
M-types and Coinduction in HoTT and Cubical Type Theory

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science

May 21, 2020

Advisor: Bas Spitters

Abstract

in English...

Resumé

in Danish...

Acknowledgments



c

*Lasse Letager Hansen,
Aarhus, May 21, 2020.*

Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	1
2 Notation	3
3 Background Theory	5
3.1 Proof assistants	5
3.2 Set Theory	5
3.3 Type Theory	5
3.3.1 Martin L�f Type Theory / Intuitionistic type Theory (MLTT/ITT)	5
3.4 The Univalence Axiom	5
3.5 Homotopy Type Theory (HoTT) / Univalent Foundations (UF)	6
3.5.1 Judgmental Equality / Definitional Equality	6
3.5.2 Propositional Equality	6
3.5.3 External (of theory)	6
3.5.4 Internal (of theory)	6
3.5.5 Constructivity	6
3.5.6 Homotopy n -types	6
3.5.7 The HoTT Book	7
3.6 Higher order inductive types (HITs)	7
3.6.1 Propositional truncation and Set Truncated Quotients	7
3.7 Cubical Type Theory (CTT)	7
3.7.1 UIP	7
3.7.2 Cubical Agda	7
3.8 Coinduction	7
3.9 W-types	8
3.10 Problems with / Getting around using AC and LEM	8
3.11 Universes ??	8

4	M-types	9
4.1	Containers / Signatures	9
4.2	Coinduction Principle for M-types	12
5	Instantiation of M-types	15
5.1	Stream Formalization using M-types	15
5.2	ITrees as M-types	15
5.2.1	Delay Monad	15
5.2.2	Tree	16
5.2.3	ITrees	17
5.3	Automaton	17
6	QM-types	19
6.1	Quotienting and constructors	19
6.2	Quotient M-type	19
6.3	Quotient inductive-inductive types (QIITs)	19
6.4	Partiality monad	20
6.4.1	Delay monad to Sequences	20
6.4.2	Sequence to Partiality Monad	22
6.4.3	Silhouette Trees	25
6.4.4	QM-types	29
6.5	Strongly Extensional (Coalgebra)	29
6.5.1	in progress	30
6.6	TODO	30
7	Properties of M-types?	31
7.1	Closure properties of M-types	31
7.1.1	Product of M-types	31
7.1.2	Co-product	33
7.1.3	33
8	Examples of M-types	35
8.1	The Partiality monad	35
8.2	TODO: Place these subsections	35
8.2.1	Identity Bisimulation	36
8.2.2	Bisimulation of Streams	36
8.2.3	Bisimulation of Delay Monad	36
8.2.4	Bisimulation of ITrees	37
8.2.5	Zip Function	37
8.2.6	Examples of Fixed Points	39
9	Conclusion	41
	Bibliography	43
A	Additions to the Cubical Agda Library	45

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: [7]

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 $B : A \rightarrow \mathcal{U}$
- A term $x : A$ or for constants $c : A$
- A function $f : A \rightarrow C$
- A constructor $f : A \rightarrow C$
- A destructor $f : A \rightarrow C$
- A path $p : A \equiv C$, heterogeneous paths are denoted \equiv_p or if the path is clear from context \equiv_* .
- A relation $R : A \rightarrow A \rightarrow \mathcal{U}$ with notation $x R y$.
- The unit type is $\mathbf{1}$ while the empty type is $\mathbf{0}$.
- A functor P
- A container is denoted as S or (A, B)
- A coalgebra $C\text{-}\gamma$
- We denote the function giving the first and second projection of a dependent pair by π_1 and π_2 .

better
descrip-
tion, not
always a
function

better
descrip-
tion, not
always a
function

Furthermore we define some useful notation

Definition 2.0.1.

$$\Downarrow x, f \Downarrow = \lambda n, \begin{cases} x & n = 0 \\ f\ m & n = m + 1 \end{cases} \quad (2.1)$$

Chapter 3

Background Theory

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

3.1 Proof assistants

3.2 Set Theory

3.3 Type Theory

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

3.3.1 Martin L f Type Theory / Intuitionistic type Theory (MLTT/ITT)

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

Intensional Type Theory

3.4 The Univalence Axiom

The map

$$\text{id}_{\mathcal{U}} A B \rightarrow \text{Equiv } A B \tag{3.1}$$

Should
this be
included

inductive
types,
uni-
verses
and
judge-
ments

extensional
vs inten-
sional

find
citation,
cur-
rently
<https://en.>

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

$$(A = B) \simeq (A \simeq B) \tag{3.2}$$

Or said in another way equivalent types are identical.

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

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

is a type theory where the identity types form path spaces, so proofs of identity are not just **refl**, as for MLTT. Types are seen as "spaces" or (higher groupoids), and we think of $a : A$ as a being a point in the space A , similarly functions are regarded as continuous maps from one space to another. [5]

3.5.1 Judgmental Equality / Definitional Equality

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

3.5.2 Propositional Equality

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

3.5.3 External (of theory)

3.5.4 Internal (of theory)

3.5.5 Constructivity

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

Proof relevant mathematics

3.5.6 Homotopy n -types

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

3.5.7 The HoTT Book

3.6 Higher order inductive types (HITs)

3.6.1 Propositional truncation and Set Truncated Quotients

3.7 Cubical Type Theory (CTT)

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

3.7.1 UIP

3.7.2 Cubical Agda

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

Axioms of cubical Agda

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

Something about the interval type!!

Path type

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

$$r, s ::= 0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s \quad (3.3)$$

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

3.8 Coinduction

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

cite:
<https://arxiv.org/abs/1808.07442>

3.9 W-types

3.10 Problems with / Getting around using AC and LEM

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

3.11 Universes ??

Chapter 4

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 $S = (A, B)$ for the types $A : \mathcal{U}$ and $B : A \rightarrow \mathcal{U}$.

From a container we can define a polynomial functor.

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

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

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

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

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

$$P_S(X) = \sum_{a:A} \mathbf{1} \rightarrow X. \tag{4.3}$$

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

$$P_S(X) = A \times X. \tag{4.4}$$

We now construct the P-coalgebra for a polynomial functor P .

Definition 4.1.3. A P-coalgebra is defined as

$$\text{Coalg}_S = \sum_{C:\mathcal{U}} C \rightarrow PC. \quad (4.5)$$

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

$$\begin{aligned} \cdot \Rightarrow \cdot &: \text{Coalg}_S \rightarrow \text{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta &= \sum_{f:C \rightarrow D} \delta \circ f = P f \circ \gamma \end{aligned} \quad (4.6)$$

We can now define M-types.

Definition 4.1.4. Given a container S , we define M-types, as the type M_S , making the coalgebra given by M_S and $\text{out} : M_S \rightarrow P_S(M_S)$ fulfill the property

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

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

Continuing our example we now construct an M-type for streams.

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

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

Figure 4.1: M-types of streams

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 $\text{stream } A = M_{(A,\lambda_,1)}$,

$$\frac{A:\mathcal{U} \quad s:\text{stream } A}{\text{hd } s:A} E_{\text{hd}} \quad (4.8)$$

$$\frac{A:\mathcal{U} \quad s:\text{stream } A}{\text{tl } s:\text{stream } A} E_{\text{tl}} \quad (4.9)$$

$$\frac{A:\mathcal{U} \quad x:A \quad xs:\text{stream } A}{\text{cons } x \text{ } xs:\text{stream } A} I_{\text{cons}} \quad (4.10)$$

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

Definition 4.1.5. We define a chain as a family of morphisms $\pi_{(n)} : X_{n+1} \rightarrow X_n$, over a family of types X_n . See Figure 4.1.5.

$$X_0 \xleftarrow{\pi_{(0)}} X_1 \xleftarrow{\pi_{(1)}} \cdots \xleftarrow{\pi_{(n-1)}} X_n \xleftarrow{\pi_{(n)}} X_{n+1} \xleftarrow{\pi_{(n+1)}} \cdots$$

Figure 4.2: Chain of TODO

$$\begin{array}{ccccccc} X_0 & \xleftarrow{\pi_{(0)}} & X_1 & \xleftarrow{\pi_{(1)}} & \cdots & \xleftarrow{\pi_{(n-1)}} & X_n \xleftarrow{\pi_{(n)}} X_{n+1} \xleftarrow{\pi_{(n+1)}} \cdots \\ & \searrow \mathbf{f}_0 & & \nwarrow \mathbf{f}_1 & & \nwarrow \mathbf{f}_n & \nearrow \mathbf{f}_{n+1} \nearrow \cdots \\ & & & & & C & \end{array}$$

Figure 4.3: Cone

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

Proof. We define an isomorphism from $C \rightarrow M_S$ to $\text{Cone}_{C\text{-}\gamma}$

$$\mathbf{fun}_{\text{collapse}} \mathbf{f} = (\lambda n z, \pi_1 (\mathbf{f} z) n), (\lambda n i a, \pi_2 (\mathbf{f} a) n i) \quad (4.11)$$

$$\mathbf{inv}_{\text{collapse}} (\mathbf{u}, \mathbf{q}) z = (\lambda n, \mathbf{u} n z), (\lambda n i, \mathbf{q} n i z) \quad (4.12)$$

$$\mathbf{rinv}_{\text{collapse}} (\mathbf{u}, \mathbf{q}) = \mathbf{refl}_{(\mathbf{u}, \mathbf{q})} \quad (4.13)$$

$$\mathbf{linv}_{\text{collapse}} \mathbf{f} = \mathbf{refl}_{\mathbf{f}} \quad (4.14)$$

□

Lemma 4.1.2. *Given $\ell : \prod_{(n:\mathbb{N})} (X_n \rightarrow X_{n+1})$ and $y : \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv \ell_n x_n$ the chain collapses as the equality $\mathcal{L} \equiv X_0$.*

Proof. We define this collapse by the isomorphism

$$\mathbf{fun}_{\mathcal{L}\text{collapse}} (x, r) = x_0 \quad (4.15)$$

$$\mathbf{inv}_{\mathcal{L}\text{collapse}} x_0 = (\lambda n, \ell^n x_0), (\lambda n, \mathbf{refl}_{(\ell^{(n+1)} x_0)}) \quad (4.16)$$

$$\mathbf{rinv} x_0 = \mathbf{refl}_{x_0} \quad (4.17)$$

where $\ell^n = \ell_n \circ \ell_{n-1} \circ \cdots \circ \ell_1 \circ \ell_0$. To define $\mathbf{linv} (x, r)$, 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 $\mathbf{GY} = \mathbf{1} + Y$, we get that sections are contractible, meaning $y \equiv \mathbf{inv}_{\mathcal{L}\text{collapse}} (\mathbf{fun}_{\mathcal{L}\text{collapse}} y)$, since both are equal to sections over \mathbb{N} . □

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

Theorem 4.1.3. *Given the container (A, B) we define the equality*

$$\mathbf{shift} : \mathcal{L} \equiv \mathbf{PL} \quad (4.18)$$

where \mathbf{PL} is the limit of a shifted sequence. Then

$$\mathbf{in} = \mathbf{transport} \mathbf{shift} \quad (4.19)$$

$$\mathbf{out} = \mathbf{transport} (\mathbf{shift}^{-1}). \quad (4.20)$$

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^P \equiv P\mathcal{L} \quad (4.21)$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^P \quad (4.22)$$

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

$$\mathbf{fun}_{\mathcal{L}unique}(\mathbf{a}, \mathbf{b}) = \mathbf{!} \star, \mathbf{a} \mathbf{!}, \mathbf{!} \mathbf{refl}_\star, \mathbf{b} \mathbf{!} \quad (4.23)$$

$$\mathbf{inv}_{\mathcal{L}unique}(\mathbf{a}, \mathbf{b}) = \mathbf{a} \circ \mathbf{succ}, \mathbf{b} \circ \mathbf{succ} \quad (4.24)$$

$$\mathbf{rinv}_{\mathcal{L}unique}(\mathbf{a}, \mathbf{b}) = \mathbf{refl}_{(\mathbf{a}, \mathbf{b})} \quad (4.25)$$

$$\mathbf{linv}_{\mathcal{L}unique}(\mathbf{a}, \mathbf{b}) = \mathbf{refl}_{(\mathbf{a}, \mathbf{b})} \quad (4.26)$$

The definition of α is then,

$$\mathcal{L}^P \equiv \sum_{(x : \prod_{(n:\mathbb{N})} \sum_{(a:A)} B a \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi_{(n+1)} x_{n+1} \equiv x_n \quad (4.27)$$

$$\equiv \sum_{(x : \sum_{(a : \prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(u : \prod_{(n:\mathbb{N})} B (\pi_1 x)_n \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathbf{u}_{n+1} \equiv_* \mathbf{u}_n \quad (4.28)$$

$$\equiv \sum_{(a:A)} \sum_{(u : \prod_{(n:\mathbb{N})} B a \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathbf{u}_{n+1} \equiv \mathbf{u}_n \quad (4.29)$$

$$\equiv \sum_{a:A} B a \rightarrow \mathcal{L} \quad (4.30)$$

$$\equiv P\mathcal{L} \quad (4.31)$$

To collapse $\sum_{(a : \prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n$ to A between (4.28) and (4.29) we use Lemma 4.1.2 . We use Lemma 4.1.1 for the equality between (4.29) and (4.30). The rest of the equalities are given by a simple isomorphism or by definition. The definition of $shift$ is

$$shift = \alpha^{-1} \cdot \mathcal{L}unique. \quad (4.32)$$

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

4.2 Coinduction Principle for \mathbf{M} -types

We can now construct a coinduction principle given a bisimulation relation

Definition 4.2.1. For all coalgebras $C\text{-}\gamma : \mathbf{Coalg}_S$, 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 $\overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} : \mathbf{Coalg}_S$, making the diagram in Figure 4.4 commute (\Rightarrow represents P-coalgebra morphisms).

Definition 4.2.2 (Coinduction principle). Given a relation \mathcal{R} , that is part of a bisimulation over a final P-coalgebra $\mathbf{M}\text{-}\mathbf{out} : \mathbf{Coalg}_S$ we get the diagram in Figure 4.5, where $\pi_1^{\overline{\mathcal{R}}} = ! = \pi_2^{\overline{\mathcal{R}}}$, which means given $r : m \mathcal{R} m'$ we get the equation

$$m = \pi_1^{\overline{\mathcal{R}}}(m, m', r) = \pi_2^{\overline{\mathcal{R}}}(m, m', r) = m'. \quad (4.33)$$

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

Figure 4.4: Bisimulation for a coalgebra

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

Figure 4.5: 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

$$\text{stream } A := M_{(A, \lambda _, 1)} \quad (5.1)$$

this is equal to an alternative definition of streams

5.2 ITrees as M-types

We want the following rules for ITrees

$$\frac{r : R}{\text{Ret } r : \text{itree } E \ R} I_{\text{Ret}} \quad (5.2)$$

$$\frac{A : \mathcal{U} \quad a : E \ A \quad f : A \rightarrow \text{itree } E \ R}{\text{Vis } a \ f : \text{itree } E \ R} I_{\text{Vis}}. \quad (5.3)$$

Elimination rules

$$\frac{t : \text{itree } E \ R}{\text{Tau } t : \text{itree } E \ R} E_{\text{Tau}}. \quad (5.4)$$

5.2.1 Delay Monad

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

check this statement

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

$$P_S(X) = \sum_{x:1+R} \lambda\{\text{inl } _ \rightarrow 1 ; \text{inr } _ \rightarrow 0\} x \rightarrow X, \quad (5.5)$$

which is equal to

$$P_S(X) = X + R \times (0 \rightarrow X). \quad (5.6)$$

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

$$P_S(X) = X + R \quad (5.7)$$

meaning we get the following diagram.

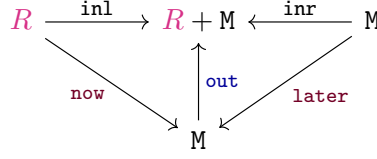


Figure 5.1: Delay monad

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

(Later = Tau, Ret = Now)

5.2.2 Tree

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

$$S = \left(R + \sum_{A:\mathcal{U}} E A, \lambda\{\mathbf{inl} _ \rightarrow \mathbf{0} ; \mathbf{inr} (A, e) \rightarrow A\} \right). \quad (5.8)$$

This will give us the polynomial functor

$$P_S(X) = \sum_{x:R + \sum_{A:\mathcal{U}} E A} \lambda\{\mathbf{inl} _ \rightarrow \mathbf{0} ; \mathbf{inr} (A, e) \rightarrow A\} x \rightarrow X, \quad (5.9)$$

which simplifies to

$$P_S(X) = (R \times (\mathbf{0} \rightarrow X)) + \left(\sum_{A:\mathcal{U}} E A \times (A \rightarrow X) \right), \quad (5.10)$$

and further

$$P_S(X) = R + \sum_{A:\mathcal{U}} E A \times (A \rightarrow X). \quad (5.11)$$

We get the following diagram for the P-coalgebra.

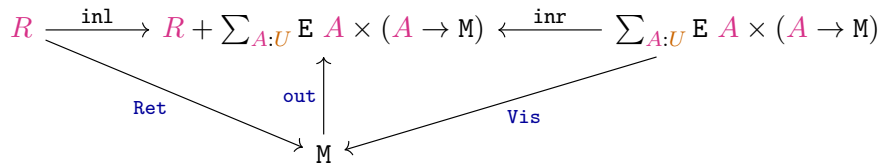


Figure 5.2: TODO

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

$$S = \left(\mathbf{1} + R + \sum_{A:\mathcal{U}} (\mathbf{E} \ A) \ , \ \lambda \{ \text{inl} \ (\text{inl} \ _) \rightarrow \mathbf{1} ; \text{inl} \ (\text{inr} \ _) \rightarrow \mathbf{0} ; \text{inr} (A, _) \rightarrow A \} \right). \quad (5.12)$$

Such that the (reduced) polynomial functor becomes

$$\mathbf{P}_S(X) = X + R + \sum_{A:\mathcal{U}} ((\mathbf{E} \ A) \times (A \rightarrow X)) \quad (5.13)$$

Giving us the diagram

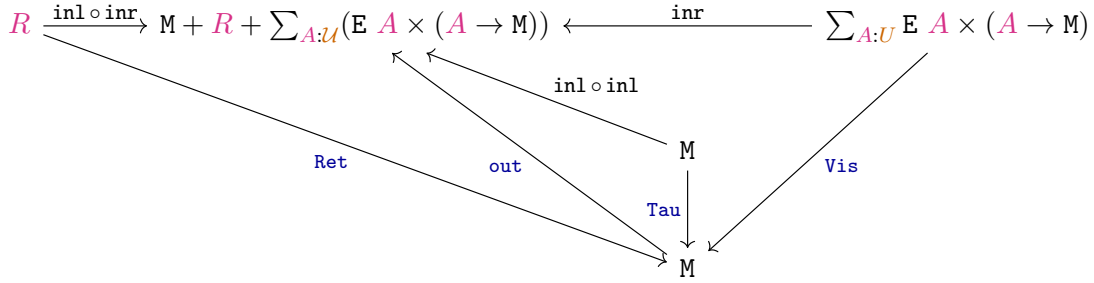


Figure 5.3: TODO

5.3 Automaton

An automaton is defined as a set of state V and an alphabet α and a transition function $\delta : V \rightarrow \alpha \rightarrow V$. This gives us the diagram in Figure 5.4

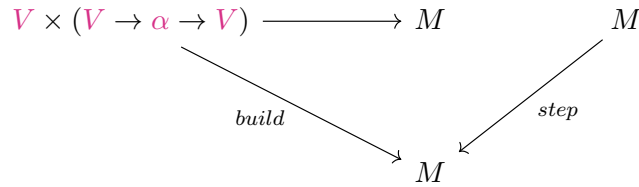


Figure 5.4: automaton

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 bisimilarity.

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

$$\frac{}{R_\perp : \mathcal{U}} \quad (6.1) \quad \frac{}{\perp : R_\perp} \quad (6.2) \quad \frac{a : R}{\eta a : R_\perp} \quad (6.3)$$

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

$$\frac{s : \mathbb{N} \rightarrow R_\perp \quad b : \prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp s_{n+1}}{\bigsqcup (s, b) : R_\perp} \quad (6.4) \quad \frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp p q : x \equiv y} \quad (6.5)$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x} \sqsubseteq_{\text{refl}} \quad (6.6) \quad \frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z} \sqsubseteq_{\text{trans}} \quad (6.7) \quad \frac{x : R_\perp}{\perp \sqsubseteq_\perp x} \sqsubseteq_{\text{never}} \quad (6.8)$$

$$\frac{s : \mathbb{N} \rightarrow R_\perp \quad b : \prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp s_{n+1}}{\prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp \bigsqcup (s, b)} \quad (6.9) \quad \frac{\prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp x}{\bigsqcup (s, b) \sqsubseteq_\perp x} \quad (6.10)$$

and finally set truncated

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

6.4.1 Delay monad to Sequences

Introduce the delay monad before this section!!

Definition 6.4.2. We define

$$\text{Seq}_R = \sum_{(s:\mathbb{N} \rightarrow R+1)} \text{isMon } s \quad (6.12)$$

where

$$\text{isMon } s = \prod_{(n:\mathbb{N})} (s_n \equiv s_{n+1}) + ((s_n \equiv \text{inr } \star) \times (s_{n+1} \not\equiv \text{inr } \star)) \quad (6.13)$$

meaning a sequences is $\text{inr } \star$ until it reaches a point where it switches to $\text{inl } r$ for some value r . There are also the special cases of already terminated, meaning only $\text{inl } r$ and never terminating meaning only $\text{inr } \star$.

For each index in a sequence, the element at that index s_n is either not terminated $s_n \equiv \text{inr } \star$, which we denote as $s_n \uparrow_{R+1}$, or it is terminated $s_n \equiv \text{inl } r$ with some value r , denoted by $s_n \downarrow_{R+1} r$ or just $s_n \downarrow_{R+1}$ to mean $s_n \not\equiv \text{inr } \star$. Thus we can write isMon as

$$\text{isMon } s = \prod_{(n:\mathbb{N})} (s_n \equiv s_{n+1}) + ((s_n \uparrow_{R+1}) \times (s_{n+1} \downarrow_{R+1})) \quad (6.14)$$

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

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

$$\text{never}_{\text{Seq}} = (\lambda _, \text{inr } \star), (\lambda _, \text{inl refl}) \quad (6.16)$$

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

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

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

$$\text{shift } (s, q) (z_s, z_q) = \begin{cases} z_s & n = 0 \\ s_m & n = m + 1 \end{cases}, \begin{cases} z_q & n = 0 \\ q_m & n = m + 1 \end{cases}, \quad (6.17)$$

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

$$\text{unshift } (s, q) = s \circ \text{succ}, q \circ \text{succ}. \quad (6.18)$$

Lemma 6.4.1. *The function*

$$\text{shift-unshift } (s, q) = \text{shift } (\text{unshift } (s, q)) (s_0, q_0) \quad (6.19)$$

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. \square

Lemma 6.4.2. *The function*

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

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. \square

We now define an equivalence between $\text{delay } R$ and Seq_R , where later are equivalent to shifts, and $\text{now } r$ is equivalent terminated sequence with value r . We do this by defining equivalence functions, and the left and right identities.

Lemma 6.4.3 ($\text{inl} \neq \text{inr}$). *For any two elements $x = \text{inl } a$ and $y = \text{inr } b$ then $x \neq y$.*

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

Definition 6.4.5. We define function from $\text{Delay } R$ to Seq_R

$$\begin{aligned} \text{Delay} \rightarrow \text{Seq} (\text{now } r) &= \text{now}_{\text{Seq}} r \\ \text{Delay} \rightarrow \text{Seq} (\text{later } x) &= \\ \text{shift } (\text{Delay} \rightarrow \text{Seq } x) &\left(\text{inr } \star, \begin{cases} \text{inr } (\text{refl}, \text{inl} \neq \text{inr}) & x = \text{now } _ \\ \text{inl } \text{refl} & x = \text{later } _ \end{cases} \right) \end{aligned} \quad (6.21)$$

Definition 6.4.6. We define function from Seq_R to $\text{Delay } R$

$$\text{Seq} \rightarrow \text{Delay} (s, q) = \begin{cases} \text{now } r & s_0 = \text{inl } r \\ \text{later } (\text{Seq} \rightarrow \text{Delay } (\text{unshift } (s, q))) & s_0 = \text{inr } \star \end{cases} \quad (6.22)$$

Theorem 6.4.4. *The type Seq_R is equal to $\text{Delay } R$*

better
for-
mualted
proof

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

$$\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (s, q)) \equiv (s, q) \quad (6.23)$$

defined by cases analysis on s_0 , if $s_0 = \text{inl } r$ then we need to show

$$\text{now}_{\text{Seq}} r \equiv (s, q) \quad (6.24)$$

This is true, since (s, q) is a monotone sequence and $\text{inl } r$ is the top element of the order, then all elements of the sequence are $\text{inl } r$. If $s_0 = \text{inr } \star$ then, we need to show

$$\text{shift} (\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (\text{unshift} (s, q)))) (\text{inr } \star, _) \equiv (s, q) \quad (6.25)$$

by the induction hypothesis we get

$$\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (\text{unshift} (s, q))) \equiv \text{unshift} (s, q) \quad (6.26)$$

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

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

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

$$\text{Seq} \rightarrow \text{Delay} (\text{Delay} \rightarrow \text{Seq} t) \equiv t \quad (6.27)$$

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

$$\text{later} (\text{Seq} \rightarrow \text{Delay} (\text{unshift} (\text{shift} (\text{Delay} \rightarrow \text{Seq} x)))) \equiv \text{later } x \quad (6.28)$$

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

Corollary. *The types Delay / \sim and Seq / \sim are equal.*

Show this

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

Show this

and we show if $x \sim_{\text{Seq}} y$ then $\text{Seq} \rightarrow \text{Delay} x \sim_{\text{Seq}} \text{Seq} \rightarrow \text{Delay} y$, \square

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 $(s, q) : \text{Seq}_R$ terminates with a given value $r : R$,

$$(s, q) \downarrow_{\text{Seq}} r = \sum_{(n:\mathbb{N})} s_n \downarrow_{R+1} r. \quad (6.29)$$

Definition 6.4.8 (Sequence Ordering).

$$(s, q) \sqsubseteq_{\text{Seq}} (t, p) = \prod_{(a:R)} (\|s \downarrow_{\text{Seq}} a\| \rightarrow \|t \downarrow_{\text{Seq}} a\|) \quad (6.30)$$

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

propositional
or set
trunca-
tion?

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

$$\begin{aligned}\text{Maybe} \rightarrow (-)_\perp (\text{inl } r) &= \eta \ r \\ \text{Maybe} \rightarrow (-)_\perp (\text{inr } \star) &= \perp\end{aligned}\tag{6.31}$$

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

$$x \sqsubseteq_{R+\mathbf{1}} y = (x \equiv y) + ((x \downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}}))\tag{6.32}$$

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

$$\text{isMon } s = \prod_{(n:\mathbb{N})} s_n \sqsubseteq_{R+\mathbf{1}} s_{n+1}\tag{6.33}$$

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

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

Proof. We do the proof by case.

$$\begin{aligned}\text{Maybe} \rightarrow (-)_\perp\text{-mono } (\text{inl } p) &= \\ \text{subst } (\lambda a, \text{Maybe} \rightarrow (-)_\perp x \sqsubseteq_\perp \text{Maybe} \rightarrow (-)_\perp a) \ p \ (\sqsubseteq_{\text{refl}} (\text{Maybe} \rightarrow (-)_\perp x)) \\ \text{Maybe} \rightarrow (-)_\perp\text{-mono } (\text{inr } (p, _)) &= \\ \text{subst } (\lambda a, \text{Maybe} \rightarrow (-)_\perp a \sqsubseteq_\perp \text{Maybe} \rightarrow (-)_\perp y) \ p^{-1} \ (\sqsubseteq_{\text{never}} (\text{Maybe} \rightarrow (-)_\perp y))\end{aligned}\tag{6.34}$$

□

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

$$\begin{aligned}\text{Seq} \rightarrow \text{incSeq} \\ \text{Seq} \rightarrow \text{incSeq } (g, q) &= \text{Maybe} \rightarrow (-)_\perp \circ g, \text{Maybe} \rightarrow (-)_\perp\text{-mono} \circ q\end{aligned}\tag{6.35}$$

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

$$\begin{aligned}\text{Seq} \rightarrow (-)_\perp : \text{Seq}_A \rightarrow A_\perp \\ \text{Seq} \rightarrow (-)_\perp (g, q) &= \bigsqcup \circ \text{Seq} \rightarrow \text{incSeq}\end{aligned}\tag{6.36}$$

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

$$\text{Seq} \rightarrow (-)_\perp\text{-mono} : \text{isSet } A \rightarrow (x \ y : \text{Seq}_A) \rightarrow x \sqsubseteq_{\text{seq}} y \rightarrow \text{Seq} \rightarrow (-)_\perp x \sqsubseteq_\perp \text{Seq} \rightarrow (-)_\perp y\tag{6.37}$$

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

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

$$\text{Seq} \rightarrow (-)_\perp \approx \Rightarrow \equiv A_{\text{set}} \ x \ y \ (p, q) = \alpha_\perp (\text{Seq} \rightarrow (-)_\perp\text{-mono } A_{\text{set}} \ x \ y \ p) (\text{Seq} \rightarrow (-)_\perp\text{-mono } A_{\text{set}} \ y \ x \ q)\tag{6.38}$$

there exists non-monotone sequences, it just follows our definition of a sequence.

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

should this be formalized entirely, or should there just be a comment about monotonicity?

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

$$\begin{aligned} \text{rec } [z] &= f z \\ \text{rec } (\text{eq}/ _ _ r i) &= g r i \\ \text{rec } (\text{squash}/ a b p q i j) &= B_{\text{set}} (\text{rec } a) (\text{rec } b) (\text{ap rec } p) (\text{ap rec } q) i j \end{aligned} \quad (6.39)$$

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

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

$$\text{Seq}/\sim \rightarrow (-)_\perp = \text{rec Seq} \rightarrow (-)_\perp (\text{Seq} \rightarrow (-)_\perp \sim \Rightarrow \equiv A_{\text{set}}) (-)_\perp \text{-isSet} \quad (6.40)$$

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

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

$$s \sim_{\text{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\text{seq}} r\| \rightarrow \|y \downarrow_{\text{seq}} r\| \right) \times \left(\prod_{(r:R)} \|y \downarrow_{\text{seq}} r\| \rightarrow \|x \downarrow_{\text{seq}} r\| \right) \quad (6.41)$$

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

Lemma 6.4.8. The function $\text{Seq}/\sim \rightarrow (-)_\perp$ is injective.

Proof. We use propositional elimination of quotients

$$\begin{aligned} \text{elimProp} : (B : \text{Seq}_R/\sim_{\text{seq}} \rightarrow \mathcal{U}) &\rightarrow ((x : \text{Seq}_R/\sim_{\text{seq}}) \rightarrow \text{isProp } (B x)) \\ &\rightarrow (f : (a : \text{Seq}_R) \rightarrow B [a]) \rightarrow (x : \text{Seq}_R/\sim_{\text{seq}}) \rightarrow B x \end{aligned} \quad (6.42)$$

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

$$\begin{aligned} \text{elimProp} & \\ &(\lambda a, \text{Seq}/\sim \rightarrow (-)_\perp a \equiv \text{Seq}/\sim \rightarrow (-)_\perp y \rightarrow a \equiv y) \\ &(\lambda a, \text{isProp}\Pi (\lambda _, \text{squash}/ a y)) \\ &(\lambda a, \text{elimProp} \\ &\quad (\lambda b, \text{Seq} \rightarrow (-)_\perp a \equiv \text{Seq}/\sim \rightarrow (-)_\perp b \rightarrow [a] \equiv b) \\ &\quad (\lambda b, \text{isProp}\Pi (\lambda _, \text{squash}/ [a] b)) \\ &\quad (\lambda b, (\text{eq}/ a b) \circ (\text{Seq} \rightarrow (-)_\perp \text{-isInjective } a b))) \end{aligned} \quad (6.43)$$

where $\text{Seq} \rightarrow (-)_\perp \text{-isInjective}$ is (6.4.7), \square

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

is this a
recursor,
and for
what?
The quo-
tient?

Should
this be
formal-
ized?

Convert
to text,
instead
of a
proof
term!?

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

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

describe countable choice (and why it is needed!)

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

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

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

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

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 \mathbf{M} -type given by the following container

$$\left(R + \mathbf{1}, \begin{cases} \perp & \text{inl } a \\ E & \text{inr } \star \end{cases} \right) \quad (6.44)$$

We get the constructors

$$\frac{a : R}{\text{leaf } a : \text{tree } R E} \quad (6.45)$$

$$\frac{k : E \rightarrow \text{tree } R E}{\text{node } k : \text{tree } R E} \quad (6.46)$$

Then we define the weak bisimilarity relation \sim_{tree}

$$\frac{}{\text{leaf } x \sim_{\text{tree}} \text{leaf } y} \sim^{\text{leaf}} \quad (6.47)$$

$$\frac{\prod_{(v:E)} k_1 v \sim_{\text{tree}} k_2 v}{\text{node } k_1 \sim_{\text{tree}} \text{node } k_2} \sim^{\text{node}} \quad (6.48)$$

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

$$\text{leaf}_{\text{sTree}} : \text{sTree } E \quad (6.49)$$

$$\frac{k : E \rightarrow \text{sTree } E}{\text{node}_{\text{sTree}} k : \text{sTree } E} \quad (6.50)$$

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

more precise description!

Complete the rest of the proof!

add all needed constructors

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

$$\frac{x \sqsubseteq_{\text{sTree}} y \quad y \sqsubseteq_{\text{sTree}} x}{\alpha_{\text{sTree}} x y : \text{sTree } E} \quad (6.51)$$

$$\frac{s : (\mathbb{N} \rightarrow E) \rightarrow \text{sTree } E}{\bigsqcup_{(e:\mathbb{N} \rightarrow E)} (s \ e)} \quad (6.52)$$

add all
needed
con-
struc-
tors

From tree to Seq_{tree}

We now want to show the equivalence between these two constructions, to do this we define an intermediate construction 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 branches are monotone increasing .

$$\text{Seq}_{tree} = \prod_{(e:\mathbb{N} \rightarrow E)} \sum_{(s:\mathbb{N} \rightarrow R+1)} \prod_{(n:\mathbb{N})} s_n \sqsubseteq_{R+1} s_{n+1} \quad (6.53)$$

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

Definition 6.4.17. We define a function to shift a Seq_{tree} , it takes $f : E \rightarrow \text{Seq}_{tree}$ as an argument. We let $s' = f \ e_0 \ (e \circ \text{succ})$, then the definition is given as

$$\text{shift-seq } f = \lambda e, \lambda \text{ inr } \star, \pi_1 s' \downarrow, \left(\lambda n, \begin{cases} \text{inr } (\text{refl}, \text{inl} \neq \text{inr}) & n = 0 \wedge \pi_1 s' 0 = \text{inl } r \\ \text{inl refl} & n = 0 \wedge \pi_1 s' 0 = \text{inr } \star \\ \pi_2 s' m & n = m + 1 \end{cases} \right) \quad (6.54)$$

Definition 6.4.18. We define a function to unshift a Seq_{tree}

$$\text{unshift-seq } s \ v = \lambda e, (\pi_1 (s (\downarrow v), e \downarrow)) \circ \text{succ}, (\pi_2 (s (\downarrow v), e \downarrow)) \circ \text{succ}) \quad (6.55)$$

Lemma 6.4.13. Shift and unshift are inverse to each other

Proof. The same reasoning as for \square

Definition 6.4.19. We get a function from trees to monotone sequences

$$\begin{aligned} \text{tree} \rightarrow \text{Seq} (\text{leaf } r) &= \lambda _, (\lambda _, \text{inl } r), (\lambda _, \text{inl refl}) \\ \text{tree} \rightarrow \text{Seq} (\text{node } k) &= \text{shift } (\text{tree} \rightarrow \text{Seq } \circ k) \end{aligned} \quad (6.56)$$

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

$$\text{Seq} \rightarrow \text{tree } s = \begin{cases} \text{leaf } r & \prod_{(e:\mathbb{N} \rightarrow E)} \pi_1 (s \ e) 0 = \text{inl } r \\ \text{node } (\text{Seq} \rightarrow \text{tree } \circ \text{unshift } s) & \text{o.w.} \end{cases} \quad (6.57)$$

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.

$$\left(\prod_{(e:\mathbb{N} \rightarrow E)} \pi_1 (s \ e) 0 = \text{inl } r \right) \Leftrightarrow (s \equiv \lambda _, (\lambda _, \text{inl } r), (\lambda _, \text{inl refl})) \quad (6.58)$$

ordering
is con-
tainer
order-
ing not
maybe?

specify
branches
increas-
ing?

how
does
it differ?
Con-
struc-
tors are
equal
type is
different
(trees in-
stead of
delay)

shift -
unshift

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

Theorem 6.4.15. *The types tree and Seq_{tree} are equal*

Proof. We construct an isomorphism by the functions $\text{tree} \rightarrow \text{Seq}$ and $\text{Seq} \rightarrow \text{tree}$, with right inverse given by two cases, one where the first element in the sequence is $\text{inl } r$, meaning representing a leaf with value r , then we need to show that $s \equiv \lambda _, (\lambda _, \text{inl } r), (\lambda _, \text{inl refl})$ which follows from Lemma 6.4.14. Otherwise we need to show that

$$\text{shift } (\text{tree} \rightarrow \text{Seq} \circ \text{Seq} \rightarrow \text{tree} \circ \text{unshift } s) \equiv s \quad (6.59)$$

By induction we get

$$\text{tree} \rightarrow \text{Seq} \circ \text{Seq} \rightarrow \text{tree} \circ \text{unshift } s \equiv \text{unshift } s \quad (6.60)$$

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

$$\begin{aligned} \text{tree-Seq } (\text{leaf } r) &= \text{refl} \\ \text{tree-Seq } (\text{node } k) &= \text{unshift-shift } (\text{tree} \rightarrow \text{Seq} \circ k) \cdot \text{tree-Seq } k \end{aligned} \quad (6.61)$$

\square

We start by defining some ordering relation on Seq_{tree}

Definition 6.4.21 (Sequence Termination). The following relations says that a branche $e : \mathbb{N} \rightarrow E$ of a sequence $s : \text{Seq}_{\text{tree}}$ terminates with some value $r : R$,

$$(s \ e) \downarrow_{\text{Seq}_{\text{tree}}} r = \sum_{(n:\mathbb{N})} (s \ e \ n) \downarrow_{R+1} r. \quad (6.62)$$

Definition 6.4.22 (Sequence Ordering).

$$s \sqsubseteq_{\text{Seq}_{\text{tree}}} t = \prod_{(e:E)} \prod_{(a:R)} (\|(s \ e) \downarrow_{\text{Seq}} a\| \rightarrow \|(t \ e) \downarrow_{\text{Seq}} a\|) \quad (6.63)$$

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

Definition 6.4.23. We define weak bisimilarity relation for sequences

$$s \sim_{\text{Seq}_{\text{tree}}} t = s \sqsubseteq_{\text{Seq}_{\text{tree}}} t \times t \sqsubseteq_{\text{Seq}_{\text{tree}}} s \quad (6.64)$$

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

Proof. We follow the same strategy as for Delay/\sim and Seq/\sim \square

propositional
or set
trunca-
tion?

Is this
defined
for par-
tiality
monad?

Seq_{tree} to sTree

Introduction
to Seq
to tree

Definition 6.4.24. We define a function converting a sequence on trees to a monotone sequence on sTree's

$$\text{Seq} \rightarrow \text{incSeq} \quad (6.65)$$

Definition 6.4.25. There is a function from Seq_{tree} to sTree

$$\text{Seq} \rightarrow \text{sTree } s = \bigsqcup_{(e: \mathbb{N} \rightarrow E)} (\text{Seq} \rightarrow \text{incSeq } s \ e) \quad (6.66)$$

Lemma 6.4.16. Given $a \sqsubseteq_{\text{Seq}_{\text{tree}}} b$, then $\text{Seq} \rightarrow \text{sTree } a \sqsubseteq_{\text{sTree}} \text{Seq} \rightarrow \text{sTree } b$.

Complete
proof

Proof. .

□

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

describe
better

Lemma 6.4.17. *If two elements are weakly bisimilar, then they are equal as sTrees*

$$\begin{aligned} \text{Seq} \rightarrow \text{sTree} \sim \rightarrow \equiv x \ y \ (p, q) = \\ \alpha_{\text{sTree}} (\text{Seq} \rightarrow \text{sTree} \text{-mono } x \ y \ p) (\text{Seq} \rightarrow \text{sTree} \text{-mono } y \ x \ q) \end{aligned} \quad (6.67)$$

We then get the definition

TODO

Definition 6.4.26. TODO!

$$\text{Seq} \rightarrow \text{sTree } s = \bigsqcup_{(e: \mathbb{N} \rightarrow E)} (s \ e) \quad (6.68)$$

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

Lemma 6.4.18. *The function tree \rightarrow sTree is monotone, meaning if $x \sqsubseteq_{\text{tree}} y$, then we have tree \rightarrow sTree $x \sqsubseteq_{\text{sTree}} \text{tree} \rightarrow \text{sTree } y$.*

Proof. By case

$$\text{tree} \rightarrow \text{sTree} \text{-mono } (\text{leaf}) \ _ = \quad (6.69)$$

□

We then want to lift this definition to the quotient using the recursor for quotients (6.39),

Definition 6.4.27. There is a function from tree $A \ E / \sim_{\text{tree}}$ to sTree E

$$\text{tree} / \sim \rightarrow \text{sTree} = \text{rec tree} \rightarrow \text{sTree treesTree} \ \sim \rightarrow \equiv \text{sTree-isSet} \quad (6.70)$$

6.4.4 QM-types

We want to define sequences based on \mathbf{M} -types

$$\text{Seq}_{\mathbf{M}_{(\mathcal{A}, \mathcal{B})}} = \prod_{(a:\mathbb{N} \rightarrow \mathcal{B} \text{ ?})} \sum_{a:\mathbb{N} \rightarrow \mathcal{A}} \prod_{(n:\mathbb{N})} \mathbf{s}_n \sqsubseteq? \mathbf{s}_{n+1} \quad (6.71)$$

which
other
QM
types
can
be ex-
pressed
as QIITs

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

$$\dots \quad (6.72)$$

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 $(\mathcal{A}, \lambda _, \mathbf{0})$.

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

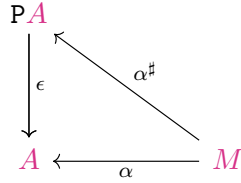


Figure 6.1: Cofree Coalgebra

A coalgebra PA is cofree on A iff for all coalgebras M and mappings $\alpha : UM \rightarrow C$ there is a unique morphism $\alpha^\# : M \rightarrow TC$ 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

$$EqSys : \sum_{(E:\mathcal{U})} \sum_{(V:E \rightarrow \mathcal{U})} ((e : E) \rightarrow T(Ve)) \times ((e : E) \rightarrow T(Ve)) \quad (6.73)$$

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 \mathbf{G} be functors and $v : \mathbf{P} \rightarrow \mathbf{G}$ a natural transformation. Suppose that for any type V , the functor $(\lambda _ \rightarrow V) \times \mathbf{F}$ has a final coalgebra. Then there exists for any \mathbf{G} -coalgebra $C-\gamma$ an P -coalgebra $S_C-\alpha$ and a \mathbf{G} -homomorphism $\varepsilon : S_C-v_{S_C} \circ \alpha \Rightarrow C-\gamma$, satisfying the universal property: for any P -coalg $U-\alpha_U$ and any \mathbf{G} -homomorphism $f : U-v_U \circ \alpha_U \Rightarrow C-\gamma$ there exists a unique P -homomorphism $\tilde{f} : U-\alpha_u \Rightarrow S_C-\alpha$ such that $\varepsilon \circ \tilde{f} = f$. The P -coalg $S_C-\alpha$ (and ε) is called cofree on the \mathbf{G} -coalgebra $C-\gamma$. [6, 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 an equation system at the same time, as we define the \mathbf{M} -type type. The equation systems is defined on a type $E : \mathcal{U}$ with variables of type $V : E \rightarrow \mathcal{U}$, each equation is given by functions $l, r : C \rightarrow A$ for some type C . A coalgebra satisfies the equation system iff $(t : B(lc) \rightarrow MQ) \rightarrow (s : B(rc) \rightarrow MQ) \rightarrow lc \equiv rc$ is inhabited.

6.6 TODO

- Resumption Monad transformer
- coinduction in Coq is broken
- bisim \Rightarrow eq
- copattern matching
- cubical Agda. Relation between \mathbf{M} -types defined by coinduction/copattern matching and constructed from \mathbf{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) \mathbf{M} -types. Like reduction of strictly positive inductive types to \mathbf{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 7

Properties of M-types?

7.1 Closure properties of M-types

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

7.1.1 Product of M-types

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

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

$$(A, B) \times (C, D) \equiv (A \times C, \lambda(a, c), B \ a \times D \ c). \quad (7.1)$$

We can lift this rule, through the diagram in Figure 7.1, used to define M-types.

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

$$P_{(A, B)}^n \mathbf{1} \times P_{(C, D)}^n \mathbf{1} \equiv P_{(A, B) \times (C, D)}^n \mathbf{1}. \quad (7.2)$$

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

$$P_{(A, B)}^m \mathbf{1} \times P_{(C, D)}^m \mathbf{1} \equiv P_{(A, B) \times (C, D)}^m \mathbf{1}, \quad (7.3)$$

in the following

$$P_{(A, B)}^{m+1} \mathbf{1} \times P_{(C, D)}^{m+1} \mathbf{1} \quad (7.4)$$

$$\equiv P_{(A, B)}(P_{(A, B)}^m \mathbf{1}) \times P_{(C, D)}(P_{(C, D)}^m \mathbf{1}) \quad (7.5)$$

$$\equiv \sum_{a:A} B \ a \rightarrow P_{(A, B)}^m \mathbf{1} \times \sum_{c:C} D \ c \rightarrow P_{(C, D)}^m \mathbf{1} \quad (7.6)$$

$$\equiv \sum_{a,c:A \times C} (B \ a \rightarrow P_{(A, B)}^m \mathbf{1}) \times (D \ c \rightarrow P_{(C, D)}^m \mathbf{1}) \quad (7.7)$$

$$\equiv \sum_{a,c:A \times C} B \ a \times D \ c \rightarrow P_{(A, B)}^m \mathbf{1} \times P_{(C, D)}^m \mathbf{1} \quad (7.8)$$

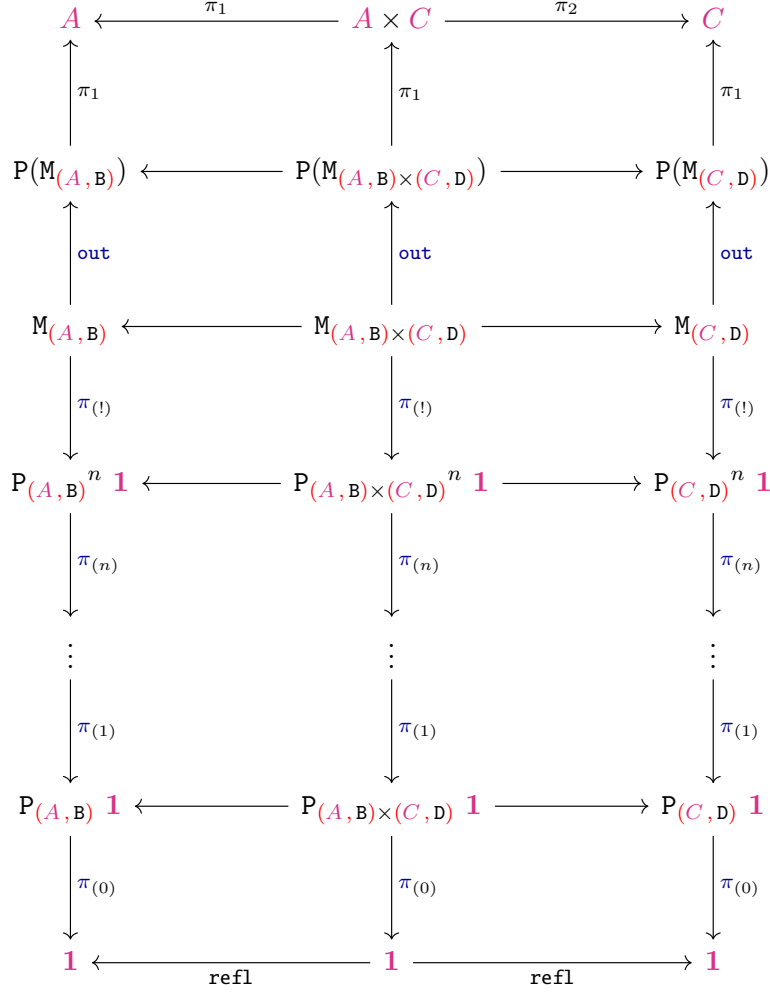


Figure 7.1: Diagram for products of chains

$$\equiv \sum_{a,c:A \times C} B \ a \times D \ c \rightarrow P_{(A,B) \times (C,D)}^m \mathbf{1} \quad (7.9)$$

$$\equiv P_{(A,B) \times (C,D)}(P_{(A,B) \times (C,D)}^m \mathbf{1}) \quad (7.10)$$

$$\equiv P_{(A,B) \times (C,D)}^{m+1} \mathbf{1} \quad (7.11)$$

taking the limit of (7.2) we get

$$M_{(A,B)} \times M_{(C,D)} \equiv M_{(A,B) \times (C,D)}. \quad (7.12)$$

□

Example 7.1.1. For streams we get

$$\text{stream } A \times \text{stream } B \equiv M_{(A, \lambda _., \mathbf{1})} \times M_{(B, \lambda _., \mathbf{1})} \equiv M_{(A, \lambda _., \mathbf{1}) \times (B, \lambda _., \mathbf{1})} \equiv \text{stream } (A \times B) \quad (7.13)$$

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

$$\forall x, \perp \sqsubseteq x \quad (8.1)$$

$$\forall x, x \sqsubseteq x \quad (8.2)$$

$$\forall x y z, x \sqsubseteq y \rightarrow y \sqsubseteq z \rightarrow x \sqsubseteq z \quad (8.3)$$

we can then define the partiality monad

The partiality monad $(-)_\perp$ 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

$$\forall x, \perp \sqsubseteq x \quad (8.4)$$

$$\forall x, x \sqsubseteq x \quad (8.5)$$

$$\forall x y z, x \sqsubseteq y \rightarrow y \sqsubseteq z \rightarrow x \sqsubseteq z \quad (8.6)$$

$$\forall x y, x \sqsubseteq y \rightarrow y \sqsubseteq x \rightarrow x \equiv y \quad (8.7)$$

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.

$$\text{now } x = x + \mathbf{1} \quad (8.8)$$

$$\text{later } y = y \quad (8.9)$$

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 $\mathbf{M}_S\text{-out}$ defined over an **M**-type.*

Proof. We first define the function

$$\begin{aligned} \alpha_{\equiv} &: \equiv \rightarrow \mathbf{P}(\equiv) \\ \alpha_{\equiv}(x, y) &:= \pi_1(\text{out } x), (\lambda b, (\pi_2(\text{out } b), \text{refl}_{(\pi_2(\text{out } x) b)})) \end{aligned} \quad (8.10)$$

and the two projections

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

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

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

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

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_{delay} for the delay monad,

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

$$\frac{R : \mathcal{U} \quad r : R}{\text{now } r \sim_{\text{delay}} \text{now } r : \mathcal{U}} \text{now} \sim \quad (8.13)$$

$$\frac{R : \mathcal{U} \quad t : \text{delay } R \quad u : \text{delay } R \quad t \sim_{\text{delay}} u : \mathcal{U}}{\text{later } t \sim_{\text{delay}} \text{later } u : \mathcal{U}} \text{later} \sim \quad (8.14)$$

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

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\text{delay}}} &: \sim_{\text{delay}} \rightarrow \mathbf{P}(\sim_{\text{delay}}) \\ \alpha_{\sim_{\text{delay}}}(a, b, \text{now} \sim r) &:= (\text{inr } r, \lambda()) \\ \alpha_{\sim_{\text{delay}}}(a, b, \text{later} \sim x \ y \ q) &:= (\text{inl } \star, \lambda _, (x, y, q)) \end{aligned} \quad (8.15)$$

then we define the projections

$$\pi_1^{\sim_{\text{delay}}} = \left(\pi_1, \text{funExt } \lambda(a, b, p), \left\{ \begin{array}{ll} (\text{inr } r, \lambda()) & p = \text{now} \sim r \\ (\text{inl } \star, \lambda _, x) & p = \text{later} \sim x \ y \ q \end{array} \right\} \right) \quad (8.16)$$

$$\pi_2^{\sim_{\text{delay}}} = \left(\pi_2, \text{funExt } \lambda(a, b, p), \left\{ \begin{array}{ll} (\text{inr } r, \lambda()) & p = \text{now} \sim r \\ (\text{inl } \star, \lambda _, y) & p = \text{later} \sim x \ y \ q \end{array} \right\} \right) \quad (8.17)$$

$$(8.18)$$

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

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

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.

$$\frac{a, b : R \quad a \equiv_R b}{\text{Ret } a \cong \text{Ret } b} \text{EqRet} \quad (8.19)$$

$$\frac{t, u : \text{itree } E \quad R \quad t \cong u}{\text{Tau } t \cong \text{Tau } u} \text{EqTau} \quad (8.20)$$

$$\frac{A : \mathcal{U} \quad e : E \quad A \quad k_1, k_2 : A \rightarrow \text{itree } E \quad R \quad t \cong u}{\text{Vis } e \ k_1 \cong \text{Tau } e \ k_2} \text{EqVis} \quad (8.21)$$

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

$$(\text{hd} \times \text{hd}) \equiv \text{hd} \circ \text{zip} \quad (8.22)$$

$$\text{zip} \circ (\text{tl} \times \text{tl}) \equiv \text{tl} \circ \text{zip} \quad (8.23)$$

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 \mathbf{M} -types

$$\text{lift}_{\mathbf{M}} \left(x : \prod_{n:\mathbb{N}} (A \rightarrow P_{S^n} \mathbf{1}) \right) \left(u : \prod_{n:\mathbb{N}} (A \rightarrow \pi_n(x_{n+1}a) \equiv x_n a) \right) (a : A) : \mathbf{M} \ S := (\lambda n, x \ n \ a), (\lambda n \ i, p \ n \ a \ i). \quad (8.24)$$

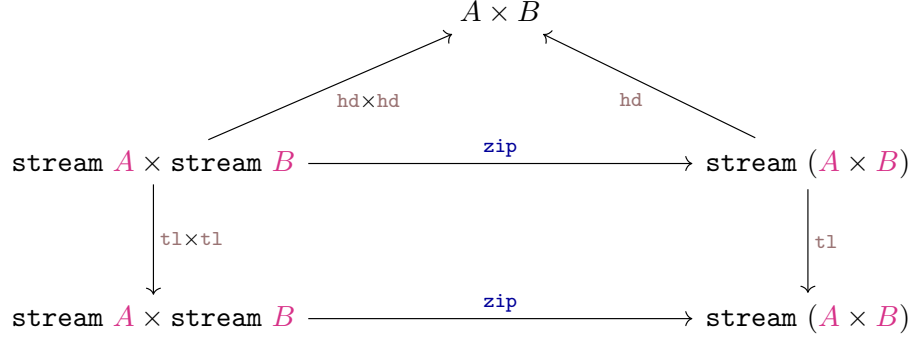


Figure 8.3: TODO

To use this definition, we first define some helper functions

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

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

we can then define

$$\mathbf{zip}_{lift} \ (x, y) := \mathbf{lift}_M \ \mathbf{zip}_X \ \mathbf{zip} \ (x, y). \quad (8.27)$$

Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\mathbf{transport}_? \ a \equiv \mathbf{zip}_{lift} \ a \quad (8.28)$$

$$\equiv \mathbf{lift}_M \ \mathbf{zip}_X \ \mathbf{zip}_\pi \ (x, y) \quad (8.29)$$

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

zero case X

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

Successor case X

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

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

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

$$\equiv \mathbf{transport}_? \ a \quad (8.35)$$

$$(8.36)$$

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

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

$$\equiv \mathbf{1}, \mathbf{refl} \quad (8.38)$$

(8.39)

successor case

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

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

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

$$\text{const}_X (n : \mathbb{N}) (c : \mathbb{N}) := \begin{cases} 1 & n = 0 \\ (c, \lambda _, \text{const}_X m c) & n = m + 1 \end{cases} \quad (8.42)$$

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

to get the definition of zero stream

$$\text{zeros} := \text{lift}_M \text{const}_X \text{const}_\pi 0. \quad (8.44)$$

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

$$\text{hd zeros} \equiv 0 \quad (8.45)$$

$$\text{tl zeros} \equiv \text{zeros} \quad (8.46)$$

Spin

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

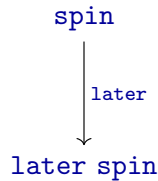


Figure 8.4: TODO

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

Conclusion

conclude on the problem statement from the introduction

Bibliography

- [1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 23–38, 2003.
- [2] Marcelo Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. *CoRR*, abs/1911.06899, 2019.
- [3] Jesse Hughes. *A study of categories of algebras and coalgebras*. PhD thesis, Carnegie Mellon University, 2001.
- [4] nLab authors. cubical type theory. <http://ncatlab.org/nlab/show/cubical%20type%20theory>, May 2020. Revision 15.
- [5] nLab authors. homotopy type theory. <http://ncatlab.org/nlab/show/homotopy%20type%20theory>, May 2020. Revision 111.
- [6] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.
- [7] Amin Timany and Matthieu Sozeau. Cumulative inductive types in coq. *LIPICs: Leibniz International Proceedings in Informatics*, 2018.
- [8] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- [9] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. Preprint available at <http://www.cs.cmu.edu/~amoertbe/papers/cubicalagda.pdf>, 2019.

Appendix A

Additions to the Cubical Agda Library

Appendix B

The Technical Details

