M-types and Co-induction in HoTT and Cubical Type Theory

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science
April 27, 2020
Advisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

in English. . .

Resumé

in Danish...

Acknowledgments

. . .

Lasse Letager Hansen,
Aarhus, April 27, 2020.

Contents

Chapter 1

Introduction

motivate and explain the problem to be addressed

example of a citation: [4]

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

Chapter 2

Background Theory

Chapter 3

M-types

3.1 Containers / Signatures

Definition 3.1.1. A Container (or Signature) is a pair $S = (A, B)$ of types $\vdash A : \mathcal{U}$ and $a : A \vdash B(a) : \mathcal{U}$. From a container we can define a polynomial functor, defined for objects (types) as

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

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

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

Example 3.1.1. The type for streams over the type A is defined by the container $S = (A, \lambda \_, \mathbf{1})$, applying the polynomial functor we get

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

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

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

Definition 3.1.2. The coalgebra for the functor $\mathtt{P}$ is defined as

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

We denote a coalgebra of C and γ as C-γ. The coalgebra morphisms are defined as

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

Definition 3.1.3. M-types can now be defined from a container S as the type $\mathtt{M}_S$ such that the coalgebra for $\mathtt{M}_S$ and $\mathtt{out} : \mathtt{M}_S \to \mathtt{P}_S(\mathtt{M}_S)$ fulfills 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 this construction of the M-type as $\mathtt{M}_{(A,B)}$ or $\mathtt{M}_S$ or just M when the Container is clear from the context.

Example 3.1.2. If we continue our example for streams this will give us the M-type, we can see that $\mathtt{P}_S(\mathtt{M}) = A \times \mathtt{M}$, meaning we have the following diagram, where out is an isomorphism (because

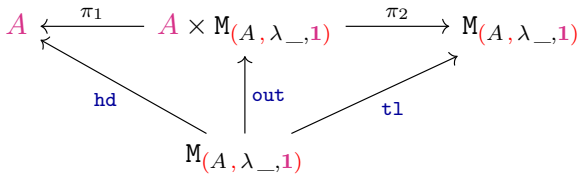

Figure 3.1: M-types of streams

of the finality of the coalgebra), with 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 $\mathtt{stream}\ A = \mathtt{M}_{(A,\lambda\_,1)}$,

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

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

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

Lemma 3.1.1. *Given* $\ell : \prod_{(n:\mathbb{N})}(X_n \to X_{n+1})$ *and* $y : \sum_{x:\prod_{(n:\mathbb{N})} X_n} x_{n+1} \equiv l_n(x_n)$ *we can collapse the chain* $\mathcal{L} \equiv X_0$.

Proof. We define this collapse by the equivalence

$$\mathtt{fun}_{\mathcal{L}collapse}\ (x, r) = x_0 \tag{3.11}$$

$$\mathtt{inv}_{\mathcal{L}collapse}\ x_0 = (\lambda n, \mathtt{transport}\ \ell^n\ x_0), (\lambda\, n, \mathtt{refl}_{\mathtt{transport}\ \ell^{(n+1)}\ x_0}) \tag{3.12}$$

$$\mathtt{rinv}\ x_0 = \mathtt{refl}_{x_0} \tag{3.13}$$

For $\mathtt{linv}\ (x, r)$ we define a fiber (X, z, l) over $\mathbb{N}$ given some $z : X_0$. Then any element of the type $\sum_{x:\prod_{(n:\mathbb{N})} X_n} x_{n+1} \equiv l_n(x_n)$ is equal to a section over this fiber, specifically y is equal to a fiber. Since this section is defined over $\mathbb{N}$ which is an initial algebra for the functor $GY = \mathbf{1} + Y$, we get that the section are contractible, meaning $y \equiv \mathtt{inv}_{\mathcal{L}collapse}(\mathtt{fun}_{\mathcal{L}collapse}\ y)$, since both are equal to such sections. □

Theorem 3.1.2. *We want to proof the construction of* in *and* out *for the container* (A, B) *more precisely we want to define the equality*

$$shift : \mathcal{L} \equiv P\mathcal{L} \tag{3.14}$$

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

Proof. We do this we the two helper lemmas

$$\alpha : \mathcal{L}^P \equiv P\mathcal{L} \tag{3.15}$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^P \tag{3.16}$$

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

$$\mathtt{fun}_{\mathcal{L}unique}\ (a,b) = \left(\lambda\, n, \begin{cases} \mathtt{tt} & n = 0 \\ a\ m & n = m+1 \end{cases}\right), \left(\lambda\, n \begin{cases} \mathtt{refl}_{\mathtt{tt}} & n = 0 \\ b\ m & n = m+1 \end{cases}\right) \tag{3.17}$$

$$\mathtt{inv}_{\mathcal{L}unique}\ (a,b) = a \circ \mathtt{incr}, b \circ \mathtt{incr} \tag{3.18}$$

$$\mathtt{rinv}_{\mathcal{L}unique}\ (a,b) = \mathtt{refl} \tag{3.19}$$

$$\mathtt{linv}_{Lunique}\ (a,b) = \mathtt{refl} \tag{3.20}$$

The definition of α follows from Lemma 3.1.1,

$$\mathcal{L}^P \equiv \sum_{(u:\mathcal{L}_{(\lambda\_,A,\mathtt{id})})} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv_{\lambda\, x, B(\pi_2\, a\, n\, x) \to X_n} u_n \tag{3.21}$$

$$\equiv \sum_{(a:A)} \sum_{(u:\prod_{(n:\mathbb{N})} Ba \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv u_n \tag{3.22}$$

$$\equiv P\mathcal{L} \tag{3.23}$$

We can then define $shift$ as

$$shift = \mathtt{sym}\ \alpha \cdot \mathcal{L}unique \tag{3.24}$$

Then we define $\mathtt{in} = \mathtt{transport}\ shift$ and $\mathtt{out} = \mathtt{transport}\ (\mathtt{sym}\ shift)$, since $\mathtt{in}$ and $\mathtt{out}$ are part of an equality relation ($shift$), they become embeddings. □

3.2 Closure properties of M-types

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

3.2.1 Product of M-types

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

Definition 3.2.1. The product of two containers is defined as

$$(A\,, B) \times (C\,, D) \equiv (A \times C\,, \lambda\,(a, c), \mathtt{B}\ a \times \mathtt{D}\ c). \tag{3.25}$$

We can lift this rule, through the diagram in Figure 3.2, used to define M-types.

Theorem 3.2.1. *For any $n : \mathbb{N}$ the following is true*

$$\mathtt{P}_{(A\,,B)}{}^n\ \mathbf{1} \times \mathtt{P}_{(C\,,D)}{}^n\ \mathbf{1} \equiv \mathtt{P}_{(A\,,B)\times(C\,,D)}{}^n\ \mathbf{1}. \tag{3.26}$$

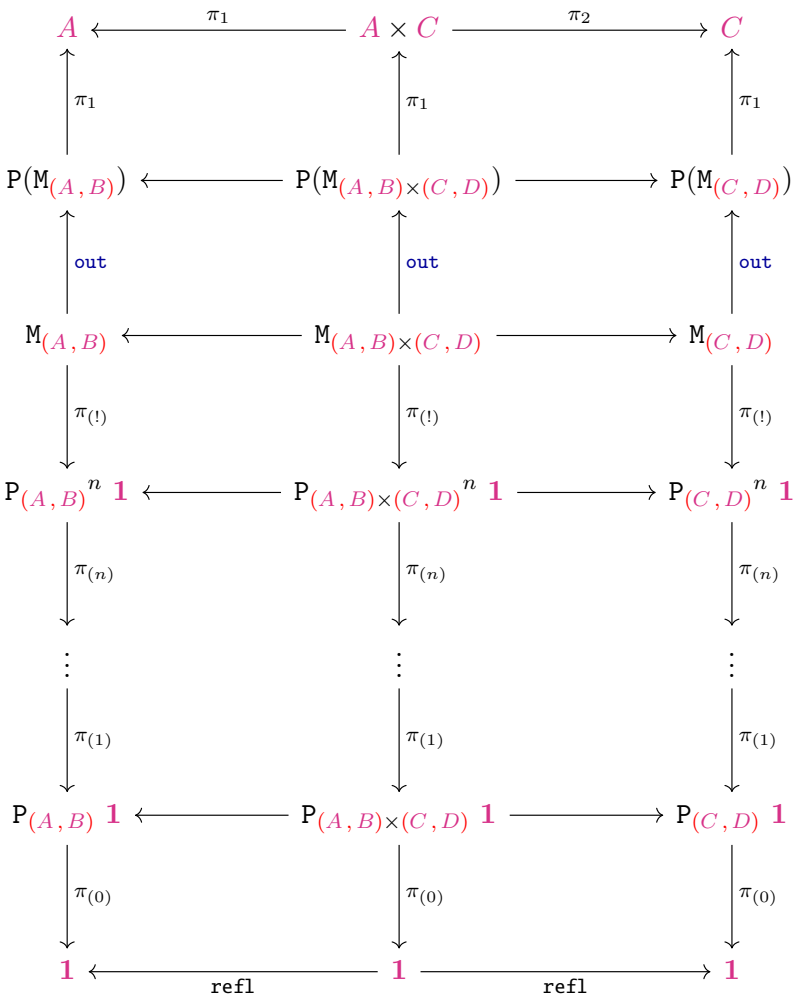

Figure 3.2: Diagram for products of chains

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

$$\mathtt{P}_{(A\,,\,B)}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C\,,\,D)}{}^{m}\ \mathbf{1} \equiv \mathtt{P}_{(A\,,\,B)\times(C\,,\,D)}{}^{m}\ \mathbf{1}, \tag{3.27}$$

in the following

$$\begin{aligned}
&\mathtt{P}_{(A\,,\,B)}{}^{m+1}\ \mathbf{1} \times \mathtt{P}_{(C\,,\,D)}{}^{m+1}\ \mathbf{1} && (3.28)\\
\equiv\ &\mathtt{P}_{(A\,,\,B)}(\mathtt{P}_{(A\,,\,B)}{}^{m}\ \mathbf{1}) \times \mathtt{P}_{(C\,,\,D)}(\mathtt{P}_{(C\,,\,D)}{}^{m}\ \mathbf{1}) && (3.29)\\
\equiv\ &\sum_{a:A} B\ a \to \mathtt{P}_{(A\,,\,B)}{}^{m}\ \mathbf{1} \times \sum_{c:C} D\ c \to \mathtt{P}_{(C\,,\,D)}{}^{m}\ \mathbf{1} && (3.30)\\
\equiv\ &\sum_{a,c:A\times C} (B\ a \to \mathtt{P}_{(A\,,\,B)}{}^{m}\ \mathbf{1}) \times (D\ c \to \mathtt{P}_{(C\,,\,D)}{}^{m}\ \mathbf{1}) && (3.31)\\
\equiv\ &\sum_{a,c:A\times C} B\ a \times D\ c \to \mathtt{P}_{(A\,,\,B)}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C\,,\,D)}{}^{m}\ \mathbf{1} && (3.32)
\end{aligned}$$

$$\equiv \sum_{a,c:A\times C} B\ a \times D\ c \to \mathtt{P}_{(A\,,B)\times(C\,,D)}{}^{m}\ \mathbf{1} \tag{3.33}$$

$$\equiv \mathtt{P}_{(A\,,B)\times(C\,,D)}(\mathtt{P}_{(A\,,B)\times(C\,,D)}{}^{m}\ \mathbf{1}) \tag{3.34}$$

$$\equiv \mathtt{P}_{(A\,,B)\times(C\,,D)}{}^{m+1}\ \mathbf{1} \tag{3.35}$$

taking the limit of (3.26) we get

$$\mathtt{M}_{(A\,,B)} \times \mathtt{M}_{(C\,,D)} \equiv \mathtt{M}_{(A\,,B)\times(C\,,D)}. \tag{3.36}$$

□

Example 3.2.1. For streams we get

$$\mathtt{stream}\ A \times \mathtt{stream}\ B \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})} \times \mathtt{M}_{(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})\times(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{stream}\ (A \times B) \tag{3.37}$$

as expected. Transporting along (3.37) gives us a definition for `zip`.

3.3 Co-induction Principle for M-types

We can now construct a co-induction principle given a bisimulation relation

Definition 3.3.1. For all coalgebras $C\text{-}\gamma : \mathtt{Coalg}_S$, 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 $\overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, making the diagram in Figure **??** 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 3.3: Bisimulation for a coalgebra

Definition 3.3.2 (Co-induction principle)**.** Given a relation $\mathcal{R}$, that is part of a bisimulation over a final P-coalgebra $\mathtt{M\text{-}out} : \mathtt{Coalg}_S$ we get the diagram in Figure **??**,

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

Figure 3.4: Bisimulation principle for final coalgebra

where $\pi_1^{\overline{\mathcal{R}}} = ! = \pi_2^{\overline{\mathcal{R}}}$, which means given $r : m\ \mathcal{R}\ m'$ we get the equation

$$m = \pi_1^{\overline{\mathcal{R}}}(m, m', r) = \pi_2^{\overline{\mathcal{R}}}(m, m', r) = m'. \tag{3.38}$$

3.4 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 asq

Quotients of the delay monad

3.4.1 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 signatureswith operationsof 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?

Partiality monad

A simple example of a quotient inductive-inductive type is the partiality monad over R, defined as a type

$$D : \texttt{delay}\ R \tag{3.39}$$
$$now : R \to D \tag{3.40}$$
$$later : D \to D \tag{3.41}$$

together with a relation (indexed over the type) and properties

$$(\cdot \approx \cdot) : D \to D \to \mathcal{U} \tag{3.42}$$
$$\approx_{now} : (r : R) \to now\ r \approx now\ r \tag{3.43}$$
$$\approx_{laterL} : (x\ y : D) \to later\ x \approx y \tag{3.44}$$
$$\approx_{laterR} : (x\ y : D) \to x \approx later\ y \tag{3.45}$$

and then the quotiented equality

$$later\ x \equiv x \tag{3.46}$$

Partiality from Delay monad

We want to define an ordering on the delay monad, we do this by counting the number of `later`s, letting the infinitely delayed element `never`, be the bottom element of the ordering. So we get

$$\frac{x : \texttt{Delay}\ R}{\texttt{never} \sqsubseteq x} \tag{3.47}$$

$$\frac{x : \texttt{Delay}\ R \quad a : R \quad x \downarrow a}{x \sqsubseteq \texttt{now}\ a}\ \sqsubseteq_{\texttt{now}} \tag{3.48}$$

$$\frac{x \sqsubseteq y}{\texttt{later}\ x \sqsubseteq \texttt{later}\ y}\ \sqsubseteq_{\texttt{later}} \tag{3.49}$$

given these two rules, we want to show that we actually have an ordering. So first we want to show that we get reflexivity, we do the proof by structural induction.

Lemma 3.4.1. *The bottom element of the relation $\sqsubseteq$ is* `never`*, meaning for all x we get* `never` $\sqsubseteq x$*.*

Proof. If $x =$ □

Lemma 3.4.2. *The relation $\sqsubseteq$ is reflexive.*

Proof. If $x =$ `now` a then we know that $x \downarrow a$ meaning by the rule $\sqsubseteq_{\texttt{now}}$, we have `now` $a \sqsubseteq$ `now` a. If we have $x =$ `later` y then by the rule $\sqsubseteq_{\texttt{later}}$ we just need to show $y \sqsubseteq y$, which is true by the induction hypothesis. □

Lemma 3.4.3. *For all x :* `Delay` R *we have* `now` $a \sqsubseteq x$ *implies that $x =$* `now` a*.*

Proof. If $x =$ `now` b, then `now` $a \downarrow b$, but this means that $a \equiv b$. If □

Lemma 3.4.4. *The relation $\sqsubseteq$ is transitive.*

Proof. Again we do case analysis $x \sqsubseteq y \sqsubseteq z$, if $x \sqsubseteq y$ is $\sqsubseteq_{\texttt{now}}$ then we know $x \downarrow a$ and $y =$ `now` a, then it must be the case that $z =$ `now` a as well, since the only element `now` a is "less defined" than is itself. □

3.4.2 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

$$asfd \tag{3.50}$$

We want to define QM-types as the final coalgebra satisfying a set of equations. The construction takes inspiration from [1]

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 `U` : C-$\gamma \to C$ as `F` : $C \to C$-γ.

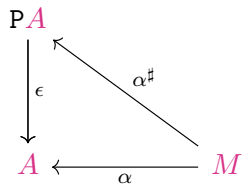

Figure 3.5: Cofree Coalgebra

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 3.5 commutes

Equation system

We start by defining a equation system called a covariety [2] of a coalgebra (dual of variety of an algebra).

Complete covarities are closed under bisimulation.

3.5 Strongly Extensional (Coalgebra)

Definition 3.5.1. A equation system is given by

$$EqSys : \sum_{(E:\mathcal{U})} \sum_{(V:E\to\mathcal{U})} ((e : E) \to T(Ve)) \times ((e : E) \to T(Ve)) \tag{3.51}$$

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

3.5.1 in progress

Let `G` be functors and v : `P` $\to$ `G` a natural transformation. Suppose that for any type V, the functor $(\lambda \_ \to V) \times$ `F` has a final coalgebra. Then there exists for any G-coalgebra C-γ an P-coalgebra S_C-α and a `G`-homomorphism $\varepsilon : S_C\text{-}v_{S_C} \circ \alpha \Rightarrow C\text{-}\gamma$, satisfying the universal property: for any P-coalg U-α_U and any G-homomorphism $f : U\text{-}v_U \circ \alpha_U \Rightarrow C\text{-}\gamma$ there exists a unique P-homomorphism $\tilde{f} : U\text{-}\alpha_u \Rightarrow S_C\text{-}\alpha$ such that $\varepsilon \circ \tilde{f} = f$. The P-coalg S_c-α (and ε) is called cofree on the G-coalgebra C-γ. [3, 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 `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.

3.6 TODO

- Resumption Monad transformer
- co-induction in Coq is broken
- bisim $\Rightarrow$ eq
- copattern matching
- cubical Agda. Relation between `M`-types defined by co-induction/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 co-induction 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 4

Contractive

We want to show that

$$\sum_{(a:A)} \sum_{(u:\prod_{(n:\mathbb{N})} B\ a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ (u_{n+1}) \equiv u_n \tag{4.1}$$

$$\equiv \sum_{(a:\sum_{(a:(n:\mathbb{N})\to A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(u:(n:\mathbb{N})\to B\ (\pi_1\ a)_n \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv_{(\lambda x, B\ ((\pi_2\ a)_n\ x)) \to X_n)} u_n \tag{4.2}$$

we have the two functions

$$\texttt{fun}\ (x, y) := ((\lambda\, n, x), (\lambda\, n\ i, x)), y \tag{4.3}$$

and its inverse (we use the notation $y_{(!n)} = y_n \cdot y_{n-1} \cdot \ \ldots\ \cdot y_1 \cdot y_0$)

$$\texttt{inv}\ ((x, y), (z, w)) := x_0, ((\lambda\, n\ a, z\ n\ (\texttt{subst}\ B\ y_{(!n)}\ a)), (\lambda\, n\ i\ a, w\ n\ i\ (\texttt{subst}\ B\ ?\ a))) \tag{4.4}$$

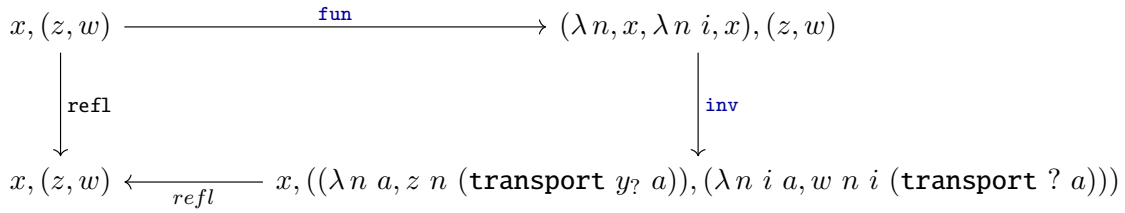

Figure 4.1: Left inverse

so we want

$$\texttt{invhelper0}\ ((\lambda\, n, (x, y), \lambda\, n\ i, (x, y)), (z, w)) \tag{4.5}$$

$$\equiv \lambda n, \begin{cases} \texttt{tt} & n = 0 \end{cases} \tag{4.6}$$

$$\equiv z \tag{4.7}$$

$$\texttt{invhelper1}\ ((\lambda\, n, (x, y), \lambda\, n\ i, (x, y)), (z, w)) \equiv w \tag{4.8}$$

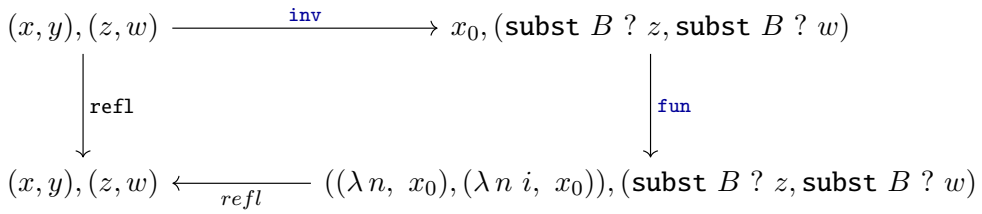

Figure 4.2: Right inverse

Chapter 5

Examples of `M`-types

5.1 The Partiality monad

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

$$\forall x, \bot \sqsubseteq x \tag{5.1}$$

$$\forall\, x, x \sqsubseteq x \tag{5.2}$$

$$\forall\, x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{5.3}$$

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

$$\forall x, \bot \sqsubseteq x \tag{5.4}$$

$$\forall\, x, x \sqsubseteq x \tag{5.5}$$

$$\forall\, x\, y\, z, x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z \tag{5.6}$$

$$\forall\, x\, y, x \sqsubseteq y \to y \sqsubseteq x \to x \equiv y \tag{5.7}$$

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.

$$\texttt{now}\ x = x + 1 \tag{5.8}$$

$$\texttt{later}\ y = y \tag{5.9}$$

5.2 TODO: Place these subsections

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

5.2.1 Identity Bisimulation

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

Lemma 5.2.1. *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

$$\begin{aligned} \alpha_{\equiv} &: \equiv \to \mathtt{P}(\equiv) \\ \alpha_{\equiv}(x, y) &:= \pi_1\ (\mathtt{out}\ x)\ ,\ (\lambda\, b, (\pi_2\ (\mathtt{out}\ x)\ b\ ,\ \mathtt{refl}_{(\pi_2\ (\mathtt{out}\ x)\ b)})) \end{aligned} \tag{5.10}$$

and the two projections

$$\pi_1^{\equiv} = (\pi_1, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{refl}_{\mathtt{out}\,a}) \tag{5.11}$$

$$\pi_2^{\equiv} = (\pi_2, \mathtt{funExt}\ \lambda\,(a, b, r), \mathtt{cong}_{\mathtt{out}}\ (\mathtt{sym}\ r)). \tag{5.12}$$

This defines the bisimulation, given by the diagram in Figure 5.1. □

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

Figure 5.1: Identity bisimulation

5.2.2 Bisimulation of Streams

TODO

5.2.3 Bisimulation of Delay Monad

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

Definition 5.2.1. The relation $\sim_{\mathtt{delay}}$ is defined by the following rules

$$\frac{R : U \quad r : R}{\mathtt{now}\ r \sim_{\mathtt{delay}} \mathtt{now}\ r : \mathcal{U}}\ \mathtt{now}\sim \tag{5.13}$$

$$\frac{R : \mathcal{U} \quad t : \mathtt{delay}\ R \quad u : \mathtt{delay}\ R \quad t \sim_{\mathtt{delay}} u : \mathcal{U}}{\mathtt{later}\ t \sim_{\mathtt{delay}} \mathtt{later}\ u : \mathcal{U}}\ \mathtt{later}\sim \tag{5.14}$$

Theorem 5.2.2. *The relation* $\sim_{\mathtt{delay}}$ *is a bisimulation for* $\mathtt{delay}\ R$.

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\mathtt{delay}}} &: \overline{\sim_{delay}} \to \mathtt{P}(\overline{\sim_{delay}}) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{now}\sim\ r) &:= (\mathtt{inr}\ r, \lambda\,(\,)) \\ \alpha_{\sim_{\mathtt{delay}}}\ (a, b, \mathtt{later}\sim\ x\ y\ q) &:= (\mathtt{inl}\ \mathtt{tt}, \lambda\,\_, (x, y, q)) \end{aligned} \tag{5.15}$$

then we define the projections

$$\pi_1^{\overline{\sim_{delay}}} = \left(\pi_1 \quad , \quad \texttt{funExt}\ \lambda\,(a, b, p), \begin{cases} (\texttt{inr}\ r, \lambda\,()) & p = \texttt{now}{\sim}\ r \\ (\texttt{inl}\ \texttt{tt}, \lambda\,\_, x) & p = \texttt{later}{\sim}\ x\ y\ q \end{cases}\right) \tag{5.16}$$

$$\pi_2^{\overline{\sim_{delay}}} = \left(\pi_2 \quad , \quad \texttt{funExt}\ \lambda\,(a, b, p), \begin{cases} (\texttt{inr}\ r, \lambda\,()) & p = \texttt{now}{\sim}\ r \\ (\texttt{inl}\ \texttt{tt}, \lambda\,\_, y) & p = \texttt{later}{\sim}\ x\ y\ q \end{cases}\right) \tag{5.17}$$

$$\tag{5.18}$$

This defines the bisimulation, given by the diagram in Figure 5.2. □

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

Figure 5.2: Strong bisimulation for delay monad

5.2.4 Bisimulation of ITrees

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

$$\frac{a, b : R \quad a \equiv_R b}{\texttt{Ret}\ a \cong \texttt{Ret}\ b}\ \texttt{EqRet} \tag{5.19}$$

$$\frac{t, u : \texttt{itree E}\ R \quad t \cong u}{\texttt{Tau}\ t \cong \texttt{Tau}\ u}\ \texttt{EqTau} \tag{5.20}$$

$$\frac{A : \mathcal{U} \quad e : \texttt{E}\ A \quad k_1, k_2 : A \to \texttt{itree E}\ R \quad t \cong u}{\texttt{Vis}\ e\ k_1 \cong \texttt{Tau}\ e\ k_2}\ \texttt{EqVis} \tag{5.21}$$

Now we just need to define $\alpha_{\mathcal{R}}$. Now we have a bisimulation relation, which is equivalent to equality, using what we showed in the previous section.

define the α_R function

5.2.5 Zip Function

We want the diagram in Figure 5.3 to commute, meaning we get the computation rules

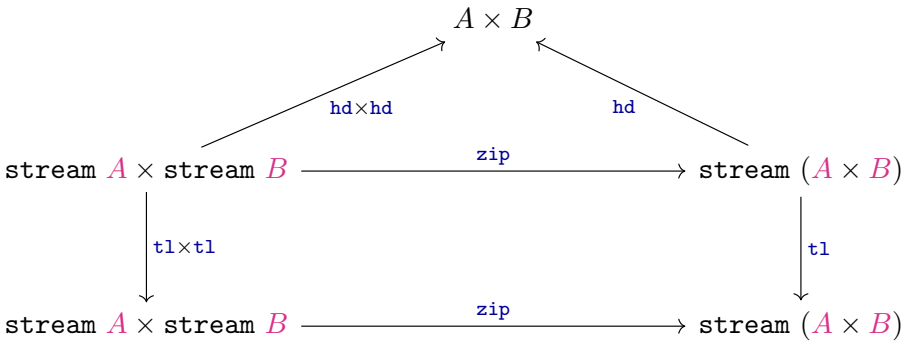

Figure 5.3: TODO

$$(\mathtt{hd} \times \mathtt{hd}) \equiv \mathtt{hd} \circ \mathtt{zip} \tag{5.22}$$

$$\mathtt{zip} \circ (\mathtt{tl} \times \mathtt{tl}) \equiv \mathtt{tl} \circ \mathtt{zip} \tag{5.23}$$

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

$$\mathtt{lift}_{\mathtt{M}} \left(x : \prod_{n:\mathbb{N}} (A \to \mathtt{P}_S{}^n\ \mathbf{1})\right) \left(u : \prod_{n:\mathbb{N}} (A \to \pi_n(x_{n+1}a) \equiv x_n a)\right) (a : A) : \mathtt{M}\ S := (\lambda\, n, x\ n\ a), (\lambda\, n\ i, p\ n\ a\ i). \tag{5.24}$$

To use this definition, we first define some helper functions

$$\mathtt{zip}_X\ n\ (x, y) = \begin{cases} \mathbf{1} & \mathtt{if}\ n = 0 \\ (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)), & \mathtt{if}\ n = m + 1 \end{cases} \tag{5.25}$$

$$\mathtt{zip}_\pi\ n\ (x, y) = \begin{cases} \mathtt{refl} & \mathtt{if}\ n = 0 \\ \lambda\, i, (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_\pi\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)\ i), & \mathtt{if}\ n = m + 1 \end{cases}, \tag{5.26}$$

we can then define

$$\mathtt{zip}_{lift}\ (x, y) := \mathtt{lift}_{\mathtt{M}}\ \mathtt{zip}_X\ \mathtt{zip}\ (x, y). \tag{5.27}$$

Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\begin{aligned} \mathtt{transport}_?\ a &\equiv \mathtt{zip}_{lift}\ a && (5.28)\\ &\equiv \mathtt{lift}_{\mathtt{M}}\ \mathtt{zip}_X\ \mathtt{zip}_\pi\ (x, y) && (5.29)\\ &\equiv (\lambda\, n, \mathtt{zip}_X\ n\ (x, y)), (\lambda\, n\ i, \mathtt{zip}_\pi\ n\ (x, y)\ i) && (5.30) \end{aligned}$$

zero case X

$$\mathtt{zip}_X\ 0\ (x, y) \equiv \mathbf{1} \tag{5.31}$$

Successor case X

$$\begin{aligned} \mathtt{zip}_X\ (m + 1)\ (x, y) &\equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)) && (5.32)\\ &\equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda\, \_, ?\ (\mathtt{tl}\ a)) && (5.33)\\ &\equiv (\mathtt{hd}\ (\mathtt{transport}_? a)), (\lambda\, \_, \mathtt{transport}_?(\mathtt{tl}\ a)) && (5.34)\\ &\equiv \mathtt{transport}_?\ a && (5.35)\\ & && (5.36) \end{aligned}$$

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

$$\begin{aligned} &\equiv (), (\lambda\, i, \mathtt{zip}_\pi\ 0\ (x, y)\ i) && (5.37)\\ &\equiv \mathbf{1}, \mathtt{refl} && (5.38)\\ & && (5.39) \end{aligned}$$

successor case

$$\equiv (\texttt{zip}_X\ (m+1)\ (x,y)), (\lambda\, i, \texttt{zip}_\pi\ (m+1)\ (x,y)\ i) \tag{5.40}$$

$$\equiv ((\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\, \_, \texttt{zip}_X\ m\ (\texttt{tl}\ x, \texttt{tl}\ y))), (\lambda\, i, (\texttt{hd}\ x, \texttt{hd}\ y), (\lambda\, \_, \texttt{zip}_\pi\ m\ (\texttt{tl}\ x, \texttt{tl}\ y)\ i)) \tag{5.41}$$

Complete this proof

5.2.6 Examples of Fixed Points

Zeros

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

$$\texttt{const}_\texttt{X}\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \mathbf{1} & n = 0 \\ (c, \lambda\, \_, \texttt{const}_\texttt{X}\ m\ c) & n = m+1 \end{cases} \tag{5.42}$$

$$\texttt{const}_\pi\ (n : \mathbb{N})\ (c : \mathbb{N}) := \begin{cases} \texttt{refl} & n = 0 \\ \lambda\, i, (c, \lambda\, \_, \texttt{const}_\pi\ m\ c\ i) & n = m+1 \end{cases} \tag{5.43}$$

to get the definition of zero stream

$$\texttt{zeros} := \texttt{lift}_\texttt{M}\ \texttt{const}_\texttt{X}\ \texttt{const}_\pi\ 0. \tag{5.44}$$

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

$$\texttt{hd}\ \texttt{zeros} \equiv 0 \tag{5.45}$$

$$\texttt{tl}\ \texttt{zeros} \equiv \texttt{zeros} \tag{5.46}$$

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 M-type (which is a final coalgebra)

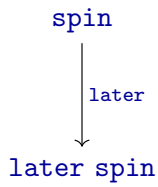

Figure 5.4: TODO

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

5.3 Stream Formalization using M-types

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

$$\texttt{stream}\ A := \texttt{M}_{(A\,,\lambda\_,\mathbf{1})} \tag{5.47}$$

When taking the head of a stream, we get

$$\begin{aligned}
\texttt{hd}\ (\texttt{cons}\ x\ xs) &\equiv \pi_1\ \texttt{out}\ (\texttt{cons}\ x\ xs) && (5.48)\\
&\equiv \pi_1\ \texttt{out}\ (\texttt{in}\ (x, \lambda_, xs)) && (5.49)\\
&\equiv \pi_1\ (x, \lambda_, xs) && (5.50)\\
&\equiv x && (5.51)
\end{aligned}$$

and similarly for the tail of the stream

$$\begin{aligned}
\texttt{tl}\ (\texttt{cons}\ x\ xs) &\equiv \pi_2\ \texttt{out}\ (\texttt{cons}\ x\ xs) && (5.52)\\
&\equiv \pi_2\ \texttt{out}\ (\texttt{in}\ (x, \lambda_, xs)) && (5.53)\\
&\equiv \pi_2\ (x, \lambda_, xs) && (5.54)\\
&\equiv xs && (5.55)
\end{aligned}$$

and the other direction is also true

$$\begin{aligned}
\texttt{cons}(\texttt{hd}\ s, \texttt{tl}\ s) &\equiv \texttt{in}\ (\texttt{hd}\ s, \texttt{tl}\ s) && (5.56)\\
&\equiv \texttt{in}\ (\pi_1\ (\texttt{out}\ s), \pi_2\ (\texttt{out}\ s)) && (5.57)\\
&\equiv \texttt{in}\ (\texttt{out}\ s) && (5.58)\\
&\equiv s. && (5.59)
\end{aligned}$$

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

$$\texttt{cons}' f = \texttt{lift}_{\texttt{M}}\ (\lambda c\, n, f\ c) \tag{5.60}$$

$$\texttt{cons}' f = \texttt{lift}_{\texttt{M}}\ (\lambda c\, n, f\ c) \tag{5.61}$$

5.4 ITrees as M-types

We want the following rules for ITrees

$$\frac{r : R}{\texttt{Ret}\ r : \texttt{itree E}\ R}\ \texttt{I}_{\texttt{Ret}} \tag{5.62}$$

$$\frac{A : \mathcal{U} \quad a : \texttt{E}\ A \quad f : A \to \texttt{itree E}\ R}{\texttt{Vis}\ a\ f : \texttt{itree E}\ R}\ \texttt{I}_{\texttt{Vis}}. \tag{5.63}$$

Elimination rules

$$\frac{t : \texttt{itree E}\ R}{\texttt{Tau}\ t : \texttt{itree E}\ R}\ \texttt{E}_{\texttt{Tau}}. \tag{5.64}$$

5.4.1 Delay Monad

check this statement

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

$$\mathtt{P}_S(X) = \sum_{x:\mathbf{1}+R} \lambda\{\texttt{inl}\ \_ \to \mathbf{1}; \texttt{inr}\ \_ \to \mathbf{0}\}\ x \to X, \tag{5.65}$$

which is equal to

$$\mathtt{P}_S(X) = X + R \times (\mathbf{0} \to X). \tag{5.66}$$

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

$$\mathtt{P}_S(X) = X + R \tag{5.67}$$

meaning we get the following diagram.

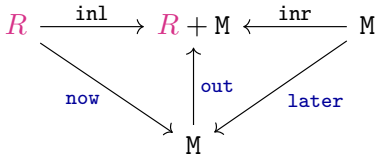

Figure 5.5: Delay monad

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

(Later = Tau, Ret = Now)

5.4.2 Tree

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

$$S = \left(R + \sum_{A:\mathcal{U}} \mathtt{E}\ A\ ,\ \lambda\{\texttt{inl}\ \_ \to \mathbf{0}\ ;\ \texttt{inr}\ (A, e) \to A\}\right). \tag{5.68}$$

This will give us the polynomial functor

$$\mathtt{P}_S(X) = \sum_{x:R+\sum_{A:\mathcal{U}} \mathtt{E}\ A} \lambda\{\texttt{inl}\ \_ \to \mathbf{0}\ ;\ \texttt{inr}\ (A, e) \to A\}\ x \to X, \tag{5.69}$$

which simplifies to

$$\mathtt{P}_S(X) = (R \times (\mathbf{0} \to X)) + (\sum_{A:\mathcal{U}} \mathtt{E}\ A \times (A \to X)), \tag{5.70}$$

and further

$$\mathtt{P}_S(X) = R + \sum_{A:\mathcal{U}} \mathtt{E}\ A \times (A \to X). \tag{5.71}$$

We get the following diagram for the P-coalgebra.
Again we can define `Ret` and `Vis` using the `in` function.

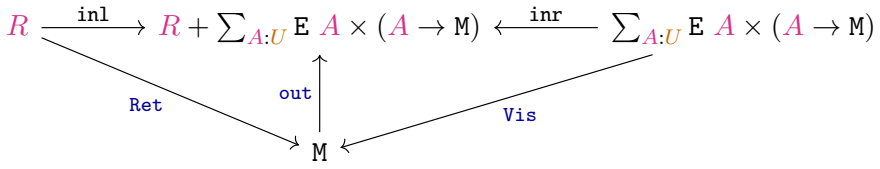

Figure 5.6: TODO

5.4.3 ITrees

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

$$S = \left(\mathbf{1} + R + \sum_{A:\mathcal{U}} (\mathtt{E}\ A) \quad , \quad \lambda \{ \mathtt{inl}\ (\mathtt{inl}\ \_) \to \mathbf{1}\ ;\ \mathtt{inl}\ (\mathtt{inr}\ \_) \to \mathbf{0}\ ;\ \mathtt{inr}(A, \_) \to A \} \right). \tag{5.72}$$

Such that the (reduced) polynomial functor becomes

$$\mathtt{P}_S(X) = X + R + \sum_{A:\mathcal{U}} ((\mathtt{E}\ A) \times (A \to X)) \tag{5.73}$$

Giving us the diagram

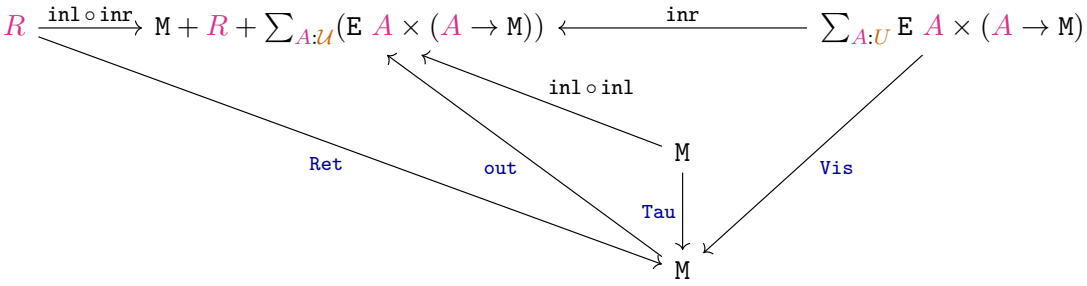

Figure 5.7: TODO

Chapter 6

Additions to the Cubical Agda Library

6.1 Σap

$$\texttt{Σap} : \left(\sum_{x:X} Y\,x \equiv \sum_{x':X'} Y'\,x'\right) \equiv \left(\sum_{p:X\equiv X'} Y \equiv_p Y'\right) \tag{6.1}$$

Describe the proof of this? / Is this relevant / Should it be in the appendix?

Chapter 7

Conclusion

conclude on the problem statement from the introduction

Bibliography

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

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

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

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

Appendix A

The Technical Details