
M-types and Coinduction in HoTT and Cubical Type Theory

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science

April 9, 2020

Advisor: Bas Spitters

Abstract

in English...

Resumé

in Danish...

Acknowledgments

...

*Lasse Letager Hansen,
Aarhus, April 9, 2020.*

Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	1
2 M-types	3
2.1 Containers / Signatures	3
2.2 Closure properties of M-types	5
2.2.1 Product of M-types	5
2.3 Co-induction Principle for M-types	7
2.4 Quotient M-type	7
2.5 TODO	8
3 Contractive	9
4 Examples of M-types	11
4.1 TODO: Place these subsections	11
4.1.1 Identity Bisimulation	11
4.1.2 Bisimulation of Streams	11
4.1.3 Bisimulation of Delay monad	12
4.1.4 Bisimulation of ITrees	13
4.1.5 Zip Function	13
4.1.6 Examples of Fixed Points	14
4.2 Stream Formalization using M-types	15
4.3 ITrees as M-types	16
4.3.1 Delay Monad	16
4.3.2 Tree	17
4.3.3 ITrees	17
5 Additions to the Cubical Agda Library	19
5.1 Zap	19
6 Conclusion	21

Bibliography	23
A The Technical Details	25

Chapter 1

Introduction

motivate and explain the problem to be addressed

example of a citation: [1]

get your bibtex entries from <https://dblp.org/>

Chapter 2

M-types

2.1 Containers / Signatures

Definition 2.1.1. 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

$$\begin{aligned} P_S &: \mathcal{U} \rightarrow \mathcal{U} \\ P(X) &:= P_S(X) = \sum_{a:A} B(a) \rightarrow X \end{aligned} \quad (2.1)$$

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

$$\begin{aligned} Pf &: PX \rightarrow PY \\ Pf(a, g) &= (a, f \circ g). \end{aligned} \quad (2.2)$$

Example 2.1.1. The type for streams over the type A is defined by the container $S = (A, \lambda _, \mathbf{1})$, applying the polynomial functor we get

$$P_S(X) = \sum_{a:A} \mathbf{1} \rightarrow X, \quad (2.3)$$

since we are working in a Category with exponentials, we get $\mathbf{1} \rightarrow X \equiv X^{\mathbf{1}} \equiv X$. Furthermore $\mathbf{1}$ and X does not depend on A , so this will be equivalent to the definition

$$P_S(X) = A \times X. \quad (2.4)$$

Definition 2.1.2. The coalgebra for the functor P is defined as

$$\text{Coalg}_S = \sum_{C:\mathcal{U}} C \rightarrow PC. \quad (2.5)$$

We denote a coalgebra of C and γ as $C\text{-}\gamma$. The coalgebra morphisms are defined as

$$\begin{aligned} \cdot \Rightarrow \cdot &: \text{Coalg}_S \rightarrow \text{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta &= \sum_{f:C \rightarrow D} \delta \circ f = Pf \circ \gamma \end{aligned} \quad (2.6)$$

Definition 2.1.3. M-types can now be defined from a container S as the type M_S such that the coalgebra for M_S and $\text{out} : M_S \rightarrow P_S(M_S)$ fulfills the property

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

that is $\prod_{(C-\gamma:\text{Coalg}_S)} \text{isContr}(C-\gamma \Rightarrow M_S-\text{out})$. We denote this construction of the M-type as $M_{(A,B)}$ or M_S or just M when the Container is clear from the context.

Example 2.1.2. If we continue our example for streams this will give us the M-type, we can see that $P_S(M) = A \times M$, meaning we have the following diagram, where out is an isomorphism (because

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times M_{(A,\lambda_,1)} & \xrightarrow{\pi_2} & M_{(A,\lambda_,1)} \\ & \searrow \text{hd} & \uparrow \text{out} & \nearrow \text{tl} & \\ & & M_{(A,\lambda_,1)} & & \end{array}$$

Figure 2.1: M-types of streams

of the finality of the coalgebra), with inverse $\text{in} : P_S(M) \rightarrow M$. We now have a semantic for the rules we would expect for streams, if we let $\text{cons} = \text{in}$ and $\text{stream } A = M_{(A,\lambda_,1)}$,

$$\frac{A:\mathcal{U} \quad s:\text{stream } A}{\text{hd } s : A} E_{\text{hd}} \quad (2.8)$$

$$\frac{A:\mathcal{U} \quad s:\text{stream } A}{\text{tl } s : \text{stream } A} E_{\text{tl}} \quad (2.9)$$

$$\frac{A:\mathcal{U} \quad x:A \quad xs:\text{stream } A}{\text{cons } x \, xs : \text{stream } A} I_{\text{cons}} \quad (2.10)$$

Lemma 2.1.1. Given $\ell : \prod_{(n:\mathbb{N})} (X_n \rightarrow X_{n+1})$ and $y : \sum_{x:\prod_{(n:\mathbb{N})} X_n} x_{n+1} \equiv l_n(x_n)$ we can collapse the chain $\mathcal{L} \equiv X_0$. We define this collapse by the equivalence

$$\text{fun}_{\mathcal{L}\text{collapse}}(x, r) = x_0 \quad (2.11)$$

$$\text{inv}_{\mathcal{L}\text{collapse}} x_0 = (\lambda n, \text{transport } \ell^n x_0), (\lambda n, \text{refl}_{\text{transport } \ell^{(n+1)} x_0}) \quad (2.12)$$

$$\text{rinv } x_0 = \text{refl}_{x_0} \quad (2.13)$$

$$(2.14)$$

For $\text{linv}(x, r)$ 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 $y \equiv \text{inv}_{\mathcal{L}\text{collapse}}(\text{fun}_{\mathcal{L}\text{collapse}} y)$, since both are equal to such sections.

Theorem 2.1.2. We want to proof the construction of in and out for the container (A, B) more precisely we want to define the equality

$$\text{shift} : \mathcal{L} \equiv P\mathcal{L} \quad (2.15)$$

where $P\mathcal{L}$ is the limit of a shifted sequence. We do this we the two helper lemmas

$$\alpha : \mathcal{L}^P \equiv P\mathcal{L} \quad (2.16)$$

$$\mathcal{L}_{\text{unique}} : \mathcal{L} \equiv \mathcal{L}^P \quad (2.17)$$

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

$$\text{fun}_{\mathcal{L}_{\text{unique}}} (a, b) = \left(\lambda n, \begin{cases} \text{tt} & n = 0 \\ a \ m & n = m + 1 \end{cases} \right), \left(\lambda n \begin{cases} \text{refl}_{\text{tt}} & n = 0 \\ b \ m & n = m + 1 \end{cases} \right) \quad (2.18)$$

$$\text{inv}_{\mathcal{L}_{\text{unique}}} (a, b) = a \circ \text{incr}, b \circ \text{incr} \quad (2.19)$$

$$\text{rinv}_{\mathcal{L}_{\text{unique}}} (a, b) = \text{refl} \quad (2.20)$$

$$\text{linv}_{\mathcal{L}_{\text{unique}}} (a, b) = \text{refl} \quad (2.21)$$

The definition of α follows from Lemma 2.1.1,

$$\mathcal{L}^P \equiv \sum_{(u : \mathcal{L}_{(\lambda _, A, \text{id})})} \prod_{(n : \mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv_{\lambda x, B(\pi_2 a \ n \ x) \rightarrow X_n} u_n \quad (2.22)$$

$$\equiv \sum_{(a : A)} \sum_{(u : \prod_{(n : \mathbb{N})} B a \rightarrow X_n)} \prod_{(n : \mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv u_n \quad (2.23)$$

$$\equiv P\mathcal{L} \quad (2.24)$$

We can then define *shift* as

$$\text{shift} = \text{sym } \alpha \cdot \mathcal{L}_{\text{unique}} \quad (2.25)$$

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

2.2 Closure properties of \mathbf{M} -types

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

2.2.1 Product of \mathbf{M} -types

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

Definition 2.2.1. The product of two containers is defined as

$$(A, B) \times (C, D) \equiv (A \times C, \lambda (a, c), B \ a \times D \ c). \quad (2.26)$$

We can lift this rule, through the diagram in Figure 2.2, used to define \mathbf{M} -types.

Theorem 2.2.1. For any $n : \mathbb{N}$ the following is true

$$P_{(A, B)}^n \ \mathbf{1} \times P_{(C, D)}^n \ \mathbf{1} \equiv P_{(A, B) \times (C, D)}^n \ \mathbf{1}. \quad (2.27)$$

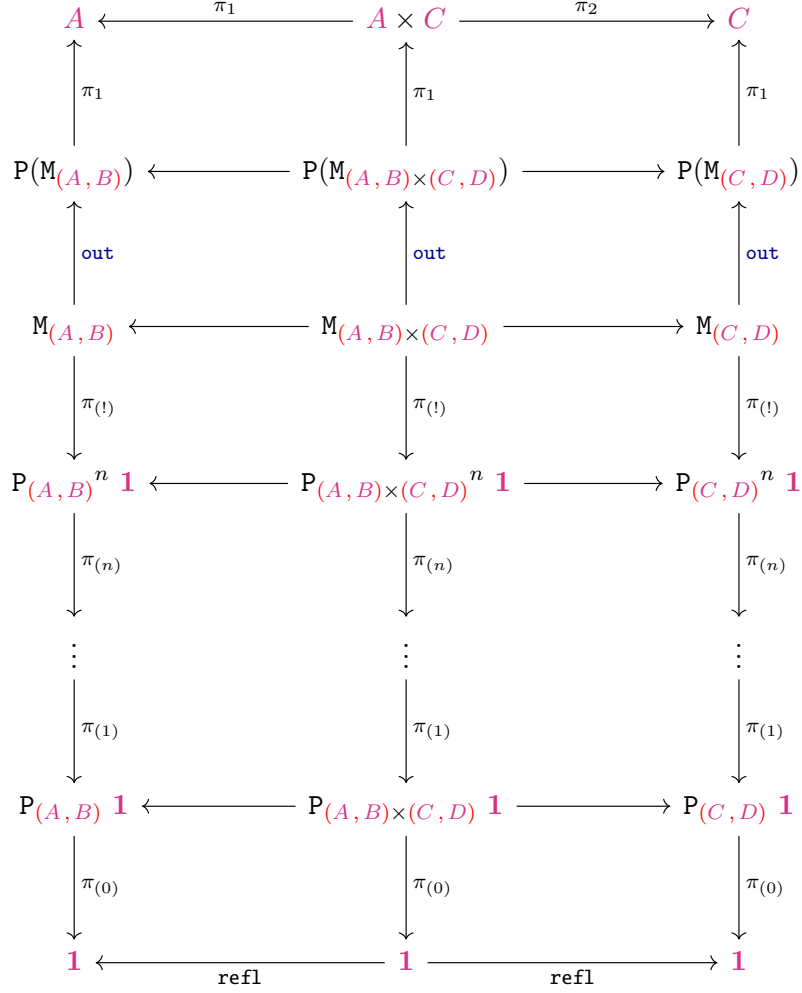


Figure 2.2: Diagram for products of chains

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

$$P_{(A,B)}^m \mathbf{1} \times P_{(C,D)}^m \mathbf{1} \equiv P_{(A,B) \times (C,D)}^m \mathbf{1}, \quad (2.28)$$

in the following

$$P_{(A,B)}^{m+1} \mathbf{1} \times P_{(C,D)}^{m+1} \mathbf{1} \quad (2.29)$$

$$\equiv P_{(A,B)}(P_{(A,B)}^m \mathbf{1}) \times P_{(C,D)}(P_{(C,D)}^m \mathbf{1}) \quad (2.30)$$

$$\equiv \sum_{a:A} B \ a \rightarrow P_{(A,B)}^m \mathbf{1} \times \sum_{c:C} D \ c \rightarrow P_{(C,D)}^m \mathbf{1} \quad (2.31)$$

$$\equiv \sum_{a,c:A \times C} (B \ a \rightarrow P_{(A,B)}^m \mathbf{1}) \times (D \ c \rightarrow P_{(C,D)}^m \mathbf{1}) \quad (2.32)$$

$$\equiv \sum_{a,c:A \times C} B \ a \times D \ c \rightarrow P_{(A,B)}^m \mathbf{1} \times P_{(C,D)}^m \mathbf{1} \quad (2.33)$$

$$\equiv \sum_{a,c:A \times C} B \ a \times D \ c \rightarrow P_{(A,B) \times (C,D)}^m \mathbf{1} \quad (2.34)$$

$$\equiv P_{(A,B) \times (C,D)}(P_{(A,B) \times (C,D)}^m \mathbf{1}) \quad (2.35)$$

$$\equiv P_{(A,B) \times (C,D)}^{m+1} \mathbf{1} \quad (2.36)$$

□

taking the limit of (2.27) we get

$$M_{(A,B)} \times M_{(C,D)} \equiv M_{(A,B) \times (C,D)}. \quad (2.37)$$

Example 2.2.1. For streams we get

$$\text{stream } A \times \text{stream } B \equiv M_{(A, \lambda _ , \mathbf{1})} \times M_{(B, \lambda _ , \mathbf{1})} \equiv M_{(A, \lambda _ , \mathbf{1}) \times (B, \lambda _ , \mathbf{1})} \equiv \text{stream } (A \times B) \quad (2.38)$$

as expected. Transporting along (2.38) gives us a definition for [zip](#).

2.3 Co-induction Principle for M-types

We can now construct a bisimulation

Definition 2.3.1. For all coalgebras $C-\gamma : \text{Coalg}_S$, given a relation $\mathcal{R} : C \rightarrow C \rightarrow \mathcal{U}$ and a type $\overline{\mathcal{R}} = \sum_{a:C} \sum_{b:C} a \ \mathcal{R} \ b$, such that $\overline{\mathcal{R}}$ and $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \rightarrow P(\overline{\mathcal{R}})$ forms a P-coalgebra $\overline{\mathcal{R}}-\alpha_{\mathcal{R}} : \text{Coalg}_S$, making the diagram in Figure 2.3 commute (\Rightarrow represents P-coalgebra morphisms).

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

Figure 2.3: Bisimulation for a coalgebra

Definition 2.3.2 (Co-induction principle). Given a relation \mathcal{R} , that is part of a bisimulation over a final P-coalgebra $M-\text{out} : \text{Coalg}_S$ we get the diagram in Figure 2.4,

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

Figure 2.4: 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

$$m = \pi_1 \overline{\mathcal{R}}(m, m', r) = \pi_2 \overline{\mathcal{R}}(m, m', r) = m'. \quad (2.39)$$

What makes a relation a bisimulation? Is bisim and equality equal. Try to complete the construction of equality from any bisimulation relation over an M-type

2.4 Quotient M-type

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 as

Quotients of the delay monad

2.5 TODO

- Resumption Monad transformer
- coinduction in Coq is broken
- bisim \Rightarrow eq
- copattern matching
- cubical agda. Relation between \mathbf{M} -types defined by coinduction/copattern matching and constructed from W-types
- In agda, coinductive 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) \mathbf{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 \mathbf{QM} -types, and what are those constructively ??

Chapter 3

Contractive

We want to show that

$$\begin{aligned}
 & \sum_{(a:A)} \sum_{(u:\prod_{(n:\mathbb{N})} B \ a \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ (u_{n+1}) \equiv u_n \tag{3.1} \\
 \equiv & \sum_{(a:\sum_{(a:(n:\mathbb{N}) \rightarrow A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(u:(n:\mathbb{N}) \rightarrow B \ (\pi_1 \ a)_n \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv_{(\lambda x, B \ ((\pi_2 \ a)_n \ x) \rightarrow X_n)} u_n \tag{3.2}
 \end{aligned}$$

we have the two functions

$$\text{fun } (x, y) := ((\lambda n, x), (\lambda n \ i, x)), y \tag{3.3}$$

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

$$\text{inv } ((x, y), (z, w)) := x_0, ((\lambda n \ a, z \ n \ (\text{subst } B \ y_{(!n)} \ a)), (\lambda n \ i \ a, w \ n \ i \ (\text{subst } B \ ? \ a))) \tag{3.4}$$

$$\begin{array}{ccc}
 x, (z, w) & \xrightarrow{\text{fun}} & (\lambda n, x, \lambda n \ i, x), (z, w) \\
 \downarrow \text{refl} & & \downarrow \text{inv} \\
 x, (z, w) & \xleftarrow{\text{refl}} & x, ((\lambda n \ a, z \ n \ (\text{transport } y? \ a)), (\lambda n \ i \ a, w \ n \ i \ (\text{transport } ? \ a)))
 \end{array}$$

Figure 3.1: Left inverse

so we want

$$\text{invhelper0 } ((\lambda n, (x, y), \lambda n \ i, (x, y)), (z, w)) \tag{3.5}$$

$$\equiv \lambda n, \left\{ \begin{array}{l} \text{tt} \quad n = 0 \end{array} \right. \tag{3.6}$$

$$\equiv z \tag{3.7}$$

$$\text{invhelper1 } ((\lambda n, (x, y), \lambda n \ i, (x, y)), (z, w)) \equiv w \tag{3.8}$$

$$\begin{array}{ccc}
(x, y), (z, w) & \xrightarrow{\text{inv}} & x_0, (\text{subst } B ? z, \text{subst } B ? w) \\
\downarrow \text{refl} & & \downarrow \text{fun} \\
(x, y), (z, w) & \xleftarrow{\text{refl}} & ((\lambda n, x_0), (\lambda n i, x_0)), (\text{subst } B ? z, \text{subst } B ? w)
\end{array}$$

Figure 3.2: Right inverse

Chapter 4

Examples of **M**-types

4.1 TODO: Place these subsections

4.1.1 Identity Bisimulation

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

Lemma 4.1.1. *The identity relation $(\cdot \equiv \cdot)$ is a bisimulation for any final coalgebra $\mathbf{M}_{\text{S-out}}$ defined over an **M**-type.*

Proof. We first define the function

$$\begin{aligned} \alpha_{\equiv} : \equiv &\rightarrow \mathbf{P}(\equiv) \\ \alpha_{\equiv}(x, y) &:= \pi_1(\text{out } x), (\lambda b, (\pi_2(\text{out } x) b, \text{refl}_{(\pi_2(\text{out } x) b)})) \end{aligned} \quad (4.1)$$

and the two projections

$$\pi_1^{\equiv} = \text{funExt } \lambda(a, b, r), \text{refl}_{\text{out } a} \quad (4.2)$$

$$\pi_2^{\equiv} = \text{funExt } \lambda(a, b, r), \text{cong}_{\text{out}}(\text{sym } r). \quad (4.3)$$

This defines the bisimulation, given by the diagram in Figure 4.1. □

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

Figure 4.1: Identity bisimulation

4.1.2 Bisimulation of Streams

TODO

4.1.3 Bisimulation of Delay monad

We want to define a bisimulation relation \approx_{delay} for the delay monad,

Definition 4.1.1. The relation \approx_{delay} is defined by the following rules

$$\frac{R : \mathcal{U} \quad r : R}{\text{now } r \approx_{\text{delay}} \text{now } r : \mathcal{U}} \text{EqNow} \quad (4.4)$$

$$\frac{R : \mathcal{U} \quad t : \text{delay } R \quad u : \text{delay } R \quad t \approx_{\text{delay}}^{\infty} u : \mathcal{U}}{\text{later } t \approx_{\text{delay}} \text{later } u : \mathcal{U}} \text{EqLater} \quad (4.5)$$

where $\approx_{\text{delay}}^{\infty}$ is defined as

$$\frac{x \approx_{\text{delay}} y}{x \approx_{\text{delay}}^{\infty} y} \text{force} \quad (4.6)$$

Lemma 4.1.2. The relation \approx_{delay} is reflexive, meaning $\forall (x : \text{delay } R), x \approx_{\text{delay}} x$.

Proof. We do case analysis on x . We start by $x = \text{now } r$ meaning we have to show

$$\text{now } r \approx_{\text{delay}} \text{now } r \quad (4.7)$$

which is true, by **EqNow**. For $x = \text{later } t$, we get

$$\text{later } t \approx_{\text{delay}} \text{later } t \quad (4.8)$$

which by **EqLater**, becomes $t \approx_{\text{delay}}^{\infty} t$. Meaning we must show $\approx_{\text{delay}}^{\infty}$ is reflexive, but this follows directly by **force** the induction hypothesis that \approx_{delay} is reflexive. \square

Theorem 4.1.3. The relation \approx_{delay} is a bisimulation for delay R .

Proof. First we define the function

$$\begin{aligned} \alpha_{\approx_{\text{delay}}} : \overline{\approx_{\text{delay}}} &\rightarrow \mathbf{P}(\overline{\approx_{\text{delay}}}) \\ \alpha_{\approx_{\text{delay}}}(x, y) &:= \pi_1(\text{out } x), (\lambda b, (\pi_2(\text{out } x) b, \text{refl}_{(\pi_2(\text{out } x) b)})) \end{aligned} \quad (4.9)$$

then we define the projections

$$\pi_1^{\equiv} = \text{funExt } \lambda(a, b, r), \text{refl}_{\text{out } a} \quad (4.10)$$

$$\pi_2^{\equiv} = \text{funExt } \lambda(a, b, r), \text{cong}_{\text{out}}(R_{eq} r), \quad (4.11)$$

where

$$R_{eq}(a, b, r) = \begin{cases} \text{refl} & r = \text{EqNow} \\ \text{refl} & r = \text{EqLater } r' \end{cases} \quad (4.12)$$

\square

4.1.4 Bisimulation of ITrees

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

$$\frac{a, b : R \quad a \equiv_R b}{\text{Ret } a \cong \text{Ret } b} \text{EqRet} \quad (4.13)$$

$$\frac{t, u : \text{itree } E \quad R \quad t \cong u}{\text{Tau } t \cong \text{Tau } u} \text{EqTau} \quad (4.14)$$

$$\frac{A : \mathcal{U} \quad e : E \quad A \quad k_1, k_2 : A \rightarrow \text{itree } E \quad R \quad t \cong u}{\text{Vis } e \, k_1 \cong \text{Tau } e \, k_2} \text{EqVis} \quad (4.15)$$

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

4.1.5 Zip Function

We want the diagram in Figure 4.2 to commute, meaning we get the computation rules

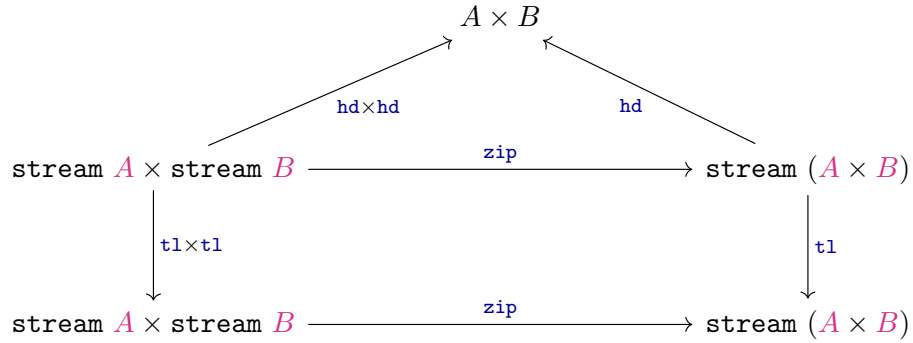


Figure 4.2: TODO

$$(\text{hd} \times \text{hd}) \equiv \text{hd} \circ \text{zip} \quad (4.16)$$

$$\text{zip} \circ (\text{tl} \times \text{tl}) \equiv \text{tl} \circ \text{zip} \quad (4.17)$$

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 \mathbf{M} -types

$$\text{lift}_{\mathbf{M}} \left(x : \prod_{n:\mathbb{N}} (A \rightarrow \text{P}_{\mathcal{S}}^n \mathbf{1}) \right) \left(u : \prod_{n:\mathbb{N}} (A \rightarrow \pi_n(x_{n+1}a) \equiv x_n a) \right) (a : A) : \mathbf{M} \, \mathcal{S} := (\lambda n, x \, n \, a), (\lambda n \, i, p \, n \, a \, i). \quad (4.18)$$

To use this definition, we first define some helper functions

$$\text{zip}_X \, n \, (x, y) = \begin{cases} \mathbf{1} & \text{if } n = 0 \\ (\text{hd } x, \text{hd } y), (\lambda _, \text{zip}_X \, m \, (\text{tl } x, \text{tl } y)), & \text{if } n = m + 1 \end{cases} \quad (4.19)$$

$$\mathbf{zip}_\pi n (x, y) = \begin{cases} \mathbf{refl} & \text{if } n = 0 \\ \lambda i, (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_\pi m (\mathbf{tl} x, \mathbf{tl} y) i), & \text{if } n = m + 1 \end{cases}, \quad (4.20)$$

we can then define

$$\mathbf{zip}_{lift} (x, y) := \mathbf{lift}_M \mathbf{zip}_X \mathbf{zip} (x, y). \quad (4.21)$$

Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\mathbf{transport}_? a \equiv \mathbf{zip}_{lift} a \quad (4.22)$$

$$\equiv \mathbf{lift}_M \mathbf{zip}_X \mathbf{zip}_\pi (x, y) \quad (4.23)$$

$$\equiv (\lambda n, \mathbf{zip}_X n (x, y)), (\lambda n i, \mathbf{zip}_\pi n (x, y) i) \quad (4.24)$$

zero case X

$$\mathbf{zip}_X 0 (x, y) \equiv \mathbf{1} \quad (4.25)$$

Successor case X

$$\mathbf{zip}_X (m + 1) (x, y) \equiv (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_X m (\mathbf{tl} x, \mathbf{tl} y)) \quad (4.26)$$

$$\equiv (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, ? (\mathbf{tl} a)) \quad (4.27)$$

$$\equiv (\mathbf{hd} (\mathbf{transport}_? a)), (\lambda _, \mathbf{transport}_? (\mathbf{tl} a)) \quad (4.28)$$

$$\equiv \mathbf{transport}_? a \quad (4.29)$$

$$(4.30)$$

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

$$\equiv (), (\lambda i, \mathbf{zip}_\pi 0 (x, y) i) \quad (4.31)$$

$$\equiv \mathbf{1}, \mathbf{refl} \quad (4.32)$$

$$(4.33)$$

successor case

$$\equiv (\mathbf{zip}_X (m + 1) (x, y)), (\lambda i, \mathbf{zip}_\pi (m + 1) (x, y) i) \quad (4.34)$$

$$\equiv ((\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_X m (\mathbf{tl} x, \mathbf{tl} y))), (\lambda i, (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_\pi m (\mathbf{tl} x, \mathbf{tl} y) i)) \quad (4.35)$$

Complete this proof

4.1.6 Examples of Fixed Points

Zeros

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

$$\mathbf{const}_X (n : \mathbb{N}) (c : \mathbb{N}) := \begin{cases} \mathbf{1} & n = 0 \\ (c, \lambda _, \mathbf{const}_X m c) & n = m + 1 \end{cases} \quad (4.36)$$

$$\text{const}_\pi (n : \mathbb{N}) (c : \mathbb{N}) := \begin{cases} \text{refl} & n = 0 \\ \lambda i, (c, \lambda _, \text{const}_\pi m c i) & n = m + 1 \end{cases} \quad (4.37)$$

to get the definition of zero stream

$$\text{zeros} := \text{lift}_M \text{const}_X \text{const}_\pi 0. \quad (4.38)$$

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

$$\text{hd zeros} \equiv 0 \quad (4.39)$$

$$\text{tl zeros} \equiv \text{zeros} \quad (4.40)$$

Spin

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

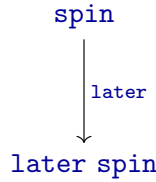


Figure 4.3: TODO

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

4.2 Stream Formalization using \mathbf{M} -types

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

$$\text{stream } A := M_{(A, \lambda _, 1)} \quad (4.41)$$

When taking the head of a stream, we get

$$\text{hd} (\text{cons } x \text{ } xs) \equiv \pi_1 \text{ out } (\text{cons } x \text{ } xs) \quad (4.42)$$

$$\equiv \pi_1 \text{ out } (\text{in } (x, \lambda _, xs)) \quad (4.43)$$

$$\equiv \pi_1 (x, \lambda _, xs) \quad (4.44)$$

$$\equiv x \quad (4.45)$$

and similarly for the tail of the stream

$$\text{tl} (\text{cons } x \text{ } xs) \equiv \pi_2 \text{ out } (\text{cons } x \text{ } xs) \quad (4.46)$$

$$\equiv \pi_2 \text{ out } (\text{in } (x, \lambda _, xs)) \quad (4.47)$$

$$\equiv \pi_2 (x, \lambda _, xs) \quad (4.48)$$

$$\equiv xs \quad (4.49)$$

and the other direction is also true

$$\text{cons}(\text{hd } s, \text{tl } s) \equiv \text{in } (\text{hd } s, \text{tl } s) \quad (4.50)$$

$$\equiv \text{in } (\pi_1 (\text{out } s), \pi_2 (\text{out } s)) \quad (4.51)$$

$$\equiv \text{in } (\text{out } s) \quad (4.52)$$

$$\equiv s. \quad (4.53)$$

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

$$\text{cons}' f = \text{lift}_{\mathbf{M}} (\lambda c n, f c) \quad (4.54)$$

$$\text{cons}' f = \text{lift}_{\mathbf{M}} (\lambda c n, f c) \quad (4.55)$$

4.3 ITrees as \mathbf{M} -types

We want the following rules for ITrees

$$\frac{r : R}{\text{Ret } r : \text{itree } E R} \text{I}_{\text{Ret}} \quad (4.56)$$

$$\frac{A : \mathcal{U} \quad a : E A \quad f : A \rightarrow \text{itree } E R}{\text{Vis } a f : \text{itree } E R} \text{I}_{\text{Vis}}. \quad (4.57)$$

Elimination rules

$$\frac{t : \text{itree } E R}{\text{Tau } t : \text{itree } E R} \text{E}_{\text{Tau}}. \quad (4.58)$$

4.3.1 Delay Monad

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 $S = (1 + R, \lambda\{\text{inl } _ \rightarrow 1; \text{inr } _ \rightarrow 0\})$, we then get the polynomial functor

$$\text{P}_S(X) = \sum_{x:1+R} \lambda\{\text{inl } _ \rightarrow 1; \text{inr } _ \rightarrow 0\} x \rightarrow X, \quad (4.59)$$

which is equal to

$$\text{P}_S(X) = X + R \times (0 \rightarrow X). \quad (4.60)$$

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

$$\text{P}_S(X) = X + R \quad (4.61)$$

meaning we get the following diagram.

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

check
this
state-
ment

(Later
= Tau,
Ret =
Now)

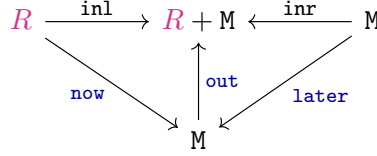


Figure 4.4: Delay monad

4.3.2 Tree

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

$$S = \left(R + \sum_{A:\mathcal{U}} E A, \lambda\{\text{inl } _ \rightarrow \mathbf{0} ; \text{inr } (A, e) \rightarrow A\} \right). \quad (4.62)$$

This will give us the polynomial functor

$$P_S(X) = \sum_{x:R + \sum_{A:\mathcal{U}} E A} \lambda\{\text{inl } _ \rightarrow \mathbf{0} ; \text{inr } (A, e) \rightarrow A\} x \rightarrow X, \quad (4.63)$$

which simplifies to

$$P_S(X) = (R \times (\mathbf{0} \rightarrow X)) + \left(\sum_{A:\mathcal{U}} E A \times (A \rightarrow X) \right), \quad (4.64)$$

and further

$$P_S(X) = R + \sum_{A:\mathcal{U}} E A \times (A \rightarrow X). \quad (4.65)$$

We get the following diagram for the P-coalgebra.

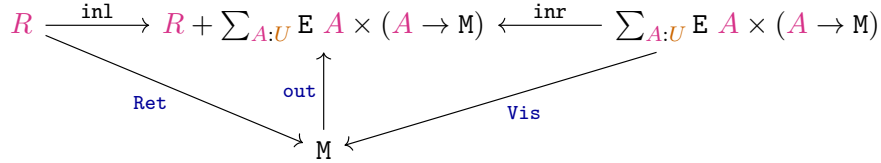


Figure 4.5: TODO

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

4.3.3 ITrees

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

$$S = \left(\mathbf{1} + R + \sum_{A:\mathcal{U}} (E A) , \lambda\{\text{inl } (\text{inl } _) \rightarrow \mathbf{1} ; \text{inl } (\text{inr } _) \rightarrow \mathbf{0} ; \text{inr } (A, _) \rightarrow A\} \right). \quad (4.66)$$

Such that the (reduced) polynomial functor becomes

$$P_S(X) = X + R + \sum_{A:\mathcal{U}} ((E A) \times (A \rightarrow X)) \quad (4.67)$$

Giving us the diagram

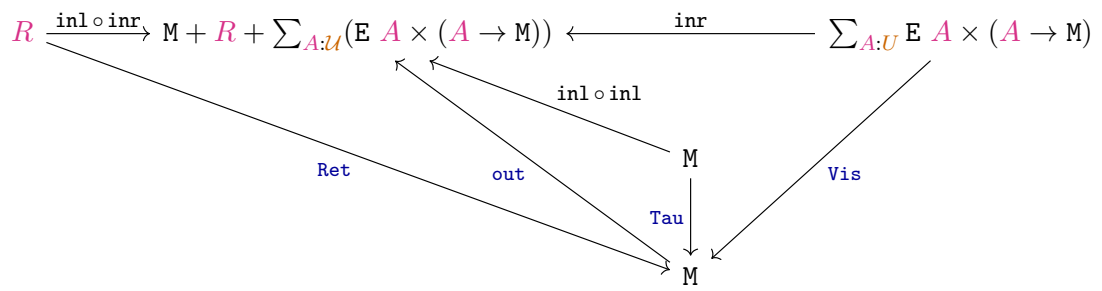


Figure 4.6: TODO

Chapter 5

Additions to the Cubical Agda Library

5.1 Σap

$$\Sigma\text{ap} : \left(\sum_{x:X} Yx \equiv \sum_{x':X'} Y'x' \right) \equiv \left(\sum_{p:X \equiv X'} Y \equiv_p Y' \right) \quad (5.1)$$

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

Chapter 6

Conclusion

conclude on the problem statement from the introduction

Bibliography

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

Appendix A

The Technical Details

