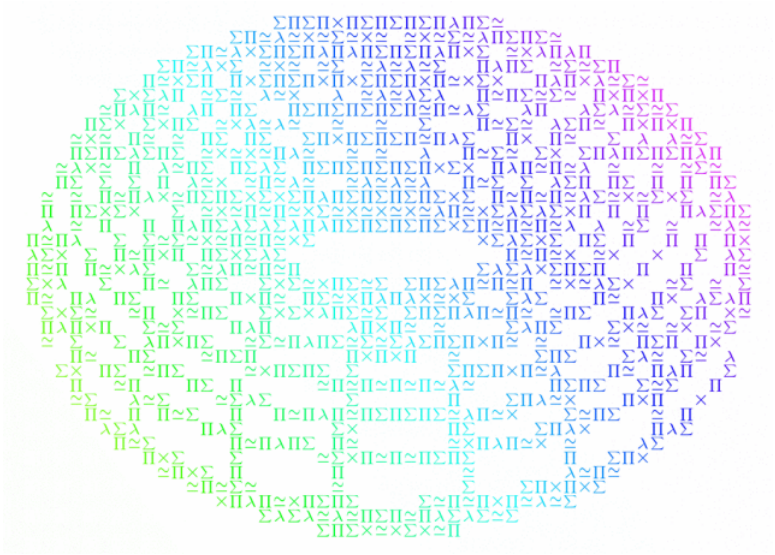
Quotient Containers in Homotopy Type Theory

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

Chapter 0

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 multiset Containers and cycle Containers. Differentiation and anti-differentiation of these quotient Containers are consequently explored and an Agda implementation of multisets is also presented, making use of Homotopy type theory's powerful higher inductive types.

Acknowledgements

This study would not have been made possible without the following people:

Prof. Thorsten Altenkirch for his support and help as my dissertation supervisor.

My parents, for their love and support.

Special thanks go to Nicolai Kraus and Ambrus Kaposi of the Functional Programming Lab, for their help over the duration of the project and for always making time for me.

Samuel Ballance for "all of [his] support, both emotional and structural".

Also, Ted, Mikey, Sam, India and Standard.

Part I

Introduction, Preliminaries and Meta-Comment

Chapter 1

Project Overview

This section serves to provide a brief overview of the project itself covering the structure, aims and objectives and motivation for this work.

1.1 Background and Motivation

Containers were first introduced in 2003 by Abbot, Altenkirch and Ghani[4]. In this, they lay the groundwork for future work through formalising containers as a template with which strictly positive types, shapely types and other types can be represented.

The notion of containers was then generalised in 2005 by Altenkirch and Morris[8] to capture not only strictly positive types, but also strictly positive families with *indexed containers*.

2011 saw Gylterud[7] further expand the container family with the introduction of *symmetric containers*. A more generalised notion of a container such that the set of shapes is a groupoid.

This project would see a formalisation and exploration of *quotient containers* building on the above and adding to the container family while exploring their usage in Homotopy type theory — a new branch of mathematics developed by The Univalent Foundations Program at The Institute for Advanced Study, Princeton during 2013.

The undertaking of this project not only allows me to work within well established fields of study such as type theory and category theory, utilising the work of influential academics but also affords me the opportunity to explore and research a nascent and bleeding edge are of theoretical computer science.

A successful project will also have applications in a number of ways. A sound implementation of multisets in Agda could potentially be appended to the Agda

Homotopy type theory library, providing a resource for further research. In addition, it adds more documentation to a young branch of mathematics and has potential implications in other fields such as datatype-generic programming.

Further background is given in the relevant sections of **Part II**.

1.2 Description of Work

Aims and Objectives

The initial aims and objectives that were outlined in my project proposal were as follows:

- To research and understand Homotopy Type Theory
- To research and understand Quotient Containers and their applications
- To formalise the notion of Quotient Containers within the context of Homotopy Type Theory
- To potentially explore applications of Quotient Containers within the context of Homotopy Type Theory
- To potentially explore interesting or novel theorems surrounding Quotient Containers in Homotopy Type Theory

While these aims remain largely valid, as the project has progressed my aims have become more focussed and less diffuse. Keeping the above strongly in mind, the primary aim of this project is to produce an Agda implementation of a multiset container in homotopy type theory. Secondarily, this written work should serve to evidence the research and understanding necessary to produce such an implementation while acting as a suitable introduction to key concepts in the subject area.

Given the abstract and theory oriented nature of the project it is difficult to give concrete aims and objectives to the code produced other than that it should be well written and fit for purpose. The code should also be written with an eye on ease of maintainability and extensibility.

Work Accomplished

While a full implementation of multiset Containers in Homotopy Type theory would have been ideal, over the course of the project it became apparent that this would be very much beyond the scope of an undergraduate dissertation.

However, this project presents a large step in fully implementing multiset Containers in Homotopy type theory, leaving only a few holes in need of completion. These holes require detailed knowledge and deeper understanding of the nuances of Agda and the HoTT Agda library that was simply infeasible to garner within the time constraints. The structure of and main idea behind the higher inductive type is present and a reasonable proportion of the proof is complete, but the remainder is left to be implemented.

In addition, this document offers insight into some of the research undertaken including research into Containers, Martin Löf Type theory, Homotopy Type theory and differentiation of data structures including cycles and multisets.

1.3 Structure of Report

This section serves to introduce the layout and organisation of the rest of the document. The understanding of this project requires a background knowledge in a number of different areas. **Part II** (Chapters 2 through 6) aims to cover those key areas and provide the reader with enough information for the subsequent part. **Part II** also presents an exposition and summary of some of the research performed during the project. **Part III** is an overview of my implementation and an examination of the novelties and usage of quotient containers in homotopy type theory. **Part III** also discusses the derivatives and anti-derivatives of quotient containers including an insight into Sattler's work. The document concludes with **Part IV** which summarises the achievements of the project and gives a critical appraisal and personal reflection of the work. Finally, there is a brief overview of further work that could be performed in the area.

Part II

Research

Chapter 2

Martin-Löf Type Theory

2.1 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.

2.2 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.

2.3 Type Connectives

Finite Types

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

- the **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 **0**-type may be referred to as a *contradiction* as there is no way to prove a contradiction and nor can the **0**-type be constructed.
- the **1**-type. Also called the *unit* type. It has only one inhabitant and invoking the Curry-Howard isomorphism, it represents True.
- the **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.

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 $b : B$. Invoking the Curry-Howard isomorphism, product types represent the logical conjunction of two operands.

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)` : $A + B$

[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.

for $b : B$. Invoking the Curry-Howard isomorphism, product types represent the logical disjunction of two operands.

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.

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.

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`, $\texttt{refl}_{\texttt{a}} : (a =_A b)$.

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 $successor : \mathbb{N} \to \mathbb{N}$.

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 $\mathcal{U}_i$.

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	$\sum_{x:A} P(x)$

[2] In 1901, Bertrand Russell used this paradox to derive a contradiction in Cantor's naïve 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."[14] Type theory was introduced by Russell as a solution to this problem.

Chapter 3

Agda

3.1 Introduction

Agda is both a programming language and proof assistant based on Martin-Löf Type Theory (See **Chapter 2**). Developed by Ulf Norell at Chalmers University of Technology, Agda has dependent types and Haskell inspired syntax. Among its other major features are MLTT's logical framework and universes, implicit arguments, and support for coinductive data types. Agda compiles via Haskell and has a powerful emacs interface that can allow the programmer to type check an unfinished program and use tactics and holes to procure useful information on how to complete missing parts. Agda also has support for unicode characters and mixfix operators.

There are a number of benefits in using Agda, including the elimination of errors at runtime, the ability to program with types as data and automatic proof checking as well as all the usual benefits of using a statically typed functional programming language.

As part of the *Univalent Foundations Program*, a Homotopy type theory and univalent foundations library was developed in Agda to provide key tools in HoTT to programmers[16].

Agda is used throughout this document and a primary aim of this project is to produce a piece of Agda code. As such, this chapter will serve as a brief overview of and introduction to Agda through a number of worked examples.

3.2 Type Declarations

One of the simplest types we can define is Bool. In Agda, this is defined:

```
data Bool : Set where
     true : Bool
     false: Bool
```

This definition states the following:

- The type Bool is a set
- The type Bool has two constructors:
  1. true
  2. false
- The type Bool has no other constructors
- true in the type bool is not equal to false in the type bool

3.2.1 Inductive Types

Inductive types are also available in Agda, such as this familiar definition of $\mathbb{N}$, the set of natural numbers:

```
data Nat  : Set where
     zero : nat
     succ : (n : nat) -> nat
```

Inductive types are touched upon later when discussing Homotopy type theory's higher inductive types (**4.2**).

3.3 Functions

Simple functions are defined in a very similar manner to Haskell. Here we define familiar functions — the *if then else* construct and the *map* function:

```
if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true then x else y = x
if false then x else y = y

map : {A B : Set} -> (A -> B) -> List A -> List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

These functions show various aspects of Agda's syntax. The `if_then_else` a parametricly polymorphic function that is exhaustively defined and makes use of pattern matching on wildcards(_) while `map` is a good example of a polymorphic function that utilises recursion and pattern matches on more complex arguments, namely a list matched by `x` — the head of a list; the cons operator (_::_) and `xs` — the tail of the list.

3.4 Dependently typed functions

Given the dependent type of finite sets, defined as an inductive family as follows:

```
data Fin : Nat -> Set where
   fzero : {n : Nat} -> Fin (succ n)
   fsucc : {n : Nat} -> Fin n -> Fin (succ n)
```

that is, *Fin n* contains exactly *n* elements, e.g. `Fin 3 = fzero, fsucc fzero, fsucc (fsucc fzero)`, we can define the dependent function `fromNat`, which takes a natural number, `n` and produces `Fin n`.

```
fromNat : (n : Nat) -> Fin (succ n)
  fromNat zero = fzero
  fromNat (succ n) = succ (fromNat n)
```

3.5 Conclusion

The above has been a *very* brief introduction to the basics of agda but should be sufficient to keep this document self contained. Although fuller understanding of this project would require further study of the Agda standard library and the Homotopy type theory library.

Chapter 4

Homotopy Type Theory

4.1 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.

4.2 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 mathematics, type theory, category theory and homotopy theory to present concepts in Homotopy type theory.

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.

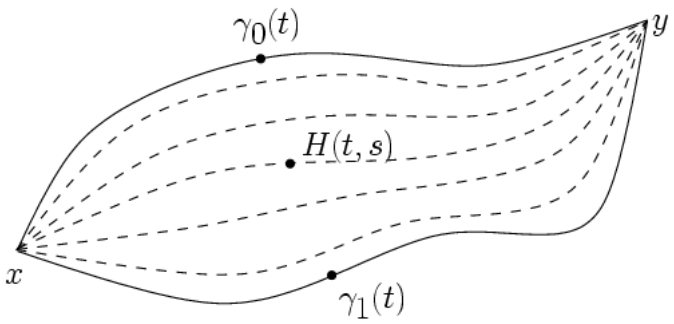
This structure is analogous to continuous paths of a space and the homotopies between them, in other words, an ∞-groupoid. This groupoid model of Martin-Löf type theory has far reaching implications in higher category theory as groupoids can be modelled as small categories in which all morphisms are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

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.

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.

Dependent types

Dependent types are represented by a fibration - a mapping from a base space to a total space. Fibrations are continuous maps that satisfy the *homotopy lifting property*, that is, a homotopy (or path) that occurs in the base space can be lifted to the total space.

Higher Inductive Types

Inductive types are those that include instructions on how to produce elements of the type within its definition. The classical example of an inductive type is the

type of natural numbers, *nat* or $\mathbb{N}$. In agda, the definition of *nat* is as follows:

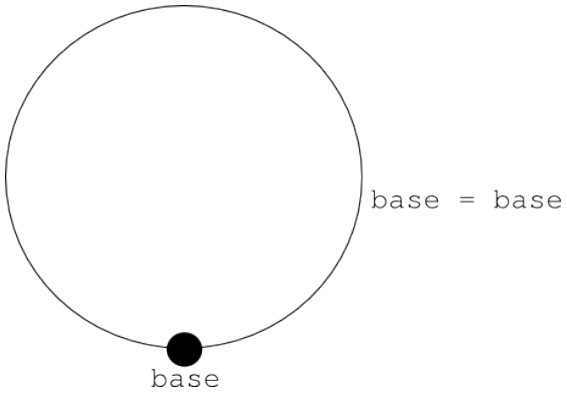
```
data nat  : Set where
     zero : nat
     succ : (n : nat) -> nat
```

Here, we define a base case `zero` and an inductive case which tales an element in `nat` and returns a new element — the successor of that natural number.

Similarly to conventional inductive types, higher inductive types provide instructions to generate not only points of a type but also paths and higher paths (paths between paths) in that type. The classical and most intuitive example of a higher inductive type is $\mathbb{S}_1$, the circle. $\mathbb{S}_1$ is defined by two constructors:

```
base :  𝕊₁
loop :  base =𝕊₁ base
```

Here, we define a base point, `base` the *point constructor* and `loop`, a path from the base point to itself, the *path (higher) constructor*. We may visualise this as follows:



4.3 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:

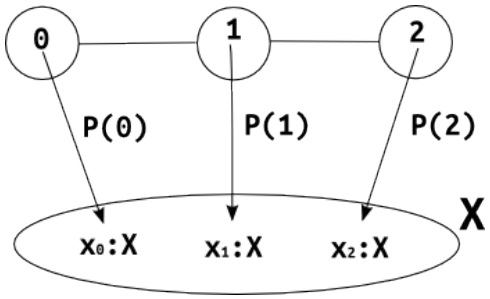
Type Theory	**Logic**	**ZFC**	**Homotopy Theory**
$\mathbf{0}$	$\bot$	$\varnothing$	$\varnothing$
$\mathbf{1}$	$\top$	$\{\varnothing\}$	$*$
A	proposition	set	topological space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
Id_A	equality	$\{(x, x)\|x \in A\}$	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 5

Containers

5.1 Introduction

The notion of Containers was first introduced in 2003 by Abbot, Altenkirch and Ghani[4] as a common 'template' with which many data types can be modelled. The fundamental idea of a container is that a data type can be represented with a pair consisting of a set of ***Shapes*** and a family of ***Positions***. The set of *shapes* provides a description of the structure of the type and the *positions* are mappings from positions in the shape to individual points of data. This can be visualised as below:



For some arbitrary container parameterised over a type X, P(x) maps a shape to an element in X for x in X.

5.2 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 $\rightarrow$ Set and for $s : S$, elements of $P(s)$ are the *positions* of s.

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
```

$\text{Con}_{List} = (S \rhd P)$ such that S, the set of shapes, is $\mathbb{N}$, the set of natural numbers and P, the family of positions is $\text{Fin}(-)$. The functor $[\![\text{Con}_{List}]\!]$ is interpreted as a map from a set to the set of finite sequences in A.

5.3 Operations on Containers

Product Containers

Given two containers $(S \rhd P)$ and $(T \rhd Q)$, a product container is defined as:

$$(S \rhd P) \times (T \rhd Q) = (s, t) : (S \times T) \rhd (P(s) + Q(t))$$

Interpretation:

$$[\![(S \rhd P) \times (T \rhd Q)]\!](A) = [\![(S \rhd P)]\!](A) \times [\![(T \rhd Q)]\!](A)$$

Coproduct Containers

Given two containers $(S \rhd P)$ and $(T \rhd Q)$, a coproduct container is defined as:

$$(S \rhd P) + (T \rhd Q) = (x : S + T) \rhd (\texttt{if (x = inl(s) then } P(s) \texttt{ else}\dagger\ Q(t) \texttt{)})$$

† The else branch implicitly means that `x` must `= inr(t)`.

Interpretation:

$$[\![(S \rhd P) + (T \rhd Q)]\!](A) = [\![(S \rhd P)]\!](A) + [\![(T \rhd Q)]\!](A)$$

Composite Containers

Given two containers $(S \rhd P)$ and $(T \rhd Q)$, a composite container is defined as:

$$(S \rhd P) \circ (T \rhd Q) = (\Sigma s : S, f : P(s) \to T) \rhd (\Sigma p : P(s).Q(f(p))))$$

Interpretation:

$$[\![(S \rhd P) + (T \rhd Q)]\!](A) = [\![(S \rhd P)]\!]([\![(T \rhd Q)]\!](A))$$

5.4 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):

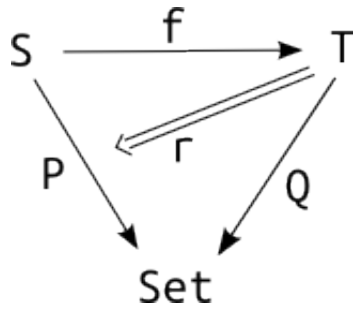
$$[\![S \rhd P]\!] : \text{Set} \to \text{Set}$$
$$[\![S \rhd P]\!]X = \Sigma s : S.Ps \to X$$

and given containers $(S \rhd P)$ and $(T \rhd Q)$ morphism $f \rhd r$ is given by:

$\text{Mor}_{Con}(S \rhd P), (T \rhd Q) =$

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

Pictorally, this is represented as:



As an example of a container morphism we define *reverse* for list containers:

$$\text{reverse} : \Pi_{A:Set}\text{List}(A) \to \text{List}(A)$$
$$\text{reverse} = \lambda n.n \rhd \lambda n, i.n - 1$$

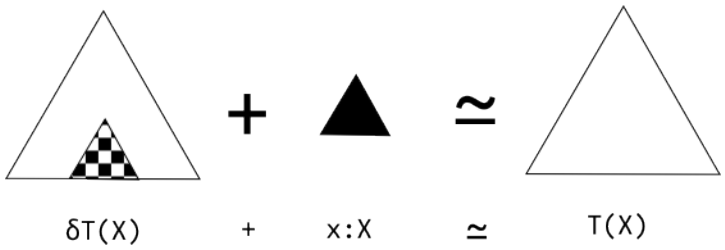
These container morphisms give rise to natural transformations, morphisms of functors. Given containers $(S \rhd P)$ and $(T \rhd Q)$, and a container morphism $(f, r) : (S \rhd P) \to (S \rhd P)$ we can define a natural transformation:

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

Chapter 6

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[15], 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:



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.

6.1 Differentiating Data Structures

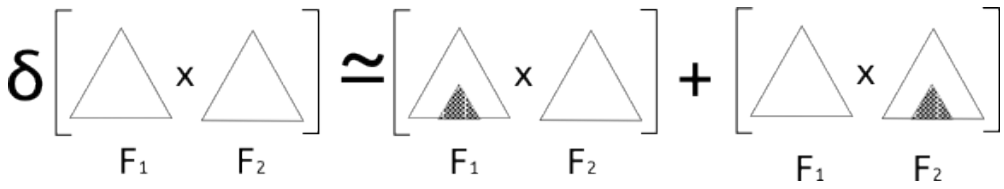
The formalisation of the differentiation of data types is given by way of example. To find the differential of a type $FX = F_1X \times F_2X$, we must find the type of its one hole context. In the case of this type, a hole is:

1. a hole in F_1, or
2. 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:



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

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

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 constructors. This is fully investigated in Abbot, 2003[4].

6.1.1 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 result of differentiating a list is two lists. More formally, this is expressed as:

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

6.2 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.

Thus, for a container $(S \rhd P)$,

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

6.2.1 Differentiation of a List Container

By way of example, we differentiate a list container. Given a list container L:

$$L = (S \rhd P) \text{ such that } S = \mathbb{N}, P = \text{Fin}(-),$$

As in **6.2**:

$$\delta(L) = \Sigma n : \mathbb{N}, p : \text{Fin}(n) \rhd \Sigma p' : \text{Fin}(n), p \neq p'$$

This effectively splits the list, yielding

$$(m, n) : \mathbb{N} \times \mathbb{N} \rhd \text{Fin}(n) + \text{Fin}(n),$$

a pair of lists (m, n) such that m `++` n would give the original list. As such, $\delta(L) = (L \times L)$, which is what we would expect from differentiating a list as in **6.1.1**.

Part III

Quotient Containers in Homotopy Type Theory

Chapter 7

Quotient Containers

7.1 Definition

Quotient containers are containers furnished with an equivalence relation. More formally, a quotient container $(S \rhd P)/G$ is defined as a container $(S \rhd P)$ 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 a degenerate form of quotient containers such that $G(s)$, the set of isomorphisms, is the empty set.

7.2 Extension

The extension of a quotient container $(S \rhd P)/G$ is a functor:

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

$\sim_s$ 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 8

Multisets

8.1 Definition

Multisets are generalised notions of sets such that members can appear more than once. The number of times an element appears in a multiset is said to be the *multiplicity* of that element. More formally, we can define a multiset as follows:

$$\text{Multiset}(X) : X \to \mathbb{N}$$

That is, a multiset parameterised over a type X when given $x : X$ will return its multiplicity.

8.2 Multiset Container?

By taking all elements of a multiset and putting them into a list we obtain a representation of that multiset. However, these representations are not comparable as there is no inherent order to a multiset whereas a list is strictly ordered. Given a list representation of a multiset of size n, we must make $n!$ checks for equality in order to check all permutations of the list. With this in mind, it becomes clear that a bijection exists between multisets and lists, quotiented by permutations. In other words:

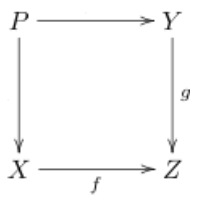
$$\begin{gathered}\text{Multiset}(A) = \text{List}(A)/\sim \text{ such that:} \\ x \sim y \text{ iff } x \text{ is a permutation of } y\end{gathered}$$

Hence, it is possible to define a quotient container: $(S \rhd P)/Q$ where $(S \rhd P)$ is the List container and for $n : \mathbb{N}$, $Q(n)$ is the set of isomorphisms on $Fin(n)$.

However, it has been shown that containers always preserve pullbacks[9][1] , but this representation of multiset containers does not[12]. Thus, it is not possible to adequately represent a multiset as a container in conventional type theory.

However...

[1] A pullback P of morphisms f, g in the commutative diagram:



is given by $P = \Sigma y : Y \Sigma x : X(f(x) = g(y))$

Chapter 9

In Homotopy Type Theory, Quotient Containers become ordinary Containers

In Homotopy type theory, if we allow the Shapes of a container to be an arbitrary type and do not restrict it to a set, quotient containers become ordinary containers.

We now briefly recap on two important points:

1. Homotopy type theory utilises Voevodsky's *univalence axiom* (see **4.1**) as the basis of its *structural identity principle* and a homotopy theoretic interpretation of identity is that *isomorphic structures can be identified.*
2. A quotient container is a pair consisting of a container $(S \rhd P)$ and a set of isomorphisms (G(s)).

Reflecting on these, it becomes clear that the set of isomorphisms become identities and effectively we get equivalences 'for free'.

9.1 Multiset Container in HoTT

With the above in mind, it is now possible to define a multiset container without issue. We do so using a higher inductive type[12] (see **4.2**):

$Con_{Multiset} = (S_M \rhd P_M)$ such that

$$S_M : \text{Type}_1$$
$$e : \mathbb{N} \to S_M$$
$$\epsilon : \text{Fin}(m) = \text{Fin}(n) \to e(m) = e(n)$$

and

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

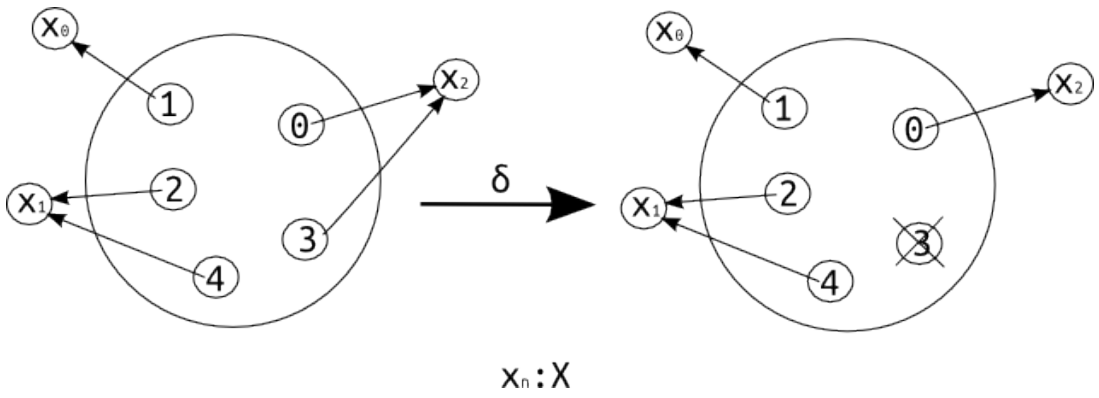
9.1.1 The derivative of a multiset is a multiset

Given $(S_M \rhd P_M)$ as above,

$$\begin{aligned} &\delta(S_M \rhd P_M) \\ &= (\Sigma e(n) : S_M.i : P_M(n)) \rhd (P_M(n) - i) \\ &= S_M \rhd P_M(n) \end{aligned}$$

Thus, $\delta(S_M \rhd P_M) = (S_M \rhd P_M)$.

The intuition here is that removing an element from a multiset does not alter its structure in the same way as removing an element from a list. We may picture the differentiation of a multiset of size 5 as:



While an arbitrary element is removed, this removal has no bearing on the other elements as there is no strict ordering or structure as in a list. As such, the result of differentiating a multiset is a multiset — the context of the hole remains a multiset.

Chapter 10

Formalisation in Agda

10.1 Definition

The implementation of multiset containers used in this project is slightly different from the definition used in **Chapter 9**. The definition is as follows:

```
data Sym (n :  Nat)
     * :  Sym n

     p :  (Fin n ≃ Fin n) → * = *
     tr :  (f₁ , f₂ :  Fin n ≃ Fin n)
           → p (f₁ ∘ f₂) = (p f₁)∘(p f₂)

     h :  is-1-type(Sym n)

     P n :  Sym n → Type of 0-Types
     P n (*) = Fin n
     P n (p(f)) = univalence f
```

This implementation sees the definition of a type `Sym n` which corresponds to the symmetric (permutation) group on the n-element set and acts as the shape type of the construction. The path constructors define paths from the `*` point constructor to itself and represent permutations.

10.2 Example

Consider two multisets x, y such that $x = \{A, B, B\}$ and $y = \{B, B, A\}$. Alternately, we may represent them in the form {length, * : Sym, mapping function}, as follows:

$$x = \left\{ 3, \quad *, \quad f = \begin{array}{l} 0 \mapsto A \\ 1 \mapsto B \\ 2 \mapsto B \end{array} \right\}, \; y = \left\{ 3, \quad *, \quad g = \begin{array}{l} 0 \mapsto B \\ 1 \mapsto B \\ 2 \mapsto A \end{array} \right\}$$

For two multisets to be the same in the above representation, all elements of the 3-tuple must be equivalent. While 3 in x is trivially the same as 3 in y by reflexivity, x and y appear completely disparate. However, when we transport through the equivalence for $*$ they become the same and as $*$ in x and $*$ in y are equivalent, we can therefore establish that $x = y$.

10.3 Code

The Agda code itself is of the form:

```
module Sym (n :  Nat)
       * :  Sym

       postulate
              p :  (Fin n ≃ Fin n) → * = *
              tr :  (f₁ , f₂ :  Fin n ≃ Fin n)
                    → p (f₁ ∘ f₂) = (p f₁)∘(p f₂)
              h :  is-1-type(Sym n)

       module Sym-Recursor (n :  Nat) {i} (A : Type i)
              (base :  A)(loops :(Fin n ≃ Fin n) → (base = base))
              (comp-loops :  (f₁ f₂ :  Fin n ≃ Fin n) →
                    loops (f₂ ∘ f₁) == (loops f₁)∘(loops f₂))
              (h :  has-level (S (0)) A)

       module Pos (n :  Nat) where
               paths :  Fin n ≃ Fin n  →
                    (Fin n , fin-is-set n) == (Fin n , fin-is-set n)
               p-trans :  (f₁ f₂ :  Fin n  ≃  Fin n)  →
                     paths (f₂∘f₁) = paths f₁ ∘ paths f₂
```

Here, we do not define the eliminator as it is sufficient to define the recursor.

The full Agda code can be found in the Appendix, (**13.2**).

Chapter 11

Anti-derivatives of Quotient Containers

Another important quotient container is the cycle or ring. This chapter briefly covers cycles as quotient containers and discusses work done by Sattler on the anti-derivatives of cycles.

11.1 Anti-derivatives

Anti-derivatives, as the name suggests, are the inverse of derivatives. Gylterud asked to what extent anti-derivatives exist for containers[7], noting that in the general case, such that we restrict the shapes of a container to be of type **Set**, anti-derivatives do not exist. However, he discovered that all analytic containers have anti-derivatives. That is, all containers such that

$$[\![(S \triangleright P)]\!] = [\![\Sigma n : \mathbb{N}.T_{(S \triangleright P)}(n) \triangleright \mathrm{Fin}(n)]\!]$$

where

$$T_{(S \triangleright P)} : \mathbb{N} \to \mathrm{Set}$$
$$T_{(S \triangleright P)}(n) = \delta^n(S \triangleright P)(0)/S_n$$

such that $T_{(S \triangleright P)}$ represents the Taylor series associated with $(S \triangleright P)$, have anti-derivatives given by a higher inductive type with elements:

$$\Sigma n : \mathbb{N} T_{(S \triangleright P)}(n)$$

11.2 Cycles

A cycle is some permutation of a set such that mappings are performed in a cyclic manner. For example, $\{3, 0, 1, 2\}$ is a cycle of the set $\{0, 1, 2, 3\}$ as 0 is

mapped to 1, 1 is mapped to 2, 2 is mapped to 3 and 3 maps to 0. In other words, it is a permutation of a set that has an orbit containing more than one element. The length of a cycle is determined by the number of elements in it's largest orbit and we call a cycle of length k a k-cycle.

Thus, we may also view cycles as lists quotiented by rotations, such that in a rotation the n^{th} element is mapped to the $n + 1^{th}$ until the final element, which is mapped to the first.

As we may view cycles as lists quotiented by rotations, it is possible to produce a quotient container, $(S_C \rhd P_C)$, that represents cycles. As with multisets, we define a higher inductive type:

$$\begin{gathered} S_C : \text{Type}_1 \\ e : \mathbb{N} \to S_C \\ \epsilon : \text{Fin}(m) \to e(m) = e(m) \\ \delta : \epsilon(0) = \text{refl} \end{gathered}$$

And a family:

$$\begin{gathered} P_C : S_C \to \text{Set} \\ P_C(e(n)) = \text{Fin}(n) \\ P_C(\epsilon(i)) = \lambda j.i + \text{mod}(n) \end{gathered}$$

Interpretation:

$$[\![(S_C \rhd P_C)]\!](X) = \text{set of all cycles over } X$$

11.3 Finite Fields

A finite field or Galois field, $\mathbb{F}_p n$, is a field that obeys canonical field axioms and in addition has the following properties:

1. Has a prime power number of elements. The number of elements is known as the *order* of the field and the prime base is known as the *characteristic* of the field.
2. For all primes, p and positive n, $\mathbb{F}_p n$ exists.
3. Finite fields of the same *order* are isomorphic.
4. Finite fields cannot exist with a non-prime *characteristic*.

Example

A classical example is the integers, $\mathbb{Z}$ modulo a prime number. e.g. $\mathbb{Z}/2\mathbb{Z}$ has characteristic 2 as $1 + 1 = 0$. Conversely, $\mathbb{Z}/6\mathbb{Z}$ is not, as 6 is not prime.

11.4 Anti-derivatives of Cyclic Actions

In 2012, Christian Sattler showed that the differential of a finite field of order n is a cycle[11]. Strictly, the cyclic group of bijective affine transformations on the field that fix zero:

$$\{x \mapsto ax | a : \mathbb{F}, a \neq 0\}$$

Thus, as $x = \frac{\delta}{\delta x} y \Leftrightarrow y = \int x$, this means that the anti-derivative of an n-cycle is a finite field of order $n + 1$. As finite fields only exist when they are of a prime power order, in general anti-derivatives of cycles cannot exist.

e.g. a 5-cycle has no anti-derivative as an order 6 finite field cannot exist.

11.5 Anti-derivatives of Multisets

As we proved in **9.1.1** that the derivative of a multiset is a multiset and as $x = \frac{\delta}{\delta x} y \Leftrightarrow y = \int x$, trivially, this means that the anti-derivative of a multiset is a multiset. As such, it is a fixed point of the functional that is differentiation of data types.

Part IV

Conclusions

Chapter 12

Evaluation

12.1 Project Evaluation

The primary aim of this project was to produce an Agda implementation of multiset containers in Homotopy type theory. Secondary to this was to explore and understand the fields of Containers and Homotopy type theory.

While it was not possible to produce full implementation of multisets due to the scope of the project and time constraints, I believe I have made substantial progress towards a complete implementation. Given additional time to program and become more familiar with Agda and the Homotopy type theory library, I believe it would be possible to finish the code. Finishing the multiset implementation could perhaps be left for another project, either undertaken by myself in the future or by another student.

This said, it is my belief that this project has been largely successful. As previously stated, I have come some way toward a multiset implementation while exploring the theory of containers, differentiation of data structures and Homotopy type theory — documented in this work.

To expand on this dissertation I would wish to perform additional research in other areas related to Homotopy type theory. For example, out of personal interest, I would like to look into and document topos theoretic interpretations of Homotopy type theory. However, with a branch of mathematics with such far reaching implications in diverse subjects like Homotopy type theory, there will always be room for more research.

12.2 Personal Reflection

I have learned a great deal during the undertaking of this project — compounding existing knowledge of type theory and category theory while performing research in a fascinating and still blossoming area of mathematics and theoretical computer science. I have found this research very engaging and have developed a genuine interest for the subject.

Theory aside, I have also learned and improved upon invaluable skills such as time management, technical writing and task planning. Particularly so time management — while I aimed to keep rigorously to the schedule I imposed upon myself in my project proposal, It may have been prudent to revise the timetable dynamically to allow additional time where required. For example, allowing more time to get to grips with the hitherto unappreciated nuances of Agda.

Moving forward, I will take with me the experience of producing a large project while garnering and developing both technical and practical skills and making a first foray into a compelling research area.

Overall, I have found this project both interesting to undertake and richly rewarding to complete and it is my belief that I have completed it successfully.

12.3 Further Work

Naturally, the first instance of further work that could be performed is in completing the Agda implementation of multisets that was initiated in this project. Secondary to this, would be proving that the derivative and anti-derivative of a multiset is a multiset in Agda. It may also be worthwhile formalising cycles in the same manner

In addition to these, there are a great deal of open problems in Homotopy type theory to be solved, such as formalising ∞-Groupoids in HoTT or constructing HIRTs — higher inductive-recursive types.

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, *Categories of Containers*. 2003.

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

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

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

[8] Thorsten Altenkirch, Peter Morris, *Indexed Containers*, 2009.

[9] Neil Ghani, Robert Atkey, Patricia Johann, *When is a Type Refinement an Inductive Type?*, Foundations of Software Science and Comnputation Structures, 2011.

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

[11] Christian Sattler, *Quotient Container Antiderivatives of Cyclic Actions*, 2012.

[12] Thorsten Altenkirch, *Containers in Homotopy Type Theory Talk*, 2013.

[13] http://homotopytypetheory.org/

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

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

[16] https://github.com/HoTT/HoTT-Agda

Chapter 13

Appendix

13.1 Ordinary Containers in Agda and related

```
record Container : Sets where
               field
                 S : Set
                 P : S -> Set

record Functor : Sets where
               field
                 obj : Set -> Set
                 morp : (A , B : Set) -> (A -> B) -> (obj A -> obj B)

record _->c_ {A , B : Container } : Set where
               field
                 shape : Shape A -> Shape B
                 position : forall {s}

record NatTrans (F , G : Functor) : Sets
               field
                 fam : (A : Set) -> obj F A -> obj G A
                 nat :
                   f : A -> B
```

13.2 Agda Multiset Container

```
{-# OPTIONS --without-K #-}

module Multiset where
open import lib.Basics
open import lib.types.Nat
open import lib.PathOver
open import lib.NType2
open import lib.types.Sigma using (Σ-·)
open import lib.Equivalences2 using (ua-∘e)

-- should be moved to a library module later
data Fin : ℕ → Type₀ where
  zf : {n : ℕ} → Fin (S n)
  sf : {n : ℕ} → Fin n → Fin (S n)

fin-is-set : (n : ℕ) → is-set (Fin n)
fin-is-set n = {!!}

module _ (n : ℕ) where

  private
    data #Sym : Type₀ where
      #* : #Sym

  #symrec : ∀ {i} (A : Type i) → A → #Sym → A
  #symrec A a _ = a

  Sym : Type₀
  Sym = #Sym

  * : Sym
  * = #*

  postulate -- HIT definition
    p : (Fin n ≃ Fin n) → * == *
    tr : (f₁ f₂ : Fin n ≃ Fin n) → p (f₂ ∘e f₁) == (p f₁) · (p f₂)
    h : has-level (S ⟨0⟩) Sym

module SymRec (n : ℕ)
              {i}
              (A : Type i)
              (base : A)
              (loops : (Fin n ≃ Fin n) → base == base)
              (comp-loops : (f₁ f₂ : Fin n ≃ Fin n) → loops (f₂ ∘e f₁) == (loops f₁) · (loops f₂))
              (h : has-level (S ⟨0⟩) A)
                where
  rec : Sym n → A
  rec = #symrec n A base
```

```
open SymRec public using () renaming (rec to sym-rec)

-- define Positions:

module _ (n : ℕ) where

  P-aux : Sym n → hSet lzero
  P-aux = sym-rec n {lsucc lzero}
                  (hSet lzero)
                  (Fin n , fin-is-set _)
                   paths p-trans hSet-is-grp where

    paths : Fin n ≃ Fin n → (Fin n , fin-is-set n) == (Fin n , fin-is-set n)
    paths e = pair= pathaux pathaux\'  where

      pathaux : Fin n == Fin n
      pathaux = ua e

      pathaux\' : PathOver is-set pathaux (fin-is-set n) (fin-is-set n)
      pathaux\' = (from-transp is-set pathaux (prop-has-all-paths is-set-is-prop _ _))

    p-trans :  (f₁ f₂ : Fin n ≃ Fin n) → paths (f₂ ∘e f₁) == paths f₁ · paths f₂
    p-trans f₁ f₂ = paths (f₂ ∘e f₁) =⟨ idp ⟩
                    pair= (ua (f₂ ∘e f₁)) s =⟨ ap (λ x → pair= x {!!}) (ua-∘e f₁ f₂) ⟩
                    pair= (ua f₁ · ua f₂) {!!} =⟨ {!!} ⟩
     -- Σ-· , and then using that the second component is propositional
                    pair= (ua f₁) _ · pair= (ua f₂) _ =⟨ idp ⟩
                    paths f₁ · paths f₂ ∎
      where
        s = (from-transp is-set (ua (f₂ ∘e f₁)) (prop-has-all-paths is-set-is-prop _ _))


    hSet-is-grp : has-level _ (hSet _)
    hSet-is-grp = hSet-level _


  P : Sym n → Type lzero
  P = fst ∘ P-aux
```