

November 2013

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

TO DO: what we are going to write

1.1 Organisation

TO DO: short introductions to each section

Chapter 2

Background

TO DO: Before 15th-Nov-2013

2.1 Type Theory

To introduce type theory, it is helpful to start from set theory first. Set theory is a language to describe most definitions of mathematical objects, in another word, it serves as a foundation system for mathematics. In 1870s, George Cantor and Richard Dedekind firstly studied set theory. However, in the 1900s, Bertrand Russell discovered the famous paradox in naive set theory. It tells people that if we can does not distinguish between small sets like natural numbers and real numbers with "larger" sets like the set of all sets, then it will lead to contradiction. To avoid this paradox, he proposed the theories of types [**?**] as an alternative to naïve set theory. The objects are assigned to types in a hierarchical structure such that "larger" sets and small sets reside in different levels. The set of all sets are no longer on the same level as its elements such that the paradox disappears.

The elementary notion in type theory is *type* which plays a similar role to set in set theory, but they have some fundamental differences. 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 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 $\mathbb{N}$.

TO DO: one page explanation of why type theory is useful

Since Russell's type theory, there are a variety of type theories have been developed by famous mathematicians and computer scientists, for example Gödel and Tarski's which is used in Gödel' 1931 paper [**?**]. 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 1930s. He first introduced an untyped lambda calculus which was proved to be inconsistent due to the Kleene-Rosser paradox. Then the typed one which 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 [**?**].

2.1.2 Per Martin-Löf's Type Theories

In 1970s, Per Martin-Löf [**?** **?**] developed his profound intuitionistic type theory. His 1971's formulation which is impredicative was proved to be inconsistent with the Girard's paradox. 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.

Since it is based on the principle of mathematical constructivism, it is also a foundation of mathematics [**?**]. 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"

TO DO: 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 [**?**].

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[**?**].

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.

$$\frac{a \equiv b}{a = b}\ trivial$$

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 [**?**]) is different to definitional equality. The definitional equality is always decidable hence type checking that depends on definitional equality is decidable as well [**?**]. 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 is undistinguished with definitional equality, in other words, two propositional equal objects are judgementally equal. This is called reflecition rule.

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

Objects of different normal forms, for example point-wise equal functions or different proofs of the same proposition, may be definitionally equal. It means that the definitional equality is not decidable and type checking becomes undecidable as well. Th

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* [**?**] 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 [**?**]. 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 [**?**].

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 [**?**]. 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 [**?**], 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 [**?**]. 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 evaulation.* 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. [**?**]

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 [**?**].

  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 [**?**]. 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 [**?**]. 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. [**?**]

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 DO: 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. Most of the basic types are from Agda standard library 0.7.

First of all, like other languages, we use = for function definition rather than propositional equality type former. We use single column : for typing judgement, for example a:A means that a is of type A.

2.2.2 Some conventions

TO DO: What conventions we will follow in this thesis and some commonly used symbols and functions

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$(i+1)$.

2.2.3 Identity Type

Identity type is the type to encode the propositional equality in intensional type theory, like Agda. We use the Paulin-Mohring's identity type which is parameterized with the left side of the identity. This also includes the identity type for arbitrary universe level.

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

There is no eliminators automatically derived for the types defined, but we have pattern matching which is stronger than eliminators, sometimes maybe too strong. As long as we pattern match on a variable of this type with the unique inhabitant refl, the variables on both sides will be turned into the same variable. We could use the feature to define the eliminator J for the identity type.

$$
\begin{aligned}
&\mathsf{J} : (A : \mathsf{Set})(a : A) \to (P : (b : A) \to a \equiv b \to \mathsf{Set}) \\
&\quad \to P\ a\ \mathsf{refl} \\
&\quad \to (b : A)(p : a \equiv b) \to P\ b\ p \\
&\mathsf{J}\ A\ .b\ P\ m\ b\ \mathsf{refl} = m
\end{aligned}
$$

2.2.4 Extensionality

In intensional type theory, usually no extensional equality is included. For example, the functional extensionality which is usually accepted in regular mathematics

$$\frac{fg : A \to B, \forall a : A, fa = ga}{f = g}\ fun - ext$$

is not availabe in intensional type theory. 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. Then the type checker will not always terminate.

As Martin Hofmann summarises in [**?**], there are several related extensional concepts, *Functional extensionality*, *Uniqueness of identity*, *Proof-irrelevance*, *Subset types*, *Propositional extensionality*, *Quotient types*. These concepts are impossible to interpret without extending the currect setting of intensional type theory.

Quotient types is one of the most interesting extensional concepts in type theory. It will be explained in detail in the next Chapter.

Chapter 3

Quotient Types

TO DO: Before 1st-Dec-2013

Quotient is a very common notion is 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 Quotients in mathematics

3.1.1 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 equivalent 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.

Quotient types In type theory, quotient type can be formalised as following:

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

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

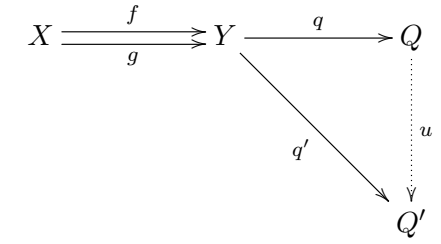
$$\frac{\begin{array}{l} B : A/\sim \to \mathbf{Set} \\ f : (a : A) \to B\,[a] \\ (a, b : A) \to (p : a \sim b) \to f a \overset{p}{=} f b \end{array}}{\hat{f} : (q : Q) \to B\,q}\ Q - \mathbf{elim}$$

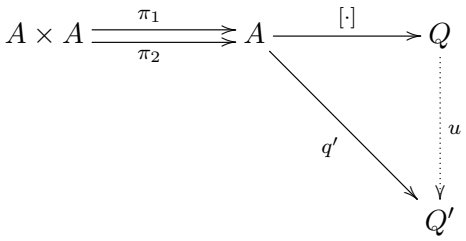
In general, quotient types are unavailable in intensional type theory. Quotients are everywhere, for example, rational numbers, real numbers, multi-sets. The introduction of quotient types is very helpful. Some of them can be defined more effectively, such as the set of integers. Some of them can only be defined with quotient types, such as real numbers (the reason will be covered in TO DO:).

3.2 Categorical intuition

Categorically speaking, a quotient is a coequalizer.

Definition 3.4. 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 $q' = u \circ q$





3.2.1 Adjuction 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 3.5. $\nabla A = (A, \equiv)$, $Q(B, \sim) = B/\sim$

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 Example of Quotients

Because different sets may contain the same elements, we have the subset relation such that we can construct equivalence classes then quotient set. In type theory we have to

give constructors for any type before we can construct elements, which is different to the situation in set theory that elements exist before we construct quotient sets. Therefore this approach to construct quotients in set theory has some problems in type theory. In fact, Voevodsky constructs quotients using this approach in Homotopy Type Theory using Coq [**?**] but here we mainly discuss how to reinterpret quotient sets in the current settings ofintensional type theory (e.g. Agda).

quotient groups, quotient space, partiality monad.

Chapter 4

Definable Quotients

TO DO: Before 18th-Dec-2013

Some types can be defined without quotient, however it does not possess good properties from being quotient types. Examples like integers, which can be defined as either negative or non-negative,

Sometimes, quotient types are more difficult to reason about than their base types. 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. Therefore it is worthwhile for us to conduct a research project on the implementation of quotients in intensional type theory.

The work of this project will be divided into several phases. This report introduces the basic notions in my project on implementing quotients in type theory, such as type theory setoids, and quotient types, reviews some work related to this topic and concludes with some results of the first phase. The results done by Altenkirch, Anberrée and I in [1] will be explained with a few instances of quotients.

4.0.1 Integers

Operations

Properties

Comparison TO DO: The advantage of use Quotient algebraic structure: the proving of distributivity

4.0.2 Rational numbers

TO DO: Complete the proving part

4.0.3 Real numbers and more

TO DO: axiomitised

TO DO: Why real numbers are not definable in intensional type theory?

TO DO: Complete the proving part, talk about the definition in HoTT?

4.0.4 Multisets(bags)

Definition 4.1. A multiset (or bag) is a set without the constraint that there is no repetitive elements.

TO DO: axiomitised? TO DO: Complete the definition and some examples or using quotients?

Chapter 5

Setoid Model

TO DO: Before 30th-Dec-2013

5.1 Literature review

5.2 What we could do in this model

5.3 Quotient types in setoid model

TO DO: With Prop universe, how can we define quotient types?

Chapter 6

Homotopy Type Theory and ω-groupoids Model

TO DO: Before 15th-Jan-2013

6.1 What is Homotopy Type Theory?

6.2 Quotient types in Homotopy Type Theory

6.3 Syntax of weak ω-groupoids

6.4 Semantics

Chapter 7

Summary and future works

Bibliography

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