$\mathtt{M}$-types and Coinduction in HoTT and Cubical Type Theory

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science

May 18, 2020

Advisor: Bas Spitters

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

in English…

Resumé

in Danish. . .

Acknowledgments

. . .

c

Lasse Letager Hansen,
Aarhus, May 18, 2020.

Contents

Chapter 1

Introduction

This work tries to formalize co-inductive types in the setting of homotopy type theory.

motivate and explain the problem to be addressed

example of a citation: [5]

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

Chapter 2

Notation

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 $\mathtt{f} : A \to C$ better description, not always a function
- A destructor $\mathtt{f} : A \to C$ better description, not always a function
- 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 $x \mathrel{\mathtt{R}} y$.
- The unit type is $\mathbf{1}$ while the empty type is $\mathbf{0}$.
- A functor $\mathtt{P}$
- A container is denoted as S or $(A , \mathtt{B})$
- A coalgebra C-γ
- We denote the function giving the first and second projection of a dependent pair by π_1 and π_2.

Chapter 3

Background Theory

3.1 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.

3.2 Homotopy Type Theory (HoTT)

Homotopy type theory

3.2.1 The HoTT Book

3.3 Cubical Type Theory

3.4 Cubical Agda

Axioms of cubical Agda

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

Something about the interval type!!

Chapter 4

$\mathtt{M}$-types

4.1 Containers / Signatures

In this section we will introduce containers (also known as signatures), and show how to use these to construct a coalgebra.

Definition 4.1.1. A Container (or signature) is a dependent pair $S = (A, \mathtt{B})$ for the types $A : \mathcal{U}$ and $\mathtt{B} : A \to \mathcal{U}$.

From a container we can define a polynomial functor.

Definition 4.1.2. A polynomial functor is defined for objects (types) as

$$
\begin{aligned}
\mathtt{P}_S &: \mathcal{U} \to \mathcal{U} \\
\mathtt{P}(X) := \mathtt{P}_S(X) &= \sum_{a:A} \mathtt{B}(a) \to X
\end{aligned}
\tag{4.1}
$$

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

$$
\begin{aligned}
\mathtt{Pf} &: \mathtt{P}X \to \mathtt{P}Y \\
\mathtt{Pf}(a, \mathtt{g}) &= (a, \mathtt{f} \circ \mathtt{g}).
\end{aligned}
\tag{4.2}
$$

Using these definitions we can now define the polynomial functor used to construct the type of streams.

Example 4.1.1. The type for streams over the type A is defined by the container $S = (A, \lambda\_, \mathbf{1})$, applying the polynomial functor for the container S, we get

$$\mathtt{P}_S(X) = \sum_{a:A} \mathbf{1} \to X. \tag{4.3}$$

Since we are working in a Category 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 this will be equivalent to the definition

$$\mathtt{P}_S(X) = A \times X. \tag{4.4}$$

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

Definition 4.1.3. A $\mathtt{P}$-coalgebra is defined as

$$\mathtt{Coalg}_S = \sum_{C:\mathcal{U}} C \to \mathtt{P}C. \tag{4.5}$$

We denote a $\mathtt{P}$-coalgebra give by C and γ as C-γ. The coalgebra morphisms are defined as

$$\begin{gathered} \cdot \Rightarrow \ \cdot : \mathtt{Coalg}_S \to \mathtt{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta = \sum_{\mathtt{f}:C\to D} \delta \circ \mathtt{f} = \mathtt{Pf} \circ \gamma \end{gathered} \tag{4.6}$$

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

Definition 4.1.4. Given a container S, we define $\mathtt{M}$-types, as the type $\mathtt{M}_S$, making the coalgebra given by $\mathtt{M}_S$ and $\mathtt{out} : \mathtt{M}_S \to \mathtt{P}_S(\mathtt{M}_S)$ fulfill the property

$$\mathtt{Final}_S := \sum_{(X\text{-}\rho:\mathtt{Coalg}_S)} \prod_{(C\text{-}\gamma:\mathtt{Coalg}_S)} \mathtt{isContr}\ (C\text{-}\gamma \Rightarrow X\text{-}\rho). \tag{4.7}$$

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 an $\mathtt{M}$-type for streams.

Example 4.1.2. Given the polynomial functor $\mathtt{P}_{(A\,,\,\lambda\,\_,\mathbf{1})}\mathtt{M} = A \times \mathtt{M}_{(A\,,\,\lambda\,\_,\mathbf{1})}$ for streams, we get the diagram in Figure 4.1, where $\mathtt{out}$ is an isomorphism (because of the finality of the coalgebra), with

$$\begin{tikzcd} A & A \times \mathtt{M}_{(A\,,\,\lambda\,\_,\mathbf{1})} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \mathtt{M}_{(A\,,\,\lambda\,\_,\mathbf{1})} \\ & \mathtt{M}_{(A\,,\,\lambda\,\_,\mathbf{1})} \arrow[ul, "\mathtt{hd}"] \arrow[u, "\mathtt{out}"'] \arrow[ur, "\mathtt{tl}"'] & \end{tikzcd}$$

Figure 4.1: $\mathtt{M}$-types of streams

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

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{hd}\ s : A}\ \mathtt{E}_{\mathtt{hd}} \tag{4.8}$$

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{tl}\ s : \mathtt{stream}\ A}\ \mathtt{E}_{\mathtt{tl}} \tag{4.9}$$

$$\frac{A : \mathcal{U} \quad x : A \quad xs : \mathtt{stream}\ A}{\mathtt{cons}\ x\ xs : \mathtt{stream}\ A}\ \mathtt{I}_{\mathtt{cons}} \tag{4.10}$$

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

Definition 4.1.5. We define a chain as a famililly of cones $\pi_{(n)} : X_{n+1} \to X_n$, over a family of types X_n. .

cocones?

define chains, $\pi_{(n)}$ and X_n

$$\begin{tikzcd} X_0 & X_1 \arrow[l, "\pi_{(0)}"'] & \cdots \arrow[l, "\pi_{(1)}"'] & X_n \arrow[l, "\pi_{(n-1)}"'] & X_{n+1} \arrow[l, "\pi_{(n)}"'] & \cdots \arrow[l, "\pi_{(n+1)}"'] \end{tikzcd}$$

$$\begin{tikzcd} X_0 & X_1 \arrow[l, "\pi_{(0)}"'] & \cdots \arrow[l, "\pi_{(1)}"'] & X_n \arrow[l, "\pi_{(n-1)}"'] & X_{n+1} \arrow[l, "\pi_{(n)}"'] & \cdots \arrow[l, "\pi_{(n+1)}"'] \\ & & C \arrow[ull, "\mathtt{f}_0"] \arrow[ul, "\mathtt{f}_1"] \arrow[u, phantom, "\dots"] \arrow[ur, "\mathtt{f}_n"'] \arrow[urr, "\mathtt{f}_{n+1}"'] & & \dots & \end{tikzcd}$$

Lemma 4.1.1. *For all coalgebras C-γ for the container S, we get $C \to \mathtt{M}_S \equiv \mathtt{Cone}_{C\text{-}\gamma}$, where* $\mathtt{Cone} = \sum_{(\mathtt{f}:\prod_{(n:\mathbb{N})} C \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ (\mathtt{f}_{(n+1)}) \equiv f_n$

Proof.

Complete proof

∎

Lemma 4.1.2. *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 equivalence

$$\mathtt{fun}_{\mathcal{L}collapse}\ (x, r) = x_0 \tag{4.11}$$

$$\mathtt{inv}_{\mathcal{L}collapse}\ x_0 = (\lambda\, n,\ \ell^n\ x_0)\ ,\ (\lambda\, n,\ \mathtt{refl}_{(\ell^{(n+1)}\ x_0)}) \tag{4.12}$$

$$\mathtt{rinv}\ x_0 = \mathtt{refl}_{x_0} \tag{4.13}$$

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 $\mathtt{in}$ and $\mathtt{out}$.

Theorem 4.1.3. *Given the container $(A\,, \mathtt{B})$ we define the equality*

$$shift : \mathcal{L} \equiv \mathtt{P}\mathcal{L} \tag{4.14}$$

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

$$\mathtt{in} = \mathtt{transport}\ shift \tag{4.15}$$

$$\mathtt{out} = \mathtt{transport}\ (shift^{-1}). \tag{4.16}$$

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^{\mathtt{P}} \equiv \mathtt{P}\mathcal{L} \tag{4.17}$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^{\mathtt{P}} \tag{4.18}$$

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

$$\mathtt{fun}_{\mathcal{L}unique}\ (\mathsf{a}, \mathsf{b}) = \llparenthesis\ \mathtt{tt}\ ,\ \mathtt{a}\ \rrparenthesis, \llparenthesis\ \mathtt{refl}_{\mathtt{tt}}\ ,\ \mathtt{b}\ \rrparenthesis \tag{4.19}$$

$$\mathtt{inv}_{\mathcal{L}unique}\ (\mathsf{a}, \mathsf{b}) = \mathsf{a} \circ \mathtt{incr}\ ,\ b \circ \mathtt{incr} \tag{4.20}$$

$$\mathtt{rinv}_{\mathcal{L}unique}\ (\mathsf{a}, \mathsf{b}) = \mathtt{refl}_{(\mathsf{a},\mathsf{b})} \tag{4.21}$$

$$\mathtt{linv}_{\mathcal{L}unique}\ (\mathsf{a}, \mathsf{b}) = \mathtt{refl}_{(\mathsf{a},\mathsf{b})} \tag{4.22}$$

The definition of α is then,

$$\begin{aligned}
\mathcal{L}^{\mathtt{P}} &\equiv \sum_{(x:\prod_{(n:\mathbb{N})} \sum_{(a:A)} \mathtt{B}\,a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n+1)}\ x_{n+1} \equiv x_n && (4.23)\\
&\equiv \sum_{(x:\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\,(\pi_1\,x)_n \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv_* \mathtt{u}_n && (4.24)\\
&\equiv \sum_{(a:A)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\,a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv \mathtt{u}_n && (4.25)\\
&\equiv \sum_{a:A} \mathtt{B}\ a \to \mathcal{L} && (4.26)\\
&\equiv \mathtt{P}\mathcal{L} && (4.27)
\end{aligned}$$

To collapse $\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n$ to A between (4.24) and (4.25) we use Lemma 4.1.2 . We use Lemma 4.1.1 for the equality between (4.25) and (4.26). The rest of the equalities are given by a simple isomorphism or by definition. The definition of $shift$ is

$$shift = \alpha^{-1} \cdot \mathcal{L}unique. \tag{4.28}$$

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. $\square$

4.2 Coinduction Principle for M-types

We can now construct a coinduction principle given a bisimulation relation

Definition 4.2.1. 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 $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \to \mathtt{P}(\overline{\mathcal{R}})$ forms a P-coalgebra $\overline{\mathcal{R}}$-$\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, making the diagram in Figure 4.2 commute ($\Longrightarrow$ represents P-coalgebra morphisms).

$$C\text{-}\gamma \xLeftarrow{\pi_1{}^{\overline{\mathcal{R}}}} \overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} \xRightarrow{\pi_2{}^{\overline{\mathcal{R}}}} C\text{-}\gamma$$

Figure 4.2: Bisimulation for a coalgebra

Definition 4.2.2 (Coinduction principle)**.** Given a relation $\mathcal{R}$, that is part of a bisimulation over a final P-coalgebra $\mathtt{M}$-$\mathtt{out} : \mathtt{Coalg}_S$ we get the diagram in Figure 4.3,
where $\pi_1^{\overline{\mathcal{R}}} = ! = \pi_2^{\overline{\mathcal{R}}}$, which means given $r : m\ \mathcal{R}\ m'$ we get the equation

$$m = \pi_1^{\overline{\mathcal{R}}}(m, m', r) = \pi_2^{\overline{\mathcal{R}}}(m, m', r) = m'. \tag{4.29}$$

$$\texttt{M-out} \xLeftarrow{\pi_1{}^{\overline{\mathcal{R}}}} \overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} \xRightarrow{\pi_2{}^{\overline{\mathcal{R}}}} \texttt{M-out}$$

Figure 4.3: Bisimulation principle for final coalgebra

Chapter 5

Instantiation of M-types

5.1 Stream Formalization using M-types

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

$$\texttt{stream}\ A := \texttt{M}_{(A\,,\,\lambda\_,\mathbf{1})} \tag{5.1}$$

When taking the head of a stream, we get

$$\begin{aligned} \texttt{hd}\ (\texttt{cons}\ x\ xs) &\equiv \pi_1\ \texttt{out}\ (\texttt{cons}\ x\ xs) && (5.2)\\ &\equiv \pi_1\ \texttt{out}\ (\texttt{in}\ (x, \lambda\_, xs)) && (5.3)\\ &\equiv \pi_1\ (x, \lambda\_, xs) && (5.4)\\ &\equiv x && (5.5) \end{aligned}$$

and similarly for the tail of the stream

$$\begin{aligned} \texttt{tl}\ (\texttt{cons}\ x\ xs) &\equiv \pi_2\ \texttt{out}\ (\texttt{cons}\ x\ xs) && (5.6)\\ &\equiv \pi_2\ \texttt{out}\ (\texttt{in}\ (x, \lambda\_, xs)) && (5.7)\\ &\equiv \pi_2\ (x, \lambda\_, xs) && (5.8)\\ &\equiv xs && (5.9) \end{aligned}$$

and the other direction is also true

$$\begin{aligned} \texttt{cons}(\texttt{hd}\ s, \texttt{tl}\ s) &\equiv \texttt{in}\ (\texttt{hd}\ s, \texttt{tl}\ s) && (5.10)\\ &\equiv \texttt{in}\ (\pi_1\ (\texttt{out}\ s), \pi_2\ (\texttt{out}\ s)) && (5.11)\\ &\equiv \texttt{in}\ (\texttt{out}\ s) && (5.12)\\ &\equiv s. && (5.13) \end{aligned}$$

When forming elements of the M-type, we want to do it by lifting it though the definition of the M-type, meaning we want to define a function $\texttt{cons}' : (\mathbb{N} \to A) \to \texttt{stream}\ A$ as

$$\texttt{cons}' f = \texttt{lift}_{\texttt{M}}\ (\lambda\, c\, n, f\ \ c) \tag{5.14}$$

$$\texttt{cons}' f = \texttt{lift}_{\texttt{M}}\ (\lambda\, c\, n, f\ \ c) \tag{5.15}$$

5.2 ITrees as $\mathtt{M}$-types

We want the following rules for ITrees

$$\frac{r : R}{\mathtt{Ret}\ r : \mathtt{itree}\ \mathtt{E}\ R}\ \mathtt{I}_{\mathtt{Ret}} \tag{5.16}$$

$$\frac{A : \mathcal{U} \quad a : \mathtt{E}\ A \quad f : A \to \mathtt{itree}\ \mathtt{E}\ R}{\mathtt{Vis}\ a\ f : \mathtt{itree}\ \mathtt{E}\ R}\ \mathtt{I}_{\mathtt{Vis}}. \tag{5.17}$$

Elimination rules

$$\frac{t : \mathtt{itree}\ \mathtt{E}\ R}{\mathtt{Tau}\ t : \mathtt{itree}\ \mathtt{E}\ R}\ \mathtt{E}_{\mathtt{Tau}}. \tag{5.18}$$

5.2.1 Delay Monad

We start by looking at ITrees without the $\mathtt{Vis}$ constructor, this type is also know as the delay monad

check this statement

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

$$\mathtt{P}_S(X) = \sum_{x : \mathbf{1} + R} \lambda\{\mathtt{inl}\ \_ \to \mathbf{1}; \mathtt{inr}\ \_ \to \mathbf{0}\}\ x \to X, \tag{5.19}$$

which is equal to

$$\mathtt{P}_S(X) = X + R \times (\mathbf{0} \to X). \tag{5.20}$$

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

$$\mathtt{P}_S(X) = X + R \tag{5.21}$$

meaning we get the following diagram.

$$\begin{tikzcd} R \arrow[r, "\mathtt{inl}"] \arrow[dr, "\mathtt{now}"'] & R + \mathtt{M} & \mathtt{M} \arrow[l, "\mathtt{inr}"'] \arrow[dl, "\mathtt{later}"] \\ & \mathtt{M} \arrow[u, "\mathtt{out}"'] & \end{tikzcd}$$

Figure 5.1: Delay monad

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

(Later = Tau, Ret = Now)

5.2.2 Tree

Now lets look at the example where we remove the $\mathtt{Tau}$ constructor. We let

$$S = \left(R + \sum_{A:\mathcal{U}} \mathtt{E}\ A\ ,\ \lambda\{\mathtt{inl}\ \_ \to \mathbf{0}\ ;\ \mathtt{inr}\ (A, e) \to A\}\right). \tag{5.22}$$

This will give us the polynomial functor

$$\texttt{P}_S(X) = \sum_{x : R + \sum_{A:\mathcal{U}} \texttt{E}\ A} \lambda\{\texttt{inl}\ \_ \to \mathbf{0}\ ;\ \texttt{inr}\ (A, e) \to A\}\ x \to X, \tag{5.23}$$

which simplifies to

$$\texttt{P}_S(X) = (R \times (\mathbf{0} \to X)) + (\sum_{A:\mathcal{U}} \texttt{E}\ A \times (A \to X)), \tag{5.24}$$

and further

$$\texttt{P}_S(X) = R + \sum_{A:\mathcal{U}} \texttt{E}\ A \times (A \to X). \tag{5.25}$$

We get the following diagram for the P-coalgebra.

$$\begin{tikzcd}
R \arrow[r, "\texttt{inl}"] \arrow[dr, "\texttt{Ret}"'] & R + \sum_{A:U} \texttt{E}\ A \times (A \to \texttt{M}) & \sum_{A:U} \texttt{E}\ A \times (A \to \texttt{M}) \arrow[l, "\texttt{inr}"'] \arrow[dl, "\texttt{Vis}"] \\
& \texttt{M} \arrow[u, "\texttt{out}"] &
\end{tikzcd}$$

Figure 5.2: TODO

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

5.2.3 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

$$S = \left(\mathbf{1} + R + \sum_{A:\mathcal{U}} (\texttt{E}\ A)\ \ ,\ \ \lambda\,\{\texttt{inl}\ (\texttt{inl}\ \_) \to \mathbf{1}\ ;\ \texttt{inl}\ (\texttt{inr}\ \_) \to \mathbf{0}\ ;\ \texttt{inr}(A, \_) \to A\}\right). \tag{5.26}$$

Such that the (reduced) polynomial functor becomes

$$\texttt{P}_S(X) = X + R + \sum_{A:\mathcal{U}} ((\texttt{E}\ A) \times (A \to X)) \tag{5.27}$$

Giving us the diagram

5.3 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 5.4

$$
\begin{tikzcd}
R \arrow[r, "\mathtt{inl} \circ \mathtt{inr}"] \arrow[ddrr, "\mathtt{Ret}"'] & \mathtt{M} + R + \sum_{A:\mathcal{U}}(\mathtt{E}\ A \times (A \to \mathtt{M})) & & \sum_{A:U} \mathtt{E}\ A \times (A \to \mathtt{M}) \arrow[ll, "\mathtt{inr}"'] \arrow[ddl, "\mathtt{Vis}"] \\
& & \mathtt{M} \arrow[ul, "\mathtt{inl} \circ \mathtt{inl}"'] \arrow[d, "\mathtt{Tau}"'] & \\
& & \mathtt{M} \arrow[uul, "\mathtt{out}"] &
\end{tikzcd}
$$

Figure 5.3: TODO

$$
\begin{tikzcd}
V \times (V \to \alpha \to V) \arrow[r] \arrow[dr, "build"'] & M & M \arrow[dl, "step"] \\
& M &
\end{tikzcd}
$$

Figure 5.4: automaton

Chapter 6

QM-types

6.1 Quotienting and constructors

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

6.2 Quotient `M`-type

We want to construct a quotient `M`-type, and we know that `M`-types are an algebraic theory? Meaning we want to define quotient algebra...

We want to construct a quotiented M type, which is given as a final bisimulation and a final coalgebra, and relations between them. This is a special case for a cofree coalgebra, namely starting at $X = \mathbf{1}$.

Since we know that `M`-types preserves the H-level, we can use set-truncated quotients, to define quotient `M`-types, for examples we can define weak bisimulation of the delay monad ...

Quotients of the delay monad

6.3 Quotient inductive-inductive types (QIITs)

"A quotient inductive-inductive type (QIIT) can be seen as a multi-sorted algebraic theory where sorts can be indexed over each other" - "Constructing Quotient Inductive-Inductive Types"

"W-types can be seen informally as the free algebras for signatures with operations of possibly infinite arity, but no equations." – `https://arxiv.org/pdf/1201.3898.pdf`

A quotient inductive-inductive type (QIIT) is a type together with a relation defined on that type, and then quotiented by that relation.

What is a QIIT concretely?

6.4 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 6.4.1 (Partiality Monad)**.** A simple example of a quotient inductive-inductive type is the partiality monad $(-)_\perp$ over a type R, defined by the constructors

$$R_\perp : \mathcal{U} \tag{6.1}$$

$$\perp : R_\perp \tag{6.2}$$

$$\frac{a : R}{\eta\ a : R_\perp} \tag{6.3}$$

Should I define what it means to be an ordering relation separately, and just say the relation here is an instance of that? (Generalize?)

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

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\bigsqcup (\mathtt{s}, \mathtt{b}) : R_\perp} \tag{6.4}$$

$$\frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp\ p\ q : x \equiv y} \tag{6.5}$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{refl}} \tag{6.6}$$

$$\frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z}\ \sqsubseteq_{\mathtt{trans}} \tag{6.7}$$

$$\frac{x : R_\perp}{\perp \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{never}} \tag{6.8}$$

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \bigsqcup(\mathtt{s}, \mathtt{b})} \tag{6.9}$$

$$\frac{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp x}{\bigsqcup(\mathtt{s}, \mathtt{b}) \sqsubseteq_\perp x} \tag{6.10}$$

and finally set truncated

$$\frac{p, q : x \sqsubseteq_\perp y}{p \equiv q}\ (-)_\perp\text{-}\mathtt{isSet} \tag{6.11}$$

6.4.1 Delay monad to Sequences

Introduce the delay monad before this section!!

Definition 6.4.2. We define

$$\mathtt{Seq}_R = \sum_{(\mathtt{s}:\mathbb{N}\to R+1)} \mathtt{isMon}\ \mathtt{s} \tag{6.12}$$

where

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \equiv \mathtt{inr}\ \mathtt{tt}) \times (\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \mathtt{tt})) \tag{6.13}$$

meaning a sequences is $\mathtt{inr}\ \mathtt{tt}$ 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 $\mathtt{inr}\ \mathtt{tt}$.

For each index in a sequence, the element at that index $\mathtt{s}_n$ is either not terminated $\mathtt{s}_n \equiv \mathtt{inr}\ \mathtt{tt}$, 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 $\mathtt{s}_n \not\equiv \mathtt{inr}\ \mathtt{tt}$. Thus we can write $\mathtt{isMon}$ as

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \uparrow_{R+1}) \times (\mathtt{s}_{n+1} \downarrow_{R+1})) \tag{6.14}$$

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

$$\mathtt{now}_{Seq}\ r = (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{6.15}$$

$$\mathtt{never}_{Seq} = (\lambda\_, \mathtt{inr}\ \mathtt{tt}), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{6.16}$$

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

Definition 6.4.3. We can shift a sequence $(\mathtt{s}, \mathtt{q})$ by inserting an element (and an equality) (z_s, z_q) at $n = 0$,

$$\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ (z_s, z_q) = \begin{cases} z_s & n = 0 \\ \mathtt{s}_m & n = m + 1 \end{cases}, \begin{cases} z_q & n = 0 \\ \mathtt{q}_m & n = m + 1 \end{cases}, \tag{6.17}$$

Definition 6.4.4. We can unshift a sequence by removing the first element of the sequence

$$\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{s} \circ \mathtt{suc}, \mathtt{q} \circ \mathtt{suc}. \tag{6.18}$$

Lemma 6.4.1. *The function*

$$\mathtt{shift\text{-}unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{shift}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}))\ (\mathtt{s}_0, \mathtt{q}_0) \tag{6.19}$$

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 6.4.2. *The function*

$$\mathtt{unshift\text{-}shift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{unshift}\ (\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ \_) \tag{6.20}$$

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 $\mathtt{Seq}_R$, 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 6.4.3 ($\mathtt{inl}{\not\equiv}\mathtt{inr}$)**.** *For any two elements* $x = \mathtt{inl}\ a$ *and* $y = \mathtt{inr}\ b$ *then* $x \not\equiv y$.

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

better formulated proof

Definition 6.4.5. We define function from `Delay` R to $\mathtt{Seq}_R$

$$\begin{aligned} &\mathtt{Delay{\to}Seq}\ (\mathtt{now}\ r) = \mathtt{now}_{Seq}\ r \\ &\mathtt{Delay{\to}Seq}\ (\mathtt{later}\ x) = \\ &\qquad \mathtt{shift}\ (\mathtt{Delay{\to}Seq}\ x) \left(\mathtt{inr}\ \mathtt{tt}, \begin{cases} \mathtt{inr}\ (\mathtt{refl}, \mathtt{inl}{\not\equiv}\mathtt{inr}) & x = \mathtt{now}\ \_ \\ \mathtt{inl}\ \mathtt{refl} & x = \mathtt{later}\ \_ \end{cases} \right) \end{aligned} \tag{6.21}$$

Definition 6.4.6. We define function from $\mathtt{Seq}_R$ to `Delay` R

$$\mathtt{Seq{\to}Delay}\ (\mathtt{g}, \mathtt{q}) = \begin{cases} \mathtt{now}\ r & \mathtt{g}_0 = \mathtt{inl}\ r \\ \mathtt{later}\ (\mathtt{Seq{\to}Delay}\ (\mathtt{unshift}\ (\mathtt{g}, \mathtt{q}))) & \mathtt{g}_0 = \mathtt{inr}\ \mathtt{tt} \end{cases} \tag{6.22}$$

Theorem 6.4.4. *The type* $\mathtt{Seq}_R$ *is equal to* `Delay` R

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

$$\texttt{Delay}{\to}\texttt{Seq}\ (\texttt{Seq}{\to}\texttt{Delay}\ (\mathtt{g}, \mathtt{q})) \equiv (\mathtt{g}, \mathtt{q}) \tag{6.23}$$

defined as

$$\texttt{Seq-Delay}\ (\mathtt{g}, \mathtt{q}) = \begin{cases} \texttt{refl} & \mathtt{g}_0 = \texttt{inl}\ r \\ \texttt{ap}\ \texttt{shift}\ (\texttt{Seq-Delay}\ (\texttt{unshift}\ (\mathtt{g}, \mathtt{q}))) & \mathtt{g}_0 = \texttt{inr}\ \texttt{tt} \\ \quad \cdot\ \texttt{shift-unshift}\ (\mathtt{g}, \mathtt{q}) & \end{cases} \tag{6.24}$$

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

and left identity, saying that for any delay monad t we get $\texttt{Seq}{\to}\texttt{Delay}\ (\texttt{Delay}{\to}\texttt{Seq}\ t) \equiv t$, defined as

$$\begin{aligned} &\texttt{Delay-Seq}\ (\texttt{now}\ a) = \texttt{refl} \\ &\texttt{Delay-Seq}\ (\texttt{later}\ x) = \texttt{ap}\ (\texttt{later} \circ \texttt{Seq}{\to}\texttt{Delay})\ (\texttt{unshift-shift}\ (\texttt{Delay}{\to}\texttt{Seq}\ x)) \\ &\qquad\qquad \cdot\ \texttt{ap}\ \texttt{later}\ (\texttt{Delay}{\to}\texttt{Seq}\ x) \end{aligned} \tag{6.25}$$

□

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

Proof. □

6.4.2 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 6.4.7 (Sequence Termination)**.** The following relations says that a sequence $(\mathtt{s}, \mathtt{q}) : \texttt{Seq}_R$ terminates with a given value $r : R$,

$$(\mathtt{s}, \mathtt{q}) \downarrow_{\texttt{Seq}} r = \sum_{(n:\mathbb{N})} \mathtt{s}_n \downarrow_{R+\mathbf{1}} r. \tag{6.26}$$

Definition 6.4.8 (Sequence Ordering)**.**

$$(s, q) \sqsubseteq_{\texttt{Seq}} (t, p) = \prod_{(a:R)} \left(\|s \downarrow_{\texttt{Seq}} a\| \to \|t \downarrow_{\texttt{Seq}} a\|\right) \tag{6.27}$$

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

propositional or set truncation?

Definition 6.4.9. There is a conversion from $R + \mathbf{1}$ to the partiality monad $R_\perp$

$$\begin{aligned} \texttt{Maybe}{\to}(-)_\perp\ (\texttt{inl}\ r) &= \eta\ r \\ \texttt{Maybe}{\to}(-)_\perp\ (\texttt{inr}\ \texttt{tt}) &= \perp \end{aligned} \tag{6.28}$$

Definition 6.4.10 (Maybe Ordering)**.** Given some $x, y : R + \mathbf{1}$, the ordering relation is defined as

$$x \sqsubseteq_{\mathrm{R}+\mathbf{1}} y = (x \equiv y) + ((x \downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}})) \tag{6.29}$$

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

$$\texttt{isMon s} = \prod_{(n:\mathbb{N})} \texttt{s}_n \sqsubseteq_{\texttt{R+1}} \texttt{s}_{n+1} \tag{6.30}$$

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 6.4.5. *The function* $\texttt{Maybe}{\rightarrow}(-)_\perp$ *is monotone, that is, if* $x \sqsubseteq_{\texttt{A+1}} y$*, for some* x *and* y*, then* $(\texttt{Maybe}{\rightarrow}(-)_\perp\ x) \sqsubseteq_\perp (\texttt{Maybe}{\rightarrow}(-)_\perp\ y)$*.*

Proof. We do the proof by case.

$$\begin{aligned}
&\texttt{Maybe}{\rightarrow}(-)_\perp\texttt{-mono}\ (\texttt{inl}\ p) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\rightarrow}(-)_\perp\ x \sqsubseteq_\perp \texttt{Maybe}{\rightarrow}(-)_\perp\ a)\ p\ (\sqsubseteq_{\texttt{refl}}(\texttt{Maybe}{\rightarrow}(-)_\perp\ x)) \\
&\texttt{Maybe}{\rightarrow}(-)_\perp\texttt{-mono}\ (\texttt{inr}\ (p, _)) = \\
&\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\rightarrow}(-)_\perp\ a \sqsubseteq_\perp \texttt{Maybe}{\rightarrow}(-)_\perp\ y)\ p^{-1}\ (\sqsubseteq_{\texttt{never}}(\texttt{Maybe}{\rightarrow}(-)_\perp\ y))
\end{aligned} \tag{6.31}$$

□

Definition 6.4.11. There is a function taking a sequence to an increasing sequence

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

$$\begin{aligned}
&\texttt{Seq}{\rightarrow}\texttt{incSeq} \\
&\texttt{Seq}{\rightarrow}\texttt{incSeq}\ (\texttt{g}, \texttt{q}) = \texttt{Maybe}{\rightarrow}(-)_\perp \circ \texttt{g}, \texttt{Maybe}{\rightarrow}(-)_\perp\texttt{-mono} \circ \texttt{q}
\end{aligned} \tag{6.32}$$

Definition 6.4.12. There is a function taking a sequence to the partiality monad

$$\begin{aligned}
&\texttt{Seq}{\rightarrow}(-)_\perp : \texttt{Seq}_A \rightarrow A_\perp \\
&\texttt{Seq}{\rightarrow}(-)_\perp\ (\texttt{g}, \texttt{q}) = \bigsqcup \circ\ \texttt{Seq}{\rightarrow}\texttt{incSeq}
\end{aligned} \tag{6.33}$$

Lemma 6.4.6. *The function* $\texttt{Seq}{\rightarrow}(-)_\perp$ *is monotone.*

$$\texttt{Seq}{\rightarrow}(-)_\perp\texttt{-mono} : \texttt{isSet} A \rightarrow (x\ y : \texttt{Seq}_A) \rightarrow x \sqsubseteq_{\texttt{seq}} y \rightarrow \texttt{Seq}{\rightarrow}(-)_\perp\ x \sqsubseteq_\perp \texttt{Seq}{\rightarrow}(-)_\perp\ y \tag{6.34}$$

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 6.4.13. If two sequences x, y are weakly bisimular, then $\texttt{Seq}{\rightarrow}(-)_\perp\ x \equiv \texttt{Seq}{\rightarrow}(-)_\perp\ y$

$$\texttt{Seq}{\rightarrow}(-)_\perp\texttt{-}{\approx}{\rightarrow}{\equiv}\ A_{set}\ x\ y\ (p, q) = \alpha_\perp\ (\texttt{Seq}{\rightarrow}(-)_\perp\texttt{-mono}\ A_{set}\ x\ y\ p)\ (\texttt{Seq}{\rightarrow}(-)_\perp\texttt{-mono}\ A_{set}\ y\ x\ q) \tag{6.35}$$

Definition 6.4.14 (Recursor for Quotient)**.** For all sequences $x, y : \texttt{Seq}_A$, functions $\texttt{f} : A \rightarrow B$ and relations $\texttt{g} : x\ \texttt{R}\ y \rightarrow \texttt{f}\ x \equiv \texttt{f}\ y$, then if B is a set $B_{set} : \texttt{isSet}\ B$, we get a function $\texttt{rec} : A/\texttt{R} \rightarrow B$, defined by case as

$$\begin{aligned}
&\texttt{rec}\ [\,z\,] = \texttt{f}\ z \\
&\texttt{rec}\ (\texttt{eq/}\ _\ _\ r\ i) = \texttt{g}\ r\ i \\
&\texttt{rec}\ (\texttt{squash/}\ a\ b\ p\ q\ i\ j) = B_{set}\ (\texttt{rec}\ a)\ (\texttt{rec}\ b)\ (\texttt{ap}\ \texttt{rec}\ p)\ (\texttt{ap}\ \texttt{rec}\ q)\ i\ j
\end{aligned} \tag{6.36}$$

This recursor allows us to lift the function $\texttt{Seq}{\rightarrow}(-)_\bot$ to the quotient

Definition 6.4.15. We can define a function $\texttt{Seq}/{\sim}{\rightarrow}(-)_\bot$ from $\texttt{Seq}_A$ to $A_\bot$, where $A_{set} : \texttt{isSet}\ A$ as

$$\texttt{Seq}/{\sim}{\rightarrow}(-)_\bot = \texttt{rec}\ \texttt{Seq}{\rightarrow}(-)_\bot\ (\texttt{Seq}{\rightarrow}(-)_\bot\text{-}{\approx}{\rightarrow}{\equiv}\ A_{set})\ (-)_\bot\text{-}\texttt{isSet} \tag{6.37}$$

Lemma 6.4.7. *Given two sequences s and t, if $\texttt{Seq}{\rightarrow}(-)_\bot\ s \equiv \texttt{Seq}{\rightarrow}(-)_\bot\ t$, then $s \sim_{\texttt{seq}} t$.*

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

$$s \sim_{\texttt{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\texttt{seq}} r\| \rightarrow \|y \downarrow_{\texttt{seq}} r\|\right) \times \left(\prod_{(r:R)} \|y \downarrow_{\texttt{seq}} r\| \rightarrow \|x \downarrow_{\texttt{seq}} r\|\right) \tag{6.38}$$

so we can just show one part and get the other 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 $x \downarrow_{\texttt{seq}} r \rightarrow y \downarrow_{\texttt{seq}} r$. Since $x \downarrow_{\texttt{seq}} r$, then $\eta\ r \sqsubseteq_\bot \texttt{Seq}{\rightarrow}(-)_\bot\ x$, but we have assumed $\texttt{Seq}{\rightarrow}(-)_\bot\ x \equiv \texttt{Seq}{\rightarrow}(-)_\bot\ y$, so we get $\eta\ r \sqsubseteq_\bot \texttt{Seq}{\rightarrow}(-)_\bot\ y$, and thereby $y \downarrow_{\texttt{seq}} r$. ☐

Lemma 6.4.8. *The function $\texttt{Seq}/{\sim}{\rightarrow}(-)_\bot$ is injective.*

Should this be formalized?

Proof. We use propositional elimination of quotients

$$\begin{aligned}\texttt{elimProp} : (\texttt{B} : \texttt{Seq}_R/{\sim_{\texttt{seq}}} \rightarrow \mathcal{U}) \rightarrow ((x : \texttt{Seq}_R/{\sim_{\texttt{seq}}}) \rightarrow \texttt{isProp}\ (\texttt{B}\ x))\\ \rightarrow (\texttt{f} : (a : \texttt{Seq}_R) \rightarrow \texttt{B}\ [\ a\ ]) \rightarrow (x : \texttt{Seq}_R/{\sim_{\texttt{seq}}}) \rightarrow \texttt{B}\ x\end{aligned} \tag{6.39}$$

Convert to text, instead of a proof term!?

to show the injectivity, meaning for all $x\ y : \texttt{Seq}_R/{\sim_{\texttt{seq}}}$ we get $\texttt{Seq}/{\sim}{\rightarrow}(-)_\bot\ x \equiv \texttt{Seq}/{\sim}{\rightarrow}(-)_\bot\ y \rightarrow x \equiv y$. We start by eliminating x, followed by elimination of y, this gives us the proof term

$$\begin{aligned}&\texttt{elimProp}\\ &\quad(\lambda\, a,\ \texttt{Seq}/{\sim}{\rightarrow}(-)_\bot\ a \equiv \texttt{Seq}/{\sim}{\rightarrow}(-)_\bot\ y \rightarrow a \equiv y)\\ &\quad(\lambda\, a,\ \texttt{isProp}\Pi\ (\lambda\_, \texttt{squash/}\ a\ y))\\ &\quad(\lambda\, a,\ \texttt{elimProp}\\ &\qquad\quad(\lambda\, b,\ \texttt{Seq}{\rightarrow}(-)_\bot\ a \equiv \texttt{Seq}/{\sim}{\rightarrow}(-)_\bot\ b \rightarrow [\ a\ ] \equiv b)\\ &\qquad\quad(\lambda\, b,\ \texttt{isProp}\Pi\ (\lambda\_, \texttt{squash/}\ [\ a\ ]\ b))\\ &\qquad\quad(\lambda\, b,\ (\texttt{eq/}\ a\ b) \circ (\texttt{Seq}{\rightarrow}(-)_\bot\text{-}\texttt{isInjective}\ a\ b)))\end{aligned} \tag{6.40}$$

where $\texttt{Seq}{\rightarrow}(-)_\bot\text{-}\texttt{isInjective}$ is (6.4.7), ☐

Lemma 6.4.9. *For all constant sequences s, where all elements have the same value v, we get $\texttt{Seq}{\rightarrow}(-)_\bot\ s \equiv \texttt{Maybe}{\rightarrow}(-)_\bot\ v$.*

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

Lemma 6.4.10. *Assuming countable choice, the function $\texttt{Seq}{\rightarrow}(-)_\bot$ is surjective*

describe countable choice (and why it is needed!)

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

more precise description!

Complete the rest of the proof!

Proof. We do the proof by case on $R_\perp$, if it is $\eta\ r$ or $\mathtt{never}$, we convert them to the sequences $\mathtt{now}_{seq}\ r$ and $\mathtt{never}_{seq}$ respectively, then we are done by (6.4.9). For the least upper bound $\bigsqcup(\mathtt{s}, \mathtt{b})$, we translate to the (increasing) sequence, defined by $(\mathtt{s}, \mathtt{b})$. ☐

Lemma 6.4.11. *Assuming countable choice, the function* $\mathtt{Seq}/\!\sim\!\to\!(-)_\perp$ *is surjective*

Theorem 6.4.12. *Assuming countable choice, we get an equivalence between sequences and the partiality monad.*

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

Building the Partiality Monad as an M-type (Dialgebra?)

Is this possible?

6.4.3 Silhouette Trees

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

$$\left(R + \mathbf{1}, \begin{cases} \perp & \mathtt{inl}\ a \\ E & \mathtt{inr}\ \mathtt{tt} \end{cases}\right) \tag{6.41}$$

We get the constructors

$$\frac{a : R}{\mathtt{leaf}\ a : \mathtt{tree}\ R\ E} \tag{6.42}$$

$$\frac{\mathtt{k} : E \to \mathtt{tree}\ R\ E}{\mathtt{node}\ \mathtt{k} : \mathtt{tree}\ R\ E} \tag{6.43}$$

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

$$\frac{}{\mathtt{leaf}\ x \sim_{\mathtt{tree}} \mathtt{leaf}\ y}\ \sim_{\mathtt{leaf}} \tag{6.44}$$

$$\frac{\prod_{(v:E)}\ \mathtt{k}_1\ v \sim_{\mathtt{tree}} \mathtt{k}_2\ v}{\mathtt{node}\ \mathtt{k}_1 \sim_{\mathtt{tree}} \mathtt{node}\ \mathtt{k}_2}\ \sim_{\mathtt{node}} \tag{6.45}$$

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

add all needed constructors

$$\frac{}{\mathtt{leaf}_{\mathtt{sTree}} : \mathtt{sTree}\ E} \tag{6.46}$$

$$\frac{\mathtt{k} : E \to \mathtt{sTree}\ E}{\mathtt{node}_{\mathtt{sTree}}\ \mathtt{k} : \mathtt{sTree}\ E} \tag{6.47}$$

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

add all needed constructors

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.

Definition 6.4.16. We define monotone increasing sequences of trees as , all breanches are monotone increasing .

ordering is container ordering not maybe?

specify branches increasing?

$$\mathtt{Seq}_{tree} = \prod_{(\mathtt{e}:\mathbb{N}\to E)} \sum_{(\mathtt{s}:\mathbb{N}\to R+1)} \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_{\mathtt{R+1}} \mathtt{s}_{n+1} \tag{6.48}$$

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

where $\sqsubseteq_{\mathtt{R+1}}$ is similar to the relation defined at (6.4.10) .

Definition 6.4.17. We define a function to shift a $\mathtt{Seq}_{tree}$, it takes $\mathtt{f} : E \to \mathtt{Seq}_{tree}$ as an argument. We let $\mathtt{s}' = \mathtt{f}\ \mathtt{e}_0\ (\mathtt{e} \circ \mathtt{suc})$, then the definition is given as

$$\begin{aligned}
&\mathtt{shift\text{-}seq}\ \mathtt{f} = \\
&\quad \lambda\, \mathtt{e}, \llparenthesis\ \mathtt{inr}\ \mathtt{tt}\ ,\ \pi_1\ \mathtt{s}'\ \rrparenthesis, \left(\lambda\, n, \begin{cases} \mathtt{inr}\ (\mathtt{refl}, \mathtt{inl}\not\equiv\mathtt{inr}) & n = 0 \wedge \pi_1\ \mathtt{s}'\ 0 = \mathtt{inl}\ r \\ \mathtt{inl}\ \mathtt{refl} & n = 0 \wedge \pi_1\ \mathtt{s}'\ 0 = \mathtt{inr}\ \mathtt{tt} \\ \pi_2\ \mathtt{s}'\ m & n = m + 1 \end{cases}\right)
\end{aligned} \tag{6.49}$$

Definition 6.4.18. We define notation for

$$\llparenthesis\ x\ ,\ f\ \rrparenthesis = \lambda\, n, \begin{cases} x & n = 0 \\ f m & n = m + 1 \end{cases} \tag{6.50}$$

Definition 6.4.19. We define a function to unshift a Seq_{tree}

$$\mathtt{unshift\text{-}seq}\ \mathtt{s}\ v = \lambda\, \mathtt{e},\ (\pi_1\ (\mathtt{s}\ (\llparenthesis\ v\ ,\ \mathtt{e}\ \rrparenthesis)) \circ \mathtt{suc}), (\pi_2\ (\mathtt{s}\ (\llparenthesis\ v\ ,\ \mathtt{e}\ \rrparenthesis)) \circ \mathtt{suc}) \tag{6.51}$$

Lemma 6.4.13. *Shift and unshift er inverse to each other*

shift - unshift

Proof. The same reasoning as for ∎

Definition 6.4.20. We get a function from trees to monotone sequences

$$\begin{aligned}
\mathtt{tree{\to}Seq}\ (\mathtt{leaf}\ r) &= \lambda\, _, (\lambda\, _, \mathtt{inl}\ r), (\lambda\, _, \mathtt{inl}\ \mathtt{refl}) \\
\mathtt{tree{\to}Seq}\ (\mathtt{node}\ \mathtt{k}) &= \mathtt{shift}\ (\mathtt{tree{\to}Seq} \circ \mathtt{k})
\end{aligned} \tag{6.52}$$

Definition 6.4.21. We get a function from monotone sequences to trees

$$\mathtt{Seq{\to}tree}\ \mathtt{s} = \begin{cases} \mathtt{leaf}\ r & \prod_{(\mathtt{e}:\mathbb{N}\to E)} \pi_1\ (\mathtt{s}\ \mathtt{e})\ 0 = \mathtt{inl}\ r \\ \mathtt{node}\ (\mathtt{Seq{\to}tree} \circ \mathtt{unshift}\ \mathtt{s}) & o.w. \end{cases} \tag{6.53}$$

Lemma 6.4.14. *If the first element in the sequence is terminated / a leaf, then the rest of the elements will also be terminated.*

$$\left(\prod_{e:\mathbb{N}\to E} \pi_1\ (\mathtt{s}\ \mathtt{e})\ 0 = \mathtt{inl}\ r\right) \Leftrightarrow (\mathtt{s} \equiv \lambda\, \_, (\lambda\, \_, \mathtt{inl}\ r), (\lambda\, \_, \mathtt{inl}\ \mathtt{refl})) \tag{6.54}$$

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 6.4.15. *The types* $\mathtt{tree}$ *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 6.4.14. Otherwise we need to show that

$$\texttt{shift}\ (\texttt{tree}{\to}\texttt{Seq} \circ \texttt{Seq}{\to}\texttt{tree} \circ \texttt{unshift s}) \equiv \texttt{s} \tag{6.55}$$

By induction we get

$$\texttt{tree}{\to}\texttt{Seq} \circ \texttt{Seq}{\to}\texttt{tree} \circ \texttt{unshift s} \equiv \texttt{unshift s} \tag{6.56}$$

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

$$\begin{aligned} \texttt{tree-Seq}\ (\texttt{leaf}\ r) &= \texttt{refl} \\ \texttt{tree-Seq}\ (\texttt{node}\ k) &= \texttt{unshift-shift}\ (\texttt{tree}{\to}\texttt{Seq} \circ \texttt{k}) \cdot \texttt{tree-Seq}\ k \end{aligned} \tag{6.57}$$

□

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

Proof. □

To complete the equality between $\texttt{tree}/{\sim}$ and $\texttt{sTree}$

Definition 6.4.22. The relation $(\cdot \sqsubseteq_{\texttt{tree}} \cdot)$ is defined as

$$\frac{x : tree}{\texttt{leaf} \sqsubseteq_{\texttt{tree}} x} \tag{6.58}$$

$$\frac{}{\texttt{leaf} \sqsubseteq_{\texttt{tree}} x} \tag{6.59}$$

$$x \sqsubseteq_{\texttt{tree}} x \tag{6.60}$$

Definition 6.4.23. There is a function from $\texttt{tree}\ A\ E$ to $\texttt{sTree}\ E$

$$\begin{aligned} \texttt{tree}{\to}\texttt{sTree}\ (\texttt{leaf}\ \_) &= \texttt{leaf}_{\texttt{sTree}} \\ \texttt{tree}{\to}\texttt{sTree}\ (\texttt{node k}) &= \texttt{node}_{\texttt{sTree}}\ (\texttt{tree}{\to}\texttt{sTree} \circ \texttt{k}) \end{aligned} \tag{6.61}$$

We then show this definition is monotone, such that it can be lifted to a function from the quotient

Lemma 6.4.16. *The function* $\texttt{tree}{\to}\texttt{sTree}$ *is monotone, meaning if* $x \sqsubseteq_{\texttt{tree}} y$*, then we have* $\texttt{tree}{\to}\texttt{sTree}\ x \sqsubseteq_{\texttt{sTree}} \texttt{tree}{\to}\texttt{sTree}\ y$.

Proof. By case

$$\texttt{tree}{\to}\texttt{sTree-mono}\ (\texttt{leaf})\ \_ = \tag{6.62}$$

□

We then want to lift this definition to the quotient using the recursor for quotients (6.35),

Definition 6.4.24. There is a function from $\texttt{tree}\ A\ E/{\sim_{\texttt{tree}}}$ to $\texttt{sTree}\ E$

$$\texttt{tree}/{\sim}{\to}\texttt{sTree} = \texttt{rec tree}{\to}\texttt{sTree treesTree} - {\approx}{\to}{\equiv}\ \texttt{sTree-isSet} \tag{6.63}$$

6.4.4 QM-types

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

$$\ldots \tag{6.64}$$

We want to define QM-types as the final coalgebra satisfying a set of equations. The construction takes inspiration from [2]

Cofree Coalgebra

We want to define a cofree coalgebra over a container $(A, \lambda\_, \mathbf{0})$.

This is defined as the left adjoint to the forgetful functor $\mathtt{U} : C\text{-}\gamma \to C$ as $\mathtt{F} : C \to C\text{-}\gamma$.

$$\begin{tikzcd} \mathtt{P}A \arrow[d, "\epsilon"] & \\ A & M \arrow[l, "\alpha"] \arrow[ul, "\alpha^\sharp"'] \end{tikzcd}$$

Figure 6.1: Cofree Coalgebra

A coalgebra PA is cofree on A iff for all coalgebras M and mappings $\alpha : UM \to C$ there is a unique morphism $\alpha^\sharp : M \to TC$ such that the diagram Figure 6.1 commutes

Equation system

We start by defining a equation system called a covariety [3] of a coalgebra (dual of variety of an algebra).

Complete covarities are closed under bisimulation.

6.5 Strongly Extensional (Coalgebra)

Definition 6.5.1. A equation system is given by

$$EqSys : \sum_{(E:\mathcal{U})} \sum_{(V:E\to\mathcal{U})} ((e : E) \to T(Ve)) \times ((e : E) \to T(Ve)) \tag{6.65}$$

where E representing the equations, and variables for the given equations, given by the type `V`, and T is the free coalgebra.

6.5.1 in progress

Let $\texttt{G}$ be functors and $v : \texttt{P} \to \texttt{G}$ a natural transformation. Suppose that for any type V, the functor $(\lambda\_ \to V) \times \texttt{F}$ has a final coalgebra. Then there exists for any G-coalgebra C-γ an P-coalgebra S_C-α and a $\texttt{G}$-homomorphism $\varepsilon : S_C$-$v_{S_C} \circ \alpha \Rightarrow C$-$\gamma$, satisfying the universal property: for any P-coalg U-α_U and any G-homomorphism $f : U$-$v_U \circ \alpha_U \Rightarrow C$-$\gamma$ there exists a unique P-homomorphism $\tilde{f} : U$-$\alpha_u \Rightarrow S_C$-α such that $\varepsilon \circ \tilde{f} = f$. The P-coalg S_c-α (and ε) is called cofree on the G-coalgebra C-γ. [4, theorem 17.1].

The coalgebra generated by the polynomial functor over the container $(A\,, B)$ is a cofree coalgebra. We can now define a quotient, by defining a equation system at the same time, as we define the $\texttt{M}$-type type. The equation systems is defined on a type $E : \mathcal{U}$ with variables of type $V : E \to \mathcal{U}$, each equation is given by functions $l, r : C \to A$ for some type C. A coalgebra satisfies the equation system iff $(t : B(lc) \to MQ) \to (s : B(rc) \to MQ) \to lc \equiv rc$ is inhabited.

6.6 TODO

- Resumption Monad transformer
- coinduction in Coq is broken
- bisim $\Rightarrow$ eq
- copattern matching
- cubical Agda. Relation between $\texttt{M}$-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) $\texttt{M}$-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 $\texttt{QM}$-types, and what are those constructively ??

Chapter 7

Properties of M-types?

7.1 Closure properties of $\mathtt{M}$-types

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

7.1.1 Product of $\mathtt{M}$-types

We start with containers and work up to $\mathtt{M}$-types.

Definition 7.1.1. The product of two containers is defined as [1]

$$(A, \mathtt{B}) \times (C, \mathtt{D}) \equiv (A \times C, \lambda (a, c), \mathtt{B}\ a \times \mathtt{D}\ c). \tag{7.1}$$

We can lift this rule, through the diagram in Figure 7.1, used to define $\mathtt{M}$-types.

Theorem 7.1.1. *For any $n : \mathbb{N}$ the following is true*

$$\mathtt{P}_{(A, \mathtt{B})}{}^{n}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{n}\ \mathbf{1} \equiv \mathtt{P}_{(A, \mathtt{B}) \times (C, \mathtt{D})}{}^{n}\ \mathbf{1}. \tag{7.2}$$

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

$$\mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1} \equiv \mathtt{P}_{(A, \mathtt{B}) \times (C, \mathtt{D})}{}^{m}\ \mathbf{1}, \tag{7.3}$$

in the following

$$\begin{aligned}
& \mathtt{P}_{(A, \mathtt{B})}{}^{m+1}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{m+1}\ \mathbf{1} && (7.4)\\
\equiv\ & \mathtt{P}_{(A, \mathtt{B})}(\mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1}) \times \mathtt{P}_{(C, \mathtt{D})}(\mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1}) && (7.5)\\
\equiv\ & \sum_{a : A} \mathtt{B}\ a \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1} \times \sum_{c : C} \mathtt{D}\ c \to \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1} && (7.6)\\
\equiv\ & \sum_{a, c : A \times C} (\mathtt{B}\ a \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1}) \times (\mathtt{D}\ c \to \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1}) && (7.7)\\
\equiv\ & \sum_{a, c : A \times C} \mathtt{B}\ a \times \mathtt{D}\ c \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1} && (7.8)
\end{aligned}$$

$$
\begin{tikzcd}
A & A \times C \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & C \\
\mathtt{P}(\mathtt{M}_{(A,\mathtt{B})}) \arrow[u, "\pi_1"'] & \mathtt{P}(\mathtt{M}_{(A,\mathtt{B})\times(C,\mathtt{D})}) \arrow[l] \arrow[r] \arrow[u, "\pi_1"'] & \mathtt{P}(\mathtt{M}_{(C,\mathtt{D})}) \arrow[u, "\pi_1"'] \\
\mathtt{M}_{(A,\mathtt{B})} \arrow[u, "\mathtt{out}"'] \arrow[d, "\pi_{(!)}"] & \mathtt{M}_{(A,\mathtt{B})\times(C,\mathtt{D})} \arrow[l] \arrow[r] \arrow[u, "\mathtt{out}"'] \arrow[d, "\pi_{(!)}"] & \mathtt{M}_{(C,\mathtt{D})} \arrow[u, "\mathtt{out}"'] \arrow[d, "\pi_{(!)}"] \\
\mathtt{P}_{(A,\mathtt{B})}{}^{n}\ \mathbf{1} \arrow[d, "\pi_{(n)}"] & \mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}{}^{n}\ \mathbf{1} \arrow[l] \arrow[r] \arrow[d, "\pi_{(n)}"] & \mathtt{P}_{(C,\mathtt{D})}{}^{n}\ \mathbf{1} \arrow[d, "\pi_{(n)}"] \\
\vdots \arrow[d, "\pi_{(1)}"] & \vdots \arrow[d, "\pi_{(1)}"] & \vdots \arrow[d, "\pi_{(1)}"] \\
\mathtt{P}_{(A,\mathtt{B})}\ \mathbf{1} \arrow[d, "\pi_{(0)}"] & \mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}\ \mathbf{1} \arrow[l] \arrow[r] \arrow[d, "\pi_{(0)}"] & \mathtt{P}_{(C,\mathtt{D})}\ \mathbf{1} \arrow[d, "\pi_{(0)}"] \\
\mathbf{1} & \mathbf{1} \arrow[l, "\mathtt{refl}"] \arrow[r, "\mathtt{refl}"'] & \mathbf{1}
\end{tikzcd}
$$

Figure 7.1: Diagram for products of chains

$$\equiv \sum_{a,c:A\times C} \mathtt{B}\ a \times \mathtt{D}\ c \to \mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}{}^{m}\ \mathbf{1} \tag{7.9}$$

$$\equiv \mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}(\mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}{}^{m}\ \mathbf{1}) \tag{7.10}$$

$$\equiv \mathtt{P}_{(A,\mathtt{B})\times(C,\mathtt{D})}{}^{m+1}\ \mathbf{1} \tag{7.11}$$

taking the limit of (7.2) we get

$$\mathtt{M}_{(A,\mathtt{B})} \times \mathtt{M}_{(C,\mathtt{D})} \equiv \mathtt{M}_{(A,\mathtt{B})\times(C,\mathtt{D})}. \tag{7.12}$$

$\square$

Example 7.1.1. For streams we get

$$\mathtt{stream}\ A \times \mathtt{stream}\ B \equiv \mathtt{M}_{(A,\lambda\ \_,\mathbf{1})} \times \mathtt{M}_{(B,\lambda\ \_,\mathbf{1})} \equiv \mathtt{M}_{(A,\lambda\ \_,\mathbf{1})\times(B,\lambda\ \_,\mathbf{1})} \equiv \mathtt{stream}\ (A \times B) \tag{7.13}$$

as expected. Transporting along (7.13) gives us a definition for `zip`.

7.1.2 Co-product

Coproducts?

7.1.3 ...

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

Chapter 8

Examples of M-types

8.1 The Partiality monad

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

$$\forall x, \bot \sqsubseteq x \tag{8.1}$$

$$\forall x, x \sqsubseteq x \tag{8.2}$$

$$\forall x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{8.3}$$

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

$$\forall x, \bot \sqsubseteq x \tag{8.4}$$

$$\forall x, x \sqsubseteq x \tag{8.5}$$

$$\forall x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{8.6}$$

$$\forall x\, y, x \sqsubseteq y \to y \sqsubseteq x \to x \equiv y \tag{8.7}$$

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.

$$\texttt{now}\ x = x + 1 \tag{8.8}$$

$$\texttt{later}\ y = y \tag{8.9}$$

8.2 TODO: Place these subsections

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

8.2.1 Identity Bisimulation

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

Lemma 8.2.1. *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

$$\begin{aligned} \alpha_{\equiv} &: \overline{\equiv} \to \mathtt{P}(\overline{\equiv}) \\ \alpha_{\equiv}(x, y) &:= \pi_1\ (\mathtt{out}\ x)\ ,\ (\lambda\, b, (\pi_2\ (\mathtt{out}\ x)\ b\, , \mathtt{refl}_{(\pi_2\,(\mathtt{out}\,x)\,b)})) \end{aligned} \tag{8.10}$$

and the two projections

$$\pi_1^{\overline{\equiv}} = (\pi_1, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{refl}_{\mathtt{out}\,a}) \tag{8.11}$$

$$\pi_2^{\overline{\equiv}} = (\pi_2, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{cong}_{\mathtt{out}}\ (r^{-1})). \tag{8.12}$$

This defines the bisimulation, given by the diagram in Figure 8.1. ☐

$$\mathtt{M}\text{-}\mathtt{out} \xLeftarrow{\pi_1^{\overline{\equiv}}} \overline{\equiv}\text{-}\alpha_{\equiv} \xRightarrow{\pi_2^{\overline{\equiv}}} \mathtt{M}\text{-}\mathtt{out}$$

Figure 8.1: Identity bisimulation

8.2.2 Bisimulation of Streams

TODO

8.2.3 Bisimulation of Delay Monad

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

Definition 8.2.1. The relation $\sim_{\mathtt{delay}}$ is defined by the following rules

$$\frac{R : U \quad r : R}{\mathtt{now}\ r \sim_{\mathtt{delay}} \mathtt{now}\ r : \mathcal{U}}\ \mathtt{now}\sim \tag{8.13}$$

$$\frac{R : \mathcal{U} \quad t : \mathtt{delay}\ R \quad u : \mathtt{delay}\ R \quad t \sim_{\mathtt{delay}} u : \mathcal{U}}{\mathtt{later}\ t \sim_{\mathtt{delay}} \mathtt{later}\ u : \mathcal{U}}\ \mathtt{later}\sim \tag{8.14}$$

Theorem 8.2.2. *The relation* $\sim_{\mathtt{delay}}$ *is a bisimulation for* $\mathtt{delay}\ R$*.*

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\mathtt{delay}}} &: \overline{\sim_{delay}} \to \mathtt{P}(\overline{\sim_{delay}}) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{now}\sim\ r) &:= (\mathtt{inr}\ r, \lambda\,(\,)) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{later}\sim\ x\ y\ q) &:= (\mathtt{inl}\ \mathtt{tt}, \lambda\,\_, (x, y, q)) \end{aligned} \tag{8.15}$$

then we define the projections

$$\pi_1^{\overline{\sim delay}} = \left(\pi_1 \quad , \quad \texttt{funExt}\ \lambda\,(a,b,p), \begin{cases} (\texttt{inr}\ r, \lambda\,(\,)) & p = \texttt{now}\!\sim\ r \\ (\texttt{inl}\ \texttt{tt}, \lambda\,\_, x) & p = \texttt{later}\!\sim\ x\ y\ q \end{cases}\right) \tag{8.16}$$

$$\pi_2^{\overline{\sim delay}} = \left(\pi_2 \quad , \quad \texttt{funExt}\ \lambda\,(a,b,p), \begin{cases} (\texttt{inr}\ r, \lambda\,(\,)) & p = \texttt{now}\!\sim\ r \\ (\texttt{inl}\ \texttt{tt}, \lambda\,\_, y) & p = \texttt{later}\!\sim\ x\ y\ q \end{cases}\right) \tag{8.17}$$

$$\tag{8.18}$$

This defines the bisimulation, given by the diagram in Figure 8.2. □

$$\texttt{delay}\ R\texttt{-out} \xLeftarrow{\pi_1^{\overline{\sim delay}}} \overline{\sim\texttt{delay}}\text{-}\alpha_{\sim\texttt{delay}} \xRightarrow{\pi_2^{\overline{\sim delay}}} \texttt{delay}\ R\texttt{-out}$$

Figure 8.2: Strong bisimulation for delay monad

8.2.4 Bisimulation of ITrees

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

$$\frac{a, b : R \quad a \equiv_R b}{\texttt{Ret}\ a \cong \texttt{Ret}\ b}\ \texttt{EqRet} \tag{8.19}$$

$$\frac{t, u : \texttt{itree E}\ R \quad t \cong u}{\texttt{Tau}\ t \cong \texttt{Tau}\ u}\ \texttt{EqTau} \tag{8.20}$$

$$\frac{A : \mathcal{U} \quad e : \texttt{E}\ A \quad k_1, k_2 : A \to \texttt{itree E}\ R \quad t \cong u}{\texttt{Vis}\ e\ k_1 \cong \texttt{Tau}\ e\ k_2}\ \texttt{EqVis} \tag{8.21}$$

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.

8.2.5 Zip Function

We want the diagram in Figure 8.3 to commute, meaning we get the computation rules

$$(\texttt{hd} \times \texttt{hd}) \equiv \texttt{hd} \circ \texttt{zip} \tag{8.22}$$

$$\texttt{zip} \circ (\texttt{tl} \times \texttt{tl}) \equiv \texttt{tl} \circ \texttt{zip} \tag{8.23}$$

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

$$\begin{aligned} &\texttt{lift}_{\texttt{M}} \left(x : \prod_{n:\mathbb{N}} (A \to \texttt{P}_S{}^n\ \mathbf{1})\right) \left(u : \prod_{n:\mathbb{N}} (A \to \pi_n(x_{n+1} a) \equiv x_n a)\right) (a : A) : \texttt{M}\ S := \\ &\quad (\lambda\, n, x\ n\ a), (\lambda\, n\ i, p\ n\ a\ i). \end{aligned} \tag{8.24}$$

$$
\begin{tikzcd}
 & A \times B & \\
\mathtt{stream}\ A \times \mathtt{stream}\ B \arrow[ur, "\mathtt{hd}\times\mathtt{hd}"'] \arrow[rr, "\mathtt{zip}"] \arrow[d, "\mathtt{tl}\times\mathtt{tl}"] & & \mathtt{stream}\ (A \times B) \arrow[ul, "\mathtt{hd}"] \arrow[d, "\mathtt{tl}"] \\
\mathtt{stream}\ A \times \mathtt{stream}\ B \arrow[rr, "\mathtt{zip}"] & & \mathtt{stream}\ (A \times B)
\end{tikzcd}
$$

Figure 8.3: TODO

To use this definition, we first define some helper functions

$$\mathtt{zip}_X\ n\ (x, y) = \begin{cases} \mathbf{1} & \mathtt{if}\ n = 0 \\ (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)), & \mathtt{if}\ n = m + 1 \end{cases} \tag{8.25}$$

$$\mathtt{zip}_\pi\ n\ (x, y) = \begin{cases} \mathtt{refl} & \mathtt{if}\ n = 0 \\ \lambda\, i, (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_\pi\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)\ i), & \mathtt{if}\ n = m + 1 \end{cases}, \tag{8.26}$$

we can then define

$$\mathtt{zip}_{lift}\ (x, y) := \mathtt{lift}_{\mathtt{M}}\ \mathtt{zip}_X\ \mathtt{zip}\ (x, y). \tag{8.27}$$

Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\mathtt{transport}_?\ a \equiv \mathtt{zip}_{lift}\ a \tag{8.28}$$

$$\equiv \mathtt{lift}_{\mathtt{M}}\ \mathtt{zip}_X\ \mathtt{zip}_\pi\ (x, y) \tag{8.29}$$

$$\equiv (\lambda\, n, \mathtt{zip}_X\ n\ (x, y)), (\lambda\, n\ i, \mathtt{zip}_\pi\ n\ (x, y)\ i) \tag{8.30}$$

zero case X

$$\mathtt{zip}_X\ 0\ (x, y) \equiv \mathbf{1} \tag{8.31}$$

Successor case X

$$\mathtt{zip}_X\ (m + 1)\ (x, y) \equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)) \tag{8.32}$$

$$\equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, ?\ (\mathtt{tl}\ a)) \tag{8.33}$$

$$\equiv (\mathtt{hd}\ (\mathtt{transport}_? a)), (\lambda\, \_, \mathtt{transport}_?(\mathtt{tl}\ a)) \tag{8.34}$$

$$\equiv \mathtt{transport}_?\ a \tag{8.35}$$

$$\tag{8.36}$$

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

$$\equiv (), (\lambda\, i, \mathtt{zip}_\pi\ 0\ (x, y)\ i) \tag{8.37}$$

$$\equiv \mathbf{1}, \mathtt{refl} \tag{8.38}$$

$$\tag{8.39}$$

successor case

$$\equiv (\texttt{zip}_X\ (m+1)\ (x,y)), (\lambda\, i, \texttt{zip}_\pi\ (m+1)\ (x,y)\ i) \tag{8.40}$$

$$\equiv ((\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\_, \texttt{zip}_X\ m\ (\texttt{tl}\ x, \texttt{tl}\ y))), (\lambda\, i, (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\_, \texttt{zip}_\pi\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)\ i)) \tag{8.41}$$

Complete this proof

8.2.6 Examples of Fixed Points

Zeros

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

$$\texttt{const}_\texttt{X}\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \mathbf{1} & n = 0 \\ (c, \lambda\_, \texttt{const}_\texttt{X}\ m\ c) & n = m + 1 \end{cases} \tag{8.42}$$

$$\texttt{const}_\pi\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \texttt{refl} & n = 0 \\ \lambda\, i, (c, \lambda\_, \texttt{const}_\pi\ m\ c\ i) & n = m + 1 \end{cases} \tag{8.43}$$

to get the definition of zero stream

$$\texttt{zeros} := \texttt{lift}_\texttt{M}\ \texttt{const}_\texttt{X}\ \texttt{const}_\pi\ 0. \tag{8.44}$$

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

$$\texttt{hd}\ \texttt{zeros} \equiv 0 \tag{8.45}$$

$$\texttt{tl}\ \texttt{zeros} \equiv \texttt{zeros} \tag{8.46}$$

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 $\texttt{M}$-type (which is a final coalgebra)

$$\begin{tikzcd} \texttt{spin} \arrow[d, "\texttt{later}"] \\ \texttt{later spin} \end{tikzcd}$$

Figure 8.4: 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 9

Additions to the Cubical Agda Library

Chapter 10

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

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

Appendix A

The Technical Details

. . .