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

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science
June 9, 2020
Advisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

We present a construction of M-types from containers in a cubical type theory (CTT), We show how the containers construct a coalgebra, for which we can define a coinduction principle, making strong bisimulation imply equality. We then show constructions of M-types, and how they can be quotiented to construct what we call QM-types. The problem with QM-types is that in general assuming that we can lift function types of equivalence classes is equivalent to the axiom of choice [6], but this can be solved by defining the quotienting relation and the type at the same time as a quotient inductive-inductive type (QIIT), which assuming the axiom of (countable) choice, is equal to the QM-type. We construct multisets and the partiality monad as some examples of quotiented M-types and show the equivalence between the two constructions. We then discuss the pros and cons of the two constructions, and relate them to an alternative construction, which defines quotient polynomial functors (QPF), and define the quotiented M-type as the final coalgebra for the QPF. However this construction requires the axiom of choice, so it is still not a suitable construction. We conclude with some examples of how to use M-types and some properties. All work is formalized in Cubical Agda, and the work on defining M-types has been accepted to the Cubical Agda github repository.

complete

Resumé

I denne afhandling vil vi give en konstruktion af M-typer ved brug af "containers". Formaliseringen af dette bliver udført i en "Cubical Type Theory". Given en "conatiner" kan vi konstruere en polynomisk funktor, og ved gentaget anvendelse af denne på enhedstypen, får vi en sekvens af typer, som vi kan tage grænsen af. Denne grænse giver os en endelig coalgebra, hvilket vi definere M-type som. Vi giver nogle eksempler på M-typer. Vi beskriver tre mulige definitioner for kvotient M-typer, kaldet en QM-type, den første er at bruge set trunkeret kvotienter givet en relation, problemet ved denne defintion er at vi ikke kan løfte konstruktørerne for M-typerne til QM-typen. Vores anden defination er kvotient inductive-inductive typer (QIITs), hvor relationen og typen defineres samtidig. Vi giver nogle eksempler på denne konstruktion, og viser eksempler på ækvivalence mellem de konstruktioner, dog kræver disse antagelsen af udvalgsaksiomet. Den tredje konstruktion er at lave en kvotient polynomisk functor, og tage grænsen af denne, for at få en QM-type, men igen kræver det udvalgsaksiomet, siden vi skal løfte operationerne fra kvotienten. complete

Acknowledgments

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

Lasse Letager Hansen,
Aarhus, June 9, 2020.

Contents

Chapter 1

Introduction

1.1 Overview

Inductive types in homotopy type theory (HoTT) / cubical type theory have been studied a lot, one body of work is `W`-types, which has been shown equal to inductive types. However there has not been much work done on the dual concept, namely coinductive types in HoTT defined as `M`-types. The goal of this thesis is to get an understanding of `M`-types and extend on the existing theory. 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 the problems with axiom of choice that arise from this construction.

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 3 we give some example constructions of `M`-types. In Chapter 4 we introduce quotiented `M`-types (`QM`-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 5 we discuss future research and conclude on the work done.

1.2 Expectations of reader

WRITE

1.3 Background Theory

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

We will be using **type theory** [15] 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 [14], where propositions are types, meaning proofs boils down to showing that there exists an element of some type representing a proposition. Specifically proofs of equality is 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) [13], 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 either take the smallest (largest) type for which these rules are true. A constructor for a type A is a function, that takes some arguments, and returns an element $a : A$, dually a destructor of A will take an element $a : A$ and return something. Inductive types are defined by giving rules for some base cases and then some 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 an element where succ is applied infinitely, namely $\infty = \texttt{succ}\ \infty$, which is not an element of $\mathbb{N}$. This type is also know as coN. 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 define an equivalence relation $\sim_{\mathbb{N}}$ on the natural numbers inductively, since the natural numbers are defined inductively. That is equivalence on the natural numbers follows the structure defined by the constructors

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

so 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 defined this way for inductive types implies equality, meaning if two elements are related then they are equal. Lets try and do a similar construcation for the coinductive type of streams. A stream is an infinite sequence of elements. Streams can be defined from the two destructors head (hd) and tail (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 for $\sim_{\mathtt{s}}$ `tream`. The univalent foundations we will be using is **Homotopy type theory (HoTT)** [16]. HoTT is an intensional dependent type theory (built on MLTT) with the univalence axiom and higher inductive types . In HoTT the idenitity types form path spaces, so proofs of identity are not just `refl` as is the case in MLTT. Types are seen as "spaces", and we think of $a : A$ as a being a point in the space A, similarly functions are regarded as continuous maps from one space to another [12]. 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)** [8], where the univalence axiom is not an axiom, but a statement that can be proven, meaning we constructivity of univalence axiom and application thereof [11]. 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.

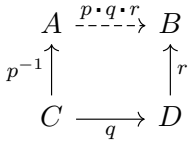

Figure 1.1: Composition square

algebra for functor ? `W`-types? should there be a decription of these?

If you are used to working in set theory, then working in HoTT will take some getting used to. 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 contractiable types, meaning inhabited types, where all element are equal to. Then there is (-1)-types which are mere propositions or hProp, where all elements of the type are equal, but types 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. 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 $n+1$ level. If we don't want to do proof relevant mathematics we can do propositional truncation (denoted $\|\cdot\|$), converting types to -1-types, meaning we ignore the difference in proofs by just look at whether a type is inhabited or not. However doing this we loose some of the reasoning power of HoTT. One of the tools we get using the full power of HoTT is **Higher order inductive types (HITs)**, which defines a type by point constructors and equality constructors. An example is the propositional truncation we just described, defined as a constructor taking any element to it its truncated version, and then defining equality between all propositional truncated elements

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

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

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

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

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

where $\mathtt{Y}$ is a type family over X [16, Section 3.8]. Using the axiom of choice is problematic in a constructive type theory, since they do not have a constructive interpretation. To maintain the computational aspects of HoTT and CTT, we try to avoid AC [16, Introduction].

We have formalized most of Chapter 2*, Chapter 3 and some of Chapter 4 in the proof assistant / programming languge Cubical Agda. A **proof assistant** helps with verifying proofs, while making the process of making proofs interactive. **Cubical Agda** [17] is an implementation of a cubical type theory (inspired by CCHM [9]) made by extending the proof assistant Agda. One of the main additions is the interval and path types. The **interval** $\mathbb{I}$ can be thought of as elements in the interval $[0, 1]$. When working with the interval, we can only access the left and right endpoint $\mathtt{i0}$ and $\mathtt{i1}$ 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 $\mathtt{A}\ \mathtt{i1}$. Using these tools we can make the univalence axiom compute and it eases the formalization, since proofs become more computational.

1.4 Notation

The following is the notation / fonts used to denote specific definitions / concepts We do a lot of casing on the natural numbers, so we will also introduce some notation, to make this more readable

Definition 1.4.1. Useful notation for casing on natural numbers.

$$\langle\!|\ x\ ,\ \mathtt{f}\ |\!\rangle = \lambda\, n, \begin{cases} x & n = 0 \\ \mathtt{f}\ m & n = m + 1 \end{cases} \tag{1.15}$$

We will also define some equalities $A \equiv B$ using an isomorphism, that is we define functions $\mathtt{fun} : A \to B$ and $\mathtt{inv} : B \to A$, and show there are right and left identities $\mathtt{rinv} : \mathtt{fun} \circ \mathtt{inv} \equiv \mathtt{id}$. We will introduce further notation as we go.

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

$\mathcal{U}_i$ 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), and use them to construct M-types and operations in and out on the M-types (Theorem 2.1.9) and showing that M-out is a final coalgebra (Theorem 2.1.11). We conclude the chapter by proving a coinduction principle for M-types (Theorem 2.2.2) [3].

2.1 Containers / Signatures

We start by introducing the notion of containers and polynomial functors for these containers.

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

Definition 2.1.2. A polynomial functor $\mathtt{P}_S$ 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 $\mathtt{P}$.

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

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

We can simplify this expression using $\mathbf{1} \to X \equiv X^{\mathbf{1}} \equiv X$. Furthermore $\mathbf{1}$ and X does not depend on A, so (2.3) is equivalent to

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

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 C and γ as $C\text{-}\gamma$. Coalgebra morphisms are defined as

$$\begin{aligned} \cdot \Rightarrow \cdot &: \mathtt{Coalg}_S \to \mathtt{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta &= \sum_{\mathtt{f}:C\to D} \delta \circ \mathtt{f} = \mathtt{Pf} \circ \gamma \end{aligned} \tag{2.6}$$

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 the M-type as $\mathtt{M}_{(A\,,\,\mathtt{B})}$ or $\mathtt{M}_S$ 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 $\mathtt{P}_{(A\,,\,\lambda\_,\,\mathbf{1})}\mathtt{M} = A \times \mathtt{M}_{(A\,,\,\lambda\_,\,\mathbf{1})}$, illustrated in Figure 2.1. We will show that $\mathtt{out}$ is an isomorphism with inverse $\mathtt{in} : \mathtt{P}_S\,\mathtt{M} \to \mathtt{M}$ later in this

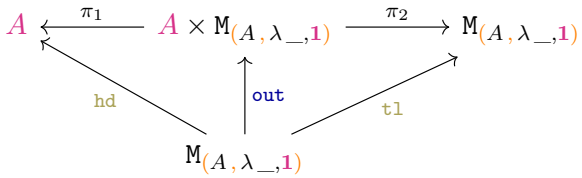

Figure 2.1: M-types of streams

section. We now have a semantic for the rules, we would expect for streams, if we let $\mathtt{cons} = \mathtt{in}$ and $\mathtt{stream}\ A = \mathtt{M}_{(A\,,\,\lambda\_,\,\mathbf{1})}$,

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

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

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

or more precisely $\mathtt{hd} = \pi_1 \circ \mathtt{out}$ and $\mathtt{tl} = \pi_2 \circ \mathtt{out}$.

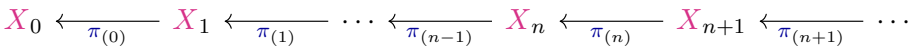

Figure 2.2: Chain of types / functions

2.1.1 Construction of M-type

We will show that M-types can be defined as the limit of a chain defined by repeatedly applying P to the unit type $\mathbf{1}$.

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* $y : \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv \ell_n\ x_n$ *the chain collapses as the equality* $\mathcal{L} \equiv X_0$.

Proof. We define this collapse 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, meaning $y \equiv \mathtt{inv}_{\mathcal{L}collapse}(\mathtt{fun}_{\mathcal{L}collapse}\ y)$, since both are equal to sections over $\mathbb{N}$. □

Lemma 2.1.8. *For all coalgebras* C-γ *define 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.*

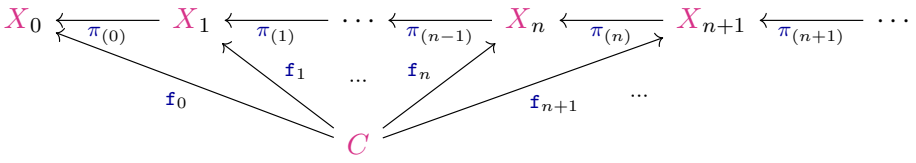

Figure 2.3: Cone

Proof. We define an isomorphism from $C \to \mathtt{M}_S$ to $\mathtt{Cone}_{C\text{-}\gamma}$

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

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

$$\mathtt{rinv}_{collapse}\ (\mathtt{u},\mathtt{q}) = \mathtt{refl}_{(\mathtt{u},\mathtt{q})} \tag{2.17}$$
$$\mathtt{linv}_{collapse}\ \mathtt{f} = \mathtt{refl}_{\mathtt{f}} \tag{2.18}$$

□

Theorem 2.1.9. *Given the 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*

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

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 ${\pi'}_{(n)} = \pi_{(n+1)}$. We can then define

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

We start by showing $\mathcal{L}unique$, by the ismorphism

$$\mathtt{fun}_{\mathcal{L}unique}\ (\mathtt{a},\mathtt{b}) = \langle\!|\ \star\ ,\ \mathtt{a}\ \langle\!|,|\!\rangle\ \mathtt{refl}_{\star}\ ,\ \mathtt{b}\ |\!\rangle \tag{2.23}$$
$$\mathtt{inv}_{\mathcal{L}unique}\ (\mathtt{a},\mathtt{b}) = \mathtt{a} \circ \mathtt{succ}\ ,\ \mathtt{b} \circ \mathtt{succ} \tag{2.24}$$
$$\mathtt{rinv}_{\mathcal{L}unique}\ (\mathtt{a},\mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{2.25}$$
$$\mathtt{linv}_{\mathcal{L}unique}\ (\mathtt{a},\mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{2.26}$$

By $\mathcal{L}unique$ we get that the limit of the chain is unique, so it just remains to show that, the limit of shifted chain is equal to $\mathtt{P}\,\mathcal{L}$, which is α 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 $\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n$ 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 chain of equalities for 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. Unfolding the definition we need to show that $\prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}\ (C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out})$, so we assume we are given some $C\text{-}\gamma : \mathtt{Coalg}_S$ to show contractability by $(C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out}) \equiv \mathbf{1}$. We get the following equalities,

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

since $\mathtt{in}$ and $\mathtt{out}$ are each others inverse, and $\mathtt{in}$ is part of an equality and therefore an embedding, which implies $(\mathtt{in} \circ \mathtt{a} \equiv \mathtt{in} \circ \mathtt{b}) \equiv (\mathtt{a} \equiv \mathtt{b})$. 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 define $\mathtt{e}$ to be the function from right to left for the equality in Lemma 2.1.8, we then get the equality

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

If we define 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

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

then we get the commuting square in Figure 2.4, which says $\psi\ (\mathtt{e}\ c) = \mathtt{e}\ (\phi\ c)$, so we can continue the simplification

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

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

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, that is for all a, b the equality $\mathtt{e}\ a \equiv \mathtt{e}\ b$ implies $a \equiv b$, using this and unfolding the definition of ϕ, we 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 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 from 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 becomes showing $\sum_{\star:\mathbf{1}} \mathbf{1} \equiv \mathbf{1}$, which is trivial. □

2.2 Coinduction Principle for M-types

We can now construct a coinduction principle for M-types given a (strong) bisimulation relation.

Definition 2.2.1. For all coalgebras $C\text{-}\gamma : \mathtt{Coalg}_S$, then a relation $\mathcal{R} : C \to C \to \mathcal{U}$ 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 P-coalgebra $\overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, that makes the diagram in Figure 2.5 commute ($\Longrightarrow$ represents P-coalgebra morphisms). That is $\gamma \circ \pi_1^{\overline{\mathcal{R}}} \equiv \mathtt{P}\ \pi_1^{\overline{\mathcal{R}}} \circ \alpha_{\mathbb{R}}$, 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 M-type, then for all x, y if $x\ \mathcal{R}\ y$ then they are equal $x \equiv y$.*

Proof. Given a relation $\mathcal{R}$, that is part of a bisimulation over a final 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 $!$, such that $\pi_1^{\overline{\mathcal{R}}} \equiv\ ! \equiv \pi_2^{\overline{\mathcal{R}}}$.

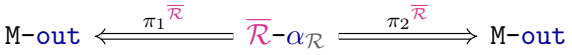

Figure 2.6: Bisimulation principle for final coalgebra

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

Chapter 3

Examples of M-types

In this section we show some examples of coinductive types that can be constructed as M-types, and show how to define the constructors/destructors of these types. We conclude this chapter with some general observation, and define some rules for how to construct M-types.

Is there anything else that is show for each M-type?

3.1 ITrees as M-types

Interaction trees (ITrees) [18] are used to model effectful behavior, where computations can interact with an external environment by events. ITrees are defined by the following constructors

$$\frac{r : R}{\texttt{Ret}\ r : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Ret}} \tag{3.1}$$

$$\frac{A : \mathcal{U} \quad a : \texttt{E}\ A \quad f : A \to \texttt{itree}\ \texttt{E}\ R}{\texttt{Vis}\ a\ f : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Vis}}. \tag{3.2}$$

$$\frac{t : \texttt{itree}\ \texttt{E}\ R}{\texttt{Tau}\ t : \texttt{itree}\ \texttt{E}\ R}\ \texttt{E}_{\texttt{Tau}}. \tag{3.3}$$

where R is the type for returned values, while $\texttt{E}$ is a dependent type for events representing external interactions. We will try and give some intuition on how to construct this type, by constructing types with two out of three of these constructors.

3.1.1 Delay Monad

We start by looking at ITrees without the $\texttt{Vis}$ constructor, this type is also know as the delay monad. It can be used to model delayed computations, either returning immediately given by the constructor $\texttt{now} = \texttt{Ret}$, or delayed some (possibly infinite) number of steps by the constructor $\texttt{later} = \texttt{Tau}$. We construct this type as an M-type.

Theorem 3.1.1. *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 and one that delays the computation, that is the

$$S = \left(R + \mathbf{1}\,,\ \begin{cases} \mathbf{0} & \texttt{inl}\ r \\ \mathbf{1} & \texttt{inr}\ \star \end{cases} \right) \tag{3.4}$$

which gives us the polynomial functor

$$\mathtt{P}_S\, X = \sum_{(x:R+\mathbf{1})} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ \mathbf{1} & x = \mathtt{inr}\ \star \end{cases} \to X, \tag{3.5}$$

since this definition is not dependent, we can simplify it to the following

$$\mathtt{P}_S\, X = R \times (\mathbf{0} \to X) + X. \tag{3.6}$$

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

$$\mathtt{P}_S\, X = R + X \tag{3.7}$$

meaning we get diagram in Figure 3.1, by applying $\mathtt{P}_S$ to $\mathtt{M}_S$ and using the functions `in` and `out` defined by the equality *shift*. We define the constructors `now` and `later` using `in` function for `M`-

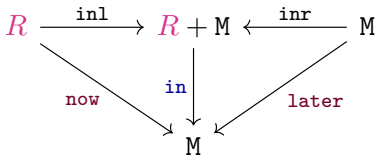

Figure 3.1: Delay monad

types, together with the injections `inl` and `inr`, such that the diagram in Figure 3.1 commutes. □

3.1.2 R-valued E-event Trees

Now lets look at the example, where we remove the `Tau` constructor. This gives us a type of tree, with leaves given by `Ret`, and nodes given by `Vis` branching based on some type A, for an event $a : \mathtt{E}\, A$.

Theorem 3.1.2. *We define R-valued* `E`*-event trees as an* `M`*-type*

Proof. We want a container, with a leaf constructor, that returning immediately with a value, and a node constructor, that branches on some event. We defined this by the container

$$S = \left(R + \sum_{(A:\mathcal{U})} \mathtt{E}\, A\,,\ \begin{cases} \mathbf{0} & \mathtt{inl}\ r \\ A & \mathtt{inr}\ (A, e) \end{cases} \right). \tag{3.8}$$

for which we get the polynomial functor

$$\mathtt{P}_S\, X = \sum_{(x:R+\sum_{(A:\mathcal{U})} \mathtt{E}\, A)} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ A & x = \mathtt{inr}\ (A, e) \end{cases} \to X, \tag{3.9}$$

we can again split the two case into a sum

$$\mathtt{P}_S\, X = (R \times (\mathbf{0} \to X)) + (\sum_{A:\mathcal{U}} \mathtt{E}\, A \times (A \to X)), \tag{3.10}$$

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

$$\mathtt{P}_S\, X = R + \sum_{A:\mathcal{U}} \mathtt{E}\, A \times (A \to X). \tag{3.11}$$

By applying this polynomial functor to the M-type, we get the diagram in Figure 3.2. We can again

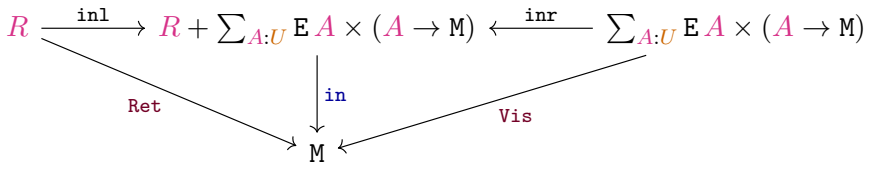

Figure 3.2: Tree Constructors

define `Ret` and `Vis` using the `in` function together with injections. We can see that `now` and `Ret` has the same structure, both in the diagram and in how they were defined in the container. □

3.1.3 ITrees

Get the correct equivalence for ITrees (Part of project description?)

Now we should have all the knowledge needed to make ITrees using M-types.

Theorem 3.1.3. *We define the type of ITrees as the M-type*

Proof. We combine the constructions of the delay monad and R-valued E-event Trees, into the container

$$S = \left(R + \mathbf{1} + \sum_{A:\mathcal{U}} (\mathtt{E}\, A)\,,\ \begin{cases} \mathbf{0} & \mathtt{inl}\ r \\ \mathbf{1} & \mathtt{inl}\ (\mathtt{inl}\ \star) \\ A & \mathtt{inr}\ (\mathtt{inr}\ (A, e)) \end{cases} \right). \tag{3.12}$$

For which the reduced polynomial functor is

$$\mathtt{P}_S\, X = R + X + \sum_{(A:\mathcal{U})} (\mathtt{E}\, A \times (A \to X)) \tag{3.13}$$

Applying this polynomial functor to the M-type, for the container, we get the diagram in Figure 3.3, from which the constructors of the ITrees type can be defined using `in` and injections. □

3.2 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 [2]. 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

$$\frac{a : A}{\mathtt{ret}\ a : T} \tag{3.14}$$

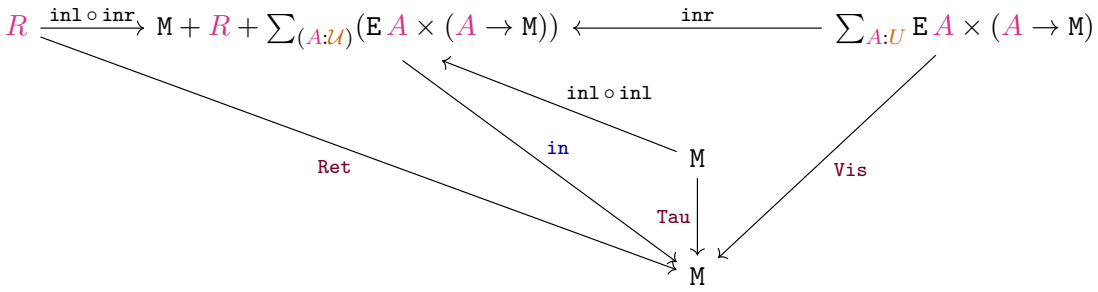

Figure 3.3: ITree constructors

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

$$\frac{a : T}{\mathtt{tl}\ a : T} \tag{3.15}$$

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

$$\left(A + \mathbf{1}\,, \begin{cases} \mathbf{0} & \mathtt{inl}\ a \\ \mathbf{1} & \mathtt{inr}\ \star \end{cases}\right) \tag{3.16}$$

which is the delay type. We define some more involved constructors, that build on the type itself

$$\frac{a : A \to T}{\mathtt{node}\ a : T} \tag{3.17}$$

has container $(\mathbf{1}\,, A)$. A type for the destructor

$$\frac{a : T}{\mathtt{hd}\ a : A} \tag{3.18}$$

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? 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

$$\left(A + C\,, \begin{cases} \mathtt{B}\ a & \mathtt{inl}\ a \\ \mathtt{D}\ c & \mathtt{inr}\ c \end{cases}\right) \tag{3.19}$$

whereas adding containers for two destructors is done by

$$(A \times C\,, \lambda \_,\, \lambda\,(a, c),\ \mathtt{B}\ a + \mathtt{D}\ c) \tag{3.20}$$

however combining destructors and constructors is not as simple. Any type T that can be defined using a (coinductive) record (except for higher inductive types), will also be definable as an M-type. 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

$$(F_1 \times F_2 \times \cdots \times F_n\,, \lambda \_,\, \mathbf{0}) \tag{3.21}$$

where each destructor $\mathtt{d_n} : T \to F_n$ for the field f_n will be defined as $\mathtt{d_n}\ t = \pi_n\ (\mathtt{out}\ t)$. 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 M-type for the container

$$\left(\sum_{(f_1 : F_1)} \sum_{(f_2 : \mathtt{F}_2)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \, , \lambda \_, \mathbf{0} \right) \tag{3.22}$$

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 may also coinductively construct an element of the record T, 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 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 by currying is equal to $f_1 \times (f_1 \to A) \to \mathtt{M}$, we can then define by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \left(\mathbf{1} \times \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \right) \, , \lambda \, (f_1, \star, f_3, \dots), \mathtt{F}_2 \right) \tag{3.23}$$

where $\mathtt{F}_2$ have been moved to the last part of the container, we can even leave out the "$\mathbf{1}\times$" from the container. 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 can get $\mathtt{F}_2 : \sum_{(x : f_1)} \mathtt{B}\, x \to \mathtt{M}$ which can be defined by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_\mathtt{n} \, , \lambda \, (f_1, f_3, \dots), \sum_{x : f_1} (\mathtt{B}\, x) \right) \tag{3.24}$$

so we would also expect that a type defined as a (coinductive) record is equal to the version defined as a M-type. However this has not been shown formally.

3.3 Wacky M-type

We end this chapter by exploring a wacky M-type, that utilizes the power of containers to the fullest. We define a wacky M-type by the following container

$$\left(\mathbb{N} + \mathbb{N}, \begin{cases} \mathbb{N} & \mathtt{inl}\ 0 \\ \mathbf{0} & \mathtt{inl}\ x \wedge x \text{ is odd} \\ \mathbf{0} & \mathtt{inr}\ x \wedge x \text{ is even} \\ \mathbf{1} & o.w. \end{cases} \right) \tag{3.25}$$

this container gives the M-type and constructors / destructors shown in Figure 3.4 The type can be interpreted as a stream of coproducts of natural numbers, that terminates whenever it is the left injection and even, or the right injection and odd, and whenever the left injection is zero, it splits in a branches indexed by the natural numbers. This show that we can define coinductive types, with rather complex structure, but a type being complex does not make it useful.

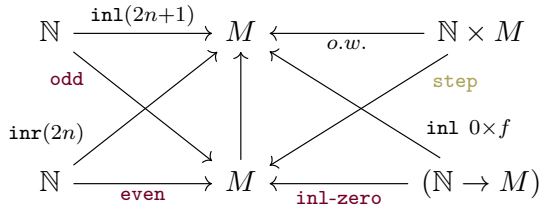

Figure 3.4: wacky M-type

Chapter 4

QM-types

In this chapter we will introduce rules for set truncated quotients, and show some ways of how to construct quotiented M-types which we call QM-types. We show examples of how you can be constructed a quotient inductive-inductive type (QIIT) that is equal to the QM-type assuming axiom of 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.

4.1 Quotienting and Constructors

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

Definition 4.1.1 (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

$$\begin{aligned}
&\mathtt{rec}\ [\,a\,] = \mathtt{f}\ a \\
&\mathtt{rec}\ (\mathtt{eq/}\ _\ _\ r\ i) = \mathtt{g}\ r\ i \\
&\mathtt{rec}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) = B_{set}\ (\mathtt{rec}\ a)\ (\mathtt{rec}\ b)\ (\mathtt{ap}\ \mathtt{rec}\ p)\ (\mathtt{ap}\ \mathtt{rec}\ q)\ i\ j
\end{aligned} \tag{4.1}$$

Definition 4.1.2 (Propositional eliminator for quotient)**.** Given a dependent type $\mathtt{P} : A/\mathtt{R} \to \mathcal{U}$, that is a proposition at all points $\mathtt{P}_{\mathtt{prop}} : \prod_{(x:A/\mathtt{R})} \mathtt{isProp}\ (\mathtt{P}\,x)$, then given a proof for the base case $\mathtt{f} : \prod_{(a:A)} \mathtt{P}\,[\,a\,]$ we get a function from $x : A/\mathtt{R}$ to $\mathtt{P}\,x$, defined as

$$\begin{aligned}
&\mathtt{elimProp}\ [\,a\,] = \mathtt{f}\ a \\
&\mathtt{elimProp}\ (\mathtt{eq/}\ a\ b\ i) = \mathtt{P}_{\mathtt{prop}}\ (\mathtt{elimProp}\ a)\ (\mathtt{elimProp}\ b)\ (\mathtt{eq/}\ a\ b)\ i \\
&\mathtt{elimProp}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) = \\
&\quad \mathtt{isSet{\to}isSetDep}\ (\mathtt{isProp{\to}isSet} \circ \mathtt{P}_{\mathtt{prop}}) \\
&\qquad (\mathtt{elimProp}\ a)\ (\mathtt{elimProp}\ b)\ (\mathtt{ap}\ \mathtt{elimProp}\ p)\ (\mathtt{ap}\ \mathtt{elimProp}\ q) \\
&\qquad (\mathtt{squash/}\ a\ b\ p\ q)\ i\ j
\end{aligned} \tag{4.2}$$

where $\mathtt{isSet{\to}isSetDep}$ takes a function $\prod_{a:A} \mathtt{isSet}\ (\mathtt{B}\,a)$ to the dependent version $\mathtt{isSetDep}\ A\ \mathtt{B}$. Using this eliminator we can to do propositional elimination of a quotient, by supplying the base case $\mathtt{P}\,[\,a\,]$.

Definition 4.1.3 (Eliminator for quotients)**.** Given a dependent type $\mathtt{P} : A/\mathbb{R} \to \mathcal{U}$ which is a set $\mathtt{P_{set}} : \prod_{(x:A/\mathbb{R})} \mathtt{isSet}\ (\mathtt{P}\, x)$, a proof for the base case $\mathtt{f} : \prod_{(a:A)} \mathtt{P}\, [\, a\,]$ and a proof for the equality case $\mathtt{f_{eq}} : \prod_{(a,b:A)} \prod_{(r:a\ \mathbb{R}\ b)} \mathtt{f}\ a \equiv_* \mathtt{f}\ b$ for the equality constructor $\mathtt{eq/}\ a\ b\ r$, we get a function $\mathtt{elim} : (x : A/\mathbb{R}) \to \mathtt{P}\, x$ as follows

$$\begin{aligned}
&\mathtt{elim}\ [\, a\,] = \mathtt{f}\ a \\
&\mathtt{elim}\ (\mathtt{eq/}\ a\ b\ r\ i) = \mathtt{f_{eq}}\ a\ b\ r\ i \\
&\mathtt{elim}\ (\mathtt{squash/}\ a\ b\ p\ q\ i\ j) = \\
&\quad \mathtt{isSet{\to}isSetDep}\ \mathtt{P_{set}}\ (\mathtt{elim}\ a)\ (\mathtt{elim}\ b)\ (\mathtt{ap\ elim}\ p)\ (\mathtt{ap\ elim}\ q)\ (\mathtt{squash/}\ a\ b\ p\ q)\ i\ j
\end{aligned} \tag{4.3}$$

which is the eliminator for quotients.

preserves the H-level

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.

4.2 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 construct the same type using quotient inductive-inductive types instead, and show these two construction gives equal types, assuming the axiom of choice. A quotient inductive-inductive type (QIIT) is a type that is defined at the same time as a relation over that type and set truncated. We believe, but do not show, that this equality between the two ways of constructing QM-types is general. However QIITs are more general than our QM-types, so we cannot expect every QIIT to have a corresponding QM-type.

4.2.1 Multiset

In this subsection we define infinite trees, where the order of subtrees does not matter also known as multisets [4][6][10], we construct the type of multisets as a QM-type and as a QIIT and show these are equal (Theorem 4.2.8). We start by defining the tree type we want to quotient.

Definition 4.2.1. We define A-valued $\mathbb{N}$-branching trees $\mathtt{T}\, A$ as the M-type defined by the container

$$\left(A + \mathbf{1}, \begin{cases} \mathbf{0} & \mathtt{inl}\ a \\ \mathbb{N} & \mathtt{inr}\ \star \end{cases} \right) \tag{4.4}$$

For which we get the constructors

$$\frac{a : A}{\mathtt{leaf}\ a : \mathtt{T}\, A} \tag{4.5}$$

$$\frac{\mathtt{f} : \mathbb{N} \to \mathtt{T}\, A}{\mathtt{node}\ \mathtt{f} : \mathtt{T}\, A} \tag{4.6}$$

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

$$\frac{\mathtt{f} : \mathbb{N} \to \mathtt{T}\, A \quad \mathtt{g} : \mathbb{N} \to \mathbb{N} \quad \mathtt{isIso\ g}}{\mathtt{node\ f} \equiv \mathtt{node}\ (\mathtt{f} \circ \mathtt{g})}\ \mathtt{perm} \tag{4.7}$$

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

$$\frac{x \equiv y}{\texttt{leaf}\ x \sim_{\mathtt{T}} \texttt{leaf}\ y}\ \sim_{\texttt{leaf}} \quad (4.8) \qquad \frac{\prod_{(n:\mathbb{N})} \mathtt{f}\ n \sim_{\mathtt{T}} \mathtt{g}\ n}{\texttt{node}\ \mathtt{f} \sim_{\mathtt{T}} \texttt{node}\ \mathtt{g}}\ \sim_{\texttt{node}} \quad (4.9)$$

$$\frac{\mathtt{f} : \mathbb{N} \to \mathtt{T}\ A \quad \mathtt{g} : \mathbb{N} \to \mathbb{N} \quad \mathtt{isIso}\ \mathtt{g}}{\texttt{node}\ \mathtt{f} \sim_{\mathtt{T}} \texttt{node}\ (\mathtt{f} \circ \mathtt{g})}\ \sim_{\texttt{perm}} \quad (4.10)$$

We have the problem that we cannot lift the definition of node to the quotiented type, without the use of the axiom of choice. To solve this problem we can define multisets as a QIIT denoted MS A, with the constructors leaf, node and 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 4.2.2. There is a function from T A to MS A

$$\begin{aligned} \mathtt{T{\to}MS}\ (\mathtt{leaf}_{\mathtt{T}}\ x) &= \mathtt{leaf}_{\mathtt{MS}}\ x \\ \mathtt{T{\to}MS}\ (\mathtt{node}_{\mathtt{T}}\ \mathtt{f}) &= \mathtt{node}_{\mathtt{MS}}\ (\mathtt{T{\to}MS} \circ \mathtt{f}) \end{aligned} \quad (4.11)$$

This function takes weakly bisimilar objects to equal once.

Lemma 4.2.3. *If x, y : T A are weakly bisimilar $p : x \sim_{\mathtt{T}} y$ then* T→MS $x \equiv$ T→MS y.

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

$$\begin{aligned} \mathtt{T{\to}MS\text{-}{\sim}{\to}{\equiv}}\ (\sim_{\texttt{leaf}}\ p) &= \mathtt{ap}\ \mathtt{leaf}_{\mathtt{MS}}\ p \\ \mathtt{T{\to}MS\text{-}{\sim}{\to}{\equiv}}\ (\sim_{\texttt{node}}\ \mathtt{k}) &= \mathtt{ap}\ \mathtt{node}_{\mathtt{MS}}\ (\mathtt{funExt}\ (\mathtt{T{\to}MS\text{-}{\sim}{\to}{\equiv}} \circ \mathtt{k})) \\ \mathtt{T{\to}MS\text{-}{\sim}{\to}{\equiv}}\ (\sim_{\texttt{perm}}\ \mathtt{f}\ \mathtt{g}\ e) &= \mathtt{perm}\ (\mathtt{T{\to}MS} \circ \mathtt{f})\ \mathtt{g}\ e \end{aligned} \quad (4.12)$$

□

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

Definition 4.2.4. There is a function T/∼→MS from T $A/\sim_{\mathtt{T}}$ to MS A, defined using the recursor for quotients defined in Definition 4.1.1, with f = T→MS and g = T→MS-∼→≡ and MS is a set by MS-isSet

Lemma 4.2.5. *Given an equality* T→MS $x \equiv$ T→MS y *then* $x \sim_{\mathtt{T}} y$.

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

Lemma 4.2.6. *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 4.1.2, with

$$\mathtt{P} = (\lambda\, x,\ \mathtt{T/{\sim}{\to}MS}\ x \equiv \mathtt{T/{\sim}{\to}MS}\ y \to x \equiv y) \quad (4.13)$$

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

$$\mathtt{elimProp}\ (\lambda\, a,\ \mathtt{elimProp}(\lambda\, b,\ \mathtt{eq/}\ a\ b \circ \mathtt{T{\to}MS\text{-}{\equiv}{\to}{\sim}}\ a\ b)\ y)\ x \tag{4.14}$$

where $\mathtt{T{\to}MS\text{-}{\equiv}{\to}{\sim}}$ is Lemma 4.2.5. □

Lemma 4.2.7. *The function* $\mathtt{T/{\sim}{\to}MS}$ *is surjective* $(\prod_{(b:\mathtt{MS})}, \|\Sigma_{(x:\mathtt{T})}\ \mathtt{T/{\sim}{\to}MS}\ x \equiv b\|)$*, assuming the axiom of choice.*

Proof. We only need to look at the point constructors of $\mathtt{MS}$. For the leaf case, we have the simple equality $\mathtt{T/{\sim}{\to}MS}\ [\mathtt{leaf_T}\ x\,] \equiv \mathtt{leaf_{MS}}\ x$. For the node case with $\mathtt{node\ f}$ then by induction, surjection of $[\,\cdot\,]$ and the axiom of choice, we get $\mathtt{g} : \mathbb{N} \to \mathtt{T}\ A$ such that $\prod_{(n:\mathbb{N})}\ \mathtt{T/{\sim}{\to}MS}\ [\,\mathtt{g}\ n\,] \equiv \mathtt{f}\ n$, making $\mathtt{T/{\sim}{\to}MS}\ [\mathtt{node\ g}\,] \equiv \mathtt{node\ f}$. □

Theorem 4.2.8. *There is an equality between the types* $\mathtt{T}\ A/{\sim_{\mathtt{T}}}$ *and* $\mathtt{MS}\ A$*, assuming the axoim of choice.*

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

4.2.2 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 choice. We do this proof by defining an intermediate representation of sequences, and then first showing the delay monad equal to this type (Subsection 4.2.2.1), and then showing that sequences quotiented by weak are equal to the partiality monad (Subsection 4.2.2.2) [5].

Definition 4.2.9 (Partiality Monad)**.** A simple example of a quotient inductive-inductive type is the partiality monad $(-)_\perp$ over a type R, defined by the constructors

$$\frac{}{R_\perp : \mathcal{U}} \tag{4.15}$$

$$\frac{}{\perp : R_\perp} \tag{4.16}$$

$$\frac{a : R}{\eta\ a : R_\perp} \tag{4.17}$$

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

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\bigsqcup\,(\mathtt{s},\mathtt{b}) : R_\perp} \tag{4.18}$$

$$\frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp\ p\ q : x \equiv y} \tag{4.19}$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{refl}} \tag{4.20}$$

$$\frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z}\ \sqsubseteq_{\mathtt{trans}} \tag{4.21}$$

$$\frac{x : R_\perp}{\perp \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{never}} \tag{4.22}$$

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \bigsqcup(\mathtt{s},\mathtt{b})} \tag{4.23}$$

$$\frac{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp x}{\bigsqcup(\mathtt{s},\mathtt{b}) \sqsubseteq_\perp x} \tag{4.24}$$

and finally set truncated

$$\frac{p, q : x \sqsubseteq_\perp y}{p \equiv q}\ (-)_\perp\text{-}\mathtt{isSet} \tag{4.25}$$

4.2.2.1 Delay monad to Sequences

We define sequences, and show they are equal to the delay monad (Theorem 4.2.18). We then extend this equality to an equality between the types quotient by weak bisimilarity (Theorem 4.2.23). We start by defining the type of sequences.

Definition 4.2.10. We define

$$\mathtt{Seq}_R = \sum_{(\mathtt{s}:\mathbb{N}\to R+1)} \mathtt{isMon\ s} \tag{4.26}$$

where

$$\mathtt{isMon\ s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \equiv \mathtt{inr}\ \star) \times (\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \star)) \tag{4.27}$$

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 also the special cases of already terminated, meaning only $\mathtt{inl}\ r$ and never teminating 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 $\mathtt{s}_n \downarrow_{R+1}$ to mean $\mathtt{s}_n \not\equiv \mathtt{inr}\ \star$. Thus we can write $\mathtt{isMon}$ as

$$\mathtt{isMon\ s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \uparrow_{R+1}) \times (\mathtt{s}_{n+1} \downarrow_{R+1})) \tag{4.28}$$

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

$$\mathtt{now}_{\mathtt{Seq}}\ r = (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl\ refl}) \tag{4.29}$$

$$\mathtt{never}_{\mathtt{Seq}} = (\lambda\_, \mathtt{inr}\ \star), (\lambda\_, \mathtt{inl\ refl}) \tag{4.30}$$

Definition 4.2.11. We can shift a sequence $(\mathtt{s}, \mathtt{q})$ by inserting an element (and an equality) (z_s, z_q) at $n = 0$,

$$\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ (z_s, z_q) = \begin{cases} z_s & n = 0 \\ \mathtt{s}_m & n = m+1 \end{cases}, \begin{cases} z_q & n = 0 \\ \mathtt{q}_m & n = m+1 \end{cases}, \tag{4.31}$$

Definition 4.2.12. We can unshift a sequence by removing the first element of the sequence

$$\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{s} \circ \mathtt{succ}, \mathtt{q} \circ \mathtt{succ}. \tag{4.32}$$

Lemma 4.2.13. *The function*

$$\mathtt{shift\text{-}unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{shift}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}))\ (\mathtt{s}_0, \mathtt{q}_0) \tag{4.33}$$

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 4.2.14. *The function*

$$\mathtt{unshift\text{-}shift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{unshift}\ (\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ \_) \tag{4.34}$$

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 4.2.15 (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` are equivalent to shifts, and `now` r is equivalent terminated sequence with value r. We do this by defining equivalence functions, and the left and right identities.

Definition 4.2.16. We define a function from `Delay` R to $\texttt{Seq}_R$

$$\begin{aligned}&\texttt{Delay→Seq}\ (\texttt{now}\ r) = \texttt{now}_{\texttt{Seq}}\ r\\&\texttt{Delay→Seq}\ (\texttt{later}\ x) =\\&\qquad \texttt{shift}\ (\texttt{Delay→Seq}\ x)\ \left(\texttt{inr}\ \star, \begin{cases}\texttt{inr}\ (\texttt{refl}, \texttt{inl≠inr}) & x = \texttt{now}\ \_\\ \texttt{inl}\ \texttt{refl} & x = \texttt{later}\ \_\end{cases}\right)\end{aligned} \tag{4.35}$$

Definition 4.2.17. We define function from $\texttt{Seq}_R$ to `Delay` R

$$\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q}) = \begin{cases}\texttt{now}\ r & \texttt{s}_0 = \texttt{inl}\ r\\ \texttt{later}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) & \texttt{s}_0 = \texttt{inr}\ \star\end{cases} \tag{4.36}$$

Theorem 4.2.18. *The type* `Delay` R *is equal to* $\texttt{Seq}_R$

Proof. We define right and left identity, saying that for any sequence $(\texttt{s}, \texttt{q})$, we get

$$\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q})) \equiv (\texttt{s}, \texttt{q}) \tag{4.37}$$

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

$$\texttt{now}_{\texttt{Seq}}\ r \equiv (\texttt{s}, \texttt{q}) \tag{4.38}$$

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

$$\texttt{shift}\ (\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q})))\ (\texttt{inr}\ \star, \_) \equiv (\texttt{s}, \texttt{q}) \tag{4.39}$$

by the induction hypothesis we get

$$\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) \equiv \texttt{unshift}\ (\texttt{s}, \texttt{q}) \tag{4.40}$$

since shift and unshift are inverse, we get the needed equality. For the left identity, we need to show that for any delay monad t we get

$$\texttt{Seq→Delay}\ (\texttt{Delay→Seq}\ t) \equiv t \tag{4.41}$$

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

$$\texttt{later}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{shift}\ (\texttt{Delay→Seq}\ x)))) \equiv \texttt{later}\ x \tag{4.42}$$

from unshift and shift being inverse and the induction hypothesis, we get the wanted equality. We have now defined a left and right identity function, so we get the wanted equality. □

Definition 4.2.19. We define the weak bisimulation relation for the delay monad

$$\frac{a \equiv b}{\texttt{now}\ a \sim_{\texttt{Delay}} \texttt{now}\ b}\ \sim_{\texttt{now}} \quad (4.43) \qquad \frac{x \sim_{\texttt{Delay}} y}{(\texttt{later}\ x) \sim_{\texttt{Delay}} y}\ \sim_{\texttt{later}_1} \quad (4.44)$$

$$\frac{x \sim_{\texttt{Delay}} y}{x \sim_{\texttt{Delay}} (\texttt{later}\ y)}\ \sim_{\texttt{later}_r} \quad (4.45) \qquad \frac{x \sim_{\texttt{Delay}} y}{\texttt{later}\ x \sim_{\texttt{Delay}} \texttt{later}\ y}\ \sim_{\texttt{later}} \quad (4.46)$$

We define weak bisimulation for sequences from the following definitions.

Definition 4.2.20 (Sequence Termination)**.** The following relations says that a sequence $(\mathtt{s}, \mathtt{q}) : \mathtt{Seq}_R$ terminates with a given value $r : R$,

$$(\mathtt{s}, \mathtt{q}) \downarrow_{\texttt{Seq}} r = \sum_{(n:\mathbb{N})} \mathtt{s}_n \downarrow_{\texttt{R+1}} r. \quad (4.47)$$

Definition 4.2.21 (Sequence Ordering)**.** We define an ordering of sequences as

$$(\mathtt{s}, \mathtt{q}) \sqsubseteq_{\texttt{Seq}} (\mathtt{t}, \mathtt{p}) = \prod_{(a:R)} (\|\mathtt{s} \downarrow_{\texttt{Seq}} a\| \to \|\mathtt{t} \downarrow_{\texttt{Seq}} a\|) \quad (4.48)$$

where $\|\cdot\|$ is propositional truncation.

Definition 4.2.22. We define weak bisimilarity for sequences as

$$(\mathtt{s}, \mathtt{q}) \sim_{\texttt{Seq}} (\mathtt{t}, \mathtt{p}) = (\mathtt{s}, \mathtt{q}) \sqsubseteq_{\texttt{Seq}} (\mathtt{t}, \mathtt{p}) \times (\mathtt{t}, \mathtt{p}) \sqsubseteq_{\texttt{Seq}} (\mathtt{s}, \mathtt{q}) \quad (4.49)$$

Theorem 4.2.23. *The types* $\mathtt{Delay}\ R/\sim_{\texttt{Delay}}$ *and* $\mathtt{Seq}_R/\sim_{\texttt{Seq}}$ *are equal.*

Proof. We show that transporting across the equality from Theorem 4.2.18, respects weak bisimilarity, starting 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 $p : a \equiv b$, since $\sim_{\texttt{Seq}}$ is reflexive, we get $\texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} \texttt{now}_{\texttt{Seq}}\ a$, which gives us $\texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} \texttt{now}_{\texttt{Seq}}\ b$ by transporting over p. If we have $\sim_{\texttt{later}_1}$ then by induction we get $\texttt{Delay}\to\texttt{Seq}\ x \sim_{\texttt{Seq}} \texttt{Delay}\to\texttt{Seq}\ y$, we can extend the equality by a shift, which interacts with $\texttt{Delay}\to\texttt{Seq}$ to add a later before the argument 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 $\sim_{\texttt{later}_r}$. To construct $\sim_{\texttt{later}}$ we add a later to both sites of the bisimilarity.

$$\begin{aligned}
&\sim_{\texttt{Delay}\to\texttt{Seq}}\ (\sim_{\texttt{now}}\ a\ b\ p) = \texttt{subst}\ (\lambda k,\ \texttt{now}_{\texttt{Seq}}\ a \sim_{\texttt{Seq}} k)\ p\ (\sim_{\texttt{refl}}\ a)\\
&\sim_{\texttt{Delay}\to\texttt{Seq}}\ (\sim_{\texttt{later}_1}\ x\ y\ r) = \sim_{\texttt{shift}}\ (\sim_{\texttt{Delay}\to\texttt{Seq}}\ r)\\
&\sim_{\texttt{Delay}\to\texttt{Seq}}\ (\sim_{\texttt{later}_r}\ x\ y\ r) = \sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{sym}}\ (\sim_{\texttt{Delay}\to\texttt{Seq}}\ r))\\
&\sim_{\texttt{Delay}\to\texttt{Seq}}\ (\sim_{\texttt{later}}\ x\ y\ r) = \sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{sym}}\ (\sim_{\texttt{shift}}\ (\sim_{\texttt{Delay}\to\texttt{Seq}}\ r)))
\end{aligned} \quad (4.50)$$

We show if $(\mathtt{s}, \mathtt{q}) \sim_{\texttt{Seq}} (\mathtt{t}, \mathtt{p})$ then $\texttt{Seq}\to\texttt{Delay}\ (\mathtt{s}, \mathtt{q}) \sim_{\texttt{Seq}} \texttt{Seq}\to\texttt{Delay}\ (\mathtt{t}, \mathtt{p})$ by case on $\mathtt{s}\ 0$ and $\mathtt{t}\ 0$. If both are terminated meaning $\mathtt{s}\ 0 = \texttt{inl}\ a$ and $\mathtt{t}\ 0 = \texttt{inl}\ b$ then there is a $p : a \equiv b$, meaning we get $\sim_{\texttt{now}}\ p$. If two sequences are weakly bisimilar, then shifting or unshifting either of them will

details?

not break the bisimilarity. We can therefore do the rest of the cases by induction.

text or proof term?

We know that if $\mathtt{s}\ 0 = \mathtt{inl}\ r$, then $(\mathtt{s}, \mathtt{q}) = \mathtt{now}_{\mathtt{Seq}}\ r$ and $(\mathtt{s}, \mathtt{q}) \downarrow_{\mathtt{Seq}} r$. If two sequences are weakly bisimilar and they terminate to different values, then the values are equal, we therefore get

$$\sim_{\mathtt{seq\text{-}val}} : \mathtt{now}_{\mathtt{Seq}}\ a \sim_{\mathtt{Seq}} \mathtt{now}_{\mathtt{Seq}}\ b \to a \equiv b \tag{4.51}$$

We can now define the function taking weakly bisimilar sequences to weakly bisimilar elements of the delay monad

$$\begin{aligned}
&\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\mathtt{now}_{\mathtt{Seq}}\ a)\ (\mathtt{now}_{\mathtt{Seq}}\ b)\ r = \sim_{\mathtt{now}}\ a\ b\ (\sim_{\mathtt{seq\text{-}val}}\ r)\\
&\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\mathtt{now}_{\mathtt{Seq}}\ a)\ (\mathtt{t}, \mathtt{p})\ r =\\
&\quad \sim_{\mathtt{later_r}}\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{s}, \mathtt{q}))\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{shift}\ (\mathtt{t}, \mathtt{p})\ _))\ (\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\sim_{\mathtt{shift_r}}\ r))\\
&\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\mathtt{s}, \mathtt{q})\ (\mathtt{now}_{\mathtt{Seq}}\ b)\ r =\\
&\quad \sim_{\mathtt{later_l}}\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ _))\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{t}, \mathtt{p}))\ (\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\sim_{\mathtt{shift_l}}\ r))\\
&\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\mathtt{s}, \mathtt{q})\ (\mathtt{p}, \mathtt{t})\ r =\\
&\quad \sim_{\mathtt{later}}\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ _))\ (\mathtt{Seq}\to\mathtt{Delay}\ (\mathtt{shift}\ (\mathtt{t}, \mathtt{p})\ _))\\
&\qquad (\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}\ (\sim_{\mathtt{shift}}\ r))
\end{aligned} \tag{4.52}$$

Do this

Now we show the right and left identity for $\sim_{\mathtt{Delay}\to\sim\mathtt{Seq}}$ and $\sim_{\mathtt{Seq}\to\sim\mathtt{Delay}}$

$$TODOz \tag{4.53}$$

We have shown that weakly bisimilar relations for the delay monad and sequences respects the equality given in Theorem 4.2.18, meaning that the quotiented types are also equal. □

There is a bit more work to show this, how much into depth should I go?

4.2.2.2 Sequence to Partiality Monad

In this section we will show that assuming the axiom of choice, we get an equivalence between sequences quotiented by weak bisimilarity and the partiality monad (Theorem 4.2.37). We start with some useful definitions.

Definition 4.2.24. There is a function from $R + \mathbf{1}$ to the partiality monad $R_\perp$

$$\begin{aligned}
\mathtt{Maybe}\to(-)_\perp\ (\mathtt{inl}\ r) &= \eta\ r\\
\mathtt{Maybe}\to(-)_\perp\ (\mathtt{inr}\ \star) &= \perp
\end{aligned} \tag{4.54}$$

Definition 4.2.25 (Maybe Ordering)**.** We define an ordering relation on $R + \mathbf{1}$ as

$$x \sqsubseteq_{\mathtt{R+1}} y = (x \equiv y) + ((x \downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}})) \tag{4.55}$$

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

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_{\mathtt{R+1}} \mathtt{s}_{n+1} \tag{4.56}$$

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

there exists non-monotone sequences, it just follows

Lemma 4.2.26. *The function* $\texttt{Maybe}{\to}(-)_\bot$ *is monotone, that is, if* $x \sqsubseteq_{\mathrm{R}+1} y$*, for some* x *and* y*, then* $(\texttt{Maybe}{\to}(-)_\bot\ x) \sqsubseteq_\bot (\texttt{Maybe}{\to}(-)_\bot\ y)$*.*

Proof. We do the proof by case.

$$\begin{aligned}
&\texttt{Maybe}{\to}(-)_\bot\texttt{-mono}\ (\texttt{inl}\ p) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\to}(-)_\bot\ x \sqsubseteq_\bot \texttt{Maybe}{\to}(-)_\bot\ a)\ p\ (\sqsubseteq_{\texttt{refl}}\ (\texttt{Maybe}{\to}(-)_\bot\ x)) \\
&\texttt{Maybe}{\to}(-)_\bot\texttt{-mono}\ (\texttt{inr}\ (p, _)) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\to}(-)_\bot\ a \sqsubseteq_\bot \texttt{Maybe}{\to}(-)_\bot\ y)\ p^{-1}\ (\sqsubseteq_{\texttt{never}}\ (\texttt{Maybe}{\to}(-)_\bot\ y))
\end{aligned} \tag{4.57}$$

□

Definition 4.2.27. There is a function taking a sequence to an increasing sequence of partiality monads

$$\texttt{Seq}{\to}\texttt{incSeq}\ (\texttt{s}, \texttt{q}) = \texttt{Maybe}{\to}(-)_\bot \circ \texttt{s}, \texttt{Maybe}{\to}(-)_\bot\texttt{-mono} \circ \texttt{q} \tag{4.58}$$

Definition 4.2.28. There is a function taking a sequence to the partiality monad

$$\begin{aligned}
&\texttt{Seq}{\to}(-)_\bot : \texttt{Seq}_R \to R_\bot \\
&\texttt{Seq}{\to}(-)_\bot\ (\texttt{g}, \texttt{q}) = \bigsqcup \circ \texttt{Seq}{\to}\texttt{incSeq}
\end{aligned} \tag{4.59}$$

Lemma 4.2.29. *The function* $\texttt{Seq}{\to}(-)_\bot$ *is monotone.*

$$\texttt{Seq}{\to}(-)_\bot\texttt{-mono} : (x\ y : \texttt{Seq}_R) \to x \sqsubseteq_{\texttt{seq}} y \to \texttt{Seq}{\to}(-)_\bot\ x \sqsubseteq_\bot \texttt{Seq}{\to}(-)_\bot\ y \tag{4.60}$$

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

should this be formalized entirely (There is alot of work here.. but not interesting.)

Lemma 4.2.30. *If two sequences* x, y *are weakly bisimular, then* $\texttt{Seq}{\to}(-)_\bot\ x \equiv \texttt{Seq}{\to}(-)_\bot\ y$

Proof. We know that the $\texttt{Seq}{\to}(-)_\bot$ is monotone, so we can use this monotonicity principle on $p : x \sqsubseteq_{\texttt{seq}} y$ and $q : y \sqsubseteq_{\texttt{seq}} x$ given by the bisimilarity, to get $\texttt{Seq}{\to}(-)_\bot\ x \sqsubseteq_\bot \texttt{Seq}{\to}(-)_\bot\ y$ and $\texttt{Seq}{\to}(-)_\bot\ y \sqsubseteq_\bot \texttt{Seq}{\to}(-)_\bot\ x$ by using $\alpha_\bot$ we get the equality $\texttt{Seq}{\to}(-)_\bot\ x \equiv \texttt{Seq}{\to}(-)_\bot\ y$. We denote this construction as the function $\texttt{Seq}{\to}(-)_\bot\texttt{-}{\approx}{\to}{\equiv}$ □

The recursor for quotients Definition 4.1.1 allows us to lift the function $\texttt{Seq}{\to}(-)_\bot$ to the quotient

Definition 4.2.31. We define a function $\texttt{Seq}/{\sim}{\to}(-)_\bot$ from $\texttt{Seq}_R$ to $R_\bot$

$$\texttt{Seq}/{\sim}{\to}(-)_\bot = \texttt{rec}\ \texttt{Seq}{\to}(-)_\bot\ (\texttt{Seq}{\to}(-)_\bot\texttt{-}{\approx}{\to}{\equiv})\ (-)_\bot\texttt{-isSet} \tag{4.61}$$

Lemma 4.2.32. *Given two sequences* s *and* t*, if* $\texttt{Seq}{\to}(-)_\bot\ s \equiv \texttt{Seq}{\to}(-)_\bot\ t$*, then* $s \sim_{\texttt{seq}} t$*.*

removed "isSet A", where is it used?

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

$$s \sim_{\texttt{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\texttt{seq}} r\| \to \|y \downarrow_{\texttt{seq}} r\|\right) \times \left(\prod_{(r:R)} \|y \downarrow_{\texttt{seq}} r\| \to \|x \downarrow_{\texttt{seq}} r\|\right) \tag{4.62}$$

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

Lemma 4.2.33. *The function* $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ *is injective.*

Proof. We use propositional elimination of quotients (see Definition 4.1.2) to show injectivity, meaning for all $x, y : \mathtt{Seq}_R/{\sim_{\mathtt{seq}}}$ we get $\mathtt{Seq}/{\sim}{\to}(-)_\perp\ x \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\ y \to x \equiv y$. We do propositional eliminating of x, followed by propositional elimination of y. We use Definition 4.1.2, with

$$\mathtt{P} = \mathtt{Seq}/{\sim}{\to}(-)_\perp\ x \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\ y \to x \equiv y \tag{4.63}$$

$$\mathtt{P}_{\mathtt{prop}} = \mathtt{isProp}\ \ (\lambda \_, \mathtt{squash}/\ x\ y) \tag{4.64}$$

where we index by x for the first elimination, and by y for the second with $x = [\,a\,]$. The proof is then completed by

$$\mathtt{elimProp}\ (\lambda\, a,\ \mathtt{elimProp}\ (\lambda\, b,\ (\mathtt{eq}/\ a\ b) \circ (\mathtt{Seq}{\to}(-)_\perp\text{-}{\equiv}{\to}{\sim}\ a\ b))\ y)\ x \tag{4.65}$$

where $\mathtt{Seq}{\to}(-)_\perp\text{-}{\equiv}{\to}{\sim}$ is (4.2.32), □

Lemma 4.2.34. *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 4.2.35 (Propositional eliminator for $(-)_\perp$)**.** Given a dependent type $\mathtt{P} : R_\perp \to \mathcal{U}$ that is a proposition at all points $\mathtt{P}_{\mathtt{prop}} : \prod_{(x:R_\perp)} \mathtt{isProp}\ (\mathtt{P}\ x)$, then 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 get a function from any $x : R_\perp$ to $\mathtt{P}\ x$

$$\begin{aligned}
&\mathtt{elimProp}_\perp\ \perp = P_\perp \\
&\mathtt{elimProp}_\perp\ (\eta\ a) = \mathtt{P}_\eta\ a \\
&\mathtt{elimProp}_\perp\ \sqcup\ (\mathtt{s}, \mathtt{q}) = P_\sqcup(\mathtt{s}, \mathtt{q})\ (\mathtt{elimProp}_\perp \circ s) \\
&\mathtt{elimProp}_\perp\ (\alpha\ p\ q\ i) = \\
&\quad \mathtt{isProp}{\to}\mathtt{isDepProp}\ \mathtt{P}_{\mathtt{prop}}\ (\mathtt{elimProp}_\perp\ x)\ (\mathtt{elimProp}_\perp\ y)\ (\alpha\ p\ q)\ i \\
&\mathtt{elimProp}_\perp\ (\perp\text{-}\mathtt{isSet}\ x\ y\ p\ q\ i\ j) = \\
&\quad \mathtt{isSet}{\to}\mathtt{isDepSet}\ (\mathtt{isProp}{\to}\mathtt{isSet} \circ \mathtt{P}_{\mathtt{prop}})\ (\mathtt{elimProp}_\perp\ x)\ (\mathtt{elimProp}_\perp\ y) \\
&\qquad (\mathtt{ap}\ \mathtt{elimProp}_\perp\ p)\ (\mathtt{ap}\ \mathtt{elimProp}_\perp\ q)\ (\perp\text{-}\mathtt{isSet}\ x\ y\ p\ q)\ i\ j
\end{aligned} \tag{4.66}$$

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

Lemma 4.2.36. *The function* $\mathtt{Seq}{\to}(-)_\perp$ *is surjective, assuming the axiom of choice.*

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

$$\mathtt{P} = \lambda y, \left\| \sum_{(x:\mathtt{Seq}_R)} \mathtt{Seq}{\to}\perp\ x \equiv y \right\| \tag{4.67}$$

and $\mathtt{P}_{\mathtt{prop}}$ follows from the propositional truncation. For $\bot$ and $\eta\ a$, we use $\mathtt{now}_{\mathtt{Seq}}\ a$ and $\mathtt{never}_{\mathtt{Seq}}$ respectively, and by (4.2.34) we get $\mathtt{P}\ \bot$ 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 gives us a pointwise equivalence

$$\prod_{(n:\mathbb{N})} \left\| \sum_{(\mathtt{f}:\mathbb{N}\to\mathtt{Seq}_R)} \mathtt{Seq}{\to}\bot\ (\mathtt{f}\ n) \equiv \mathtt{s}\ n) \right\| \tag{4.68}$$

which by the axoim of choice becomes

$$\left\| \prod_{(n:\mathbb{N})} \sum_{(\mathtt{f}:\mathbb{N}\to\mathtt{Seq}_R)} \mathtt{Seq}{\to}\bot\ (\mathtt{f}\ n) \equiv \mathtt{s}\ n) \right\| \tag{4.69}$$

Since we have a pointwise equivalence, we get an equivalence for the upper bound

$$\left\| \sum_{(\mathtt{x}:\mathtt{Seq}_R)} \mathtt{Seq}{\to}\bot\ x \equiv \bigsqcup(\mathtt{s}, \mathtt{q})) \right\| \tag{4.70}$$

Should I go into more depth about pointwise equivalence?

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* $\mathtt{Seq}/{\sim}{\to}(-)_\bot$ *is surjective assuming the axoim of choice.*

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

Theorem 4.2.37. *Sequences quotiented by weak bisimilarity is equivalent to the partiality monad* $Seq_R/{\sim_{\mathtt{seq}}} \equiv R_\bot$*, assuming the axiom of choice.*

Proof. The function $\mathtt{Seq}/{\sim}{\to}(-)_\bot$ is injective (Lemma 4.2.33) and surjective (Corollary 1) assuming the axoim of choice, meaning we get an equivalence, since we are working in hSets. □

4.3 How should QM-types be defined

We want to define what a QM-type means in general, we draw inspiration from QW-types [10], quotient containers [1] and quotient polynomial functors (QPF) [7]. From what we have seen as examples in the previous section, equality construction between Set Quotiented M-types and QIITs are given as

- A function $\mathtt{QM}{\to}\mathtt{QIIT}$ from QM to QIIT
- A proof $\mathtt{QM}{\to}\mathtt{QIIT}\text{-}{\sim}{\to}{\equiv}$ that $x \sim_{\mathtt{QM}} y$ implies $\mathtt{QM}{\to}\mathtt{QIIT}\ x \equiv \mathtt{QM}{\to}\mathtt{QIIT}\ y$
- Lifting $\mathtt{QM}{\to}\mathtt{QIIT}$ to $\mathtt{QM}/{\sim}{\to}\mathtt{QIIT}$ using the quotient recursor with $\mathtt{QM}{\to}\mathtt{QIIT}\text{-}{\sim}{\to}{\equiv}$ and $\mathtt{QIIT\text{-}isSet}$
- Showing injectivity by using propositional elimination of the quotient together with the inverse of $\mathtt{QM}{\to}\mathtt{QIIT}\text{-}{\sim}{\to}{\equiv}$, namely $\mathtt{QM}{\to}\mathtt{QIIT\text{-}injective}$ saying that $\mathtt{QM}{\to}\mathtt{QIIT}\ x \equiv \mathtt{QM}{\to}\mathtt{QIIT}\ y$ implies $x \sim_{\mathtt{QM}} y$.
- Lastly we show surjectivity by induction using the eliminator of QIIT and the axiom of choice and the surjectivity of $[\cdot]$ [16, 6.10.2].

4.3.1 Lifting Quotient Construction from Containers

An alternative to directly set quotienting the M-types we have defined, is to do the quotienting on the underlying container [1] / polynomial functor [7], and then do the fixed point construction we did for polynomial functors, but on the quotiented functor instead. We start by defining quotiented polynomial functors

Definition 4.3.1. Given a container $(A, \mathtt{B})$ and for all 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 can define a quotiented polynomial functor (QPF). We define it for types as

$$\mathtt{F}\, X = \sum_{a:A} ((\mathtt{B}\, a \to X)/\sim_a) \tag{4.71}$$

and for a function $\mathtt{f} : X \to Y$, we use the quotient eliminator from Definition 4.1.3, with $\mathtt{P} = \lambda \_, (\mathtt{B}\, a \to Y)/\sim_a$ which is a set $\mathtt{squash}/$, since we are using set truncated quotients. The base case $\mathtt{f} = \lambda \_, [\mathtt{f} \circ \mathtt{g}]$ and equality case $\mathtt{f}_{\mathtt{eq}} = \lambda\, \mathtt{x}\, \mathtt{y}\, r,\ \mathtt{eq}/\ (\mathtt{f} \circ \mathtt{x})\ (\mathtt{f} \circ \mathtt{y})\ (\sim_{\mathtt{ap}}\ \mathtt{f}\ r)$, where $\sim_{\mathtt{ap}}$ says that given $\mathtt{x} \sim_a \mathtt{y}$ and a function $\mathtt{f}$ then $\mathtt{f} \circ \mathtt{x} \sim_a \mathtt{g} \circ \mathtt{y}$. With this the definition for the quotient polynomial functor for functions is

$$\mathtt{F}\, \mathtt{f}\ (a, \mathtt{g}) = (a, \mathtt{elim}\ \mathtt{g}) \tag{4.72}$$

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 4.1 commute (as in [7]), then we can construct the final F-coalgebra, to get another notion of quotiented M-type. If $\mathtt{abs}$

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

Figure 4.1: Quotiented polynomial function

is surjective, then the square will commute, so we just have to define a relation that has the $\sim_{\mathtt{ap}}$ property, and a surjective function $\mathtt{abs}_X$ to construct a quotient M-type. The quotiented M-type has a surjective function $\mathtt{QM\text{-}intro} : \mathtt{M}_S \to \mathtt{M}_S/\sim$, given as $\mathtt{F}^\infty \circ \mathtt{abs}_{\mathtt{M}}$, where $\mathtt{F}^\infty$ is the function into the limit. As an example, lets try and construct the QM-type and the QPF for multisets, with this alternative approach

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

$$\mathtt{P}\, X = \sum_{a:A+\mathbf{1}} \begin{cases} \mathbf{0} \to X & a = \mathtt{inl}\ r \\ \mathbb{N} \to X & a = \mathtt{inr}\ \star \end{cases} \tag{4.73}$$

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

$$\frac{\mathtt{f}, \mathtt{h} : \mathbb{N} \to X \quad \mathtt{g} : \mathbb{N} \to \mathbb{N} \quad \mathtt{isIso}\ \mathtt{g} \quad \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}}{\mathtt{f} \sim_{\mathrm{PT}} \mathtt{h}} \tag{4.74}$$

we can see with this approach, we only need to define the non-trival equalities, meaning those that are not just reflexivity. This relations fulfills the $\sim_{\mathtt{ap}}$ property, since if $a = \mathtt{inl}\ r$ then it holds trivially, and if $a = \mathtt{inr}\ \star$ then we have $f \sim_{\mathtt{PT}} h$ 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 $p : \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}$, which is done by $\mathtt{ap}\ \mathtt{k}\ p$. We therefore have a QPF $\mathtt{F}$. Now we want to define $\mathtt{abs}$,

$$\begin{aligned} \mathtt{abs}\ (\mathtt{inl}\ r\ ,\ \lambda\,()) &= (\mathtt{inl}\ r\ ,\ [\lambda\,()]) \\ \mathtt{abs}\ (\mathtt{inr}\ \star\ ,\ \mathtt{f}) &= (\mathtt{inr}\ \star\ ,\ [\mathtt{f}]) \end{aligned} \tag{4.75}$$

and show that it is surjective, which follows directly from the surjectivity of $[\cdot]$. Taking the limit we get the commuting square in Figure 4.2.

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

Figure 4.2: 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.

The construction of $\mathtt{QW}$-types in [10] 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

Complete construction of QW inspired QM-types?

Chapter 5

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, and given some rules 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 shown that the type that arise from this construction is equal to the quotiented M-type, if we assume the axiom of choice. We 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.

conclude on the problem statement from the introduction

5.1 Future Work

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. One thing to do is should the construction of M-types equal to a 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.

cite indexed types

When looking at QM-types, we did not formalize the alternative construction of quotiented M-types as the limit of a quotiented polynomial functor, this should follow the same lines as the construction of M-types from polynomial functors, however it is not enough, since we need the axiom of choice for this construction. Formalizing this and comparing it to the directly quotiented version

of QM-types would be interesting. 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.

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.

Remove red text

We have not proof the equality between types defined as (coinducitve) records and M-types.

All the work done here, should generalize to indexed M-types, which would be nice to have formalized.

Building the weak bisimulation on the M-type as a M-type - Is this possible? Yes! Should it be included?
Define the weak bisimilarity relations as M-types, since these are also coindcutve types. However this seems to need something more general than "Index M-types"

We have removed the section of strong bisimulations, should it be re-added. Should the section about properties of M-types (stream) be re-added?

Building the Partiality Monad as a limit (Dialgebra?) - Is this possible?
Cofree Coalgebra / Dialgebra – Is this relevant?

Bibliography

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

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

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

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

[17] 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.

[18] 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

$\times \cdot \div \pm \surd \sim \approx \cong \neg \forall \quad \mathbb{N} \quad \infty \Sigma \quad \pi$ 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`.

A.1 Examples of M-types

A.1.1 Stream

src/stream.agda

```
{-# OPTIONS --cubical --guardedness #-}

module Cubical.Codata.M.AsLimit.stream where

open import Cubical.Data.Nat
open import Cubical.Data.Unit
open import Cubical.Data.Sigma

open import Cubical.Foundations.Prelude
open import Cubical.Foundations.Function using ( __ )
open import Cubical.Foundations.Isomorphism

open import Cubical.Codata.Stream
open import Cubical.Codata.M.AsLimit.M
open import Cubical.Codata.M.AsLimit.helper
open import Cubical.Codata.M.AsLimit.Container

--------------------------------------
-- Stream definitions using M.AsLimit --
--------------------------------------

stream-S : ∀ A → Container ℓ-zero
stream-S A = (A , λ( _ → Unit))

stream : ∀ (A : Type) → Type
stream A = M (stream-S A)

```

```
cons : ∀ {A} → A → stream A → stream A
cons x xs = in-fun (x , λ { tt → xs } )

hd : ∀ {A} → stream A → A
hd {A} s = out-fun s .fst

tl : ∀ {A} → stream A → stream A
tl {A} s = out-fun s .snd tt

------------------------------
-- Equality of stream types --
------------------------------

open Stream

stream-to-Stream : ∀ {A : Type} → stream A → Stream A
head (stream-to-Stream x) = hd x
tail (stream-to-Stream x) = stream-to-Stream (tl x)

private
  Stream-to-stream-func-x : ∀ {A : Type} (n : ℕ) → Stream A → W (stream-S
     A) n
  Stream-to-stream-func-x 0 x = lift tt
  Stream-to-stream-func-x (suc n) x = head x , λ _ → Stream-to-stream-func-x
     n (tail x)

  Stream-to-stream-funcπ- : ∀ {A : Type} (n : ℕ) (x : Stream A) →π
     (stream-S A) (Stream-to-stream-func-x (suc n) x) ≡
     Stream-to-stream-func-x n x
  Stream-to-stream-funcπ- 0 x = refl {x = lift tt}
  Stream-to-stream-funcπ- (suc n) x = λ i → head x , λ _ →
     Stream-to-stream-funcπ- n (tail x) i

Stream-to-stream : ∀ {A : Type} → Stream A → stream A
Stream-to-stream s = lift-to-M Stream-to-stream-func-x
   Stream-to-stream-funcπ- s

hd-to-head : ∀ {A : Type} (b : Stream A) → hd (Stream-to-stream b) ≡ head b
hd-to-head {A = A} b = refl

head-to-hd : ∀ {A : Type} (b : stream A) → head (stream-to-Stream b) ≡ hd b
head-to-hd {A = A} b = refl

tail-to-tl : ∀ {A : Type} (b : stream A) → tail (stream-to-Stream b) ≡
   stream-to-Stream (tl b)
tail-to-tl b = refl

private
  tail-eq-x : ∀ {A : Type} → (b : Stream A) (n : ℕ) → fst (tl
     (Stream-to-stream b)) n ≡ fst (Stream-to-stream (tail b)) n
  tail-eq-x {A = A} b n =
```

```
71      fst (tl (Stream-to-stream b)) n -- = transport λ( i → ( W (stream-S A)
           n)) (Stream-to-stream-func-x n (tail b))≡⟨
72         sym (transport-filler λ( i →  W (stream-S A) n)
              (Stream-to-stream-func-x n (tail b))) ⟩
73      fst (Stream-to-stream (tail b)) n
74
75    fgsa :∀
76       {A :  Type} → (b : Stream A) → (n : ℕ) →
77      PathP λ( i →π   (stream-S A) (tail-eq-x b (suc n) i) ≡ tail-eq-x b n i)
           (snd (tl (Stream-to-stream b)) n) (transport λ( i →π   (stream-S A)
           (tail-eq-x b (suc n) i) ≡ tail-eq-x b n i) (snd (tl (Stream-to-stream
           b)) n))
78    fgsa {A} b n = transport-filler λ( i →π   (stream-S A) (tail-eq-x b (suc n)
         i) ≡ tail-eq-x b n i) (snd (tl (Stream-to-stream b)) n)
79
80    postulate
81      tail-eqπ- :∀
82         {A :  Type} → (b : Stream A) → (n : ℕ) →
83        PathP λ( i →π   (stream-S A) (tail-eq-x b (suc n) i) ≡ tail-eq-x b n i)
84          (snd (tl (Stream-to-stream b)) n) -- transport ? refl
85          (Stream-to-stream-funcπ- n (tail b)) -- = refl
86    -- tail-eqπ- {A} b 0 = toPathP refl
87    -- tail-eqπ- {A} b (suc n) i =
88    --   compPathP-filler
89    --     {x = transport λ( i →π   (stream-S A) (tail-eq-x b (suc n) i) ≡
         tail-eq-x b n i) (snd (tl (Stream-to-stream b)) n)}
90    --     {y = (snd (tl (Stream-to-stream b)) n)}
91    --     {B = λ( i →π   (stream-S A) (tail-eq-x b (suc n) i) ≡ tail-eq-x b n
         i)}
92    --     {z = Stream-to-stream-funcπ- n (tail b)}
93    --     (symP (fgsa b n)) {!!} i {!!}
94
95  tl-to-tail :∀
96     {A :  Type} (b : Stream A)→
97     tl (Stream-to-stream b) ≡ Stream-to-stream (tail b)
98  tl-to-tail {A = A} b =Σ
99    PathP (funExt (tail-eq-x b) , λ i n j → tail-eqπ- b n i j)
100
101 nth : ∀ {A :  Type} →ℕ→   (b : Stream A) → A
102 nth 0 b = head b
103 nth (suc n) b = nth n (tail b)
104
105 stream-equality-iso-1 : ∀ {A :  Type} → (b : Stream A) → stream-to-Stream
       (Stream-to-stream b) ≡ b
106 stream-equality-iso-1 b = bisim-nat (stream-to-Stream (Stream-to-stream b)) b
       (helper b)
107   where
108     helper : ∀ {A :  Type} → (b : Stream A) → ((n : ℕ) → nth n
           (stream-to-Stream (Stream-to-stream b)) ≡ nth n b)
109     helper b 0 = head-to-hd (Stream-to-stream b)   hd-to-head b
110     helper b (suc n) =
```

```
      nth (suc n) (stream-to-Stream (Stream-to-stream b))≡⟨
         refl ⟩
      nth n (tail (stream-to-Stream (Stream-to-stream b)))≡⟨
         cong (nth n) (tail-to-tl (Stream-to-stream b)  cong
            stream-to-Stream (tl-to-tail b)) ⟩
      nth n (stream-to-Stream (Stream-to-stream (tail b)))≡⟨
         helper (tail b) n ⟩
      nth n (tail b)

    bisim-nat : ∀ {A :  Type} → (a b : Stream A) → ((n : ℕ) → nth n a ≡ nth
       n b) → a ≡ b
    bisim-nat a b nat-bisim = bisim (bisim-nat' a b nat-bisim)
      where
        open ≅EqualityBisimulation
        open ≈__

        bisim-nat' : ∀ {A :  Type} → (a b : Stream A) → ((n : ℕ) → nth n a
           ≡ nth n b) → a ≈ b≈
        head (bisim-nat' a b nat-bisim) = nat-bisim 0≈
        tail (bisim-nat' a b nat-bisim) = bisim-nat' (tail a) (tail b)
           (nat-bisim  suc)

private
  stream-equality-iso-2-x : ∀ {A :  Type} → (a : stream A) (n : ℕ) →
     Stream-to-stream-func-x n (stream-to-Stream a) ≡ fst a n
  stream-equality-iso-2-x a 0 = refl
  stream-equality-iso-2-x {A} a (suc n) =
    Stream-to-stream-func-x (suc n) (stream-to-Stream a)≡⟨
      refl ⟩
    (head (stream-to-Stream a) , λ( _ → Stream-to-stream-func-x n (tail
       (stream-to-Stream a))))≡⟨Σ
      PathP (head-to-hd a , funExt λ _ → cong (Stream-to-stream-func-x n)
         (tail-to-tl a)) ⟩
    (hd a , λ( _ → Stream-to-stream-func-x n (stream-to-Stream (tl a))))≡⟨Σ
      PathP (refl , funExt λ _ → stream-equality-iso-2-x (tl a) n) ⟩
    (hd a , λ( _ → fst (tl a) n))≡⟨Σ
      PathP (refl , refl) ⟩
    (out-fun a .fst , λ( _ → fst (tl a) n))≡⟨Σ
      PathP (refl , refl) ⟩
    (fst (fst a (suc 0)) , λ( _ → fst (tl a) n))≡⟨Σ
      PathP (temp n , refl) ⟩
    (fst (fst a (suc n)) , λ( _ → fst (tl a) n))≡⟨Σ
      PathP (refl , funExt λ _ → temp') ⟩
    fst (fst a (suc n)) , snd (fst a (suc n))≡⟨
      refl ⟩
    fst a (suc n)
    where
      temp : ∀ (n : ℕ) → fst (fst a (suc 0)) ≡ fst (fst a (suc n))
      temp 0 = refl
      temp (suc n) = temp n  cong fst (sym (snd a (suc n)))

```

```
155       temp' : fst (tl a) n ≡ snd (fst a (suc n)) tt
156       temp' = sym (transport-filler λ( i → W (stream-S A) n) (a .fst (suc n)
              .snd tt))
157
158   postulate
159     stream-equality-isoπ-2- :∀
160        {A : Type} → (a : stream A) (n : ℕ)→
161        PathP λ( i →π  (stream-S A) (funExt (stream-equality-iso-2-x a) i
              (suc n)) ≡ funExt (stream-equality-iso-2-x a) i n)
162         (Stream-to-stream-funcπ- n (stream-to-Stream a))
163         (snd a n)
164
165 stream-equality-iso-2 : ∀ {A : Type} → (a : stream A) → Stream-to-stream
       (stream-to-Stream a) ≡ a
166 stream-equality-iso-2 a =
167   Stream-to-stream (stream-to-Stream a)≡⟨
168      refl ⟩
169   lift-to-M Stream-to-stream-func-x Stream-to-stream-funcπ- (stream-to-Stream
       a)≡⟨
170      refl ⟩λ
171   ( n → Stream-to-stream-func-x n (stream-to-Stream a)) ,λ
172   ( n → Stream-to-stream-funcπ- n (stream-to-Stream a))≡⟨Σ
173      PathP (funExt (stream-equality-iso-2-x a) , λ i n j →
           stream-equality-isoπ-2- a n i j) ⟩
174   a
175
176 stream-equality : ∀ {A : Type} → stream A ≡ Stream A
177 stream-equality = isoToPath (iso stream-to-Stream Stream-to-stream
       stream-equality-iso-1 stream-equality-iso-2)
178
179 ------------------------------------------------------
180 -- Defining stream examples by transporting records --
181 ------------------------------------------------------
182
183 -- Zeros defined as a record type
184 Zeros : Stream ℕ
185 head Zeros = 0
186 tail Zeros = Zeros
187
188 zeros-transported : stream ℕ
189 zeros-transported = transport (sym stream-equality) Zeros
190
191 -- It is now easy to show computation properties for the M-types:
192 hd-zeros-transported : hd zeros-transported ≡ 0
193 hd-zeros-transported = hd-to-head (transportRefl Zeros i0)
194
195 tl-zeros-transported : tl zeros-transported ≡ zeros-transported
196 tl-zeros-transported = tl-to-tail (transportRefl Zeros i0)
197
198 -- zeros defined for stream M-type
199 zeros : stream ℕ
```

```
zeros = lift-direct-M zeros-x zerosπ-
  where
    zeros-x : (n : ℕ) → W (stream-S ℕ) n
    zeros-x 0 = lift tt
    zeros-x (suc n) = 0 , λ( _ → zeros-x n)

    zerosπ- : (n : ℕ) →π  (stream-S ℕ) (zeros-x (suc n)) ≡ zeros-x n
    zerosπ- 0 i = lift tt
    zerosπ- (suc n) i = 0 , λ( _ → zerosπ- n i)
```