(Q)M-types and Coinduction in HoTT / CTT

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science
June 13, 2020
Advisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This work builds on homotopy type theory (HoTT) and is formalized in a cubical type theory (CTT). We give a construction of coinductive- / M-types. Given a container, we construct a polynomial functor, which by repeated application to the unit type, gives us a sequence. Taking the limit of this sequence gives us the M-type as a final coalgebra. Since the M-type is a final coalgebra, we get a coinduction principle, making strong bisimulation for M-types imply equality. We then give some example M-types, and describe how to construct M-types from a set of constructors/destructors. We introduce quotiented M-types (QM-types), defined as set truncated quotients. However this way of constructing QM-types has the problem that we need to lift constructors of the M-type to the quotient, which requires the use of the axiom of countable choice [7]. We solve this problem by defining the quotienting relation and the type at the same time as a quotient inductive-inductive type (QIIT), which is equal to the QM-type, assuming the axiom of (countable) choice. We construct multisets and the partiality monad as some examples of quotiented M-types and show the equivalence between the two constructions of these examples. We then discuss the pros and cons of the two constructions, and relate them to an alternative construction. This alternative construction is as the final coalgebra for a quotient polynomial functor (QPF). However this construction again requires the axiom of (countable) choice. All work is formalized in Cubical Agda, and the work on defining M-types has been accepted to the Cubical Agda Github repository (pull request: `https://github.com/agda/cubical/pull/245`).

Resumé

I denne afhandling bygger vi på homotopisk typeteori (HoTT) formaliseret i en kubisk typeteori (CTT). Vi giver en konstruktion af `M`-typer ved brug af "containers". Given en "conatiner" kan vi konstruere en polynomisk funktor, og ved gentaget anvendelse af denne på enhedstypen, får vi en sekvens af typer. Grænsen af denne sekvens giver os en `M`-type som en endelig coalgebra. Vi giver nogle eksempler på `M`-typer, hvorefter vi beskriver konstruktionen af kvotient `M`-typer, kaldet en Q`M`-type. Denne konstruktion bruger set trunkeret kvotienter. Dog kræver denne konstruktion at vi kan løfte konstruktørerne for en `M`-type til kvotienten, hvilket kræver det tællelige udvalgsaksiom. For at undgå dette definere vi kvotient typen som en kvotient inductive-inductive type (QIIT), hvor typen og relationen defineres samtidig. Vi giver nogle eksempler på ækvivalence mellem de to konstruktioner, under antagelse af det tællelige udvalgsaksiom. Vi diskuttere fordele og ulemper ved disse konstruktioner, og introducere en tredje konstruktion, hvor kvotient typen er given som grænsen / den endelige coalgebra for en kvotient polynomisk functor. Denne konstruktion kræver dog stadig det endelige udvalgsaksiomet.

Acknowledgments

I would like to thank my supervisor Bas Spitters for some enlightening discussions and quick responses and for taking the time to meet weekly. I would also like to thank my fellow students and friends, who made this challenging but fun time less lonely despite the Corona pandemic.

Lasse Letager Hansen,
Aarhus, June 13, 2020.

Contents

Chapter 1

Introduction

1.1 Overview

Inductive types in the form of `W`-types has been studied a lot in the setting of homotopy type theory (HoTT) / cubical type theory (CTT). However not much work has been done on the dual concept, namely `M`-types as a formalization of coinductive types. The goal of this thesis is to get an understanding of `M`-types and extend on the existing theory in HoTT and CTT. We construct some examples of `M`-types, as an attempt to make the theory of `M`-types more accessible. A useful technique for defining a type with some properties, is quotienting a free objects, as such we will look at quotients of `M`-types in the setting of cubical type theory and to what extend this construction needs the axiom of choice.

In the rest of this chapter we will introduce some of the background theory and notation used in this thesis. In Chapter 2 we construct `M`-types from containers, and define a coinduction principle for the `M`-types. In Chapter 2.3 we give some example constructions of `M`-types. In Chapter 3 we introduce quotiented `M`-types (Q`M`-types), and show equalities between these and quotient inductive-inductive types (QIITs), we show the construction of multisets and the partiality monad as example, and discuss how quotients of `M`-types should be constructed. Finally in Chapter 4 we discuss future research and conclude on the work done. We expect readers to have entry level knowledge of category theory, type theory and homotopy type theory.

1.2 Background Theory

We give some background theory and summarize important concepts used in the rest of this thesis.

We are using **type theory** [16] as the basis for mathematics. In type theory every term x is an element of some type A, written $x : A$. A paradigm in type theory is the idea "propositions as types" [15], meaning proofs boils down to showing the existence of an element of the type representing the proposition. Specifically proofs of equality is done as a construction of an element of the equality type. The type theory we are working in is build on **Martin Löf Type Theory (MLTT)** / Intuitionistic type Theory (ITT) [14], which is designed on the principles of mathematical constructivism, where any existence proof must contain a witness. Meaning a proof of existence can be converted into an algorithm that finds the element making the statement true. MLTT is

built from the three finite types $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$, and type constructors Σ, Π and $=$. The types Σ and Π can be read as exists and for all. There is only a single way to make terms of $=$-type, and that is $\texttt{refl} : \prod_{a:A}(a = a)$. MLTT also has universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$, however we will leave the index implicit and write $\mathcal{U}$. We will say that a type is given inductively (coinductively) from a set of constructors and/or destructors, meaning we take the smallest (largest) type for which these rules are true. A constructor for a type A is a rule, that says given some arguments we can construct an element $a : A$, dually a destructor of A is a rule that says given an element $a : A$ we can construct some result. Inductive types can be defined by giving rules for the base cases and then the inductive cases. An example is the natural numbers, defined as

$$\frac{}{0 : \mathbb{N}} \quad (1.1) \qquad\qquad \frac{n : \mathbb{N}}{\texttt{succ}\ n : \mathbb{N}} \quad (1.2)$$

with base case 0, and inductive case $\texttt{succ}\ n$, which says that $n+1$ is a natural number if n is. If we take the type defined coinductively over the same set of constructors, we get the largest type with these constructors, meaning we get an element from applying $\texttt{succ}$ infinitely, namely $\infty = \texttt{succ}\ \infty$. This coinductive type is also know as $\texttt{coN}$ and differs from $\mathbb{N}$. A way to look at inductive definitions is that the rules describe the structure of the type, while for a coinduction definition, the rules describe the observable behavior of the type.

We will now describe how equality differs for inductive and coinductive types. We can make a inductively defined equivalence relation $\sim_{\mathbb{N}}$ on the natural numbers, since the natural numbers are defined inductively. Meaning equivalence on the natural numbers follows the structure defined by the constructors. The equivalence relation is given as

$$\frac{}{0 = 0}\ \sim_{0} \quad (1.3) \qquad\qquad \frac{n \sim_{\mathbb{N}} m}{\texttt{succ}\ n \sim_{\mathbb{N}} \texttt{succ}\ m}\ \sim_{\texttt{succ}} \quad (1.4)$$

where two natural numbers are equal if they both are zero or if they are equal when we subtract one from each of them. Equivalence relations for inductive types implies equality, so if two elements are related, then they are equal. We do a similar construction for the coinductive type of streams. A stream is an infinite sequence of elements. Streams can be defined from the two destructors head ($\texttt{hd}$) and tail ($\texttt{tl}$), where head represents the first element, and tail represents the rest of the sequence

$$\frac{s : \texttt{stream}\ A}{\texttt{hd}\ s : A} \quad (1.5) \qquad\qquad \frac{s : \texttt{stream}\ A}{\texttt{tl}\ s : \texttt{stream}\ A} \quad (1.6)$$

We can again define an equivalence relation $\sim_{\texttt{stream}}$, but this time coinductively, focusing on the observable behavior instead

$$\frac{\texttt{hd}\ s = \texttt{hd}\ t \quad \texttt{tl}\ s \sim_{\texttt{stream}} \texttt{tl}\ t}{s \sim_{\texttt{stream}} t} \quad (1.7)$$

This equivalence relation does not give an equality in MLTT, we just get bisimilarity meaning elements behave the same, but they are not equivalent. We solve this problem by working in a type theory, where the univalence axiom holds, that is we use a **Univalent Foundations (UF)** for mathematics in this work. The **univalence axiom** says that equality is equivalent to equivalence

$$(A = B) \simeq (A \simeq B) \quad (1.8)$$

meaning if two types are equivalent, then there is an equality between them. This makes (strong) bisimilarity imply equality, solving the problem described for $\sim_{\texttt{stream}}$. The univalent foundations we will be using is **Homotopy type theory (HoTT)** [17]. HoTT is an intensional dependent

type theory (built on MLTT) with the univalence axiom and higher inductive types . In HoTT the identity types form path spaces, so proofs of identity are not just `refl` which was the case in MLTT. Types are seen as spaces and we think of $a : A$ as a being a point in the space A. Functions are similarly regarded as continuous maps from one space to another [13]. As such we can construct higher inductive types, with point constructors as well as equality constructors. One of the problems with "plain" HoTT is that the univalence axiom is just postulated, meaning it is not constructive. To remedy this we will be working in a **cubical type theory (CTT)** [9], where the univalence axiom is not an axiom, but a statement that can be proven, meaning we get constructivity of univalence axiom and application thereof [12]. The reason for the name cubical type theory, is because composition is defined by square, that is given three sides of a square we get the last one, see Figure 1.1.

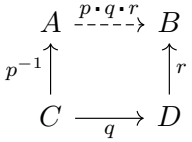

Figure 1.1: Composition square

Homotopy type theory is proof relevant, which means that there might be multiple proofs of one statement, and these proofs might not be interchangeable (equal). This can be explained by the use of H-level in HoTT, which describes how equality behaves. We start from H-level (-2) which are contractible types, meaning inhabited types where all element are equal. Then there is (-1)-types which are mere propositions or hProps, where all elements of the type are equal, but the type might not be inhabited. If the type is inhabited, then we say the proposition is true. The 0-types are the hSets, where all equalities between two elements x, y are equal, an example is the natural numbers. For 1-types (1-groupoids) we get equalities of equalities are equal, and so on for homotopy n-types. Any n-type is also a $(n+1)$-type, but with trivial equalities at the $(n+1)$th level. If we do not want to do proof relevant mathematics, we can do propositional truncation (denoted $\|\cdot\|$), which converts a n-type to a (-1)-type. This construction can be thought of as ignoring the difference in proofs and just look at whether a type is inhabited or not. However doing propositional truncation makes us loose some of the reasoning power of HoTT, so it should not be used unless there is a need for it. One useful tool in HoTT is **Higher order inductive types (HITs)**, where a type is defined by point constructors and equality constructors. An example is the propositional truncation we just described, with point constructor taking any element to it its truncated version, and equality constructor defined between all propositional truncated elements

$$\frac{x : A}{|\,x\,| : \|A\|} \quad (1.9) \qquad \frac{x, y : \|A\|}{\texttt{squash}\ x\ y : x \equiv y} \quad (1.10)$$

Another useful example is set truncated quotients. Given some free type, we can quotient it by a relation to get a type with equalities that respect the relation. A set truncated quotient of a type A with a relation $\mathtt{R}$ is given by the HIT

$$\frac{x : A}{[\,x\,] : A/\mathcal{R}} \quad (1.11) \qquad \frac{x, y : A/\mathcal{R} \quad r : x\ \mathcal{R}\ y}{\texttt{eq/}\ x\ y\ r : x \equiv y} \quad (1.12) \qquad \frac{}{\texttt{squash/} : \texttt{isSet}\ (A/\mathtt{R})} \quad (1.13)$$

When working with these two HITs, you might need to use the axiom of choice (AC), which is

defined in HoTT as follows

$$\prod_{(x:X)} \|\mathtt{Y}\ x\| \to \left\| \prod_{(x:X)} \mathtt{Y}\ x \right\| \tag{1.14}$$

where $\mathtt{Y}$ is a type family over set X [17, Section 3.8], or the weaker version axiom of countable choice (AC_ω)

$$\prod_{(n:\mathbb{N})} \|\mathtt{Y}\ n\| \to \left\| \prod_{(n:\mathbb{N})} \mathtt{Y}\ n \right\| \tag{1.15}$$

Using the axiom of choice is problematic in a constructive type theory, since it does not have a constructive interpretation. To maintain the computational aspects of HoTT and CTT, we try to avoid the axiom of choice [17, Introduction].

We have formalized most of Chapter 2*, the examples in Chapter 2.3 and some of Chapter 3 in the proof assistant / programming language Cubical Agda. A **proof assistant** verifies proof and helps with the interactive process of making a proof. **Cubical Agda**† [18] is an implementation of a cubical type theory (inspired by CCHM [10]) made by extending the proof assistant Agda. One of the main additions is the interval. The **interval** $\mathbb{I}$ can be thought of as the continuous unit interval $[0, 1]$. When working with the interval, we can only access the left and right endpoint `i0` and `i1` or some unspecified point in the middle i, keeping with the intuition of a continuous interval. Cubical Agda also generalizes transporting, given a type line $\mathtt{A} : \mathbb{I} \to \mathcal{U}$, and the endpoint $\mathtt{A}\ \mathtt{i0}$ you get a line from $\mathtt{A}\ \mathtt{i0}$ to $\mathtt{A}\ \mathtt{i1}$.

1.3 Notation

The notation and fonts used in the rest of the report is described in Table 1.1. We do a lot of casing on the natural numbers, so we will also introduce some notation, to make this more readable

Definition 1.3.1. Useful notation for casing on natural numbers.

$$\langle\!|\ x\ ,\ \mathtt{f}\ |\!\rangle = \lambda\, n, \begin{cases} x & n = 0 \\ \mathtt{f}\ m & n = m + 1 \end{cases} \tag{1.16}$$

We define the isomorphism $Iso\ A\ B$, by defining functions $\mathtt{fun} : A \to B$ and $\mathtt{inv} : B \to A$, and show there is a right identity $\mathtt{rinv} : \mathtt{fun} \circ \mathtt{inv} \equiv \mathtt{id}_B$ and a left identity $\mathtt{linv} : \mathtt{inv} \circ \mathtt{fun} \equiv \mathtt{id}_A$. This is enough to define an equivalence between A and B, so sometimes we will give the equality as an isomorphism. We will introduce further notation as we go.

*Accepted to the Cubical Agda Github, pull request: `https://github.com/agda/cubical/pull/245`

†Cubical Agda commit: e8d22a50079df1186b111c1f096eb29a1e3daa20, Agda Version: Agda-2.6.1

$\mathcal{U}_i$ or $\mathcal{U}$	Universe
$\mathtt{B} : A \to \mathcal{U}$	Type
$\mathtt{B} : A \to \mathcal{U}$	Type former
$\mathtt{T} : \Pi_{(a:A)} \mathtt{B}\ a$	Dependent product type
$\mathtt{T} : \sum_{(a:A)} \mathtt{B}\ a$	Dependent sum type
$x : A$	Term of a type
$\mathtt{c} : A$	Constant
$\mathtt{f} : A \to C$	Function
$\mathtt{f}$	Constructor
$\mathtt{f}$	Destructor
$p : A \equiv C$	Homogeneous path
$q : A \equiv_p C$	Heterogeneous path, denoted $\equiv_*$ if p is clear from context
$\mathtt{R} : A \to A \to \mathcal{U}$	Relation, elements denoted $x\ \mathtt{R}\ y$
$\mathbf{1}$	Unit type
$\mathbf{0}$	Empty / Zero / Bottom type
$\mathtt{P}$	Functor
π_1, π_2	First and second projection for dependent type $\sum_{(a:A)} \mathtt{B}\ a$

Table 1.1: Notation

Chapter 2

M-types

In this chapter we will introduce containers (aka. signatures) for the construction of M-types and the operations in and out on M-types (Theorem 2.1.9). We will show that M-out is a final coalgebra (Theorem 2.1.11) for the polynomial functor defined from the container. We then prove a coinduction principle for M-types (Theorem 2.2.2). This first half of the chapter draws heavy inspiration from the "Non-Wellfounded Trees in Homotopy Type Theory" [4]. We attempt to make this work more accessibly, as such our work is formalized in Cubical Agda and we go into more detail on how to construct the in and out functions. We conclude this chapter with some examples of M-types, namely the delay monad, infinite R-valued E-event trees and interaction tree (ITrees [19]).

2.1 Construction of M-types

We start by introducing containers and polynomial functors,

Definition 2.1.1. A Container (or signature) is a dependent pair $S = (A \, , \mathtt{B})$ for the types $A : \mathcal{U}$ and $\mathtt{B} : A \to \mathcal{U}$.

Definition 2.1.2. A polynomial functor $\mathtt{P}_S$ (or extension) for a container $S = (A \, , \mathtt{B})$ is defined, for types as

$$\begin{aligned} &\mathtt{P}_S : \mathcal{U} \to \mathcal{U} \\ &\mathtt{P}_S \, X = \sum_{(a:A)} \mathtt{B}(a) \to X \end{aligned} \tag{2.1}$$

and for a function $\mathtt{f} : X \to Y$ as

$$\begin{aligned} &\mathtt{P}_S \, \mathtt{f} : \mathtt{P}_S X \to \mathtt{P}_S Y \\ &\mathtt{P}_S \, \mathtt{f} \ (a, \mathtt{g}) = (a, \mathtt{f} \circ \mathtt{g}). \end{aligned} \tag{2.2}$$

If the container is clear from context we will just write P.

Example 1. The polynomial functor for streams over the type A is defined by the container $S = (A \, , \lambda \_ , \, \mathbf{1})$, we get

$$\mathtt{P} \, X = \sum_{(a:A)} \mathbf{1} \to X. \tag{2.3}$$

This expression can be simplified by the property $\mathbf{1} \to X \equiv X^{\mathbf{1}} \equiv X$. Furthermore $\mathbf{1}$ and X does not depend on A, so (2.3) is equivalent to

$$\mathtt{P}\, X = A \times X. \tag{2.4}$$

We define the $\mathtt{P}_S$-coalgebra for a polynomial functor $\mathtt{P}_S$, and the morphisms between coalgebras.

Definition 2.1.3. A $\mathtt{P}_S$-coalgebra is defined as

$$\mathtt{Coalg}_S = \sum_{(C:\mathcal{U})} C \to \mathtt{P}_S C. \tag{2.5}$$

where we denote a $\mathtt{P}_S$-coalgebra given by a type C and function γ as $C\text{-}\gamma$. Coalgebras morphisms are defined as

$$C\text{-}\gamma \Rightarrow D\text{-}\delta = \sum_{(\mathtt{f}:C\to D)} \delta \circ \mathtt{f} = \mathtt{Pf} \circ \gamma \tag{2.6}$$

Definition 2.1.4 (Final Coalgebra / M-type)**.** Given a container S, we define M-types as the type, making the coalgebra given by $\mathtt{M}_S$ and $\mathtt{out} : \mathtt{M}_S \to \mathtt{P}_S\, \mathtt{M}_S$ fulfill the property

$$\mathtt{Final}_S := \sum_{(X\text{-}\rho:\mathtt{Coalg}_S)} \prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}\ (C\text{-}\gamma \Rightarrow X\text{-}\rho). \tag{2.7}$$

That is $\prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}(C\text{-}\gamma \Rightarrow \mathtt{M}_S\text{-}\mathtt{out})$. We denote M-types as $\mathtt{M}_{(A\,,\mathtt{B})}$, $\mathtt{M}_S$ or just $\mathtt{M}$ when the container is clear from the context. When writing $\mathtt{isContr}\ A$, we mean A is of H-level (-2), that is $\sum_{(x:A)} \prod_{y:A} y \equiv x$ or equivalently $A \equiv \mathbf{1}$.

Continuing our example we now construct streams as an M-type.

Example 2. We define streams over the type A as the M-type over the container $(A\,, \lambda\_,\, \mathbf{1})$. If we apply the polynomial functor to the M-type, then we get $\mathtt{P}_{(A\,,\lambda\_,\mathbf{1})}\, \mathtt{M} = A \times \mathtt{M}_{(A\,,\lambda\_,\mathbf{1})}$. The consequences of this are illustrated in Figure 2.1. This gives us a semantic for the rules, we would

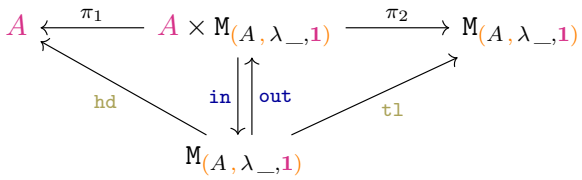

Figure 2.1: M-types of streams

expect for streams. If we let $\mathtt{cons} = \mathtt{in}$ where in is the inverse of $\mathtt{out}$ and $\mathtt{stream}\ A = \mathtt{M}_{(A\,,\lambda\_,\mathbf{1})}$, then we get

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{hd}\ s : A} \tag{2.8}$$

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{tl}\ s : \mathtt{stream}\ A} \tag{2.9}$$

$$\frac{A : \mathcal{U} \quad x : A \quad xs : \mathtt{stream}\ A}{\mathtt{cons}\ x\ xs : \mathtt{stream}\ A} \tag{2.10}$$

with $\mathtt{hd} = \pi_1 \circ \mathtt{out}$ and $\mathtt{tl} = \pi_2 \circ \mathtt{out}$.

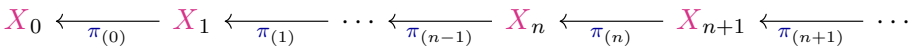

Figure 2.2: Chain of types / functions

We will define some general rules for chains, and then show that the limit of a chain defined by repeatedly applying `P` to the unit type $\mathbf{1}$, is a final coalgebra and therefore a definition for `M`-types.

Definition 2.1.5. We define a chain as a family of morphisms $\pi_{(n)} : X_{n+1} \to X_n$, over a family of types X_n. See Figure 2.2.

Definition 2.1.6. The limit of a chain is given as

$$\mathcal{L} = \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} \prod_{(n:\mathbb{N})} (\pi_{(n)}\ x_{n+1} \equiv x_n) \tag{2.11}$$

Lemma 2.1.7. *Given* $\ell : \prod_{(n:\mathbb{N})}(X_n \to X_{n+1})$ *and* $y : \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv \ell_n\ x_n$ *makes the chain collapse, giving us the equality* $\mathcal{L} \equiv X_0$.

Proof. We define the equality by the isomorphism

$$\mathtt{fun}_{\mathcal{L}collapse}\ (x, r) = x_0 \tag{2.12}$$

$$\mathtt{inv}_{\mathcal{L}collapse}\ x_0 = (\lambda\, n,\ \ell^{(n)}\ x_0)\ ,\ (\lambda\, n,\ \mathtt{refl}_{(\ell^{(n+1)}\ x_0)}) \tag{2.13}$$

$$\mathtt{rinv}_{\mathcal{L}collapse}\ x_0 = \mathtt{refl}_{x_0} \tag{2.14}$$

where $\ell^{(n)} = \ell_n \circ \ell_{n-1} \circ \cdots \circ \ell_1 \circ \ell_0$. To define $\mathtt{linv}_{\mathcal{L}collapse}\ (x, r)$, we first define a fiber (X, z, ℓ) over $\mathbb{N}$ given some $z : X_0$. Then any element of the type $\sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv \ell_n\ x_n$ is equal to a section over the fiber we defined. This means y is equal to a section. Since the sections are defined over $\mathbb{N}$, which is an initial algebra for the functor $\mathtt{G}Y = \mathbf{1} + Y$, we get that sections are contractible. This means $y \equiv \mathtt{inv}_{\mathcal{L}collapse}(\mathtt{fun}_{\mathcal{L}collapse}\ y)$, since both sides are equal to sections over $\mathbb{N}$. □

Lemma 2.1.8. *For all coalgebras* C-γ *defined over the container* S*, we get* $C \to \mathtt{M}_S \equiv \mathtt{Cone}_{C\text{-}\gamma}$*, where* $\mathtt{Cone} = \sum_{(\mathtt{f}:\prod_{(n:\mathbb{N})} C \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{f}_{n+1} \equiv \mathtt{f}_n$ *(illustrated in Figure 2.3).*

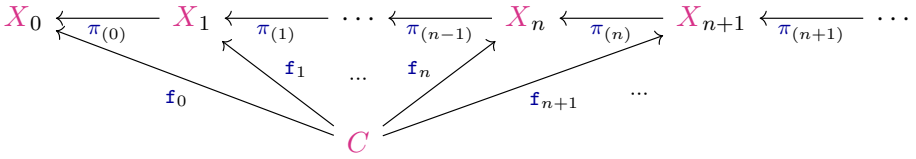

Figure 2.3: Cone

Proof. We show this equality between $C \to \mathtt{M}_S$ to $\mathtt{Cone}_{C\text{-}\gamma}$ by defining the isomorphism

$$\mathtt{fun}_{collapse}\ \mathtt{f} = (\lambda\, n\, z, \pi_1\ (\mathtt{f}\ z)\ n), (\lambda\, n\ i\ a,\ \pi_2\ (\mathtt{f}\ a)\ n\ i) \tag{2.15}$$

$$\mathtt{inv}_{collapse}\ (\mathtt{u}, \mathtt{q})\ z = (\lambda\, n,\ \mathtt{u}\ n\ z), (\lambda\, n\, i,\ \mathtt{q}\ n\ i\ z) \tag{2.16}$$

$$\mathtt{rinv}_{collapse}\ (\mathtt{u}, \mathtt{q}) = \mathtt{refl}_{(\mathtt{u},\mathtt{q})} \tag{2.17}$$

$$\mathtt{linv}_{collapse}\ \mathtt{f} = \mathtt{refl}_{\mathtt{f}} \tag{2.18}$$

□

Theorem 2.1.9. *Given a container* $(A, \mathtt{B})$*, we define a chain as the repeated application of* $\mathtt{P}$ *to the unit element* $X_n = \mathtt{P}^n\ \mathbf{1}$*, and* $\pi_{(n)} = \mathtt{P}^n\ !$ *where* $! : A \to \mathbf{1}$ *is the unique function into the unit type. Then there is an equality*

$$shift : \mathcal{L} \equiv \mathtt{P}\mathcal{L} \tag{2.19}$$

where $\mathcal{L}$ *is the limit of this chain.*

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^{\mathtt{P}} \equiv \mathtt{P}\,\mathcal{L} \tag{2.20}$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^{\mathtt{P}} \tag{2.21}$$

where $\mathcal{L}^{\mathtt{P}}$ is the limit of the shifted chain defined as ${X'}_n = X_{n+1}$ and ${\pi'}_{(n)} = \pi_{(n+1)}$. With these two lemmas we get

$$shift = \alpha \cdot \mathcal{L}unique. \tag{2.22}$$

We start by showing $\mathcal{L}unique$, which says the limit of a chain is unique. The equality is given by the isomorphism

$$\mathtt{fun}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = ⟫ \star\ ,\ \mathtt{a}\ ⟪, ⟫\ \mathtt{refl}_\star\ ,\ \mathtt{b}\ ⟪ \tag{2.23}$$

$$\mathtt{inv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{a} \circ \mathtt{succ}\ ,\ \mathtt{b} \circ \mathtt{succ} \tag{2.24}$$

$$\mathtt{rinv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{2.25}$$

$$\mathtt{linv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{2.26}$$

We then show α, which says the limit of the shifted chain is equal to $\mathtt{P}\,\mathcal{L}$. The proof is given by the equalities

$$\mathcal{L}^{\mathtt{P}} \equiv \sum_{(x:\prod_{(n:\mathbb{N})} X_{n+1})} \prod_{(n:\mathbb{N})} \pi_{(n+1)}\ x_{n+1} \equiv x_n \tag{2.27}$$

$$\equiv \sum_{(x:\prod_{(n:\mathbb{N})} \sum_{(a:A)} \mathtt{B}\,a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n+1)}\ x_{n+1} \equiv x_n \tag{2.28}$$

$$\equiv \sum_{((a,\mathtt{p}):\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\,a_n \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv_* \mathtt{u}_n \tag{2.29}$$

$$\equiv \sum_{(a:A)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\,a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv \mathtt{u}_n \tag{2.30}$$

$$\equiv \sum_{(a:A)} \mathtt{B}\,a \to \mathcal{L} \tag{2.31}$$

$$\equiv \mathtt{P}\mathcal{L} \tag{2.32}$$

The equality from (2.28) to (2.29) is rearranging terms, and adding a trivial point to the start of for the shifted chain. The equality from (2.29) and (2.30) is done with Lemma 2.1.7 to collapse $\sum_{(a:\mathbb{N}\to A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n$ to A. For the equality between (2.30) and (2.31), we use Lemma 2.1.8. The rest of the equalities are trivial, completing the proof. □

Definition 2.1.10. For the equality $shift$ we denote the functions back and forth as

$$\mathtt{out} = \mathtt{transport}\ shift \tag{2.33}$$
$$\mathtt{in} = \mathtt{transport}\ (shift^{-1}). \tag{2.34}$$

Theorem 2.1.11. *The* M*-type* $\mathtt{M}_S$ *is defined as the limit for a polynomial functor* $\mathtt{P}_S$*. This definition fulfills the requirement for the* M*-type given in Definition 2.1.4 namely* $\mathtt{Final}_S\ \mathcal{L}$*.*

Proof. By unfolding the definition, we need to show $\prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}\ (C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out})$, so we assume, that we are given some coalgebra $C\text{-}\gamma : \mathtt{Coalg}_S$ and want to show that morphisms into $\mathcal{L}\text{-}\mathtt{out}$ are contractible. We do this by showing $(C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out}) \equiv \mathbf{1}$. We get

$$C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out} \tag{2.35}$$
$$\equiv \sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{out} \circ \mathtt{f} \equiv \mathtt{Pf} \circ \gamma) \tag{2.36}$$
$$\equiv \sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{in} \circ \mathtt{out} \circ \mathtt{f} \equiv \mathtt{in} \circ \mathtt{Pf} \circ \gamma) \tag{2.37}$$
$$\equiv \sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{f} \equiv \mathtt{in} \circ \mathtt{Pf} \circ \gamma) \tag{2.38}$$

since $\mathtt{in}$ and $\mathtt{out}$ are each others inverse, furthermore $\mathtt{in}$ is part of an equality, which implies $(\mathtt{in} \circ \mathtt{a} \equiv \mathtt{in} \circ \mathtt{b}) \equiv (\mathtt{a} \equiv \mathtt{b})$, since functions that are part of an equality are embeddings. We let $\psi = \mathtt{in} \circ \mathtt{Pf} \circ \gamma$, which simplifies the expression to $\sum_{(\mathtt{f}:C\to\mathcal{L})}(\mathtt{f} \equiv \psi\ \mathtt{f})$. We then define $\mathtt{e}$ to be the function from right to left for the equality in Lemma 2.1.8, this gives us the equality

$$\sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{f} \equiv \psi\ \mathtt{f}) \quad \equiv \quad \sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{e}\ c \equiv \psi\ (\mathtt{e}\ c)) \tag{2.39}$$

By defining a function $\phi : \mathtt{Cone}_{C\text{-}\gamma} \to \mathtt{Cone}_{C\text{-}\gamma}$ as $\phi\ (\mathtt{u}, \mathtt{g}) = (\phi_0\ \mathtt{u}, \phi_1\ \mathtt{u}\ \mathtt{g})$ where

$$\phi_0\ \mathtt{u} = [\![\ (\lambda \_, \star)\ ,\ \mathtt{Pf} \circ \gamma \circ \mathtt{u}\ ]\!] \tag{2.40}$$
$$\phi_1\ \mathtt{u}\ \mathtt{g} = [\![\ \mathtt{funExt}\ \lambda \_, \mathtt{refl}_\star\ ,\ \mathtt{ap}\ (\mathtt{Pf} \circ \gamma) \circ \mathtt{g}\ ]\!] \tag{2.41}$$

we get the commuting square in Figure 2.4, which says $\psi\ (\mathtt{e}\ c) = \mathtt{e}\ (\phi\ c)$. We can then simplify to

$$\sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{e}\ \mathtt{c} \equiv \mathtt{e}\ (\phi\ \mathtt{c})) \tag{2.42}$$

$$\begin{array}{ccc} \mathtt{Cone}_{C\text{-}\gamma} & \xrightarrow{\mathtt{e}} & (C \to \mathcal{L}) \\ \downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle \psi} \\ \mathtt{Cone}_{C\text{-}\gamma} & \xrightarrow{\mathtt{e}} & (C \to \mathcal{L}) \end{array}$$

Figure 2.4: commuting square

We know that $\mathtt{e}$ is part of an equality (namely Lemma 2.1.8), so it is an embedding, we use this and unfolding the definition of ϕ, to get the equalities

$$\sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{c} \equiv \phi\ \mathtt{c}) \tag{2.43}$$

$$\equiv \sum_{((\mathtt{u},\mathtt{g}):\mathtt{Cone}_{C\text{-}\gamma})} ((\mathtt{u},\mathtt{g}) \equiv (\phi_0\ \mathtt{u}, \phi_1\ \mathtt{u}\ \mathtt{g})) \tag{2.44}$$

$$\equiv \sum_{((\mathtt{u},\mathtt{g}):\mathtt{Cone}_{C\text{-}\gamma})} \sum_{(p:\mathtt{u}\equiv\phi_0\ \mathtt{u})} \mathtt{g} \equiv_* \phi_1\ \mathtt{u}\ \mathtt{g} \tag{2.45}$$

We rearrange the terms and unfold the definition of $\mathtt{Cone}$ to get

$$\sum_{\left((\mathtt{u},p):\sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} C\to X_n)} \mathtt{u}\equiv\phi_0\,\mathtt{u}\right)} \sum_{(\mathtt{g}:\prod_{(n:\mathbb{N})} \pi_{(n)}\circ\mathtt{u}_{n+1}\equiv\mathtt{u}_n)} \mathtt{g} \equiv_* \phi_1\ \mathtt{u}\ \mathtt{g} \tag{2.46}$$

We define the following two equalities as instances of Lemma 2.1.7

$$\mathbf{1} \equiv \left(\sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} C\to X_n)} \mathtt{u} \equiv \phi_0\,\mathtt{u} \right) \tag{2.47}$$

$$\mathbf{1} \equiv_* \left(\sum_{(\mathtt{g}:\prod_{(n:\mathbb{N})} \pi_{(n)}\circ\mathtt{u}_{n+1}\equiv\mathtt{u}_n)} \mathtt{g} \equiv_* \phi_1\ \mathtt{u}\ \mathtt{g} \right) \tag{2.48}$$

Using these two equalities the proof simplifies to $\sum_{(\star:\mathbf{1})} \mathbf{1} \equiv \mathbf{1}$, which is trivial. □

2.2 Coinduction Principle for M-types

We can now construct a coinduction principle for $\mathtt{M}$-types, which gives equality between (strongly) bisimilar elements.

Definition 2.2.1. A relation $\mathcal{R} : C \to C \to \mathcal{U}$ for a coalgebra $C\text{-}\gamma : \mathtt{Coalg}_S$, is a (strong) bisimulation relation if the type $\overline{\mathcal{R}} = \sum_{(a:C)} \sum_{(b:C)} a\ \mathcal{R}\ b$ and the function $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \to \mathtt{P}_S\,\overline{\mathcal{R}}$ forms a $\mathtt{P}_S$-coalgebra $\overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, making the diagram in Figure 2.5 commute ($\Longrightarrow$ represents $\mathtt{P}_S$-coalgebra morphisms). That is $\gamma \circ \pi_1^{\overline{\mathcal{R}}} \equiv \mathtt{P}_S\,\pi_1^{\overline{\mathcal{R}}} \circ \alpha_{\mathbb{R}}$, and similarly for $\pi_2^{\overline{\mathcal{R}}}$.

$$C\text{-}\gamma \xleftarrow{\pi_1^{\overline{\mathcal{R}}}} \overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} \xrightarrow{\pi_2^{\overline{\mathcal{R}}}} C\text{-}\gamma$$

Figure 2.5: Bisimulation for a coalgebra

Theorem 2.2.2 (Coinduction principle)**.** *Given a relation $\mathcal{R}$, that is a bisimulation for an $\mathtt{M}$-type, then if two elements are (strongly) bisimilar $x\ \mathcal{R}\ y$, then they are equal $x \equiv y$.*

Proof. Given a relation $\mathcal{R}$, that is part of a bisimulation over a final $\mathtt{P}$-coalgebra $\mathtt{M\text{-}out} : \mathtt{Coalg}_S$ we get the diagram in Figure 2.6. By the finality of $\mathtt{M\text{-}out}$, we get a function $!$ from $\mathtt{M}$ to $\overline{R}$, which

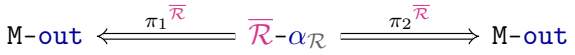

Figure 2.6: Bisimulation principle for final coalgebra

is unique meaning $\pi_1^{\overline{\mathcal{R}}} \equiv\ !\equiv \pi_2^{\overline{\mathcal{R}}}$. Now if we are given $r : x\ \mathcal{R}\ y$, we can construct the equality

$$x \equiv \pi_1^{\overline{\mathcal{R}}}(x, y, r) \equiv \pi_2^{\overline{\mathcal{R}}}(x, y, r) \equiv y. \tag{2.49}$$

Giving us the coinduction principle for M-types. □

We therefore get equalities for M-types, when we can define a bisimulation. For example the bisimilarity given for streams in the introduction (1.7), will give us an equality principle for streams by this coinduction principle.

2.3 Examples of M-types

In this section we show some examples of coinductive types constructed as M-types. We show how to define the constructors/destructors of these types, and give some intuition for the construction.

2.3.1 Delay Monad

The delay monad is used to model delayed computations, either returning immediately or delayed some (possibly infinite) number of steps.

Theorem 2.3.1. *The delay monad can be defined as the M-type.*

Proof. We want to construct a container, that will give us two constructors, one that returns immediately (now) and one that delays the computation (later). We do this with the container

$$S = \left(R + \mathbf{1}\,,\ \begin{cases} \mathbf{0} & \mathtt{inl}\ r \\ \mathbf{1} & \mathtt{inr}\ \star \end{cases} \right) \tag{2.50}$$

which gives us the polynomial functor

$$\mathtt{P}_S\, X = \sum_{(x : R + \mathbf{1})} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ \mathbf{1} & x = \mathtt{inr}\ \star \end{cases} \to X, \tag{2.51}$$

This definition is not dependent, so we can simplify it

$$\mathtt{P}_S\, X = R \times (\mathbf{0} \to X) + X. \tag{2.52}$$

from the property that $(\mathbf{0} \to X) \equiv \mathbf{1}$, we can simplify the definition further to

$$\mathtt{P}_S\, X = R + X \tag{2.53}$$

We get diagram in Figure 2.7, by applying $\mathtt{P}_S$ to $\mathtt{M}_S$. We define the constructors now and later using the in function for M-types together with the injections inl and inr, making that the diagram in Figure 2.7 commute. □

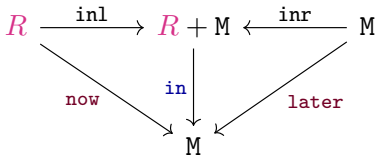

Figure 2.7: Delay monad

2.3.2 Infinite R-valued E-event Trees

We will look at a type of tree where the values of leaves are given by the type R and branching is indexed by events $\mathtt{E}\,A$ given some type A.

Theorem 2.3.2. *We can define R-valued* E*-event trees as an* M*-type*

Proof. We want a container, with a leaf constructor (`leaf`) returning immediately with a value, and a node constructor (`node`) that branches on events. We defined the container

$$S = \left(R + \sum_{(A:\mathcal{U})} \mathtt{E}\,A\,,\ \begin{cases} \mathbf{0} & \mathtt{inl}\ r \\ A & \mathtt{inr}\ (A, e) \end{cases} \right). \tag{2.54}$$

for which we get the polynomial functor

$$\mathtt{P}_S\,X = \sum_{(x:R+\sum_{(A:\mathcal{U})}\mathtt{E}\,A)} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ A & x = \mathtt{inr}\ (A, e) \end{cases} \to X, \tag{2.55}$$

We split the two case into a sum, since they are not dependent

$$\mathtt{P}_S\,X = (R \times (\mathbf{0} \to X)) + (\sum_{(A:\mathcal{U})} \mathtt{E}\,A \times (A \to X)), \tag{2.56}$$

We simplify further using $(\mathbf{0} \to X) \equiv \mathbf{1}$, to get the definition

$$\mathtt{P}_S\,X = R + \sum_{(A:\mathcal{U})} \mathtt{E}\,A \times (A \to X). \tag{2.57}$$

By applying the polynomial functor to the M-type, we get the diagram in Figure 2.8. We can define

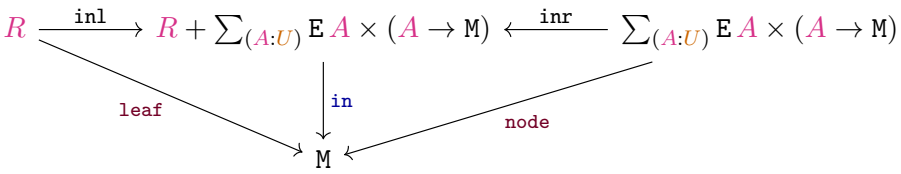

Figure 2.8: Tree Constructors

`leaf` and `node` using the `in` function together with injections making the diagram commute. □

We can see that `now` and `leaf` have the same definitional structure in the container and diagramatic structure.

2.3.3 ITrees

Interaction trees (ITrees) [19] are used to model effectful behavior, where computations can interact with an external environment by events. ITrees are defined by the following constructors

$$\frac{r : R}{\texttt{Ret}\ r : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Ret}} \tag{2.58}$$

$$\frac{A : \mathcal{U} \quad a : \texttt{E}\ A \quad f : A \to \texttt{itree}\ \texttt{E}\ R}{\texttt{Vis}\ a\ f : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Vis}}. \tag{2.59}$$

$$\frac{t : \texttt{itree}\ \texttt{E}\ R}{\texttt{Tau}\ t : \texttt{itree}\ \texttt{E}\ R}\ \texttt{E}_{\texttt{Tau}}. \tag{2.60}$$

where R is the type for returned values, while $\texttt{E}$ is a dependent type for events representing external interactions. ITrees without the $\texttt{Vis}$ constructor is the delay monad. If we remove the $\texttt{Tau}$ constructor, then we get infinite R-valued $\texttt{E}$-event trees. So it should just be a matter of combining the definitions for the previous two examples to construct ITrees.

Theorem 2.3.3. *We can define the type of ITrees as an $\texttt{M}$-type*

Proof. We combine the constructions of the delay monad and R-valued $\texttt{E}$-event Trees, into the container

$$S = \left(R + \mathbf{1} + \sum_{(A:\mathcal{U})} (\texttt{E}\ A)\,,\ \begin{cases} \mathbf{0} & \texttt{inl}\ r \\ \mathbf{1} & \texttt{inl}\ (\texttt{inl}\ \star) \\ A & \texttt{inr}\ (\texttt{inr}\ (A, e)) \end{cases} \right). \tag{2.61}$$

for which the reduced polynomial functor is

$$\texttt{P}_S\ X = R + X + \sum_{(A:\mathcal{U})} (\texttt{E}\ A \times (A \to X)) \tag{2.62}$$

Applying this polynomial functor to the $\texttt{M}$-type, for the container, we get the diagram in Figure 2.9, from which the constructors of the ITrees type can be defined using $\texttt{in}$ and injections. □

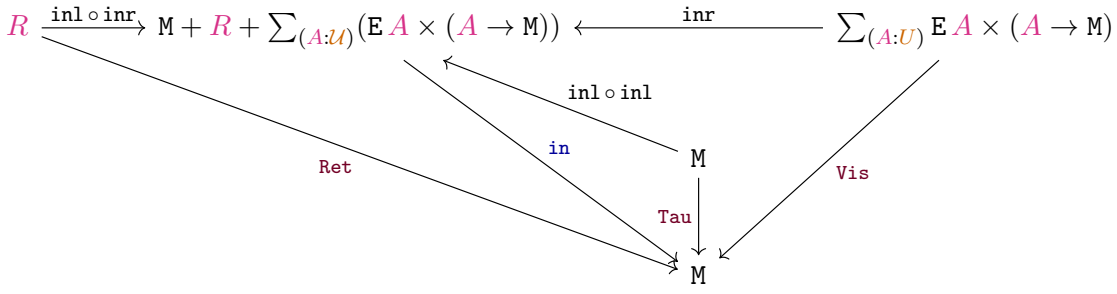

Figure 2.9: ITree constructors

2.4 General rules for constructing M-types

We want to create a rule set for how to define coinductive types as M-types, we take inspiration from the rules for containers [3][1]. We would like to be able to define a type from a given set of constructor / destructors. If we for example is given the rule

$$\frac{a : A}{\texttt{ret}\ a : T} \tag{2.63}$$

we get that it corresponds to the M-type for the container $(A\,, \lambda\_,\ \mathbf{0})$, while if we have something that produces an element of it self as

$$\frac{a : T}{\texttt{tl}\ a : T} \tag{2.64}$$

the container is $(\mathbf{1}\,, \lambda\_,\ \mathbf{1})$. If we want a type with both these rules, then we just take the disjoint sum of the two containers

$$\left(A + \mathbf{1}\,, \begin{cases} \mathbf{0} & \texttt{inl}\ a \\ \mathbf{1} & \texttt{inr}\ \star \end{cases}\right) \tag{2.65}$$

which is the container used to define the delay monad. We might want a type for a rule, where the type appears positively, for example

$$\frac{a : A \to T}{\texttt{node}\ a : T} \tag{2.66}$$

has container $(\mathbf{1}\,, A)$. We can see that the arguments to such a constructor must appear in the last part of the container, whereas if we have simple functions where the type we want to construct does not appear, then it suffices to use the first part of the container. A type for the destructor

$$\frac{a : T}{\texttt{hd}\ a : A} \tag{2.67}$$

can be constructed by the container $(A\,, \lambda\_,\ \mathbf{0})$, but this container is the same as the one for `ret`, so do we get both? The answer is yes, if we look at it isolated. Whether we get a constructor or destructor depends on how we combine the containers with the containers for other constructors / destructors. In general adding containers $(A\,, \mathtt{B})$ and $(C\,, \mathtt{D})$ for two constructors together is done by

$$\left(A + C\,, \begin{cases} \mathtt{B}\,a & \texttt{inl}\ a \\ \mathtt{D}\,c & \texttt{inr}\ c \end{cases}\right) \tag{2.68}$$

whereas adding containers for two destructors is done by

$$(A \times C\,, \lambda\_,\ \lambda\,(a, c),\ \mathtt{B}\,a + \mathtt{D}\,c) \tag{2.69}$$

However combining destructors and constructors is not as simple, which is also the case if we use records or other constructions to define such types. We argue that any type T that can be defined using a (coinductive) record (except for higher inductive types), will also be definable as an M-type, with the following construction. Given a record, which is a list of fields $f_1 : F_1, f_2 : F_2, \ldots, f_n : F_n$, we can construct the M-types by the container

$$(F_1 \times F_2 \times \cdots \times F_n\,, \lambda\_,\ \mathbf{0}) \tag{2.70}$$

where each destructor $\mathtt{d_n} : T \to F_n$ for the field f_n will be defined as $\mathtt{d_n}\ t = \pi_n\ (\texttt{out}\ t)$. However fields in a coinductive record may depend on previous defined fields. That is we can have the list

of fields $f_1 : F_1, f_2 : \mathtt{F}_2, \cdots, f_n : \mathtt{F}_n$, where each field depends on all the previous once. We define this as the $\mathtt{M}$-type for the container

$$\left(\sum_{(f_1 : F_1)} \sum_{(f_2 : \mathtt{F}_2)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \, , \lambda \_, \, \mathbf{0} \right) \tag{2.71}$$

however, if any of the fields are non dependent, then they can be added as a product ($\times$) instead of a dependent sum (Σ). Fields can also coinductively define an element of the record T itself, however other fields may not refer to such a field, since it will break the strictness requirements of the record / coinductive type. As an example of a coinductive field let f_1 be a type and $\mathtt{f}_2$ be the function with type $\mathtt{F}_2 = f_1 \to (f_1 \to A) \to \mathtt{M}$, which can be curried to $f_1 \times (f_1 \to A) \to \mathtt{M}$. This can be define by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \left(\mathbf{1} \times \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \right) \, , \lambda \, (f_1, \star, f_3, \dots), \mathtt{F}_2 \right) \tag{2.72}$$

where $\mathtt{F}_2$ has been moved to the last part of the container. We can even leave out the "$\mathbf{1}\times$" and $\star$ case from the container, since it does not add any information. Another case is that the type of a field is dependent $\mathtt{F}_2 = (x : f_1) \to \mathtt{B}\, x \to \mathtt{M}$, but again by currying we get $\mathtt{F}_2 : \sum_{(x : f_1)} \mathtt{B}\, x \to \mathtt{M}$ which can be defined by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_\mathtt{n} \, , \lambda \, (f_1, f_3, \dots), \sum_{(x : f_1)} (\mathtt{B}\, x) \right) \tag{2.73}$$

so we would expect that a type defined as a (coinductive) record is equal to the version defined as a $\mathtt{M}$-type. However this has not been shown formally. We have explored examples of these equalities, and they seem doable on paper, but issues with the termination checker for Cubical Agda has hindered the formalization. A solution might be to use sized-types or switch to a guarded cubical type theory.

Chapter 3

QM-types

In this chapter we will introduce rules for set truncated quotients and show some constructions of quotiented M-types, which we call QM-types. If we define QM-types with set truncated quotients, then we need the axiom of countable choice, to raise the constructors to the quotient. In an effort to avoid the axiom of countable choice, we use quotient inductive-inductive type (QIIT) [5] to define the quotiented M-types. We show examples of equalities between these two construction assuming axiom of countable choice. The examples are multisets and the partiality monad, defined as quotients with weak bisimilarity relations. We conclude the chapter by discussion, how we should define QM-types.

3.1 Eliminators and Recursors for the Set Truncated Quotient

We start by defining the recursor and some eliminators for the set truncated quotient, which will be used in the upcoming sections.

Definition 3.1.1 (Recursor for quotient)**.** For all elements $x, y : A$, functions $\mathtt{f} : A \to B$ and relations $\mathtt{g} : x\ \mathtt{R}\ y \to \mathtt{f}\ x \equiv \mathtt{f}\ y$, then if B is a set, we get a function from $A/\mathtt{R}$ to B, defined by case as

$$\begin{aligned}
&\mathtt{rec}\ [\,a\,] = \mathtt{f}\ a \\
&\mathtt{rec}\ (\mathtt{eq/}\ _\ _\ r\ i) = \mathtt{g}\ r\ i \\
&\mathtt{rec}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) = B_{set}\ (\mathtt{rec}\ a)\ (\mathtt{rec}\ b)\ (\mathtt{ap}\ \mathtt{rec}\ p)\ (\mathtt{ap}\ \mathtt{rec}\ q)\ i\ j
\end{aligned} \tag{3.1}$$

Definition 3.1.2 (Propositional eliminator for quotient)**.** Given a dependent type $\mathtt{P} : A/\mathtt{R} \to \mathcal{U}$, that is a proposition at all points $\mathtt{P}_{\mathtt{prop}} : \prod_{(x:A/\mathtt{R})} \mathtt{isProp}\ (\mathtt{P}\ x)$, and given a proof for the base case $\mathtt{f} : \prod_{(a:A)} \mathtt{P}\,[\,a\,]$, we get a function $\mathtt{elimProp} : \prod_{(x:A/\mathtt{R})} \mathtt{P}\ x$, defined as

$$\begin{aligned}
&\mathtt{elimProp}\ [\,a\,] = \mathtt{f}\ a \\
&\mathtt{elimProp}\ (\mathtt{eq/}\ a\ b\ i) = \mathtt{P}_{\mathtt{prop}}\ (\mathtt{elimProp}\ a)\ (\mathtt{elimProp}\ b)\ (\mathtt{eq/}\ a\ b)\ i \\
&\mathtt{elimProp}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) = \\
&\quad \mathtt{isSet{\to}isSetDep}\ (\mathtt{isProp{\to}isSet} \circ \mathtt{P}_{\mathtt{prop}}) \\
&\quad\ (\mathtt{elimProp}\ a)\ (\mathtt{elimProp}\ b)\ (\mathtt{ap}\ \mathtt{elimProp}\ p)\ (\mathtt{ap}\ \mathtt{elimProp}\ q) \\
&\quad\ (\mathtt{squash/}\ a\ b\ p\ q)\ i\ j
\end{aligned} \tag{3.2}$$

where $\mathtt{isSet{\to}isSetDep}$ takes a function $\prod_{a:A} \mathtt{isSet}\ (\mathtt{B}\,a)$ to the dependent version $\mathtt{isSetDep}\ A\ \mathtt{B}$. This eliminator allows us to show a proposition for a quotient, by supplying just the base case $\mathtt{P}\,[\,a\,]$.

Definition 3.1.3 (Eliminator for quotients)**.** Given a dependent type $\mathtt{P} : A/\mathtt{R} \to \mathcal{U}$ which is a set at all points $\mathtt{P_{set}} : \prod_{(x:A/\mathtt{R})} \mathtt{isSet}\ (\mathtt{P}\,x)$, a proof for the base case $\mathtt{f} : \prod_{(a:A)} \mathtt{P}\,[\,a\,]$ and a proof of the equality case $\mathtt{f_{eq}} : \prod_{(a,b:A)} \prod_{(r:a\ \mathtt{R}\ b)} \mathtt{f}\ a \equiv_* \mathtt{f}\ b$ for the equality constructor $\mathtt{eq/}\ a\ b\ r$, we get a function $\mathtt{elim} : \prod_{(x:A/\mathtt{R})} \mathtt{P}\,x$ as follows

$$\begin{aligned}
&\mathtt{elim}\,[\,a\,] = \mathtt{f}\ a\\
&\mathtt{elim}\ (\mathtt{eq/}\ a\ b\ r\ i) = \mathtt{f_{eq}}\ a\ b\ r\ i\\
&\mathtt{elim}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) =\\
&\quad \mathtt{isSet{\to}isSetDep}\ \mathtt{P_{set}}\ (\mathtt{elim}\ a)\ (\mathtt{elim}\ b)\ (\mathtt{ap}\ \mathtt{elim}\ p)\ (\mathtt{ap}\ \mathtt{elim}\ q)\ (\mathtt{squash/}\ a\ b\ p\ q)\ i\ j
\end{aligned} \tag{3.3}$$

which is the eliminator for the set truncated quotient.

We can now construct some more interesting data types, by set quotienting M-types, which we call QM-types, some examples follow in the following section.

3.2 QM-types and Quotient inductive-inductive types (QIITs)

In this section we will show some examples of quotiented M-type defined using set truncated quotients. We will also construct the types as a quotient inductive-inductive types instead, and show these two construction gives equal types, assuming the axiom of countable choice. A quotient inductive-inductive type (QIIT) [5] is a type that is defined at the same time as a relation over that type and then set truncated. We believe, but do not show, that this equality between the two ways of constructing QM-types is general.

3.2.1 Multiset

In this subsection we define infinite trees, where the order of subtrees does not matter also known as multisets. We draw inspiration from similar equality given in [5], [7] and [11] for the dual case of finite multisets. We construct the type of multisets as a QM-type and as a QIIT and show these are equal (Theorem 3.2.9). We start by defining the tree type we want to quotient to get the multiset.

Definition 3.2.1. We define A-valued $\mathbb{N}$-branching trees $\mathtt{T}\,A$ as the M-type defined by the container

$$\left(A + \mathbf{1}, \begin{cases} \mathbf{0} & \mathtt{inl}\ a \\ \mathbb{N} & \mathtt{inr}\ \star \end{cases}\right) \tag{3.4}$$

For which we get the constructors

$$\frac{a : A}{\mathtt{leaf}\ a : \mathtt{T}\,A} \tag{3.5}$$

$$\frac{\mathtt{f} : \mathbb{N} \to \mathtt{T}\,A}{\mathtt{node}\ \mathtt{f} : \mathtt{T}\,A} \tag{3.6}$$

Trees where the permutation of subtrees does not matter is multisets. We define equality between permutations of subtrees by the following rule

$$\frac{\mathtt{f} : \mathbb{N} \to \mathtt{T}\,A \quad \mathtt{g} : \mathbb{N} \to \mathbb{N} \quad \mathtt{isIso}\ \mathtt{g}}{\mathtt{node}\ \mathtt{f} \equiv \mathtt{node}\ (\mathtt{f} \circ \mathtt{g})}\ \mathtt{perm} \tag{3.7}$$

One way to define this type is as the QM-type given by quotienting $\mathtt{T}\,A$ by the relation $\sim_{\mathtt{T}}$ defined by the constructors

$$\frac{x \equiv y}{\texttt{leaf}\ x \sim_{\texttt{T}} \texttt{leaf}\ y}\ \sim_{\texttt{leaf}} \quad (3.8) \qquad \frac{\prod_{(n:\mathbb{N})} \texttt{f}\ n \sim_{\texttt{T}} \texttt{g}\ n}{\texttt{node}\ \texttt{f} \sim_{\texttt{T}} \texttt{node}\ \texttt{g}}\ \sim_{\texttt{node}} \quad (3.9)$$

$$\frac{\texttt{f} : \mathbb{N} \to \texttt{T}\ A \quad \texttt{g} : \mathbb{N} \to \mathbb{N} \quad \texttt{isIso}\ \texttt{g}}{\texttt{node}\ \texttt{f} \sim_{\texttt{T}} \texttt{node}\ (\texttt{f} \circ \texttt{g})}\ \sim_{\texttt{perm}} \tag{3.10}$$

The problem with this construction is that we cannot lift the definition of node to the quotiented type without the use of the axiom of countable choice. To solve this problem, we define multisets as a QIIT denoted MS A, with the constructors leaf, node, perm and set truncation constructor MS-isSet. We show these two ways of constructing a type for multisets are equal assuming the axiom of countable choice.

Definition 3.2.2. There is a function from T A to MS A

$$\begin{aligned} \texttt{T→MS}\ (\texttt{leaf}_{\texttt{T}}\ x) &= \texttt{leaf}_{\texttt{MS}}\ x \\ \texttt{T→MS}\ (\texttt{node}_{\texttt{T}}\ \texttt{f}) &= \texttt{node}_{\texttt{MS}}\ (\texttt{T→MS} \circ \texttt{f}) \end{aligned} \tag{3.11}$$

This function takes weakly bisimilar objects to equal once.

Lemma 3.2.3. *If x, y : T A are weakly bisimilar $p : x \sim_{\texttt{T}} y$ then* T→MS $x \equiv$ T→MS y.

Proof. We do the proof by casing on the constructor for the weak bisimilarity.

$$\begin{aligned} \texttt{T→MS-}\sim\texttt{→}\equiv\ (\sim_{\texttt{leaf}}\ p) &= \texttt{ap}\ \texttt{leaf}_{\texttt{MS}}\ p \\ \texttt{T→MS-}\sim\texttt{→}\equiv\ (\sim_{\texttt{node}}\ \texttt{k}) &= \texttt{ap}\ \texttt{node}_{\texttt{MS}}\ (\texttt{funExt}\ (\texttt{T→MS-}\sim\texttt{→}\equiv \circ\ \texttt{k})) \\ \texttt{T→MS-}\sim\texttt{→}\equiv\ (\sim_{\texttt{perm}}\ \texttt{f}\ \texttt{g}\ e) &= \texttt{perm}\ (\texttt{T→MS} \circ \texttt{f})\ \texttt{g}\ e \end{aligned} \tag{3.12}$$

□

With this lemma, we can lift the function T→MS to the quotient.

Definition 3.2.4. There is a function T/∼→MS from T A/$\sim_{\texttt{T}}$ to MS A defined using the recursor for quotients, given in Definition 3.1.1, with f = T→MS and g = T→MS-∼→≡ and using that MS is a set by MS-isSet

Lemma 3.2.5. *Given an equality* T→MS $x \equiv$ T→MS y *then* $x \sim_{\texttt{T}} y$.

Proof. If x and y are leaves with values a and b then $a \equiv b$ by the injectivity of the constructor $\texttt{leaf}_{\texttt{MS}}$, giving a bisimilarity by $\sim_{\texttt{leaf}}$. If x and y are nodes defined by functions f and g, then by the injectivity of the constructor $\texttt{node}_{\texttt{MS}}$, we get $\prod_{(n:\mathbb{N})} \texttt{f}\ n \equiv \texttt{g}\ n$ and by induction we get $\prod_{(n:\mathbb{N})} \texttt{f}\ n \sim_{\texttt{T}} \texttt{g}\ n$, making us able to use $\sim_{\texttt{node}}$ to construct the bisimilarity. We do not get any other equalities, since $\texttt{leaf}_{\texttt{MS}}$ and $\texttt{node}_{\texttt{MS}}$ are disjoint. □

Lemma 3.2.6. *The function* T/∼→MS *is injective, meaning* T/∼→MS $x \equiv$ T/∼→MS y *implies* $x \equiv y$.

Proof. We show injectivity by doing propositional elimination defined in Definition 3.1.2, with

$$\mathtt{P} = (\lambda\, x,\ \mathtt{T/{\sim}{\to}MS}\ x \equiv \mathtt{T/{\sim}{\to}MS}\ y \to x \equiv y) \tag{3.13}$$

which is a proposition for all x by $\mathtt{P}_{\mathtt{prop}} = \lambda\, x,\ \mathtt{isPropΠ}\ (\lambda\_, \mathtt{squash/}\ x\ y)$. We do the propositional elimination twice, first for x and then for y, where we define $\mathtt{P}$ and $\mathtt{P}_{\mathtt{prop}}$ similarly, but with $x = [\,a\,]$ since it has already been propositionally eliminated.

$$\mathtt{elimProp}\ (\lambda\, a,\ \mathtt{elimProp}(\lambda\, b,\ \mathtt{eq/}\ a\ b \circ \mathtt{T{\to}MS\text{-}{\equiv}{\to}{\sim}}\ a\ b)\ y)\ x \tag{3.14}$$

where $\mathtt{T{\to}MS\text{-}{\equiv}{\to}{\sim}}$ is Lemma 3.2.5. □

Definition 3.2.7. We define the propositional eliminator for $\mathtt{MS}$. Given a dependent type $\mathtt{P} : \mathtt{MS}\ X \to \mathcal{U}$ which is a proposition at all points $\mathtt{P}_{\mathtt{prop}} : \prod_{(x:\mathtt{MS}\ X)} \mathtt{isProp}\ (\mathtt{P}\ x)$, and given leaf case $\mathtt{P}_{\mathtt{leaf}} : \prod_{x:X} \mathtt{P}\ (\mathtt{leaf}\ x)$ and node case $\mathtt{P}_{\mathtt{node}}\ \prod_{(\mathtt{f}:\mathbb{N}\to\mathtt{MS}\ X)} \prod_{(\mathtt{f}':\prod_{(n:\mathbb{N})} \mathtt{P}\ (\mathtt{f}\ n))} \mathtt{P}\ (\mathtt{node\ f})$, we get a function $\mathtt{elimProp}_{\mathtt{MS}} : \prod_{(t:\mathtt{MS}\ X)} \mathtt{P}\ t$, defined as

$$\begin{aligned}
&\mathtt{elimProp}\ (\mathtt{leaf}\ x) = \mathtt{P}_{\mathtt{leaf}}\ x\\
&\mathtt{elimProp}\ (\mathtt{node}\ f) = \mathtt{P}_{\mathtt{node}}\ (\mathtt{elimProp} \circ f)\\
&\mathtt{elimProp}\ (\mathtt{perm\ f\ g}\ e\ i) =\\
&\quad \mathtt{isProp{\to}isDepProp}\ \mathtt{P}_{\mathtt{prop}}\ (\mathtt{elimProp}\ (\mathtt{node\ f}))\ (\mathtt{elimProp}\ (\mathtt{node}\ (\mathtt{f} \circ \mathtt{g})))\\
&\qquad (\mathtt{perm\ f\ g}\ e)\ i\\
&\mathtt{elimProp}\ (\mathtt{MS\text{-}isSet}\ a\ b\ p\ q\ i\ j) =\\
&\quad \mathtt{isSet{\to}isDepSet}\ (\mathtt{isProp{\to}isSet} \circ \mathtt{P}_{\mathtt{prop}})\ (\mathtt{elimProp}\ a)\ (\mathtt{elimProp}\ b)\\
&\qquad (\mathtt{ap\ elimProp}\ p)\ (\mathtt{ap\ elimProp}\ q)\ (\mathtt{MS\text{-}isSet}\ a\ b\ p\ q)\ i\ j
\end{aligned} \tag{3.15}$$

Lemma 3.2.8. *The function* $\mathtt{T/{\sim}{\to}MS}$ *is surjective, assuming the axiom of countable choice. Being surjective means for all* $b : \mathtt{MS}\ X$ *we have* $\|\Sigma_{(x:\mathtt{T}\ X/{\sim_\mathtt{T}})}\ \mathtt{T/{\sim}{\to}MS}\ x \equiv b\|$.

Proof. We do propositional elimination of $\mathtt{MS}\ X$, with $\mathtt{P}\ y = \|\Sigma_{(x:\mathtt{T}\ X/{\sim_\mathtt{T}})}\ \mathtt{T/{\sim}{\to}MS}\ x \equiv y\|$, which is a proposition by truncation. For the leaf case, we have the simple equality $\mathtt{T/{\sim}{\to}MS}\ [\mathtt{leaf}_\mathtt{T}\ x/{\sim_\mathtt{T}}] \equiv \mathtt{leaf}_{\mathtt{MS}}\ x$. For the node case with $\mathtt{node\ f}$, we get the following function by induction

$$\mathtt{f}' : \prod_{(n:\mathbb{N})} \left\| \sum_{(y:\mathtt{T}\ X/{\sim_\mathtt{T}})} \mathtt{T/{\sim}{\to}MS}\ y \equiv \mathtt{f}\ n \right\| \tag{3.16}$$

Using the axiom of countable choice, we get

$$\left\| \prod_{(n:\mathbb{N})} \sum_{(y:\mathtt{T}\ X/{\sim_\mathtt{T}})} \mathtt{T/{\sim}{\to}MS}\ y \equiv \mathtt{f}\ n \right\| \tag{3.17}$$

By the surjectivity of $[\,\cdot\,]$ we get

$$\left\| \prod_{(n:\mathbb{N})} \sum_{(y:\mathtt{T}\ X)} \mathtt{T{\to}MS}\ y \equiv \mathtt{f}\ n \right\| \tag{3.18}$$

By swapping Π and Σ and functional extensionality we get

$$\left\| \sum_{(g:\mathbb{N}\to \mathtt{T}\ X)} \mathtt{T}{\to}\mathtt{MS} \circ \mathtt{g} \equiv \mathtt{f} \right\| \tag{3.19}$$

which gives us the equalities

$$\begin{aligned} \mathtt{T}/{\sim}{\to}\mathtt{MS}\ [\mathtt{node}\ \mathtt{g}] &\equiv \mathtt{T}{\to}\mathtt{MS}\ (\mathtt{node}\ \mathtt{g}) && (3.20)\\ &\equiv \mathtt{node}\ (\mathtt{T}{\to}\mathtt{MS} \circ \mathtt{g}) && (3.21)\\ &\equiv \mathtt{node}\ \mathtt{f} && (3.22) \end{aligned}$$

completing the case for the node constructor and thereby the proof. □

Theorem 3.2.9. *There is an equality between the types* $\mathtt{T}\ A/{\sim_{\mathtt{T}}}$ *and* $\mathtt{MS}\ A$*, assuming the axiom of countable choice.*

Proof. Since the function $\mathtt{T}/{\sim}{\to}\mathtt{MS}$ is injective and surjective, it becomes an equality. □

3.2.2 Partiality monad

In this subsection we will define the partiality monad (see below) and show that it is equal to the delay monad quotiented by weak bisimilarity, assuming the axiom of countable choice. We do this proof by defining an intermediate representation as sequences. We then show that the delay monad is equal to sequences (Subsection 3.2.2.1), followed by showing that sequences quotiented by weak bisimilarity are equal to the partiality monad (Subsection 3.2.2.2). This section draws heavy inspiration from "Partiality, Revisited" [6].

Definition 3.2.10 (Partiality Monad)**.** A simple example of a quotient inductive-inductive type is the partiality monad $(-)_\perp$ over a type R, defined by the constructors

$$\frac{}{R_\perp : \mathcal{U}} \tag{3.23}$$

$$\frac{}{\perp : R_\perp} \tag{3.24}$$

$$\frac{a : R}{\eta\ a : R_\perp} \tag{3.25}$$

and a relation $(\cdot \sqsubseteq_\perp \cdot)$ indexed twice over $R_\perp$, with properties

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\bigsqcup\ (\mathtt{s}, \mathtt{b}) : R_\perp} \tag{3.26}$$

$$\frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp\ p\ q : x \equiv y} \tag{3.27}$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{refl}} \tag{3.28}$$

$$\frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z}\ \sqsubseteq_{\mathtt{trans}} \tag{3.29}$$

$$\frac{x : R_\perp}{\perp \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{never}} \tag{3.30}$$

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \bigsqcup(\mathtt{s}, \mathtt{b})} \tag{3.31}$$

$$\frac{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp x}{\bigsqcup(\mathtt{s}, \mathtt{b}) \sqsubseteq_\perp x} \tag{3.32}$$

and finally set truncated by the following constructor

$$\frac{p, q : x \sqsubseteq_\perp y}{p \equiv q}\ (-)_\perp\text{-}\mathtt{isSet} \tag{3.33}$$

3.2.2.1 Delay monad to Sequences

We define sequences, and show they are equal to the delay monad (Theorem 3.2.19). We then extend this equality to an equality between the types quotient by weak bisimilarity (Theorem 3.2.24). We start by defining the type of sequences.

Definition 3.2.11. We define

$$\mathtt{Seq}_R = \sum_{(\mathtt{s}:\mathbb{N}\to R+1)} \mathtt{isMon\ s} \tag{3.34}$$

where

$$\mathtt{isMon\ s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \equiv \mathtt{inr}\ \star) \times (\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \star)) \tag{3.35}$$

meaning a sequences is $\mathtt{inr}\ \star$ until it reaches a point where it switches to $\mathtt{inl}\ r$ for some value r. There are two special cases, the already terminated sequence meaning only $\mathtt{inl}\ r$, and the never terminating sequence meaning only $\mathtt{inr}\ \star$.

For each index in a sequence the element at that index $\mathtt{s}_n$ is either not terminated $\mathtt{s}_n \equiv \mathtt{inr}\ \star$, which we denote as $\mathtt{s}_n \uparrow_{R+1}$, or it is terminated $\mathtt{s}_n \equiv \mathtt{inl}\ r$ with some value r denoted by $\mathtt{s}_n \downarrow_{R+1} r$ or just $\mathtt{s}_n \downarrow_{R+1}$ if we want to say $\mathtt{s}_n \not\equiv \mathtt{inr}\ \star$. Thus we can write $\mathtt{isMon}$ as

$$\mathtt{isMon\ s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \uparrow_{R+1}) \times (\mathtt{s}_{n+1} \downarrow_{R+1})) \tag{3.36}$$

We also introduce notation for the two special cases of sequences given above

$$\mathtt{now}_{\mathtt{Seq}}\ r = (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl\ refl}) \tag{3.37}$$

$$\mathtt{never}_{\mathtt{Seq}} = (\lambda\_, \mathtt{inr}\ \star), (\lambda\_, \mathtt{inl\ refl}) \tag{3.38}$$

Definition 3.2.12. We can shift a sequence $(\mathtt{s}, \mathtt{q})$ by inserting an element and an equality (z_s, z_q) at the front of the sequence,

$$\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ (z_s, z_q) = \begin{cases} z_s & n = 0 \\ \mathtt{s}_m & n = m+1 \end{cases}, \begin{cases} z_q & n = 0 \\ \mathtt{q}_m & n = m+1 \end{cases}, \tag{3.39}$$

Definition 3.2.13. We can unshift a sequence by removing the first element of the sequence

$$\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{s} \circ \mathtt{succ}, \mathtt{q} \circ \mathtt{succ}. \tag{3.40}$$

Lemma 3.2.14. *The function*

$$\mathtt{shift\text{-}unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{shift}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}))\ (\mathtt{s}_0, \mathtt{q}_0) \tag{3.41}$$

is equal to the identity function.

Proof. Unshifting a value followed by a shift, where we reintroduce the value we just remove, gives the sequence we started with. □

Lemma 3.2.15. *The function*

$$\texttt{unshift-shift}\ (\texttt{s}, \texttt{q}) = \texttt{unshift}\ (\texttt{shift}\ (\texttt{s}, \texttt{q})\ \_) \tag{3.42}$$

is equal to the identity function.

Proof. If we shift followed by an unshift, we just introduce a value to instantly remove it, meaning the value does not matter. □

Lemma 3.2.16 (inl≠inr)**.** *For any two elements* $x = \texttt{inl}\ a$ *and* $y = \texttt{inr}\ b$ *then* $x \not\equiv y$.

Proof. The constructors `inl` and `inr` are disjoint, so there does not exists a path between them, meaning constructing one is a contradiction. □

We now define an equivalence between `Delay` R and $\texttt{Seq}_R$, where `later`'s are equivalent to shifts, and `now` r is equivalent to the terminated sequence $\texttt{now}_{\texttt{Seq}}\ r$. We show the equivalence as an isomorphism, meaning we define functions between the types and show the left and right identities for them.

Definition 3.2.17. There is a function from `Delay` R to $\texttt{Seq}_R$

$$\begin{aligned}
&\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{now}\ r) = \texttt{now}_{\texttt{Seq}}\ r \\
&\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{later}\ x) = \\
&\qquad \texttt{shift}\ (\texttt{Delay}{\to}\texttt{Seq}\ x)\ \left(\texttt{inr}\ \star, \begin{cases} \texttt{inr}\ (\texttt{refl}, \texttt{inl}{\not\equiv}\texttt{inr}) & x = \texttt{now}\ _ \\ \texttt{inl}\ \texttt{refl} & x = \texttt{later}\ _ \end{cases}\right)
\end{aligned} \tag{3.43}$$

Definition 3.2.18. We define function from $\texttt{Seq}_R$ to `Delay` R

$$\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{s}, \texttt{q}) = \begin{cases} \texttt{now}\ r & \texttt{s}_0 = \texttt{inl}\ r \\ \texttt{later}\ (\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) & \texttt{s}_0 = \texttt{inr}\ \star \end{cases} \tag{3.44}$$

Theorem 3.2.19. *The type* `Delay` R *is equal to* $\texttt{Seq}_R$

Proof. We start by defining the right identity, which says that for any sequence $(\texttt{s}, \texttt{q})$, we get

$$\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{s}, \texttt{q})) \equiv (\texttt{s}, \texttt{q}) \tag{3.45}$$

We show this by cases analysis on $\texttt{s}_0$, if $\texttt{s}_0 = \texttt{inl}\ r$ then we need to show

$$\texttt{now}_{\texttt{Seq}}\ r \equiv (\texttt{s}, \texttt{q}) \tag{3.46}$$

This is true, since $(\texttt{s}, \texttt{q})$ is a monotone sequence and `inl` r is the top element of the order, so all elements of the sequence are `inl` r. If $\texttt{s}_0 = \texttt{inr}\ \star$, then we need to show

$$\texttt{shift}\ (\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q})))\ (\texttt{inr}\ \star, \_) \equiv (\texttt{s}, \texttt{q}) \tag{3.47}$$

By the induction hypothesis we get

$$\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) \equiv \texttt{unshift}\ (\texttt{s}, \texttt{q}) \tag{3.48}$$

Meaning we are left with showing $\texttt{shift}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))\ (\texttt{inr}\ \star, \_) \equiv (\texttt{s}, \texttt{q})$, which follows from shift and unshift being inverses. For the left identity, we need to show that for any delay monad t we get

$$\texttt{Seq}\to\texttt{Delay}\ (\texttt{Delay}\to\texttt{Seq}\ t) \equiv t \tag{3.49}$$

We do case analysis on t. If $t = \texttt{now}\ a$, then the equality is $\texttt{refl}$. If $t = \texttt{later}\ x$, then we need to show

$$\texttt{later}\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{unshift}\ (\texttt{shift}\ (\texttt{Delay}\to\texttt{Seq}\ x)))) \equiv \texttt{later}\ x \tag{3.50}$$

We again get the equality from unshift and shift being inverse and the induction hypothesis. This completes the isomorphism between $\texttt{Delay}\ R$ and $\texttt{Seq}_R$ and thereby the proof of equality. □

We will define weakly bisimilar relations and show the equality extends to the types quotiented by these relations.

Definition 3.2.20. Weak bisimilarity for the delay monad is given as

$$\frac{a \equiv b}{\texttt{now}\ a \sim_{\texttt{Delay}} \texttt{now}\ b}\ \sim_{\texttt{now}} \tag{3.51}$$

$$\frac{x \sim_{\texttt{Delay}} y}{(\texttt{later}\ x) \sim_{\texttt{Delay}} y}\ \sim_{\texttt{later}_\texttt{l}} \tag{3.52}$$

$$\frac{x \sim_{\texttt{Delay}} y}{x \sim_{\texttt{Delay}} (\texttt{later}\ y)}\ \sim_{\texttt{later}_\texttt{r}} \tag{3.53}$$

$$\frac{x \sim_{\texttt{Delay}} y}{\texttt{later}\ x \sim_{\texttt{Delay}} \texttt{later}\ y}\ \sim_{\texttt{later}} \tag{3.54}$$

We define weak bisimulation for sequences from the following definitions.

Definition 3.2.21 (Sequence Termination)**.** The following relations says that a sequence $(\texttt{s}, \texttt{q}) : \texttt{Seq}_R$ terminates with a given value $r : R$,

$$(\texttt{s}, \texttt{q}) \downarrow_{\texttt{Seq}} r = \sum_{(n:\mathbb{N})} \texttt{s}_n \downarrow_{\texttt{R+1}} r. \tag{3.55}$$

Definition 3.2.22 (Sequence Ordering)**.** We define an ordering of sequences as

$$(\texttt{s}, \texttt{q}) \sqsubseteq_{\texttt{Seq}} (\texttt{t}, \texttt{p}) = \prod_{(a:R)} (\|\texttt{s} \downarrow_{\texttt{Seq}} a\| \to \|\texttt{t} \downarrow_{\texttt{Seq}} a\|) \tag{3.56}$$

Definition 3.2.23. Weak bisimilarity for sequences is given as

$$(\texttt{s}, \texttt{q}) \sim_{\texttt{Seq}} (\texttt{t}, \texttt{p}) = (\texttt{s}, \texttt{q}) \sqsubseteq_{\texttt{Seq}} (\texttt{t}, \texttt{p}) \times (\texttt{t}, \texttt{p}) \sqsubseteq_{\texttt{Seq}} (\texttt{s}, \texttt{q}) \tag{3.57}$$

Theorem 3.2.24. *The types* $\texttt{Delay}\ R/\sim_{\texttt{Delay}}$ *and* $\texttt{Seq}_R/\sim_{\texttt{Seq}}$ *are equal.*

Proof. We show that transporting across the equality from Theorem 3.2.19, respects weak bisimilarity. We start by showing if $x \sim_{\texttt{Delay}} y$, then $\texttt{Delay}\to\texttt{Seq}\ x \sim_{\texttt{Seq}} \texttt{Delay}\to\texttt{Seq}\ y$ by case. If the weak bisimilarity is $\sim_{\texttt{now}}$, then we need to construct $\texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} \texttt{now}_{\texttt{Seq}}\ b$ given a path $p : a \equiv b$. By reflexivity of $\sim_{\texttt{Seq}}$ we get $\texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} \texttt{now}_{\texttt{Seq}}\ a$. By transporting over p, we get the wanted term $\texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} \texttt{now}_{\texttt{Seq}}\ b$. If we have $\sim_{\texttt{later}_1}$, then by induction we get $\texttt{Delay}\to\texttt{Seq}\ x \sim_{\texttt{Seq}} \texttt{Delay}\to\texttt{Seq}\ y$. Shifting one side will not break the bisimilarity, and a shift interacts with $\texttt{Delay}\to\texttt{Seq}$ to add a $\texttt{later}$ that is we get $\texttt{Delay}\to\texttt{Seq}\ (\texttt{later}\ x) \sim_{\texttt{Seq}} \texttt{Delay}\to\texttt{Seq}\ y$,

which is what we needed to show. By a symmetric argument we get $\sim_{\texttt{later}_\texttt{r}}$. To construct $\sim_{\texttt{later}}$ we shift both sides, resulting in a `later` being added to both sides of the bisimilarity.

$$\begin{aligned}
&\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ (\sim_{\texttt{now}}\ a\ b\ p) = \texttt{subst}\ (\lambda k,\ \texttt{now}_{\texttt{Seq}}\ a\ \sim_{\texttt{Seq}}\ k)\ p\ (\sim_{\texttt{refl}}\ a)\\
&\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ (\sim_{\texttt{later}_\texttt{l}}\ x\ y\ r) = \sim_{\texttt{shift}}\ (\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ r)\\
&\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ (\sim_{\texttt{later}_\texttt{r}}\ x\ y\ r) = \sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{sym}}\ (\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ r)))\\
&\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ (\sim_{\texttt{later}}\ x\ y\ r) = \sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{Delay}\to\sim_{\texttt{Seq}}}\ r))))
\end{aligned} \tag{3.58}$$

Now for the other direction from $(\texttt{s},\texttt{q})\ \sim_{\texttt{Seq}}\ (\texttt{t},\texttt{p})$ to $\texttt{Seq}\to\texttt{Delay}\ (\texttt{s},\texttt{q})\ \sim_{\texttt{Delay}}\ \texttt{Seq}\to\texttt{Delay}\ (\texttt{t},\texttt{p})$. We start with some helpful properties. We know that if $\texttt{s}\ 0 = \texttt{inl}\ r$, then $(\texttt{s},\texttt{q}) = \texttt{now}_{\texttt{Seq}}\ r$ and $(\texttt{s},\texttt{q})\downarrow_{\texttt{Seq}} r$. If two sequences are weakly bisimilar and they terminate to different values, then the values are equal, we therefore have

$$\sim_{\texttt{seq-val}} : \texttt{now}_{\texttt{Seq}}\ a\ \sim_{\texttt{Seq}}\ \texttt{now}_{\texttt{Seq}}\ b \to a \equiv b \tag{3.59}$$

We do the proof by case on $\texttt{s}\ 0$ and $\texttt{t}\ 0$. If both sequences are terminated meaning $(\texttt{s},\texttt{q}) = \texttt{now}_{\texttt{Seq}}\ a$ and $(\texttt{t},\texttt{p}) = \texttt{now}_{\mathcal{S}eq}\ b$, then we can make a term $p : a \equiv b$ using $\sim_{\texttt{seq-val}}$. The base case is then done by $\sim_{\texttt{now}}\ p$. If two sequences are weakly bisimilar, then shifting or unshifting either of them will not break the bisimilarity. We can therefore do the rest of the cases by induction.

$$\begin{aligned}
&\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\texttt{now}_{\texttt{Seq}}\ a)\ (\texttt{now}_{\texttt{Seq}}\ b)\ r = \sim_{\texttt{now}}\ a\ b\ (\sim_{\texttt{seq-val}}\ r)\\
&\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\texttt{now}_{\texttt{Seq}}\ a)\ (\texttt{t},\texttt{p})\ r =\\
&\quad \sim_{\texttt{later}_\texttt{r}}\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{s},\texttt{q}))\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{shift}\ (\texttt{t},\texttt{p})\ _))\ (\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\sim_{\texttt{shift}_\texttt{r}}\ r))\\
&\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\texttt{s},\texttt{q})\ (\texttt{now}_{\texttt{Seq}}\ b)\ r =\\
&\quad \sim_{\texttt{later}_\texttt{l}}\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{shift}\ (\texttt{s},\texttt{q})\ _))\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{t},\texttt{p}))\ (\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\sim_{\texttt{shift}_\texttt{l}}\ r))\\
&\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\texttt{s},\texttt{q})\ (\texttt{p},\texttt{t})\ r =\\
&\quad \sim_{\texttt{later}}\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{shift}\ (\texttt{s},\texttt{q})\ _))\ (\texttt{Seq}\to\texttt{Delay}\ (\texttt{shift}\ (\texttt{t},\texttt{p})\ _))\\
&\qquad (\sim_{\texttt{Seq}\to\sim_{\texttt{Delay}}}\ (\sim_{\texttt{shift}}\ r))
\end{aligned} \tag{3.60}$$

We have now given functions in both direction, meaning we get that weakly bisimilar relations for the delay monad and sequences respects the equality given in Theorem 3.2.19 so the quotiented types are also equal. □

3.2.2.2 Sequence to Partiality Monad

In this section we will show that assuming the axiom of countable choice, we get an equivalence between sequences quotiented by weak bisimilarity and the partiality monad (Theorem 3.2.38). We start with some useful definitions.

Definition 3.2.25. There is a function from $R + \mathbf{1}$ to the partiality monad $R_\perp$

$$\begin{aligned}
&\texttt{Maybe}\to(-)_\perp\ (\texttt{inl}\ r) = \eta\ r\\
&\texttt{Maybe}\to(-)_\perp\ (\texttt{inr}\ \star) = \perp
\end{aligned} \tag{3.61}$$

Definition 3.2.26 (Maybe Ordering)**.** We define an ordering relation on $R + \mathbf{1}$ as

$$x \sqsubseteq_{\texttt{R+1}}\ y = (x \equiv y) + ((x\downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}})) \tag{3.62}$$

This ordering definition is basically `isMon` at a specific index, so we can rewrite `isMon` to

$$\texttt{isMon}\ \texttt{s} = \prod_{(n:\mathbb{N})} \texttt{s}_n \sqsubseteq_{\texttt{R+1}} \texttt{s}_{n+1} \tag{3.63}$$

This confirms the idea that if `isMon s`, then `s` is monotone and therefore a sequence of partial values.

Lemma 3.2.27. *The function* $\texttt{Maybe}{\to}(-)_\perp$ *is monotone, meaning if* $x \sqsubseteq_{\texttt{R+1}} y$ *for some* x *and* y*, then* $(\texttt{Maybe}{\to}(-)_\perp\ x) \sqsubseteq_\perp (\texttt{Maybe}{\to}(-)_\perp\ y)$.

Proof. We do the proof by case.

$$\begin{aligned}
&\texttt{Maybe}{\to}(-)_\perp\texttt{-mono}\ (\texttt{inl}\ p) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\to}(-)_\perp\ x \sqsubseteq_\perp \texttt{Maybe}{\to}(-)_\perp\ a)\ p\ (\sqsubseteq_{\texttt{refl}}\ (\texttt{Maybe}{\to}(-)_\perp\ x)) \\
&\texttt{Maybe}{\to}(-)_\perp\texttt{-mono}\ (\texttt{inr}\ (p, _)) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\to}(-)_\perp\ a \sqsubseteq_\perp \texttt{Maybe}{\to}(-)_\perp\ y)\ p^{-1}\ (\sqsubseteq_{\texttt{never}}\ (\texttt{Maybe}{\to}(-)_\perp\ y))
\end{aligned} \tag{3.64}$$

□

Definition 3.2.28. There is a function taking a sequence to an increasing sequence of partiality monads

$$\texttt{Seq}{\to}\texttt{incSeq}\ (\texttt{s}, \texttt{q}) = \texttt{Maybe}{\to}(-)_\perp \circ \texttt{s}\ ,\ \texttt{Maybe}{\to}(-)_\perp\texttt{-mono} \circ \texttt{q} \tag{3.65}$$

Definition 3.2.29. There is a function taking a sequence to the partiality monad

$$\begin{aligned}
&\texttt{Seq}{\to}(-)_\perp : \texttt{Seq}_R \to R_\perp \\
&\texttt{Seq}{\to}(-)_\perp\ (\texttt{s}, \texttt{q}) = \bigsqcup (\texttt{Seq}{\to}\texttt{incSeq}\ (\texttt{s}, \texttt{q}))
\end{aligned} \tag{3.66}$$

Lemma 3.2.30. *The function* $\texttt{Seq}{\to}(-)_\perp$ *is monotone.*

$$\texttt{Seq}{\to}(-)_\perp\texttt{-mono} : (x\ y : \texttt{Seq}_R) \to x \sqsubseteq_{\texttt{Seq}} y \to \texttt{Seq}{\to}(-)_\perp\ x \sqsubseteq_\perp \texttt{Seq}{\to}(-)_\perp\ y \tag{3.67}$$

Proof. Given two sequences, where one is smaller than the other, then taking the least upper bounds of each sequence respects the ordering. □

Lemma 3.2.31. *If two sequences* x, y *are weakly bisimular, then* $\texttt{Seq}{\to}(-)_\perp\ x \equiv \texttt{Seq}{\to}(-)_\perp\ y$

Proof. By the definition of bisimilarity for sequences we get $x \sqsubseteq_{\texttt{Seq}} y$ and $y \sqsubseteq_{\texttt{Seq}} x$ from the bisimilarity of x and y. By monotonicity of $\texttt{Seq}{\to}(-)_\perp$ we get $\texttt{Seq}{\to}(-)_\perp\ x \sqsubseteq_\perp \texttt{Seq}{\to}(-)_\perp\ y$ and $\texttt{Seq}{\to}(-)_\perp\ y \sqsubseteq_\perp \texttt{Seq}{\to}(-)_\perp\ x$. We then use $\alpha_\perp$ to get the equality $\texttt{Seq}{\to}(-)_\perp\ x \equiv \texttt{Seq}{\to}(-)_\perp\ y$. We denote this construction as the function $\texttt{Seq}{\to}(-)_\perp\texttt{-}{\sim}{\to}{\equiv}$. □

The recursor for quotients Definition 3.1.1 allows us to lift the function $\texttt{Seq}{\to}(-)_\perp$ to the quotient.

Definition 3.2.32. We define a function $\texttt{Seq}/{\sim}{\to}(-)_\perp$ from $\texttt{Seq}_R/{\sim_{\texttt{Seq}}}$ to $R_\perp$

$$\texttt{Seq}/{\sim}{\to}(-)_\perp = \texttt{rec}\ \texttt{Seq}{\to}(-)_\perp\ \texttt{Seq}{\to}(-)_\perp\texttt{-}{\sim}{\to}{\equiv}\ (-)_\perp\texttt{-isSet} \tag{3.68}$$

Lemma 3.2.33. *Given two sequences* s *and* t*, if* $\texttt{Seq}{\to}(-)_\perp\ s \equiv \texttt{Seq}{\to}(-)_\perp\ t$*, then* $s \sim_{\texttt{Seq}} t$.

Proof. We can reduce the burden of the proof, since

$$s \sim_{\mathtt{Seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\mathtt{Seq}} r\| \to \|y \downarrow_{\mathtt{Seq}} r\|\right) \times \left(\prod_{(r:R)} \|y \downarrow_{\mathtt{Seq}} r\| \to \|x \downarrow_{\mathtt{Seq}} r\|\right) \tag{3.69}$$

meaning we can show one part and get the other for free by symmetry. We assume $\|x \downarrow_{\mathtt{Seq}} r\|$, to show $\|y \downarrow_{\mathtt{Seq}} r\|$. By the mapping property of propositional truncation, we reduce the proof to defining a function from $x \downarrow_{\mathtt{Seq}} r$ to $y \downarrow_{\mathtt{Seq}} r$. Since $x \downarrow_{\mathtt{Seq}} r$, then $\eta\, r \sqsubseteq_\perp \mathtt{Seq}{\to}(-)_\perp\, x$, but we have assumed $\mathtt{Seq}{\to}(-)_\perp\, x \equiv \mathtt{Seq}{\to}(-)_\perp\, y$, so we get $\eta\, r \sqsubseteq_\perp \mathtt{Seq}{\to}(-)_\perp\, y$ and thereby $y \downarrow_{\mathtt{Seq}} r$. □

Lemma 3.2.34. *The function* $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ *is injective, meaning* $\mathtt{Seq}/{\sim}{\to}(-)_\perp\, x \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\, y$ *implies* $x \equiv y$.

Proof. We use propositional elimination of quotients (see Definition 3.1.2) to show injectivity. We let

$$\mathtt{P} = \mathtt{Seq}/{\sim}{\to}(-)_\perp\, x \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\, y \to x \equiv y \tag{3.70}$$

which is a proposition by $\mathtt{P}_{\mathtt{prop}} = \mathtt{isPropΠ}\,(\lambda \_, \mathtt{squash}/\, x\, y)$. We do the propositional elimination twice, first we index by x, and then by y for the second elimination with $x = [\,a\,]$. The proof is then given as

$$\mathtt{elimProp}\,(\lambda\, a,\ \mathtt{elimProp}\,(\lambda\, b,\ \mathtt{eq}/\ a\ b \circ \mathtt{Seq}{\to}(-)_\perp\text{-}{\equiv}{\to}{\sim}\ a\ b)\ y)\ x \tag{3.71}$$

where $\mathtt{Seq}{\to}(-)_\perp\text{-}{\equiv}{\to}{\sim}$ is Lemma 3.2.33. □

Lemma 3.2.35. *For all constant sequences* s*, where all elements have the same value* v*, we get* $\mathtt{Seq}{\to}(-)_\perp\, s \equiv \mathtt{Maybe}{\to}(-)_\perp\, v$.

Proof. The left side of the equality reduces to $\mathtt{Maybe}{\to}(-)_\perp$ applied on the least upper bound of the constant sequence, which is exactly the right hand side of the equality. □

Definition 3.2.36 (Propositional eliminator for $(-)_\perp$)**.** Given a dependent type $\mathtt{P} : R_\perp \to \mathcal{U}$ that is a proposition at all points $\mathtt{P}_{\mathtt{prop}} : \prod_{(x:R_\perp)} \mathtt{isProp}\ (\mathtt{P}\ x)$, and given a proof for the base cases $P_\perp : \mathtt{P}\ \perp$ and $\mathtt{P}_\eta : \prod_{(a:R)} \mathtt{P}\ (\eta\ a)$, and a proof $\mathtt{P}_\sqcup$ that for all $(\mathtt{s}, \mathtt{q}) : \sum_{(\mathtt{s}:\mathbb{N}\to R_\perp)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}$ we get $\left(\prod_{(n:\mathbb{N})} \mathtt{P}\ \mathtt{s}_n\right) \to \mathtt{P} \bigsqcup (\mathtt{s}, \mathtt{q})$. Then we can define a function $\mathtt{elimProp}_\perp : \prod_{(x:R_\perp)} \mathtt{P}\ x$, as

$$\begin{aligned}
&\mathtt{elimProp}_\perp\ \perp = P_\perp \\
&\mathtt{elimProp}_\perp\ (\eta\ a) = \mathtt{P}_\eta\ a \\
&\mathtt{elimProp}_\perp\ \sqcup\ (\mathtt{s}, \mathtt{q}) = \mathtt{P}_\sqcup\ (\mathtt{s}, \mathtt{q})\ (\mathtt{elimProp}_\perp \circ \mathtt{s}) \\
&\mathtt{elimProp}_\perp\ (\alpha\ p\ q\ i) = \\
&\quad \mathtt{isProp}{\to}\mathtt{isDepProp}\ \mathtt{P}_{\mathtt{prop}}\ (\mathtt{elimProp}_\perp\ x)\ (\mathtt{elimProp}_\perp\ y)\ (\alpha\ p\ q)\ i \\
&\mathtt{elimProp}_\perp\ (\perp\text{-}\mathtt{isSet}\ x\ y\ p\ q\ i\ j) = \\
&\quad \mathtt{isSet}{\to}\mathtt{isDepSet}\ (\mathtt{isProp}{\to}\mathtt{isSet} \circ \mathtt{P}_{\mathtt{prop}})\ (\mathtt{elimProp}_\perp\ x)\ (\mathtt{elimProp}_\perp\ y) \\
&\qquad (\mathtt{ap}\ \mathtt{elimProp}_\perp\ p)\ (\mathtt{ap}\ \mathtt{elimProp}_\perp\ q)\ (\perp\text{-}\mathtt{isSet}\ x\ y\ p\ q)\ i\ j
\end{aligned} \tag{3.72}$$

where x and y in the α case are the endpoints of the paths p and q.

Lemma 3.2.37. *The function* $\mathtt{Seq}{\to}(-)_\perp$ *is surjective, assuming the axiom of countable choice. This means that for all* $b : R_\perp$ *we have* $\|\Sigma_{(x:\mathtt{Seq}_R)}\ \mathtt{Seq}{\to}(-)_\perp\ x \equiv b\|$.

Proof. We do the proof by using the propositional eliminator for the partiality monad, with

$$\mathtt{P} = \lambda b, \left\| \sum_{(x:\mathtt{Seq}_R)} \mathtt{Seq}{\to}(-)_\perp\ x \equiv b \right\| \tag{3.73}$$

and $\mathtt{P}_{\mathtt{prop}}$ follows from the propositional truncation. For $\perp$ and $\eta\ a$, we use $\mathtt{now}_{\mathtt{Seq}}\ a$ and $\mathtt{never}_{\mathtt{Seq}}$ respectively, and by Lemma 3.2.35 we get $\mathtt{P}\ \perp$ and $\mathtt{P}\ (\eta\ a)$. For the least upper bound $\bigsqcup(\mathtt{s}, \mathtt{q})$, we may assume $\prod_{(n:\mathbb{N})}\ \mathtt{P}\ \mathtt{s}_n$, which unfolds to the expression

$$\prod_{(n:\mathbb{N})} \left\| \sum_{(x:\mathtt{Seq}_R)} \mathtt{Seq}{\to}(-)_\perp\ x \equiv \mathtt{s}\ n \right\| \tag{3.74}$$

which by the axiom of countable choice becomes

$$\left\| \prod_{(n:\mathbb{N})} \sum_{(x:\mathtt{Seq}_R)} \mathtt{Seq}{\to}(-)_\perp\ x \equiv \mathtt{s}\ n \right\| \tag{3.75}$$

By swapping Π and Σ and functional extensionality, we get

$$\left\| \sum_{(\mathtt{f}:\mathbb{N}\to\mathtt{Seq}_R)} \mathtt{Seq}{\to}(-)_\perp \circ \mathtt{f} \equiv \mathtt{s} \right\| \tag{3.76}$$

which is a pointwise equivalence, which implies an equivalence for the upper bounds, giving us

$$\left\| \sum_{(x:\mathtt{Seq}_R)} \mathtt{Seq}{\to}(-)_\perp\ x \equiv \bigsqcup(\mathtt{s}, \mathtt{q}) \right\| \tag{3.77}$$

Thus we have shown all the cases needed to invoke the propositional eliminator for the partiality monad, completing the proof of surjectivity. □

Corollary 1. *The function* $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ *is surjective assuming the axiom of countable choice.*

Proof. Since $\mathtt{Seq}{\to}(-)_\perp$ is surjective, we can map into the quotient with $[\cdot]$. □

Theorem 3.2.38. *Sequences quotiented by weak bisimilarity is equivalent to the partiality monad, that is* $\mathtt{Seq}_R/{\sim_{\mathtt{Seq}}} \equiv R_\perp$*, assuming the axiom of countable choice.*

Proof. The function $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ is injective (Lemma 3.2.34) and surjective (Corollary 1) assuming the axiom of countable choice, meaning we get an equivalence. □

3.3 How should QM-types be defined

We want to define quotient M-type with a good semantic, we draw inspiration from QW-types [11], quotient containers [2] and quotient polynomial functors (QPF) [8]. We start by looking at an alternative way of defining the QM-types.

3.3.1 Lifting Quotient Construction from Containers

An alternative to directly set quotienting the M-types is to do the quotienting on the underlying container [2] / polynomial functor [8], and then do the same fixed point construction we did for polynomial functors, but on the quotiented functor instead. We start by defining quotiented polynomial functors

Definition 3.3.1. Given a container $(A, \mathtt{B})$ and any type X a family of relations $\sim_a : (\mathtt{B}\, a \to X) \to (\mathtt{B}\, a \to X) \to \mathcal{U}$ indexed by A, we define a quotiented polynomial functor (QPF), for types as

$$\mathtt{F}\, X = \sum_{(a:A)} ((\mathtt{B}\, a \to X)/\sim_a) \tag{3.78}$$

and for a function $\mathtt{f} : X \to Y$, we use the quotient eliminator from Definition 3.1.3 with $\mathtt{P} = \lambda \_, (\mathtt{B}\, a \to Y)/\sim_a$ which is a set by squash/, since we are using set truncated quotients. The base case is $\mathtt{f} = \lambda \_, [\mathtt{f} \circ \mathtt{g}]$ and equality case is $\mathtt{f}_{\mathtt{eq}} = \lambda\, \mathtt{x}\, \mathtt{y}\, r$, eq/ $(\mathtt{f} \circ \mathtt{x})$ $(\mathtt{f} \circ \mathtt{y})$ $(\sim_{\mathtt{ap}}\ \mathtt{f}\ r)$, where $\sim_{\mathtt{ap}}$ says that given a function f and $\mathtt{x} \sim_a \mathtt{y}$ then $\mathtt{f} \circ \mathtt{x} \sim_a \mathtt{f} \circ \mathtt{y}$. With this the definition for the quotient polynomial functor for functions is

$$\mathtt{F}\, \mathtt{f}\ (a, \mathtt{g}) = (a, \mathtt{elim}\ \mathtt{g}) \tag{3.79}$$

completing the definition of a quotiented polynomial functor.

If there is a function $\mathtt{abs}_X : \mathtt{P}\, X \to \mathtt{F}\, X$ that makes the diagram in Figure 3.1 commute (as in [8]), then we can construct the final F-coalgebra, to get another notion of quotiented M-type. The construction is done by taking the limit of the repeated application of P and F to the unit type $\mathbf{1}$. The proof that this limit exists and that it makes a final coalgebra is left as future research. If

$$\begin{array}{ccc} \mathtt{P}\, X & \xrightarrow{\mathtt{P}\, \mathtt{f}} & \mathtt{P}\, Y \\ {\scriptstyle \mathtt{abs}_X}\downarrow & & \downarrow{\scriptstyle \mathtt{abs}_Y} \\ \mathtt{F}\, X & \xrightarrow{\mathtt{F}\, \mathtt{f}} & \mathtt{F}\, Y \end{array}$$

Figure 3.1: Quotiented polynomial function

abs is surjective, then the square will commute. We therefore just have to define a relation that has the $\sim_{\mathtt{ap}}$ property together with a surjective function $\mathtt{abs}_X$ to construct a quotient M-type. We get a surjective function $\mathtt{QM\text{-}intro} : \mathtt{M}_S \to \mathtt{M}_S/\sim$ from this construction. The function is given as $\mathtt{abs}_{\mathtt{M}}$, which is the version of abs between the limits of the two sequences. As an example of this alternative construction, lets try and construct the QM-type and the QPF for multisets,

Example 3. We have the following polynomial functor for A-valued $\mathbb{N}$-branching trees

$$\mathtt{P}\, X = \sum_{(a:A+\mathbf{1})} \begin{cases} \mathbf{0} \to X & a = \mathtt{inl}\ r \\ \mathbb{N} \to X & a = \mathtt{inr}\ \star \end{cases} \tag{3.80}$$

for which we define the family of relations $(\sim_{\mathtt{PT}})_{(a:A+\mathbf{1})}$, if $a = \mathtt{inl}\ r$, then the equality relations is just the trivial equalities, since the relation is between elements of a contractive type $\mathbf{0} \to X$. On the other hand if $a = \mathtt{inr}\ \star$, then we define the relation as

$$\frac{\mathtt{f}, \mathtt{h} : \mathbb{N} \to X \quad \mathtt{g} : \mathbb{N} \to \mathbb{N} \quad \mathtt{isIso}\ \mathtt{g} \quad \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}}{\mathtt{f} \sim_{\mathtt{PT}} \mathtt{h}} \tag{3.81}$$

With this approach, we only need to define the non-trivial equalities, meaning those that are not just reflexivity. This relations fulfills the $\sim_{\mathtt{ap}}$ property. We can see this by case. If $a = \mathtt{inl}\ r$, then it holds trivially. If $a = \mathtt{inr}\ \star$, then we have $\mathtt{f} \sim_{\mathtt{PT}} \mathtt{h}$ and want to show $\mathtt{k} \circ \mathtt{f} \sim_{\mathtt{PT}} \mathtt{k} \circ \mathtt{h}$ for any function $\mathtt{k}$, which just boils down to showing $\mathtt{k} \circ \mathtt{f} \circ \mathtt{g} \equiv \mathtt{k} \circ \mathtt{h}$ given the path $p : \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}$, which is done by $\mathtt{ap}\ \mathtt{k}\ p$. We therefore have a quotient polynomial functor $\mathtt{F}$. We then define $\mathtt{abs}$ as

$$\begin{aligned} \mathtt{abs}\ (\mathtt{inl}\ r\ ,\ \lambda\,()) &= (\mathtt{inl}\ r\ ,\ [\lambda\,()]) \\ \mathtt{abs}\ (\mathtt{inr}\ \star\ ,\ \mathtt{f}) &= (\mathtt{inr}\ \star\ ,\ [\mathtt{f}]) \end{aligned} \tag{3.82}$$

which is surjective, because of the surjectivity of $[\cdot]$. Taking the limit we get the commuting square in Figure 3.2.

$$\begin{array}{ccc} \mathtt{T}\ A & \xrightarrow{\mathtt{out}} & \mathtt{P}\ (\mathtt{T}\ A) \\ {\scriptstyle \mathtt{QM\text{-}intro}}\downarrow & & \downarrow{\scriptstyle \mathtt{abs}_{\mathtt{P}\ (\mathtt{T}\ A)}} \\ \mathtt{MS}\ A & \xrightarrow{\mathtt{out}_{\mathtt{QM}}} & \mathtt{F}\,(\mathtt{MS}\ A) \end{array}$$

Figure 3.2: QPF and limit diagram for multiset construction

We have just postulated that the quotiented $\mathtt{M}$-type, is a final $\mathtt{F}$-coalgebra, however this is not necessarily the case. When trying to proof this we need to show that given $\mathtt{fq} : (\mathtt{B}\ a \to X)/\mathtt{R}$ then we can construct a function $(\mathtt{B}\ a \to X)$, however this requires the axiom of choice in general. We therefore does not get around the axiom of choice with this construction, however it is an alternative way of defining the quotient, and we get some more structure using it.

3.3.2 Constructing QIITs from M-types and Relations

The construction of QW-types in [11] got around using the axiom of choice by using QIIT, that is defining the type and the relation to quotient it by at the same time. As we have seen in the examples in this thesis, using QIITs gets around the use of the axiom of choice without adding much complexity, and if we use another construction and assume the axiom of choice, that construction will be equal to the QIIT. We still need to show that the type defined by the QIIT is a final coalgebra, however this is left at future research. We will however describe how to construct QIITs from an $\mathtt{M}$-type and a relation. You simply take all the constructors/destructors of the $\mathtt{M}$-type, and define them as the type forming rules for the quotient. Then for all constructors of the relations, we replace the relation with equality and add those rules as well. This will give us an higher inductive-inductive type (HIIT), so we add a set truncation constructor to get a QIIT.

The structure for translating between the set quotiented M-type and this construction of a QIIT is rather simple

- A function $\mathtt{QM{\to}QIIT}$ from $\mathtt{QM}$ to QIIT, taking the $\mathtt{QM}$ constructors to their corresponding QIIT constructor.
- A proof $\mathtt{QM{\to}QIIT\text{-}{\sim}{\to}{\equiv}}$ that $x \sim_{\mathtt{QM}} y$ implies $\mathtt{QM{\to}QIIT}\ x \equiv \mathtt{QM{\to}QIIT}\ y$, which should follow by the constructors we added to the QIIT for the relation.
- Lifting $\mathtt{QM{\to}QIIT}$ to $\mathtt{QM/{\sim}{\to}QIIT}$ using the quotient recursor with $\mathtt{QM{\to}QIIT\text{-}{\sim}{\to}{\equiv}}$ and $\mathtt{QIIT\text{-}isSet}$

- Showing injectivity by using propositional elimination of the quotient together with the inverse of QM→QIIT-∼→≡, namely QM→QIIT-injective saying that QM→QIIT $x \equiv$ QM→QIIT y implies $x \sim_{\mathtt{QM}} y$, which again should follow from how the relations constructors were defined.
- Lastly we show surjectivity by induction using the eliminator of QIIT and the axiom of choice and the surjectivity of $[\cdot]$.

That this translation exists enforces the idea that QIITs have the right semantic, while behaving better than the QM-types. We followed this construction in Subsection 3.2.1 on multisets, however in Subsection 3.2.2 on the partiality monad, we constructed a more advanced QIIT. Lets look at the reason for constructing more advanced QIITs, by looking at a use of the simple method.

Example 4. We combine the constructors for the delay monad given in Figure 2.7 and the weak bisimilarity relation given in Definition 3.2.20 to get a QIIT defined by

$$\frac{x : R}{\mathtt{now}\ x : R_\perp} \quad (3.83) \qquad \frac{t : R_\perp}{\mathtt{later}\ t : R_\perp} \quad (3.84)$$

$$\frac{x \equiv y}{\mathtt{later}\ x \equiv y}\ \mathtt{later\text{-}l}_\perp \quad (3.85) \qquad \frac{x \equiv y}{x \equiv \mathtt{later}\ y}\ \mathtt{later\text{-}r}_\perp \quad (3.86)$$

$$\frac{R : \mathcal{U}}{\mathtt{isSet}\ R_\perp}\ (-)_\perp\text{-}\mathtt{isSet} \quad (3.87)$$

Constructing a function from the QM-type to this QIIT, we can show the equality to this QIIT without the use of the axiom of countable choice. The reason is that this type is not as expressive as the partiality monad, that is we quotient away too much information. In the partiality monad we have an ordering, however this ordering is not present here.

We can see that doing set truncated quotients probably removes more information than we would want. We would like to quantify this loss of information, however this is left at future research.

Chapter 4

Conclusion

We have described the construction of M-types as the limit of a sequence defined from repeated application of a polynomial functor for a container. We have showed that M-types are a final coalgebra, and used this to define a coinduction principle for M-types, giving us an equality relation from (strong) bisimulation. This work has been formalized in the Cubical Agda proof assistant, and part of the formalization has been accepted to the Cubical Agda GitHub repository. We have used this formalization to construct a couple of examples of M-types, to make the theory of M-types more accessible. We have given some guidelines for how to construct a container, that will give an M-type with the wanted set of constructors/destructors. We have also explored the concept of quotiented M-types (QM-types), and shown examples where we need the axiom of choice, to lift the constructor to the quotient. We have shown this problem can be solved by using quotient inductive-inductive types (QIITs), and that the type that arise from this construction is equal to the quotiented M-type if we assume the axiom of choice. We have show an alternative construction of QM-types, where we quotient the polynomial functor before taking the limit, however this construction also encounters the same problem as quotienting the M-type directly, namely it needs the axiom of choice to work. We have seen that constructing QIITs are not easy, however sometimes a simple construction is enough.

4.1 Future Work

We would like to formally show that the construction of quotiented M-types presented in this paper, will result in a type that is a final-coalgebra. We would also like to do closer comparison between the constructions. We would also like to get a formal description/proof of how to construct the quotient inductive-inductive types representing the quotient of a given M-type over a given relation.

In this work we have described M-types, however we have not looked at the notion of indexed M-types, which are coinductive types indexed by another type. We hope that what is done in this thesis can be generalized to work for indexed M-types, but formally showing this is future work.

We similarly only discussed that types defined as (coinductive) records should be equal to types defined as M-types, but we did not discuss how exactly to formulate this equality and proof it. We would like to show that the construction of M-types is equal to the construction of M-types as a record. This would give us the external interpretation of constructors in records, to be able to use more features of the termination checker and the likes in the Cubical Agda proof assistant.

We would also like to explore how these constructions behave in a guarded cubical type theory (GCTT), and whether there are any improvements to gain from using a GCTT in this regard, since most of the issues in the Cubical Agda formalization stem from the termination checker.

Bibliography

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 23–38, 2003.

[2] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, pages 2–15, 2004.

[3] Michael Gordon Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. δ for data: Differentiating data structures. *Fundam. Inform.*, 65(1-2):1–28, 2005.

[4] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, pages 17–30, 2015.

[5] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. *CoRR*, abs/1612.02346, 2016.

[6] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures*, pages 534–549, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[7] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29, 2016.

[8] Jeremy Avigad, Mario M. Carneiro, and Simon Hudon. Data types as quotients of polynomial functors. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, pages 6:1–6:19, 2019.

[9] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[10] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[11] Marcelo Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. *CoRR*, abs/1911.06899, 2019.

[12] nLab authors. cubical type theory. http://ncatlab.org/nlab/show/cubical%20type%20theory, May 2020. Revision 15.

[13] nLab authors. homotopy type theory. http://ncatlab.org/nlab/show/homotopy%20type%20theory, May 2020. Revision 111.

[14] nLab authors. Martin-Löf dependent type theory. http://ncatlab.org/nlab/show/Martin-L%C3%B6f%20dependent%20type%20theory, June 2020. Revision 22.

[15] nLab authors. propositions as types. http://ncatlab.org/nlab/show/propositions%20as%20types, June 2020. Revision 40.

[16] nLab authors. type theory. http://ncatlab.org/nlab/show/type%20theory, June 2020. Revision 121.

[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[18] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. Preprint available at http://www.cs.cmu.edu/ amoertbe/papers/cubicalagda.pdf, 2019.

[19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.

Appendix A

The Technical Details

Most of the work presented has been formalized in Cubical Agda. Part of the formalization of this work has been accepted to the Cubical Agda GitHub in the pull request:
`https://github.com/agda/cubical/pull/245`
The rest can be accessed at:
`https://github.com/cmester0/MastersThesis/tree/a05099b2e433d5002c1151749aa3b65e1eb18b3d/src`