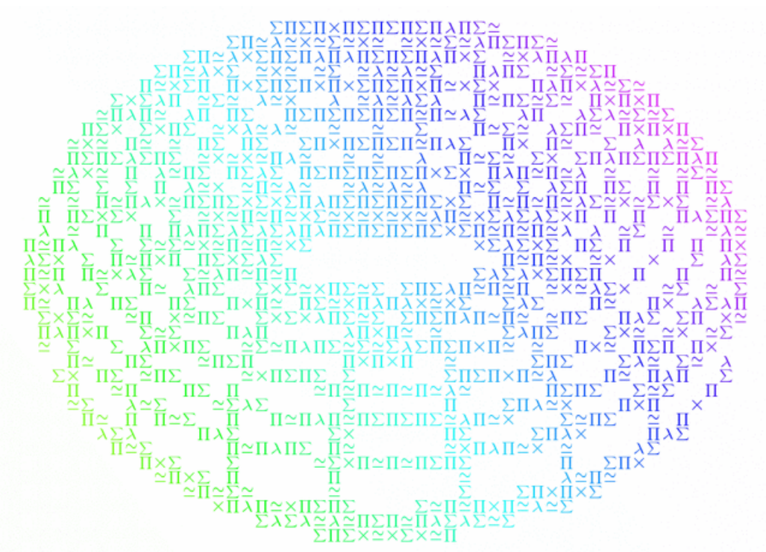
Quotient Containers in Homotopy Type Theory

Jonathan E. Sherry

School of Computer Science and Information Technology
University of Nottingham



Submitted May 2014 in partial fulfilment of the conditions of the award of the degree of BSc (Hons) Computer Science.

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature ______________________________

Date ______/______/______

Contents

Part I

Introduction

Chapter 1

Abstract

This paper serves to introduce and explain the notions of Martin Löf Type Theory, Homotopy Type Theory, Containers and Differentiation of Data structures. Using ideas from each of these fields, Quotient Containers are introduced - specifically a multiset container and implmented in *Agda*. The veracity of the implementation is then tested through differentiating a multiset container and concluding that the implementation behaves as one would expect and is true to theory.

Chapter 2

Acknowledgements

This study would not have been made possible without the following: Thorsten
Mum, Dad Tony
etc.

Chapter 3

Motivation

Part II

Research

Chapter 4

Martin-Löf Type Theory

4.1 Introduction

Martin-Löf type theory is a form of constructive, intuitionistic type theory that is both a programming language and an alternative foundation of mathematics. Devised by Swedish mathematician and philosopher Per Martin-Löf in 1972, Martin-Löf type theory (MLTT) serves to provide a set of formal rules and type-theoretic connectives to perform mathematical reasoning. MLTT is said to be *constructive* and *intuitionistic* as it replaces the classical concept of Truth, with the notion of constructive provability. That is, construction of a mathematical object is proof of its existence. In MLTT, objects are classified by the basal notion of a *type* primitive — objects can have a certain type and are said to inhabit that type. These structured types can be used to provide a specification for the elements within it, providing a means to reason about objects of that type. For example, from a type:

$$A \to B$$

, the type of functions from type A to type B, instructions on how to construct an object are implicitly known. In this case, an object of the type $A \to B$ is formed from an object of type B, parametrised by an object of type A.

The rigorous rules and predictable nature of objects and types within MLTT afford it strength in formal reasoning. This strength has led to its implementation as a base for a number of languages and proof assistants including Agda *(see chapter 8)*, Epigram and Coq, among others.

This chapter constitutes a brief introduction to key concepts and ideas within Martin-Löf type theory required for the rest of the text.

4.2 Basic Constructions

A Type	Declaration of a type A.
$a : A$	Object a exists and inhabits type A.
$A \equiv B$	Type A is judgementally equal to type B.
$a \equiv b : A$	a and b are judgementally equal terms of type A.
$\Gamma \vdash A$	A is well-formed in the context Γ.
$\Gamma \vdash a : A$	a is well-formed term of the type A in the context Γ.

4.3 Type Connectives

4.3.1 Finite Types

Finite types are those that have a strictly finite, enumerable number of inhabitants. Examples include:

- the $\mathbf{0}$-type. Otherwise known as the *empty* or *bottom* type, it has no inhabitants and invoking the Curry-Howard isomorphism[1], it represents False. An inhabitant of the $\mathbf{0}$-type may be referred to as a *contradiction* as there is no way to prove a contradiction and nor can the $\mathbf{0}$-type be constructed.
- the $\mathbf{1}$-type. Also called the *unit* type. It has only one inhabitant and invoking the Curry-Howard isomorphism, it represents True.
- the $\mathbf{2}$-type. Intended to have exactly two inhabitants: $0_{\mathbf{2}}$ and $1_{\mathbf{2}}$, it could be constructed as $\mathbf{1} + \mathbf{1}$. This is the type of Booleans.

4.3.2 Product Types

Given types $A, B : \mathcal{U}$, $A \times B : \mathcal{U}$ is the type of their cartesian product where elements of $A \times B$ are pairs $(a, b) : A \times B$ such that $a : A$ and $b : B$.

4.3.3 Coproduct Types

Given types $A, B : \mathcal{U}$, $A + B : \mathcal{U}$ is the type of their coproduct. Inhabitants of $A + B$ can be constructed with `inl(a)` $: A + B$ for $a : A$ or `inr(b)` $: A + B$ for $b : B$.

[1] Discovered by Haskell Curry and William Alvin Howard, the Curry-Howard isomorphism states that there is an equivalence between logic and programming. More formally, there is a syntactic analogy between formal logic and computational calculi.

4.3.4 Π-Types

Π-types, also known as dependent product type, is a type whose inhabitants are functions whose codomain is dependent on the domain to which the function is applied. It can be regarded as the cartesian product over a type.
Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, we can construct the type of dependent functions:

$$\prod_{(x:A)} B(x) : \mathcal{U}$$

For a constant family B, this is judgementally equal to a function $A \to B$.
Polymorphic functions are an example of a dependent type element. A polymorphic function would be of the type:

$$\prod_{(A:\mathcal{U})} A \to B$$

Invoking the Curry-Howard isomorphism, Π-types represent implication and universal quantification.

4.3.5 Σ-Types

Σ-types, or dependent pair types, are types where the latter component of a pair depends on the former. It is analogous to an indexed sum over a given type.
Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, we can construct the type of dependent pair functions:

$$\sum_{(x:A)} B(x) : \mathcal{U}$$

For a constant family B, this is judgementally equal to the cartesian product type: $(A \times B)$. Invoking the Curry-Howard isomorphism, Σ-types represent conjunction and existential quantification.

4.3.6 Identity Types

Identity or equality types are of the form $a =_A b$ such that $a : A$ and $b : A$.
Given a type $A : \mathcal{U}$ and elements $a, b : A$ we can form type $a =_A b : \mathcal{U}$ in the same universe. There is only one inhabitant of this type — the proof of reflexivity:

$$\texttt{refl} : \prod_{(a:A)} (a =_A a)$$

`refl` states that all elements in A are equal to themselves. This means that if a and b are judgementally equal, we also have a witness of `refl`, $\texttt{refl}_{\texttt{a}} : (a =_A b)$.

4.3.7 Inductive Types

Inductive types are those that includes a constructor that encodes how to create new elements of the type. Normally, there will be a base case and an inductive case. A good example of this is an inductive definition of the natural numbers. Elements of type $\mathbb{N} : \mathcal{U}$ are constructed with $0 : \mathbb{N}$ and $successor : \mathbb{N} \to \mathbb{N}$.

4.3.8 Universes

A universe, $\mathcal{U}$, is a type whose elements are types. To avoid Russell's paradox[2], universes are ordered in a heirarchy. That is, $\mathcal{U}_0 : \mathcal{U}_1$, $\mathcal{U}_1 : \mathcal{U}_2$... Universes are *cumulative*, meaning all elements of the i^{th} universe are also elements of the $(i+1)^{th}$ universe. When we declare a type, we say implicitly that it inhabits some universe $\mathcal{U}_i$.

4.3.9 Propositions as Types

Using the types above, it is possible to construct types that represent logical propositions, that is, logical sentences with a truth value, as such:

Logic	**Type Theory**
True	$\mathbf{1}$
False	$\mathbf{0}$
A and B	$A \times B$
A or B	$A + B$
If A then B	$A \to B$
A iff B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$
For all $x : A, P(x)$ is true	$\prod_{x:A} P(x)$
There exists $x : A$ such that $P(x)$ is true	$\sum_{x:A} P(x)$

4.4 Equality proofs form a groupoid?

[2] In 1901, Bertrand Russell used this paradox to derive a contradiction in Cantor's naive set theory. The paradox is as follows: "Let R be the set of all sets which are not members of themselves. Then R is neither a member of itself nor not a member of itself."[9] Type theory was introduced by Russell as a solution to this problem.

Chapter 5

Homotopy Type Theory

5.1 Introduction

Homotopy type theory is a new branch of mathematics that came about after a special year on *Univalent Foundations of Mathematics* during 2012 and 2013 at The Institute for Advanced Study, Princeton. Organised by Vladamir Voevodsky, Thierry Coqand and Steve Awodey, the *Univalent Foundations* program sought to explore implications of Voevodsky's *Univalence Axiom*, $(A = B) \simeq (A \simeq B)$, that is "identity is equivalent to equivalence. In particular, one may say that 'equivalent types are identical'."[3].

Homotopy type theory, an amalgam of type theory; homotopy theory and higher dimensional category theory, is *intentional, dependent* type theory with the *Univalence Axiom* forming the basis of its *structural identity principle.* Formally, this means that

$$(A \cong_C B) \to Id_C(A, B)$$

where C is the type of structured objects with arbitrary signature, $(A \cong_C B)$ is the type of isomorphisms between A and B and $Id_C(A, B)$ is the type of objects that witness the identification of A with B. In other words, ***Isomorphic structures can be identified***.

The interpretation of types as structured objects also means that it is possible to apply type-theoretic logic to other structured objects — namely topological objects, leading to a homotopical interpretation and the ability to reason about homotopy theory using type theory. This homotopical interpretation of types coupled with the *Univalence Axiom* is the basis of homotopy type theory.

5.2 Key Concepts in Homotopy Type Theory

The fundamental idea behind homotopy type theory is the marriage between Martin-Löf type theory and homotopical notions. The following uses the languages of type theory, category theory and homotopy theory to present concepts in Homotopy type theory.

5.2.1 Types and their Inhabitants

Types as spaces

In a homotopical interpretation, types are regarded as topological spaces and terms of a type are represented by points in a space.

Types as ∞-Groupoids

In algebraic topology, a (pointed[1]) space has an associated group that stores information about the 'basic shape' of the space. More formally, it stores information on when two paths, beginning and ending at a basepoint can be continuously deformed onto one another. Naturally, it follows that for homeomorphic spaces, this group will be the same. This group is known as the *fundamental group.*

In homotopy theory, every space has a *fundamental ∞-groupoid* whose k-morphisms (morphisms between morphisms at level k) are the k-dimensional paths in the space. Homotopy type theory views types as having the structure of this *fundamental ∞-groupoid.* This can be seen through iterating the identity type in the following way: given an identity type $a =_A b$ we can consider the type of identified elements : $c =_{(a=_A b)} d$, that is, identifications between identifications and so on.

This structure is analogous to continuous paths of a space and the homotopies between them, in other words, an ∞-groupoid. This groupoid model of Martin-Löf type theory has far reaching implications in higher category theory as groupoids can be modelled as small categories in which all morphisms are isomorphic.

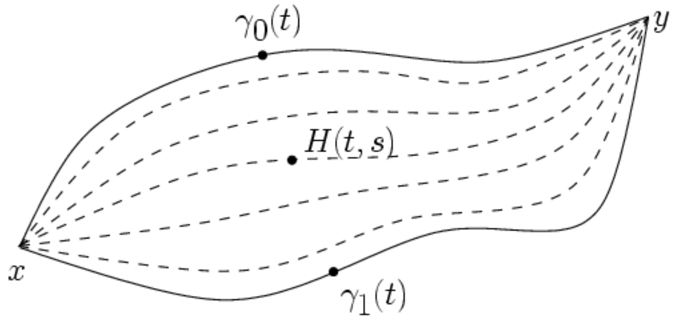
5.2.2 Equality

Given the above, it is consistent to interpret identity types as paths. Thus, for an element $p : a =_A b$, a path $p : a \rightsquigarrow b$ is said to exist between a and b in the

[1]

A pointed topological space is a space X such that it has a distinct basepoint $x_0 \in X$. This basepoint will often dictate the *fundamental group.*

space of A. For two elements of the same identity type i.e. for $p, q : a =_A b$, paths are said to be parallel as they have the same start and end points. Thus, it follows that an element of the type $c =_{(a=_A b)} d$ represents a homotopy — or a morphism between morphisms. This can also be thought of as a 2-dimensional path in A. Similarly, the next iteration, the type of identifications between identifications between identifications can be thought of as a 3-dimensional path or a morphism between morphisms between morphisms.



Homotopy between two curves: the curves γ_0 and γ_1 are homotopic by the homotopy H

In the image above, γ_0, γ_1 could be considered of the type $x = y$ for some space.

5.2.3 Functions

Functions are represented by continuous mappings between spaces. For example, a function $f : A \to B$ represents a continuous mapping from space A to space B. Functions can be identified if they are homotopic.

Functions also behave functorially on paths, that is, all functions are continuous and preserve paths.

5.2.4 Dependent types

Dependent types are represented by a fibration - a mapping from a base space to a total space.

5.2.5 Higher Inductive Types

Multisets as a HIT

$$S_M : Type_1$$

$$e : \mathbb{N} \to S_M$$
$$\epsilon : Fin(m) = Fin(n) \to e(m) = e(n)$$

And family:

$$P_M : S_M \to Set$$
$$P_M(e(n)) = Fin(n)$$
$$P_M(\epsilon(\alpha)) = transport(Fin, \alpha)$$

5.3 Summary

The following table summarises the different interpretations of analogous concepts in the languages of type theory, logic, Zermelo-Fraenkel set theory and homotopy type theory:

Type Theory	**Logic**	**ZFC**	**Homotopy Theory**
$\mathbf{0}$	$\perp$	$\varnothing$	$\varnothing$
$\mathbf{1}$	$\top$	$\{\varnothing\}$	$*$
A	proposition	set	topological space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
Id_A	equality	$\{(x,x) \mid x \in A\}$	path space A^I
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists x : A, B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall x : A, B(x)$	product	space of sections

Chapter 6

Containers

6.1 Introduction

6.2 Definition

A unary container is defined as a pair $(S \triangleright P)$ where:

1. $S : Set$ and elements of S are the *shapes* of the container.
2. $P : Set \to Set$ and for $s : S$, elements of $P(s)$ are the *positions* of s.

6.2.1 List Container

As an example, we define a list container, Con_{List} for conventional lists over a type A:

$$L(A) \simeq 1 + A \times L(A)$$

or in Agda:

```
data List (A : Set) : Set where
     Nil : List A
     Cons : A -> List A -> List A
```

$Con_{List} = (S \triangleright P)$ such that S, the set of shapes, is $\mathbb{N}$, the set of natural numbers and P, the family of positions is $Fin(-)$. The functor $[\![Con_{List}]\!]$ maps a set to the set of finite sequences in A.

6.3 Category of Containers

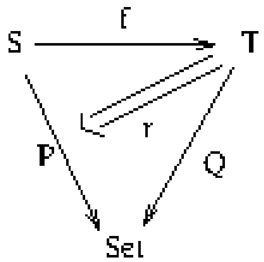
The category of containers is composed of containers as objects and container morphisms. A container can be interpreted as an endofunctor (a categorical mapping from a category to itself):

$$[\![S \triangleright P]\!] : Set \to Set$$
$$[\![S \triangleright P]\!]X = \Sigma s : S.Ps \to X$$

and given containers $(S \triangleright P)$ and $(T \triangleright Q)$ morphism $f \triangleright r$ is given by: $Mor_{Con}(S \triangleright P), (T \triangleright Q) =$

$$\sum_{f:S \to T} r : \prod_{x \in S} Q(f(x)) \to P(x)$$

Pictorally, this is represented as:



These container morphisms give rise to natural transformations, morphisms of functors. Given containers $(S \triangleright P)$ and $(T \triangleright Q)$, and a container morphism $(f, r) : (S \triangleright P) \to (S \triangleright P)$ we can define a natural transformation:

$$[\![f, r]\!] : [\![S \triangleright P]\!] \Rightarrow [\![T \triangleright Q]\!]$$
$$[\![f, r]\!](s, p) = (f(s), p \circ r_s)$$

6.4 Quotient Containers

6.4.1 Definition

Quotient containers are containers furnished with an equivalence relation. More formally, a quotient container $S \triangleright P/G$ is defined as a container $S \triangleright P$ and for each shape s in S, a set of isomorphisms $G(s)$. This set of isomorphisms can be seen as some subgroup of the automorphism group of $P(s)$.

From this definition, it follows that all containers are degenerate quotient containers where $G(s)$, the set of isomorphisms, is the empty set.

6.4.2 Extention

The extention of a quotient container $S \rhd P/G$ is a functor:

$$T_{S \rhd P/G} : Set \to Set$$
$$T_{S \rhd P/G}(x) = \Sigma s : S.(P(s) \to X)/\sim_s, \text{ such that}$$

$\sim_s$ is the equivalence relation on the set of functions $P(s) \to X$, such that

$$f \sim f' \text{ iff } \exists g \in G(s) \text{ where } f' = f \circ g$$

Chapter 7

δ for Data

The idea of differentiation of parametric datatypes was introduced by Conor McBride in 2001. Expanding on Gerard Huet's *Zipper* data structure[10], McBride established that the differential of a type is the ***type of its one hole contexts***. That is, if we take an arbitrary type parameterised over a type X, removing an item of this type X from the data structure and replacing it with a 'hole' or empty element will yeild its *one hole context*. Replacing this hole with an element of type X will produce the original type. This can be easily represented pictorally:

DIAGRAM

This chapter borrows its title from a 2005 paper by Abbott, Altenkirch, Ghani and McBride that serves to explain and analyse derivatives of data structures.

7.1 Differentiating Data Structures

The formalisation of the differentiation of data types is given by way of example. To find the differential of a type $FX = F_1X \times F_2X$, we must find the type of its one hole context. In the case of this type, a hole is:

- a hole in F_1, or
- a hole in F_2.

Thus, the type of δF, is the differential of F_1 and F_2 or the differential of F_2 and F_1. More formally,

$$\delta F \simeq \delta F_1 \times F_2 + F_1 \times \delta F_2$$

We may picture this as:

DIAGRAM

It is worth comparing the rule for differentiating this type F with the product rule from classical, differential calculus:

$$\delta F \simeq \delta F_1 \times F_2 + F_1 \times \delta F_2$$
$$\text{c.f.}$$
$$\tfrac{d}{dx}(u \cdot v) = \tfrac{dv}{dx} \cdot u + v \cdot \tfrac{du}{dx}$$

On observation, it would appear that these are analogous. Indeed they are and all usual rules and laws of differential calculus such as the chain rule, product rule, et cetera apply to all parametric type contructors. This is fully investigated in (Abbot, 2003).

7.1.1 Differentiation of a List

By way of additional example for further illumination, we differentiate a list. Given the list functor,

$$L(X) \simeq 1 + X \times L(X),$$

we may think of its type of one hole contexts to be the type where a hole is placed at an arbitrary position in a non-empty list. A list with a hole placed in the centre would give us two equally sized lists, while edge cases, in the form of holes at the head and tail of the list, would yeild a non-empty list and an empty list. Thus, our intuition tells us that the differential of a list is two lists. More formally, this is expressed as:

$$\delta L(X) \simeq L(X) + X \times \delta L(X)$$

7.2 Differentiating Containers

Similar to the notion of "types of one hole contexts", the derivative of a container is the container that comprises all posible ways to remove a position from that container.
For a container $(S \triangleright P)$,

$$\delta(S \triangleright P) = (s,p)\Sigma s : S.P(s) \triangleright \Sigma p' : P(s), p \neq p'.$$

Considering the extention of a container, $T_{S \triangleright P/G}(x) = \Sigma s : S.(P(s) \to X)/ \sim_s$, we may express the differentiation of a container as:

7.2.1 Differentiation of a List Container

$$
\begin{gathered}
L = (S \rhd P) \text{ such that } S = \mathbb{N}, P = Fin(-) \\
\delta(L) = \Sigma n : \mathbb{N}, p : Fin(n) \rhd \Sigma p' : Fin(n), p \neq p' \\
\simeq \\
(m, n) : \mathbb{N} \times \mathbb{N} \rhd Fin(n) + Fin(n) \\
= (L \times L)
\end{gathered}
$$

Chapter 8

Agda

Agda is a programming language come proof assistant based on Martin-Löf Type Theory (See **Chapter 4**).

Part III

Quotient Containers in Homotopy Type Theory

Chapter 9

Implementation

Part IV

Conclusions

Chapter 10

Evaluation

Chapter 11

Conclusion

Chapter 12

Further Work

Bibliography

[1] Per Martin-Löf, *Intuitionistic Type Theory*, 1980

[2] Bengt Nordström, Kent Petersson, Jan M. Smith, *Programming in Martin-Löf's Type Theory, An Introduction*, 1990

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

[4] Michael Abbott, Thorsten Altenkirch, Neil Ghani, *Constructing Strictly Positive Types*. 2004.

[5] Michael Abbott, Thorsten Altenkirch, Neil Ghani, Connor McBride, *δ for Data, Differentiating Data Structures*. 2005.

[6] Hakon Robbestad Gylterud, *Symmetric Containers (MSc Thesis)*. 2005.

[7] Thorsten Altenkirch, Paul Levy, Sam Staton, *Higher Order Containers* 2010.

[8] http://homotopytypetheory.org/

[9] http://mathworld.wolfram.com/RussellsAntinomy.html

[10] http://en.wikibooks.org/wiki/Haskell/Zippers

Chapter 13

Appendix