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

Lasse Letager Hansen, 201912345

Master's Thesis, Computer Science
June 3, 2020
Advisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

We present a construction of `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 `M`-types, and how they can be quotiented to construct what we call `QM`-types. The problem with `QM`-types is that in general assuming that we can lift function types of equivalence classes is equivalent to the axiom of choice [6], 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 `QM`-type. We conclude with some examples of how to use `M`-types and some properties. All work is formalized in Cubical Agda, and the work on defining `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 3, 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 Q`M`-types 19
- 4.1 Quotienting and Constructors . . . 19
- 4.2 Q`M`-types and Quotient inductive-inductive types (QIITs) . . . 20
  - 4.2.1 Multiset . . . 20
  - 4.2.2 Partiality monad . . . 22
- 4.3 QM-types . . . 28
  - 4.3.1 Lifting Quotient Construction from Containers . . . 28

5 Properties of M-types? 31
- 5.1 Closure properties of `M`-types . . . 31
  - 5.1.1 Product of `M`-types . . . 31

5.1.2 Co-product . . . 33
5.1.3 33

6 TODO: `M`-types **35**
6.1 TODO: Place these subsections . . . 35
6.1.1 Identity Bisimulation . . . 35
6.1.2 Bisimulation of Streams . . . 35
6.1.3 Bisimulation of Delay Monad . . . 36
6.1.4 Bisimulation of ITrees . . . 36
6.1.5 Zip Function . . . 37
6.1.6 Examples of Fixed Points . . . 38

7 Conclusion **41**
7.1 Future Work . . . 41

Bibliography **43**

A Additions to the Cubical Agda Library **45**

B The Technical Details **47**

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 (Q`M`-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.

Should a description of Set Theory be included

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 Löf 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 algorithm that finds the element making the statement true. MLTT is built from the three finite types $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$, and type constructors Σ, π and $=$. There is only a single way to make terms of $=$-type, and that is $\texttt{refl} : \prod_{a:A}(a = a)$. .

inductive types, universes and judgements

extensional vs intensional https://ncatlab.org/nlab/show/intensional+type+theory

find citation, currently https://en.wikipedia.org/wiki/Intuitionistic_type_theory

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

$$\frac{}{0 : \mathbb{N}} \quad (1.1) \qquad\qquad \frac{n : \mathbb{N}}{\texttt{succ}\ n : \mathbb{N}} \quad (1.2)$$

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

$$\frac{}{0 = 0}\ \sim_0 \quad (1.3) \qquad\qquad \frac{n \sim_{\mathbb{N}} m}{\texttt{succ}\ n \sim_{\mathbb{N}} \texttt{succ}\ m}\ \sim_{\texttt{succ}} \quad (1.4)$$

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 ($\texttt{hd}$) and tail ($\texttt{tl}$), where head represents the first element, and tail represents the rest of the sequence. The inference rules are given as

$$\frac{s : \texttt{stream}\ A}{\texttt{hd}\ s : A} \quad (1.5) \qquad\qquad \frac{s : \texttt{stream}\ A}{\texttt{tl}\ s : \texttt{stream}\ A} \quad (1.6)$$

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

$$\frac{\texttt{hd}\ s = \texttt{hd}\ t \quad \texttt{tl}\ s \sim_{\texttt{stream}} \texttt{tl}\ t}{s \sim_{\texttt{stream}} t} \tag{1.7}$$

This relation does not give an equality in MLTT, we just get bisimilarity meaning elements "behave" the same, but they are not equivalent.

Inductive vs Coinductive definition for a set of constructors

To remedy this constraint, we will work in a type theory where the univalence axiom is 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

$$(A = B) \simeq (A \simeq B) \tag{1.8}$$

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)** [13]. HoTT is an intensional dependent type theory (built on MLTT) with the univalence axiom and higher inductive types. In HoTT the idenitity 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 [12]. 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

.. be working implementation of hott namely ctt

cubical type theory (CTT) [8], 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 [11]. 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.

example of what computational "axioms" mean

$$
\begin{tikzcd}
A \arrow[r, dashed, "p \cdot q \cdot r"] & B \\
C \arrow[u, "p^{-1}"] \arrow[r, "q"'] & D \arrow[u, "r"']
\end{tikzcd}
$$

Figure 1.1: Composition square

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 $\texttt{Delay}/{\sim_{\texttt{weak}}}$, 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 [13, Introduction].

...

cite something

...

final coalgebra is scary words, find something more "noob" friendly

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 a 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 contractiable types, meaning there is an element which all other element are equal to. Then there is (-1)-types which are mere propositions or 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 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 loose 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.

W-types, Induction , M-types, Coinduction...

describe quotienting and its usefullness by a couple examples

...

What is a monad?

...

add remaing chapters

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 languge Cubical Agda. A **proof assistant** helps with verifying proofs, while making the process of making proofs interactive. **Cubical Agda** [14] is an implementation of a cubical type theory (inspired by CCHM [9]) 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 $\mathtt{A} : \mathbb{I} \to \mathcal{U}$, and the endpoint `A i0` you get a line from `A i0` to `A i1`.

Axioms of cubical Agda

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

Universes??

Some better (intuitive) description of the interval type!!

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 $\mathtt{B} : A \to \mathcal{U}$
- A term $x : A$ or for constants $\mathtt{c} : A$
- A function $\mathtt{f} : A \to C$
- A constructor $\mathtt{f} : A \to C$

better description, not always a function

- A destructor $\mathtt{f} : A \to C$

better description, not always a function

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

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

Definition 1.3.1.

$$\rrangle\ x\ ,\ \mathtt{f}\ \llangle = \lambda\, n, \begin{cases} x & n = 0 \\ \mathtt{f}\ m & n = m + 1 \end{cases} \tag{1.9}$$

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) [4].

Some formalization of coalgebras is missing?

2.1 Containers / Signatures

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

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

$$\begin{aligned} &\mathtt{P}_S : \mathcal{U} \to \mathcal{U} \\ &\mathtt{P}_S(X) = \sum_{a:A} \mathtt{B}(a) \to X \end{aligned} \tag{2.1}$$

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

$$\begin{aligned} &\mathtt{P}_S\, \mathtt{f} : \mathtt{P}_S X \to \mathtt{P}_S Y \\ &\mathtt{P}_S\, \mathtt{f}\ (a, \mathtt{g}) = (a, \mathtt{f} \circ \mathtt{g}). \end{aligned} \tag{2.2}$$

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

$$\mathtt{P}_S(X) = \sum_{a:A} \mathbf{1} \to X. \tag{2.3}$$

Since we are working in a logic with exponentials, we get $\mathbf{1} \to X \equiv X^{\mathbf{1}} \equiv X$. Furthermore $\mathbf{1}$ and X does not depend on A, so (2.3) is equivalent to

$$\mathtt{P}_S(X) = A \times X. \tag{2.4}$$

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

Definition 2.1.3. A $\mathtt{P}_S$-coalgebra is defined as

$$\mathtt{Coalg}_S = \sum_{C:\mathcal{U}} C \to \mathtt{P}_S C. \tag{2.5}$$

We denote a $\mathtt{P}_S$-coalgebra given by C and γ as $C\text{-}\gamma$. Coalgebra morphisms are defined as

$$\begin{gathered}\cdot \Rightarrow \cdot : \mathtt{Coalg}_S \to \mathtt{Coalg}_S \\ C\text{-}\gamma \Rightarrow D\text{-}\delta = \sum_{\mathtt{f}:C\to D} \delta \circ \mathtt{f} = \mathtt{Pf} \circ \gamma\end{gathered} \tag{2.6}$$

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

Definition 2.1.4. Given a container S, we define $\mathtt{M}$-types as the type, making the coalgebra given by $\mathtt{M}_S$ and $\mathtt{out} : \mathtt{M}_S \to \mathtt{P}_S(\mathtt{M}_S)$ fulfill the property

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

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

Example 2. We define streams over the type A as the $\mathtt{M}$-type over the container $(A\,,\lambda\_,\ \mathbf{1})$. If we apply the polynomial functor to the $\mathtt{M}$-type, then we get $\mathtt{P}_{(A\,,\,\lambda\_,\,\mathbf{1})}\mathtt{M} = A \times \mathtt{M}_{(A\,,\,\lambda\_,\mathbf{1})}$, illustrated in Figure 2.1. We will show that $\mathtt{out}$ is an isomorphism with inverse $\mathtt{in} : \mathtt{P}_S(\mathtt{M}) \to \mathtt{M}$ later in this

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

Figure 2.1: $\mathtt{M}$-types of streams

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

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{hd}\ s : A}\ \mathtt{E}_{\mathtt{hd}} \tag{2.8}$$

$$\frac{A : \mathcal{U} \quad s : \mathtt{stream}\ A}{\mathtt{tl}\ s : \mathtt{stream}\ A}\ \mathtt{E}_{\mathtt{tl}} \tag{2.9}$$

$$\frac{A : \mathcal{U} \quad x : A \quad xs : \mathtt{stream}\ A}{\mathtt{cons}\ x\ xs : \mathtt{stream}\ A}\ \mathtt{I}_{\mathtt{cons}} \tag{2.10}$$

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

Definition 2.1.5. We define a chain as a family of morphisms $\pi_{(n)} : X_{n+1} \to X_n$, over a family of types X_n. See Figure 2.2.

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

Figure 2.2: Chain of types / functions

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

Figure 2.3: Cone

Lemma 2.1.6. *For all coalgebras C-γ for the container S, we get $C \to \mathtt{M}_S \equiv \mathtt{Cone}_{C\text{-}\gamma}$, where $\mathtt{Cone} = \sum_{(\mathtt{f}:\prod_{(n:\mathbb{N})} C \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{f}_{n+1} \equiv \mathtt{f}_n$ illustrated in Figure 2.3.*

Proof. We define an isomorphism from $C \to \mathtt{M}_S$ to $\mathtt{Cone}_{C\text{-}\gamma}$

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

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

$$\mathtt{rinv}_{collapse}\ (\mathtt{u}, \mathtt{q}) = \mathtt{refl}_{(\mathtt{u},\mathtt{q})} \tag{2.13}$$

$$\mathtt{linv}_{collapse}\ \mathtt{f} = \mathtt{refl}_{\mathtt{f}} \tag{2.14}$$

□

Lemma 2.1.7. *Given $\ell : \prod_{(n:\mathbb{N})}(X_n \to X_{n+1})$ and $y : \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} x_{n+1} \equiv \ell_n\ x_n$ the chain collapses as the equality $\mathcal{L} \equiv X_0$.*

Proof. We define this collapse by the isomorphism

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

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

$$\mathtt{rinv}_{\mathcal{L}collapse}\ x_0 = \mathtt{refl}_{x_0} \tag{2.17}$$

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

We can now define the construction of $\mathtt{in}$ and $\mathtt{out}$.

Theorem 2.1.8. *Given the container $(A\, , \mathtt{B})$ we define the equality*

$$shift : \mathcal{L} \equiv \mathtt{P}\mathcal{L} \tag{2.18}$$

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

$$\mathtt{in} = \mathtt{transport}\ shift \tag{2.19}$$

$$\mathtt{out} = \mathtt{transport}\ (shift^{-1}). \tag{2.20}$$

Proof. The proof is done using the two helper lemmas

$$\alpha : \mathcal{L}^{\mathtt{P}} \equiv \mathtt{P}\mathcal{L} \tag{2.21}$$

$$\mathcal{L}unique : \mathcal{L} \equiv \mathcal{L}^{\mathtt{P}} \tag{2.22}$$

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

$$\begin{aligned}
\mathtt{fun}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) &= \llparenthesis\ \star\ ,\ \mathtt{a}\ \rrparenthesis, \llparenthesis\ \mathtt{refl}_{\star}\ ,\ \mathtt{b}\ \rrparenthesis && (2.23)\\
\mathtt{inv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) &= \mathtt{a} \circ \mathtt{succ}\ ,\ \mathtt{b} \circ \mathtt{succ} && (2.24)\\
\mathtt{rinv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) &= \mathtt{refl}_{(\mathtt{a},\mathtt{b})} && (2.25)\\
\mathtt{linv}_{\mathcal{L}unique}\ (\mathtt{a}, \mathtt{b}) &= \mathtt{refl}_{(\mathtt{a},\mathtt{b})} && (2.26)
\end{aligned}$$

The definition of α is then,

$$\begin{aligned}
\mathcal{L}^{\mathtt{P}} &\equiv \sum_{(x:\prod_{(n:\mathbb{N})} \sum_{(a:A)} \mathtt{B}\, a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n+1)}\ x_{n+1} \equiv x_n && (2.27)\\
&\equiv \sum_{(x:\sum_{(a:\prod_{(n:\mathbb{N})} A)} \prod_{(n:\mathbb{N})} a_{n+1} \equiv a_n)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\, (\pi_1\, x)_n \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv_* \mathtt{u}_n && (2.28)\\
&\equiv \sum_{(a:A)} \sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} \mathtt{B}\, a \to X_n)} \prod_{(n:\mathbb{N})} \pi_{(n)} \circ \mathtt{u}_{n+1} \equiv \mathtt{u}_n && (2.29)\\
&\equiv \sum_{(a:A)} \mathtt{B}\, a \to \mathcal{L} && (2.30)\\
&\equiv \mathtt{P}\mathcal{L} && (2.31)
\end{aligned}$$

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

$$shift = \alpha^{-1} \cdot \mathcal{L}unique. \tag{2.32}$$

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

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

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

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

$$\begin{aligned}
&C\text{-}\gamma \Rightarrow \mathcal{L}\text{-}\mathtt{out} && (2.33)\\
&\equiv \sum_{(\mathtt{f}:C \to \mathcal{L})} (\mathtt{out} \circ \mathtt{f} \equiv \mathtt{Pf} \circ \gamma) && (2.34)\\
&\equiv \sum_{(\mathtt{f}:C \to \mathcal{L})} (\mathtt{in} \circ \mathtt{out} \circ \mathtt{f} \equiv \mathtt{in} \circ \mathtt{Pf} \circ \gamma) && (2.35)
\end{aligned}$$

$$\equiv \sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{f} \equiv \mathtt{in} \circ \mathtt{Pf} \circ \gamma) \tag{2.36}$$

$$\tag{2.37}$$

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

$$\sum_{(\mathtt{f}:C\to\mathcal{L})} (\mathtt{f} \equiv \psi\ \mathtt{f}) \quad \equiv \quad \sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{e}\ c \equiv \psi\ (\mathtt{e}\ c)) \tag{2.38}$$

if we define the function $\phi : \mathtt{Cone}_S \to \mathtt{Cone}_S$ as $\phi\ (\mathtt{u}, \mathtt{g}) = (\phi_0\ \mathtt{u}, \phi_1\ \mathtt{u}\ \mathtt{g})$ where

$$\phi_0\ \mathtt{u} = \rrbracket\ (\lambda\_, \star)\ ,\ \mathtt{Pf} \circ \gamma \circ \mathtt{u}\ \llbracket \tag{2.39}$$

$$\phi_1\ \mathtt{u}\ \mathtt{g} = \rrbracket\ \mathtt{funExt}\ \lambda\_, \mathtt{refl}_\star\ ,\ \mathtt{ap}\ (\mathtt{Pf} \circ \gamma) \circ \mathtt{g}\ \llbracket \tag{2.40}$$

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

$$\begin{tikzcd}
\mathtt{Cone}_{C\text{-}\gamma} \arrow[r, "e"] \arrow[d, "\phi"] & (C \to \mathcal{L}) \arrow[d, "\psi"] \\
\mathtt{Cone}_{C\text{-}\gamma} \arrow[r, "e"] & (C \to \mathcal{L})
\end{tikzcd}$$

Figure 2.4: commuting square

the simplification

$$\sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{e}\ \mathtt{c} \equiv \mathtt{e}\ (\phi\ \mathtt{c})) \tag{2.41}$$

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

$$\sum_{(c:\mathtt{Cone}_{C\text{-}\gamma})} (\mathtt{c} \equiv \phi\ \mathtt{c}) \tag{2.42}$$

$$\equiv \sum_{((\mathtt{u},\mathtt{g}):\mathtt{Cone}_{C\text{-}\gamma})} ((\mathtt{u}, \mathtt{g}) \equiv (\phi_0\ \mathtt{u}, \phi_1\ \mathtt{u}\ \mathtt{g})) \tag{2.43}$$

$$\equiv \sum_{((\mathtt{u},\mathtt{g}):\mathtt{Cone}_{C\text{-}\gamma})} \sum_{(p:\mathtt{u}\equiv\phi_0\ \mathtt{u})} \mathtt{g} \equiv_* \phi_1\ \mathtt{u}\ \mathtt{g} \tag{2.44}$$

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

$$\sum_{\left((\mathtt{u},p):\sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} C\to X_n)} \mathtt{u}\equiv\phi_0\ \mathtt{u}\right)} \sum_{(\mathtt{g}:\prod_{(n:\mathbb{N})} \pi_{(n)}\circ\mathtt{u}_{n+1}\equiv\mathtt{u}_n)} \mathtt{g} \equiv_* \phi_1\ \mathtt{u}\ \mathtt{g} \tag{2.45}$$

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

$$\mathbf{1} \equiv \left(\sum_{(\mathtt{u}:\prod_{(n:\mathbb{N})} C\to X_n)} \mathtt{u} \equiv \phi_0\ \mathtt{u} \right) \tag{2.46}$$

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

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

2.2 Coinduction Principle for M-types

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

Definition 2.2.1. For all coalgebras $C\text{-}\gamma : \mathtt{Coalg}_S$, given a relation $\mathcal{R} : C \to C \to \mathcal{U}$ and a type $\overline{\mathcal{R}} = \sum_{(a:C)} \sum_{(b:C)} a \ \mathcal{R} \ b$, such that $\overline{\mathcal{R}}$ and $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \to \mathtt{P}_S(\overline{\mathcal{R}})$ forms a P-coalgebra $\overline{\mathcal{R}}\text{-}\alpha_{\mathcal{R}} : \mathtt{Coalg}_S$, making the diagram in Figure 2.5 commute ($\Longrightarrow$ represents P-coalgebra morphisms).

What does commute mean here?

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

Figure 2.5: Bisimulation for a coalgebra

Theorem 2.2.2 (Coinduction principle)**.** *Given a relation $\mathcal{R}$, that is a bisimulation for an 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 $\mathtt{M}\text{-}\mathtt{out} : \mathtt{Coalg}_S$ 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)

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

Figure 2.6: Bisimulation principle for final coalgebra

what is !

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

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

What is the consequence of this?

$\square$

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.

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

3.1 Stream Formalization using `M`-types

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

complete this section

3.2 ITrees as `M`-types

Anything to add to this section?

Interaction trees (ITrees) [15] are used to model effectful behavior, where computations can interact with an external environment by events. ITrees are defined by the following constructors

$$\frac{r : R}{\texttt{Ret}\ r : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Ret}} \tag{3.1}$$

$$\frac{A : \mathcal{U} \quad a : \texttt{E}\ A \quad f : A \to \texttt{itree}\ \texttt{E}\ R}{\texttt{Vis}\ a\ f : \texttt{itree}\ \texttt{E}\ R}\ \texttt{I}_{\texttt{Vis}}. \tag{3.2}$$

$$\frac{t : \texttt{itree}\ \texttt{E}\ R}{\texttt{Tau}\ t : \texttt{itree}\ \texttt{E}\ R}\ \texttt{E}_{\texttt{Tau}}. \tag{3.3}$$

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

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

The polynomial functor for this container is

$$\mathtt{P}_S(X) = \sum_{(x:R+\mathbf{1})} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ \mathbf{1} & x = \mathtt{inr}\ \star \end{cases} \to X, \tag{3.5}$$

which we can simplify

$$\mathtt{P}_S(X) = R \times (\mathbf{0} \to X) + X. \tag{3.6}$$

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

$$\mathtt{P}_S(X) = X + R \tag{3.7}$$

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

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

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 $a : \mathtt{E}\,A$.

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

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

The polynomial functor for this container is

$$\mathtt{P}_S(X) = \sum_{(x:R+\sum_{(A:\mathcal{U})} \mathtt{E}\,A)} \begin{cases} \mathbf{0} & x = \mathtt{inl}\ r \\ A & x = \mathtt{inr}\ (A, e) \end{cases} \to X, \tag{3.9}$$

which simplifies to

$$\mathtt{P}_S(X) = (R \times (\mathbf{0} \to X)) + (\sum_{A:\mathcal{U}} \mathtt{E}\,A \times (A \to X)), \tag{3.10}$$

and further

$$\mathtt{P}_S(X) = R + \sum_{A:\mathcal{U}} \mathtt{E}\,A \times (A \to X). \tag{3.11}$$

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

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

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

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

The (reduced) polynomial functor for this container is

$$
\mathtt{P}_S(X) = R + X + \sum_{(A:\mathcal{U})} (\mathtt{E}\, A \times (A \to X)) \tag{3.13}
$$

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

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

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 $\delta : V \to \alpha \to V$. This gives us the diagram in Figure 3.4

Complete section about automaton

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

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

$$\frac{a : A}{\texttt{ret}\ a : T} \tag{3.14}$$

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

$$\frac{a : T}{\texttt{tl}\ a : T} \tag{3.15}$$

why give some arguments or illustration showing these reasonings

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

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

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

$$\frac{a : A \to T}{\texttt{node}\ a : T} \tag{3.17}$$

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

$$\frac{a : T}{\texttt{hd}\ a : A} \tag{3.18}$$

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 $(A, \mathtt{B})$ and $(C, \mathtt{D})$ for two constructors together is done by

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

Is this correct? Seems correct..

whereas adding containers for two destructors is done by

$$(A \times C, \lambda\_, \lambda\,(a, c),\ \mathtt{B}\,a + \mathtt{D}\,c) \tag{3.20}$$

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 $f_1 : F_1, f_2 : F_2, \ldots, f_n : F_n$, we can construct the M-types by the container

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

where each destructor $\mathrm{d_n} : T \to F_n$ for the field f_n will be defined as $\mathrm{d_n}\ t = \pi_n\ (\mathtt{out}\ t)$. However fields in a coinductive container may depend on previous defined fields, as given by the general list of fields $f_1 : F_1, f_2 : \mathtt{F}_2, \cdots, f_n : \mathtt{F}_n$, where each field depends on all the previous once, this can be defined by the container

$$\left(\sum_{(f_1 : F_1)} \sum_{(f_2 : \mathtt{F}_2)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \, , \lambda\, \_, \, \mathbf{0} \right) \tag{3.22}$$

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 $\mathtt{f}_2$ be the function with type $\mathtt{F}_2 = f_1 \to (f_1 \to A) \to \mathtt{M}$, which by currying is equal to $f_1 \times (f_1 \to A) \to \mathtt{M}$, we can then define by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \left(\mathbf{1} \times \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{n-1})} \mathtt{F}_n \right) , \lambda\, (f_1, \star, f_3, \dots), \mathtt{F}_2 \right) \tag{3.23}$$

where $\mathtt{F}_2$ 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 $\mathtt{F}_2 = (x : f_1) \to \mathtt{B}\, x \to \mathtt{M}$, but again by currying we can get $\mathtt{F}_2 : \sum_{(x : f_1)} \mathtt{B}\, x \to \mathtt{M}$ which is defined by the container

$$\left(\sum_{(f_1 : \mathcal{U})} \sum_{(f_3 : \mathtt{F}_3)} \cdots \sum_{(f_{n-1} : \mathtt{F}_{\mathtt{n}-1})} \mathtt{F}_{\mathtt{n}} \, , \lambda\, (f_1, f_3, \dots), \sum_{x : f_1} (\mathtt{B}\, x) \right) \tag{3.24}$$

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

which is the case, proof needed!

But we run into problems if ...

Problem cases

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.

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

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

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

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

$$\begin{tikzcd}
\mathbb{N} \arrow[r, "\mathtt{inl}(2n{+}1)"] \arrow[dr, "\mathtt{odd}"'] & M & \mathbb{N} \times M \arrow[l, "o.w."'] \arrow[dl, "\mathtt{step}"] \\
\mathbb{N} \arrow[r, "\mathtt{even}"'] \arrow[ur, "\mathtt{inr}(2n)"] & M \arrow[u] & (\mathbb{N} \to M) \arrow[l, "\mathtt{inl\text{-}zero}"] \arrow[ul, "\mathtt{inl}\ 0 \times f"']
\end{tikzcd}$$

Figure 3.5: wacky `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 M-type showing some of the use of 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 introduction to chapter, and reference 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

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

with recursor and eliminator defined as follows.

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

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

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

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

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

Definition 4.1.3 (Eliminator for quotients)**.** Given a statement $\texttt{P} : A/\texttt{R} \to \mathcal{U}$ which is a set $\texttt{P}_{\texttt{set}} : \prod_{(x:A/\texttt{R})} \texttt{isSet}\ (\texttt{P}\ x)$, a proof for the base case $\texttt{f} : \prod_{(a:A)} \texttt{P}\,[\,a\,]$ and a proof for equality $\texttt{f}_{\texttt{eq}} : \prod_{(a,b:A)} \prod_{(r:a\ \texttt{R}\ b)} \texttt{f}\ a \equiv_{*} \texttt{f}\ b$ over $\texttt{eq/}\ a\ b\ r$, we get a function $\texttt{elim} : (x : A/\texttt{R}) \to \texttt{P}\ x$ as follows

$$
\begin{aligned}
&\texttt{elim}\ [\,a\,] = \texttt{f}\ a \\
&\texttt{elim}\ (\texttt{eq/}\ a\ b\ r\ i) = \texttt{f}_{\texttt{eq}}\ a\ b\ r\ i \\
&\texttt{elim}\ (\texttt{squash/}\ a\ b\ p\ q\ i\ j) = \\
&\quad \texttt{isSet→isSetDep}\ \texttt{P}_{\texttt{set}}\ (\texttt{elim}\ a)\ (\texttt{elim}\ b)\ (\texttt{ap}\ \texttt{elim}\ p)\ (\texttt{ap}\ \texttt{elim}\ q)\ (\texttt{squash/}\ a\ b\ p\ q)\ i\ j
\end{aligned}
\tag{4.6}
$$

which is the eliminator for quotients.

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

preserves the H-level

4.2 QM-types and Quotient inductive-inductive types (QIITs)

we cannot always lift the constructors of M-type to the quotiented type

In this section we will show some examples of quotiented 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 belive that every QM-types is equal (under the axiom of choice) to a QITT, however QIITs are more general than our QM-types, so we cannot expect every QIIT to have a corresponding QM-type.

Go into more details!

4.2.1 Multiset

In this subsection we define infinite trees, where the order of subtrees does not matter also known as multisets [5][6][10].

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

$$
\left(A + \mathbf{1}, \begin{cases} \mathbf{0} & \texttt{inl}\ a \\ \mathbb{N} & \texttt{inr}\ \star \end{cases}\right)
\tag{4.7}
$$

For which we get the constructors

$$
\frac{a : A}{\texttt{leaf}\ a : \texttt{T}\ A} \quad (4.8) \qquad\qquad \frac{\texttt{f} : \mathbb{N} \to \texttt{T}\ A}{\texttt{node}\ \texttt{f} : \texttt{T}\ A} \quad (4.9)
$$

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

$$
\frac{\texttt{f} : \mathbb{N} \to \texttt{T}\ A \quad \texttt{g} : \mathbb{N} \to \mathbb{N} \quad \texttt{isIso}\ \texttt{g}}{\texttt{node}\ \texttt{f} \equiv \texttt{node}\ (\texttt{f} \circ \texttt{g})}\ \texttt{perm}
\tag{4.10}
$$

One way to define this type is as the QM-type given by quotienting $\texttt{T}\ A$ by the relation $\sim_{\texttt{T}}$ defined by the constructors

$$\frac{x \equiv y}{\texttt{leaf}\ x \sim_{\texttt{T}} \texttt{leaf}\ y} \sim_{\texttt{leaf}} \quad (4.11) \qquad\qquad \frac{\prod_{(n:\mathbb{N})} \texttt{f}\ n \sim_{\texttt{T}} \texttt{g}\ n}{\texttt{node}\ \texttt{f} \sim_{\texttt{T}} \texttt{node}\ \texttt{g}} \sim_{\texttt{node}} \quad (4.12)$$

$$\frac{\texttt{f} : \mathbb{N} \to \texttt{T}\,A \quad \texttt{g} : \mathbb{N} \to \mathbb{N} \quad \texttt{isIso}\ \texttt{g}}{\texttt{node}\ \texttt{f} \sim_{\texttt{T}} \texttt{node}\ (\texttt{f} \circ \texttt{g})} \sim_{\texttt{perm}} \tag{4.13}$$

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 denoted MS A, with the constructors leaf, node, perm and set quotiented MS-isSet. These two ways of constructing a type for multisets are equal assuming the axiom of (countable) choice.

Definition 4.2.2. There is a function from T A to MS A

$$\begin{aligned} \texttt{T→MS}\ (\texttt{leaf}_{\texttt{T}}\ x) &= \texttt{leaf}_{\texttt{MS}}\ x \\ \texttt{T→MS}\ (\texttt{node}_{\texttt{T}}\ \texttt{f}) &= \texttt{node}_{\texttt{MS}}\ (\texttt{T→MS} \circ \texttt{f}) \end{aligned} \tag{4.14}$$

This function takes weakly bisimilar objects to equal once.

Lemma 4.2.3. *If $x, y : \texttt{T}\,A$ are weakly bisimilar $p : x \sim_{\texttt{T}} y$ then $\texttt{T→MS}\ x \equiv \texttt{T→MS}\ y$.*

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

$$\begin{aligned} \texttt{T→MS-∼→≡}\ (\sim_{\texttt{leaf}}\ p) &= \texttt{ap}\ \texttt{leaf}_{\texttt{MS}}\ p \\ \texttt{T→MS-∼→≡}\ (\sim_{\texttt{node}}\ \texttt{k}) &= \texttt{ap}\ \texttt{node}_{\texttt{MS}}\ (\texttt{funExt}\ (\texttt{T→MS-∼→≡} \circ \texttt{k})) \\ \texttt{T→MS-∼→≡}\ (\sim_{\texttt{perm}}\ \texttt{f}\ \texttt{g}\ e) &= \texttt{perm}\ (\texttt{T→MS} \circ \texttt{f})\ \texttt{g}\ e \end{aligned} \tag{4.15}$$

□

With this lemma, we can lift the function T→MS to the quotient.

Definition 4.2.4. There is a function T/∼→MS from $\texttt{T}\,A/\sim_{\texttt{T}}$ to MS A, defined using the recursor for quotients defined in Definition 4.1.1, with $\texttt{f} = \texttt{T→MS}$ and $\texttt{g} = \texttt{T→MS-∼→≡}$ and MS is a set by MS-isSet

Lemma 4.2.5. *Given an equality $\texttt{T→MS}\ x \equiv \texttt{T→MS}\ y$ then $x \sim_{\texttt{T}} y$.*

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

Lemma 4.2.6. *The function T/∼→MS is injective, meaning $\texttt{T/∼→MS}\ x \equiv \texttt{T/∼→MS}\ y$ implies $x \equiv y$.*

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

$$\texttt{P} = (\lambda\, x,\ \texttt{T/∼→MS}\ x \equiv \texttt{T/∼→MS}\ y \to x \equiv y) \tag{4.16}$$

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

$$\mathtt{elimProp}\ (\lambda\, a,\ \mathtt{elimProp}(\lambda\, b,\ \mathtt{eq/}\ a\ b \circ \mathtt{T{\to}MS{-}{\equiv}{\to}{\sim}}\ a\ b)\ y)\ x \tag{4.17}$$

where $\mathtt{T{\to}MS{-}{\equiv}{\to}{\sim}}$ is Lemma 4.2.5. ☐

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

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

Theorem 4.2.8. *The is an equality between the types* $\mathtt{T}\ A/{\sim_{\mathtt{T}}}$ *and* $\mathtt{MS}\ A$, *assuming the axoim of choice.*

Proof. Since the function $\mathtt{T/{\sim}{\to}MS}$ is injective and surjective, it becomes an equality. ☐

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

$$\frac{}{R_\perp : \mathcal{U}} \tag{4.18}$$

$$\frac{}{\perp : R_\perp} \tag{4.19}$$

$$\frac{a : R}{\eta\ a : R_\perp} \tag{4.20}$$

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

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

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\bigsqcup\ (\mathtt{s}, \mathtt{b}) : R_\perp} \tag{4.21}$$

$$\frac{x, y : R_\perp \quad p : x \sqsubseteq_\perp y \quad q : y \sqsubseteq_\perp x}{\alpha_\perp\ p\ q : x \equiv y} \tag{4.22}$$

$$\frac{x : R_\perp}{x \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{refl}} \tag{4.23}$$

$$\frac{x \sqsubseteq_\perp y \quad y \sqsubseteq_\perp z}{x \sqsubseteq_\perp z}\ \sqsubseteq_{\mathtt{trans}} \tag{4.24}$$

$$\frac{x : R_\perp}{\perp \sqsubseteq_\perp x}\ \sqsubseteq_{\mathtt{never}} \tag{4.25}$$

$$\frac{\mathtt{s} : \mathbb{N} \to R_\perp \quad \mathtt{b} : \prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \mathtt{s}_{n+1}}{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp \bigsqcup(\mathtt{s}, \mathtt{b})} \tag{4.26}$$

$$\frac{\prod_{(n:\mathbb{N})} \mathtt{s}_n \sqsubseteq_\perp x}{\bigsqcup(\mathtt{s}, \mathtt{b}) \sqsubseteq_\perp x} \tag{4.27}$$

and finally set truncated

$$\frac{p, q : x \sqsubseteq_\perp y}{p \equiv q}\ (-)_\perp\mathtt{-isSet} \tag{4.28}$$

4.2.2.1 Delay monad to Sequences

introduction to sub-section

Definition 4.2.10. We define

$$\mathtt{Seq}_R = \sum_{(\mathtt{s}:\mathbb{N}\to R+\mathbf{1})} \mathtt{isMon}\ \mathtt{s} \tag{4.29}$$

where

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \equiv \mathtt{inr}\ \star) \times (\mathtt{s}_{n+1} \not\equiv \mathtt{inr}\ \star)) \tag{4.30}$$

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

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

$$\mathtt{isMon}\ \mathtt{s} = \prod_{(n:\mathbb{N})} (\mathtt{s}_n \equiv \mathtt{s}_{n+1}) + ((\mathtt{s}_n \uparrow_{R+\mathbf{1}}) \times (\mathtt{s}_{n+1} \downarrow_{R+\mathbf{1}})) \tag{4.31}$$

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

$$\mathtt{now}_{Seq}\ r = (\lambda\_, \mathtt{inl}\ r), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{4.32}$$

$$\mathtt{never}_{Seq} = (\lambda\_, \mathtt{inr}\ \star), (\lambda\_, \mathtt{inl}\ \mathtt{refl}) \tag{4.33}$$

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

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

$$\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ (z_s, z_q) = \begin{cases} z_s & n = 0 \\ \mathtt{s}_m & n = m + 1 \end{cases}, \begin{cases} z_q & n = 0 \\ \mathtt{q}_m & n = m + 1 \end{cases}, \tag{4.34}$$

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

$$\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{s} \circ \mathtt{succ}, \mathtt{q} \circ \mathtt{succ}. \tag{4.35}$$

Lemma 4.2.13. *The function*

$$\mathtt{shift\text{-}unshift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{shift}\ (\mathtt{unshift}\ (\mathtt{s}, \mathtt{q}))\ (\mathtt{s}_0, \mathtt{q}_0) \tag{4.36}$$

is equal to the identity function.

Proof. Unshifting a value followed by a shift, where we reintroduce the value we just remove, gives the sequence we started with. □

Lemma 4.2.14. *The function*

$$\mathtt{unshift\text{-}shift}\ (\mathtt{s}, \mathtt{q}) = \mathtt{unshift}\ (\mathtt{shift}\ (\mathtt{s}, \mathtt{q})\ \_) \tag{4.37}$$

is equal to the identity function.

Proof. If we shift followed by an unshift, we just introduce a value to instantly remove it, meaning the value does not matter. □

We now define an equivalence between `delay` R and $\texttt{Seq}_R$, 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≢inr)**.** *For any two elements* $x = \texttt{inl}\ a$ *and* $y = \texttt{inr}\ b$ *then* $x \not\equiv y$*.*

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

Definition 4.2.16. We define a function from `Delay` R to $\texttt{Seq}_R$

$$\begin{aligned} &\texttt{Delay→Seq}\ (\texttt{now}\ r) = \texttt{now}_{Seq}\ r \\ &\texttt{Delay→Seq}\ (\texttt{later}\ x) = \\ &\qquad \texttt{shift}\ (\texttt{Delay→Seq}\ x) \left(\texttt{inr}\ \star, \begin{cases} \texttt{inr}\ (\texttt{refl}, \texttt{inl≢inr}) & x = \texttt{now}\ \_ \\ \texttt{inl refl} & x = \texttt{later}\ \_ \end{cases}\right) \end{aligned} \tag{4.38}$$

Definition 4.2.17. We define function from $\texttt{Seq}_R$ to `Delay` R

$$\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q}) = \begin{cases} \texttt{now}\ r & \texttt{s}_0 = \texttt{inl}\ r \\ \texttt{later}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) & \texttt{s}_0 = \texttt{inr}\ \star \end{cases} \tag{4.39}$$

Theorem 4.2.18. *The type* $\texttt{Seq}_R$ *is equal to* `Delay` R

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

$$\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{s}, \texttt{q})) \equiv (\texttt{s}, \texttt{q}) \tag{4.40}$$

defined by cases analysis on $\texttt{s}_0$, if $\texttt{s}_0 = \texttt{inl}\ r$ then we need to show

$$\texttt{now}_{Seq}\ r \equiv (\texttt{s}, \texttt{q}) \tag{4.41}$$

This is true, since $(\texttt{s}, \texttt{q})$ is a monotone sequence and $\texttt{inl}\ r$ is the top element of the order, then all elements of the sequence are $\texttt{inl}\ r$. If $\texttt{s}_0 = \texttt{inr}\ \star$ then, we need to show

$$\texttt{shift}\ (\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))))\ (\texttt{inr}\ \star, \_) \equiv (\texttt{s}, \texttt{q}) \tag{4.42}$$

by the induction hypothesis we get

$$\texttt{Delay→Seq}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{s}, \texttt{q}))) \equiv \texttt{unshift}\ (\texttt{s}, \texttt{q}) \tag{4.43}$$

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

$$\texttt{Seq→Delay}\ (\texttt{Delay→Seq}\ t) \equiv t \tag{4.44}$$

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

$$\texttt{later}\ (\texttt{Seq→Delay}\ (\texttt{unshift}\ (\texttt{shift}\ (\texttt{Delay→Seq}\ x))) \equiv \texttt{later}\ x \tag{4.45}$$

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. □

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

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

and we show if $x \sim_{\texttt{Seq}} y$ then $\texttt{Seq}{\rightarrow}\texttt{Delay}\ x \sim_{\texttt{Seq}} \texttt{Seq}{\rightarrow}\texttt{Delay}\ y$, 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 $(\texttt{s}, \texttt{q}) : \texttt{Seq}_R$ terminates with a given value $r : R$,

$$(\texttt{s}, \texttt{q}) \downarrow_{\texttt{Seq}} r = \sum_{(n:\mathbb{N})} \texttt{s}_n \downarrow_{R+\mathbf{1}} r. \tag{4.46}$$

Definition 4.2.20 (Sequence Ordering)**.**

$$(\texttt{s}, \texttt{q}) \sqsubseteq_{\texttt{Seq}} (\texttt{t}, \texttt{p}) = \prod_{(a:R)} (\|\texttt{s} \downarrow_{\texttt{Seq}} a\| \rightarrow \|\texttt{t} \downarrow_{\texttt{Seq}} a\|) \tag{4.47}$$

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

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

$$\begin{aligned} \texttt{Maybe}{\rightarrow}(-)_\perp\ (\texttt{inl}\ r) &= \eta\ r \\ \texttt{Maybe}{\rightarrow}(-)_\perp\ (\texttt{inr}\ \star) &= \perp \end{aligned} \tag{4.48}$$

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

$$x \sqsubseteq_{\texttt{R+1}} y = (x \equiv y) + ((x \downarrow_{R+\mathbf{1}}) \times (y \uparrow_{R+\mathbf{1}})) \tag{4.49}$$

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

$$\texttt{isMon}\ \texttt{s} = \prod_{(n:\mathbb{N})} \texttt{s}_n \sqsubseteq_{\texttt{R+1}} \texttt{s}_{n+1} \tag{4.50}$$

This rewriting confirms that if isMon s, then s is monotone, and therefore a sequence of partial values. there exists non-monotone sequences, it just follows our definition of a sequence.

Lemma 4.2.23. *The function* $\texttt{Maybe}{\rightarrow}(-)_\perp$ *is monotone, that is, if* $x \sqsubseteq_{\texttt{A+1}} y$*, for some* x *and* y*, then* $(\texttt{Maybe}{\rightarrow}(-)_\perp\ x) \sqsubseteq_\perp (\texttt{Maybe}{\rightarrow}(-)_\perp\ y)$*.*

Proof. We do the proof by case.

$$\begin{aligned} &\texttt{Maybe}{\rightarrow}(-)_\perp\texttt{-mono}\ (\texttt{inl}\ p) = \\ &\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\rightarrow}(-)_\perp\ x \sqsubseteq_\perp \texttt{Maybe}{\rightarrow}(-)_\perp\ a)\ p\ (\sqsubseteq_{\texttt{refl}}\ (\texttt{Maybe}{\rightarrow}(-)_\perp\ x)) \\ &\texttt{Maybe}{\rightarrow}(-)_\perp\texttt{-mono}\ (\texttt{inr}\ (p, \_)) = \\ &\qquad \texttt{subst}\ (\lambda\, a,\ \texttt{Maybe}{\rightarrow}(-)_\perp\ a \sqsubseteq_\perp \texttt{Maybe}{\rightarrow}(-)_\perp\ y)\ p^{-1}\ (\sqsubseteq_{\texttt{never}}\ (\texttt{Maybe}{\rightarrow}(-)_\perp\ y)) \end{aligned} \tag{4.51}$$

□

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

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

$$\begin{aligned}&\mathtt{Seq{\to}incSeq}\\&\mathtt{Seq{\to}incSeq}\ (\mathtt{g},\mathtt{q}) = \mathtt{Maybe{\to}}(-)_\bot \circ \mathtt{g}, \mathtt{Maybe{\to}}(-)_\bot\mathtt{\text{-}mono} \circ \mathtt{q}\end{aligned} \tag{4.52}$$

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

$$\begin{aligned}&\mathtt{Seq{\to}}(-)_\bot : \mathtt{Seq}_A \to A_\bot\\&\mathtt{Seq{\to}}(-)_\bot\ (\mathtt{g},\mathtt{q}) = \bigsqcup \circ \mathtt{Seq{\to}incSeq}\end{aligned} \tag{4.53}$$

Lemma 4.2.26. *The function* $\mathtt{Seq{\to}}(-)_\bot$ *is monotone.*

$$\mathtt{Seq{\to}}(-)_\bot\mathtt{\text{-}mono} : \mathtt{isSet} A \to (x\ y : \mathtt{Seq}_A) \to x \sqsubseteq_{\mathtt{seq}} y \to \mathtt{Seq{\to}}(-)_\bot\ x \sqsubseteq_\bot \mathtt{Seq{\to}}(-)_\bot\ y \tag{4.54}$$

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

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

Lemma 4.2.27. *If two sequences* x, y *are weakly bisimular, then* $\mathtt{Seq{\to}}(-)_\bot\ x \equiv \mathtt{Seq{\to}}(-)_\bot\ y$

Proof.

describe instead of proof term!

$$\begin{aligned}&\mathtt{Seq{\to}}(-)_\bot\mathtt{\text{-}{\approx}{\to}{\equiv}}\ A_{set}\ x\ y\ (p, q) =\\&\qquad \alpha_\bot\ (\mathtt{Seq{\to}}(-)_\bot\mathtt{\text{-}mono}\ A_{set}\ x\ y\ p)\ (\mathtt{Seq{\to}}(-)_\bot\mathtt{\text{-}mono}\ A_{set}\ y\ x\ q)\end{aligned} \tag{4.55}$$

□

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

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

$$\mathtt{Seq/{\sim}{\to}}(-)_\bot = \mathtt{rec}\ \mathtt{Seq{\to}}(-)_\bot\ (\mathtt{Seq{\to}}(-)_\bot\mathtt{\text{-}{\approx}{\to}{\equiv}}\ A_{set})\ (-)_\bot\mathtt{\text{-}isSet} \tag{4.56}$$

Lemma 4.2.29. *Given two sequences* s *and* t*, if* $\mathtt{Seq{\to}}(-)_\bot\ s \equiv \mathtt{Seq{\to}}(-)_\bot\ t$*, then* $s \sim_{\mathtt{seq}} t$*.*

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

$$s \sim_{\mathtt{seq}} t = \left(\prod_{(r:R)} \|x \downarrow_{\mathtt{seq}} r\| \to \|y \downarrow_{\mathtt{seq}} r\|\right) \times \left(\prod_{(r:R)} \|y \downarrow_{\mathtt{seq}} r\| \to \|x \downarrow_{\mathtt{seq}} r\|\right) \tag{4.57}$$

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

Lemma 4.2.30. *The function* $\mathtt{Seq/{\sim}{\to}}(-)_\bot$ *is injective.*

Should this be formalized?

Convert to text, instead of a proof term!?

Proof. We use propositional elimination of quotients Definition 4.1.2 to show the injectivity, meaning for all $x\ y : \mathtt{Seq}_R/{\sim}_{\mathtt{seq}}$ we get $\mathtt{Seq/{\sim}{\to}(-)_\bot}\ x \equiv \mathtt{Seq/{\sim}{\to}(-)_\bot}\ y \to x \equiv y$. We start by eliminating x, followed by elimination of y, this gives us the proof term

$$
\begin{aligned}
&\mathtt{elimProp} \\
&\quad (\lambda\, a,\ \mathtt{Seq/{\sim}{\to}(-)_\bot}\ a \equiv \mathtt{Seq/{\sim}{\to}(-)_\bot}\ y \to a \equiv y) \\
&\quad (\lambda\, a,\ \mathtt{isProp\Pi}\ (\lambda\,\_, \mathtt{squash/}\ a\ y)) \\
&\quad (\lambda\, a,\ \mathtt{elimProp} \\
&\qquad\qquad (\lambda\, b,\ \mathtt{Seq{\to}(-)_\bot}\ a \equiv \mathtt{Seq/{\sim}{\to}(-)_\bot}\ b \to [\ ]a \equiv b) \\
&\qquad\qquad (\lambda\, b,\ \mathtt{isProp\Pi}\ (\lambda\,\_, \mathtt{squash/}\ [\,a\,]\ b)) \\
&\qquad\qquad (\lambda\, b,\ (\mathtt{eq/}\ a\ b) \circ (\mathtt{Seq{\to}(-)_\bot\text{-}isInjective}\ a\ b)) \\
&\qquad\qquad y) \\
&\quad x
\end{aligned}
\tag{4.58}
$$

where $\mathtt{Seq{\to}(-)_\bot\text{-}isInjective}$ is (4.2.29), □

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

Proof. The left side of the equality reduces to $\mathtt{Maybe{\to}(-)_\bot}$ 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 $\mathtt{Seq{\to}(-)_\bot}$ is surjective*

describe countable choice (and why it is needed!)

Proof. We do the proof by case on $R_\bot$, if it is $\eta\ r$ or $\mathtt{never}$, we convert them to the sequences $\mathtt{now}_{seq}\ r$ and $\mathtt{never}_{seq}$ respectively, then we are done by (4.2.31). For the least upper bound $\bigsqcup(\mathtt{s}, \mathtt{b})$, we translate to the (increasing) sequence, defined by $(\mathtt{s}, \mathtt{b})$. □

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

more precise description!

Lemma 4.2.33. *Assuming countable choice, the function $\mathtt{Seq/{\sim}{\to}(-)_\bot}$ is surjective*

Proof. □

Complete the rest of the proof!

Complete proof

Theorem 4.2.34. *Assuming countable choice, we get an equivalence between sequences and the partiality monad.*

Proof. The function $\mathtt{Seq/{\sim}{\to}(-)_\bot}$ 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?

4.3 QM-types

We want to define what a QM-type means in general, we draw inspiration from QW-types [10], quotient containers [3] and the LEAN paper "Data types as quotients of polynomial functors" [7]. From what we have seen as examples in the previous section, equality construction between Set Quotiented M-types and QIITs are given as

- A function `QM→QIIT` from QM to QIIT
- A proof `QM→QIIT-∼→≡` that $x \sim_{\texttt{QM}} y$ implies `QM→QIIT` $x \equiv$ `QM→QIIT` y
- Lifting `QM→QIIT` to `QM/∼→QIIT` using the quotient recursor with `QM→QIIT-∼→≡` and `QIIT-isSet`
- Showing injectivity by using propositional elimination of the quotient together with the inverse of `QM→QIIT-∼→≡`, namely `QM→QIIT-injective` saying that `QM→QIIT` $x \equiv$ `QM→QIIT` y implies $x \sim_{\texttt{QM}} y$.
- 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 $[\cdot]$.

Cofree Coalgebra / Dialgebra – Is this relevant?

4.3.1 Lifting Quotient Construction from Containers

"The category of containers lacks good coequalisers" - [1]

An alternative to directly set quotienting the M-types we have defined, is to do the quotienting on the underlying container [3] / polynomial functor [7], and then do the fixed point construction we did for polynomial functors, but on the quotiented functor instead. We start by defining quotients on containers.

Definition 4.3.1. Given a container $(A, \texttt{B})$, and a relation we can form a quotiented container $(A, \texttt{B})/\texttt{R}$

...

Which gives us the following definition of a polynomial functor.

Complete definition

Definition 4.3.2. Given a container $(A, \texttt{B})$ and for all X a family of relations $\sim_a : (\texttt{B}\, a \to X) \to (\texttt{B}\, a \to X) \to \mathcal{U}$ indexed by A, we can define a quotiented polynomial functor (QPF). We define it for types as

$$\texttt{F}\, X = \sum_{a:A} ((\texttt{B}\, a \to X)/\sim_a) \tag{4.59}$$

and for a function $\texttt{f} : X \to Y$, we use the quotient eliminator from Definition 4.1.3, with $\texttt{P} = \lambda\, \_, (\texttt{B}\, a \to Y)/\sim_a$ which is a set `squash/`, since we are using set truncated quotients. The base case $\texttt{f} = \lambda\, \_, [\, \texttt{f} \circ \texttt{g}\,]$ and equality case $\texttt{f}_{\texttt{eq}} = \lambda\, \texttt{x}\, \texttt{y}\, r,\ \texttt{eq/}\ (\texttt{f} \circ \texttt{x})\ (\texttt{f} \circ \texttt{y})\ (\sim_{\texttt{ap}}\ \texttt{f}\ r)$, where $\sim_{\texttt{ap}}$ says that given $\texttt{x} \sim_a \texttt{y}$ and a function $\texttt{f}$ then $\texttt{f} \circ \texttt{x} \sim_a \texttt{g} \circ \texttt{y}$. With this the definition for the quotient polynomial functor for functions is

Composition?

$$\texttt{F}\, \texttt{f}\ (a, \texttt{g}) = (a, \texttt{elim}\ \texttt{g}) \tag{4.60}$$

completing the definition of a quotiented polynomial functor.

$$\begin{tikzcd} \mathtt{P}\ X \arrow[r, "\mathtt{P}\ \mathtt{f}"] \arrow[d, "\mathtt{abs}_X"'] & \mathtt{P}\ Y \arrow[d, "\mathtt{abs}_Y"] \\ \mathtt{F}\ X \arrow[r, "\mathtt{F}\ \mathtt{f}"] & \mathtt{F}\ Y \end{tikzcd}$$

Figure 4.1: Quotiented polynomial function

If there is a function $\mathtt{abs}_X : \mathtt{P}\ X \rightarrow \mathtt{F}\ X$ that makes the diagram in Figure 4.1 commute (as in [7]), then we can construct the final $\mathtt{F}$-coalgebra, to get another notion of quotiented $\mathtt{M}$-type. If $\mathtt{abs}$ is surjective, then the square will commute, so we just have to define a relation that has the $\sim_{\mathtt{ap}}$ property, and a surjective function $\mathtt{abs}_X$ to construct a quotient $\mathtt{M}$-type, with a function $\mathtt{in}_{\mathtt{Q}} : \mathtt{M}_S \rightarrow \mathtt{M}_S/\sim$. As an example, lets try and construct the QPF and QM type for multisets, with this alternative approach

Example 3. We have the following polynomial functor for A-valued $\mathbb{N}$-branching trees

$$\mathtt{P}\ X = \sum_{a:A+\mathbf{1}} \begin{cases} \mathbf{0} \rightarrow X & a = \mathtt{inl}\ r \\ \mathbb{N} \rightarrow X & a = \mathtt{inr}\ \star \end{cases} \tag{4.61}$$

for which we define the family of relations $(\sim_{\mathrm{PT}})_{(a:A+\mathbf{1})}$, if $a = \mathtt{inl}\ r$ then the equality relations is just the trivial equalities, since the relation is between elements of type $\mathbf{0} \rightarrow X$, which is contractive. On the other hand if $a = \mathtt{inr}\ \star$, then we define the relation as

$$\frac{\mathtt{f}, \mathtt{h} : \mathbb{N} \rightarrow X \quad \mathtt{g} : \mathbb{N} \rightarrow \mathbb{N} \quad \mathtt{isIso}\ \mathtt{g} \quad \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}}{\mathtt{f} \sim_{\mathrm{PT}} \mathtt{h}} \tag{4.62}$$

we can see with this approach, we only need to define the non-trival equalities, meaning those that are not just reflexivity. This relations fulfills the $\sim_{\mathtt{ap}}$ property, since if $a = \mathtt{inl}\ r$ then it holds trivially, and if $a = \mathtt{inr}\ \star$ then we have $f \sim_{\mathrm{PT}} h$ and want to show $\mathtt{k} \circ \mathtt{f} \sim_{\mathrm{PT}} \mathtt{k} \circ \mathtt{h}$ for any function $\mathtt{k}$, which just boils down to showing $\mathtt{k} \circ \mathtt{f} \circ \mathtt{g} \equiv \mathtt{k} \circ \mathtt{h}$ given $p : \mathtt{f} \circ \mathtt{g} \equiv \mathtt{h}$, which is done by $\mathtt{ap}\ \mathtt{k}\ p$. We therefore have a QPF $\mathtt{F}$. Now we want to define $\mathtt{abs}$,

$$\begin{aligned} \mathtt{abs}\ (\mathtt{inl}\ r\ ,\ \lambda\,()) &= (\mathtt{inl}\ r\ ,\ [\lambda\,()]) \\ \mathtt{abs}\ (\mathtt{inr}\ \star\ ,\ \mathtt{f}) &= (\mathtt{inr}\ \star\ ,\ [\mathtt{f}]) \end{aligned} \tag{4.63}$$

and show that it is surjective, which follows directly from the surjectivity of $[\cdot]$. Taking the limit we get the square in Figure 4.2

$$\begin{tikzcd} \mathtt{T}\ A \arrow[r, "\mathtt{out}"] \arrow[d, "\mathtt{abs}_{\mathtt{T}\ A}"'] & \mathtt{P}\ (\mathtt{T}\ A) \arrow[d, "\mathtt{abs}_{\mathtt{P}\ (\mathtt{T}\ A)}"] \\ \mathtt{MS}\ A \arrow[r, "\mathtt{out}_{\mathtt{QM}}"] & \mathtt{F}\,(\mathtt{MS}\ A) \end{tikzcd}$$

Figure 4.2: QPF and limit diagram for multiset construction

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 [2]

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

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*

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

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

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

in the following

$$\begin{aligned}
& \mathtt{P}_{(A, \mathtt{B})}{}^{m+1}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{m+1}\ \mathbf{1} && (5.4)\\
\equiv\ & \mathtt{P}_{(A, \mathtt{B})}(\mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1}) \times \mathtt{P}_{(C, \mathtt{D})}(\mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1}) && (5.5)\\
\equiv\ & \sum_{a:A} \mathtt{B}\,a \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1} \times \sum_{c:C} \mathtt{D}\,c \to \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1} && (5.6)\\
\equiv\ & \sum_{a,c:A \times C} (\mathtt{B}\,a \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1}) \times (\mathtt{D}\,c \to \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1}) && (5.7)\\
\equiv\ & \sum_{a,c:A \times C} \mathtt{B}\,a \times \mathtt{D}\,c \to \mathtt{P}_{(A, \mathtt{B})}{}^{m}\ \mathbf{1} \times \mathtt{P}_{(C, \mathtt{D})}{}^{m}\ \mathbf{1} && (5.8)
\end{aligned}$$

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

Figure 5.1: Diagram for products of chains

$$\equiv \sum_{a,c:A\times C} \mathtt{B}\,a \times \mathtt{D}\,c \to \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m}\ \mathbf{1} \tag{5.9}$$

$$\equiv \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}(\mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m}\ \mathbf{1}) \tag{5.10}$$

$$\equiv \mathtt{P}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}{}^{m+1}\ \mathbf{1} \tag{5.11}$$

taking the limit of (5.2) we get

$$\mathtt{M}_{(A\,,\mathtt{B})} \times \mathtt{M}_{(C\,,\mathtt{D})} \equiv \mathtt{M}_{(A\,,\mathtt{B})\times(C\,,\mathtt{D})}. \tag{5.12}$$

□

Example 4. For streams we get

$$\mathtt{stream}\ A \times \mathtt{stream}\ B \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})} \times \mathtt{M}_{(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{M}_{(A\,,\lambda\ \_,\mathbf{1})\times(B\,,\lambda\ \_,\mathbf{1})} \equiv \mathtt{stream}\ (A \times B) \tag{5.13}$$

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" [2]

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

Proof. We first define the function

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

and the two projections

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

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

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

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

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 $\sim_{\texttt{delay}}$ for the delay monad,

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

$$\frac{R : U \quad r : R}{\texttt{now}\ r \sim_{\texttt{delay}} \texttt{now}\ r : \mathcal{U}}\ \texttt{now}\sim \tag{6.4}$$

$$\frac{R : \mathcal{U} \quad t : \texttt{delay}\ R \quad u : \texttt{delay}\ R \quad t \sim_{\texttt{delay}} u : \mathcal{U}}{\texttt{later}\ t \sim_{\texttt{delay}} \texttt{later}\ u : \mathcal{U}}\ \texttt{later}\sim \tag{6.5}$$

Theorem 6.1.3. *The relation $\sim_{\texttt{delay}}$ is a bisimulation for* `delay` R.

Proof. First we define the function

$$\begin{aligned} \alpha_{\sim_{\texttt{delay}}} &: \overline{\sim\mathit{delay}} \to \mathtt{P}(\overline{\sim\mathit{delay}}) \\ \alpha_{\sim_{\texttt{delay}}}\ (a, b, \texttt{now}\sim\ r) &:= (\texttt{inr}\ r, \lambda\,(\,)) \\ \alpha_{\sim_{\texttt{delay}}}\ (a, b, \texttt{later}\sim\ x\ y\ q) &:= (\texttt{inl}\ \star, \lambda\,\_, (x, y, q)) \end{aligned} \tag{6.6}$$

then we define the projections

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

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

$$\tag{6.9}$$

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

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

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.

$$\frac{a, b : R \quad a \equiv_R b}{\texttt{Ret}\ a \cong \texttt{Ret}\ b}\ \texttt{EqRet} \tag{6.10}$$

$$\frac{t, u : \texttt{itree}\ \texttt{E}\ R \quad t \cong u}{\texttt{Tau}\ t \cong \texttt{Tau}\ u}\ \texttt{EqTau} \tag{6.11}$$

$$\frac{A : \mathcal{U} \quad e : \texttt{E}\ A \quad k_1, k_2 : A \to \texttt{itree}\ \texttt{E}\ R \quad t \cong u}{\texttt{Vis}\ e\ k_1 \cong \texttt{Tau}\ e\ k_2}\ \texttt{EqVis} \tag{6.12}$$

Now we just need to define $\alpha_{\mathcal{R}}$

define the α_{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

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

Figure 6.3: TODO

$$(\texttt{hd} \times \texttt{hd}) \equiv \texttt{hd} \circ \texttt{zip} \tag{6.13}$$

$$\texttt{zip} \circ (\texttt{tl} \times \texttt{tl}) \equiv \texttt{tl} \circ \texttt{zip} \tag{6.14}$$

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 $\texttt{M}$-types

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

To use this definition, we first define some helper functions

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

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

we can then define

$$\texttt{zip}_{lift}\ (x, y) := \texttt{lift}_{\texttt{M}}\ \texttt{zip}_X\ \texttt{zip}\ (x, y). \tag{6.18}$$

6.1.5.1 Equality of Zip Definitions

We would expect that the two definitions for zip are equal

$$\begin{aligned}\texttt{transport}_?\ a &\equiv \texttt{zip}_{lift}\ a & (6.19)\\ &\equiv \texttt{lift}_{\texttt{M}}\ \texttt{zip}_X\ \texttt{zip}_\pi\ (x, y) & (6.20)\\ &\equiv (\lambda\, n, \texttt{zip}_X\ n\ (x, y)), (\lambda\, n\ i, \texttt{zip}_\pi\ n\ (x, y)\ i) & (6.21)\end{aligned}$$

zero case X

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

Successor case X

$$\begin{aligned}
\mathtt{zip}_X\ (m+1)\ (x, y) &\equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)) && (6.23)\\
&\equiv (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda_, ?\ (\mathtt{tl}\ a)) && (6.24)\\
&\equiv (\mathtt{hd}\ (\mathtt{transport}_? a)), (\lambda_, \mathtt{transport}_? (\mathtt{tl}\ a)) && (6.25)\\
&\equiv \mathtt{transport}_?\ a && (6.26)\\
& && (6.27)
\end{aligned}$$

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

$$\begin{aligned}
&\equiv (), (\lambda\, i, \mathtt{zip}_\pi\ 0\ (x, y)\ i) && (6.28)\\
&\equiv \mathbf{1}, \mathtt{refl} && (6.29)\\
& && (6.30)
\end{aligned}$$

successor case

$$\begin{aligned}
&\equiv (\mathtt{zip}_X\ (m+1)\ (x, y)), (\lambda\, i, \mathtt{zip}_\pi\ (m+1)\ (x, y)\ i) && (6.31)\\
&\equiv ((\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda_, \mathtt{zip}_X\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y))), (\lambda\, i, (\mathtt{hd}\ x, \mathtt{hd}\ y), (\lambda_, \mathtt{zip}_\pi\ m\ (\mathtt{tl}\ x, \mathtt{tl}\ y)\ i)) && (6.32)
\end{aligned}$$

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

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

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

to get the definition of zero stream

$$\mathtt{zeros} := \mathtt{lift_M}\ \mathtt{const_X}\ \mathtt{const}_\pi\ 0. \tag{6.35}$$

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

$$\mathtt{hd}\ \mathtt{zeros} \equiv 0 \tag{6.36}$$

$$\mathtt{tl}\ \mathtt{zeros} \equiv \mathtt{zeros} \tag{6.37}$$

6.1.6.2 Spin

We want to define spin, as being the fixed point `spin` = `later 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

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

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 (coinducitve) 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 coindcutve types. However this seems to need something more general than "Index M-types"

Bibliography

[1] Michael Gordon Abbott. *Categories of containers*. PhD thesis, University of Leicester, England, UK, 2003.

[2] 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.

[3] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, pages 2–15, 2004.

[4] 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.

[5] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. *CoRR*, abs/1612.02346, 2016.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] nLab authors. cubical type theory. `http://ncatlab.org/nlab/show/cubical%20type%20theory`, May 2020. Revision 15.

[12] nLab authors. homotopy type theory. `http://ncatlab.org/nlab/show/homotopy%20type%20theory`, May 2020. Revision 111.

[13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[14] 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.

[15] 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

...