M-types and Coinduction in HoTT and Cubical Type Theory

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science
May 25, 2020
Advisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

in English. . .

Resumé

in Danish...

Acknowledgments

c

Lasse Letager Hansen,
Aarhus, May 25, 2020.

Contents

Chapter 1

Introduction

In the second chapter we introduce/summarize some background theory. In the third chapter we construct `M`-types from containers, and define a coinduction principle for the M-types. In the fourth chapter we introduce quotiented `M`-types (QM-types), and show equalities between these and quotient inductive inductive types (QIITs). We then go through various applications of the theory developed in the first couple chapters. Finally we conclude by discussion future research and improvements.

motivate and explain the problem to be addressed

get your bibtex entries from `https://dblp.org/`

Most of the work is formalized in the Cubical Agda proof assistant. We end this introduction, with a description of the notation used throughout this thesis. We use the following notation / font:

- Universe $\mathcal{U}_i$ or $\mathcal{U}$
- Type $A : \mathcal{U}$
- A type former or dependent type $\mathtt{B} : A \to \mathcal{U}$
- A term $x : A$ or for constants $\mathtt{c} : A$
- A function $\mathtt{f} : A \to C$
- A constructor $\mathtt{f} : A \to C$
- A destructor $\mathtt{f} : A \to C$
- A path $p : A \equiv C$, heterogeneous paths are denotes $\equiv_p$ or if the path is clear from context $\equiv_*$.
- A relation $\mathtt{R} : A \to A \to \mathcal{U}$ with notation $x \ \mathtt{R} \ y$.
- The unit type is $\mathbf{1}$ while the empty type is $\mathbf{0}$.
- A functor `P`
- A container is denoted as S or $(A \, , \mathtt{B})$

better description, not always a function

better description, not always a function

- A coalgebra C-γ
- We denote the function giving the first and second projection of a dependent pair by π_1 and π_2.

Furthermore we define some useful notation

Definition 1.0.1.

$$\langle\!| \; x \; , \; \mathtt{f} \; |\!\rangle = \lambda\, n, \begin{cases} x & n = 0 \\ \mathtt{f}\ m & n = m + 1 \end{cases} \tag{1.1}$$

Chapter 2

Background Theory

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

2.1 Proof assistants

2.2 Set Theory

Should this be included

2.3 Type Theory

Type theory is the computational angle on a basis for mathematics. In type theory every term x is an element of some type A, written $x : A$. The idea in type theory is that propositions are types, so proofs boils down to showing that there exists an element of some type representing a proposition. Specifically proofs of equality becomes construction of an element of an equality type.

2.3.1 Martin Löf Type Theory / Intuitionistic type Theory (MLTT/ITT)

Per Matin-Löf designed the type theory on the principles of mathmatical constructivism, where any existance proof must contain a witness. Meaning a proof that there exists an element, such that something is true, can be converted into an algorithm that finds that element. The type theory is built from the three finite types **0**, **1** and **2**, and type constructors Σ, π and $=$. There is only a single way to make terms of $=$-type, and that is $\texttt{refl} : \prod_{a:A}(a = a)$. .

inductive types, universes and judgements

2.3.1.1 Intensional Type Theory

extensional vs intensional

2.4 The Univalence Axiom

The map

$$\texttt{id}_{\mathcal{U}}\ A\ B \to \texttt{Equiv}\ A\ B \tag{2.1}$$

find citation, currently `https://en.`

is an equivalence, simplifying a bit, identity is equivalent to equivalence

$$(A = B) \simeq (A \simeq B) \tag{2.2}$$

Or said in another way equivalent types are identical.

2.5 Homotopy Type Theory (HoTT) / Univalent Foundations (UF)

Homotopy type theory is an intensional dependent type theory (built on MLTT) with the univalence axiom and higher inductive types. Homotopy type theory is constructive by default, but can be made classical.

is a type theory where the idenitity types form path spaces, so proofs of identity are not just `refl`, as for MLTT. Types are seen as "spaces" or (higher groupoids), 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. [5]

2.5.1 Judgmental Equality / Definitional Equality

This means equal by definition, denoted $a \equiv b$. It does not make sense to assume definitional equality (inside the theory). The equality is algorithmically decidable (meta-theoretic algorithm, not internal).

2.5.2 Propositional Equality

If the equality $a =_A b$ is inhabited, then we get propositional equality.

2.5.3 External (of theory)

2.5.4 Internal (of theory)

2.5.5 Constructivity

From every proof we can extract an algorithm, that produces an element of the proposition.

2.5.5.1 Proof relevant mathematics

2.5.6 Homotopy n-types

The 0-types are the hSets, (-1)-types are mere propositions or hProp. So all elements of a type that is an hProp, will be equal. All equalities between elements of types that are hSets, are equal. For 1-types (1-groupoids) we get equalities of equalities are equal, and then so on for homotopy n-types.

2.5.7 The HoTT Book

2.6 Higher order inductive types (HITs)

2.6.1 Propositional truncation and Set Truncated Quotients

2.7 Cubical Type Theory (CTT)

Cubical type theory is a version of homotopy type theory where univalence actually computes, meaning it is not just an axiom, but can actually be proven. [4]

2.7.1 UIP

2.7.2 Cubical Agda

Cubical Agda is an implementation of a cubical type theory by extending the proof assistant Agda. One of the main additions is the interval and path types. The interval can be thought of as elements in $[0, 1]$. When working with the interval, we can only access the left and right endpoint `i0` and `i1` or some unspecified point in the middle i, modeling the intuition of a continuous interval. Cubical agda also generalizes transporting, given a type line `A` : $\mathbb{I} \to \mathcal{U}$, and the endpoint `A i0` you get a line from `A i0` to `A i1` [9]

Axioms of cubical Agda

The theory of cubical Agda is a Cartesian closed category, meaning get exponentials.

Something about the interval type!!

2.7.2.1 Path type

We add a type $\mathbb{I}$, which is defined to be a free de Morgan algebra on a discrete infinite set of names . The elements of $\mathbb{I}$ can be described by the grammar

cite: https://arxi

$$r, s ::= 0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s \tag{2.3}$$

The set $\mathbb{I}$ has decidable equality. The elements in $\mathbb{I}$ can be thought as formal representations of elements in the unit interval $[0, 1]$. There is a special substitution with `i0` and `i1` being the endpoints of $[0, 1]$.

2.8 Coinduction

Coinduction is the dual concept (in a categorical manner) of induction. The induction principle is an equivalence principle for congruent elements in an initial algebra.

2.9 W-types

2.10 Problems with / Getting around using AC and LEM

These axioms does not have a computational interpretation, so to maintain the computational aspects of HoTT and CTT, we try to not use these axioms. [8, Introduction]

2.11 Universes ??

Chapter 3

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 3.1.8). We conclude the chapter by defining a coinduction principle for M-types.

Some formalization of coalgebras is missing?

3.1 Containers / Signatures

Definition 3.1.1. A Container (or signature) is a dependent pair $S = (A, \mathtt{B})$ for the types $A : \mathcal{U}$ and $\mathtt{B} : A \to \mathcal{U}$.

Definition 3.1.2. A polynomial functor over the container $S = (A, \mathtt{B})$ is defined for types as

$$\begin{aligned} &\mathtt{P}_S : \mathcal{U} \to \mathcal{U} \\ &\mathtt{P}_S(X) = \sum_{a:A} \mathtt{B}(a) \to X \end{aligned} \tag{3.1}$$

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

$$\begin{aligned} &\mathtt{P}_S\mathtt{f} : \mathtt{P}_S X \to \mathtt{P}_S Y \\ &\mathtt{P}_S\mathtt{f}(a, \mathtt{g}) = (a, \mathtt{f} \circ \mathtt{g}). \end{aligned} \tag{3.2}$$

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

$$\mathtt{P}_S(X) = \sum_{a:A} \mathbf{1} \to X. \tag{3.3}$$

Since we are working in a logic with exponentials, we get $\mathbf{1} \to X \equiv X^{\mathbf{1}} \equiv X$. Furthermore $\mathbf{1}$ and X does not depend on A, so (3.3) is equivalent to

$$\mathtt{P}_S(X) = A \times X. \tag{3.4}$$

We now construct the $\mathtt{P}_S$-coalgebra for a polynomial functor $\mathtt{P}_S$.

Definition 3.1.3. A $\mathtt{P}_S$-coalgebra is defined as

$$\mathtt{Coalg}_S = \sum_{C:\mathcal{U}} C \to \mathtt{P}_S C. \tag{3.5}$$

We denote a $\mathtt{P}_S$-coalgebra given by C and γ as $C\text{-}\gamma$. Coalgebra morphisms are defined as

$$\begin{aligned} \cdot \Rightarrow \cdot &: \mathtt{Coalg}_S \to \mathtt{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta &= \sum_{\mathtt{f}:C\to D} \delta \circ \mathtt{f} = \mathtt{Pf} \circ \gamma \end{aligned} \tag{3.6}$$

We can now define $\mathtt{M}$-types.

Definition 3.1.4. Given a container S, we define $\mathtt{M}$-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

$$\mathtt{Final}_S := \sum_{(X\text{-}\rho:\mathtt{Coalg}_S)} \prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}\ (C\text{-}\gamma \Rightarrow X\text{-}\rho). \tag{3.7}$$

That is $\prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}(C\text{-}\gamma \Rightarrow \mathtt{M}_S\text{-}\mathtt{out})$. We denote the $\mathtt{M}$-type as $\mathtt{M}_{(A\,,\mathtt{B})}$ or $\mathtt{M}_S$ or just $\mathtt{M}$ when the Container is clear from the context.

Continuing our example we now construct streams as an $\mathtt{M}$-type.

Example 2. We define streams over the type A as the $\mathtt{M}$-type over the container $(A\,, \lambda\_, \mathbf{1})$. If we apply the polynomial functor to the $\mathtt{M}$-type, then we get $\mathtt{P}_{(A\,,\lambda\_,\mathbf{1})}\mathtt{M} = A \times \mathtt{M}_{(A\,,\lambda\_,\mathbf{1})}$, illustrated in Figure 3.1. We will that $\mathtt{out}$ is an isomorphism with inverse $\mathtt{in} : \mathtt{P}_S(\mathtt{M}) \to \mathtt{M}$ later in this section.

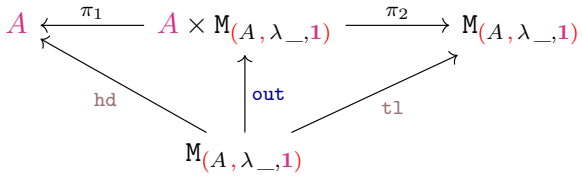

Figure 3.1: $\mathtt{M}$-types of streams

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

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{hd}\ s : A}\ \mathtt{E}_{\mathtt{hd}} \tag{3.8}$$

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{tl}\ s : \mathtt{stream}\ A}\ \mathtt{E}_{\mathtt{tl}} \tag{3.9}$$

$$\frac{A : \mathcal{U} \quad x : A \quad xs : \mathtt{stream}\ A}{\mathtt{cons}\ x\ xs : \mathtt{stream}\ A}\ \mathtt{I}_{\mathtt{cons}} \tag{3.10}$$

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

Definition 3.1.5. 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 3.1.5.

Lemma 3.1.6. *For all coalgebras $C\text{-}\gamma$ for the container S, we get $C \to \mathtt{M}_S \equiv \mathtt{Cone}_{C\text{-}\gamma}$, where $\mathtt{Cone} = \sum_{(\mathtt{f}:\prod_{(n:\mathbb{N})} C\to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ (\mathtt{f}_{(n+1)}) \equiv f_n$ illustrated in Figure 3.1.6.*

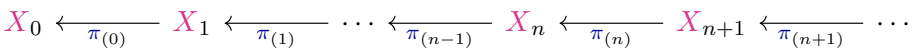

Figure 3.2: Chain of types / functions

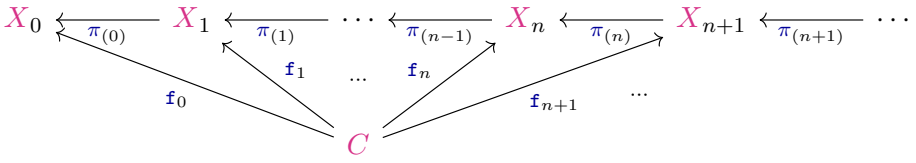

Figure 3.3: Cone

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

$$\mathtt{fun}_{collapse}\ \mathtt{f} = (\lambda\, n\, z, \pi_1\ (\mathtt{f}\ z)\ n), (\lambda\, n\ i\ a,\ \pi_2\ (\mathtt{f}\ a)\ n\ i) \tag{3.11}$$

$$\mathtt{inv}_{collapse}\ (\mathtt{u}, \mathtt{q})\ z = (\lambda\, n,\ \mathtt{u}\ n\ z), (\lambda\, n\, i,\ \mathtt{q}\ n\ i\ z) \tag{3.12}$$

$$\mathtt{rinv}_{collapse}\ (\mathtt{u}, \mathtt{q}) = \mathtt{refl}_{(\mathtt{u},\mathtt{q})} \tag{3.13}$$

$$\mathtt{linv}_{collapse}\ \mathtt{f} = \mathtt{refl}_{\mathtt{f}} \tag{3.14}$$

□

Lemma 3.1.7. *Given* $\ell : \prod_{(n:\mathbb{N})}(X_n \to X_{n+1})$ *and* $y : \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv l_n\ x_n$ *the chain collapses as the equality* $\mathcal{L} \equiv X_0$.

Proof. We define this collapse by the isomorphism

$$\mathtt{fun}_{\mathcal{L}collapse}\ (x, r) = x_0 \tag{3.15}$$

$$\mathtt{inv}_{\mathcal{L}collapse}\ x_0 = (\lambda\, n,\ \ell^n\ x_0)\ ,\ (\lambda\, n,\ \mathtt{refl}_{(\ell^{(n+1)}\ x_0)}) \tag{3.16}$$

$$\mathtt{rinv}\ x_0 = \mathtt{refl}_{x_0} \tag{3.17}$$

where $\ell^n = \ell_n \circ \ell_{n-1} \circ \cdots \circ \ell_1 \circ \ell_0$. To define $\mathtt{linv}\ (x, r)$, 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 $\mathbb{N}$. □

We can now define the construction of `in` and `out`.

Theorem 3.1.8. *Given the container* $(A, \mathtt{B})$ *we define the equality*

$$shift : \mathcal{L} \equiv \mathtt{P}\mathcal{L} \tag{3.18}$$

where $\mathtt{P}\mathcal{L}$ *is the limit of a shifted sequence. Then*

$$\mathtt{in} = \mathtt{transport}\ shift \tag{3.19}$$

$$\mathtt{out} = \mathtt{transport}\ (shift^{-1}). \tag{3.20}$$

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^{\mathtt{P}} \equiv \mathtt{P}\mathcal{L} \tag{3.21}$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^{\mathtt{P}} \tag{3.22}$$

We define $\mathcal{L}unique$ by the ismorphism

$$\mathtt{fun}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \langle\!|\ \star\ ,\ \mathtt{a}\ \langle\!|, |\!\rangle\ \mathtt{refl}_{\star}\ ,\ \mathtt{b}\ |\!\rangle \tag{3.23}$$

$$\mathtt{inv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{a} \circ \mathtt{succ}\ ,\ \mathtt{b} \circ \mathtt{succ} \tag{3.24}$$

$$\mathtt{rinv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{3.25}$$

$$\mathtt{linv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) = \mathtt{refl}_{(\mathtt{a},\mathtt{b})} \tag{3.26}$$

The definition of α is then,

$$\mathcal{L}^{\mathtt{P}} \equiv \sum_{(x:\prod_{(n:\mathbb{N})} \sum_{(a:A)} \mathtt{B}\, a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n+1)}\ x_{n+1} \equiv x_n \tag{3.27}$$

$$\equiv \sum_{(x:\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\,(\pi_1\, x)_n \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv_* \mathtt{u}_n \tag{3.28}$$

$$\equiv \sum_{(a:A)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\, a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv \mathtt{u}_n \tag{3.29}$$

$$\equiv \sum_{(a:A)} \mathtt{B}\ a \to \mathcal{L} \tag{3.30}$$

$$\equiv \mathtt{P}\mathcal{L} \tag{3.31}$$

To collapse $\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n$ to A between (3.28) and (3.29) we use Lemma 3.1.7 . We use Lemma 3.1.6 for the equality between (3.29) and (3.30). The rest of the equalities are trivial. The definition of $shift$ is

$$shift = \alpha^{-1} \cdot \mathcal{L}unique. \tag{3.32}$$

We furthermore get the definitions $\mathtt{in} = \mathtt{transport}\ shift$ and $\mathtt{out} = \mathtt{transport}\ (shift^{-1})$, since $\mathtt{in}$ and $\mathtt{out}$ are part of an equality relation $shift$, they are both surjective and embeddings. □

is surjectivity and embedding important here? Describe this where relevant instead!

3.2 Coinduction Principle for M-types

We can now construct a coinduction principle given a bisimulation relation.

Definition 3.2.1. For all coalgebras C-$\gamma : \mathtt{Coalg}_S$, given a relation $\mathcal{R} : C \to C \to \mathcal{U}$ and a type $\overline{\mathcal{R}} = \sum_{(a:C)} \sum_{(b:C)} a\ \mathcal{R}\ b$, such that $\overline{\mathcal{R}}$ and $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \to \mathtt{P}_S(\overline{\mathcal{R}})$ forms a P-coalgebra $\overline{\mathcal{R}}$-$\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, making the diagram in Figure 3.4 commute ($\Longrightarrow$ represents P-coalgebra morphisms).

What does commute mean here?

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

Figure 3.4: Bisimulation for a coalgebra

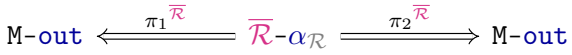

Figure 3.5: Bisimulation principle for final coalgebra

Definition 3.2.2 (Coinduction principle)**.** Given a relation $\mathcal{R}$, that is part of a bisimulation over a final P-coalgebra M-out : $\texttt{Coalg}_S$ we get the diagram in Figure 3.5, where $\pi_1^{\overline{\mathcal{R}}} = ! = \pi_2^{\overline{\mathcal{R}}}$ where ! is the unique mapping property (UMP) out of the final coalgebra. Given $r : m\ \mathcal{R}\ m'$ we get the equation

$$m = \pi_1^{\overline{\mathcal{R}}}(m, m', r) = \pi_2^{\overline{\mathcal{R}}}(m, m', r) = m'. \tag{3.33}$$

what is !

What is the consequence of this?

Chapter 4

Instantiation of M-types

In this section we show some examples of types that can be constructed as M-types, and show how their constructors can be defined.

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

4.1 Stream Formalization using M-types

As described earlier, given a type A we define the stream of that type as

$$\texttt{stream}\ A := \texttt{M}_{(A\,,\lambda\_,\mathbf{1})} \tag{4.1}$$

this is equal to an alternative definition of streams

...

4.2 ITrees as M-types

We want the following rules for ITrees

$$\frac{r : R}{\texttt{Ret}\ r : \texttt{itree E}\ R}\ \texttt{I}_{\texttt{Ret}} \tag{4.2}$$

$$\frac{A : \mathcal{U} \quad a : \texttt{E}\ A \quad f : A \to \texttt{itree E}\ R}{\texttt{Vis}\ a\ f : \texttt{itree E}\ R}\ \texttt{I}_{\texttt{Vis}}. \tag{4.3}$$

Elimination rules

$$\frac{t : \texttt{itree E}\ R}{\texttt{Tau}\ t : \texttt{itree E}\ R}\ \texttt{E}_{\texttt{Tau}}. \tag{4.4}$$

4.2.1 Delay Monad

We start by looking at ITrees without the `Vis` constructor, this type is also know as the delay monad

check this statement

. We construct this type by letting $S = (\mathbf{1} + R, \lambda\{\texttt{inl}\ \_ \to \mathbf{1}\ ;\ \texttt{inr}\ \_ \to \mathbf{0}\})$, we then get the polynomial functor

$$\texttt{P}_S(X) = \sum_{x:\mathbf{1}+R} \lambda\{\texttt{inl}\ \_ \to \mathbf{1}; \texttt{inr}\ \_ \to \mathbf{0}\}\ x \to X, \tag{4.5}$$

which is equal to

$$\mathtt{P}_S(X) = X + R \times (\mathbf{0} \to X). \tag{4.6}$$

We know that $(\mathbf{0} \to X) \equiv \mathbf{1}$, so we can reduce further to

$$\mathtt{P}_S(X) = X + R \tag{4.7}$$

meaning we get the following diagram.

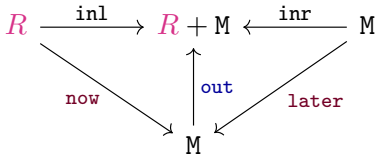

Figure 4.1: Delay monad

Meaning we can define the operations `now` and `later` using $\mathtt{in} = \mathtt{out}^{-1}$ together with the injections `inl` and `inr`.

(Later = Tau, Ret = Now)

4.2.2 Tree

Now lets look at the example where we remove the `Tau` constructor. We let

$$S = \left(R + \sum_{A:\mathcal{U}} \mathtt{E}\ A\ ,\ \lambda\{\mathtt{inl}\ \_ \to \mathbf{0}\ ;\ \mathtt{inr}\ (A, e) \to A\} \right). \tag{4.8}$$

This will give us the polynomial functor

$$\mathtt{P}_S(X) = \sum_{x:R+\sum_{A:\mathcal{U}} \mathtt{E}\ A} \lambda\{\mathtt{inl}\ \_ \to \mathbf{0}\ ;\ \mathtt{inr}\ (A, e) \to A\}\ x \to X, \tag{4.9}$$

which simplifies to

$$\mathtt{P}_S(X) = (R \times (\mathbf{0} \to X)) + (\sum_{A:\mathcal{U}} \mathtt{E}\ A \times (A \to X)), \tag{4.10}$$

and further

$$\mathtt{P}_S(X) = R + \sum_{A:\mathcal{U}} \mathtt{E}\ A \times (A \to X). \tag{4.11}$$

We get the following diagram for the P-coalgebra.

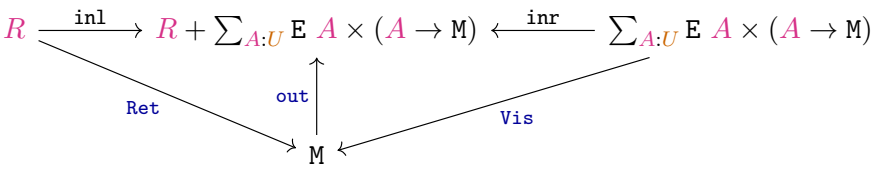

Figure 4.2: TODO

Again we can define `Ret` and `Vis` using the `in` function.

4.2.3 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. We define ITrees by the container

$$S = \left(\mathbf{1} + R + \sum_{A:\mathcal{U}} (\mathtt{E}\ A) \quad , \quad \lambda \{ \mathtt{inl}\ (\mathtt{inl}\ \_) \to \mathbf{1}\ ;\ \mathtt{inl}\ (\mathtt{inr}\ \_) \to \mathbf{0}\ ;\ \mathtt{inr}(A, \_) \to A \} \right). \quad (4.12)$$

Such that the (reduced) polynomial functor becomes

$$\mathtt{P}_S(X) = X + R + \sum_{A:\mathcal{U}} ((\mathtt{E}\ A) \times (A \to X)) \quad (4.13)$$

Giving us the diagram

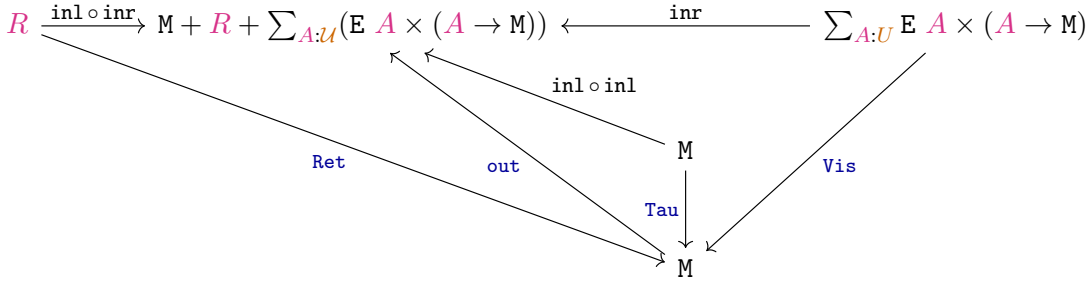

Figure 4.3: TODO

4.3 Automaton

An automaton is defined as a set of state V and an alphabet α and a transition function $\delta : V \to \alpha \to V$. This gives us the diagram in Figure 4.4

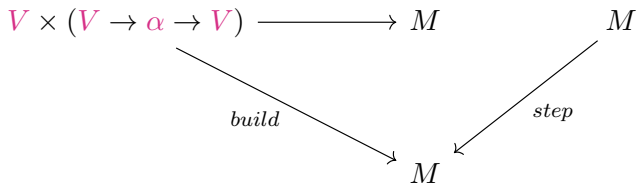

Figure 4.4: automaton

Chapter 5

QM-types

In this chapter we will introduce quotients, and show how quotients of M-types, are equal to QIIT.

better introduction to chapter, and reference to main points

5.1 Quotienting and Constructors

Describe set truncated quotients and their construction / elimination principles, and how it relates to quotienting M-types

5.2 Quotient M-type

Since we know that M-types preserves the H-level, we can use set-truncated quotients, to define quotient M-types, though we run into the problem of , as a solution we use QIIT to define the relation and type at the same type.

problem of direct quotients

5.3 Quotient inductive-inductive types (QIITs)

A quotient inductive-inductive type (QIIT) is a type together with a relation defined on that type. QIITs are HIITs that are set truncated.

5.4 Partiality monad

In this section we will define the partiality monad (see below) and show that (assuming the axiom of countable choice) the delay monad quotiented by weak bisimularity.

Definition 5.4.1 (Partiality Monad)**.** A simple example of a quotient inductive-inductive type is the partiality monad $(-)_\perp$ over a type R, defined by the constructors

$$\overline{R_\perp : \mathcal{U}} \quad (5.1) \qquad \overline{\perp : R_\perp} \quad (5.2) \qquad \frac{a : R}{\eta\ a : R_\perp} \quad (5.3)$$

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

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\bigsqcup\ (\mathtt{s}, \mathtt{b}) : R_\perp} \quad (5.4) \qquad \frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp\ p\ q : x \equiv y} \quad (5.5)$$

Should I define what it means to be an ordering relation separately, and just say the

$$\frac{x : R_\bot}{x \sqsubseteq_\bot x}\ \sqsubseteq_{\texttt{refl}} \tag{5.6}$$

$$\frac{x \sqsubseteq_\bot y \quad y \sqsubseteq_\bot z}{x \sqsubseteq_\bot z}\ \sqsubseteq_{\texttt{trans}} \tag{5.7}$$

$$\frac{x : R_\bot}{\bot \sqsubseteq_\bot x}\ \sqsubseteq_{\texttt{never}} \tag{5.8}$$

$$\frac{\mathtt{s} : \mathbb{N} \to R_\bot \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\bot \mathtt{s}_{n+1}}{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\bot \bigsqcup(\mathtt{s}, \mathtt{b})} \tag{5.9}$$

$$\frac{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\bot x}{\bigsqcup(\mathtt{s}, \mathtt{b}) \sqsubseteq_\bot x} \tag{5.10}$$

and finally set truncated

$$\frac{p, q : x \sqsubseteq_\bot y}{p \equiv q}\ (-)_\bot\text{-}\texttt{isSet} \tag{5.11}$$

5.4.1 Delay monad to Sequences

Definition 5.4.2. We define

$$\mathtt{Seq}_R = \sum_{(\mathtt{s}:\mathbb{N}\to R+\mathbf{1})} \mathtt{isMon}\ \mathtt{s} \tag{5.12}$$

where

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \equiv \mathtt{inr}\ \star) \times (\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \star)) \tag{5.13}$$

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 $\mathtt{s}_n \uparrow_{R+\mathbf{1}}$, or it is terminated $\mathtt{s}_n \equiv \mathtt{inl}\ r$ with some value r, denoted by $\mathtt{s}_n \downarrow_{R+\mathbf{1}} r$ or just $\mathtt{s}_n \downarrow_{R+\mathbf{1}}$ to mean $\mathtt{s}_n \not\equiv \mathtt{inr}\ \star$. Thus we can write $\mathtt{isMon}$ as

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \uparrow_{R+\mathbf{1}}) \times (\mathtt{s}_{n+1} \downarrow_{R+\mathbf{1}})) \tag{5.14}$$

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

$$\mathtt{now}_{Seq}\ r = (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{5.15}$$

$$\mathtt{never}_{Seq} = (\lambda\_, \mathtt{inr}\ \star), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{5.16}$$

Some comment about decidable equivalence needed to show that $\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \star$

Definition 5.4.3. We can shift a sequence $(\mathtt{s}, \mathtt{q})$ by inserting an element (and an equality) (z_s, z_q) at $n = 0$,

$$\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ (z_s, z_q) = \begin{cases} z_s & n = 0 \\ \mathtt{s}_m & n = m+1 \end{cases}, \begin{cases} z_q & n = 0 \\ \mathtt{q}_m & n = m+1 \end{cases}, \tag{5.17}$$

Definition 5.4.4. We can unshift a sequence by removing the first element of the sequence

$$\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{s} \circ \mathtt{succ}, \mathtt{q} \circ \mathtt{succ}. \tag{5.18}$$

Lemma 5.4.5. *The function*

$$\texttt{shift-unshift}\ (\texttt{s}, \texttt{q}) = \texttt{shift}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))\ (\texttt{s}_0, \texttt{q}_0) \tag{5.19}$$

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 5.4.6. *The function*

$$\texttt{unshift-shift}\ (\texttt{s}, \texttt{q}) = \texttt{unshift}\ (\texttt{shift}\ (\texttt{s}, \texttt{q})\ \_) \tag{5.20}$$

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. □

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.

Lemma 5.4.7 (inl≠inr)**.** *For any two elements* $x = \texttt{inl}\ a$ *and* $y = \texttt{inr}\ b$ *then* $x \not\equiv y$.

Proof. The constructors are disjoint, so there is not a path between them . □

better formulated proof

f

Definition 5.4.8. We define a function from `Delay` R to $\texttt{Seq}_R$

$$\begin{aligned}
&\texttt{Delay→Seq}\ (\texttt{now}\ r) = \texttt{now}_{Seq}\ r \\
&\texttt{Delay→Seq}\ (\texttt{later}\ x) = \\
&\qquad \texttt{shift}\ (\texttt{Delay→Seq}\ x)\ \left(\texttt{inr}\ \star, \begin{cases} \texttt{inr}\ (\texttt{refl}, \texttt{inl}\not\equiv\texttt{inr}) & x = \texttt{now}\ _ \\ \texttt{inl refl} & x = \texttt{later}\ _ \end{cases}\right)
\end{aligned} \tag{5.21}$$

Definition 5.4.9. We define function from $\texttt{Seq}_R$ to `Delay` R

$$\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q}) = \begin{cases} \texttt{now}\ r & \texttt{s}_0 = \texttt{inl}\ r \\ \texttt{later}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) & \texttt{s}_0 = \texttt{inr}\ \star \end{cases} \tag{5.22}$$

Theorem 5.4.10. *The type* $\texttt{Seq}_R$ *is equal to* `Delay` R

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

$$\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q})) \equiv (\texttt{s}, \texttt{q}) \tag{5.23}$$

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

$$\texttt{now}_{Seq}\ r \equiv (\texttt{s}, \texttt{q}) \tag{5.24}$$

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

$$\mathtt{shift}\ (\mathtt{Delay}{\to}\mathtt{Seq}\ (\mathtt{Seq}{\to}\mathtt{Delay}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q})))) \ (\mathtt{inr}\ \star, \_) \equiv (\mathtt{s}, \mathtt{q}) \tag{5.25}$$

by the induction hypothesis we get

$$\mathtt{Delay}{\to}\mathtt{Seq}\ (\mathtt{Seq}{\to}\mathtt{Delay}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}))) \equiv \mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) \tag{5.26}$$

since shift and unshift are inverse, we get the needed equality.

Shift takes two arguemnts, either clarify that its shift' that inserts inr tt or ...

For the left identity, we need to show that for any delay monad t we get

$$\mathtt{Seq}{\to}\mathtt{Delay}\ (\mathtt{Delay}{\to}\mathtt{Seq}\ t) \equiv t \tag{5.27}$$

defined by case analysis on t, if $t = \mathtt{now}\ a$ then the equality is $\mathtt{refl}$. If $t = \mathtt{later}\ x$ then we need to show

$$\mathtt{later}\ (\mathtt{Seq}{\to}\mathtt{Delay}\ (\mathtt{unshift}\ (\mathtt{shift}\ (\mathtt{Delay}{\to}\mathtt{Seq}\ x))) \equiv \mathtt{later}\ x \tag{5.28}$$

By unshift and shift being inverse, and the induction hypothesis we get the wanted equality. Since we are able to define a left and right identity function, we get the wanted equality. □

Corollary. *The types* $\mathtt{Delay}/\sim$ *and* $\mathtt{Seq}/\sim$ *are equal.*

Show this

Proof. We show that if $a \sim_{\mathtt{delay}} b$ then $\mathtt{Delay}{\to}\mathtt{Seq}\ a \sim_{\mathtt{Seq}} \mathtt{Delay}{\to}\mathtt{Seq}\ b$,

Show this

and we show if $x \sim_{\mathtt{Seq}} y$ then $\mathtt{Seq}{\to}\mathtt{Delay}\ x \sim_{\mathtt{Seq}} \mathtt{Seq}{\to}\mathtt{Delay}\ y$, □

5.4.2 Sequence to Partiality Monad

In this section we will show that assuming the "Axiom of Countable Choice", we get an equivalence between sequences and the partiality monad.

Definition 5.4.11 (Sequence Termination)**.** The following relations says that a sequence $(\mathtt{s}, \mathtt{q}) : \mathtt{Seq}_R$ terminates with a given value $r : R$,

$$(\mathtt{s}, \mathtt{q}) \downarrow_{\mathtt{Seq}} r = \sum_{(n:\mathbb{N})} \mathtt{s}_n \downarrow_{R+\mathbf{1}} r. \tag{5.29}$$

Definition 5.4.12 (Sequence Ordering)**.**

$$(\mathtt{s}, \mathtt{q}) \sqsubseteq_{\mathtt{Seq}} (\mathtt{t}, \mathtt{p}) = \prod_{(a:R)} \left(\|\mathtt{s} \downarrow_{\mathtt{Seq}} a\| \to \|\mathtt{t} \downarrow_{\mathtt{Seq}} a\|\right) \tag{5.30}$$

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

Definition 5.4.13. There is a conversion from $R + \mathbf{1}$ to the partiality monad $R_\perp$

$$\begin{aligned} \mathtt{Maybe}{\to}(-)_\perp\ (\mathtt{inl}\ r) &= \eta\ r \\ \mathtt{Maybe}{\to}(-)_\perp\ (\mathtt{inr}\ \star) &= \perp \end{aligned} \tag{5.31}$$

Definition 5.4.14 (Maybe Ordering)**.** Given some $x, y : R + \mathbf{1}$, the ordering relation is defined as

$$x \sqsubseteq_{\mathtt{R+1}} y = (x \equiv y) + ((x \downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}})) \tag{5.32}$$

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

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_{\mathtt{R+1}} \mathtt{s}_{n+1} \tag{5.33}$$

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 our definition of a sequence.

Lemma 5.4.15. *The function* $\mathtt{Maybe}{\to}(-)_\perp$ *is monotone, that is, if* $x \sqsubseteq_{A+\mathbf{1}} y$*, for some* x *and* y*, then* $(\mathtt{Maybe}{\to}(-)_\perp\ x) \sqsubseteq_\perp (\mathtt{Maybe}{\to}(-)_\perp\ y)$*.*

Proof. We do the proof by case.

$$\begin{aligned}
&\mathtt{Maybe}{\to}(-)_\perp\text{-}\mathtt{mono}\ (\mathtt{inl}\ p) = \\
&\qquad \mathtt{subst}\ (\lambda a,\ \mathtt{Maybe}{\to}(-)_\perp\ x \sqsubseteq_\perp \mathtt{Maybe}{\to}(-)_\perp\ a)\ p\ (\sqsubseteq_{\mathtt{refl}}\ (\mathtt{Maybe}{\to}(-)_\perp\ x)) \\
&\mathtt{Maybe}{\to}(-)_\perp\text{-}\mathtt{mono}\ (\mathtt{inr}\ (p, _)) = \\
&\qquad \mathtt{subst}\ (\lambda a,\ \mathtt{Maybe}{\to}(-)_\perp\ a \sqsubseteq_\perp \mathtt{Maybe}{\to}(-)_\perp\ y)\ p^{-1}\ (\sqsubseteq_{\mathtt{never}}\ (\mathtt{Maybe}{\to}(-)_\perp\ y))
\end{aligned} \tag{5.34}$$

□

Definition 5.4.16. There is a function taking a sequence to an increasing sequence

$$\begin{aligned}
&\mathtt{Seq}{\to}\mathtt{incSeq} \\
&\mathtt{Seq}{\to}\mathtt{incSeq}\ (\mathtt{g}, \mathtt{q}) = \mathtt{Maybe}{\to}(-)_\perp \circ \mathtt{g}, \mathtt{Maybe}{\to}(-)_\perp\text{-}\mathtt{mono} \circ \mathtt{q}
\end{aligned} \tag{5.35}$$

What is an increasing sequence ??, this is not defined anywhere!!

Definition 5.4.17. There is a function taking a sequence to the partiality monad

$$\begin{aligned}
&\mathtt{Seq}{\to}(-)_\perp : \mathtt{Seq}_A \to A_\perp \\
&\mathtt{Seq}{\to}(-)_\perp\ (\mathtt{g}, \mathtt{q}) = \bigsqcup \circ \mathtt{Seq}{\to}\mathtt{incSeq}
\end{aligned} \tag{5.36}$$

Lemma 5.4.18. *The function* $\mathtt{Seq}{\to}(-)_\perp$ *is monotone.*

$$\mathtt{Seq}{\to}(-)_\perp\text{-}\mathtt{mono} : \mathtt{isSet} A \to (x\ y : \mathtt{Seq}_A) \to x \sqsubseteq_{\mathtt{seq}} y \to \mathtt{Seq}{\to}(-)_\perp\ x \sqsubseteq_\perp \mathtt{Seq}{\to}(-)_\perp\ y \tag{5.37}$$

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

should this be formalized entirely, or should there just be a comment about monotonicity? Does not seem relevant? (There is alot of work

Definition 5.4.19. If two sequences x, y are weakly bisimular, then $\mathtt{Seq}{\to}(-)_\perp\ x \equiv \mathtt{Seq}{\to}(-)_\perp\ y$

$$\mathtt{Seq}{\to}(-)_\perp\text{-}{\approx}{\to}{\equiv}\ A_{set}\ x\ y\ (p, q) = \alpha_\perp\ (\mathtt{Seq}{\to}(-)_\perp\text{-}\mathtt{mono}\ A_{set}\ x\ y\ p)\ (\mathtt{Seq}{\to}(-)_\perp\text{-}\mathtt{mono}\ A_{set}\ y\ x\ q) \tag{5.38}$$

Definition 5.4.20 (Recursor for Quotient)**.** For all sequences $x, y : \mathtt{Seq}_A$, 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 $B_{set} : \mathtt{isSet}\ B$, we get a function $\mathtt{rec} : A/\mathtt{R} \to B$, defined by case as

$$\begin{aligned}
&\mathtt{rec}\ [\, z \,] = \mathtt{f}\ z \\
&\mathtt{rec}\ (\mathtt{eq}/\ _\ _\ r\ i) = \mathtt{g}\ r\ i \\
&\mathtt{rec}\ (\mathtt{squash}/\ a\ b\ p\ q\ i\ j) = B_{set}\ (\mathtt{rec}\ a)\ (\mathtt{rec}\ b)\ (\mathtt{ap}\ \mathtt{rec}\ p)\ (\mathtt{ap}\ \mathtt{rec}\ q)\ i\ j
\end{aligned} \tag{5.39}$$

This recursor allows us to lift the function $\mathtt{Seq}{\to}(-)_\perp$ to the quotient

Definition 5.4.21. We can define a function $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ from $\mathtt{Seq}_A$ to $A_\perp$, where $A_{set} : \mathtt{isSet}\ A$ as

$$\mathtt{Seq}/{\sim}{\to}(-)_\perp = \mathtt{rec}\ \mathtt{Seq}{\to}(-)_\perp\ (\mathtt{Seq}{\to}(-)_\perp\text{-}{\approx}{\to}{\equiv}\ A_{set})\ (-)_\perp\text{-}\mathtt{isSet} \tag{5.40}$$

Lemma 5.4.22. *Given two sequences s and t, if $\mathtt{Seq}{\to}(-)_\perp\ s \equiv \mathtt{Seq}{\to}(-)_\perp\ t$, then $s \sim_{\mathtt{seq}} t$.*

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

$$s \sim_{\mathtt{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\mathtt{seq}} r\| \to \|y \downarrow_{\mathtt{seq}} r\|\right) \times \left(\prod_{(r:R)} \|y \downarrow_{\mathtt{seq}} r\| \to \|x \downarrow_{\mathtt{seq}} r\|\right) \tag{5.41}$$

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

Lemma 5.4.23. *The function $\mathtt{Seq}/{\sim}{\to}(-)_\perp$ is injective.*

Should this be formalized?

Proof. We use propositional elimination of quotients

$$\begin{aligned}\mathtt{elimProp} : (\mathtt{B} : \mathtt{Seq}_R/{\sim}_{\mathtt{seq}} \to \mathcal{U}) \to ((x : \mathtt{Seq}_R/{\sim}_{\mathtt{seq}}) \to \mathtt{isProp}\ (\mathtt{B}\ x)) \\ \to (\mathtt{f} : (a : \mathtt{Seq}_R) \to \mathtt{B}\ [\ a\ ]) \to (x : \mathtt{Seq}_R/{\sim}_{\mathtt{seq}}) \to \mathtt{B}\ x\end{aligned} \tag{5.42}$$

to show the 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 start by eliminating x, followed by elimination of y, this gives us the proof term

Convert to text, instead of a proof term!?

$$\begin{aligned}&\mathtt{elimProp}\\ &\quad(\lambda\, a,\ \mathtt{Seq}/{\sim}{\to}(-)_\perp\ a \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\ y \to a \equiv y)\\ &\quad(\lambda\, a,\ \mathtt{isPropΠ}\ (\lambda\, \_, \mathtt{squash}/\ a\ y))\\ &\quad(\lambda\, a,\ \mathtt{elimProp}\\ &\qquad\qquad(\lambda\, b,\ \mathtt{Seq}{\to}(-)_\perp\ a \equiv \mathtt{Seq}/{\sim}{\to}(-)_\perp\ b \to [\ a\ ] \equiv b)\\ &\qquad\qquad(\lambda\, b,\ \mathtt{isPropΠ}\ (\lambda\, \_, \mathtt{squash}/\ [\ a\ ]\ b))\\ &\qquad\qquad(\lambda\, b,\ (\mathtt{eq}/\ a\ b) \circ (\mathtt{Seq}{\to}(-)_\perp\text{-}\mathtt{isInjective}\ a\ b)))\end{aligned} \tag{5.43}$$

where $\mathtt{Seq}{\to}(-)_\perp\text{-}\mathtt{isInjective}$ is (5.4.22), □

Lemma 5.4.24. *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. □

Lemma 5.4.25. *Assuming countable choice, the function $\mathtt{Seq}{\to}(-)_\perp$ is surjective*

describe countable choice (and why it is needed!)

describe what it means to do the surjective proof by case!

Proof. We do the proof by case on $R_\perp$, if it is $\eta\ r$ or $\texttt{never}$, we convert them to the sequences $\texttt{now}_{seq}\ r$ and $\texttt{never}_{seq}$ respectively, then we are done by (5.4.24). For the least upper bound $\bigsqcup(\texttt{s}, \texttt{b})$, we translate to the (increasing) sequence, defined by $(\texttt{s}, \texttt{b})$. □

more precise description!

Lemma 5.4.26. *Assuming countable choice, the function* $\texttt{Seq}/{\sim} \to (-)_\perp$ *is surjective*

Theorem 5.4.27. *Assuming countable choice, we get an equivalence between sequences and the partiality monad.*

Proof. The function $\texttt{Seq}/{\sim} \to (-)_\perp$ is injective and surjective assuming countable choice, meaning we get an equivalence, since we are working in hSets. □

Complete the rest of the proof!

5.4.2.1 Building the Partiality Monad as a limit (Dialgebra?)

Is this possible?

5.4.3 Silhouette Trees

We start by defining an R valued E branching tree, as the $\texttt{M}$-type given by the following container

$$\left(R + \mathbf{1}, \begin{cases} \perp & \texttt{inl}\ a \\ E & \texttt{inr}\ \star \end{cases}\right) \tag{5.44}$$

We get the constructors

$$\frac{a : R}{\texttt{leaf}\ a : \texttt{tree}\ R\ E} \tag{5.45}$$

$$\frac{\texttt{k} : E \to \texttt{tree}\ R\ E}{\texttt{node}\ \texttt{k} : \texttt{tree}\ R\ E} \tag{5.46}$$

Then we define the weak bisimularity relation $\sim_{\texttt{tree}}$

$$\frac{}{\texttt{leaf}\ x \sim_{\texttt{tree}} \texttt{leaf}\ y}\ {\sim}\texttt{leaf} \tag{5.47}$$

$$\frac{\prod_{(v:E)}\ \texttt{k}_1\ v \sim_{\texttt{tree}} \texttt{k}_2\ v}{\texttt{node}\ \texttt{k}_1 \sim_{\texttt{tree}} \texttt{node}\ \texttt{k}_2}\ {\sim}\texttt{node} \tag{5.48}$$

This is enough to define, what we call, silhouette trees, which are trees quotiented by this notion of weak bisimularity, namely $\texttt{tree}/{\sim_{\texttt{tree}}}$. We can also construct this type directly as a QIIT, with type constructors

add all needed constructors

$$\frac{}{\texttt{leaf}_{\texttt{sTree}} : \texttt{sTree}\ E} \tag{5.49}$$

$$\frac{\texttt{k} : E \to \texttt{sTree}\ E}{\texttt{node}_{\texttt{sTree}}\ \texttt{k} : \texttt{sTree}\ E} \tag{5.50}$$

And the ordering relation $(\cdot \sqsubseteq_{\texttt{sTree}} \cdot)$ of how "defined" the trees are by the constructors

add all needed constructors

$$\frac{x \sqsubseteq_{\texttt{sTree}} y \quad y \sqsubseteq_{\texttt{sTree}} x}{\alpha_{\texttt{sTree}}\ x\ y : \texttt{sTree}\ E} \tag{5.51}$$

$$\frac{\texttt{s} : (\mathbb{N} \to E) \to \texttt{sTree}\ E}{\bigsqcup_{(\texttt{e}:\mathbb{N}\to E)}\ (\texttt{s}\ \texttt{e})} \tag{5.52}$$

5.4.3.1 From tree to $\mathtt{Seq}_{tree}$

We now want to show the equivalence between these two constructions, to do this we define an intermediate construction $\mathtt{Seq}_{tree}$, where we get an ordering on the "definedness" of trees.

ordering is container ordering not maybe?

Definition 5.4.28. We define monotone increasing sequences of trees as , all breanches are monotone increasing .

$$\mathtt{Seq}_{tree} = \prod_{(\mathtt{e}:\mathbb{N}\to E)} \sum_{(\mathtt{s}:\mathbb{N}\to R+1)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_{\mathtt{R+1}} \mathtt{s}_{n+1} \tag{5.53}$$

where $\sqsubseteq_{\mathtt{R+1}}$ is similar to the relation defined at (5.4.14) .

specify branches increasing?

Definition 5.4.29. We define a function to shift a $\mathtt{Seq}_{tree}$, it takes $\mathtt{f} : E \to \mathtt{Seq}_{tree}$ as an argument. We let $\mathtt{s}' = \mathtt{f}\ \mathtt{e}_0\ (\mathtt{e} \circ \mathtt{succ})$, then the definition is given as

$$\begin{aligned}&\mathtt{shift\text{-}seq}\ \mathtt{f} = \\ &\quad \lambda\, \mathtt{e}, \langle\!|\ \mathtt{inr}\ \star\ ,\ \pi_1\ \mathtt{s}'\ |\!\rangle, \left(\lambda\, n, \begin{cases} \mathtt{inr}\ (\mathtt{refl}, \mathtt{inl}\not\equiv\mathtt{inr}) & n = 0 \wedge \pi_1\ \mathtt{s}'\ 0 = \mathtt{inl}\ r \\ \mathtt{inl}\ \mathtt{refl} & n = 0 \wedge \pi_1\ \mathtt{s}'\ 0 = \mathtt{inr}\ \star \\ \pi_2\ \mathtt{s}'\ m & n = m+1 \end{cases}\right)\end{aligned} \tag{5.54}$$

how does it differ? Constructors are equal type is different (trees instead of delay)

Definition 5.4.30. We define a function to unshift a Seq_{tree}

$$\mathtt{unshift\text{-}seq}\ \mathtt{s}\ v = \lambda\, \mathtt{e},\ (\pi_1\ (\mathtt{s}\ (\langle\!|\ v\ ,\ \mathtt{e}\ |\!\rangle)) \circ \mathtt{succ}), (\pi_2\ (\mathtt{s}\ (\langle\!|\ v\ ,\ \mathtt{e}\ |\!\rangle)) \circ \mathtt{succ}) \tag{5.55}$$

Lemma 5.4.31. *Shift and unshift er inverse to each other*

Proof. The same reasoning as for □

shift - unshift

Definition 5.4.32. We get a function from trees to monotone sequences

$$\begin{aligned}&\mathtt{tree}\to\mathtt{Seq}\ (\mathtt{leaf}\ r) = \lambda\_, (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \\ &\mathtt{tree}\to\mathtt{Seq}\ (\mathtt{node}\ \mathtt{k}) = \mathtt{shift}\ (\mathtt{tree}\to\mathtt{Seq} \circ \mathtt{k})\end{aligned} \tag{5.56}$$

Definition 5.4.33. We get a function from monotone sequences to trees

$$\mathtt{Seq}\to\mathtt{tree}\ \mathtt{s} = \begin{cases} \mathtt{leaf}\ r & \prod_{(\mathtt{e}:\mathbb{N}\to E)} \pi_1\ (\mathtt{s}\ \mathtt{e})\ 0 = \mathtt{inl}\ r \\ \mathtt{node}\ (\mathtt{Seq}\to\mathtt{tree} \circ \mathtt{unshift}\ \mathtt{s}) & o.w. \end{cases} \tag{5.57}$$

Lemma 5.4.34. *If the first element in the sequence is terminated / a leaf, then the rest of the elements will also be terminated.*

$$\left(\prod_{e:\mathbb{N}\to E} \pi_1\ (\mathtt{s}\ \mathtt{e})\ 0 = \mathtt{inl}\ r\right) \Leftrightarrow (\mathtt{s} \equiv \lambda\_, (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl})) \tag{5.58}$$

Proof. Since the sequence is monotone, and $\mathtt{inl}\ r$ is the top element of the order, if the first element is $\mathtt{inl}\ r$, then the sequence must be $\lambda\_, (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl})$. The other direction is trivial. □

Theorem 5.4.35. *The types* tree *and* $\mathtt{Seq}_{tree}$ *are equal*

Proof. We construct an isomorphism by the functions $\texttt{tree}\to\texttt{Seq}$ and $\texttt{Seq}\to\texttt{tree}$, with right inverse given by two cases, one where the first element in the sequence is $\texttt{inl}\ r$, meaning representing a leaf with value r, then we need to show that $\texttt{s} \equiv \lambda\_, (\lambda\_, \texttt{inl}\ r), (\lambda\_, \texttt{inl refl})$ which follows from Lemma 5.4.34. Otherwise we need to show that

$$\texttt{shift}\ (\texttt{tree}\to\texttt{Seq} \circ \texttt{Seq}\to\texttt{tree} \circ \texttt{unshift}\ \texttt{s}) \equiv \texttt{s} \tag{5.59}$$

By induction we get

$$\texttt{tree}\to\texttt{Seq} \circ \texttt{Seq}\to\texttt{tree} \circ \texttt{unshift}\ \texttt{s} \equiv \texttt{unshift}\ \texttt{s} \tag{5.60}$$

then by the right inverse of the equality between shift and unshift, we are done. For the left inverse we do case analysis, using induction and the left inverse of the equality between shift and unshift

$$\begin{aligned} &\texttt{tree-Seq}\ (\texttt{leaf}\ r) = \texttt{refl} \\ &\texttt{tree-Seq}\ (\texttt{node}\ k) = \texttt{unshift-shift}\ (\texttt{tree}\to\texttt{Seq} \circ \texttt{k}) \cdot \texttt{tree-Seq}\ k \end{aligned} \tag{5.61}$$

□

We start by defining some ordering relation on $\texttt{Seq}_{tree}$

Definition 5.4.36 (Sequence Termination)**.** The following relations says that a branche $\texttt{e} : \mathbb{N} \to E$ of a sequence $\texttt{s} : \texttt{Seq}_{tree}$ terminates with some value $r : R$,

$$(\texttt{s}\ \texttt{e}) \downarrow_{\texttt{Seq}_{\texttt{tree}}} r = \sum_{(n:\mathbb{N})} (\texttt{s}\ \texttt{e}\ n) \downarrow_{R+\mathbf{1}} r. \tag{5.62}$$

Definition 5.4.37 (Sequence Ordering)**.**

$$\texttt{s} \sqsubseteq_{\texttt{Seq}_{\texttt{tree}}} \texttt{t} = \prod_{(e:E)} \prod_{(a:R)} \left(\|(\texttt{s}\ \texttt{e}) \downarrow_{\texttt{Seq}} a\| \to \|(\texttt{t}\ \texttt{e}) \downarrow_{\texttt{Seq}} a\| \right) \tag{5.63}$$

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

Definition 5.4.38. We define weak bisimularity relation for sequences

$$\texttt{s} \sim_{\texttt{Seq}_{\texttt{tree}}} \texttt{t} = \texttt{s} \sqsubseteq_{\texttt{Seq}_{\texttt{tree}}} \texttt{t} \times \texttt{t} \sqsubseteq_{\texttt{Seq}_{\texttt{tree}}} \texttt{s} \tag{5.64}$$

Is this defined for partiality monad?

Corollary. *The types* $\texttt{tree}/\sim_{\texttt{tree}}$ *and* $\texttt{Seq}_{tree}/\sim_{\texttt{Seq}_{\texttt{tree}}}$ *are equal.*

Proof. We follow the same strategy as for $\texttt{Delay}/\sim$ and $\texttt{Seq}/\sim$ □

5.4.3.2 $\texttt{Seq}_{tree}$ to sTree

Introduction to Seq to tree

Definition 5.4.39. We define a function converting a sequence on trees to a monotone sequence on sTree's

$$\texttt{Seq}\to\texttt{incSeq} \tag{5.65}$$

Definition 5.4.40. There is a function from $\texttt{Seq}_{tree}$ to sTree

$$\texttt{Seq}\to\texttt{sTree}\ \texttt{s} = \bigsqcup_{(\texttt{e}:\mathbb{N}\to E)} (\texttt{Seq}\to\texttt{incSeq}\ \texttt{s}\ \texttt{e}) \tag{5.66}$$

Lemma 5.4.41. *Given* $a \sqsubseteq_{\mathtt{Seq_{tree}}} b$, *then* $\mathtt{Seq{\to}sTree}\ a \sqsubseteq_{\mathtt{sTree}} \mathtt{Seq{\to}sTree}\ b$.

Proof. . □ Complete proof

Since this definition is monotone, it can be lifted to the quotiented sequences.

describe better **Lemma 5.4.42.** *If two elements are weakly bisimilar, then they are equal as* sTrees

$$\begin{aligned}\mathtt{Seq{\to}sTree\text{-}{\approx}{\to}{\equiv}}\ x\ y\ (\mathtt{p},\mathtt{q}) = \\ \alpha_{\mathtt{sTree}}\ (\mathtt{Seq{\to}sTree\text{-}mono}\ x\ y\ \mathtt{p})\ (\mathtt{Seq{\to}sTree\text{-}mono}\ y\ x\ \mathtt{q})\end{aligned} \tag{5.67}$$

Definition 5.4.43. Function from Seq/∼ to sTree

$$\mathtt{Seq/{\sim}{\to}sTree} = \mathtt{rec\ Seq{\to}sTree\ Seq{\to}sTree\text{-}{\approx}{\to}{\equiv}\ sTree\text{-}isSet} \tag{5.68}$$

TODO **Lemma 5.4.44.** Seq/∼→sTree *is injective*

TODO **Lemma 5.4.45.** Seq/∼→sTree *is surjective*

TODO **Theorem 5.4.46.** Seq/∼→sTree *is an equivalence.*

5.4.4 QM-types

We want to define sequences based on M-types, here are some examples

$$\mathtt{Seq}_{\mathtt{M}_{(A\,,\,0)}} = A \tag{5.69}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A+1\,,\,[0,1])}} = \sum_{(\mathtt{s}:\mathbb{N}\to A+1)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.70}$$

$$\mathtt{Seq}_{\mathtt{M}_{(1+1\,,\,[0,E])}} = \prod_{(\mathtt{e}:\mathbb{N}\to E)} \sum_{(\mathtt{s}:\mathbb{N}\to \mathbf{1}+\mathbf{1})} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.71}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A+1\,,\,[0,E])}} = \prod_{(\mathtt{e}:\mathbb{N}\to E)} \sum_{(\mathtt{s}:\mathbb{N}\to A+1)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.72}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A+B\,,\,[0,E])}} = \prod_{(\mathtt{e}:\mathbb{N}\to E)} \sum_{(\mathtt{s}:\mathbb{N}\to A+B)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.73}$$

$$\mathtt{Seq}_{\mathtt{M}_{(1\,,\,1+1)}} = \prod_{(\mathtt{e}:\mathbb{N}\to \mathbf{1}+\mathbf{1})} \sum_{(\mathtt{s}:\mathbb{N}\to 1)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.74}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A\,,\,1+1)}} = \prod_{(\mathtt{e}:\mathbb{N}\to \mathbf{1}+\mathbf{1})} \sum_{(\mathtt{s}:\mathbb{N}\to A)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.75}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A\,,\,\mathtt{B})}} = \prod_{(\mathtt{e}:(n:\mathbb{N})\to \mathtt{B}\,\mathtt{s}_{n-1})} \sum_{(\mathtt{s}:\mathbb{N}\to A)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.76}$$

$$\mathtt{Seq}^{(n)}_{\mathtt{M}_{(A\,,\,\mathtt{B})}}\ (\mathtt{s}_{n-1} : A) = \sum_{(\mathtt{s}_n:A)} \prod_{(n:\mathbb{N})} \prod_{(\mathtt{e}:(n:\mathbb{N})\to \mathtt{B}\,\mathtt{s}_{n-1})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.77}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A+B\,,\,[X,Y])}} = \prod_{\left(\mathtt{e}:(n:\mathbb{N})\to \begin{cases} X\ a & \mathtt{s}_{n-1}=\mathtt{inl}\ a \\ Y\ b & \mathtt{s}_{n-1}=\mathtt{inr}\ b \end{cases}\right)} \sum_{(\mathtt{s}:\mathbb{N}\to A)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq \mathtt{s}_{n+1} \tag{5.78}$$

$$\mathtt{Seq}_{\mathtt{M}_{(A\,,\,B)}} = \sum_{(s' : \mathtt{M}_{(A\,,\,\mathtt{B})})} \sum_{(\mathtt{s} : \mathbb{N} \to \sum_{(a:A)} \mathtt{B}\, a \to \mathtt{M}_{(A\,,\,\mathtt{B})})} \left(\prod_{p : \sum_{a:A} \mathtt{B}\, a} s' \sqsubseteq (\mathtt{s}_0\ p) \right) \times \left(\prod_{(n:\mathbb{N})} \prod_{(p : \sum_{(a:A)} B\, a)} (s_n\ p) \sqsubseteq (s_{n+1}\ p) \right) \tag{5.79}$$

A QM-type is a quotiented M-type, we try to define this as a quotient on containers. We define container quotients as

which other QM types can be expressed as QIITs

5.5 Cofree Coalgebra / Dialgebra

5.6 TODO

- Resumption Monad transformer
- coinduction in Coq is broken
- bisim $\Rightarrow$ eq
- copattern matching
- cubical Agda. Relation between `M`-types defined by coinduction/copattern matching and constructed from W-types
  - In Agda, co-inductive types are defined using Record types, which are Sigma-types.
  - In cubical Agda, 3.2.2 the issue of productivity is discussed. This can probably be made precise using guarded types.
- streams defined by guarded recursion vs coinduction in guarded cubical Agda.
  - p3 of the guarded cubical Agda paper describes how semantic productivity improves over syntactic productivity
- Reduction of co-inductive types in Coq/Agda to (indexed) `M`-types. Like reduction of strictly positive inductive types to W-types. https://ncatlab.org/nlab/show/W-type
- QIITs have been formalized in Agda using private types. Can this also be done in cubical Agda (ie without cheating).
- Show that this is the final (quotiented) coalgebra. Does this generalize to `QM`-types, and what are those constructively ??

Chapter 6

Properties of M-types?

6.1 Closure properties of M-types

We want to show that M-types are closed under simple operations, we start by looking at the product.

6.1.1 Product of M-types

We start with containers and work up to M-types.

Definition 6.1.1. The product of two containers is defined as [1]

$$(A \triangleleft \mathtt{B}) \times (C \triangleleft \mathtt{D}) \equiv (A \times C \triangleleft \lambda\,(a, c), \mathtt{B}\ a \times \mathtt{D}\ c). \tag{6.1}$$

We can lift this rule, through the diagram in Figure 6.1, used to define M-types.

Theorem 6.1.2. *For any $n : \mathbb{N}$ the following is true*

$$\mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{n}\ \mathbf{1} \times \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{n}\ \mathbf{1} \equiv \mathtt{P}_{(A \triangleleft \mathtt{B}) \times (C \triangleleft \mathtt{D})}{}^{n}\ \mathbf{1}. \tag{6.2}$$

Proof. We do induction on n, for $n = 0$, we have $\mathbf{1} \times \mathbf{1} \equiv \mathbf{1}$. For $n = m + 1$, we may assume

$$\mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1} \equiv \mathtt{P}_{(A \triangleleft \mathtt{B}) \times (C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1}, \tag{6.3}$$

in the following

$$\begin{aligned}
&\mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m+1}\ \mathbf{1} \times \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m+1}\ \mathbf{1} && (6.4)\\
\equiv\ &\mathtt{P}_{(A \triangleleft \mathtt{B})}(\mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m}\ \mathbf{1}) \times \mathtt{P}_{(C \triangleleft \mathtt{D})}(\mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1}) && (6.5)\\
\equiv\ &\sum_{a:A} \mathtt{B}\ a \to \mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m}\ \mathbf{1} \times \sum_{c:C} \mathtt{D}\ c \to \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1} && (6.6)\\
\equiv\ &\sum_{a,c:A \times C} (\mathtt{B}\ a \to \mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m}\ \mathbf{1}) \times (\mathtt{D}\ c \to \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1}) && (6.7)\\
\equiv\ &\sum_{a,c:A \times C} \mathtt{B}\ a \times \mathtt{D}\ c \to \mathtt{P}_{(A \triangleleft \mathtt{B})}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C \triangleleft \mathtt{D})}{}^{m}\ \mathbf{1} && (6.8)
\end{aligned}$$

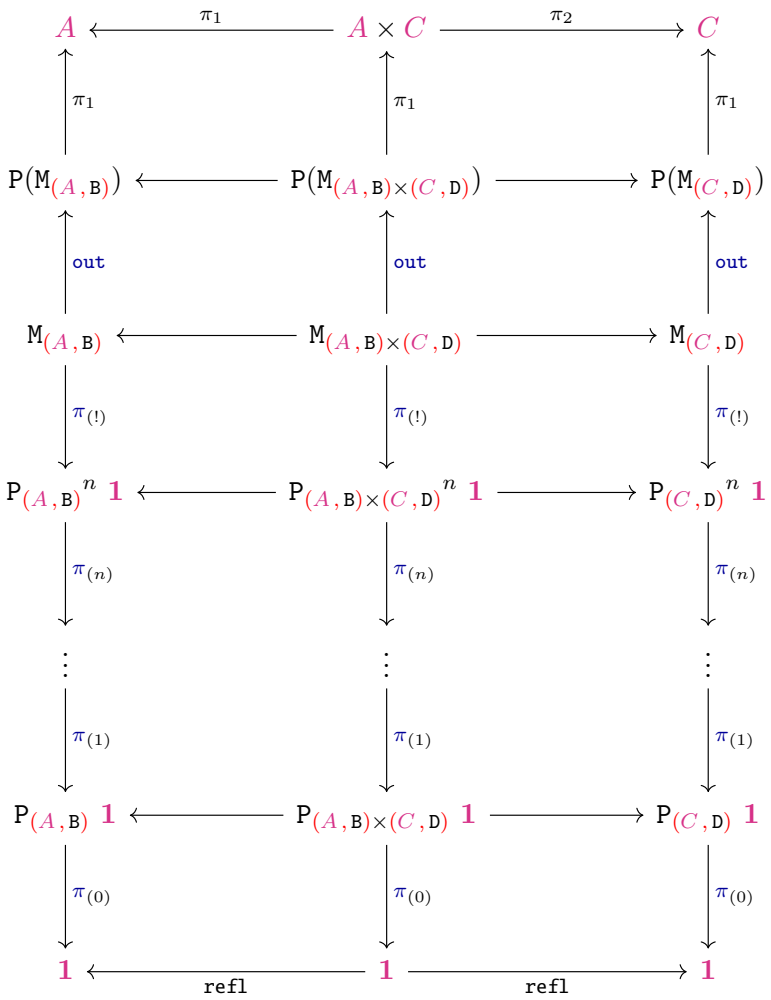

Figure 6.1: Diagram for products of chains

$$\equiv \sum_{a,c:A\times C} \mathtt{B}\ a \times \mathtt{D}\ c \to \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m}\ \mathbf{1} \tag{6.9}$$

$$\equiv \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}(\mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m}\ \mathbf{1}) \tag{6.10}$$

$$\equiv \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m+1}\ \mathbf{1} \tag{6.11}$$

taking the limit of (6.2) we get

$$\mathtt{M}_{(A\,,\mathtt{B})} \times \mathtt{M}_{(C\,,\mathtt{D})} \equiv \mathtt{M}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}. \tag{6.12}$$

□

Example 3. For streams we get

$$\mathtt{stream}\ A \times \mathtt{stream}\ B \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})} \times \mathtt{M}_{(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})\times(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{stream}\ (A \times B) \tag{6.13}$$

as expected. Transporting along (6.13) gives us a definition for `zip`.

6.1.2 Co-product

Coproducts?

6.1.3 ...

The rest of the closures defined in "Categories of Containers" [1]

Chapter 7

Examples of `M`-types

7.1 The Partiality monad

To construct the partiality monad, we start with the delay monad, and the preorder

$$\forall x, \bot \sqsubseteq x \tag{7.1}$$

$$\forall\, x, x \sqsubseteq x \tag{7.2}$$

$$\forall\, x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{7.3}$$

we can then define the partiality monad

The partiality monad $(-)_\bot$ is a way of adding partiality to a given computation. Along with the partiality monad, we also get a partial ordering $(\cdot \sqsubseteq \cdot)$, by

$$\forall x, \bot \sqsubseteq x \tag{7.4}$$

$$\forall\, x, x \sqsubseteq x \tag{7.5}$$

$$\forall\, x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{7.6}$$

$$\forall\, x\, y, x \sqsubseteq y \to y \sqsubseteq x \to x \equiv y \tag{7.7}$$

We now want to show that we can construct the partiality monad from the delay monad. We need an operation that given an element of the delay monad, maps to an element of the partiality monad.

$$\texttt{now}\ x = x + \mathbf{1} \tag{7.8}$$

$$\texttt{later}\ y = y \tag{7.9}$$

7.2 TODO: Place these subsections

What makes a relation a bisimulation? Is bisim and equality equal.

7.2.1 Identity Bisimulation

Lets start with a simple example of a bisimulation namely the one given by the identity relation for any M-type.

Lemma 7.2.1. *The identity relation* $(\cdot \equiv \cdot)$ *is a bisimulation for any final coalgebra* $\mathtt{M}_S\text{-}\mathtt{out}$ *defined over an* M*-type.*

Proof. We first define the function

$$\begin{aligned} \alpha_{\equiv} &: \overline{\equiv} \to \mathtt{P}(\overline{\equiv}) \\ \alpha_{\equiv}(x, y) &:= \pi_1\ (\mathtt{out}\ x)\ ,\ (\lambda\, b, (\pi_2\ (\mathtt{out}\ x)\ b\ , \mathtt{refl}_{(\pi_2\ (\mathtt{out}\ x)\ b)})) \end{aligned} \tag{7.10}$$

and the two projections

$$\pi_1^{\overline{\equiv}} = (\pi_1, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{refl}_{\mathtt{out}\,a}) \tag{7.11}$$

$$\pi_2^{\overline{\equiv}} = (\pi_2, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{cong}_{\mathtt{out}}\ (r^{-1})). \tag{7.12}$$

This defines the bisimulation, given by the diagram in Figure 7.1. □

$$\mathtt{M}\text{-}\mathtt{out} \xleftarrow{\pi_1^{\overline{\equiv}}} \equiv\text{-}\alpha_{\equiv} \xrightarrow{\pi_2^{\overline{\equiv}}} \mathtt{M}\text{-}\mathtt{out}$$

Figure 7.1: Identity bisimulation

7.2.2 Bisimulation of Streams

TODO

7.2.3 Bisimulation of Delay Monad

We want to define a strong bisimulation relation $\sim_{\mathtt{delay}}$ for the delay monad,

Definition 7.2.2. The relation $\sim_{\mathtt{delay}}$ is defined by the following rules

$$\frac{R : U \quad r : R}{\mathtt{now}\ r \sim_{\mathtt{delay}} \mathtt{now}\ r : \mathcal{U}}\ \mathtt{now}\sim \tag{7.13}$$

$$\frac{R : \mathcal{U} \quad t : \mathtt{delay}\ R \quad u : \mathtt{delay}\ R \quad t \sim_{\mathtt{delay}} u : \mathcal{U}}{\mathtt{later}\ t \sim_{\mathtt{delay}} \mathtt{later}\ u : \mathcal{U}}\ \mathtt{later}\sim \tag{7.14}$$

Theorem 7.2.3. *The relation* $\sim_{\mathtt{delay}}$ *is a bisimulation for* $\mathtt{delay}\ R$.

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\mathtt{delay}}} &: \overline{\sim_{delay}} \to \mathtt{P}(\overline{\sim_{delay}}) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{now}\sim\ r) &:= (\mathtt{inr}\ r, \lambda\,(\,)) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{later}\sim\ x\ y\ q) &:= (\mathtt{inl}\ \star, \lambda\,\_, (x, y, q)) \end{aligned} \tag{7.15}$$

then we define the projections

$$\pi_1^{\overline{\sim_{delay}}} = \left(\pi_1 \quad , \quad \texttt{funExt}\ \lambda\,(a,b,p), \begin{cases} (\texttt{inr}\ r, \lambda\,()) & p = \texttt{now}\sim\ r \\ (\texttt{inl}\ \star, \lambda\,\_, x) & p = \texttt{later}\sim\ x\ y\ q \end{cases}\right) \tag{7.16}$$

$$\pi_2^{\overline{\sim_{delay}}} = \left(\pi_2 \quad , \quad \texttt{funExt}\ \lambda\,(a,b,p), \begin{cases} (\texttt{inr}\ r, \lambda\,()) & p = \texttt{now}\sim\ r \\ (\texttt{inl}\ \star, \lambda\,\_, y) & p = \texttt{later}\sim\ x\ y\ q \end{cases}\right) \tag{7.17}$$

$$\tag{7.18}$$

This defines the bisimulation, given by the diagram in Figure 7.2. □

$$\texttt{delay}\ R\text{-}\texttt{out} \xleftarrow{\pi_1^{\overline{\sim_{delay}}}} \overline{\sim_{\texttt{delay}}}\text{-}\alpha_{\sim_{\texttt{delay}}} \xrightarrow{\pi_2^{\overline{\sim_{delay}}}} \texttt{delay}\ R\text{-}\texttt{out}$$

Figure 7.2: Strong bisimulation for delay monad

7.2.4 Bisimulation of ITrees

We define our bisimulation coalgebra from the strong bisimulation relation $\mathcal{R}$, defined by the following rules.

$$\frac{a, b : R \quad a \equiv_R b}{\texttt{Ret}\ a \cong \texttt{Ret}\ b}\ \texttt{EqRet} \tag{7.19}$$

$$\frac{t, u : \texttt{itree E}\ R \quad t \cong u}{\texttt{Tau}\ t \cong \texttt{Tau}\ u}\ \texttt{EqTau} \tag{7.20}$$

$$\frac{A : \mathcal{U} \quad e : \texttt{E}\ A \quad k_1, k_2 : A \to \texttt{itree E}\ R \quad t \cong u}{\texttt{Vis}\ e\ k_1 \cong \texttt{Tau}\ e\ k_2}\ \texttt{EqVis} \tag{7.21}$$

Now we just need to define $\alpha_{\mathcal{R}}$

define the $\alpha_{\mathcal{R}}$ function

. Now we have a bisimulation relation, which is equivalent to equality, using what we showed in the previous section.

7.2.5 Zip Function

We want the diagram in Figure 7.3 to commute, meaning we get the computation rules

$$(\texttt{hd} \times \texttt{hd}) \equiv \texttt{hd} \circ \texttt{zip} \tag{7.22}$$

$$\texttt{zip} \circ (\texttt{tl} \times \texttt{tl}) \equiv \texttt{tl} \circ \texttt{zip} \tag{7.23}$$

we can define the zip function as we did in the end of the last section. Another way to define the zip function is more directly, using the following lifting property of M-types

$$\texttt{lift}_{\texttt{M}} \left(x : \prod_{n:\mathbb{N}} (A \to \texttt{P}_S{}^n\ \texttt{1})\right) \left(u : \prod_{n:\mathbb{N}} (A \to \pi_n(x_{n+1}a) \equiv x_n a)\right) (a : A) : \texttt{M}\ S := \tag{7.24}$$
$$(\lambda\, n, x\ n\ a), (\lambda\, n\ i, p\ n\ a\ i).$$

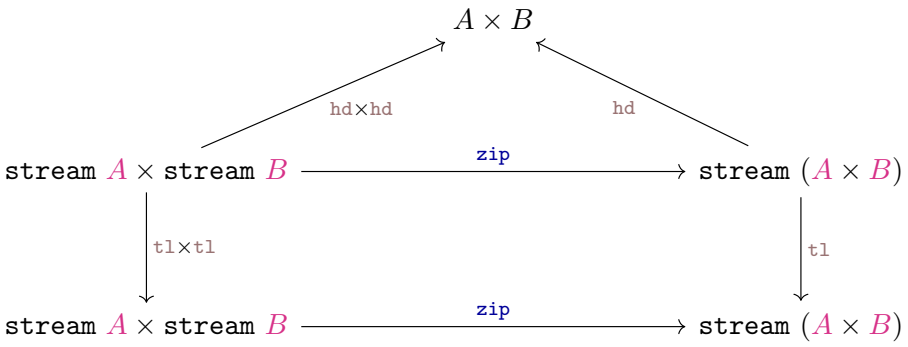

Figure 7.3: TODO

To use this definition, we first define some helper functions

$$\texttt{zip}_X\ n\ (x, y) = \begin{cases} \texttt{1} & \texttt{if } n = 0 \\ (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda \_, \texttt{zip}_X\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)), & \texttt{if } n = m + 1 \end{cases} \tag{7.25}$$

$$\texttt{zip}_\pi\ n\ (x, y) = \begin{cases} \texttt{refl} & \texttt{if } n = 0 \\ \lambda\, i, (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda \_, \texttt{zip}_\pi\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)\ i), & \texttt{if } n = m + 1 \end{cases}, \tag{7.26}$$

we can then define

$$\texttt{zip}_{lift}\ (x, y) := \texttt{lift}_\texttt{M}\ \texttt{zip}_X\ \texttt{zip}\ (x, y). \tag{7.27}$$

7.2.5.1 Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\begin{aligned} \texttt{transport}_?\ a &\equiv \texttt{zip}_{lift}\ a && (7.28)\\ &\equiv \texttt{lift}_\texttt{M}\ \texttt{zip}_X\ \texttt{zip}_\pi\ (x, y) && (7.29)\\ &\equiv (\lambda\, n, \texttt{zip}_X\ n\ (x, y)), (\lambda\, n\ i, \texttt{zip}_\pi\ n\ (x, y)\ i) && (7.30) \end{aligned}$$

zero case X

$$\texttt{zip}_X\ 0\ (x, y) \equiv \texttt{1} \tag{7.31}$$

Successor case X

$$\begin{aligned} \texttt{zip}_X\ (m + 1)\ (x, y) &\equiv (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda \_, \texttt{zip}_X\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)) && (7.32)\\ &\equiv (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda \_, ?\ (\texttt{tl}\ a)) && (7.33)\\ &\equiv (\texttt{hd}\ (\texttt{transport}_? a)), (\lambda \_, \texttt{transport}_?(\texttt{tl}\ a)) && (7.34)\\ &\equiv \texttt{transport}_?\ a && (7.35)\\ & && (7.36) \end{aligned}$$

Zero case π: $(\lambda\, i, \texttt{zip}_\pi\ 0\ (x, y)\ i \equiv refl)$.

$$\begin{aligned} &\equiv (), (\lambda\, i, \texttt{zip}_\pi\ 0\ (x, y)\ i) && (7.37)\\ &\equiv \texttt{1}, \texttt{refl} && (7.38) \end{aligned}$$

(7.39)

successor case

$$\equiv (\texttt{zip}_X\ (m+1)\ (x,y)), (\lambda\, i, \texttt{zip}_\pi\ (m+1)\ (x,y)\ i) \tag{7.40}$$

$$\equiv ((\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\, \_, \texttt{zip}_X\ m\ (\texttt{tl}\ x, \texttt{tl}\ y))), (\lambda\, i, (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\, \_, \texttt{zip}_\pi\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)\ i)) \tag{7.41}$$

Complete this proof

7.2.6 Examples of Fixed Points

7.2.6.1 Zeros

Let us try to define the zero stream, we do this by lifting the functions

$$\texttt{const}_\mathbb{X}\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \mathbf{1} & n = 0 \\ (c, \lambda\, \_, \texttt{const}_\mathbb{X}\ m\ c) & n = m+1 \end{cases} \tag{7.42}$$

$$\texttt{const}_\pi\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \texttt{refl} & n = 0 \\ \lambda\, i, (c, \lambda\, \_, \texttt{const}_\pi\ m\ c\ i) & n = m+1 \end{cases} \tag{7.43}$$

to get the definition of zero stream

$$\texttt{zeros} := \texttt{lift}_\mathbb{M}\ \texttt{const}_\mathbb{X}\ \texttt{const}_\pi\ 0. \tag{7.44}$$

We want to show that we get the expected properties, such as

$$\texttt{hd}\ \texttt{zeros} \equiv 0 \tag{7.45}$$

$$\texttt{tl}\ \texttt{zeros} \equiv \texttt{zeros} \tag{7.46}$$

7.2.6.2 Spin

We want to define spin, as being the fixed point $\texttt{spin} = \texttt{later spin}$, so that is again a final coalgebra, but of a $\mathbb{M}$-type (which is a final coalgebra)

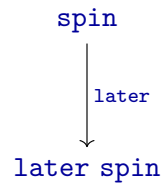

Figure 7.4: TODO

Since it is final, it also must be unique, meaning that there is just one program that spins forever, without returning a value, meaning every other program must return a value. If we just

Chapter 8

Conclusion

conclude on the problem statement from the introduction

Bibliography

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

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

[3] Jesse Hughes. *A study of categories of algebras and coalgebras.* PhD thesis, Carnegie Mellon University, 2001.

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

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

[6] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[7] Amin Timany and Matthieu Sozeau. Cumulative inductive types in coq. *LIPIcs: Leibniz International Proceedings in Informatics*, 2018.

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

[9] 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.

Appendix A

Additions to the Cubical Agda Library

Appendix B

The Technical Details