
(Q)M-types and Coinduction in HoTT / CTT

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science

June 1, 2020

Advisor: Bas Spitters

Abstract

We present a construction of \mathbf{M} -types from containers in a cubical type theory (CTT). We show how the containers construct a coalgebra, for which we can define a coinduction principle, making strong bisimulation imply equality. We then show constructions of \mathbf{M} -types, and how they can be quotiented to construct what we call \mathbf{QM} -types. The problem with \mathbf{QM} -types is that in general assuming that we can lift function types of equivalence classes is equivalent to the axiom of choice [4], but this can be solved by defining the quotienting relation and the type at the same time as a quotient inductive-inductive type (QIIT), which assuming the axiom of (countable) choice, is equal to the \mathbf{QM} -type. We conclude with some examples of how to use \mathbf{M} -types and some properties. All work is formalized in Cubical Agda, and the work on defining \mathbf{M} -types has been accepted to the Cubical Agda github repository.

Resumé

in Danish...

Acknowledgments

I would like to thank my supervisor Bas Spitters for some enlightening discussions and quick responses and for taking the time to do weekly meetings regarding this masters thesis. I would also like to thank my fellow students, who made this challenging but fun time less lonely even in the time of the Corona pandemic.

*Lasse Letager Hansen,
Aarhus, June 1, 2020.*

Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	1
1.1 Overview	1
1.2 Background Theory	1
1.3 Notation	4
2 M-types	7
2.1 Containers / Signatures	7
2.2 Coinduction Principle for M-types	12
3 Examples of M-types	13
3.1 Stream Formalization using M-types	13
3.2 ITrees as M-types	13
3.2.1 Delay Monad	13
3.2.2 Tree	14
3.2.3 ITrees	15
3.3 Automaton	15
3.4 General rules for constructing M-types	16
3.5 Wacky M-type	17
4 QM-types	19
4.1 Quotienting and Constructors	19
4.2 QM-types and Quotient inductive-inductive types (QIITs)	20
4.2.1 Multiset	20
4.2.2 Partiality monad	22
4.3 QM-types	28
5 Properties of M-types?	29
5.1 Closure properties of M-types	29
5.1.1 Product of M-types	29
5.1.2 Co-product	31

5.1.3	...	31
6	TODO: M-types	33
6.1	TODO: Place these subsections	33
6.1.1	Identity Bisimulation	33
6.1.2	Bisimulation of Streams	33
6.1.3	Bisimulation of Delay Monad	34
6.1.4	Bisimulation of ITrees	34
6.1.5	Zip Function	35
6.1.6	Examples of Fixed Points	36
7	Conclusion	39
7.1	Future Work	39
	Bibliography	41
A	Additions to the Cubical Agda Library	43
B	The Technical Details	45

Chapter 1

Introduction

1.1 Overview

There has been done a lot of work on understanding and formalizing inductive types in homotopy type theory (HoTT) / cubical type theory. One such body of work is W-types, which have been shown equal to inductive types. However there has not been much work on the dual concept, namely coinductive types defined using M-types. The goal of this thesis is to get an understanding of M-types. We also want to construct examples and show properties of M-types, since we want to make the theory of M-types more accessible. A useful technique used alot in constructive mathematics is quotienting free objects, as such we will also look into what it means to quotient M-types in the setting of cubical type theory, and the problems the obvious approach encounters.

In the rest of this chapter we will introduce some of the background theory and notation used in the rest of the thesis. In the second chapter we construct **M**-types from containers, and define a coinduction principle for the **M**-types. In the third chapter we give some example constructions of **M**-types. In the fourth chapter we introduce quotiented **M**-types (**QM**-types), and show equalities between these and quotient inductive-inductive types (**QIITs**), we show the construction of the partiality monad as an example. In the fifth and sixth chapter we go through various applications of the theory developed in the first couple chapters. Finally we conclude with a discussion of future research and improvements.

1.2 Background Theory

We start by giving some background theory / history on cubical type theory and summarize important concepts used in the rest of this thesis.

We will be using **type theory** as the 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. The type theory we are working in is inspired by **Martin L f Type Theory (MLTT)** / Intuitionistic type Theory (ITT), which is designed on the principles of mathematical constructivism, where any existence proof must contain a witness. Meaning a proof of existence, can be converted into an

Should a description of Set Theory be included

algorithm that finds the element making the statement true. MLTT 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 $\text{refl} : \prod_{a:A} (a = a)$.

A constructor for a type A is a function, that takes some arguments, and returns an element $a : A$, dually a destructor of A will return something given an element $a : A$. Types can be defined from a set of constructors (or destructors). We can define a type **inductively** from a set of constructors, for example the natural numbers \mathbb{N} , which can be defined as 0 or a natural number n plus one

$$\frac{}{0 : \mathbb{N}} \quad (1.1)$$

$$\frac{n : \mathbb{N}}{\text{succ } n : \mathbb{N}} \quad (1.2)$$

with an equivalence relation $\sim_{\mathbb{N}}$ defined inductively, meaning we follow the structure defined by the constructors

$$\frac{}{0 = 0} \sim_0 \quad (1.3)$$

$$\frac{n \sim_{\mathbb{N}} m}{\text{succ } n \sim_{\mathbb{N}} \text{succ } m} \sim_{\text{succ}} \quad (1.4)$$

This relation implies equality, so if $a \sim_{\mathbb{N}} b$ then $a = b$. Likewise there is a **coinductive** construction, where the focus is on the destructors instead of the constructors, an example is streams, which represents an infinite sequence of elements. Streams can be defined from the two destructors head (**hd**) and tail (**tl**), where head represents the first element, and tail represents the rest of the sequence. The inference rules are given as

$$\frac{s : \text{stream } A}{\text{hd } s : A} \quad (1.5)$$

$$\frac{s : \text{stream } A}{\text{tl } s : \text{stream } A} \quad (1.6)$$

we can again define an equivalence relation \sim_{stream} , but this time coinductively, focusing on the structure of the destructors instead

$$\frac{\text{hd } s = \text{hd } t \quad \text{tl } s \sim_{\text{stream}} \text{tl } t}{s \sim_{\text{stream}} t} \quad (1.7)$$

This relation does not give an equality in MLTT, we just get bisimilarity meaning elements "behave" the same, but they are not equivalent. To remedy this constraint, we will work in a type theory where the univalence axiom holds, using such a type theory as the foundation for mathematics is called **Univalent Foundations (UF)**. The **univalence axiom** says that equality is equivalent to equivalence

$$(A = B) \simeq (A \simeq B) \quad (1.8)$$

meaning if two objects are equivalent, then there is an equality between them, such that we can replace one by the other. This makes (strong) bisimilarity imply equality. The univalent foundations we will be using is **Homotopy type theory (HoTT)** [11]. HoTT is an intensional dependent type theory (built on MLTT) with the univalence axiom and higher inductive types. In HoTT the identity types form path spaces, so proofs of identity are not just **refl** as is the case in MLTT. Types are seen as "spaces", 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 [10]. One of the problems with "plain" HoTT is that the univalence axiom is not constructive, since it is an axiom. To remedy this we will be working in a **cubical type theory (CTT)** [6], where the univalence axiom is not an axiom, but a statement that can actually be proven, meaning we can reduce the use of the

univalence axiom, making it easier to do proofs involving the univalence axiom [9]. The reason for the name cubical type theory, is because composition is defined by square, that is given three sides of a square we get the last one, see Figure 1.1.

$$\begin{array}{ccc} A & \xrightarrow{p \cdot q \cdot r} & B \\ p^{-1} \uparrow & & \uparrow r \\ C & \xrightarrow{q} & D \end{array}$$

Figure 1.1: Composition square

example
of what
computational
"axioms"
mean

Inductively defined data types are initial algebras, meaning that they are the smallest type, with a given set of constructors / destructors. ... An algebra is an operator F with some closure relation. ... inductive types are given as the initial algebra for some functor, this can be formalized as W -types, dually the coinductive types that we are interested in, can be formalized as the final coalgebra for some functor. We will be looking at how to define some coinductive types, as M -types, and define some bisimilarity relations for these types, showing we get equality when using homotopy type theory. We will then introduce weaker notion of bisimilarity, that does not yield equality, but can be used to construct a new type, by quotienting with the relation, giving us a type where the relation gives equality. ... When working with quotiented coinductive types. ... We define the quotiented delay monad $\mathbf{Delay}/\sim_{\text{weak}}$, and want to show that we can construct a partiality monad from this construction. A problem with the partiality operation $D(-)/\sim_D$ is that countable choice is needed to show that it is a monad, however using QIIT types we can get around this problem, furthermore we can show that assuming countable choice, these two constructions are equal. Using the axiom of choice (AC) and the law of excluded middle (LEM), has problematic side effects, when using a constructive type theory, since AC and LEM does not have a constructive interpretation, so to maintain the computational aspects of HoTT and CTT, we try to not use these axioms [11, Introduction].

If you are used to working in set theory, then working in HoTT will take some getting used to. Homotopy type theory is proof relevant, which means that there might be multiple proofs of one statement, and these proofs might not be interchangeable (equal). The reason is that types in HoTT have a H -level, describing how equality behaves. We start from (-2) with contractible types, meaning there is an element which all other elements are equal to. Then there is (-1) -types which are mere propositions or \mathbf{hProp} , where all elements of the type are equal, but there might not be any. If the type is inhabited, then we say the proposition is true. The 0 -types are the \mathbf{hSets} , where all equalities between two elements x, y are equal. For 1 -types (1 -groupoids) we get equalities of equalities are equal, and then so on for homotopy n -types. Any n -type is also a $n+1$ -type, but with trivial equalities at the $n+1$ level. If we don't want to do proof relevant mathematics we can do propositional truncation, converting types to -1 -type, meaning we ignore the difference in proofs by just look at whether a type is inhabited or not. However doing this we lose some of the reasoning power of HoTT. One of the tools we get using the full power of HoTT is **Higher order inductive types (HITs)**, where we define types with point constructors and equality constructors, an example is the propositional truncation we just described, another useful example is set truncated quotients.

...

cite
some-
thing

...

final
coalge-
bra is
scary
words,
find
some-
thing
more
"noob"
friendly

W -types,
Induc-
tion,
 M -types,
Coinduc-
tion...

describe
quoti-
enting
and its
usefull-
ness by
a couple
exam-
ples

...

What
is a
monad?

...

We have formalized most of Chapter 2 (accepted to the cubical agda github: <https://github.com/agda/cubical/pull/245>) and Chapter 3 and some of Chapter 4.3 in the proof assistant / programming language Cubical Agda. A **proof assistant** helps with verifying proofs, while making the process of making proofs interactive. **Cubical Agda** [12] is an implementation of a cubical type theory (inspired by CCHM [7]) made by extending the proof assistant Agda. One of the main additions is the interval and path types. The **interval** \mathbb{I} 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 , keeping with 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**.

Cubical Agda has a hierarchy of universes, $\mathcal{U}_0, \mathcal{U}_1, \dots$, however we will leave the index implicit and write \mathcal{U} .

Universes??

1.3 Notation

The following is the notation / fonts used to denote specific definitions / concepts

- 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 $\mathbf{f} : A \rightarrow C$
- A destructor $\mathbf{f} : A \rightarrow C$
- 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 **1** while the empty type is **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 .

Furthermore we define some useful notation for casing on natural numbers.

Definition 1.3.1.

$$\langle x, f \rangle = \lambda n, \begin{cases} x & n = 0 \\ f m & n = m + 1 \end{cases} \quad (1.9)$$

Chapter 2

M-types

In this chapter we will introduce containers (aka. signatures), and use them to construct **M**-types and operations **in** and **out** on the **M**-types (Theorem 2.1.8) and showing that **M-out** is a final coalgebra (Theorem 2.1.9). We conclude the chapter by proving a coinduction principle for **M**-types (Theorem 2.2.2) [2].

Some formalization of coalgebras is missing?

2.1 Containers / Signatures

Definition 2.1.1. A Container (or signature) is a dependent pair $S = (A, B)$ for the types $A : \mathcal{U}$ and $B : A \rightarrow \mathcal{U}$.

Definition 2.1.2. A polynomial functor over the container $S = (A, B)$ is defined for types as

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

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

$$\begin{aligned} P_S f &: P_S X \rightarrow P_S Y \\ P_S f(a, g) &= (a, f \circ g). \end{aligned} \quad (2.2)$$

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

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

Since we are working in a logic 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 (2.3) is equivalent to

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

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

Definition 2.1.3. A P_S -coalgebra is defined as

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

We denote a P_S -coalgebra given by C and γ as $C-\gamma$. Coalgebra morphisms are defined as

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

We can now define M -types.

Definition 2.1.4. Given a container S , we define M -types as the type, making the coalgebra given by M_S and $\text{out} : M_S \rightarrow P_S(M_S)$ fulfill the property

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

That is $\prod_{(C-\gamma: \text{Coalg}_S)} \text{isContr}(C-\gamma \Rightarrow M_S-\text{out})$. We denote the M -type as $M_{(A,B)}$ or M_S or just M when the Container is clear from the context. When writing $\text{isContr } A$, we mean A is of H-level (-2) , that is $\sum_{x:A} \prod_{y:A} y \equiv x$ or equivalently $A \equiv \mathbf{1}$.

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

Example 2. We define streams over the type A as the M -type over the container $(A, \lambda_{-}, \mathbf{1})$. If we apply the polynomial functor to the M -type, then we get $P_{(A, \lambda_{-}, \mathbf{1})} M = A \times M_{(A, \lambda_{-}, \mathbf{1})}$, illustrated in Figure 2.1. We will that out is an isomorphism with inverse $\text{in} : P_S(M) \rightarrow M$ later in this section.

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

Figure 2.1: M -types of streams

We now have a semantic for the rules, we would expect for streams, if we let $\text{cons} = \text{in}$ and $\text{stream } A = M_{(A, \lambda_{-}, \mathbf{1})}$,

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

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

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

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

Definition 2.1.5. 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 2.2.

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

Figure 2.2: Chain of types / functions

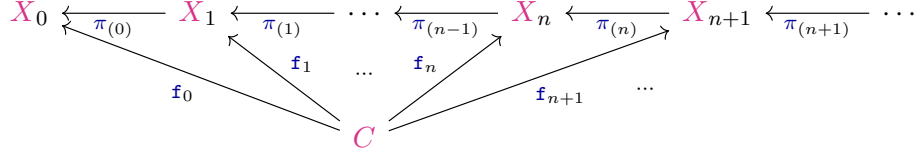


Figure 2.3: Cone

Lemma 2.1.6. For all coalgebras $C\text{-}\gamma$ for the container S , we get $C \rightarrow M_S \equiv \text{Cone}_{C\text{-}\gamma}$, where $\text{Cone} = \sum_{(f: \prod_{(n:\mathbb{N})} C \rightarrow X_n)} \prod_{(n:\mathbb{N})} \pi(n) \circ f_{n+1} \equiv f_n$ illustrated in Figure 2.3.

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

$$\text{fun}_{\text{collapse}} f = (\lambda n z, \pi_1 (f z) n), (\lambda n i a, \pi_2 (f a) n i) \quad (2.11)$$

$$\text{inv}_{\text{collapse}} (u, q) z = (\lambda n, u n z), (\lambda n i, q n i z) \quad (2.12)$$

$$\text{rinv}_{\text{collapse}} (u, q) = \text{refl}_{(u, q)} \quad (2.13)$$

$$\text{linv}_{\text{collapse}} f = \text{refl}_f \quad (2.14)$$

□

Lemma 2.1.7. 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

$$\text{fun}_{\mathcal{L}\text{collapse}} (x, r) = x_0 \quad (2.15)$$

$$\text{inv}_{\mathcal{L}\text{collapse}} x_0 = (\lambda n, \ell^{(n)} x_0), (\lambda n, \text{refl}_{(\ell^{(n+1)} x_0)}) \quad (2.16)$$

$$\text{rinv}_{\mathcal{L}\text{collapse}} x_0 = \text{refl}_{x_0} \quad (2.17)$$

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

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

Theorem 2.1.8. Given the container (A, B) we define the equality

$$\text{shift} : \mathcal{L} \equiv P\mathcal{L} \quad (2.18)$$

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

$$\text{in} = \text{transport shift} \quad (2.19)$$

$$\text{out} = \text{transport } (\text{shift}^{-1}). \quad (2.20)$$

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^P \equiv P\mathcal{L} \quad (2.21)$$

$$\mathcal{L}_{\text{unique}} : \mathcal{L} \equiv \mathcal{L}^P \quad (2.22)$$

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

$$\text{fun}_{\mathcal{L}_{\text{unique}}}(\mathbf{a}, \mathbf{b}) = \mathbf{a} \star \mathbf{b}, \quad \text{a} \langle \mathbf{b} \rangle \text{ refl}_\star, \quad \mathbf{b} \langle \mathbf{a} \rangle \quad (2.23)$$

$$\text{inv}_{\mathcal{L}_{\text{unique}}}(\mathbf{a}, \mathbf{b}) = \mathbf{a} \circ \text{succ}, \quad \mathbf{b} \circ \text{succ} \quad (2.24)$$

$$\text{rinv}_{\mathcal{L}_{\text{unique}}}(\mathbf{a}, \mathbf{b}) = \text{refl}_{(\mathbf{a}, \mathbf{b})} \quad (2.25)$$

$$\text{linv}_{\mathcal{L}_{\text{unique}}}(\mathbf{a}, \mathbf{b}) = \text{refl}_{(\mathbf{a}, \mathbf{b})} \quad (2.26)$$

The definition of α is then,

$$\mathcal{L}^P \equiv \sum_{(x : \prod_{(n : \mathbb{N})} \sum_{(a : A)} B a \rightarrow X_n)} \prod_{(n : \mathbb{N})} \pi_{(n+1)} x_{n+1} \equiv x_n \quad (2.27)$$

$$\equiv \sum_{(x : \sum_{(a : \prod_{(n : \mathbb{N})} A)} \prod_{(n : \mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(u : \prod_{(n : \mathbb{N})} B(\pi_1 x)_n \rightarrow X_n)} \prod_{(n : \mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv_* u_n \quad (2.28)$$

$$\equiv \sum_{(a : A)} \sum_{(u : \prod_{(n : \mathbb{N})} B a \rightarrow X_n)} \prod_{(n : \mathbb{N})} \pi_{(n)} \circ u_{n+1} \equiv u_n \quad (2.29)$$

$$\equiv \sum_{(a : A)} B a \rightarrow \mathcal{L} \quad (2.30)$$

$$\equiv P\mathcal{L} \quad (2.31)$$

To collapse $\sum_{(a : \prod_{(n : \mathbb{N})} A)} \prod_{(n : \mathbb{N})} a_{n+1} \equiv a_n$ to A between (2.28) and (2.29) we use Lemma 2.1.7. We use Lemma 2.1.6 for the equality between (2.29) and (2.30). The rest of the equalities are trivial. The definition of shift is

$$\text{shift} = \alpha^{-1} \cdot \mathcal{L}_{\text{unique}}. \quad (2.32)$$

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

Theorem 2.1.9. *The M-type M_S is defined as the limit for a polynomial functor P_S , meaning that it fulfills Definition 2.1.4 $\text{Final}_S \mathcal{L}$.*

Proof. So we need to show that $\prod_{(C-\gamma : \text{Coalg}_S)} \text{isContr } (C-\gamma \Rightarrow \mathcal{L}\text{-out})$, so we assume we are given some $C-\gamma : \text{Coalg}_S$, to show $(C-\gamma \Rightarrow \mathcal{L}\text{-out}) \equiv \mathbf{1}$. We have

$$C-\gamma \Rightarrow \mathcal{L}\text{-out} \quad (2.33)$$

$$\equiv \sum_{(f : C \rightarrow \mathcal{L})} (\text{out} \circ f \equiv P f \circ \gamma) \quad (2.34)$$

$$\equiv \sum_{(f : C \rightarrow \mathcal{L})} (\text{in} \circ \text{out} \circ f \equiv \text{in} \circ P f \circ \gamma) \quad (2.35)$$

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

$$\equiv \sum_{(\mathbf{f}: \mathbf{C} \rightarrow \mathbf{L})} (\mathbf{f} \equiv \mathbf{in} \circ \mathbf{Pf} \circ \gamma) \quad (2.36)$$

$$(2.37)$$

we let $\psi = \mathbf{in} \circ \mathbf{Pf} \circ \gamma$, which simplifies the expression to $\sum_{(\mathbf{f}: \mathbf{C} \rightarrow \mathbf{L})} (\mathbf{f} \equiv \psi \mathbf{f})$. We define e to be the function from right to left for the equality in Lemma 2.1.6, we then get the equality

$$\sum_{(\mathbf{f}: \mathbf{C} \rightarrow \mathbf{L})} (\mathbf{f} \equiv \psi \mathbf{f}) \equiv \sum_{(c: \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma})} (\mathbf{e} c \equiv \psi (\mathbf{e} c)) \quad (2.38)$$

if we define the function $\phi : \mathbf{Cone}_{\mathbf{S}} \rightarrow \mathbf{Cone}_{\mathbf{S}}$ as $\phi(\mathbf{u}, \mathbf{g}) = (\phi_0 \mathbf{u}, \phi_1 \mathbf{u} \mathbf{g})$ where

$$\phi_0 \mathbf{u} = \Downarrow (\lambda _, \star), \mathbf{Pf} \circ \gamma \circ \mathbf{u} \Downarrow \quad (2.39)$$

$$\phi_1 \mathbf{u} \mathbf{g} = \Downarrow \mathbf{funExt} \lambda _, \mathbf{refl}_\star, \mathbf{ap} (\mathbf{Pf} \circ \gamma) \circ \mathbf{g} \Downarrow \quad (2.40)$$

then we get the commuting square in Figure 2.4, which says $\psi(\mathbf{e} c) = \mathbf{e}(\phi c)$, so we can continue

$$\begin{array}{ccc} \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma} & \xrightarrow{e} & (\mathbf{C} \rightarrow \mathbf{L}) \\ \downarrow \phi & & \downarrow \psi \\ \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma} & \xrightarrow{e} & (\mathbf{C} \rightarrow \mathbf{L}) \end{array}$$

Figure 2.4: commuting square

the simplification

$$\sum_{(c: \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma})} (\mathbf{e} c \equiv \mathbf{e}(\phi c)) \quad (2.41)$$

We know that \mathbf{e} is part of an equality (namely Lemma 2.1.6), so it is an embedding, that is for all a, b the equality $\mathbf{e} a \equiv \mathbf{e} b$ implies $a \equiv b$, so we get

$$\sum_{(c: \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma})} (c \equiv \phi c) \quad (2.42)$$

$$\equiv \sum_{((\mathbf{u}, \mathbf{g}): \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma})} ((\mathbf{u}, \mathbf{g}) \equiv (\phi_0 \mathbf{u}, \phi_1 \mathbf{u} \mathbf{g})) \quad (2.43)$$

$$\equiv \sum_{((\mathbf{u}, \mathbf{g}): \mathbf{Cone}_{\mathbf{C} \rightarrow \gamma})} \sum_{(p: \mathbf{u} \equiv \phi_0 \mathbf{u})} \mathbf{g} \equiv_* \phi_1 \mathbf{u} \mathbf{g} \quad (2.44)$$

We split the order of the Σ 's, by unfolding the definition of \mathbf{Cone} .

$$\sum_{((\mathbf{u}, p): \sum_{(\mathbf{u}: \prod_{(n: \mathbb{N})} \mathbf{C} \rightarrow \mathbf{X}_n)} \mathbf{u} \equiv \phi_0 \mathbf{u})} \sum_{(\mathbf{g}: \prod_{(n: \mathbb{N})} \pi_{(n)} \mathbf{ou}_{n+1} \equiv \mathbf{u}_n)} \mathbf{g} \equiv_* \phi_1 \mathbf{u} \mathbf{g} \quad (2.45)$$

Now with the following two properties, given by Lemma 2.1.7, we are done

$$\mathbf{1} \equiv \left(\sum_{(\mathbf{u}: \prod_{(n: \mathbb{N})} \mathbf{C} \rightarrow \mathbf{X}_n)} \mathbf{u} \equiv \phi_0 \mathbf{u} \right) \quad (2.46)$$

$$\mathbf{1} \equiv_* \left(\sum_{(\mathbf{g} : \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathbf{u}_{n+1} \equiv \mathbf{u}_n)} \mathbf{g} \equiv_* \phi_1 \mathbf{u} \mathbf{g} \right) \quad (2.47)$$

since we get the type $\sum_{\star:1} \mathbf{1}$ which is equal to $\mathbf{1}$. \square

2.2 Coinduction Principle for \mathbf{M} -types

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

Definition 2.2.1. For all coalgebras $C-\gamma : \mathbf{Coalg}_S$, 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 $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \rightarrow \mathbf{P}_S(\overline{\mathcal{R}})$ forms a P-coalgebra $\overline{\mathcal{R}}-\alpha_{\mathcal{R}} : \mathbf{Coalg}_S$, making the diagram in Figure 2.5 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 2.5: Bisimulation for a coalgebra

Theorem 2.2.2 (Coinduction principle). *Given a relation \mathcal{R} , that is a bisimulation for an \mathbf{M} -type, then for all x, y if they are related $x \mathcal{R} y$ then they are equal $x \equiv y$.*

Proof. Given a relation \mathcal{R} , that is part of a bisimulation over a final P-coalgebra $\mathbf{M-out} : \mathbf{Coalg}_S$ we get the diagram in Figure 2.6, where $\pi_1 \overline{\mathcal{R}} = ! = \pi_2 \overline{\mathcal{R}}$ where $!$ is the unique mapping property (UMP)

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

Figure 2.6: Bisimulation principle for final coalgebra

out of the final coalgebra. Given $r : x \mathcal{R} y$ we get the equation

$$x = \pi_1 \overline{\mathcal{R}}(x, y, r) = \pi_2 \overline{\mathcal{R}}(x, y, r) = y. \quad (2.48)$$

\square

What does commute mean here?

what is !

What is the consequence of this?

Chapter 3

Examples of M-types

In this section we show some examples of types that can be constructed as M -types, and show how their constructors can be defined. We then conclude the chapter with some general observation, and define some rules for how to construct M -types.

3.1 Stream Formalization using M -types

As shown in Example 1 and Example 2 above, we can define streams as an M -type.

3.2 ITrees as M -types

Interaction trees (ITrees) [13] are used to model effectful behavior, where computations can interact with an external environment by events. ITrees are defined by the following constructors

$$\frac{r : R}{\text{Ret } r : \text{itree } E \ R} \text{I}_{\text{Ret}} \quad (3.1)$$

$$\frac{A : \mathcal{U} \quad a : E \ A \quad f : A \rightarrow \text{itree } E \ R}{\text{Vis } a \ f : \text{itree } E \ R} \text{I}_{\text{Vis}}. \quad (3.2)$$

$$\frac{t : \text{itree } E \ R}{\text{Tau } t : \text{itree } E \ R} \text{E}_{\text{Tau}}. \quad (3.3)$$

where R is the type for returned values, while E is a dependent type for events representing external interactions.

3.2.1 Delay Monad

We start by looking at ITrees without the **Vis** constructor, this type is also know as the delay monad. It can be used to model delayed computations, either returning immediately given by the constructor **now** = **Ret**, or delayed some (possibly infinite) number of steps by the constructor **later** = **Tau**. We construct this type as an M -type.

Definition 3.2.1. The delay monad can be defined as the M -type for the container

$$S = \left(R + \mathbf{1}, \begin{cases} \mathbf{0} & \text{inl } r \\ \mathbf{1} & \text{inr } \star \end{cases} \right) \quad (3.4)$$

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

complete this section

Anything to add to this section?

The polynomial functor for this container is

$$P_S(X) = \sum_{(x:R+1)} \begin{cases} \mathbf{0} & x = \text{inl } r \rightarrow X, \\ \mathbf{1} & x = \text{inr } \star \end{cases} \quad (3.5)$$

which we can simplify

$$P_S(X) = R \times (\mathbf{0} \rightarrow X) + X. \quad (3.6)$$

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

$$P_S(X) = X + R \quad (3.7)$$

meaning we get diagram in Figure 3.1. We can define the constructors **now** and **later** using **in**

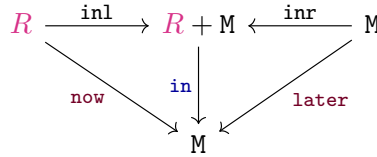


Figure 3.1: Delay monad

function for **M**-types, together with the injections **inl** and **inr**.

3.2.2 Tree

Now lets look at the example, where we remove the **Tau** constructor. This gives us a type of tree, with leaves given by **Ret**, and nodes given by **Vis** branching based on some type **A**, for an event $a : E \ A$.

Definition 3.2.2. We can define **R**-valued **E**-event trees as the **M**-type defined by the container

$$S = \left(R + \sum_{(A:\mathcal{U})} E \ A, \begin{cases} \mathbf{0} & \text{inl } r \\ A & \text{inr } (A, e) \end{cases} \right). \quad (3.8)$$

The polynomial functor for this container is

$$P_S(X) = \sum_{(x:R+\sum_{(A:\mathcal{U})} E \ A)} \begin{cases} \mathbf{0} & x = \text{inl } r \\ A & x = \text{inr } (A, e) \end{cases} \rightarrow X, \quad (3.9)$$

which simplifies to

$$P_S(X) = (R \times (\mathbf{0} \rightarrow X)) + \left(\sum_{A:\mathcal{U}} E \ A \times (A \rightarrow X) \right), \quad (3.10)$$

and further

$$P_S(X) = R + \sum_{A:\mathcal{U}} E \ A \times (A \rightarrow X). \quad (3.11)$$

We get the diagram in Figure 3.2 for the P-coalgebra. Again we can define **Ret** and **Vis** using the **in** function.

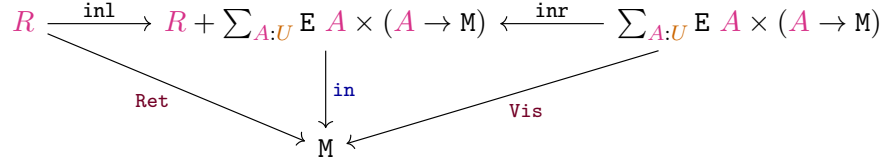


Figure 3.2: Tree Constructors

3.2.3 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.

Definition 3.2.3. We define the type of ITrees as the **M**-type given by the container

$$S = \left(R + \mathbf{1} + \sum_{A:\mathcal{U}} (E A), \begin{cases} \mathbf{0} & \text{inl } r \\ \mathbf{1} & \text{inl } (\text{inl } \star) \\ A & \text{inr } (\text{inr } (A, e)) \end{cases} \right). \quad (3.12)$$

The (reduced) polynomial functor for this container is

$$P_S(X) = R + X + \sum_{(A:\mathcal{U})} (E A \times (A \rightarrow X)) \quad (3.13)$$

Giving us the diagram in Figure 3.3, from which the constructors of the type can be defined using **in**.

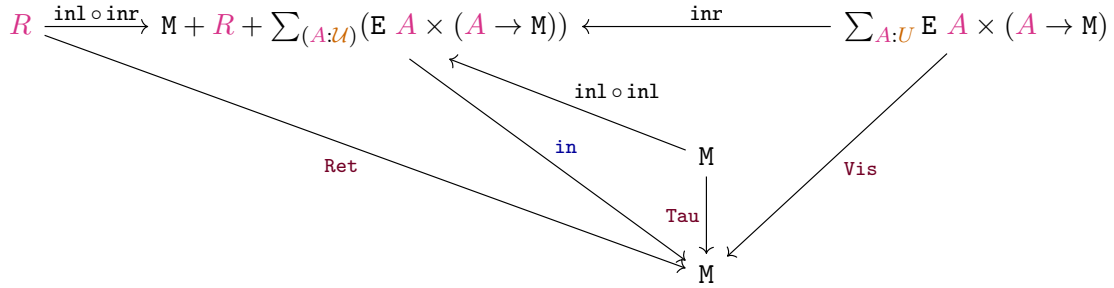


Figure 3.3: ITree constructors

3.3 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 3.4

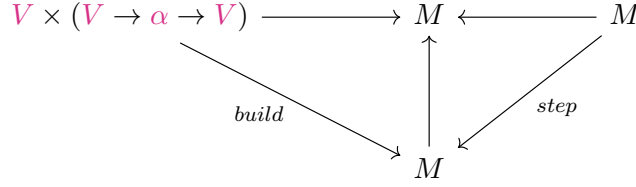


Figure 3.4: automaton

3.4 General rules for constructing **M**-types

We want to create a calculus for defining coinductive types as **M**-types. We would like to be able to define a type that has a given set of constructor / destructors rules. If we for example is given the rule

$$\frac{a : A}{\mathbf{ret} \ a : T} \quad (3.14)$$

we get that it corresponds to the **M**-type for the container $(A, \lambda _, \mathbf{0})$, while if we have something that produces an element of it self as

$$\frac{a : T}{\mathbf{tl} \ a : T} \quad (3.15)$$

the container is $(\mathbf{1}, \lambda _, \mathbf{1})$. If we want a type with both these rules, then we just take the disjoint union of the two containers

$$\left(A + \mathbf{1}, \begin{cases} \mathbf{0} & \mathbf{inl} \ a \\ \mathbf{1} & \mathbf{inr} \ \star \end{cases} \right) \quad (3.16)$$

which is the delay type. We can also define some more involved constructors, that build on the type itself

$$\frac{a : A \rightarrow T}{\mathbf{node} \ a : T} \quad (3.17)$$

has container $(\mathbf{1}, A)$. We can types with a given destructor

$$\frac{a : T}{\mathbf{hd} \ a : A} \quad (3.18)$$

has container $(A, \lambda _, \mathbf{0})$, but this is the same as for **ret**, and we do not always want both. The difference is how they are added to other constructors / destructors. Destructors are easily added together, take for example **hd** and **tl**, $(A, \lambda _, \mathbf{1})$. In general adding containers (A, B) and (C, D) for two constructors together is done by

$$\left(A + C, \begin{cases} B \ a & \mathbf{inl} \ a \\ D \ c & \mathbf{inr} \ c \end{cases} \right) \quad (3.19)$$

whereas adding containers for two destructors is done by

$$(A \times C, \lambda _, \lambda (a, c), B \ a + D \ c) \quad (3.20)$$

however combining destructors and constructors is not as simple. Anything type T that is defined using a record (except higher inductive types), will also be definable as an **M**-type. Given a record, which is a list of fields $f_1 : F_1, f_2 : F_2, \dots, f_n : F_n$, we can construct the **M**-types by the container

$$(F_1 \times F_2 \times \dots \times F_n, \lambda _, \mathbf{0}) \quad (3.21)$$

why give some arguments or illustration showing these reasonings

Is this correct? Seems correct..

where each destructor $d_n : T \rightarrow F_n$ for the field f_n will be defined as $d_n t = \pi_n (\text{out } t)$. However fields in a coinductive container may depend on previous defined fields, as given by the general list of fields $f_1 : F_1, f_2 : F_2, \dots, f_n : F_n$, where each field depends on all the previous once, this can be defined by the container

$$\left(\sum_{(f_1:F_1)} \sum_{(f_2:F_2)} \cdots \sum_{(f_{n-1}:F_{n-1})} F_n, \lambda _, \mathbf{0} \right) \quad (3.22)$$

however, if any of the destructors/fields are non dependent, then the can be added as a product (\times) instead of a dependent product (Σ). Furthermore the fields may construct an element of the type of the record T , however anything after that field cannot on it, since it will break the strictness requirements of the record / coinductive type. As an example let f_1 be a type and f_2 be the function with type $F_2 = f_1 \rightarrow (f_1 \rightarrow A) \rightarrow M$, which by currying is equal to $f_1 \times (f_1 \rightarrow A) \rightarrow M$, we can then define by the container

$$\left(\sum_{(f_1:\mathcal{U})} \left(\mathbf{1} \times \sum_{(f_3:F_3)} \cdots \sum_{(f_{n-1}:F_{n-1})} F_n \right), \lambda (f_1, \star, f_3, \dots), F_2 \right) \quad (3.23)$$

where F_2 have been moved to the last part of the container, we can even leave out the " $\mathbf{1} \times$ " from the container. The types of the field can also be dependent $F_2 = (x : f_1) \rightarrow B \ x \rightarrow M$, but again by currying we can get $F_2 : \sum_{(x:f_1)} B \ x \rightarrow M$ which is defined by the container

$$\left(\sum_{(f_1:\mathcal{U})} \sum_{(f_3:F_3)} \cdots \sum_{(f_{n-1}:F_{n-1})} F_n, \lambda (f_1, f_3, \dots), \sum_{x:f_1} (B \ x) \right) \quad (3.24)$$

so we would also expect that a type defined as a (coinductive) record is equal to the version defined as a M -type.

But we run into problems if ...

3.5 Wacky M -type

We end this chapter by showing of some wacky M -type, that utilizes the definition of the M -type to the fullest.

Definition 3.5.1. We define a wacky M -type by the following container

$$\left(\mathbb{N} + \mathbb{N}, \begin{cases} \mathbb{N} & \text{inl } 0 \\ \mathbf{0} & \text{inl } x \wedge x \text{ is odd} \\ \mathbf{0} & \text{inr } x \wedge x \text{ is even} \\ \mathbf{1} & o.w. \end{cases} \right) \quad (3.25)$$

this container gives the M -type and constructors / destructors shown in Figure 3.5 The type can be interpreted as a stream of coproducts of natural numbers, that terminates whenever it is the

which is the case, proof needed!

Problem cases

What differs from W and M types for closure of constructors / destructors?

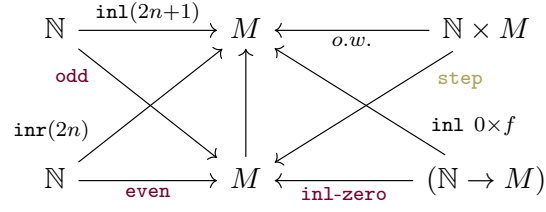


Figure 3.5: wacky \mathbf{M} -type

left injection and even, or the right injection and odd, and whenever the left injection is zero, it splits in a branches indexed by the natural numbers.

This show that we can define coinductive types, with rather complex structure.

define a wacky \mathbf{M} -type showing some of the use of \mathbf{M} -types / complexity

Chapter 4

QM-types

In this chapter we will introduce quotients, and show how to construct quotiented **M**-types which we call **QM**-types. We show you can be constructed a QIIT that is equal to the **QM**-type assuming axiom of choice.

better
introduc-
tion to
chapter,
and ref-
erence
to main
points

4.1 Quotienting and Constructors

A very useful way to construct a specific type, is by constructing the "free" such type, and then quotienting with a relation. We can define quotiented types by using the higher inductive type defined by the constructors

$$\frac{x : A}{[x] : A/\mathcal{R}} \quad (4.1) \quad \frac{x, y : A/\mathcal{R} \quad r : x \mathcal{R} y}{\text{eq}/ x y r : x \equiv y} \quad (4.2) \quad \frac{}{\text{squash}/ : \text{isSet } (A/\mathcal{R})} \quad (4.3)$$

with recursor and eliminator defined as follows.

Definition 4.1.1 (Recursor for quotient). For all elements $x, y : A$, functions $\mathbf{f} : A \rightarrow B$ and relations $\mathbf{g} : x \mathcal{R} y \rightarrow \mathbf{f} x \equiv \mathbf{f} y$, then if B is a set, we get a function from A/\mathcal{R} to B , defined by case as

$$\begin{aligned} \text{rec } [a] &= \mathbf{f} a \\ \text{rec } (\text{eq}/ _ _ r i) &= \mathbf{g} r i \\ \text{rec } (\text{squash}/ a b p q i j) &= B_{\text{set}} (\text{rec } a) (\text{rec } b) (\text{ap rec } p) (\text{ap rec } q) i j \end{aligned} \quad (4.4)$$

Definition 4.1.2 (Propositional eliminator for quotient). Given a proposition $\mathbf{P} : A/\mathcal{R} \rightarrow \mathcal{U}$ for quotients, that fulfills $\mathbf{P}_{\text{prop}} : \prod_{(x:A/\mathcal{R})} \text{isProp } (\mathbf{P} x)$, then if $\mathbf{f} : \prod_{(a:A)} \mathbf{P} [a]$ we get a function from $x : A/\mathcal{R}$ to $\mathbf{P} x$, defined as

$$\begin{aligned} \text{elimProp } [a] &= \mathbf{f} a \\ \text{elimProp } (\text{eq}/ a b i) &= \mathbf{P}_{\text{prop}} (\text{elimProp } a) (\text{elimProp } b) (\text{eq}/ a b) i \\ \text{elimProp } (\text{squash}/ a b p q i j) &= \\ &\quad \text{isSet} \rightarrow \text{isSetDep } (\text{isProp} \rightarrow \text{isSet} \circ \mathbf{P}_{\text{prop}}) \\ &\quad (\text{elimProp } a) (\text{elimProp } b) (\text{ap elimProp } p) (\text{ap elimProp } q) \\ &\quad (\text{squash}/ a b p q) i j \end{aligned} \quad (4.5)$$

where $\text{isSet} \rightarrow \text{isSetDep}$ takes a function $\prod_{a:A} \text{isSet} (B\ a)$ to the dependent version $\text{isSetDep } A\ B$. Using this eliminator we can do propositional elimination of a quotient, by supplying the base case $P\ [a]$.

We can now construct some more interesting data types, by set quotienting \mathbf{M} -types, which we call \mathbf{QM} -types, some examples follow in the following section.

4.2 \mathbf{QM} -types and Quotient inductive-inductive types (QIITs)

In this section we will show some examples of quotiented \mathbf{M} -types, and alternative ways of defining an equal type, but using quotient inductive-inductive types instead. A quotient inductive-inductive type (QIIT) is a type defined at the same time as a relation over that type. Furthermore QIITs are set truncated. We believe that every \mathbf{QM} -type is equal (under the axiom of choice) to a QIIT, however QIITs are more general than our \mathbf{QM} -types, so we cannot expect every QIIT to have a corresponding \mathbf{QM} -type.

4.2.1 Multiset

In this subsection we define infinite trees, where the order of subtrees does not matter also known as multisets [3][4][8].

Definition 4.2.1. We define X -valued \mathbb{N} -branching trees $T\ X$ as the \mathbf{M} -type defined by the container

$$\left(X + 1, \begin{cases} 0 & \text{inl } x \\ \mathbb{N} & \text{inr } \star \end{cases} \right) \quad (4.6)$$

For which we get the constructors

$$\frac{x : X}{\text{leaf } x : T\ X} \quad (4.7) \qquad \frac{f : \mathbb{N} \rightarrow T\ X}{\text{node } f : T\ X} \quad (4.8)$$

We want to define trees where the permutation does not matter, that is multisets, we do this by ensuring the following rule is true

$$\frac{f : \mathbb{N} \rightarrow T\ X \quad g : \mathbb{N} \rightarrow \mathbb{N} \quad \text{isIso } g}{\text{node } f \equiv \text{node } (f \circ g)} \text{perm} \quad (4.9)$$

One way to define this type is as the \mathbf{QM} -type given by quotienting $T\ X$ by the relation \sim_T defined by the constructors

$$\frac{x \equiv y}{\text{leaf } x \sim \text{leaf } y} \sim_{\text{leaf}} \quad (4.10) \qquad \frac{\prod_{(n:\mathbb{N})} f\ n \sim g\ n}{\text{node } f \sim \text{node } g} \sim_{\text{node}} \quad (4.11)$$

$$\frac{f : \mathbb{N} \rightarrow T\ X \quad g : \mathbb{N} \rightarrow \mathbb{N} \quad \text{isIso } g}{\text{node } f \equiv \text{node } (f \circ g)} \sim_{\text{perm}} \quad (4.12)$$

Again we have the problem that we cannot lift the node to the quotiented type, without the use of the axiom of choice. However we can define multisets as a QIIT $\mathbf{MS}\ X$, with the constructors leaf , node , perm and set quotiented $\mathbf{MS-isSet}$. These two ways of constructing a type for multisets are equal assuming the axiom of (countable) choice.

preserves
the
H-level

we can-
not al-
ways lift
the con-
struc-
tors of
 \mathbf{M} -type
to the quo-
tiented
type

Go into
more
details!

Definition 4.2.2. There is a function from $\mathsf{T} \ X$ to $\mathsf{MS} \ X$

$$\begin{aligned} \mathsf{T} \rightarrow \mathsf{MS} \ (\mathsf{leaf}_T x) &= \mathsf{leaf}_{\mathsf{MS}} x \\ \mathsf{T} \rightarrow \mathsf{MS} \ (\mathsf{node}_T f) &= \mathsf{node}_{\mathsf{MS}} (\mathsf{T} \rightarrow \mathsf{MS} \circ f) \end{aligned} \quad (4.13)$$

This function takes weakly bisimilar objects to equal once.

Lemma 4.2.3. *If $x, y : \mathsf{T} \ X$ are weakly bisimilar $p : x \sim_T y$ then $\mathsf{T} \rightarrow \mathsf{MS} \ x \equiv \mathsf{T} \rightarrow \mathsf{MS} \ y$.*

Proof. We do the proof by casing on the weak bisimilarity.

$$\begin{aligned} \mathsf{T} \rightarrow \mathsf{MS} \sim \rightarrow \equiv (\sim_{\mathsf{leaf}} p) &= \mathsf{ap} \ \mathsf{leaf}_{\mathsf{MS}} \ p \\ \mathsf{T} \rightarrow \mathsf{MS} \sim \rightarrow \equiv (\sim_{\mathsf{node}} k) &= \mathsf{ap} \ \mathsf{node}_{\mathsf{MS}} \ (\mathsf{funExt} \ (\mathsf{T} \rightarrow \mathsf{MS} \sim \rightarrow \equiv \circ k)) \\ \mathsf{T} \rightarrow \mathsf{MS} \sim \rightarrow \equiv (\sim_{\mathsf{perm}} f \ g \ e) &= \mathsf{perm} \ (\mathsf{T} \rightarrow \mathsf{MS} \circ f) \ g \ e \end{aligned} \quad (4.14)$$

□

With this lemma, we can lift the function $\mathsf{T} \rightarrow \mathsf{MS}$ to the quotient.

Definition 4.2.4. There is a function $\mathsf{T} / \sim \rightarrow \mathsf{MS}$ from $\mathsf{T} \ X / \sim_T$ to $\mathsf{MS} \ X$, defined using the recursor for quotients defined in Definition 4.1.1, with $f = \mathsf{T} \rightarrow \mathsf{MS}$ and $g = \mathsf{T} \rightarrow \mathsf{MS} \sim \rightarrow \equiv$ and MS is a set by $\mathsf{MS-isSet}$

Lemma 4.2.5. *Given an equality $\mathsf{T} \rightarrow \mathsf{MS} \ x \equiv \mathsf{T} \rightarrow \mathsf{MS} \ y$ then $x \sim_T y$.*

Proof. If x and y are leafs with values a and b then $a \equiv b$ by the injectivity of the constructor $\mathsf{leaf}_{\mathsf{MS}}$, making a bisimilarity using \sim_{leaf} . If x and y are nodes defined by functions f and g , then by the injectivity of the constructor $\mathsf{node}_{\mathsf{MS}}$, we get $\prod_{(n:\mathbb{N})} f \ n \equiv g \ n$ and by induction we get $\prod_{(n:\mathbb{N})} f \ n \sim_T g \ n$, making us able to use \sim_{node} to construct the bisimilarity. We do not get any other equalities, since $\mathsf{leaf}_{\mathsf{MS}}$ and $\mathsf{node}_{\mathsf{MS}}$ are disjoint. □

Lemma 4.2.6. *The function $\mathsf{T} / \sim \rightarrow \mathsf{MS}$ is injective, meaning $\mathsf{T} / \sim \rightarrow \mathsf{MS} \ x \equiv \mathsf{T} / \sim \rightarrow \mathsf{MS} \ y$ implies $x \equiv y$.*

Proof. We show injectivity by doing propositional elimination defined in Definition 4.1.2, with

$$P = (\lambda x, \mathsf{T} / \sim \rightarrow \mathsf{MS} \ x \equiv \mathsf{T} / \sim \rightarrow \mathsf{MS} \ y \rightarrow x \equiv y) \quad (4.15)$$

which is a proposition for all x by $P_{\mathsf{prop}} = \lambda x, \mathsf{isProp}\Pi (\lambda _, \mathsf{squash} / x \ y)$. We do it twice, first for x and then for y where we define P and P_{prop} similarly, but with $x = [a]$ since it has already been propositionally eliminated.

$$\mathsf{elimProp} (\lambda a, \mathsf{elimProp} (\lambda b, \mathsf{eq} / a \ b \circ \mathsf{T} \rightarrow \mathsf{MS} \equiv \rightarrow \sim a \ b) \ y) \ x \quad (4.16)$$

where $\mathsf{T} \rightarrow \mathsf{MS} \equiv \rightarrow \sim$ is Lemma 4.2.5. □

Lemma 4.2.7. *The function $\mathsf{T} / \sim \rightarrow \mathsf{MS}$ is surjective ($\prod_{(b:\mathsf{MS})} \|\Sigma_{(x:\mathsf{T})} \mathsf{T} / \sim \rightarrow \mathsf{MS} \ x \equiv b\|$), assuming the axiom of choice.*

Proof. We only need to look at the point constructors of \mathbf{MS} . For the leaf case, we have the simple equality $\mathbf{T}/\sim \rightarrow \mathbf{MS} [\mathbf{leaf}_T x] \equiv \mathbf{leaf}_{\mathbf{MS}} x$. For the node case with $\mathbf{node} f$ then by induction, surjection of $[\cdot]$ and the axiom of choice, we get $g : \mathbb{N} \rightarrow \mathbf{T} X$ such that $\prod_{(n:\mathbb{N})} \mathbf{T}/\sim \rightarrow \mathbf{MS} [g n] \equiv f n$, making $\mathbf{T}/\sim \rightarrow \mathbf{MS} [\mathbf{node} g] \equiv \mathbf{node} f$. \square

Theorem 4.2.8. *There is an equality between the types $\mathbf{T} X / \sim_T$ and $\mathbf{MS} X$, assuming the axiom of choice.*

Proof. Since the function $\mathbf{T}/\sim \rightarrow \mathbf{MS}$ is injective and surjective, it becomes an equality. \square

4.2.2 Partiality monad

In this subsection 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 4.2.9 (Partiality Monad). A simple example of a quotient inductive-inductive type is the partiality monad $(-)_\perp$ over a type R , defined by the constructors

$$\overline{R_\perp : \mathcal{U}} \quad (4.17) \quad \overline{\perp : R_\perp} \quad (4.18) \quad \frac{a : R}{\eta a : R_\perp} \quad (4.19)$$

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

$$\frac{s : \mathbb{N} \rightarrow R_\perp \quad b : \prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp s_{n+1}}{\bigsqcup (s, b) : R_\perp} \quad (4.20) \quad \frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp p q : x \equiv y} \quad (4.21)$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x} \sqsubseteq_{\text{refl}} \quad (4.22) \quad \frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z} \sqsubseteq_{\text{trans}} \quad (4.23) \quad \frac{x : R_\perp}{\perp \sqsubseteq_\perp x} \sqsubseteq_{\text{never}} \quad (4.24)$$

$$\frac{s : \mathbb{N} \rightarrow R_\perp \quad b : \prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp s_{n+1}}{\prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp \bigsqcup (s, b)} \quad (4.25) \quad \frac{\prod_{(n:\mathbb{N})} s_n \sqsubseteq_\perp x}{\bigsqcup (s, b) \sqsubseteq_\perp x} \quad (4.26)$$

and finally set truncated

$$\frac{p, q : x \sqsubseteq_\perp y}{p \equiv q} (-)_\perp\text{-isSet} \quad (4.27)$$

4.2.2.1 Delay monad to Sequences

Definition 4.2.10. We define

$$\text{Seq}_R = \sum_{(s:\mathbb{N} \rightarrow R+1)} \text{isMon } s \quad (4.28)$$

where

$$\text{isMon } s = \prod_{(n:\mathbb{N})} (s_n \equiv s_{n+1}) + ((s_n \equiv \text{inr } \star) \times (s_{n+1} \neq \text{inr } \star)) \quad (4.29)$$

meaning a sequence 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 $\text{inr } \star$.

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

introduction to subsection

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 $s_n \neq \text{inr } \star$. Thus we can write **isMon** as

$$\text{isMon } s = \prod_{(n:\mathbb{N})} (s_n \equiv s_{n+1}) + ((s_n \uparrow_{R+1}) \times (s_{n+1} \downarrow_{R+1})) \quad (4.30)$$

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

$$\text{now}_{Seq} r = (\lambda _, \text{inl } r), (\lambda _, \text{inl refl}) \quad (4.31)$$

$$\text{never}_{Seq} = (\lambda _, \text{inr } \star), (\lambda _, \text{inl refl}) \quad (4.32)$$

Some comment about decidable equivalence needed to show that $s_{n+1} \neq \text{inr } \star$

Definition 4.2.11. We can shift a sequence (s, q) by inserting an element (and an equality) (z_s, z_q) at $n = 0$,

$$\text{shift } (s, q) (z_s, z_q) = \begin{cases} z_s & n = 0 \\ s_m & n = m + 1 \end{cases}, \begin{cases} z_q & n = 0 \\ q_m & n = m + 1 \end{cases}, \quad (4.33)$$

Definition 4.2.12. We can unshift a sequence by removing the first element of the sequence

$$\text{unshift } (s, q) = s \circ \text{succ}, q \circ \text{succ}. \quad (4.34)$$

Lemma 4.2.13. *The function*

$$\text{shift-unshift } (s, q) = \text{shift } (\text{unshift } (s, q)) (s_0, q_0) \quad (4.35)$$

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 4.2.14. *The function*

$$\text{unshift-shift } (s, q) = \text{unshift } (\text{shift } (s, q) _) \quad (4.36)$$

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 **delay** R and **Seq** R , where **later** are equivalent to shifts, and **now** r is equivalent terminated sequence with value r . We do this by defining equivalence functions, and the left and right identities.

Lemma 4.2.15 (**inl** \neq **inr**). *For any two elements $x = \text{inl } a$ and $y = \text{inr } b$ then $x \neq y$.*

Proof. The constructors **inl** and **inr** are disjoint, so there does not exists a path between them, meaning constructing one is a contradiction. \square

Definition 4.2.16. We define a function from $\text{Delay } R$ to Seq_R

$$\begin{aligned} \text{Delay} \rightarrow \text{Seq} (\text{now } r) &= \text{now}_{\text{Seq}} r \\ \text{Delay} \rightarrow \text{Seq} (\text{later } x) &= \text{shift } (\text{Delay} \rightarrow \text{Seq } x) \left(\text{inr } \star, \begin{cases} \text{inr } (\text{refl}, \text{inl} \neq \text{inr}) & x = \text{now } _ \\ \text{inl } \text{refl} & x = \text{later } _ \end{cases} \right) \end{aligned} \quad (4.37)$$

Definition 4.2.17. We define function from Seq_R to $\text{Delay } R$

$$\text{Seq} \rightarrow \text{Delay} (s, q) = \begin{cases} \text{now } r & s_0 = \text{inl } r \\ \text{later } (\text{Seq} \rightarrow \text{Delay} (\text{unshift } (s, q))) & s_0 = \text{inr } \star \end{cases} \quad (4.38)$$

Theorem 4.2.18. The type Seq_R is equal to $\text{Delay } R$

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

$$\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (s, q)) \equiv (s, q) \quad (4.39)$$

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

$$\text{now}_{\text{Seq}} r \equiv (s, q) \quad (4.40)$$

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

$$\text{shift } (\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (\text{unshift } (s, q)))) (\text{inr } \star, _) \equiv (s, q) \quad (4.41)$$

by the induction hypothesis we get

$$\text{Delay} \rightarrow \text{Seq} (\text{Seq} \rightarrow \text{Delay} (\text{unshift } (s, q))) \equiv \text{unshift } (s, q) \quad (4.42)$$

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

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

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

$$\text{Seq} \rightarrow \text{Delay} (\text{Delay} \rightarrow \text{Seq } t) \equiv t \quad (4.43)$$

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

$$\text{later } (\text{Seq} \rightarrow \text{Delay} (\text{unshift } (\text{shift } (\text{Delay} \rightarrow \text{Seq } x)))) \equiv \text{later } x \quad (4.44)$$

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.

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

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

Show
this

Show
this

4.2.2.2 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 4.2.19 (Sequence Termination). The following relations says that a sequence $(\mathbf{s}, \mathbf{q}) : \text{Seq}_R$ terminates with a given value $r : R$,

$$(\mathbf{s}, \mathbf{q}) \downarrow_{\text{Seq}} r = \sum_{(n:\mathbb{N})} \mathbf{s}_n \downarrow_{R+1} r. \quad (4.45)$$

Definition 4.2.20 (Sequence Ordering).

$$(\mathbf{s}, \mathbf{q}) \sqsubseteq_{\text{Seq}} (\mathbf{t}, \mathbf{p}) = \prod_{(a:R)} (\|\mathbf{s} \downarrow_{\text{Seq}} a\| \rightarrow \|\mathbf{t} \downarrow_{\text{Seq}} a\|) \quad (4.46)$$

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

Definition 4.2.21. There is a conversion from $R + 1$ to the partiality monad R_\perp

$$\begin{aligned} \text{Maybe} \rightarrow (-)_\perp (\text{inl } r) &= \eta \ r \\ \text{Maybe} \rightarrow (-)_\perp (\text{inr } \star) &= \perp \end{aligned} \quad (4.47)$$

Definition 4.2.22 (Maybe Ordering). Given some $x, y : R + 1$, the ordering relation is defined as

$$x \sqsubseteq_{R+1} y = (x \equiv y) + ((x \downarrow_{R+1}) \times (y \uparrow_{R+1})) \quad (4.48)$$

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

$$\text{isMon } \mathbf{s} = \prod_{(n:\mathbb{N})} \mathbf{s}_n \sqsubseteq_{R+1} \mathbf{s}_{n+1} \quad (4.49)$$

This rewriting confirms that if `isMon s`, then \mathbf{s} is monotone, and therefore a sequence of partial values.

Lemma 4.2.23. The function `Maybe → (-)⊥` is monotone, that is, if $x \sqsubseteq_{A+1} y$, 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.

$$\begin{aligned} \text{Maybe} \rightarrow (-)_\perp \text{-mono } (\text{inl } p) &= \\ \text{subst } (\lambda a, \text{Maybe} \rightarrow (-)_\perp x \sqsubseteq_\perp \text{Maybe} \rightarrow (-)_\perp a) \ p &(\sqsubseteq_{\text{refl}} (\text{Maybe} \rightarrow (-)_\perp x)) \\ \text{Maybe} \rightarrow (-)_\perp \text{-mono } (\text{inr } (p, _)) &= \\ \text{subst } (\lambda a, \text{Maybe} \rightarrow (-)_\perp a \sqsubseteq_\perp \text{Maybe} \rightarrow (-)_\perp y) \ p^{-1} &(\sqsubseteq_{\text{never}} (\text{Maybe} \rightarrow (-)_\perp y)) \end{aligned} \quad (4.50)$$

□

Definition 4.2.24. There is a function taking a sequence to an increasing sequence

$$\begin{aligned} \text{Seq} \rightarrow \text{incSeq} \\ \text{Seq} \rightarrow \text{incSeq } (g, q) &= \text{Maybe} \rightarrow (-)_\perp \circ g, \text{Maybe} \rightarrow (-)_\perp \text{-mono} \circ q \end{aligned} \quad (4.51)$$

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

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

Definition 4.2.25. There is a function taking a sequence to the partiality monad

$$\begin{aligned} \text{Seq} \rightarrow (-)_\perp &: \text{Seq}_A \rightarrow A_\perp \\ \text{Seq} \rightarrow (-)_\perp (g, q) &= \bigsqcup \circ \text{Seq} \rightarrow \text{incSeq} \end{aligned} \quad (4.52)$$

Lemma 4.2.26. The function $\text{Seq} \rightarrow (-)_\perp$ is monotone.

$$\text{Seq} \rightarrow (-)_\perp \text{-mono} : \text{isSet } A \rightarrow (x \ y : \text{Seq}_A) \rightarrow x \sqsubseteq_{\text{seq}} y \rightarrow \text{Seq} \rightarrow (-)_\perp x \sqsubseteq_\perp \text{Seq} \rightarrow (-)_\perp y \quad (4.53)$$

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

Lemma 4.2.27. If two sequences x, y are weakly bisimilar, then $\text{Seq} \rightarrow (-)_\perp x \equiv \text{Seq} \rightarrow (-)_\perp y$

Proof.

$$\begin{aligned} \text{Seq} \rightarrow (-)_\perp \sim \rightarrow \equiv A_{\text{set}} \ x \ y \ (p, q) = \\ \alpha_\perp (\text{Seq} \rightarrow (-)_\perp \text{-mono} \ A_{\text{set}} \ x \ y \ p) (\text{Seq} \rightarrow (-)_\perp \text{-mono} \ A_{\text{set}} \ y \ x \ q) \end{aligned} \quad (4.54)$$

\square

The recursor for quotients Definition 4.1.1 allows us to lift the function $\text{Seq} \rightarrow (-)_\perp$ to the quotient

Definition 4.2.28. We can define a function $\text{Seq}/\sim \rightarrow (-)_\perp$ from Seq_A to A_\perp , where $A_{\text{set}} : \text{isSet } A$ as

$$\text{Seq}/\sim \rightarrow (-)_\perp = \text{rec Seq} \rightarrow (-)_\perp (\text{Seq} \rightarrow (-)_\perp \sim \rightarrow \equiv A_{\text{set}}) (-)_\perp \text{-isSet} \quad (4.55)$$

Lemma 4.2.29. 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

$$s \sim_{\text{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\text{seq}} r\| \rightarrow \|y \downarrow_{\text{seq}} r\| \right) \times \left(\prod_{(r:R)} \|y \downarrow_{\text{seq}} r\| \rightarrow \|x \downarrow_{\text{seq}} r\| \right) \quad (4.56)$$

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

Proof. We use propositional elimination of quotients Definition 4.1.2 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

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

describe instead of proof term!

Should this be formalized?

Convert to text, instead of a proof term!?

$$\begin{aligned}
& \text{elimProp} \\
& (\lambda a, \text{Seq}/\sim\rightarrow(-)_{\perp} a \equiv \text{Seq}/\sim\rightarrow(-)_{\perp} y \rightarrow a \equiv y) \\
& (\lambda a, \text{isProp}\Pi (\lambda _, \text{squash}/ a y)) \\
& (\lambda a, \text{elimProp} \\
& \quad (\lambda b, \text{Seq}\rightarrow(-)_{\perp} a \equiv \text{Seq}/\sim\rightarrow(-)_{\perp} b \rightarrow [a] \equiv b) \\
& \quad (\lambda b, \text{isProp}\Pi (\lambda _, \text{squash}/ [a] b)) \\
& \quad (\lambda b, (\text{eq}/ a b) \circ (\text{Seq}\rightarrow(-)_{\perp} \text{-isInjective } a b)) \\
& \quad y) \\
& x
\end{aligned} \tag{4.57}$$

where $\text{Seq}\rightarrow(-)_{\perp} \text{-isInjective}$ is (4.2.29), □

Lemma 4.2.31. *For all constant sequences s , where all elements have the same value v , we get $\text{Seq}\rightarrow(-)_{\perp} s \equiv \text{Maybe}\rightarrow(-)_{\perp} v$.*

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. □

Lemma 4.2.32. *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 (4.2.31). For the least upper bound $\bigsqcup(\mathbf{s}, \mathbf{b})$, we translate to the (increasing) sequence, defined by (\mathbf{s}, \mathbf{b}) . □

Lemma 4.2.33. *Assuming countable choice, the function $\text{Seq}/\sim\rightarrow(-)_{\perp}$ is surjective*

Proof. □

Theorem 4.2.34. *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. □

Building the weak bisimulation on the M-type as a M-type - Is this possible? Yes! Should it be included?

Building the Partiality Monad as a limit (Dialgebra?) - Is this possible?

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

more precise description!

Complete the rest of the proof!

Complete proof

4.3 QM-types

We want to define what a QM-type means in general, we draw inspiration from QW-types [8], quotient containers [?] and the LEAN paper "Data types as quotients of polynomial functors" [5]. From what we have seen in the previous section, equality construction between QM-types and QIITs are given as

- A function $\text{QM} \rightarrow \text{QIIT}$ from QM to QIIT
- A proof $\text{QM} \rightarrow \text{QIIT} \sim \rightarrow \equiv$ that $x \sim_{\text{QM}} y$ implies $\text{QM} \rightarrow \text{QIIT } x \equiv \text{QM} \rightarrow \text{QIIT } y$
- Lifting $\text{QM} \rightarrow \text{QIIT}$ to $\text{QM} / \sim \rightarrow \text{QIIT}$ using the quotient recursor with $\text{QM} \rightarrow \text{QIIT} \sim \rightarrow \equiv$ and QIIT-isSet
- Showing injectivity by using propositional elimination of the quotient together with the inverse of $\text{QM} \rightarrow \text{QIIT} \sim \rightarrow \equiv$, namely $\text{QM} \rightarrow \text{QIIT-injective}$ saying that $\text{QM} \rightarrow \text{QIIT } x \equiv \text{QM} \rightarrow \text{QIIT } y$ implies $x \sim_{\text{QM}} y$.
- Lastly we show surjectivity by induction using the eliminator of QIIT and the axiom of choice. Another thing that comes into play is the surjectivity of $[\cdot]$.

Lets try this strategy in general, lets say we are given an M-type and weak bisimulation \sim_M on that type. We then define a QIIT with all the constructors of the M-type, given as

Cofree Coalgebra / Dialgebra – Is this relevant?

Chapter 5

Properties of M-types?

5.1 Closure properties of M-types

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

5.1.1 Product of M-types

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

Definition 5.1.1. The product of two containers is defined as [1]

$$(\mathbf{A}, \mathbf{B}) \times (\mathbf{C}, \mathbf{D}) \equiv (\mathbf{A} \times \mathbf{C}, \lambda(a, c), \mathbf{B} a \times \mathbf{D} c). \quad (5.1)$$

We can lift this rule, through the diagram in Figure 5.1, used to define M-types.

Theorem 5.1.2. For any $n : \mathbb{N}$ the following is true

$$\mathbf{P}_{(\mathbf{A}, \mathbf{B})}^n \mathbf{1} \times \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^n \mathbf{1} \equiv \mathbf{P}_{(\mathbf{A}, \mathbf{B}) \times (\mathbf{C}, \mathbf{D})}^n \mathbf{1}. \quad (5.2)$$

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

$$\mathbf{P}_{(\mathbf{A}, \mathbf{B})}^m \mathbf{1} \times \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^m \mathbf{1} \equiv \mathbf{P}_{(\mathbf{A}, \mathbf{B}) \times (\mathbf{C}, \mathbf{D})}^m \mathbf{1}, \quad (5.3)$$

in the following

$$\mathbf{P}_{(\mathbf{A}, \mathbf{B})}^{m+1} \mathbf{1} \times \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^{m+1} \mathbf{1} \quad (5.4)$$

$$\equiv \mathbf{P}_{(\mathbf{A}, \mathbf{B})}(\mathbf{P}_{(\mathbf{A}, \mathbf{B})}^m \mathbf{1}) \times \mathbf{P}_{(\mathbf{C}, \mathbf{D})}(\mathbf{P}_{(\mathbf{C}, \mathbf{D})}^m \mathbf{1}) \quad (5.5)$$

$$\equiv \sum_{a:A} \mathbf{B} a \rightarrow \mathbf{P}_{(\mathbf{A}, \mathbf{B})}^m \mathbf{1} \times \sum_{c:C} \mathbf{D} c \rightarrow \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^m \mathbf{1} \quad (5.6)$$

$$\equiv \sum_{a,c:A \times C} (\mathbf{B} a \rightarrow \mathbf{P}_{(\mathbf{A}, \mathbf{B})}^m \mathbf{1}) \times (\mathbf{D} c \rightarrow \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^m \mathbf{1}) \quad (5.7)$$

$$\equiv \sum_{a,c:A \times C} \mathbf{B} a \times \mathbf{D} c \rightarrow \mathbf{P}_{(\mathbf{A}, \mathbf{B})}^m \mathbf{1} \times \mathbf{P}_{(\mathbf{C}, \mathbf{D})}^m \mathbf{1} \quad (5.8)$$

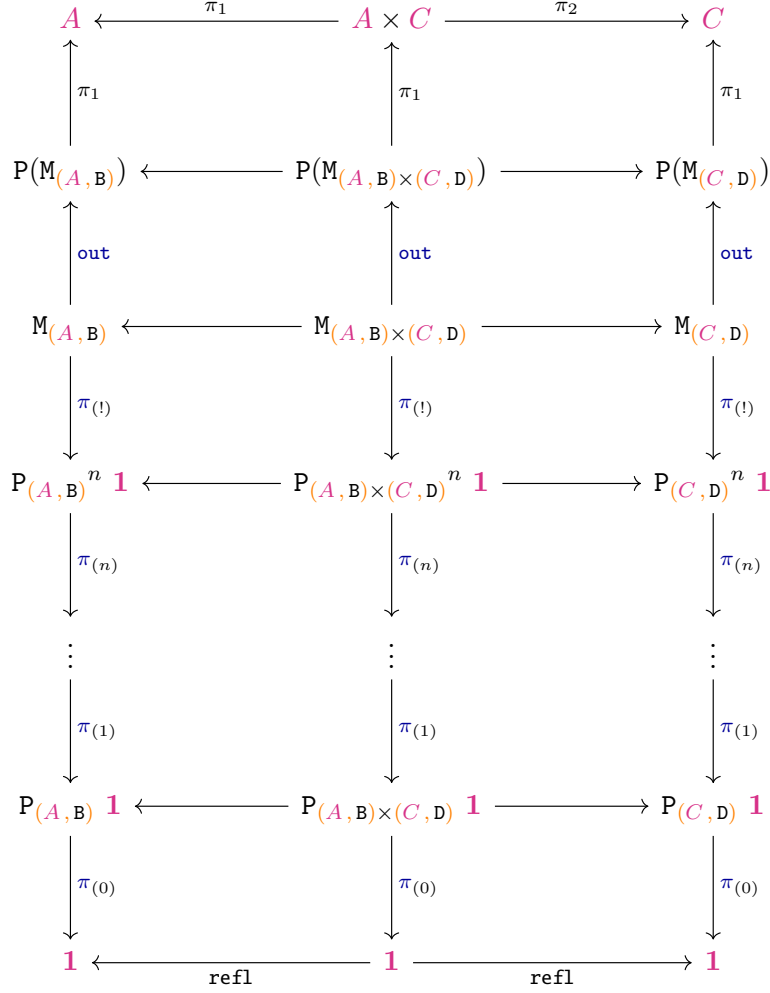


Figure 5.1: Diagram for products of chains

$$\equiv \sum_{a,c:A \times C} B \ a \times D \ c \rightarrow P_{(A,B) \times (C,D)}^m \mathbf{1} \quad (5.9)$$

$$\equiv P_{(A,B) \times (C,D)}(P_{(A,B) \times (C,D)}^m \mathbf{1}) \quad (5.10)$$

$$\equiv P_{(A,B) \times (C,D)}^{m+1} \mathbf{1} \quad (5.11)$$

taking the limit of (5.2) we get

$$M_{(A,B)} \times M_{(C,D)} \equiv M_{(A,B) \times (C,D)}. \quad (5.12)$$

□

Example 3. For streams we get

$$\text{stream } A \times \text{stream } B \equiv M_{(A, \lambda _ , \mathbf{1})} \times M_{(B, \lambda _ , \mathbf{1})} \equiv M_{(A, \lambda _ , \mathbf{1}) \times (B, \lambda _ , \mathbf{1})} \equiv \text{stream } (A \times B) \quad (5.13)$$

as expected. Transporting along (5.13) gives us a definition for **zip**.

5.1.2 Co-product

Coproducts?

5.1.3 ...

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

Chapter 6

TODO: M-types

6.1 TODO: Place these subsections

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

6.1.1 Identity Bisimulation

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

Lemma 6.1.1. *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

$$\begin{aligned} \alpha_{\equiv} : \equiv &\rightarrow \mathbf{P}(\equiv) \\ \alpha_{\equiv}(x, y) &:= \pi_1(\text{out } x), (\lambda b, (\pi_2(\text{out } x) b, \text{refl}_{(\pi_2(\text{out } x) b)})) \end{aligned} \tag{6.1}$$

and the two projections

$$\pi_1^{\equiv} = (\pi_1, \text{funExt } \lambda(a, b, r), \text{refl}_{\text{out } a}) \tag{6.2}$$

$$\pi_2^{\equiv} = (\pi_2, \text{funExt } \lambda(a, b, r), \text{cong}_{\text{out}}(r^{-1})). \tag{6.3}$$

This defines the bisimulation, given by the diagram in Figure 6.1. □

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

Figure 6.1: Identity bisimulation

6.1.2 Bisimulation of Streams

TODO

6.1.3 Bisimulation of Delay Monad

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

Definition 6.1.2. The relation \sim_{delay} is defined by the following rules

$$\frac{R : \mathcal{U} \quad r : R}{\text{now } r \sim_{\text{delay}} \text{now } r : \mathcal{U}} \text{now} \sim \quad (6.4)$$

$$\frac{R : \mathcal{U} \quad t : \text{delay } R \quad u : \text{delay } R \quad t \sim_{\text{delay}} u : \mathcal{U}}{\text{later } t \sim_{\text{delay}} \text{later } u : \mathcal{U}} \text{later} \sim \quad (6.5)$$

Theorem 6.1.3. The relation \sim_{delay} is a bisimulation for delay R .

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\text{delay}}} &: \overline{\sim_{\text{delay}}} \rightarrow \mathbf{P}(\overline{\sim_{\text{delay}}}) \\ \alpha_{\sim_{\text{delay}}}(a, b, \text{now} \sim r) &:= (\text{inr } r, \lambda ()) \\ \alpha_{\sim_{\text{delay}}}(a, b, \text{later} \sim x \ y \ q) &:= (\text{inl } \star, \lambda _, (x, y, q)) \end{aligned} \quad (6.6)$$

then we define the projections

$$\pi_1^{\overline{\sim_{\text{delay}}}} = \left(\pi_1, \text{funExt } \lambda (a, b, p), \begin{cases} (\text{inr } r, \lambda ()) & p = \text{now} \sim r \\ (\text{inl } \star, \lambda _, x) & p = \text{later} \sim x \ y \ q \end{cases} \right) \quad (6.7)$$

$$\pi_2^{\overline{\sim_{\text{delay}}}} = \left(\pi_2, \text{funExt } \lambda (a, b, p), \begin{cases} (\text{inr } r, \lambda ()) & p = \text{now} \sim r \\ (\text{inl } \star, \lambda _, y) & p = \text{later} \sim x \ y \ q \end{cases} \right) \quad (6.8)$$

$$(6.9)$$

This defines the bisimulation, given by the diagram in Figure 6.2. \square

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

Figure 6.2: Strong bisimulation for delay monad

6.1.4 Bisimulation of ITrees

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

$$\frac{a, b : R \quad a \equiv_R b}{\text{Ret } a \cong \text{Ret } b} \text{EqRet} \quad (6.10)$$

$$\frac{t, u : \text{itree } E \ R \quad t \cong u}{\text{Tau } t \cong \text{Tau } u} \text{EqTau} \quad (6.11)$$

$$\frac{A : \mathcal{U} \quad e : E \ A \quad k_1, k_2 : A \rightarrow \text{itree } E \ R \quad t \cong u}{\text{Vis } e \ k_1 \cong \text{Tau } e \ k_2} \text{EqVis} \quad (6.12)$$

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.

6.1.5 Zip Function

We want the diagram in Figure 6.3 to commute, meaning we get the computation rules

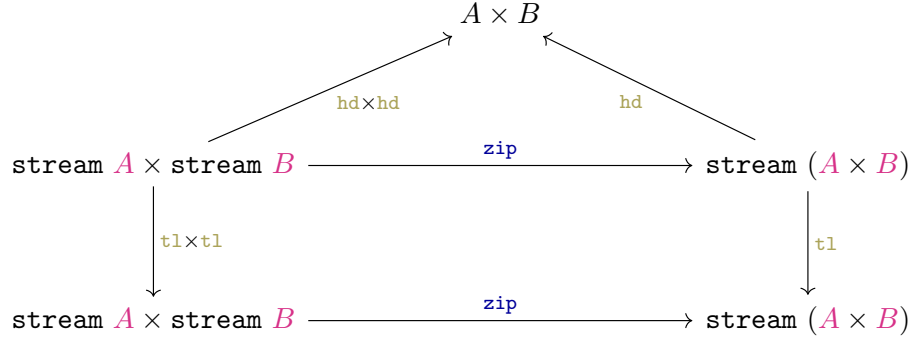


Figure 6.3: TODO

$$(\text{hd} \times \text{hd}) \equiv \text{hd} \circ \text{zip} \quad (6.13)$$

$$\text{zip} \circ (\text{tl} \times \text{tl}) \equiv \text{tl} \circ \text{zip} \quad (6.14)$$

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

$$\begin{aligned}
 \text{lift}_{\mathbf{M}} \left(x : \prod_{n:\mathbb{N}} (A \rightarrow \mathbf{P}_{\mathbf{S}}^n \mathbf{1}) \right) & \left(u : \prod_{n:\mathbb{N}} (A \rightarrow \pi_n(x_{n+1}a) \equiv x_n a) \right) (a : A) : \mathbf{M} \mathbf{S} := \\
 & (\lambda n, x \ n \ a), (\lambda n \ i, p \ n \ a \ i).
 \end{aligned} \quad (6.15)$$

To use this definition, we first define some helper functions

$$\text{zip}_X \ n \ (x, y) = \begin{cases} \mathbf{1} & \text{if } n = 0 \\ (\text{hd } x, \text{hd } y), (\lambda _, \text{zip}_X \ m \ (\text{tl } x, \text{tl } y)), & \text{if } n = m + 1 \end{cases} \quad (6.16)$$

$$\text{zip}_{\pi} \ n \ (x, y) = \begin{cases} \text{refl} & \text{if } n = 0 \\ \lambda i, (\text{hd } x, \text{hd } y), (\lambda _, \text{zip}_{\pi} \ m \ (\text{tl } x, \text{tl } y) \ i), & \text{if } n = m + 1 \end{cases}, \quad (6.17)$$

we can then define

$$\text{zip}_{\text{lift}} \ (x, y) := \text{lift}_{\mathbf{M}} \ \text{zip}_X \ \text{zip} \ (x, y). \quad (6.18)$$

6.1.5.1 Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\text{transport}_{?} \ a \equiv \text{zip}_{\text{lift}} \ a \quad (6.19)$$

$$\equiv \text{lift}_{\mathbf{M}} \ \text{zip}_X \ \text{zip}_{\pi} \ (x, y) \quad (6.20)$$

$$\equiv (\lambda n, \text{zip}_X \ n \ (x, y)), (\lambda n \ i, \text{zip}_{\pi} \ n \ (x, y) \ i) \quad (6.21)$$

zero case X

$$\mathbf{zip}_X 0 (x, y) \equiv \mathbf{1} \quad (6.22)$$

Successor case X

$$\mathbf{zip}_X (m + 1) (x, y) \equiv (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_X m (\mathbf{tl} x, \mathbf{tl} y)) \quad (6.23)$$

$$\equiv (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, ? (\mathbf{tl} a)) \quad (6.24)$$

$$\equiv (\mathbf{hd} (\mathbf{transport}_? a), (\lambda _, \mathbf{transport}_? (\mathbf{tl} a))) \quad (6.25)$$

$$\equiv \mathbf{transport}_? a \quad (6.26)$$

$$(6.27)$$

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

$$\equiv (), (\lambda i, \mathbf{zip}_\pi 0 (x, y) i) \quad (6.28)$$

$$\equiv \mathbf{1}, \mathbf{refl} \quad (6.29)$$

$$(6.30)$$

successor case

$$\equiv (\mathbf{zip}_X (m + 1) (x, y)), (\lambda i, \mathbf{zip}_\pi (m + 1) (x, y) i) \quad (6.31)$$

$$\equiv ((\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_X m (\mathbf{tl} x, \mathbf{tl} y))), (\lambda i, (\mathbf{hd} x, \mathbf{hd} y), (\lambda _, \mathbf{zip}_\pi m (\mathbf{tl} x, \mathbf{tl} y) i)) \quad (6.32)$$

Complete this proof

6.1.6 Examples of Fixed Points

6.1.6.1 Zeros

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

$$\mathbf{const}_X (n : \mathbb{N}) (c : \mathbb{N}) := \begin{cases} \mathbf{1} & n = 0 \\ (c, \lambda _, \mathbf{const}_X m c) & n = m + 1 \end{cases} \quad (6.33)$$

$$\mathbf{const}_\pi (n : \mathbb{N}) (c : \mathbb{N}) := \begin{cases} \mathbf{refl} & n = 0 \\ \lambda i, (c, \lambda _, \mathbf{const}_\pi m c i) & n = m + 1 \end{cases} \quad (6.34)$$

to get the definition of zero stream

$$\mathbf{zeros} := \mathbf{lift}_M \mathbf{const}_X \mathbf{const}_\pi 0. \quad (6.35)$$

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

$$\mathbf{hd} \mathbf{zeros} \equiv 0 \quad (6.36)$$

$$\mathbf{tl} \mathbf{zeros} \equiv \mathbf{zeros} \quad (6.37)$$

6.1.6.2 Spin

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

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

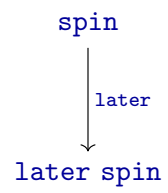


Figure 6.4: TODO

Chapter 7

Conclusion

conclude on the problem statement from the introduction

7.1 Future Work

We have not proof the equality between types defined as (coinductive) records and M-types.

All the work done here, should generalize to indexed M-types, which would be nice to have formalized.

Define the weak bisimilarity relations as M-types, since these are also coinductive types. However this seems to need something more general than "Index M-types"

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] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, pages 17–30, 2015.
- [3] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. *CoRR*, abs/1612.02346, 2016.
- [4] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29, 2016.
- [5] Jeremy Avigad, Mario M. Carneiro, and Simon Hudon. Data types as quotients of polynomial functors. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, pages 6:1–6:19, 2019.
- [6] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.
- [7] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.
- [8] Marcelo Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. *CoRR*, abs/1911.06899, 2019.
- [9] nLab authors. cubical type theory. <http://ncatlab.org/nlab/show/cubical%20type%20theory>, May 2020. Revision 15.
- [10] nLab authors. homotopy type theory. <http://ncatlab.org/nlab/show/homotopy%20type%20theory>, May 2020. Revision 111.
- [11] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.

- [12] 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.
- [13] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.

Appendix A

Additions to the Cubical Agda Library

Appendix B

The Technical Details

