The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA DOCTORAL THESIS

Thesis Title

Author:
Nuo LI

Supervisor:
Dr. Thorsten ALTENKIRCH

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

in the

Functional Programmin Lab
Department or Computer Science

October 2013

THE UNIVERSITY OF NOTTINGHAM

Abstract

Faculty Name
Department or Computer Science

Doctor of Philosophy

Thesis Title

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

2.1.1 Russell's Type Theories

In the 1900s, Russell find a problem in naive set theory which is known as Russell's paradox. To avoid it, he introduced the theories of types [**?**] as an alternative to naive set theory. Since then, mathematicians and computer scientists developed a number of variants of type theories.

The type theory in this discourse is the one developed by Per Martin-Löf [**? ?**] which is also called intuitionistic type theory. As a foundation of mathamatics, it does not based on predicate logic like set theory. It is developed based on the Curry-Howard isomorphism

"propositions can be interpreted as types and their proofs are inhabitants of that types" (2.1)

It means that a proposition or a spefication of a program can be formalised as a type, and to prove is just to construct an element of that type. Type theory could be a programming language which we can verify the correctness of program within itself.

For a detailed introduction, refer to [**?**].

Set Theory	Type Theory
$a \in A$	$a : A$
$A \cap B$	$A \times B$
$A \cup B$	$A + B$

TABLE 2.1: Correspondance

Simlilarities

$$a \in A \tag{2.2}$$

$$a : A \tag{2.3}$$

Differences Set theory requires logic as basis. But type theory doesn't.

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

2.1.2 Per Martin-Löf's Type Theories

Per Martin-Löf proposed both an intensional and an extensional variants of his intuitionistic type theory. The difference arises from one of the primitive notions, *equality*. There are two ways to interpret equalty in type theory, one is called definitional equality and the other is propositional equality.

In intensional type theory, identical types and terms are definitional equal and things on the two side of an equal symbol = are definitional equal. It means that the definitional equality is part of the intensional type theory's basics rather than some type we defined. Definitional equality can be judged and decided by type-checker.

But propositional equality is types like $\mathsf{a} \equiv \mathsf{b}$ which stands for a propostion that a equals b. However propostional equality is some type which we need to prove it or disprove by construction.

Anything is only definitionally equal to itself and all terms that can be normalised to it, which means that definitional equality is decidable in intensional type theory. Therefore type checking that depends on definitional equality is decidable as well [?]. The type for propositional equality inintensional type theory is usually written as $\mathsf{Id}(\mathsf{A}, \mathsf{a}, \mathsf{b})$ (which is also called *intensional equality* [?]). In Agda[?], an implementation of intensional type theory, it is written as $\mathsf{a} \equiv \mathsf{b}$ with the type A often kept implicit. This set has an unique element refl only if a and b are definitionally equal and is uninhabited if not.

However in extensional type theory, the two kinds of equalities are not distinguished, so if we have $\mathsf{p} : \mathsf{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{b})$ (which is called extensional equality [?]), a and b are definitionally equal. It means that terms which have different normal forms may be definitionally equal. In other words, definitional equality is undecidable and type checking becomes undecidable as well. However Altenkirch and McBride have introduced a variant of extensional type theory called *Observational Type Theory* [?] in which definitional equality is decidable and propositional equality is extensional.

Type theory can also serve as a programming language in which the evaluation of a well-typed program always terminates [?]. There are various implementations based on different type theories, such as NuPRL, LEGO, Coq, Epigram and Agda. Agda is one of the most recent implementations of intensional version of Martin-Löf type theory. It is a dependently typed programming language, we can write program specifications as types. As we have seen, Martin-Löf type theory is based on the Curry-Howard isomorphism: types are identified with propositions and programs (or terms) are identified with proofs. Therefore it is not only a programming language but also a theorem prover which allows user to verify Agda programs in itself. Compared to other implementations, it has a package of useful features such as pattern matching, unicode input, and implicit arguments [?], but it does not have tactics and consequently its proofs are less readable than implementations that do. Since this project is based on Martin-Löf type theory, it is a good choice to implement our definitions and verify our theorems and properties in Agda. For a detailed introduction of Agda, refer to [?].

To move from set theory to type theory, the similarities and differences should be made clear. Although type theory has some similarities to set theory, their foundations are different. Types play a similar role to sets and they are also called sets in many situations. However we can only create elements after we declare their types, while in set theory elements exist there before we have sets. For example, we have the type $\mathbb{N}$ for natural numbers corresponding to the set of natural numbers in set theory. In set theory, 2 is a shared element of the set of natural numbers and the set of integers. While in type theory, $\mathbb{N}$ provides us two constructors $\mathsf{zero} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \longrightarrow \mathbb{N}$, and 2 can be constructed as $\mathsf{suc}\,(\mathsf{suc}\ \mathsf{zero})$ which is of type $\mathbb{N}$ and does not have any other types like $\mathbb{Z}$. Because different sets may contain the same elements, we have the subset relation such that we can construct equivalence classes and 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).

2.2 A dependently typed language: Agda

TO DO: copied from undergraduate final year report, most of them have to be rewritten
Agda is a dependently typed functional programming language which is designed based on Martin-Löf's Type Theory [**?**]. We can find three key elements in the definition of Agda, the "functional programming language", "dependently typed" and "Per Martin-Löf Type Theory".

- *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.

- *Dependent type.* Dependent types are types that depends on values of other types [**?**]. It is one of the most important features that makes Agda a proof assistant. 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 **Vector A n**.

  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. With the type checker of Agda, we can set more constraints in the type so that type-unmatched problems will always be detected by complier. Therefore we could define the function more precisely as there more partitions of types. For instance, to use the dependently typed vector, it could avoid defining a function which will cause exceptions like out of bounds in Java.

- *Per Martin-Löf Type Theory.* It has different names like Intuitionistic type theory or Constructive type theory and is developed by Per Martin-Löf in 1980s. It associated functional programs with proofs of mathematical propositions written as dependent types. That means we can now represent propositions we want to prove as types in Agda by dependent types and Curry-Howard isomorphism [**?**]. Then we only need to construct a program of the corresponding type to prove that propostion. For example:

> As Nordström et al. [?] pointed out that we could express both specifications and programs at the same time when using the type theory to construct proofs using programs. 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. There are no tactics like in Coq. However it is more flexible to construct a proof. The process of building proofs is very similar to the process of constructing proofs in regular mathematics. The logic behind it is that if we could construct an instance of the type (proposition), we prove it. It is actually the Curry-Howard isomorphism.

Agda is an extention of this type theory [?] with some nice features which could benefit the 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. We can also use view to pattern match a condition specially in Agda. For example,

  Here the "parity" after with is a view function that allows us to pattern match on the result of it.

- *Recursive definition.*The availability of recursive definition enables programmers to prove propositions in the same manner of mathematical induction. Generally the natural numbers are defined inductively in fucntional programming languages. Then the program of natural numbers can be written using recursive style. There are a lot of types defined using recursive styles in Agda.

- *Construction of functions.* The construction of functions makes the proving more flexible. We could prove lemmas as we do in maths and reuse them as functions.

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

As Agda is primarily used to undertake theorem proofs tasks, the designer enhanced it to be more professional proof assistant. There are several beneficial features facilitating theorem proving,

- *Type Checker.* Type checker is an essential part of Agda. It is the type checker that detect unmatched-type problem which means the proof is incorrect. It also shows the goals and the environment information related to a proof. Moreover a definition of function must cover all possible cases and must terminate as Agda are not allowed to crash [**?**]. The coverage checker makes sure that the patterns cover all possible cases [**?**]. And 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. The type checker then ensures that the proof is complete and not been proved by itself. Also we are forced to write the type signature due to the presence of type checker.

- *Emacs interface.* It has a Emacs-based interface for interactively writing and verifying proofs. It allows programmers leaving part of the proof unfinished so that the type-checker will provide useful information of what is missing [**?**]. Therefore programmers could gradually refine their incomplete proofs of correct types.

- *Unicode support.* It supports Unicode identifiers and keywords like: $\forall$, $\exists$ etc. It also supports mixfix operators like: $+$, $-$ etc. The benefits are obvious. Firstly we could define symbols which look the same and behave the same in mathematics. These are the expressions of commutativity for natural numbers, the first line is mathematical proposition and the second line is code in Agda:

  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.* We can simply navigate to the definition of functions from our current code. It is a wonderful features for programmers as it alleviate a great deal of work to look up the library.

- *Implicit arguments.* We could omit the arguments which could be inferred by the type-checker. In this way, we do not need to present obvious targets of some properties. For example,

  The implicit argument in curly bracket is unnecessay to give explicitly when applying this property.

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

- *Coinduction.* We could define coinductive types like streams in Agda which are typically infinite data structures. It is a proof technique that could prove the equality satisfied all possible implementation of the specification defined in the codata. It is often used in conjunction with lazy evaluation. [**?**]

With these helpful features, Agda has the potentional be a more powerful proof assistant. Therefore, in order to provide the availability of all the numbers, this project should be beneficial. With the numbers defined and basic properties proven, mathematicians could prove some famous theorems like Fermat's little theorem then.

2.2.1 basic syntax

TO DO: to help the reader understand the Agda code in text

2.2.2 Some conventions in this paper

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

Chapter 3

Quotient Types

TO DO: Before 1st-Dec-2013

Quotient is a very common notion is mathematics. It refers to the result of division at first and is extended to other abstract branches of mathematics. More generally, 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 Quotients in mathematics

Quotient sets In set theory, the old brother of type theory, 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.

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

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

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

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

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

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

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 3.1. 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 $q' = u \circ q$

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

$$\begin{tikzcd}
A \times A \arrow[r, shift left, "\pi_1"] \arrow[r, shift right, "\pi_2"'] & A \arrow[r, "{[\cdot]}"] \arrow[dr, "q'"'] & Q \arrow[d, dotted, "u"] \\
 & & Q'
\end{tikzcd}$$

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 3.2. $\nabla A = (A, \equiv)$, $Q(B, \sim) = B/\sim$

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

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

3.3 Example of Quotients

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.