Quotient inductive-inductive definitions

Gabe Dijkstra

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

April 2017

Abstract

In this thesis we present a theory of *quotient inductive-inductive definitions*, which are inductive-inductive definitions extended with constructors for equations. The resulting theory is an improvement over previous treatments of inductive-inductive and indexed inductive definitions in that it unifies and generalises these into a single framework. The framework can also be seen as a first approximation towards a theory of higher inductive types, but done in a set truncated setting.

We give the type of specifications of quotient inductive-inductive definitions mutually with its interpretation as categories of algebras. A categorical characterisation of the induction principle is given and is shown to coincide with the property of being an initial object in the categories of algebras. From the categorical characterisation of induction, we derive a more type theoretic induction principle for our quotient inductive-inductive definitions that looks like the usual induction principles.

The existence of initial objects in the categories of algebras associated to quotient inductive-inductive definitions is established for a class of definitions. This is done by a colimit construction that can be carried out in type theory itself in the presence of natural numbers, sum types and quotients or equivalently, coequalisers.

Acknowledgements

Front and foremost I would like to thank my supervisor, Thorsten Altenkirch, for taking me on as a student and guiding me through my PhD. I am grateful for his support and for teaching me about type theory and doing research. He has provided me with the freedom to follow my own ideas as well as providing necessary guidance.

Second of all, I would like to thank Natasha Alechina and Venanzio Capretta for looking at my work and making valuable suggestions during my first and second annual reviews respectively.

I also owe my colleagues Paolo Capriotti and Fredrik Nordvall Forsberg a lot, as this thesis would have not existed if not for the collaboration with them. I have learned a great deal from discussions at the whiteboard with them.

Furthermore, I would like to thank Nicolai Kraus for his thorough reading of this thesis, leading to fixes in proofs and readability improvements.

The reading groups with Paolo, Nicolai Kraus, Ambrus Kaposi and Manuel Bärenz have taught me everything I know about higher category theory. Without these meetings I would never have reached the level of understanding of the subject matter I have now.

I am much indebted to my examiners Bernhard Reus (external) and Venanzio Capretta (internal) for having spent a lot of time with this thesis. They have provided with valuable feedback that has improved the thesis considerably.

The other members of the Functional Programming Lab have also greatly contributed to making my PhD experience an enjoyable one: Graham, Henrik, Christian, Florent, Laurence, Bas, Nuo, Jenny, Iván, Jan, Jon and Jakob.

Everybody who visited me in Nottingham or received me with open arms when I visited them, deserves a very big thanks: Paul, Justin, Joshua, Robert, Wout, Edo, Ingo, Felix, Wout, Jonas, Koen, Marnix and countless of people I forgot to mention.

Last but not least, I would like to thank my parents and Rehma for their unconditional support.

Contents

1 Introduction **1**

1.1 Induction in mathematics 1
1.2 Induction in computer science 2
1.3 Formal treatment of induction 5
1.3.1 In type theory 6
1.3.2 In category theory 6
1.4 Higher inductive types and homotopy type theory 7
1.5 A theory of quotient inductive-inductive definitions 11
1.6 Related work 12
1.7 Overview of the thesis and contributions 13
1.7.1 List of main contributions 14
1.7.2 Declaration of authorship and previous work 15

2 Preliminaries **17**

2.1 Basic type formers 17
2.1.1 Universes 19
2.1.2 Implicit arguments 20
2.1.3 Inductive data types 20
2.2 Equality 23
2.2.1 Dependent equality 25
2.2.2 Functoriality of functions 25
2.2.3 Truncation levels 27
2.2.4 Equivalence 29
2.2.5 Univalence 31

2.2.6 Function extensionality . . . 32
2.2.7 Equivalences of Σ-types . . . 33
2.2.8 Alternative formulation of identity types . . . 34
2.3 Category theory in type theory . . . 37
2.3.1 Higher categories . . . 40
2.4 Core type theory . . . 40

3 Quotient inductive-inductive definitions 43
3.1 Examples . . . 43
3.1.1 Interval type . . . 43
3.1.2 Quotients and colimits . . . 47
3.1.3 Propositional truncation . . . 49
3.1.4 Infinitely branching trees . . . 50
3.1.5 Cauchy reals . . . 54
3.1.6 Syntax of type theory . . . 55
3.2 Implementation . . . 57
3.2.1 Cubical type theory . . . 60
3.3 Related work . . . 60

4 Describing inductive definitions 63
4.1 Algebraic semantics . . . 66
4.1.1 Monad algebras . . . 68
4.2 $\mathcal{S}et$-sorted inductive-inductive definitions . . . 71
4.2.1 Avoiding induction-recursion . . . 74
4.3 Dependent sorts . . . 76
4.3.1 Sort membership . . . 78
4.3.2 Makkai's dependent sorts . . . 79
4.3.3 Sort categories via comma categories . . . 81
4.4 Categories of algebras . . . 81
4.4.1 A $\mathcal{R}el$-sorted quotient inductive-inductive type . . . 82
4.4.2 Specification of a quotient inductive-inductive definition . . . 86
4.4.3 Point constructors . . . 88

4.4.4 Path constructors . . . 92
4.4.5 Worked example . . . 94
4.5 Other forms of constructors . . . 96
4.5.1 Dependent dialgebras . . . 98
4.5.2 Currying . . . 99
4.6 Positivity . . . 100
4.7 Related work . . . 101
4.7.1 Inductive-inductive definitions . . . 101
4.7.2 Inductive definitions in Agda . . . 101
4.7.3 Higher inductive types . . . 103

5 Induction versus initiality **105**
5.1 Categorical characterisation of induction . . . 106
5.2 The section principle is logically equivalent to initiality . . . 108
5.3 Limits in categories of algebras . . . 111
5.3.1 Sort categories . . . 112
5.3.2 Categories of algebras . . . 114
5.4 Deriving the induction principle . . . 122
5.4.1 Induction for F-algebras . . . 123
5.4.2 General framework . . . 125
5.4.3 Induction for quotient inductive-inductive definitions 129
5.4.4 Putting it all together . . . 142
5.5 Related work . . . 143

6 Constructing quotient inductive-inductive definitions **145**
6.1 Strict positivity . . . 146
6.2 Initial objects in sort categories . . . 146
6.3 Initial objects via sequential colimits . . . 148
6.3.1 Internal sequential colimits . . . 151
6.3.2 Constructing $\mathcal{S}et$-sorted quotient inductive-inductive definitions . . . 153
6.3.3 Putting it all together . . . 164
6.4 Related work . . . 165

7 Concluding remarks **167**
7.1 Future work . . . 171
7.1.1 Metaprogramming and generic programming . . . 171
7.1.2 Invariance of descriptions under equivalence of constructors . . . 171
7.1.3 Generalised containers . . . 171
7.1.4 Constructing initial algebras . . . 172
7.1.5 Generalising to higher inductive types . . . 172

A Containers for quotient inductive-inductive definitions **173**
A.1 Containers for $\mathcal{S}et$-sorted definitions . . . 174
A.2 Containers for arbitrarily sorted definitions . . . 175
A.3 Limitations of containers . . . 176

B Moving to an untruncated setting **177**
B.1 Coherence laws for functors . . . 179
B.1.1 Generalised containers . . . 180
B.2 Sort categories . . . 181
B.3 Categories of dialgebras . . . 181
B.3.1 Identity morphisms . . . 181
B.3.2 Composition . . . 182
B.3.3 Category laws . . . 183
B.4 Untruncated $\mathcal{T}ype$-sorted inductive-inductive definitions . . . 185
B.5 Path constructors and their computation rules . . . 187

Chapter 1

Introduction

In this thesis we set out to develop a theory of *quotient inductive-inductive definitions*, which are inductive-inductive definitions [Nor13] extended with path constructors. In this first chapter we will give some context of the problem and discuss prior art and related concepts. The chapter is concluded by an overview of the thesis and a list of contributions.

1.1 Induction in mathematics

In mathematics, induction is an important proof technique. The most common and perhaps oldest form of induction is induction on the natural numbers, dating back to at least Plato [Ace00]. Induction on the natural numbers gives us a way to prove that a formula $\phi(n)$ holds for any $n \in \mathbb{N}$: we have to prove $\phi(0)$ and prove that for any $n \in \mathbb{N}$, $\phi(n)$ implies $\phi(n+1)$. That the natural numbers satisfy this property can be seen as one of the defining properties of the natural numbers. This was first written down formally by Guiseppe Peano [Pea89]. He defined the natural numbers to be a set $\mathbb{N}$ with properties such as:

- $0 \in \mathbb{N}$
- for any $n \in \mathbb{N}$, $\mathsf{succ}(n) \in \mathbb{N}$.
- $\mathbb{N}$ satisfies the induction principle

The remaining axioms describe the equality relation on the natural numbers and postulate the injectivity of the succ function symbol and that zero $\neq$ succ n for all $n \in \mathbb{N}$.

A consequence of the induction principle for natural numbers is that we can define functions $\mathbb{N} \to X$, for some set X, *recursively*: it suffices to define $f\ 0$ and $f\ (\mathsf{succ}(n))$, where we may refer to $f\ n$. We can use this to define addition $n + m$ on the natural numbers by recursion on the second argument m, i.e. we define $n + 0 :\equiv n$ and $n + \mathsf{succ}(m) :\equiv \mathsf{succ}(n + m)$.

In mathematics, the construction of new sets is often done by taking the natural numbers as a given and building upon this and quotienting where needed, as opposed to giving an inductive definition directly. For example, the rational numbers can be constructed as $\mathbb{N} \times \mathbb{N}$ quotiented by the relation $(a, b) \sim (c, d)$ if and only $ad = bc$. Quotienting infinite sets is not always unproblematic. The usual construction of the real numbers as a quotient of Cauchy sequences of rational numbers requires the axiom of choice to show that it forms a complete metric space. Direct inductive definitions (with equations) can avoid such problems, as we will see in section 3.1.5.

1.2 Induction in computer science

Recursion is a central concept to computer science. Data structures are often defined in terms of themselves: for example, a binary tree is either a leaf or a pair of binary trees. Functional programming languages therefore often come with a mechanism to express definitions such as these, usually under the name of *algebraic data types*.

In Haskell [Jon03], one can define (linked) lists as the algebraic data type:

$$\mathbf{data}\ \mathsf{List}\ a = \mathsf{nil} \mid \mathsf{cons}\ a\ (\mathsf{List}\ a)$$

As opposed to having a recursion principle associated with the algebraic data type, we have *pattern matching* and general recursion. For example,

we can define a function as follows:

$$\begin{aligned}
&\mathsf{map} : (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \\
&\mathsf{map}\ f\ \mathsf{nil} = \mathsf{nil} \\
&\mathsf{map}\ f\ (\mathsf{cons}\ x\ xs) = \mathsf{cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)
\end{aligned}$$

Pattern matching and general recursion are powerful enough to let us implement the recursion operator associated with the inductive type. In the case of lists, this is usually called foldr:

$$\begin{aligned}
&\mathsf{foldr} : b \to (a \to b \to b) \to \mathsf{List}\ a \to b \\
&\mathsf{foldr}\ e\ op\ \mathsf{nil} = e \\
&\mathsf{foldr}\ e\ op\ (\mathsf{cons}\ x\ xs) = op\ x\ (\mathsf{foldr}\ e\ op\ xs)
\end{aligned}$$

If we care about the totality of our definitions, pattern matching and general recursion are too powerful. First of all we have to restrict ourselves to *structurally* recursive definitions: recursive calls may only be done on subterms of the patterns on the left hand side of the pattern matching clause. This is however not enough to ensure termination of definitions. The inductive types themselves also have to be of the right shape: they have to be *strictly positive*. If we were to have an inductive type:

$$\begin{aligned}
&\mathbf{data}\ T : \mathsf{Type}\ \mathbf{where} \\
&\quad a : (T \to T) \to T
\end{aligned}$$

then we could define:

$$\begin{aligned}
&\mathsf{oh} : T \\
&\mathsf{oh} :\equiv a\ (\lambda x.x) \\
&\mathsf{uh} : T \to 0 \\
&\mathsf{uh}\ (a\ f) :\equiv \mathsf{uh}\ (f\ \mathsf{oh})
\end{aligned}$$

where 0 denotes the empty type.

Since f oh is structurally smaller than a f, the definition of uh is structurally recursive. However, it does give us a term that does not have a normal form, namely uh oh.

Algebraic data types allow us to specify types or a family of types parametrised by type variables. As these are *parameters*, we are not allowed to vary them in the result type of the constructors. Lifting this restriction, i.e. turning the parameters into *indices*, gives us *generalised algebraic data types* (GADTs) or *inductive families*. We can use the indices to store extra information in the type, allowing us to encode invariants. They have been used to implement a type of well-typed abstract syntax trees [PL04] and red-black trees [Kah01], among many other uses.

Inductive families are especially powerful in a dependently typed setting in conjunction with *dependent pattern matching* [Coq92], such as it is implemented in Agda[1] [Nor07]. The information encoded in the indices may tell us that certain cases are impossible and need not be treated, or they may tell us that certain variables in the patterns are equal. We can define the type of length-indexed lists, also referred to as vectors as follows:

```
data Vec (A : Type) : ℕ → Type where
  nil : Vec A zero
  cons : A → (n : ℕ) (xs : Vec A n) → Vec A (succ n)
```

We can define a function that returns the first element of a non-empty list as follows:

```
head : (A : Type) (n : ℕ) → Vec A (succ n) → A
head A n (cons x .n xs) = x
```

By pattern matching on the argument of type Vec A (succ n), we get two cases: the value is either constructed with constructor nil or cons. If we unify the type of the argument with that of the constructors, we see that the argument can never be nil, as that produces something of type Vec A zero.

[1] `http://wiki.portal.chalmers.se/agda/pmwiki.php`

Furthermore, if we consider the cons case, we notice that the natural number argument of the constructor has to coincide with the one we already had in our context, hence we get the non-linear pattern head $A\, n\, (\mathsf{cons}\, x\, .n\, xs)$, where the dot indicates that it is a repeated variable.

It has been shown that under certain assumptions, i.e. if the data types are strictly positive and the recursion is structural, along with certain assumptions about the equality in the type theory, then these dependent pattern matching definitions can be translated to ones using only elimination principles [GMM06]. The restrictions on the type of equality used can be relaxed in certain cases [CDP14], giving us a form of dependent pattern matching which is compatible with homotopy type theory [Uni13].

In Agda, apart from inductive families, we can also define inductive types mutually with other inductive types/families or functions, giving us inductive-inductive [Nor13] and inductive-recursive definitions [DS99] respectively.

1.3 Formal treatment of induction

So far we have given several examples of inductive sets and types and recursive definitions, but have not given a formal definition of what an inductive definition is. In [Acz77], the author writes:

> "Inductive definitions of sets are often informally presented by giving some rules for generating elements of the set and then adding that an object is to be in the set only if it has been generated according to the rules."

In [Mar71], the author gives a scheme of the kind of rules that comprise inductive definitions in first-order logic. In this thesis we are concerned with inductive definitions in Martin-Löf Type Theory [Mar72], but we will also look at category theoretic characterisations in section 1.3.2, to guide us to appropriate generalisations.

1.3.1 In type theory

Extending a type theory with a particular inductive definition means that we have to extend the theory with four sets of inference rules:

- *type formation rules* ($\mathbb{N}$ is a type)
- *introduction rules* ($\mathsf{zero} : \mathbb{N}$, $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$)
- *elimination rules* (given $P : \mathbb{N} \to \mathsf{Type}$, $m_{\mathsf{zero}} : P\ \mathsf{zero}$, $m_{\mathsf{succ}} : (n : \mathbb{N}) \to P\ n \to P\ (\mathsf{succ}\ n)$, we get $\mathbb{N}\text{-}\mathsf{ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{succ}} : (x : \mathbb{N}) \to P\ x$)
- *computation rules* ($\mathbb{N}\text{-}\mathsf{ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{succ}}\ \mathsf{zero} = m_{\mathsf{zero}}$, $\mathbb{N}\text{-}\mathsf{ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{succ}}\ (\mathsf{succ}\ n) = m_{\mathsf{succ}}\ n\ (\mathbb{N}\text{-}\mathsf{ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{succ}} n)$)

In the case of the natural numbers, the type formation, introduction and elimination rules are not essentially different from Peano's rules. Missing however are rules defining an equality relation on $\mathbb{N}$. As we will see, we can define the equality relation as a single parametric inductive definition uniformly for all types. Given this notion of equality and given that we have a universe of types available, one can derive that the constructors are disjoint and injective.

The declaration of an inductive definition involves giving rules in all these classes. However, as observed by Backhouse et al. [Bac+89], it is enough to give the type formation rules and introduction rules: the elimination principle along with its computation rules can be derived from them. This fact is also reflected in how one declares inductive definitions in type theory-based proof assistants such as Coq [BC04] and Agda by simply giving a sequence of constructors, i.e. introduction rules.

1.3.2 In category theory

Inductive definitions can be characterised in category theory as *initial F-algebras*, for some endofunctor F on an appropriately chosen category. For example, the set of natural numbers with its operations zero and succ form an initial algebra for the functor $FX :\equiv 1 + X$. The property of being an

initial algebra contains the same information as given by the four classes of rules, for the appropriately chosen endofunctor.

The perspective on inductive definitions as initial algebras allows us to generalise easily. If we associate ordinary inductive types with initial algebras of endofunctors on a category $\mathcal{C}$, which is a model of type theory, e.g. Set, then inductive families correspond to initial algebras of endofunctors on slice categories of $\mathcal{C}$.

Another way to generalise is based on the observation that F-algebras for an endofunctor F coincide with F^*-monad algebras, where F^* is the free monad of F. (Note that F^* may not exist: F needs to be a strictly positive functor.) As is described in [Shu11b], we can interpret this as ordinary inductive types being associated with free monads. Generalising these inductive types would be the same as considering a larger class of monads.

As monads and monad algebras are also used to talk about algebraic theories, such as the theory of groups, the aforementioned observation makes clear the relationship between inductive definitions and algebraic theories. An essential ingredient of algebraic theories is the ability to talk about equations. This is something which is lacking in the inductive definitions we have seen so far.

1.4 Higher inductive types and homotopy type theory

Higher inductive types are a generalisation of inductive types, stemming from homotopy type theory, where apart from the usual constructors, called *point constructors*, we may also have equations as constructors, called *path constructors*.

Before we go on and give some examples, we will give a brief recap of homotopy type theory. For a more in depth introduction, we refer the reader to the usual book [Uni13].

In type theory, we can define an equality relation on any type induc-

tively, as follows: suppose A is a type, we define:

$$\begin{aligned}&\textbf{data}\ \_ =_A \_ : A \to A \to \mathsf{Type}\ \textbf{where}\\ &\quad \mathsf{refl} : (x : A) \to x =_A x\end{aligned}$$

If we have a term $\mathsf{refl}\ x : x =_A y$ for some $x, y : A$ then this only type checks if x is *definitionally* equal to y, i.e. they are the same up to β- and η-equality (and possibly more equalities).

The equality defined above is usually referred to as *propositional* equality. This name comes from the idea that equality is propositional, i.e. any two terms of that type are equal. If we normalise a closed term of an identity type $x = y$, it normalises to refl, hence it gives us a definitional equality $x \equiv y$. As such, one would expect that if we have two terms $p, q : x = y$, then also $p = q$, i.e. we have *uniqueness of identity proofs*. Using dependent pattern matching this is easy to prove:

$$\begin{aligned}&\mathsf{uip} : (A : \mathsf{Type})\ (x\ y : A)\ (p\ q : x = y) \to p = q\\ &\mathsf{uip}\ A\ x\ .x\ (\mathsf{refl}\ x)\ (\mathsf{refl}\ x) :\equiv \mathsf{refl}\ (\mathsf{refl}\ x)\end{aligned}$$

However, if we consider the induction principle for this type, called the J rule, this is not so clear at all. What we can show with J is that given a type A, the relation $=_A$ is an equivalence relation. Furthermore we can show that it forms a groupoid with transitivity as its binary operation, reflexivity as its unit and symmetry as its inverse operation. Using J, we can show that these operations all satisfy the groupoid laws up to propositional equality again, but not definitionally. Using this idea of types as groupoids, a model of type theory has been given in the category of groupoids [HS98]. Since there are non-trivial groupoids, the groupoid model contains types which refute the uniqueness of identity types principle.

The story does not end with groupoids, however. Since the groupoid laws are satisfied up to propositional equality, a type itself again, we get a tower of groupoids: we get ∞-groupoids. ∞-groupoids are also the object of study in homotopy theory: they can be thought of as topological spaces

up to homotopy. Types can therefore be thought of as ∞-groupoids [VG11; Lum09] and therefore also as topological spaces up to homotopy. This correspondence leads to a geometric intuition for type theory: types can be seen as spaces with their identity types as their path spaces.

One important axiom that is considered in homotopy type theory, is the univalence axiom, which roughly tells us that isomorphic types are also propositionally equal. This axiom is inspired by and holds in the simplicial set model of type theory [KLV12]. A univalent universe is then one example of a type that does not satisfy uniqueness of identity proofs: we may have different isomorphisms, giving rise to different propositional equalities between types. An example of this are the identity map and the negation map on the booleans: these are two distinct isomorphisms, yielding by univalence two distinct propositional equalities $\mathsf{Bool} = \mathsf{Bool}$. In fact, if we have a hierarchy of univalent universes $\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \ldots$, then Type_{n+1} has a strictly more complicated higher equality structure [KS15].

Apart from univalent universes of types invalidating uniqueness of identity proofs, there are also plenty of geometric examples to take from homotopy theory, such as the circle and the torus. As we have seen, we define new data types in type theory usually as an inductive type. However, ordinary inductive definitions do not give us a way to create new paths between points that were not already there. The solution is to generalise the idea of constructor to also allow for paths to be constructed. We can then define the circle, which is just a point with a non-trivial loop, as the following *higher inductive type*:

$$\begin{aligned}&\textbf{data } S^1 : \mathsf{Type} \textbf{ where}\\ &\quad \mathsf{base} : S^1\\ &\quad \mathsf{loop} : \mathsf{base} = \mathsf{base}\end{aligned}$$

In higher inductive types, we have the ordinary constructors, such as base in this example. These are called *point constructors*, as they can be thought of as constructing points in the space we are defining inductively. The constructor loop is a *path constructor*: it constructs a new path from base to base.

The induction principle for the circle is as follows:

$$S^1\text{-elim} : (P : S^1 \to \mathsf{Type})\,(m_{\mathsf{base}} : P\,\mathsf{base})\,(m_{\mathsf{loop}} : m_{\mathsf{base}} =^{P}_{\mathsf{loop}} m_{\mathsf{base}}) \to (x : S^1) \to P\,x$$

To show that P holds for any $x : S^1$, we need to show that it holds for the base point and that transporting this value along the loop is equal to the value itself.

The induction principle for the circle can also be used to show that the loop is in fact a non-trivial loop, i.e. $\mathsf{loop} \neq \mathsf{refl}_{\mathsf{base}}$. To prove this, we need to have a universe at hand which is univalent. This is similar to how we cannot show that $\mathsf{true} \neq \mathsf{false}$, if there is no universe to eliminate into, with the additional requirement of that universe being univalent.

Apart from allowing constructors to construct paths between points, higher inductive types also allow for *higher* path constructors which construct paths between paths. For example, we can describe the torus as the following higher inductive type:

$$\begin{aligned}
&\mathbf{data}\ T^2 : \mathsf{Type}\ \mathbf{where}\\
&\quad \mathsf{base} : T^2\\
&\quad \mathsf{p} : \mathsf{base} = \mathsf{base}\\
&\quad \mathsf{q} : \mathsf{base} = \mathsf{base}\\
&\quad \mathsf{r} : \mathsf{p} \cdot \mathsf{q} = \mathsf{q} \cdot \mathsf{p}
\end{aligned}$$

The approach of using higher inductive types to define these topological spaces (up to homotopy) has been used successfully to formalise various results of homotopy theory in type theory, leading to the field of *synthetic homotopy theory*. Formalisations include the fundamental group of circle [LS13a] and more general results on the homotopy groups of spheres [LB13; Bru16], the Blakers-Massey theorem [Hou+16] and the Mayer-Vietoris sequence [Cav15].

Applications aside, as of yet a general definition and theory of higher inductive types is still lacking. There is a note on the semantics of higher inductive types [LS13b], i.e. a metatheoretic and semantic treatment. The

ideas in this thesis are very much inspired by this note. In [Soj14], the author gives an internal specification of a class of higher inductive types called W-suspensions, i.e. the author defines an induction principle and the category of algebras and shows that initiality in the category of algebras coincides with satisfying the induction principle. This is all carried out inside type theory itself, which is also a goal of this thesis.

1.5 A theory of quotient inductive-inductive definitions

Our goal in this thesis is to give a theory of quotient inductive-inductive definitions, which can be seen as a stepping stone towards a theory of higher inductive types. Quotient inductive-inductive types can either be seen as a generalisation of inductive-inductive types or as a subclass of higher inductive-inductive types. We extend inductive-inductive definitions with path constructors, but we truncate the result to be a set: our definitions satisfy uniqueness of identity proofs. As such, they can be seen as a subclass of higher inductive-inductive types. Since we work with sets, we only consider path constructors that construct paths between points, as any higher path constructors would not add anything new.

Another goal of this thesis is to give a theory that can be expressed inside type theory itself. While formalising the theory is already a good idea on its own, having an internal definition of the inductive definitions means that anything we prove about them internally automatically holds for all models of type theory. Furthermore, if the theory happens to be expressible in a small core theory and we manage to construct the inductive types also in this core, we have an implementation of our inductive definitions in this small type theory.

From a programming point of view, having the specification of inductive definitions as a first-class citizen in your language allows for metaprogramming: we can write generic programs by recursion on the structure of the inductive definitions [BDJ03].

Working on the theory of inductive definitions internal to type theory is not without problems, however. When we talk about the computation rules of type theoretic inductive definitions, we talk about *definitional* equalities. These definitional equalities have none of the non-trivial structure that its internal counterpart, propositional equality, may have. They are equalities *on the nose*. In type theory itself, we can only talk about the weaker notion of propositional equality. This complicates the situation already when we talk about ordinary inductive types: in [AGS12] the authors internally prove for W-types that initiality and the induction principle coincide. Doing so requires some involved reasoning about equalities between propositional equalities: there are so called coherence problems one needs to solve. In this thesis we observe in appendix B that these coherence problems grow in number with the number of constructors, even if they are all just point constructors.

Dealing with these coherences in a uniform way means that we have to talk about $(\infty, 1)$-categories [Cam13]. Formalising the usual notions of $(\infty, 1)$-category in type theory requires us to work with simplicial sets in type theory. However, we want to work with *types*, not simplicial sets. A more promising approach is extending type theory with a strict equality [AK16; ACK16a]. In this thesis, we will deal with the coherences by simply working with the truncated types: we work with sets instead, i.e. types that satisfy uniqueness of identity proofs.

1.6 Related work

The work in this thesis builds on the work on inductive-inductive types [Nor13; Alt+11]. The framework we present extends the work in that paper in several dimensions: as opposed to having one constructor of sort $\mathcal{S}et$ and one constructor of sort $\mathcal{F}am$, our framework supports a larger class of sorts and any number of constructors. Apart from that, our framework also introduces support for path constructors.

The notion of path constructors comes from the concept of higher inductive types, as used in [Uni13]. A first attempt at describing the semantics

of higher inductive types has been done in [LS13b], working at the level of model categories. This approach has inspired [Cap14] and [Alt+15], which both have led to the work presented in this thesis.

Furthermore, as the title of the thesis suggests, the work on quotients in type theory, most notably [Li15] and [Hof95], is related to what we present here. Quotient inductive-inductive definitions combine a quotienting mechanism with inductive types in the sense that the quotienting happens "at the same time" as the induction. The benefit over ordinary quotients will be discussed in chapter 3.

1.7 Overview of the thesis and contributions

The thesis is organised as follows:

- In chapter 2, we give the basic concepts and notation we will use in this thesis, along with proofs of basic propositions and lemmata used throughout the thesis.

- Chapter 3 presents examples of quotient inductive-inductive definitions and their applications and compares them to other notions such as quotients and coequalisers.

- Chapter 4 contains the formal specification of quotient inductive-inductive definitions. They are specified as being a certain kind of iterated dialgebras, which generalises the presentation of ordinary inductive definitions as algebras of functors.

- Having a formal specification of quotient inductive-inductive definitions and the corresponding categories of algebras, we show in chapter 5 that the initial algebra semantics coincides with a categorical formulation of the type theoretic induction principle. From the categorical formulation we derive the type theoretic formulation, showing that induction and initiality coincide for our inductive definitions.

- In chapter 6 we give some preliminary results on constructing quotient inductive-inductive definitions given some reasonable assumptions on the type theory.

- The last chapter, chapter 7, presents the conclusions of the thesis along with a discussion of the presented results. We also point out several avenues for future work.

- Appendix A discusses some ways to generalise containers to accommodate the kind of functors we need in our framework.

- Appendix B is about what the difficulties are when moving from a set-based setting to the untruncated case. Here we go into detail about the coherence issues we face and how some of these can be solved in an ad-hoc manner.

1.7.1 List of main contributions

The main contributions of this thesis are:

- a formal specification of quotient inductive-inductive definitions, alongside with their categorical interpretation, given in type theory: definition 4.4.1 gives the general scheme of the specification with the details of point constructors in definition 4.4.2 and definition 4.4.3 and the details of path constructors in definition 4.4.5 and definition 4.4.6.

- a proof of the logical equivalence of initiality and the category theoretic induction principle: theorem 5.3.9.

- a derivation of the type theoretic induction principle from the category theoretic one: section 5.4.3.

- construction of initial algebras for a class of quotient inductive types: theorem 6.3.13.

1.7.2 Declaration of authorship and previous work

This thesis would not have existed in this form if not for the fruitful discussions with Thorsten Altenkirch, Paolo Capriotti and Fredrik Nordvall Forsberg. Our first approach to a theory of higher inductive types was presented at the TYPES workshop in 2015 [Alt+15]. My efforts to make the details work for this dependent dialgebra approach, led me away from dependent dialgebras and eventually to the approach presented in this thesis. A preprint [Alt+16] presenting this new work was written in July 2016 by me with editorial efforts from the coauthors. This preprint contains most of the material presented in this thesis, except for chapter 6, appendix A and appendix B, albeit in highly condensed form, lacking in details and proofs.

Chapter 2

Preliminaries

In this chapter we will introduce the type theoretic background material for the thesis. We will not give a detailed overview of Martin-Löf Type Theory in this chapter. For such an overview, we refer the reader to the first chapter of [Uni13]. Instead, we introduce notation and basic propositions and lemmata that we will use throughout the thesis. As we will be using category theoretic approaches, the chapter is concluded with our approach to category theory in type theory.

2.1 Basic type formers

In type theory we can form Π-types, or dependent functions, given a type A and for every $a : A$ a type $B\ a$, we have the type:

$$(x : A) \to B\ a$$

Traditionally this is denoted as $\Pi(x : A).B\ a$ or $\Pi A.B$. We will use Agda's notation for Π-types instead, along with its convention of leaving out arrows in the case of nested Π-types, where convenient, e.g. we may write $(a : A) \to (b : B\ a) \to C\ a\ b$ as $(a : A)\ (b : B\ a) \to C\ a\ b$. The introduction rule for Π-types is that if we have $x : A$ in our context and a term $b : B\ x$,

then we can form the term:

$$\lambda x.b : (x : A) \to B\ x$$

Π-types have an elimination rule, called *application*: if $f : (x : A) \to B\ x$ and $x : A$, then $f\ x : B\ x$. Terms of Π-types are subject to two classes of definitional equalities, β-equality:

$$(\lambda x.e)\ y \equiv e[x/y]$$

where $e[x/y]$ denotes the substitution of every occurrence in e of x with y. We use the symbol $\equiv$ to denote definitional equality on terms.

Furthermore we have η-equality, for every $f : (x : A) \to B\ x$:

$$f \equiv \lambda x.f\ x$$

Note that if the codomain of a Π-type does not refer to elements of its domain, i.e. when we have a non-dependent function, we denote it as $A \to B$.

Apart from Π-types, we also have Σ-types, or dependent pairs/products. The type formation rule is similar to that of Π-types: if we have a type A such that if we have $a : A$ in our context, the term $B\ a$ is a type, then we have that the following is a type:

$$(a : A) \times B\ a$$

This is traditionally written as $\Sigma(a : A).B\ a$ or $\Sigma A.B$, but we are using a notation here which mirrors Agda's notation for Π-types.

If we have $a : A$ and $B\ a$, then we can form the term:

$$(a, b) : (a : A) \times B\ a$$

The type $(a : A) \times B\ a$ comes with two projection functions:

- $\pi_0 : (a : A) \times B\ a \to A$
- $\pi_1 : (x : (a : A) \times B\ a) \to B\ (\pi_0\ x)$

Since we are explicitly giving a name to the first element of the pair, we can also use the following notation, where we let the scope of a range over the whole expression after its introduction:

$$\pi_1 : (a : A) \times B\ a \to B\ a$$

The projection functions have as β-laws: for every $a : A$ and $b : B\ a$, $\pi_0\ (a, b) \equiv a$ and $\pi_1\ (a, b) \equiv b$.

The η-law for Σ-types gives us the following definitional equality for every pair $z : \Sigma AB$:

$$z \equiv (\pi_0\ z, \pi_1\ z)$$

2.1.1 Universes

Another basic type is the type whose inhabitants themselves are again types, the universe Type. Using such a universe, we can now rephrase the type formation for Π-types as having a term:

$$\Pi : (A : \mathsf{Type}) \to (B : A \to \mathsf{Type}) \to \mathsf{Type}$$

Such a universe may not have the property that $\mathsf{Type} : \mathsf{Type}$ as that gives rise to a paradox [Gir72; Coq86], similar to Burali-Forti paradox in set theory. Instead, we have a hierarchy of universes:

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \ldots$$

While avoiding the paradox, this hierarchy brings a lot of complexity to our language in terms of usability. Every time we use a universe, we have to explicitly indicate its level, cluttering our terms.

One solution to this problem is to leave the level indices as implicit variables with constraints. One can then let the type checker attempt to solve this system, which is the approach Coq takes.

Another problem is that we may want to give definitions that can be instantiated to more than one level. If we want to define a function com-

position operator, we would like that it works for functions between types of any level. In Agda, we have the notion of *universe polymorphism*. We can define terms such as:

$$\begin{aligned}&\mathsf{id} : (n : \mathsf{Level})\ (A : \mathsf{Type}_n) \to A \to A\\&\mathsf{id}\ n\ A\ x :\equiv x\end{aligned}$$

The function id is polymorphic over all levels n. While this saves us from having to define id for any level, universe polymorphism does introduce some complexity of its own. Level is *not* a type, which means that while $(n : \mathsf{Level})\ (A : \mathsf{Type}_n) \to A \to A$ is a valid Agda *expression*, it is not a type, i.e. it is not an inhabitant of Type_i for any $i : \mathsf{Level}$.

In this thesis we will leave the universe levels implicit.

2.1.2 Implicit arguments

An important feature of Agda is that we can denote arguments of a function as being an *implicit argument*. One use case is the aforementioned universe levels. If we consider again the type and definition of id, with implicit arguments, we can write:

$$\begin{aligned}&\mathsf{id} : \{\ n : \mathsf{Level}\ \}\ \{\ A : \mathsf{Type}_n\ \} \to A \to A\\&\mathsf{id}\ x :\equiv x\end{aligned}$$

The idea here is that the context in which a function is called gives information we can use to fill in the blanks: if we call id with a certain argument, we know the type of the argument and in turn also know the universe level of that type. To improve readability we will oftentimes implicitly pass around parameters, while sometimes making them explicit by adding subscripts.

2.1.3 Inductive data types

As mentioned in the introduction, an important feature to have in a type theory is to have a way to define custom inductive data types. In Agda, one

can define an inductive type by giving a list of constructors. For example, the natural numbers can be defined as follows:

```
data ℕ : Type where
  zero : ℕ
  succ : ℕ → ℕ
```

We will often use Agda-like notation to give definitions/specifications of inductive types.

Two trivial, yet important types we will use are the empty and the unit types, which can be defined as inductive types:

```
data 0 : Type where
```

```
data 1 : Type where
  tt : 1
```

The "smallest non-trivial" data type, Bool can also be defined inductively:

```
data Bool : Type where
  true : Bool
  false : Bool
```

Inductive definitions may also be parametrised. We may define the coproduct of two types as follows:

```
data _+_ (A B : Set) : Set where
  inl : A → A+B
  inr : B → A+B
```

Inductive data type declarations in Agda do not give us elimination

principles directly. Instead Agda has a notion of *dependent pattern matching* [Coq92]: we write our functions out of inductive types by matching against the possible constructors of said types. These function definitions may be recursive, as long as the recursive occurrences are on a subterm of the pattern. Dependent pattern matching has been shown to be equivalent to having eliminators, assuming all types satisfy uniqueness of identity proofs [GMM06]. However, with some further restrictions on pattern matching, translations to eliminators without requiring uniqueness of identity proofs do exist [CDP14]. Moreover, the translation of pattern matching definitions into definitions with eliminators does not always preserve definitional equalities [McB06].

As opposed to adding inductive types to the theory in an ad hoc manner, one can add W-types as a primitive and define all inductive types in terms of this primitive. Suppose we have a type of *shapes* $S : \mathsf{Type}$ and a type family of *positions* $P : S \to \mathsf{Type}$, then the W-type defined by S and P is:

$$\begin{aligned}&\mathbf{data}\ \mathsf{W}\ S\ P : \mathsf{Type}\ \mathbf{where}\\ &\quad \mathsf{sup} : (s : S) \times (P\ s \to W\ S\ P) \to W\ S\ P\end{aligned}$$

We can think of the shapes as the type giving the contexts for constructors. For example, for the type of lists containing natural numbers, we choose $S :\equiv 1{+}\mathbb{N}$: the nil constructor has no context and the cons constructor takes something of type $\mathbb{N}$. The positions family gives us the recursive positions. For the nil constructor we have zero recursive positions and for cons we have exactly one, so we define P with $P\ (\mathsf{inl}\ \mathsf{tt}) :\equiv 0$ and $P\ (\mathsf{inr}\ n) :\equiv 1$ for any $n : \mathbb{N}$.

W-types are enough to give us ordinary inductive types as well as indexed inductive types [Mor07]. In this thesis, the inductive definitions we need to formalise the theory of quotient inductive-inductive types, can all be written using W-types. However, for sake of convenience and presentational clarity, we will use Agda notation to present these. We will also sometimes give function definitions by pattern matching, but these can be

trivially rewritten using just the eliminators.

2.2 Equality

So far we have mentioned definitional equality between terms. This equality is a metatheoretic notion, it is external to the type theory, i.e. $x \equiv y$ is not a type nor a term. We can define a notion of equality between terms in the type theory itself, referred to as *propositional equality*:

Definition 2.2.1 (Identity types)**.** Given a type $A : \mathsf{Type}$ and $x, y : A$, we define the type $x = y : \mathsf{Type}$, inductively with constructor:

$$\mathsf{refl} : (x : A) \to x = x$$

A type $x = y$ is called an *identity type*.

The elimination principle for identity types is:

$$\begin{aligned} \mathsf{J} : (A : \mathsf{Type})\ (P : (x\ y : A) \to x = y \to \mathsf{Type}) \\ (m : (x : A) \to P\ x\ x\ (\mathsf{refl}\ x)) \\ \to (x\ y : A)\ (p : x = y) \to P\ x\ y\ p \end{aligned}$$

which satisfies the computation rule:

$$\mathsf{J}\ A\ P\ m\ x\ x\ (\mathsf{refl}\ x) \equiv m\ x$$

Remark 2.2.2. The elimination principle for identity types is also referred to as the *path induction* principle.

Remark 2.2.3. While it is called *propositional* equality, the type $x = y$ need not be propositional in general.

Lemma 2.2.4. *Given a type $A : \mathsf{Type}$, the identity types form an equivalence relation on A.*

Proof. We need to show that the relation $\_ = \_ : A \to A \to \mathsf{Type}$ is reflexive, symmetric and transitive. Reflexivity we immediately get from the refl

constructor. For symmetry we need to appeal to the elimination principle. Let $p : x = y$, we define $p^{-1} : y = x$:

$$p^{-1} :\equiv \mathsf{J}\ A\ (\lambda x\ y\ p.y = x)\ (\lambda x.\mathsf{refl}\ x)\ x\ y\ p$$

For transitivity, we need an operation:

$$\_ \cdot \_ : x = y \to y = z \to x = z$$

This can be defined by path induction on any of combination of its arguments. □

Lemma 2.2.5. *The identity types on a type $A : \mathsf{Type}$ form a groupoid with respect to propositional equality.*

Proof. The identity types on A form a groupoid with transitivity as its composition, its inverses given by symmetry and the unit being reflexivity. These operations need to adhere to the following laws:

- associativity: $(p : x = y)\ (q : y = z)\ (r : z = w) \to (p \cdot q) \cdot r = p \cdot (q \cdot r)$
- left identity: $(p : x = y) \to \mathsf{refl}_x \cdot p = p$
- right identity: $(p : x = y) \to p \cdot \mathsf{refl}_y = p$
- left inverse: $(p : x = y) \to p^{-1} \cdot p = \mathsf{refl}_y$
- right inverse: $(p : x = y) \to p \cdot p^{-1} = \mathsf{refl}_x$

These can all be shown to hold by performing path induction on the paths. □

Remark 2.2.6. The computational properties of transitivity obviously depend on how it is defined: we either get the left identity or the right identity law to hold definitionally if we perform path induction on one argument. If it is defined by path induction on both arguments, we get neither law "for free". We can make a new definition of propositional equality on top of the original one, which satisfies the category laws definitionally, as we will see in section 2.2.8.

Since identity types are types again, they themselves are also equipped with a groupoid structure. We can always consider identity types of identity types of identity types, et cetera. Such an infinite tower of groupoids on top of each other is called an ∞-groupoid. An external proof of the fact that every type gives rise to an ∞-groupoid has been given [VG11; Lum09].

2.2.1 Dependent equality

When considering type families $B : A \to \mathsf{Type}$ over some type A, we sometimes find ourselves in the situation that we want to talk about equality of $a : B\ x$ and $b : B\ y$ where $x = y$ but not necessarily $x \equiv y$. In this case the term $a = b$ is well-typed. However, given a path $x = y$, inhabitants of $B\ x$ can be *transported* to inhabitants of $B\ y$:

Definition 2.2.7. Given $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$, we define:

$$\mathsf{transport}\ B : (p : x = y) \to B\ x \to B\ y$$
$$\mathsf{transport}\ B\ p\ a :\equiv J\ A\ (\lambda x\ y\ q.B\ x \to B\ y)\ (\lambda x\ a.a)\ x\ y\ p\ a$$

Using transport we can define the notion of *dependent path*:

Definition 2.2.8. Given $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$ and a path $p : x = y$ for $x, y : A$, the type of dependent equalities between $a : B\ x$ and $b : B\ y$ is defined as:

$$a =_p^B b : \mathsf{Type}$$
$$a =_p^B b :\equiv \mathsf{transport}\ B\ p\ a = b$$

2.2.2 Functoriality of functions

The equality relation is a congruence: it is preserved by any function.

Proposition 2.2.9. *Given a function $f : A \to B$ and an equality $x = y$, we have $f\ x = f\ y$.*

Proof. We can define the operation ap (action (of the function) on paths) by path induction:

$$\mathsf{ap} : (f : X \to Y) \to x = y \to f\ x = f\ y$$
$$\mathsf{ap}\ f\ p :\equiv \mathsf{J}\ X\ (\lambda x\ y\ p.f\ x = f\ y)\ (\lambda x.\mathsf{refl}_{f\ x})\ x\ y\ p$$

□

We can regard the identity type on a type as a category: we have identity morphisms given by refl and composition given by $\_ \centerdot \_$, which is associative and satisfies the left and right unit laws. In this light, any function $A \to B$ can be thought of as a functor from the identity types on A to the identity types on B:

Proposition 2.2.10. *For any function $f : X \to Y$, $\mathsf{ap}\ f$ is functorial:*

- $\mathsf{ap}\ f\ \mathsf{refl}_x = \mathsf{refl}_{f\ x}$
- $\mathsf{ap}\ f\ (p \centerdot q) = \mathsf{ap}\ f\ p \centerdot \mathsf{ap}\ f\ q$

Proof. By path induction. □

ap is also functorial in its first argument:

Proposition 2.2.11. *For any $p : x = y$, the following holds:*

- $\mathsf{ap}\ \mathsf{id}_A\ p = p$
- $\mathsf{ap}\ (g \circ f)\ p = \mathsf{ap}\ g\ (\mathsf{ap}\ f\ p)$

Proof. By path induction. □

Apart from having an action on paths for functions, we can define a similar operation for dependent functions:

Definition 2.2.12.

$$\mathsf{apd} : (f : (x : A) \to B\ x) \to (p : x = y) \to f\ x =^B_p f\ y$$

is defined by path induction similar to ap.

A useful property of dependent paths over families of which the fibres are equalities is the following:

Proposition 2.2.13. *Let $f, g : X \to Y$ be two functions, $x, y : X$ and suppose we have the following data:*

- $p : f\ x = g\ x$
- $q : f\ y = g\ y$
- $r : x = y$

then a dependent equality:

$$p =_{r}^{\lambda x.f\ x = g\ x} q$$

is equivalent to:

$$p \cdot \mathsf{ap}\ g\ r = \mathsf{ap}\ f\ r \cdot q$$

This proposition allows us to rewrite dependent equalities to non-dependent ones, which are easier to manipulate and reason about.

Remark 2.2.14. When we are dealing with equalities between compositions of paths such as in the above proposition, then it makes sense to represent this as a commutative diagram. The above example translates to the following diagram:

$$\begin{array}{ccc} f\ x & \xrightarrow{\ p\ } & g\ x \\ {\scriptstyle \mathsf{ap}\ f\ r}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{ap}\ g\ r} \\ f\ y & \xrightarrow[\ q\ ]{} & g\ y \end{array}$$

Paths are denoted with lines with no arrow heads, as they are invertible. Since direction is not always relevant, we usually denote p^{-1} as p in such a diagram.

2.2.3 Truncation levels

If we consider the tower of identity types for a specific type A, it is sometimes the case that after several levels the identity types *vanish*, i.e. they are equivalent to the unit type. Another way of stating that a type is equivalent to the unit type is that it is *contractible*:

Definition 2.2.15. A type A is *contractible* if there exists a "central" point $c : A$ such that every point is connected to the centre via an equality:

$$\begin{aligned}&\mathsf{is\text{-}contr} : (A : \mathsf{Type}) \to \mathsf{Type}\\ &\mathsf{is\text{-}contr}A :\equiv (c : A) \times ((x : A) \to c = x)\end{aligned}$$

We can show that the identity types for a contractible type are again contractible: $\mathsf{is\text{-}contr}\ A \to (x\ y : A) \to \mathsf{is\text{-}contr}\ (x = y)$. The level at which the identity types vanish is the so called *truncation level* or *h-level* of the type. The level is indicated using natural numbers, but starting at -2 instead of 0, denoted as $\mathbb{N}_{-2}$:

Definition 2.2.16. A type has truncation level $n : \mathbb{N}_{-2}$, if we have a proof of $\mathsf{is\text{-}n\text{-}trunc}\ A$ where $\mathsf{is\text{-}n\text{-}trunc}$ is defined recursively over n:

$$\begin{aligned}&\mathsf{is\text{-}n\text{-}trunc} : (A : \mathsf{Type}) \to \mathsf{Type}\\ &\mathsf{is}{-}\mathsf{2\text{-}trunc}\ A :\equiv \mathsf{is\text{-}contr}\ A\\ &\mathsf{is\text{-}n} + \mathsf{1\text{-}trunc}\ A :\equiv (x\ y : A) \to \mathsf{is\text{-}n\text{-}trunc}\ (x = y)\end{aligned}$$

For our purposes, there are two important truncation levels, apart from contractibility:

Definition 2.2.17. A type is a *proposition* or *propositional* if it has truncation level -1:

$$\mathsf{is\text{-}prop}\ X :\equiv \mathsf{is}{-}\mathsf{1\text{-}trunc}\ X$$

A useful characterisation of propositions is given by the following proposition:

Proposition 2.2.18. *If the type A satisfies $(x\ y : A) \to x = y$ then A is propositional.*

From this characterisation, we immediately see that if A is propositional, then it is either empty or contractible, i.e. uniquely inhabited up to propositional equality. Of course, we do not have an internal proof of $(A : \mathsf{Type}) \to$

is-prop $A \to (A \to 0) + (\text{is-contr } A)$, which is essentially the law of excluded middle.

The next truncation level, level 0, are the so called sets:

Definition 2.2.19. A type is a *set* if it has truncation level 0, i.e. for all $x, y : A$, $x = y$ is propositional.

$$\text{is-set } X :\equiv \text{is-0-trunc } X$$

The property of being a set is also called "uniqueness of identity proofs".

Definition 2.2.20. We define the universe of sets as follows:

$$\begin{aligned} &\text{Set} : \text{Type} \\ &\text{Set} :\equiv (X : \text{Type}) \times \text{is-set } X \end{aligned}$$

For the sake of brevity, we will not be explicit about projecting the type out of inhabitants of Set and will write things such as $(A : \text{Set}) \to A \to A$, as though it were Type.

2.2.4 Equivalence

We have a notion of equality on types, so far have not talked in great detail about equality on a universe of types, i.e. equality *between* types.

A very weak notion of types being equal is "logical equivalence":

Definition 2.2.21. Types A and B are *logically equivalent* if we have functions $A \to B$ and $B \to A$:

$$\begin{aligned} &A \leftrightarrow B : \text{Type} \\ &A \leftrightarrow B :\equiv (A \to B) \times (B \to A) \end{aligned}$$

With logical equivalence we do not have any requirements on the two functions between the two types, just that there exist any. There being a logical equivalence is also not a proposition: the type Bool $\leftrightarrow$ Bool has at least two distinct inhabitants: $(\text{id}_{\text{Bool}}, \text{id}_{\text{Bool}})$ and (not, not).

The notion of isomorphism also requires the two functions to be each other's inverses:

Definition 2.2.22 (Isomorphism)**.** Types A and B are *isomorphic* if they are each other's inverse:

$$\begin{aligned} &A \simeq B : \mathsf{Type} \\ &A \simeq B :\equiv (f : A \to B) \times (g : B \to A) \\ &\qquad \times ((x : A) \to g\,(f\,a) = a) \times ((y : B) \to f\,(g\,y) = y) \end{aligned}$$

The type of isomorphisms is not necessarily propositional: the examples given for logical equivalence can be shown to be distinct isomorphisms as well.

One downside of the notion of isomorphism is that given a function $f : A \to B$, the "proposition" whether f is an isomorphism or not is not propositional, i.e. the following type is not in general a proposition:

$$(g : B \to A) \times (f \circ g = \mathsf{id}_B) \times (g \circ f = \mathsf{id}_A)$$

A more well-behaved notion is the notion of equivalence:

Definition 2.2.23 (Equivalence)**.** A function $f : A \to B$ is an *equivalence* if the following type is inhabited:

$$\begin{aligned} &\mathsf{is\text{-}equiv}\ f : \mathsf{Type} \\ &\mathsf{is\text{-}equiv}\ f :\equiv (g : B \to A) \times f \circ g = \mathsf{id}_B \times (h : B \to A) \times h \circ f = \mathsf{id}_A \end{aligned}$$

If there exists an equivalence between types A and B, we denote this as $A \cong B$:

$$\begin{aligned} &A \cong B : \mathsf{Type} \\ &A \cong B :\equiv (f : A \to B) \times \mathsf{is\text{-}equiv}\ f \end{aligned}$$

Equivalences satisfy the following useful properties (for proofs we refer the reader to [Uni13]):

Proposition 2.2.24. *Every isomorphism gives rise to an equivalence.*

Proposition 2.2.25. *Given a function* $f : A \to B$, is-equiv f *is a proposition.*

For a function to be an equivalence is a proposition, but for two types to be equivalent is not propositional. The same example as for logical equivalence and isomorphism can be used to show this.

Note that all these relations are equivalence relations. We can also show that composing them is associative. Isomorphisms and equivalences furthermore satisfy the groupoid laws.

2.2.5 Univalence

We have seen that we can formulate an appropriate notion of equality between types, but now we have two different ways of stating equality on the universe of types: equivalence and identity types. All the things we have proven about identity types, have to proven again for equivalences.

While identity types come with the path induction principle, equivalences do not enjoy such an induction principle. Even though we lack such a principle, we still can show that equivalences form an equivalence relation and also satisfy the groupoid laws. Once we try to show that equivalence gives us a congruence relation, we get stuck. In other words: we do not have a proof of the statement that every construction in type theory is invariant under equivalence.

Definition 2.2.26. Every equality between types gives rise to an isomorphism:

$$\begin{aligned} &\text{id-to-equiv} : (A = B) \to A \cong B \\ &\text{id-to-equiv} :\equiv \mathsf{J}\ (\lambda A\ B\ p.A \cong B))\ (\lambda A \to \text{id-equiv}\ A) \end{aligned}$$

where id-equiv is the identity equivalence $A \cong A$.

Definition 2.2.27 (Univalence)**.** A universe of types is *univalent* if the function id-to-equiv is an equivalence.

Hence univalence gives us an equality:

$$(A \cong B) = (A = B)$$

As equivalence and equality between types are equal things, we will speak about equivalences between types whilst using the symbol for equality.

Often we will give an equality between types by providing an isomorphism, which gives us an equivalence, which gives us an equality by virtue of univalence.

We will assume univalence holds for our universes of types unless otherwise indicated.

2.2.6 Function extensionality

In mathematics, one usually proves that two functions are equal by showing that they are pointwise equal: we appeal to function extensionality. In type theory this can be formulated as follows:

Definition 2.2.28 (Function extensionality)**.** Suppose we have functions $f, g : (x : X) \to P\ x$, function extensionality gives us a term of type:

$$((x : X) \to f\ x = g\ x) \to f = g$$

In type theory we can define a function that goes in the other direction. We have shown that the identity types form a congruence relation. In particular, we can define the following function, given two dependent functions $f, g : (x : X) \to P\ x$:

$$\mathsf{happly}\ f\ g : f = g \to ((x : X) \to f\ x = g\ x)$$
$$\mathsf{happly}\ f\ g\ p = \lambda x.\mathsf{ap}\ (\lambda h.h\ x)\ p$$

Function extensionality is not provable in ordinary Martin-Löf Type Theory. However, it does follow from univalence and as we will see, it also follows from having quotients. In fact, these imply we have *strong function extensionality*:

Proposition 2.2.29 (Strong function extensionality)**.** *For any two functions* $f, g : (x : X) \to P\ x$*, we have a witness of:*

$$\mathsf{is\text{-}equiv}\ (\mathsf{happly}\ f\ g)$$

A proof of this statement can be found in [Uni13].

2.2.7 Equivalences of Σ-types

We often work with nested Σ-types and have to provide equivalences between them. There are basic equivalences that we often use to build up our bigger equivalences.

For the non-dependent product, we can show that it satisfies laws of a commutative monoid using univalence (these do not hold definitionally):

- $A \times 1 = 1 \times A = A$
- $A \times (B \times C) = (A \times B) \times C$
- $A \times B = B \times A$

We have similar properties for Σ-types:

- $(a : A) \times 1 = (x : 1) \times A = A$
- $(a : A) \times ((b : B) \times C\ (a, b)) = (x : (a : A) \times B\ a) \times C\ x$

Commutativity only makes sense when the types do not depend on each other, so we do not have such a statement for Σ-types. In equational reasoning we will not be explicit about applying associativity or other laws and often will neglect parentheses.

Proposition 2.2.30 (Singleton contraction)**.** *For any type* $A : \mathsf{Type}$ *and* $a : A$*, we have the following equivalence:*

$$((x : A) \times (x = a)) = 1$$

Proof. This follows directly from path induction. □

The type $(x : A) \times (x = a)$ is called a *singleton*. This fact is very useful in equational reasoning. It is comparable to high school algebra where multiplying by $\frac{a}{a}$ for a cleverly chosen expression a, or recognising that two factors cancel each other out, are a useful techniques.

Proposition 2.2.31 (Equality of inhabitants of Σ-types)**.** *Equality between inhabitants of a Σ-type is equivalent to a Σ-type of equalities. Given a family $A : Type$, $B : A \to \mathsf{Type}$, and $x, y : A$, $z : B\ a$, $w : B\ y$, we have:*

$$((x, z) = (y, w)) = (p : x = y) \times (q : z =^B_p w)$$

Proof. Using univalence, it is enough to show that there exists an equivalence between the two types. The function $((x, z) = (y, w)) \to (p : x = y) \times (q : z =^B_p w)$ can be given by path induction on the argument. Similarly for the function in the other direction, we can define this by applying path induction to both elements of the pair. Showing that these two functions are each other's inverses is again a simple case of path induction. □

2.2.8 Alternative formulation of identity types

We have seen that the identity types form a groupoid on its underlying type. The unit and associativity laws do not hold definitionally for this groupoid. When reasoning about equalities *between* paths, it can be annoying if we have to take associativity into account. For example, suppose we want to prove $a \cdot p \cdot (p^{-1} \cdot b) = a \cdot b$, for some expressions a and b. The easiest way to prove this is by applying path induction to p, which will reduce the goal to $a \cdot b = a \cdot b$. However, sometimes we are in comparable situations where we cannot perform path induction on all of the paths involved. In such a situation, we have to take a lot of intermediate steps invoking associativity and the unit laws explicitly, e.g.:

$$\begin{aligned} a \cdot p \cdot (p^{-1} \cdot b) &= a \cdot ((p \cdot p^{-1}) \cdot b) \\ &= a \cdot (\mathsf{refl} \cdot b) \\ &= a \cdot b \end{aligned}$$

Because associativity does not hold definitionally, we cannot give as proof term (sym-inv is the witness of $p \cdot p^{-1} = \mathsf{refl}$):

$$\mathsf{ap}\ (\lambda h.a \cdot h \cdot b)\ (\mathsf{sym\text{-}inv}\ p) : a \cdot (p \cdot p^{-1}) \cdot b = a \cdot b$$

as it has the wrong type, hence we are stuck with filling in associativity and unit laws here and there. This is particularly frustrating when one then has to prove things about the resulting path being equal to something else.

We can define an alternative formulation of identity types which have different computational properties. This formulation is based on the insight that functions on Type do satisfy the associativity and unit laws definitionally with respect to function composition. We define:

$$x ='_A y :\equiv (z : A) \to z = x \to z = y$$

Reflexivity and transitivity are then defined as follows

$$\begin{aligned}
&\mathsf{refl}' : x ='_A x \\
&\mathsf{refl}' :\equiv \lambda w\ p\ .\ p
\end{aligned}$$

$$\begin{aligned}
&\mathsf{trans}' : x ='_A y \to y ='_A z \to x ='_A z \\
&\mathsf{trans}'\ p\ q :\equiv \lambda w\ r\ .\ q\ w\ (p\ w\ r)
\end{aligned}$$

Since reflexivity and transitivity are essentially given by the identity function and function composition respectively, the unit laws and associativity are now satisfied definitionally.

There are functions to and fro between the two formulations:

$$\mathsf{to} : x = y \to x =' y$$
$$\mathsf{to}\ p = \lambda z\ q.q \cdot p$$

$$\mathsf{from} : x =' y \to x = y$$
$$\mathsf{from}\ p = p\ x\ \mathsf{refl}_x$$

Proposition 2.2.32. *Given a type A : Type with elements $x, y : A$, we have an equivalence:*

$$(x ='_A y) = (x =_A y)$$

Proof. The functions to and from give us an isomorphism. Let $p : x = y$, then we have that:

$$\mathsf{from}\ (\mathsf{to}\ p) = \mathsf{from}\ (\lambda z\ q.q \cdot p) = \mathsf{refl} \cdot p = p$$

In the other direction we get for $p : x =' y$:

$$\mathsf{to}\ (\mathsf{from}\ p) = \mathsf{to}\ (p\ x\ \mathsf{refl}) = \lambda z\ q.q \cdot p\ x\ \mathsf{refl}$$

Showing that this is equal to p can be done by employing function extensionality and applying path induction on the second argument. This is essentially showing that p is a natural transformation. □

This equivalence also maps reflexivitiy and transitivity of the original identity types onto composition of the alternative version and vice versa. The alternative identity types also satisfy the path induction principle.

The equivalence of $x = y$ and $x =' y$ can also be seen as a direct consequence of the type theoretic Yoneda lemma [Rij12].

2.3 Category theory in type theory

When defining the concept of category in type theory, we have to make several choices. A naive way of defining a category would be as the following Σ-type:

$$\mathsf{Cat} :\equiv (obj : \mathsf{Type}) \times (hom : obj \to obj \to \mathsf{Type}) \times (\ldots)$$

where on the place of the ellipsis one would have to fill in the category structure and laws. This definition of category gives rise to certain problems, which we briefly mention in section 2.3.1 and in more detail in appendix B.

We will use the definition of category as in [AKS15]. (However, what we call "category" here is called "precategory" in that article.)

Definition 2.3.1 (Category)**.** We define the type $\mathsf{Cat} : \mathsf{Type}$ as the following Σ-type:

$$\begin{aligned}
\mathsf{Cat} :\equiv\ & (obj : \mathsf{Type}) \\
& \times (hom : obj \to obj \to \mathsf{Type}) \\
& \times (hom\text{-}is\text{-}set : (X\ Y : obj) \to \mathsf{is\text{-}set}(hom\ X\ Y)) \\
& \times (id : (X : obj) \to hom\ X\ X) \\
& \times (comp : \{\ X\ Y\ Z : obj\ \} \to hom\ Y\ Z \to hom\ X\ Y \to hom\ X\ Z) \\
& \times (left\text{-}id : \{\ X\ Y : obj\ \}\ (f : hom\ X\ Y) \to comp\ (id\ Y)\ f = f) \\
& \times (right\text{-}id : \{\ X\ Y : obj\ \}\ (f : hom\ X\ Y) \to comp\ f\ (id\ X) = f) \\
& \times (assoc : \{\ X\ Y\ Z\ W : obj\ \}\ (h : hom\ Z\ W)\ (g : hom\ Y\ Z)\ (f : hom\ X\ Y) \\
& \qquad \to comp\ (comp\ h\ g)\ f = comp\ h\ (comp\ g\ f))
\end{aligned}$$

The type of objects of a category $\mathcal{C}$ is denoted with $|\mathcal{C}|$. The type of morphisms given objects $X, Y : |\mathcal{C}|$ is denoted as $\mathcal{C}(X, Y)$.

Example 2.3.2. The universe Set of types that are sets forms a category, denoted $\mathcal{S}et$, with its morphisms defined as functions between sets.

Remark 2.3.3. The universe Type is *not* a category in this sense, as for arbitrary types X, Y, the function space $X \to Y$ will in general not be a set.

Definition 2.3.4 (Functor)**.** Suppose $\mathcal{C}, \mathcal{D} : \mathsf{Cat}$, then we define the type of functors $\mathcal{C}$ to $\mathcal{D}$ as the following Σ-type:

$$\begin{aligned}\mathcal{C} \Rightarrow \mathcal{D} :\equiv\ & (obj : |\mathcal{C}| \to |\mathcal{D}|) \\ & \times (hom : \{\ X\ Y : |\mathcal{C}|\ \} \to \mathcal{C}(X, Y) \to \mathcal{C}(obj\ X, obj\ Y)) \\ & \times (id : \{\ X : |\mathcal{C}|\ \} \to hom\ (\mathsf{id}_{\mathcal{C}}\ X) = \mathsf{id}_{\mathcal{D}}\ (obj\ X)) \\ & \times (comp : \{\ X\ Y\ Z : |\mathcal{C}|\ \}\ (g : \mathcal{C}(Y, Z))\ (h : \mathcal{C}(X, Y)) \\ & \quad \to hom\ (g \circ_{\mathcal{C}} f) = hom\ g \circ_{\mathcal{D}} hom\ f)\end{aligned}$$

As is usual, we will use the notation FX, where $F : \mathcal{C} \Rightarrow \mathcal{D}$ and $X : |\mathcal{C}|$ for the action of F on object X and Ff for the action on morphisms $f : \mathcal{C}(X, Y)$. When defining functors, we will often only define the action on objects and leave the rest implicit.

Definition 2.3.5. The *empty category* $\mathbb{0}$ is defined as the category with no objects. The *unit category* $\mathbb{1}$ is defined as the category with one object and no non-trivial automorphisms.

The categories $\mathbb{0}$ and $\mathbb{1}$ are respectively initial and terminal in the category of categories.

Definition 2.3.6 (Natural transformation)**.** Given two functors $F, G : \mathcal{C} \Rightarrow \mathcal{D}$, we can define the type of *natural transformations* between them as follows:

$$\begin{aligned}F \dot{\to} G :\equiv\ & (\alpha : (X : |\mathcal{C}|) \to \mathcal{D}(FX, GX)) \\ \times\ & (\{\ X\ Y : |\mathcal{C}|\ \}\ (f : \mathcal{C}(X, Y)) \to Gf \circ_{\mathcal{D}} \alpha_X = \alpha_Y \circ_{\mathcal{D}} Ff)\end{aligned}$$

As the objects are given by a type, propositional equality gives us a way to talk about equality between objects. When doing category theory, we should always work with isomorphism if we want to talk about objects being equal:

Definition 2.3.7 (Isomorphism)**.** Let $X, Y : |\mathcal{C}|$ for some $\mathcal{C} : \mathsf{Cat}$, a mor-

phism $f : \mathcal{C}(X, Y)$ is an *isomorphism* if the following is inhabited:

$$\begin{aligned}&\mathsf{is\text{-}iso}\ f : \mathsf{Type}\\ &\mathsf{is\text{-}iso}\ f :\equiv (g : \mathcal{C}(Y, X)) \times (g \circ f = \mathsf{id}_{\mathcal{C}}\ X) \times (f \circ g = \mathsf{id}_{\mathcal{C}}\ Y)\end{aligned}$$

Note that since we are working with hom-sets, the property of a morphism being an isomorphism is propositional, unlike for isomorphisms on Type.

Definition 2.3.8. The type of isomorphisms between X and Y is defined as:

$$\begin{aligned}&X \simeq Y : \mathsf{Type}\\ &X \simeq Y :\equiv (f : \mathcal{C}(X, Y)) \times \mathsf{is\text{-}iso}\ f\end{aligned}$$

For any object $X : |\mathcal{C}|$, the identity morphism $\mathsf{id}_{\mathcal{C}}\ X$ is an isomorphism, denoted as $\mathsf{id\text{-}equiv}_{\mathcal{C}}\ X$.

The univalence axiom gives us that for sets the notions of isomorphism and propositional equality coincide. We can state a similar axiom for categories. Just as we have the function id-to-equiv that lifts any propositional equality on types to an equivalence, we have a function, for any two objects X, Y, $\mathsf{id\text{-}to\text{-}iso}_{\mathcal{C}}\ X\ Y : X = Y \to X \simeq Y$, defined by path induction.

Definition 2.3.9 (Univalent category)**.** A category $\mathcal{C}$ is called *univalent* if for any two objects $X, Y : |\mathcal{C}|$ the function $\mathsf{id\text{-}to\text{-}iso}_{\mathcal{C}}\ X\ Y$ is an equivalence.

Proposition 2.3.10. *The category $\mathcal{S}et$ is univalent.*

Proof. This follows directly from univalence for the universe Type. □

Example 2.3.11 (Category of type families)**.** Type families form a category, denoted $\mathcal{F}am$. Just as with $\mathcal{S}et$ we restricted ourselves to those types that satisfy uniqueness of identity proofs, we will only consider those families $X : \mathsf{Type}$, $P : X \to \mathsf{Type}$ of which the carrier X is a set and its fibres $P\ x$ are

sets, for any $x : X$. We can define a type of families:

$$\begin{aligned}&\mathsf{Fam} : \mathsf{Set}\\&\mathsf{Fam} :\equiv (X : \mathsf{Set}) \times (P : X \to \mathsf{Set})\end{aligned}$$

A morphism between two families is a function between the underlying types and a witness of the fact that this function "preserves the predicate":

$$\mathcal{F}am((X, P), (Y, Q)) :\equiv (f : X \to Y) \times (g : (x : X) \to P\ x \to Q\ (f\ x))$$

2.3.1 Higher categories

As mentioned before, the universe of types Type does not form a category if we take functions as morphisms. Function spaces will in general not be sets. However, we can define composition and identity morphisms in the usual way. These will satisfy the category laws definitionally as well.

We can relax the definition of category to not have hom-*sets* but hom-*types*. Composition and identity morphisms still make sense with this generalisation. We may run into issues with the category *laws*: in some situations it is not enough to simply have the laws, they also need to be coherent in some sense: the interaction of the laws with each other needs to satisfy certain laws as well. Naturally, how these coherence laws interact also has to adhere to additional higher coherence laws, ad infinitum. In appendix B we will describe the problems and possible solutions in greater detail.

2.4 Core type theory

The type theory we are working with in this thesis can be reduced to a type theory containing the following primitives:

- Π- and Σ-types
- The finite types 0 (empty type), 1 (unit type), and the booleans Bool
- a hierarchy of universes Type_0, Type_1, Type_2, ...

- W-types
- Identity types
- Univalence for the universes Type_0, Type_1, Type_2, ...

Chapter 3

Quotient inductive-inductive definitions

In this chapter we will give several examples of quotient inductive-inductive definitions, motivating their usefulness, before we set out giving a precise definition. We will also compare it to other notions such as higher inductive types and quotient types. We will discuss the status of current (partial) implementations of quotient and higher inductive types in various proof assistants.

3.1 Examples

3.1.1 Interval type

The simplest (non-empty) example of an inductive type with a path constructor which is still a set, is the interval type: two point constructors with a path constructor connecting them:

```
data I : Type where
  zero : I
  one : I
  seg : zero = one
```

It should come as no surprise that this type can be shown to be equivalent to the unit type. However, this does not mean it is entirely uninteresting: unlike the unit type, the interval type implies function extensionality (proposition 3.1.1).

As the interval type can be shown to be contractible, it does not matter whether we take the set truncation of it or not. As such, we will work with untruncated types for the remainder of this example.

To define a function out of an inductive type, we need to say what each constructor is mapped to. For the interval type, we first of all need to give two points. Since every function in type theory is a congruence, i.e. we have ap, these two points need to be equal to each other. The recursion principle is therefore as follows:

$$\mathsf{I\text{-}rec} : (A : \mathsf{Type})\ (a_{\mathsf{zero}} : A)\ (a_{\mathsf{one}} : A)\ (a_{\mathsf{seg}} : a_{\mathsf{zero}} = a_{\mathsf{one}}) \to \mathsf{I} \to A$$

which comes with the computation rules:

$$\mathsf{I\text{-}rec\text{-}}\beta_{\mathsf{zero}} : \mathsf{I\text{-}rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}}\ \mathsf{zero} = a_{\mathsf{zero}}$$
$$\mathsf{I\text{-}rec\text{-}}\beta_{\mathsf{one}} : \mathsf{I\text{-}rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}}\ \mathsf{one} = a_{\mathsf{one}}$$

Definitional versus propositional computation rules

For ordinary inductive types, the computation rules customarily hold definitionally, as they are called *computation* rules after all. However, inside Martin-Löf Type Theorywe can only talk about propositional equality. As such, our formalisation of quotient inductive-inductive types talks about computation rules holding propositionally.

There is an avenue of research in type theory that considers adding syntax to talk about definitional (also called *strict*) equalities to the theory [Coh+15]. In such a system, we would be able to model the computation rules of inductive definitions as definitional equalities.

Path computation rules

The recursion principle of an inductive type comes with a computation rule for every constructor. So far we have only given the computation rules for the *point constructors*. The computation rule for the path constructor is similar to those of the point constructors: it tells us that the action of $\mathsf{I}\text{-rec}$ on the path seg gives us back the a_{seg} we put in, that is:

$$\mathsf{ap}\ (\mathsf{I}\text{-rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}})\ \mathsf{seg} = a_{\mathsf{seg}}$$

However, the above does not type check if $\mathsf{I}\text{-rec}\text{-}\beta_{\mathsf{zero}}$ and $\mathsf{I}\text{-rec}\text{-}\beta_{\mathsf{one}}$ are not strict equalities. We would need to transport the left-hand side over these point computation rules. We end up with the following square:

$$\begin{array}{ccc} \mathsf{I}\text{-rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}}\ \mathsf{zero} & \xrightarrow{\mathsf{ap}\ (\mathsf{I}\text{-rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}})\ \mathsf{seg}} & \mathsf{I}\text{-rec}\ A\ a_{\mathsf{zero}}\ a_{\mathsf{one}}\ a_{\mathsf{seg}}\ \mathsf{one} \\ {\scriptstyle \mathsf{I}\text{-rec}\text{-}\beta_{\mathsf{zero}}}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{I}\text{-rec}\text{-}\beta_{\mathsf{one}}} \\ a_{\mathsf{zero}} & \xrightarrow[a_{\mathsf{seg}}]{} & a_{\mathsf{one}} \end{array}$$

A path computation rule gives us an equation between paths in the type we are eliminating into. If that type happens to be a set, then these equations would be trivial. Hence path computation rules for *quotient* inductive types do not add anything new and will be omitted.

Induction principle

The induction principle gives us a way to show that for some predicate P on I, we have a dependent function $(x : \mathsf{I}) \to P\ x$. We have to give a method for each point constructor and each path constructor. The method we need to give for the constructor seg is not a simple path as it was with the recursion principle: m_{zero} and m_{one} may have different types. We do know that they can be related by transporting along seg, so we end up having to

give a dependent path as the method for seg:

$$\begin{aligned}\mathsf{I\text{-}ind} : (P : \mathsf{I} \to \mathsf{Type})\\ (m_{\mathsf{zero}} : P\ \mathsf{zero})\\ (m_{\mathsf{one}} : P\ \mathsf{one})\\ (m_{\mathsf{seg}} : m_{\mathsf{zero}} =^{P}_{\mathsf{seg}} m_{\mathsf{one}})\\ \to (x : \mathsf{I}) \to P\ x\end{aligned}$$

As with the recursion principle, we of course have computation rules:

$$\mathsf{I\text{-}ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{one}}\ m_{\mathsf{seg}}\ \mathsf{zero} = m_{\mathsf{zero}}$$
$$\mathsf{I\text{-}ind}\ P\ m_{\mathsf{zero}}\ m_{\mathsf{one}}\ m_{\mathsf{seg}}\ \mathsf{one} = m_{\mathsf{one}}$$

Function extensionality

Even though the interval is equivalent to the unit type, it has the surprising property that it implies function extensionality. Having an inductive type with two inhabitants that are propositionally equal but not definitionally allows us to represent equalities using functions. From the recursion principle we get the following logical equivalence for any type A:

$$I \to A \leftrightarrow (x\ y : A) \times (x = y)$$

This is like the universal property for the interval with equivalence weakened to logical equivalence.

Proposition 3.1.1 ([Hof95; Shu11a])**.** *The interval type implies function extensionality.*

Proof. Suppose we have types A, B with two functions $f, g : A \to B$ and a family of equations $p : (x : A) \to f\ x = g\ x$. We can define $\tilde{p} : A \to I \to B$ as follows:

$$\tilde{p}\ a :\equiv \mathsf{I\text{-}rec}\ B\ (f\ a)\ (g\ a)\ (p\ a)$$

We can then construct the following term of type $f = g$:

$$\mathsf{ap}\ (\lambda\ i\ a.\tilde{p}\ a\ i)\ \mathsf{seg}$$

This assumes that the computation rules of I-rec hold definitionally and that the type theory satisfies the η-law for functions definitionally. However, the proof can be easily modified to also work when the η-law holds only up to propositional equality. It is however essential to this proof that the computation laws for I-rec hold definitionally. □

3.1.2 Quotients and colimits

Quotient types can be realised as a higher inductive type as follows: suppose we have a type $A : \mathsf{Type}$ equipped a binary relation $R : A \to A \to \mathsf{Type}$, we define:

$$\begin{aligned}
&\mathbf{data}\ A/R : \mathsf{Type}\ \mathbf{where}\\
&\quad [_] : A \to A/R\\
&\quad q : (x\ y : A) \to R\ x\ y \to [\,x\,] = [\,y\,]
\end{aligned}$$

The elimination principle of the quotient A/R is as follows:

$$\begin{aligned}
\mathsf{A/R\text{-}ind} : &(P : A/R \to \mathsf{Type})\\
&(m_{[_]} : (a : A) \to P\ [\,a\,])\\
&(m_q : (x\ y : A)\ (p : R\ x\ y) \to m_{[_]}\ x =^{P}_{q\ x\ y\ p} m_{[_]}\ y)\\
&\to (x : A/R) \to P\ x
\end{aligned}$$

Unlike with the interval types, whether we truncate or not does make a difference here. One would expect A to be a set and R to be Prop-valued. These restrictions are not enough to ensure the quotient will also be a set. If we take A to be the unit type and quotient it by the relation that is constantly the unit type as well, A/R will be equivalent to the circle.

It is not necessary to require R to be an equivalence relation. Since the

path constructor q constructs elements of an identity type, we can apply symmetry and other operations to the paths constructed by q. We effectively take the reflexive-symmetric-transitive closure of R.

In the category of sets, one can construct colimits by using quotients. The same is true for types: we can construct coequalisers using quotient types. In the untruncated setting, these coequalisers will be *homotopy* coequalisers. Since we already have arbitrary coproducts (Σ-types), we can then go on and construct arbitrary colimits using these two building blocks.

Quotient types implement coequalisers

If we have quotient types, we use them to define colimits such as coequalisers: suppose $A, B : \mathsf{Type}$ with $f, g : A \to B$ two functions, we can define a relation R on B:

$$R\ x\ y :\equiv (z : A) \times (x = f\ z) \times (y = g\ z)$$

Proposition 3.1.2 ([AAL11])**.** *The quotient B/R is the coequaliser of f and g, i.e. $[\_] : B \to B/R$ satisfies $[\_] \circ f = [\_] \circ g$ and B/R satisfies the appropriate universal property.*

Proof. We can show that the type of q is equivalent to $[\_] \circ f = [\_] \circ g$ by the following equational reasoning:

$$\begin{aligned}
& (x\ y : B) \to (z : A) \times (x = f\ z) \times (y = g\ z) \to [x] = [y] \\
= \quad & \{ \text{ currying and singleton contraction } \} \\
& (z : A) \to [f\ z] = [g\ z] \\
= \quad & \{ \text{ function extensionality } \} \\
& [_] \circ f = [_] \circ g
\end{aligned}$$

The universal property can be shown to follow directly from the elimination principle. □

Coequalisers implement quotient types

We can also define coequalisers directly as a higher inductive type. Given types A, B with functions $f, g : A \to B$, we define:

$$\begin{aligned}&\textbf{data } \mathsf{coeq}_{f,g} : \mathsf{Type} \textbf{ where}\\ &\quad c : B \to \mathsf{coeq}_{f,g}\\ &\quad \mathsf{eq} : (x : A) \to c\,(f\ x) = c\,(g\ x)\end{aligned}$$

which comes with the following elimination principle:

$$\begin{aligned}\mathsf{coeq}_{f,g}\text{-}\mathsf{ind} : (P : \mathsf{coeq}_{f,g} \to \mathsf{Type})\\ (m_c : (b : B) \to P\,(c\,b))\\ (m_{\mathsf{eq}} : (a : A) \to m_c\,(f\ a) =^{P}_{\mathsf{eq}\ a} m_c\,(g\ a))\\ \to (x : \mathsf{coeq}_{f,g}) \to P\ x\end{aligned}$$

Given a type $A : \mathsf{Type}$ and a relation $R : A \to A \to \mathsf{Type}$, we can define a type $\widetilde{R} :\equiv (x\ y : A) \times R\ x\ y$ which has two projections $\pi_0, \pi_1 : \widetilde{R} \to A$. We can then take the coequaliser of these two functions:

$$\widetilde{R} \underset{\pi_0}{\overset{\pi_1}{\rightrightarrows}} A \xrightarrow{c} \mathsf{coeq}_{\pi_0,\pi_1}$$

Proposition 3.1.3 ([AAL11]). *The coequaliser* $\mathsf{coeq}_{\pi_0,\pi_1}$ *is the quotient of* A *by* R.

3.1.3 Propositional truncation

For any type $A : \mathsf{Type}$, the propositional truncation of A is defined as the following quotient inductive type:

$$\begin{aligned}&\textbf{data } ||A|| : \mathsf{Type} \textbf{ where}\\ &\quad [\_] : A \to ||A||\\ &\quad \mathsf{trunc} : (x\ y : ||A||) \to x = y\end{aligned}$$

It can be shown that we indeed have is-prop $||A||$ for all types A, using proposition 2.2.18.

With propositional truncation we can define the notion of surjectiveness. Suppose we have a function $f : X \to Y$, then we define:

$$\text{is-surjective } f :\equiv (y : Y) \to ||(x : X) \times f\ x = y||$$

If we take the definition of is-surjective without the propositional truncation, then we can retrieve the inverse of f from any proof of is-surjective f. With the truncated version this is not possible in general.

Using this definition of surjectivity, we can now also show that the projection function of quotients ($[\_] : A \to A/R$) as defined in section 3.1.2 are surjections. We cannot in general show that we have inverses of the projection functions.

Using propositional truncation, we can formulate the following form of the axiom of choice:

Definition 3.1.4 (Axiom of choice)**.** Every surjection has a right inverse:

$$\begin{aligned}\text{axiom-of-choice} :\equiv\ & (X\ Y : \text{Type})\ (f : X \to Y) \\ \to\ & \text{is-surjective } f \to ||(g : Y \to X) \times ((y : Y) \to f\ (g\ y) = y)||\end{aligned}$$

The above form of choice with truncation is not provable in type theory, as opposed to the untruncated version.

Note that if A is a set, we can quotient it by the trivial relation and we get something equivalent to $||A||$. However, if A is not a set, this no longer holds. Instead, one has to repeatedly take quotients [Doo16; Kra15].

3.1.4 Infinitely branching trees

Except for the propositional truncation, the examples we have seen so far seem to be expressible as quotients of ordinary inductive types. Another example where we can observe a difference, even if we restrict ourself to the realm of sets, is with the type of infinitely branching trees modulo per-

mutation. We define the following inductive type:

```
data Tree : Set where
  leaf : Tree
  node : (ℕ → Tree) → Tree
  perm : (f : ℕ → Tree) (ϕ : ℕ → ℕ) → is-equiv ϕ → node f = node (f ∘ ϕ)
```

The perm constructor says that two nodes $f, f' : \mathbb{N} \to \mathsf{Tree}$ are considered to be equal if there exists a permutation ϕ such that $f = f' \circ \phi$.

We can try and define this type as a quotient of the type without path constructor perm:

```
data Tree₀ : Set where
  leaf₀ : Tree₀
  node₀ : (ℕ → Tree₀) → Tree₀
```

with the relation defined inductively as:

```
data _∼_ : Tree₀ → Tree₀ → Set where
  rel-node : (f : ℕ → Tree₀) (ϕ : ℕ → ℕ) → is-equiv ϕ → node₀ f ∼ node₀ (f ∘ ϕ)
```

Note that the relation as defined here is not an equivalence relation. It fails to be reflexive as we do not have a proof of $\mathsf{leaf}_0 \sim \mathsf{leaf}_0$. However, taking the quotient by a relation means that we quotient by the reflexive-transitive-symmetric closure of that relation. As such the relation as defined above is sufficient for our purposes.

If we then look at the "constructors" of the quotient $\mathsf{Tree}_0/\sim$, we have the following:

```
[ leaf₀ ] : Tree₀/ ∼
[ _ ] ∘ node₀ : (ℕ → Tree₀) → Tree₀/ ∼
```

The latter does not have the same type as node. With the quotient inductive

definition, the induction is performed "simultaneously" with the quotienting. This is also where Tree differs from our previous examples in which there were no recursive occurrences in the point constructors.

Lifting the node constructor to the quotient is problematic. It seems we need an inverse to the projection function $[\_] : \mathsf{Tree}_0 \to \mathsf{Tree}_0/\sim$, which we cannot expect to exist.

Note however that if we had *finitely* branching trees, we would not need such an inverse. In the finite case, we can apply the recursion principle to each argument and then apply the node_0 and $[\_]$ constructors.

In the infinitely branching case we can define the lifting of node if we have the axiom of choice at our disposal:

Proposition 3.1.5. *Assuming the axiom of choice, we can define a function* $(\mathbb{N} \to \mathsf{Tree}_0/\sim) \to \mathsf{Tree}_0/\sim$.

Proof. We can show that the function $[\_]$ is a surjection, i.e. using induction on $\mathsf{Tree}_0/\sim$ we can define a dependent function:

$$(y : \mathsf{Tree}_0/\sim) \to ||(x : \mathsf{Tree}_0) \times [\,x\,] = y||$$

The axiom of choice gives us:

$$||(s : \mathsf{Tree}_0/\sim \to \mathsf{Tree}_0) \times ((x : \mathsf{Tree}_0/\sim) \to [\,s\ x\,] = x)||$$

Since we only get this in truncated form, we have some more work to do. We are working with sets, hence if we want to eliminate out of a propositionally truncated type, it is enough to give a function out of the untruncated type and show that this is constant. In our case, we have to show that the following construction:

$$\widetilde{\mathsf{node}_0}\ f :\equiv [\,\mathsf{node}_0\ (s \circ f)\,]$$

is invariant under the choice of s. Now suppose we have two such functions $s, s' : \mathsf{Tree}_0/\sim \to \mathsf{Tree}_0$, such that for all $x : \mathsf{Tree}_0/\sim$, $[\,s\ x\,] = [\,s'\ x\,] = x$,

then for any $f : \mathbb{N} \to \mathsf{Tree}_0/\sim$, we have:

$$(x : \mathsf{Tree}_0/\sim) \to [\, s\ (f\ x)\,] = [\, s'\ (f\ x)\,] = f\ x$$

So far we have defined $\sim$ with just one constructor that relates nodes if there exists a permutation on $\mathbb{N}$ that makes them equal. This is not enough to show $\sim$ is an equivalence relation, nor that it is a congruence. When looking at the quotiented type, this is not a problem: we effectively quotient by the smallest equivalence and congruence relation generated by $\sim$. However, to complete this proof, we do in fact need that it is an equivalence relation and congruence relation, so we will assume in the remainder of this proof that we have added constructors to the definition of $\sim$ such that it is.

If $\sim$ is an equivalence relation, then $\mathsf{Tree}_0/\sim$ is an *effective quotient* [Hof95], i.e. we have:

$$(x\ y : \mathsf{Tree}_0) \to [\, x\,] = [\, y\,] \to x \sim y$$

We have for any $f : \mathbb{N} \to \mathsf{Tree}_0/\sim$ and $x : \mathbb{N}$ that $[\, s\ (f\ x)\,] = [\, s'\ (f\ x)\,]$ and therefore also $s\ (f\ x) \sim s'\ (f\ x)$. Since $\sim$ is a congruence relation, we then have that $\mathsf{node}_0\ (s \circ f) \sim \mathsf{node}_0\ (s' \circ f)$, hence $[\, \mathsf{node}_0\ (s \circ f)\,] = [\, \mathsf{node}_0\ (s' \circ f)\,]$. This establishes that our definition of $\widetilde{\mathsf{node}_0}$ is independent of the particular choice of s, which means we can eliminate out of the truncation $||(s : \mathsf{Tree}_0/\sim \to \mathsf{Tree}_0) \times ((x : \mathsf{Tree}_0/\sim) \to [\, s\ x\,] = x)||$ we get from the axiom of choice. □

In the presence of choice, it seems that $\mathsf{Tree}_0/\sim$ and Tree are equivalent. The axiom of choice in type theory is a constructive taboo in and of itself, it also has the law of excluded middle as a consequence [Dia75]. Going in the other direction, having an inverse to the quotient projection function in general implies the axiom of choice [Hof95]. However, it is still an open problem whether $\widetilde{\mathsf{node}_0}$ is definable without using the axiom of choice. Having quotient inductive types available allows us to work with definitions such as Tree, avoiding the need for choice principles to use them.

3.1.5 Cauchy reals

Another situation where we benefit from the ability to define an inductive type and "at the same time" quotient it is when defining the real numbers. If we want to define them as equivalence classes of Cauchy sequences of rational numbers, we traditionally run into problems. To show that this definition yields a version of the reals that is Cauchy complete, we need to have countable choice at our disposal.

Instead of taking that approach, one can define the reals as the following quotient *inductive-inductive* definition. We define the real numbers along with a closeness relation on it inductively and mutually:

$$\begin{aligned}&\textbf{data}\ \mathbb{R} : \mathsf{Set}\\ &\textbf{data}\ \_\sim\_\ \_ : \mathbb{Q}^+ \to \mathbb{R} \to \mathbb{R} \to \mathsf{Set}\end{aligned}$$

Note that $\mathbb{Q}^+$ denotes the type of positive rationals. The rationals can be defined as a quotient themselves or directly [AAL11]. $\mathbb{R}$ is inductively defined by the following constructors:

$$\begin{aligned}&\textbf{data}\ \mathbb{R}\ \textbf{where}\\ &\quad \mathsf{rat} : \mathbb{Q} \to \mathbb{R}\\ &\quad \mathsf{lim} : (f : \mathbb{Q}^+ \to \mathbb{R}) \to ((\delta\ \epsilon : \mathbb{Q}^+) \to f\ \delta \sim_{\delta+\epsilon} f\ \epsilon) \to \mathbb{R}\\ &\quad \mathsf{eq} : (u\ v : \mathbb{R}) \to ((\epsilon : \mathbb{Q}^+) \to u \sim_\epsilon v) \to u = v\end{aligned}$$

The first constructor rat is the inclusion of the rationals into the reals. The second constructor "adds" all the *limit points* for Cauchy approximations and the third (path) constructor eq tells us that any two real numbers are equal if they are arbitrarily close to each other via the relation we in-

ductively define simultaneously.

$$
\begin{aligned}
&\textbf{data}\ \_\sim_\epsilon\_\ \textbf{where}\\
&\quad \mathsf{rat\text{-}rat\text{-}eq} : (q\ r : \mathbb{Q})\ (\epsilon : \mathbb{Q}^+) \to -\epsilon < q - r < \epsilon \to \mathsf{rat}\ q \sim_\epsilon \mathsf{rat}\ r\\
&\quad \mathsf{rat\text{-}lim\text{-}eq} : (q : \mathbb{Q})\ (y : \mathbb{Q}^+ \to \mathbb{R})\ (\epsilon\ \delta : \mathbb{Q}^+)\ (t : (\delta\ \epsilon : \mathbb{Q}^+) \to y\ \delta \sim_{\delta+\epsilon} y\ \epsilon)\\
&\qquad \to \mathsf{rat}\ q \sim_{\epsilon-\delta} y\ \delta\\
&\qquad \to \mathsf{rat}\ q \sim_\epsilon \mathsf{lim}\ y\ t\\
&\quad \mathsf{lim\text{-}rat\text{-}eq} : (x : \mathbb{Q}^+ \to \mathbb{R})\ (r : \mathbb{Q})\ (\epsilon\ \delta : \mathbb{Q}^+)\ (s : (\delta\ \epsilon : \mathbb{Q}^+) \to x\ \delta \sim_{\delta+\epsilon} x\ \epsilon)\\
&\qquad \to x\ \delta \sim_{\epsilon-\delta} \mathsf{rat}\ r\\
&\qquad \to \mathsf{lim}\ x\ s \sim_\epsilon \mathsf{rat}\ r\\
&\quad \mathsf{lim\text{-}lim\text{-}eq} : (x\ y : \mathbb{Q}^+ \to \mathbb{R})\ (\epsilon\ \delta\ \eta : \mathbb{Q}^+)\\
&\qquad (s : (\delta\ \epsilon : \mathbb{Q}^+) \to x\ \delta \sim_{\delta+\epsilon} x\ \epsilon)\\
&\qquad (t : (\delta\ \epsilon : \mathbb{Q}^+) \to y\ \delta \sim_{\delta+\epsilon} y\ \epsilon)\\
&\qquad \to x\ \delta \sim_{\epsilon-\delta-\eta} y\ \eta\\
&\qquad \to \mathsf{lim}\ x\ s \sim_\epsilon \mathsf{lim}\ y\ t
\end{aligned}
$$

The constructors of the relation tell us when rational points are close, when a rational point is close to a limit point and when two limit points are close. For more details, we refer the reader to [Uni13].

3.1.6 Syntax of type theory

One of the classic examples of an inductive-inductive definition [Nor13] is the definition of the syntax of type theory in type theory itself [Dan06] [Cha09]. We see the need of a mutual/inductive-inductive definition already popping up when formalising the notion of contexts and types in a context:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Con} : \mathsf{Set}\\
&\textbf{data}\ \mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}
\end{aligned}
$$

Contexts are either empty or can be extended by a type in a given context:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Con}\ \textbf{where}\\
&\quad \epsilon : \mathsf{Con}\\
&\quad \_,\_ : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Con}
\end{aligned}
$$

Types in a context Γ can be defined inductively with constructors such as the following, i.e. we have constructors for atomic types and constructors for type formers such as Π-types, which gives us a type for any pair type A in context Γ and type B in the extended context Γ,A:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Ty}\ \textbf{where}\\
&\quad \text{`}0 : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma\\
&\quad \text{`}1 : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma\\
&\quad \vdots\\
&\quad \text{`}\Pi : (\Gamma : \mathsf{Con})\ (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Gamma,A) \to \mathsf{Ty}\ \Gamma\\
&\quad \vdots
\end{aligned}
$$

The constructor for Π-types is where we see one of the defining characteristics of inductive-inductive definitions: `Π refers to a previous constructor $\_,\_$. This sets it aside from mutual inductive definitions which only refer to the *elements* of the types being defined. While the latter can be reduced to ordinary inductive definitions, inductive-inductive definitions do not admit such a translation.

Continuing formalising the syntax of type theory, we need to have a type of terms:

$$\textbf{data}\ \mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}$$

For example, the constructors for lambda terms and the application of one

term to another may be given as follows:

```
data Tm where
  ⋮
  app : (Γ : Con) (A : Ty Γ) (B : Ty (Γ,A)) → Tm Γ (‘Π A B) → Tm (Γ,A) B
  lam : (Γ : Con) (A : Ty Γ) (B : Ty (Γ,A)) → Tm (Γ,A)B → Tm Γ (‘Π A B)
  ⋮
```

One important aspect of the syntax of type theory is that it comes with an equivalence relation on it defined by β- and η-equalities, i.e. the definitional or judgmental equality of the syntax. One way to deal with this is to separately define the equivalence relation on the terms and quotient by this relation. In the presence of quotient inductive-inductive definitions, we can simply add the equations as path constructors, as is done in [Kap16] [AK16]. For example, we can add the following constructor to Tm as a witness of β-equality for λ-terms:

$$\begin{aligned}\Pi\beta &: (\Gamma : \mathsf{Con})\ (A : \mathsf{Ty}\ \Gamma)\ (B : \mathsf{Ty}\ (\Gamma,A)) \\ &\quad \to (t : \mathsf{Tm}\ (\Gamma,A)\ B) \to \mathsf{app}\ \Gamma\ A\ B\ (\mathsf{lam}\ \Gamma\ A\ B\ t) = t\end{aligned}$$

3.2 Implementation

One way to “implement” a particular quotient inductive type, is by postulating its constructors along with the eliminator with its computation rules. The downside of such an approach is that as soon as we want to use it and not just define it, we will have to manually “call” the computation rules.

A first approximation to implement quotient inductive types (or any higher inductive type) in a more practical manner was discovered by [Lic11]. The idea is to leverage the inductive type mechanism of Agda itself and add path constructors and their computation rules as postulates. One can then define the eliminator by pattern matching on the inductive type, adding the computation rule for the path constructors as postulates. Care has to

be taken to not expose the constructors of the type in such a way that one can circumvent the eliminator and use pattern matching directly, in a possible unsound way.

Having definitional computation rules for the point constructors is a big improvement over everything being propositional. However, the path computation rules still only hold propositionally.

In order to experiment with higher inductive types and cubical type theory in Agda, rewrite rules have been introduced. We can denote any relation as being a rewrite relation, for example if we want to rewrite according to $\_ = \_$, we write:

$$\{-\#\ \mathsf{BUILTIN\ REWRITE}\ \_ = \_\ \#-\}$$

Suppose we have a term $p : x = y$ that we want to add as a definitional equality, i.e. we want Agda to always rewrite x to y, then we write:

$$\{-\#\ \mathsf{REWRITE\ p}\ \#-\}$$

Using these, a type such as the interval can be "implemented" as follows:

```
postulate
  I : Set
  zero : I
  one : I
  seg : zero = one
  I-rec : (A : Type) (a_zero : A) (a_one : A) (a_seg : a_zero = a_one)
    → I → A
  I-rec-β_zero : (A : Type) (a_zero : A) (a_one : A) (a_seg : a_zero = a_one)
    → I-rec A a_zero a_one a_seg zero = a_zero
  I-rec-β_one : (A : Type) (a_zero : A) (a_one : A) (a_seg : a_zero = a_one)
    → I-rec A a_zero a_one a_seg one = a_one
```

We can turn the β-equalities for the recursion principle into definitional ones using rewrite pragmas as follows:

```
{-# REWRITE I-rec-β_zero #-}
{-# REWRITE I-rec-β_one #-}
```

This approach has the advantage that Agda does not see I as an inductive type and does not recognise zero and one as constructors. Therefore we cannot accidentally pattern match on them and define functions that do not respect seg: I-rec is our only way to write (non-trivial) functions out of the interval.

As Agda does not perform checks on the rewrite rules, these rewrite pragmas are not safe in any way: we can easily add rules that make the type checker loop or that allow us to inhabit the empty type. Nonetheless they provide a good platform for experimentation.

3.2.1 Cubical type theory

As a way to study the computational behaviour of univalence and also higher inductive types, the cubical set model of type theory has been investigated [BCH14]. Turning this model back into type theory has led to cubical type theory [Coh+15], which has an implementation `cubicaltt`[1]. This implementation has some support for higher inductive types.

3.3 Related work

As mentioned in this chapter, quotient inductive-inductive definitions are related to quotient types. Quotient types have been studied in several forms in type theory [Hof95; AAL11; Li15].

Quotient inductive-inductive definitions are furthermore an extension of inductive-inductive definitions [Alt+11]. The idea of path constructors comes from the notion of higher inductive types, of which the semantics are described in [LS13b]. As we only consider set truncated types, the definitions we describe are in some sense a restriction of the higher inductive types described in that note. However, the authors do not consider the combination of higher inductive and inductive-inductive definitions.

Inductive definitions with equalities have been studied in Miranda [Tur85] under the name of "data types with laws" [Tho86] [Tho90]. In Miranda, the "path constructors" were interpreted as rewrite rules that would ensure that the inhabitants of an inductive type would be of a certain normal form or satisfy a certain property. For example, the type of sorted lists would be the usual list type along with a rewrite rule that rewrites $y :: x :: xs$ to $x :: y :: xs$ whenever $x < y$. However, data types with laws in Miranda were later abandoned as they made the usual equational reasoning about functions defined by pattern matching difficult. The idea was later explored further in the context of OCaml [BHW07].

With quotient inductive-inductive types we do not read the path constructors as rewrite rules. If we were to define a type of unordered pairs

[1] See `https://github.com/mortberg/cubicaltt`

P_X with values in some type X, having constructors $\mathsf{pair} : X \to X \to P_X$ and $\mathsf{swap} : (x\ y : X) \to \mathsf{pair}\ x\ y = \mathsf{pair}\ y\ x$, treating swap as a rewriting rule would make the system loop.

Chapter 4

Describing inductive definitions

In this chapter we will give a formal specification of quotient inductive-inductive definitions. The specification is given mutually with the interpretation of the specification as a category of algebras.

In order to specify an inductive definition, we have to give four sets of rules:

- *type formation rules*: we have to specify the *sorts* of the definition. For example, for the contexts and types definition we have a rule stating that Con is a type and that for every Γ : Con, Ty Γ is a type.
- *introduction rules*: we have to have means of creating instances of the inductive data we are defining: we need *constructors*.
- *elimination rules*: we also need means to inspect the data: we need an induction principle.
- *computation rules*: the elimination rules and introduction rules need to interact in a certain way.

As soon as we have the first two sets of rules, the latter two follow from these in a systematic way [Bac+89]. This is also reflected in how one declares an inductive definition in proof assistants such as Agda and Coq: we provide the type signature of the inductive definition (its type formation rules) and a list of constructors (its introduction rules).

There are several equivalent ways to characterise inductive definitions formally: we can formulate a syntax of constructors and describe the elimination principle by induction on terms in this syntax. This syntax is a subset of the syntax of type theory as a whole: we need to be able to write down a list of constructors which is just a sequence of types of a certain form. One can take an "analytic" approach to this: take the syntax of type theory and define a predicate on it that tells us whether a term is in the appropriate subset. This is effectively what systems like Agda do: you can write down a list of constructors and Agda checks whether it makes sense, e.g. whether the result types of the constructors are correct and whether the arguments are strictly positive. Doing this in type theory itself will be rather involved, as formalising the syntax of type theory in type theory is generally not something lightweight.

A more semantic approach is to start from the characterisation of inductive definitions as the initial algebra given some endofunctor on an appropriately chosen category. Ordinary inductive types can be explained as initial algebras of strictly positive endofunctors on $\mathcal{S}et$. These functors can be described as *containers* [AAG05], which can be easily formalised in type theory. The initiality property only uses the objects and morphisms of the category. Algebras of a container and algebra morphisms can be readily stated in type theory. It has been shown in type theory that for these inductive types, the property of being an initial algebra and that of satisfying the induction principle are logically equivalent [AGS12]. In this case this concretely means that having W-types available is the same as having initial algebras of containers.

From the initial algebra semantics we can directly read off the introduction rules along with the recursion principle and its computation rules. Algebras package the type along with its constructors. Algebra morphisms package the functions between the types along with a proof that the functions preserve the algebra structure. This categorical setting is on the one hand close to the syntax, but it also abstracts enough that it allows us to generalise without having to keep track of too many moving parts.

While the recursion principle can be straightforwardly recovered from

initiality, the induction principle is another story. Formulating the induction principle requires a bit more machinery, as does showing its logical equivalence to initiality. This is covered in chapter 5. In this chapter we will concern ourselves with uncovering what a description of a quotient inductive-inductive definition ought to be in a formal way.

We have seen in the examples in chapter 3 that quotient inductive-inductive definitions differ from ordinary inductive definitions in three ways:

- constructors may refer to any previous constructor
- we are not defining one type, but a collection of types and type families
- apart from point constructors we also have path constructors

The goal of this chapter is to find a notion of algebra that supports all of this. We start out by reviewing initial algebra semantics of ordinary inductive types in section 4.1, as well as generalisations of this . In section 4.2 we see how this approach can be generalised to support having constructors that refer to previous constructors by considering iterated dialgebras. Section 4.3 formalises the notion of *dependent sorts*, which deals with the type formation rules for quotient inductive-inductive definitions. In section 4.4 the full formal definition of declarations of quotient inductive-inductive definitions is given as a form of iterated dialgebras with equations. In this formal definition, we only consider certain functors for the result type of the constructors. In section 4.5 and section 4.6 we discuss the consequences of working with dialgebras, both the limitations as well as the opportunities for generalising the current definitions. Finally in section 4.7 we discuss related work.

4.1 Algebraic semantics

Let us look at a list of constructors of an ordinary data type, e.g.:

$$\begin{aligned}&\textbf{data } A : \mathsf{Set} \textbf{ where}\\ &\quad c_0 : F_0\ A \to A\\ &\quad c_1 : F_1\ A \to A\\ &\quad \vdots\\ &\quad c_k : F_k\ A \to A\end{aligned}$$

with every $F_i : \mathsf{Set} \to \mathsf{Set}$ being a function that gives us the arguments of the constructors. Note that we have to rewrite constructors with no arguments to $c : 1 \to A$ and have to uncurry constructors with multiple arguments. One observation is that if we have coproducts, we can rewrite the definition into one with just a single constructor:

$$\begin{aligned}&\textbf{data } A : \mathsf{Set} \textbf{ where}\\ &\quad c : F_0\ A + F_1\ A + \ldots + F_k\ A \to A\end{aligned}$$

For ordinary inductive types it is therefore sufficient to consider definitions with a single constructor. The type formation rule for such a definition is always the same: A is a type. The introduction rule is given by supplying a function $F : \mathsf{Set} \to \mathsf{Set}$, yielding the constructor $c : F\ A \to A$.

Moving on the recursion principle, we notice that we need more structure on the function F. The recursion principle gives us a way to define a function $A \to X$ for some type X, given some additional structure on X. Intuitively we need to specify for every constructor what the corresponding "constructor" in the target type of the recursion is. For example, for the natural numbers we have:

$$\mathbb{N}\text{-rec} : (X : \mathsf{Set})\ (\theta_0 : A)\ (\theta_1 : X \to X) \to \mathbb{N} \to X$$

with computation rules giving us equalities $\mathbb{N}\text{-rec}\ X\ \theta_0\ \theta_1\ \mathsf{zero} = \theta_0$ and

$\mathbb{N}\text{-rec}\ X\ \theta_0\ \theta_1\ (\mathsf{succ}\ n) = \theta_1\ (\mathbb{N}\text{-rec}\ A\ \theta_0\ \theta_1\ n)$. These rules tell us that applying $\mathbb{N}\text{-rec}$ to constructors of $\mathbb{N}$ is the same as first applying $\mathbb{N}\text{-rec}$ to the recursive arguments and then applying the "constructors" of the target of the recursion.

In order to generalise the recursion principle and its computation rules to any inductive type A with constructor $c : F\ A \to A$, we need F to be a functor. We can then state the recursion principle for A as:

$$A\text{-rec} : (X : \mathsf{Set})\ (\theta : F\ X \to X) \to A \to X$$

with as its computation rule:

$$A\text{-rec}\ X\ \theta\ (c\ x) = \theta\ (F\ (A\text{-rec}\ X\ \theta)\ x)$$

The type A along with its constructor c and the X and θ in the recursion principle are both F-algebras, where F is an endofunctor on Set. The notion of F-algebra can be defined for any category $\mathcal{C}$:

Definition 4.1.1. Given an endofunctor F on $\mathcal{C}$, an *F-algebra* is an object $X : |\mathcal{C}|$ along with an *F-algebra structure* $\theta : \mathcal{C}(FX, X)$. The object X is also referred to as the *carrier* or *underlying object* of the algebra.

The function $A\text{-rec}\ X\ \theta$ is a function between the carriers of the algebras (A, c) and (X, θ). Its computation rule states that it respects the algebra structure: it is an algebra morphism:

Definition 4.1.2. Given F-algebras (X, θ) and (Y, ρ) an *F-algebra morphism* consists of a morphism $f : \mathcal{C}(X, Y)$ such that the following commutes:

$$\begin{array}{ccc} FX & \xrightarrow{\theta} & X \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FY & \xrightarrow{\rho} & Y \end{array}$$

Algebra morphisms can be composed and we have identity algebra morphisms: algebras form a category:

Definition 4.1.3. The category F-alg is the category with F-algebras as objects and F-algebra morphisms as morphisms.

4.1.1 Monad algebras

Given an endofunctor F on $\mathcal{S}et$, we can consider its *free monad* F^*, which is defined pointwise as the carrier of the initial algebra of the functor $\bar{F}_X Y :\equiv X + FY$. Note that the free monad need not exist: F is required to be a strictly positive functor for $\bar{F}_X$ to be strictly positive as well.

We can write the functor F^* down as a parametrised inductive type, with $X : \mathsf{Set}$ as its parameter:

$$\begin{aligned} &\mathbf{data}\ F^*X : \mathsf{Set}\ \mathbf{where} \\ &\quad \eta : X \to F^*X \\ &\quad c : F(F^*X) \to F^*X \end{aligned}$$

Note that if we want to write down the abovementioned definition in Agda, we have to make sure that F is a strictly positive functor. We cannot have the definition be parametric in the functor F.

Proposition 4.1.4. *F^*, if it exists, is a monad on $\mathcal{S}et$*

Proof. We show that it is a monad by showing that F^* gives us a left adjoint to the forgetful functor $U : F\text{-}\mathsf{alg} \Rightarrow \mathcal{S}et$ where $U(X, \theta) :\equiv X$. We define the left adjoint $L : \mathcal{S}et \Rightarrow F\text{-}\mathsf{alg}$ as follows:

$$LX :\equiv (F^*X, c_X)$$

We then show that for any $X : \mathsf{Set}$ and $(Y, \rho) : |F\text{-}\mathsf{alg}|$ that

$$F\text{-}\mathsf{alg}(LX, (Y, \rho)) = (X \to Y)$$

using the equational reasoning in fig. 4.1.

This shows that $L \dashv U$ and since $UL = F^*$, that therefore F^* is a monad. □

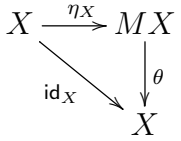
$$
\begin{aligned}
& X \to Y \\
= & \{ \text{ initiality of } F^*X \ \} \\
& (p : X \to Y) \times \bar{F}_X\text{-alg}((F^*X, c_X, \eta_X), (Y, \rho, p)) \\
= & \{ \text{ definition } \bar{F}_X\text{-algebra morphism } \} \\
& (p : X \to Y) \times (f : F^*X \to Y) \times (f_0 : f \circ c_X = \rho \circ Ff) \times (f_1 : f \circ \eta_X = p) \\
= & \{ \text{ singleton contraction of } p \text{ and } f_1 \ \} \\
& (f : F^*X \to Y) \times (f_0 : f \circ c_X = \rho \circ Ff) \\
= & \{ \text{ definition of } F\text{-algebra morphisms and } L \ \} \\
& F\text{-alg}(LX, (Y, \rho))
\end{aligned}
$$

Figure 4.1: F^* is a monad

Note that since L is a left adjoint it preserves colimits, in particular initial objects, hence $L\emptyset$ is the initial object of F-alg. The carrier of $L\emptyset$ is $F^*\emptyset$, which is the inductive type as defined by the endofunctor F.

F^* is called the *free* monad as it is a free object with regards to the forgetful functor of the category of monads on $\mathcal{S}et$ to endofunctors on $\mathcal{S}et$. The free monad F^* allows us to make precise what the relationship between monads and algebras of an endofunctor is. To this end, let us first recall the definition of *monad algebras*:

Definition 4.1.5. Given a monad $M : \mathcal{C} \Rightarrow \mathcal{C}$ with $\eta : 1_{\mathcal{C}} \dot{\to} M$ its unit and $\mu : M^2 \dot{\to} M$ is multiplication, a monad algebra on M is an object $X : |\mathcal{C}|$ and a morphism $\theta : \mathcal{C}(MX, X)$ such that it respects the monad operations, i.e. the following commutes:



and

$$\begin{array}{ccc} M^2X & \xrightarrow{M\theta} & MX \\ {\scriptstyle \mu_X}\downarrow & & \downarrow{\scriptstyle \theta} \\ MX & \xrightarrow[\theta]{} & X \end{array}$$

Monad algebra morphisms are defined in the same way as morphisms of algebras of an endofunctor.

Monad algebras and monad algebra morphisms form a category, denoted M-Alg, also called the *Eilenberg-Moore category* of M.

Theorem 4.1.6 ([GK13])**.** *Let $F : \mathcal{C} \Rightarrow \mathcal{C}$ be an endofunctor on $\mathcal{C}$ with $F^* : \mathcal{C} \Rightarrow \mathcal{C}$ its free monad, then F^*-Alg is equivalent to F-alg.*

This theorem shows us that we can think of the class free monads being the same as the class of ordinary inductive types. This suggests that generalising ordinary inductive types amounts to carving out a class of monads that includes free monads. In [Shu11b] the author proposes that a higher inductive type should correspond to a notion of *presented monad*, i.e. a monad that is defined in terms of generators and relations, or in HIT terminology: point and path constructors. In [LS13b], the authors give semantics of HITs by constructing these presented monads. As these constructions are done in a wide class of model categories, it is a construction external to type theory.

As opposed to constructing the monads directly, we will first define the categories of algebras in this chapter. The reason for this is twofold. In the case of algebras of an endofunctor we have fewer equations to work with: they only pop up in the definition of algebra morphisms. The definition of monads already requires us to talk about functors that satisfy certain equalities. Monad algebras also come with equations. Generalising from the endofunctor situation, we hope that we also do not have to introduce equations, which might lead to more coherence issues.

The other reason why we chose to do it this way is that it allows us to talk about algebras (the "syntax") separately from their existence ("semantics"). In this chapter we give a definition of the algebras and in chapter 6 we

show how one can construct initial algebras and left adjoints to the forgetful functors of categories of algebras.

4.2 $\mathcal{S}et$-sorted inductive-inductive definitions

We have mentioned that one of the defining features of the inductive definitions we want to be able to handle, is any constructor may refer to any of its previous constructors. Some examples we have seen of this phenomenon are:

- the constructor seg : zero = one of the interval type, I, which refers to both its previous constructors,
- the constructor ‘Π : (Γ : Con) (A : Ty Γ) → Ty (Γ,A) → Ty Γ, which refers to the previous constructor _,_ : (Γ : Con) → Ty Γ → Con.

Describing inductive definitions with such constructors is not something we can do with a single endofunctor on an appropriately chosen category. Instead of having one functor, we need a functor for every constructor. The domain of the n-th functor will then be the "category of algebras of all previous constructors".

Example 4.2.1. To illustrate this idea, we consider the following example (which we can easily show to be equivalent to the booleans):

$$\begin{aligned}&\textbf{data } T : \mathsf{Set} \textbf{ where}\\ &\quad a : 1 \to T\\ &\quad b : (t : T) \times a\ \mathsf{tt} = t \to T\end{aligned}$$

The first constructor a can be described as being an F_0-algebra structure on T, with $F_0 : \mathcal{S}et \Rightarrow \mathcal{S}et$ being defined as $F_0 X :\equiv 1$. The arguments of b are described by the functor $F_1 : F_0\text{-alg} \Rightarrow \mathcal{S}et$ with:

$$F_1(X, a) :\equiv (x : X) \times a\ \mathsf{tt} = x$$

Since F_1 is not an endofunctor, b cannot be described as an F_1-algebra structure. However, F_0-alg comes with a forgetful functor $U_0 : F_0\text{-alg} \Rightarrow \mathcal{S}et$ which gives us the carrier of the algebra. We therefore have that $b : F_1(T, a) \to U_0(T, a)$, i.e. b is an (F_1, U_0)-*dialgebra* [Hag87] structure on $(T, a) : F_0$-alg.

Definition 4.2.2. Let $F, G : \mathcal{C} \to \mathcal{D}$ be functors. The category (F, G)-dialg has as objects pairs (X, θ) where $X : |\mathcal{C}|$ and $\theta : FX \to GX$. A morphism from (X, θ) to (Y, ρ) is a morphism $f : X \to Y$ in $\mathcal{C}$ such that the following commutes:

$$\begin{array}{ccc} FX & \xrightarrow{\theta} & GX \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow{\rho} & GY \end{array}$$

Remark 4.2.3. Any dialgebra category (F, G)-dialg comes with a forgetful functor $V : (F, G)\text{-dialg} \to \mathcal{C}$ which projects out the carrier of the algebra.

Recall the example T with constructors a and b, where we have given functors $F_0 : \mathcal{S}et \Rightarrow \mathcal{S}et$ and $F_1 : F_0\text{-alg} \Rightarrow \mathcal{S}et$. We have a forgetful functor $V_1 : (F_1, U_0)\text{-dialg} \to F_0\text{-alg}$ which we can compose with the forgetful functor $U_0 : F_0\text{-alg} \Rightarrow \mathcal{S}et$, which we will denote as U_1. If we were to add a third and a fourth constructor, we would have to define functors $F_2 : (F_1, U_0)\text{-dialg} \Rightarrow \mathcal{S}et$ and $F_3 : (F_2, U_1)\text{-dialg} \Rightarrow \mathcal{S}et$ to describe their arguments. The objects we are interested in are *iterated* dialgebras. For the situation described above we have:

$$\begin{array}{ccccccccc} \mathcal{S}et & \xleftarrow{U_0} & F_0\text{-alg} & \xleftarrow{V_1} & (F_1, U_0)\text{-dialg} & \xleftarrow{V_2} & (F_2, U_1)\text{-dialg} & \xleftarrow{V_3} & (F_3, U_2)\text{-dialg} \\ {\scriptstyle F_0}\downarrow & & {\scriptstyle F_1}\downarrow & & {\scriptstyle F_2}\downarrow & & {\scriptstyle F_3}\downarrow & & \\ \mathcal{S}et & & \mathcal{S}et & & \mathcal{S}et & & \mathcal{S}et & & \end{array}$$

Note that we build the category of dialgebras over another category of dialgebras, i.e. we are working with *iterated* dialgebras. A constructor has arguments described by a functor out of the category of algebras containing all the previous constructors. We can formalise the concept $\mathcal{S}et$-sorted inductive-inductive definitions as a inductive-recursive type, defining the specification and the interpretation of the specification as a category of algebras simultaneously:

Definition 4.2.4. The type Spec of specifications of a $\mathcal{S}et$-sorted inductive-inductive definition and its category of algebras (Alg) and underlying carrier functor (U) is given by the following inductive-recursive definition of

$$\begin{aligned}
&\mathbf{data}\ \mathsf{Spec} : \mathsf{Set}\\
&\mathsf{Alg} : \mathsf{Spec} \to \mathsf{Cat}\\
&\mathsf{U} : (s : \mathsf{Spec}) \to \mathsf{Alg}_s \Rightarrow \mathcal{S}et
\end{aligned}$$

where Spec is inductively generated by

$$\begin{aligned}
&\mathbf{data}\ \mathsf{Spec}\ \mathbf{where}\\
&\quad \mathsf{nil} : \mathsf{Spec}\\
&\quad \mathsf{snoc} : (s : \mathsf{Spec}) \to (F : \mathsf{Alg}_s \Rightarrow \mathcal{S}et) \to \mathsf{Spec}
\end{aligned}$$

and Alg and U are defined by

$$\begin{aligned}
&\mathsf{Alg}_{\mathsf{nil}} :\equiv \mathcal{S}et\\
&\mathsf{Alg}_{(\mathsf{snoc}\ s\ F)} :\equiv (F, \mathsf{U}_s)\text{-dialg}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{U}_{\mathsf{nil}} :\equiv \mathsf{id}\\
&\mathsf{U}_{(\mathsf{snoc}\ s\ F)} :\equiv U_s \circ V
\end{aligned}$$

where V is the forgetful functor that gives the underlying object of a dialgebra.

As Spec is essentially a list type, we will use list notation for its values, e.g. we denote snoc (snoc nil F_0) F_1 as $[F_0, F_1]$.

Example 4.2.5. The data type given in example 4.2.1 can be represented by the specification $[F_0, F_1]$ where F_0 and F_1 are the same functors as before, as we have that $\mathcal{S}et \equiv \mathsf{Alg}_{[]}$, F_0-alg $\equiv \mathsf{Alg}_{[F_0]}$ and $(F_1, \mathsf{U}_{[F_0]})$-dialg $\equiv \mathsf{Alg}_{[F_0,F_1]}$.

Example 4.2.6. The natural numbers can be represented in the usual way with one endofunctor $FX :\equiv 1 + X$. In our framework, however, we can

treat the constructors separately, staying a bit closer to the syntax. We define:

$$
\begin{aligned}
&F_0 : \mathcal{S}et \Rightarrow \mathcal{S}et \\
&F_0\ X :\equiv 1 \\
&F_1 : \mathsf{Alg}_{[F_0]} \Rightarrow \mathcal{S}et \\
&F_1\ (X, \theta_0) :\equiv X
\end{aligned}
$$

Unfolding definitions, we have $\mathsf{Alg}_{[F_0,F_1]} = (F_1, \mathsf{U}_{[F_0]})$-dialg and $\mathsf{Alg}_{[F_0]} = (F_0, \mathsf{id})$-dialg $= F_0$-alg. The objects of $\mathsf{Alg}_{[F_0,F_1]}$ are then:

$$
\begin{aligned}
\mathsf{Alg}_{[F_0,F_1]} &= (X : |\mathsf{Alg}_{[F_0]}|) \times (\theta : F_1 X \to \mathsf{U}_{[F_0]} X) \\
&= (X : \mathsf{Set}) \times (\theta_0 : 1 \to X) \times (\theta_1 : X \to X)
\end{aligned}
$$

For the morphisms of $\mathsf{Alg}_{[F_0,F_1]}$, we have for algebras $(X, \theta_0, \theta_1), (Y, \rho_0, \rho_1)$:

$$
\begin{aligned}
&\mathsf{Alg}_{[F_0,F_1]}((X, \theta_0, \theta_1), (Y, \rho_0, \rho_1)) \\
&= (f : \mathsf{Alg}_{[F_0]}((X, \theta_0), (Y, \rho_0))) \times (f_1 : \mathsf{U}_{[F_0]} f \circ \theta_1 = \rho_1 \circ F_1 f) \\
&= (f : X \to Y) \times (f_0 : f \circ \theta_0 = \rho_0) \times (f_1 : f \circ \theta_1 = \rho_1 \circ f)
\end{aligned}
$$

4.2.1 Avoiding induction-recursion

The type of specifications in definition 4.2.4 is given inductive-recursively. However, we want our framework to be implementable in a small core type theory. In such a setting, one would not expect to have induction-recursion available. A class of inductive-recursive definitions can be translated into definitions making use of indexed inductive definitions instead [Mal+12]. In this section we will use a different translation, which does not need indexed inductive definitions.

Intuitively the type is just a snoc-list of functors. The induction-recursion allows us to succinctly make sure that the domain of the functors is always a category of algebras. We can avoid induction-recursion by separately defining the snoc-list of functors and a predicate on that list that ensures

the domain of the functors is correct:

Definition 4.2.7.

```
data Spec' : Set where
  nil : Spec'
  snoc : Spec' → (𝒞 : Cat) (𝒞 ⇒ 𝒮et) → Spec'
```

On this type, we define a predicate mutually with its interpretation function Alg' with forgetful functor U':

```
is-correct : Spec' → Set
Alg' : (s : Spec') × (is-correct s) → Cat
U' : (s : Spec') × (p : is-correct s) → Alg'₍s,p₎ ⇒ 𝒮et
```

where

```
is-correct nil :≡ 1
is-correct (snoc s 𝒞 F) :≡ (p : is-correct s) × (𝒞 = Alg'₍s,p₎)
```

The definitions of Alg′ and U′ are similar to the previous definitions: we can pattern match on the equality proofs we get from is-correct and then use the previous definitions.

Remark 4.2.8. The mutual definition of is-correct, Alg′ and U′ can be avoided by combining all three definitions into one function with all its arguments and result types combined in a big Σ-type.

Proposition 4.2.9. *The types* Spec *and* $(s : \text{Spec'}) \times \text{is-correct}\ s$ *are equivalent.*

Proof. This is a straightforward proof by induction on the types involved and applying singleton contraction where needed. □

4.3 Dependent sorts

In the last section we saw how we can deal with referring to previous constructors. In this section we will tackle the issue of generalising the form of the type formation rules, which were hitherto just of the form "A is a set". We want to generalise this to arbitrary lists of types and type families. Such a list is referred to as the *dependent sorts* of the inductive definition. Examples can be something simple like $A : \mathsf{Set}, B : A \to \mathsf{Set}$, which we have seen in the example of contexts and type in a context in section 3.1.6, or something more involved like: $A : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}, B : (x : \mathbb{N}) \to A\ n\ n \to \mathsf{Set}$.

Describing how a sort depends on the previous sorts can be done by providing a functor of the previous category of sorts into $\mathcal{S}et$. A complete description is then a snoc-list of functors, which can be formalised inductive-recursively together with the function that interprets the list as a category.

Definition 4.3.1. The specification of sorts and their interpretation as a category is given by the following inductive-recursive definition

$$\begin{aligned}&\mathbf{data}\ \mathsf{Sorts} : \mathsf{Set}\\ &[\![\ \_\ ]\!] : \mathsf{Sorts} \to \mathsf{Cat}\end{aligned}$$

where Sorts is inductively generated by

$$\begin{aligned}&\mathbf{data}\ \mathsf{Sorts}\ \mathbf{where}\\ &\quad \mathsf{nil} : \mathsf{Sorts}\\ &\quad \mathsf{snoc} : (\mathcal{S} : \mathsf{Sorts}) \to (R : [\![\ \mathcal{S}\ ]\!] \Rightarrow \mathcal{S}et) \to \mathsf{Sorts}\end{aligned}$$

with $[\![\ \mathsf{nil}\ ]\!]$ defined as the terminal category $\mathbb{1}$, and given $\mathcal{S} : \mathsf{Sorts}$ and $R : [\![\ \mathcal{S}\ ]\!] \Rightarrow \mathcal{S}et$, the category $[\![\ \mathsf{snoc}\ \mathcal{S}\ R\ ]\!]$ has:

- objects: $(X : |[\![\ \mathcal{S}\ ]\!]|) \times (RX \to \mathsf{Set})$,
- morphisms $(X, Z) \to (Y, W)$ consist of
  - a morphism $f : [\![\ \mathcal{S}\ ]\!](X, Y)$

– a dependent function $g : (x : RX) \to Z\ x \to W\ (R\ f\ x)$.

Remark 4.3.2. The use of induction-recursion can be avoided here as well, using the same techniques as in section 4.2.1

Example 4.3.3. The sort of an ordinary inductive definition can be represented by the list $[R_0]$ (i.e. snoc nil R_0) where $R_0 : \mathbb{1} \Rightarrow \mathcal{S}et$ is defined as the constant functor $R_0\ x :\equiv 1$. The resulting category $[\![\ R_0\ ]\!]$ has objects $(\mathsf{tt} : 1) \times (A : 1 \to \mathsf{Set})$ and a morphism $(x, A) \to (y, B)$ is given by, since trivially $x = y = \mathsf{tt}$, a trivial morphism $1 \to 1$ together with a function $f\ x : A\ x \to B\ x$ for every $x : 1$. In other words, this category is equivalent to the category $\mathcal{S}et$.

In the above example it may seem superfluous to have the empty list interpreted as the terminal category and not as $\mathcal{S}et$. However, this choice allows us to have the first sort be indexed by some other type, e.g. $\mathbb{N}$. Hence a definition of the vectors would have as sort specification the list $[R_0]$ with $R_0\ x :\equiv \mathbb{N}$.

Example 4.3.4. The sort of the context and types example $(\mathsf{Con}, \mathsf{Ty})$ can be represented by the list $[R_0, R_1]$ with

- $R_0 : \mathbb{1} \Rightarrow \mathcal{S}et$, $R_0\ x :\equiv 1$
- $R_1 : [\![\ R_0\ ]\!] \Rightarrow \mathcal{S}et$, $R_1(x, A) :\equiv A\ x$

The category $[\![\ R_0, R_1\ ]\!]$ has objects $(x : 1) \times (A : 1 \to \mathsf{Set}) \times (B : A\ x \to \mathsf{Set})$. We see that this category is equivalent to the category $\mathcal{F}am$ of families of sets.

Example 4.3.5. Similarly, the category $\mathcal{R}el$ can be represented by the list $[R_0, R_1]$ with

- $R_0 : \mathbb{1} \Rightarrow \mathcal{S}et$, $R_0\ x :\equiv 1$
- $R_1 : [\![\ R_0\ ]\!] \Rightarrow \mathcal{S}et$, $R_1(x, A) :\equiv A\ x \times A\ x$

We see that $[\![\ R_0, R_1\ ]\!]$ is equivalent to the category with objects $(X : \mathsf{Set}, R : X \to X \to \mathsf{Set})$ and morphisms $(X, R) \to (Y, S)$ given by $f : X \to Y$ together with $g : (x\ y : X) \to R\ x\ y \to S\ (f\ x)\ (f\ y)$.

4.3.1 Sort membership

When we have multiple sorts, we need a way to select a particular sort from this collection. When defining a constructor, we first have to say what its sort is. To this end we define in this section a sort membership relation.

Given a specification $\mathcal{S}$: Sorts and a functor $R : [\![\ \mathcal{S}\ ]\!] \Rightarrow \mathcal{S}et$, we can define a forgetful functor $t : [\![\ \mathsf{snoc}\ s\ R\ ]\!] \Rightarrow [\![\ \mathcal{S}\ ]\!]$ which maps an object $(X, P) : [\![\ \mathsf{snoc}\ s\ R\ ]\!]$ to X. A specification $\mathcal{S} = [R_0, \ldots, R_n] : \mathsf{Sorts}$ therefore defines a chain of categories:

$$\mathbb{1} \xleftarrow{t_0} S_0 \xleftarrow{t_1} S_1 \xleftarrow{t_2} \ldots \xleftarrow{t_n} S_n$$

where n is the number of functors in the list and $S_i = [\![\ R_0, \ldots, R_i\ ]\!]$ is the category of sorts truncated to the first i elements. Every t_i is the forgetful functor into the previous category.

Example 4.3.6. In the case of $\mathcal{R}el$, we get the sequence

$$\mathbb{1} \xleftarrow{t_0} \mathcal{S}et \xleftarrow{t_1} \mathcal{R}el$$

We define a membership relation

$$\_ \in \_ : \mathsf{Cat} \to \mathsf{Sorts} \to \mathsf{Set}$$

where $\mathcal{C} \in \mathcal{S}$ means that $\mathcal{C}$ is one of the S_i in the chain $S_0 \leftarrow \ldots \leftarrow S_n$. We do not want $\mathbb{1} \in \mathcal{S}$, as adding a constructor of that sort does not add anything to the inductive definition.

Definition 4.3.7 (Sort membership)**.** We formalise the membership relation as the following inductive type:

$$\begin{aligned}
&\textbf{data}\ _ \in _ : \mathsf{Cat} \to \mathsf{Sorts} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{here} : (\mathcal{S} : \mathsf{Sorts})\ (R : [\![\ \mathcal{S}\ ]\!] \Rightarrow \mathcal{S}et) \to [\![\ \mathsf{snoc}\ \mathcal{S}\ R\ ]\!] \in \mathsf{snoc}\ \mathcal{S}\ R\\
&\quad \mathsf{there} : (\mathcal{S} : \mathsf{Sorts})\ (R : [\![\ \mathcal{S}\ ]\!] \Rightarrow \mathcal{S}et)\ (\mathcal{C} : \mathsf{Cat}) \to \mathcal{C} \in \mathcal{S} \to \mathcal{C} \in \mathsf{snoc}\ \mathcal{S}\ R
\end{aligned}$$

Example 4.3.8. Consider the specification of $\mathcal{Rel}$ (example 4.3.5). If we want to say that $[\![\ [R_0]\ ]\!] \in [R_0, R_1]$ (recall that $[\![\ [R_0]\ ]\!]$ is equivalent to $\mathcal{S}et$), then we can do so by giving the following term:

$$\mathsf{there}\ [R_0]\ R_1\ [\![\ R_0\ ]\!]\ (\mathsf{here}\ \mathsf{nil}\ R_0) : [\![\ [R_0]\ ]\!] \in [R_0, R_1]$$

Suppose we have a chain of sorts $S_0 \leftarrow S_1 \leftarrow \ldots \leftarrow S_n$ and a functor $\mathsf{U} : \mathcal{C} \Rightarrow S_n$, then naturally we can extend this functor to $\widehat{\mathsf{U}} : \mathcal{C} \Rightarrow S_i$ for any S_i in the chain by composing with the forgetful functors. We can implement this operation with our inductive definition of sort membership:

Definition 4.3.9 (Extending functors along sort membership)**.** For every specification $\mathcal{S} : \mathsf{Sorts}$ with a functor $\mathsf{U} : \mathcal{C} \Rightarrow [\![\ \mathcal{S}\ ]\!]$, we define the function:

$$\widehat{\mathsf{U}} : (S_i : \mathsf{Cat}) \to S_i \in \mathcal{S} \to \mathcal{C} \Rightarrow S_i$$

This function is defined by induction over the proof of $S_i \in \mathcal{S}$.

For the action of $\widehat{\mathsf{U}}$ on objects and morphisms we will usually leave the membership proof argument implicit, e.g. we just write $\widehat{\mathsf{U}}X$ and $\widehat{\mathsf{U}}f$.

4.3.2 Makkai's dependent sorts

As mentioned before, giving an inductive definition is similar to defining an equational theory: we specify the type formation rules, or the sorts, we specify the point constructors, or the function symbols and we specify the path constructors, or the equations. The inductive types themselves are the term models of the theory described by the constructors. One important aspect of our inductive definitions is that our type formation rules do not just give us a collection of types, but also type *families*: we do have *dependent sorts*.

In Makkai's FOLDS (first-order logic with dependent sorts) [Mak95], these dependent sorts are represented as presheaves over certain categories. Observe that the category of families $\mathcal{F}am$ can either be defined as having objects $(X : \mathsf{Set}) \times (P : X \to \mathsf{Set})$ or as having as objects $(X\ Y : \mathsf{Set}) \times (p :$

$Y \to X$). The latter category can be explained as the presheaf category $\mathcal{S}et^I$, where I is the arrow category $\cdot \to \cdot$.

Example 4.3.10. Suppose we have an inductive definition of a category, which has sorts

- O : Set
- $A : O \to O \to$ Set (a family of arrows)
- $T : (x\ y\ z : O) \to A\ x\ y \to A\ y\ z \to A\ x\ z \to$ Set (a family of triangles)

The corresponding category of sorts can be represented as presheaves over the category:

$$\cdot \rightrightarrows \cdot \;\substack{\rightarrow\\ \rightarrow\\ \rightarrow}\; \cdot$$

In Makkai, they consider dependent sorts to be specified by presheaves on a *direct category*.

Definition 4.3.11 (Direct category)**.** A category $\mathcal{C}$ is *direct* if $\mathcal{C}$ contains no infinite descending chain of non-identity morphisms.

Intuitively this means that all the arrows go in the "same direction" and that there are no non-trivial automorphisms.

If a category $\mathcal{C}$ is such that $\mathcal{C}^{\mathrm{op}}$ is direct, then $\mathcal{C}$ is an *inverse category*. Considering presheaves on a direct category is the same as considering functors of an inverse category into $\mathcal{S}et$.

There is also no finiteness restriction on direct categories, so one could also talk about presheaves from the simplex category, without the degeneracies, Δ^+:

$$\cdot \rightrightarrows \cdot \;\substack{\rightarrow\\ \rightarrow\\ \rightarrow}\; \cdot \;\substack{\rightarrow\\ \rightarrow\\ \rightarrow\\ \rightarrow}\; \cdots$$

Our definition of sorts does not support infinitely many sorts, however one could take a coinductive interpretation of the Sorts type.

Some sorts do not translate easily to the presheaf approach. Take for example the sorts:

- A : Set

- B : List $A \to$ Set

With the Sorts datatype, we are allowed to use arbitrary functors, so this readily translates to that setting. In the presheaf setting, there is no such a straightforward translation possible.

We have chosen our specific definition of sorts in this way as it follows our syntax closely. Most importantly, the sort membership relation can be defined easily and used in a straightforward manner.

4.3.3 Sort categories via comma categories

Another perspective on sort categories is seeing them as comma categories. Recall that the category $\mathcal{F}am$ can be seen as the arrow category $\mathcal{S}et^I$. $\mathcal{S}et^I$ is a presheaf category having as objects functors $I \Rightarrow \mathcal{S}et$. Alternatively, this arrow category is the comma category $\mathcal{S}et \downarrow \mathcal{S}et$. By considering other functors into $\mathcal{S}et$ on the right of this comma category instead of just the identity functor, we arrive at our sort categories. For example, suppose we have the following sequence of sort categories:

$$\begin{array}{ccccccc} \mathbb{1} & \xleftarrow{t_0} & S_0 & \xleftarrow{t_1} & S_1 & \xleftarrow{t_2} & S_2 \\ \downarrow{\scriptstyle R_0} & & \downarrow{\scriptstyle R_1} & & \downarrow{\scriptstyle R_2} & & \\ \mathcal{S}et & & \mathcal{S}et & & \mathcal{S}et & & \end{array}$$

where R_i are the defining functors of the sorts. We have that $S_0 = \mathcal{S}et \downarrow R_0$, $S_1 = \mathcal{S}et \downarrow R_1$ and $S_2 = \mathcal{S}et \downarrow R_2$.

4.4 Categories of algebras

Now that we have a way of specifying the type formation rules of an inductive definition, we can talk about the introduction rules. We have also seen how to deal with constructors referring to previous constructors in a $\mathcal{S}et$-sorted setting. We move on to the final two pieces of the puzzle: constructors of arbitrary sort and path constructors.

4.4.1 A $\mathcal{R}el$-sorted quotient inductive-inductive type

As a warm-up we will give an example of how the specification and the categories of algebras look like for the following quotient inductive-inductive definition:

$$\begin{aligned}
&\textbf{data } A : \mathsf{Set} \\
&\textbf{data } B : A \to A \to \mathsf{Set}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data } A \textbf{ where} \\
&\quad c_0 : A \\
&\quad c_1 : A
\end{aligned}$$

$$\begin{aligned}
&\textbf{data } B \textbf{ where} \\
&\quad c_2 : \mathbb{N} \to B\ c_0\ c_1 \\
&\quad c_3 : (n : \mathbb{N}) \to c_2\ n = c_2\ (n + 1)
\end{aligned}$$

In this section we will show step by step, i.e. constructor by constructor, how the category of algebras of the above inductive definition is built up. We will get a chain of categories:

$$\mathsf{Alg}_0 \xleftarrow{V_0} \mathsf{Alg}_1 \xleftarrow{V_1} \mathsf{Alg}_2 \xleftarrow{V_2} \mathsf{Alg}_3 \xleftarrow{V_3} \mathsf{Alg}_4$$

where Alg_i is the category that "contains" the first i constructors. The functors V_i are the forgetful functors. Alg_4 is the category we are ultimately interested in: this is the category of algebras that is associated with the full inductive definition.

Sorts: $(A : \mathsf{Set}) \times (B : A \to A \to \mathsf{Set})$

The category of sorts is the category $\mathcal{R}el$ of (proof-relevant) relations. In example 4.3.5 we saw how this can be represented as a list of functors. This

category is also the first category of algebras, i.e. the algebras with no constructors, and will as such also be referred to as Alg_0. It is important to note that we have the forgetful functor $t_1 : \mathcal{Rel} \Rightarrow \mathcal{Set}$, giving us the underlying set of a relation.

First constructor: $c_0 : A$

The first constructor c_0 has no arguments and is of sort $A : \mathsf{Set}$. Its arguments can be described by the functor $F_0 : \mathcal{Rel} \Rightarrow \mathcal{Set}$ with $F_0\ (X, R) :\equiv 1$. The category of algebras for the first constructor is then (F_0, t_1)-dialg, where $t_1 : \mathcal{Rel} \Rightarrow \mathcal{Set}$ is the forgetful functor. Strictly speaking, an (F_0, t_1)-dialgebra structure on a relation (X, R) is a function $1 \to X$, but for this example we will work with the equivalent definition: the category Alg_1 has:

- objects: $(X : \mathsf{Set}) \times (R : X \to X \to \mathsf{Set}) \times (\theta_0 : X)$.
- morphisms $(X, R, \theta_0) \to (Y, S, \rho_0)$ consist of:
  - a function $f : X \to Y$
  - a dependent function $g : (x\ y : X) \to R\ x\ y \to S\ (f\ x)\ (f\ y)$
  - an equality $f_0 : f\ \theta_0 = \rho_0$

We see that we get a morphism (f, g) in $\mathcal{Rel}$ along with a computation rule f_0 that tells us that the morphism in $\mathcal{Rel}$ preserves the (F_0, t_1)-dialgebra structure. Note that F_0 is a functor from $\mathcal{Rel}$ to $\mathcal{Set}$ and is not an endofunctor on $\mathcal{Set}$: the constructor may refer to elements of fibres of the relation $X \to X \to \mathsf{Set}$ being defined. The category Alg_1 also comes with a forgetful functor $V_1 : \mathsf{Alg}_1 \Rightarrow \mathsf{Alg}_0$ defined by $V_1((X, R), \theta_0) :\equiv (X, R)$ — in fact, every category of algebras Alg_{i+1} has a similarly defined forgetful functor $V_{i+1} : \mathsf{Alg}_{i+1} \Rightarrow \mathsf{Alg}_i$. Chaining all these together until we get to $\mathcal{Rel}$, we get functors $U_i : Alg_{i+1} \Rightarrow \mathcal{Rel}$.

Second constructor: $c_1 : A$

For the second constructor c_1, the specification is largely similar: it is given by the functor $F_1 : \mathsf{Alg}_1 \Rightarrow \mathcal{S}et$ defined by $F_1\ (X, R, \theta_0) :\equiv 1$. Alg_2 has the same objects as Alg_1, but extended with an extra point: $|\mathsf{Alg}_2| = ((X, R, \theta_0) : |\mathsf{Alg}_1|) \times (\theta_1 : X)$. Equivalently, Alg_2 can be thought of as being the dialgebra category $(F_1, t_1 \circ U_1)$-dialg.

Third constructor: $c_2 : \mathbb{N} \to B\ c_0\ c_1$

The third constructor c_2 maps into a different sort which is not $\mathcal{S}et$, hence its definition will be slightly different. We want the resulting category Alg_3 of algebras for constructors c_0, c_1, c_2 to have as objects:

$$((X, R, \theta_0, \theta_1) : |\mathsf{Alg}_2|) \times (\theta_2 : \mathbb{N} \to R\ \theta_0\ \theta_1)$$

The morphisms is where it gets a bit hairy: we want morphisms $((X, R), \theta_0, \theta_1, \theta_2) \to ((Y, S), \rho_0, \rho_1, \rho_2)$ to consist of $(f, g, f_0, f_1) : \mathsf{Alg}_2(((X, R), \theta_0, \theta_1), ((Y, S), \rho_0, \rho_1))$ together with an equality

$$g_2 : (n : \mathbb{N}) \to g\ \theta_0\ \theta_1\ (\theta_2\ n) =^S_{(f_0, f_1)}\ \rho_2\ n$$

Note how we have to use the equalities $f_0 : f\ \theta_0 = \rho_0$ and $f_1 : f\ \theta_1 = \rho_1$ to reconcile the types of

$$g\ \theta_1\ \theta_2\ (\theta_2\ n) : S\ (f\ \theta_0)\ (f\ \theta_1)$$

and

$$\rho_2 : S\ \rho_0\ \rho_1.$$

Realising the above as a dialgebra category is a bit tricky. We now have to deal with dependent functions and dependent equalities. The situation can be simplified however. Observe that if we have the following type

$$(x : A) \to P\ (e\ x)$$

for some $P : B \to \mathsf{Set}$ and $e : A \to B$, we can rewrite it to the equivalent type:

$$(x : A)\ (b : B) \to b = e\ x \to P\ b$$

By singleton contraction, we can show that the two types are indeed equivalent. The latter version is sometimes referred to as the "Henry Ford"-style type (you can have any $b : B$ you want, as long as it is $e\ x$). Along these lines, we can rewrite the type of θ_2 to the equivalent:

$$\hat{\theta_2} : (x\ y : X) \to ((x = \theta_0) \times (y = \theta_1) \times \mathbb{N}) \to R\ x\ y$$

and define the functor $F_2 : \mathsf{Alg}_2 \Rightarrow \mathcal{Rel}$ by $F_2((X, Y), \theta_0, \theta_1) :\equiv (X, \lambda x\, y.(x = \theta_0) \times (y = \theta_1) \times \mathbb{N})$ in order to see that $\hat{\theta_2}$ is a (F_2, U_2)-dialgebra. The objects in this category give us "too much": such a dialgebra gives us a morphism $F_2((X, Y), \theta_0, \theta_1) \to (X, Y)$ in $\mathcal{Rel}$, so we also get a function $X \to X$. We can solve this problem by adding an equation that the function $X \to X$ need be the identity. Alg_3 is therefore not a category of dialgebras, but a subcategory of one, as we will elucidate later on in section 4.4.3.

Fourth constructor: $c_3 : (n : \mathbb{N}) \to c_2\ n = c_2\ (n + 1)$

The fourth constructor c_3 is a path constructor, hence we not only need to supply a functor $F_3 : \mathsf{Alg}_3 \Rightarrow \mathsf{Alg}_0$ to specify the arguments, but we also need to specify the endpoints of the path. Just as for c_2, we will first need to rewrite the type of the constructor. We denote again by $\hat{\theta_2}$ the Henry Ford-version of θ_2. We then have that the type $(n : \mathbb{N}) \to \theta_2\ n = \theta_2\ (n + 1)$ is equivalent to the type

$$(x\ y : X) \to \big((p : x = \theta_0) \times (q : y = \theta_1) \times (n : \mathbb{N})\big) \\ \to \hat{\theta_2}\ x\ y\ p\ q\ n = \hat{\theta_2}\ x\ y\ p\ q\ (n + 1)$$

The endpoints can then be specified as natural transformations $\ell, r : F_3 \dot{\to} U_3$, where $F_3 : \mathsf{Alg}_3 \Rightarrow \mathcal{Rel}$ is defined in a way similar to F_2. Given an algebra $A : \mathsf{Alg}_3$, ℓ_A and r_A both define a morphism in $\mathcal{Rel}$. $\ell_{(X,R,\theta_0,\theta_1,\theta_2)}$ and

$r_{(X,R,\theta_0,\theta_1,\theta_2)}$ are defined as $(\mathsf{id}_X, \ell^1_{(X,R,\theta_0,\theta_1,\theta_2)})$ and $(\mathsf{id}_X, r^1_{(X,R,\theta_0,\theta_1,\theta_2)})$ respectively, with

$$\ell^1_{(X,R,\theta_0,\theta_1,\theta_2)}\ a\ b\ (p,q,n) :\equiv \hat{\theta}_2\ a\ b\ p\ q\ n$$
$$r^1_{(X,R,\theta_0,\theta_1,\theta_2)}\ a\ b\ (p,q,n) :\equiv \hat{\theta}_2\ a\ b\ p\ q\ (n+1)$$

By function extensionality, we can then say the category of algebras Alg_4 has objects $(X : |\mathsf{Alg}_4|) \times (\theta_3 : \ell_X = r_X)$, which can be described as an equaliser category. The morphisms are just morphisms in Alg_3 with no extra structure. For higher inductive types, one usually expects a path computation rule for any path constructor, but as we are working with sets, equalities between paths are trivial.

4.4.2 Specification of a quotient inductive-inductive definition

So far we have seen how to specify dependent sorts, deal with constructors referring to previous ones and have worked out how the category of algebras for a specific example, exhibiting point and path constructors of arbitrary sort, can be built up. In this section we will put everything together, arriving at our formal definition of a specification of a quotient inductive-inductive definition.

A quotient inductive-inductive definition is given by a specification of its sorts $\mathcal{S} : \mathsf{Sorts}$ plus a list of constructors, which may be point or path constructors, each of which has a sort $\mathcal{A} \in \mathcal{S}$ and builds upon the category of algebras of the previous constructors. We can formalise this as follows:

Definition 4.4.1 (Specification of quotient inductive-inductive definitions)**.** The specification of a quotient inductive-inductive definition with sorts $\mathcal{S} : \mathsf{Sorts}$, along with its category of algebras and underlying carrier functor is

given by the following inductive-recursive definition of

$$\begin{aligned}
&\mathbf{data}\ \mathsf{Spec} : \mathsf{Set}\\
&\mathsf{Constr} : (s : \mathsf{Spec})\ (\mathcal{A} : \mathsf{Cat}) \to \mathcal{A} \in \mathcal{S} \to \mathsf{Set}\\
&\mathsf{Alg} : \mathsf{Spec} \to \mathsf{Cat}\\
&\mathsf{U} : (s : \mathsf{Spec}) \to \mathsf{Alg}_s \Rightarrow [\![\ \mathcal{S}\ ]\!]
\end{aligned}$$

with

$$\begin{aligned}
&\mathbf{data}\ \mathsf{Spec}\ \mathbf{where}\\
&\quad \mathsf{nil} : \mathsf{Spec}\\
&\quad \mathsf{snoc} : (s : \mathsf{Spec})\ (\mathcal{A} : \mathsf{Cat})\ (p : \mathcal{A} \in \mathcal{S}) \to \mathsf{Constr}\ s\ \mathcal{A}\ p \to \mathsf{Spec}
\end{aligned}$$

where $\mathsf{Constr}\ s\ \mathcal{A}\ p$ is defined as:

$$\mathsf{Constr}\ s\ \mathcal{A}\ p :\equiv \mathsf{Constr}_{\mathsf{point}}\ s\ \mathcal{A}\ p + \mathsf{Constr}_{\mathsf{path}}\ s\ \mathcal{A}\ p$$

The functions $\mathsf{Constr}_{\mathsf{point}}$, $\mathsf{Constr}_{\mathsf{path}}$, Alg and U will be defined in the remainder of this section, treating point constructors in section 4.4.3 and path constructors in section 4.4.4.

Let us compare this definition to the previous descriptions of $\mathcal{S}et$-sorted definitions (definition 4.2.4). It looks largely similar in that we mutually define the type of specifications with the definition of its categories of algebras and forgetful functors. There is the addition of the function Constr, snoc constructor has a different type, as does the forgetful functor U.

In the constructor snoc, instead of having as argument a functor from a category of algebras into $\mathcal{S}et$, we have:

$$\mathsf{snoc} : (s : \mathsf{Spec})\ (\mathcal{A} : \mathsf{Cat})\ (p : \mathcal{A} \in \mathcal{S}) \to \mathsf{Constr}\ s\ \mathcal{A}\ p \to \mathsf{Spec}$$

The arguments $\mathcal{A} : \mathsf{Cat}$, $p : \mathcal{A} \in \mathcal{S}$ define the sort of this particular constructor, where the type $\mathsf{Constr}\ s\ \mathcal{A}\ p$ contains the further data needed to

specify the constructor. What this data is, i.e. the definition of $\mathsf{Constr}_{\mathsf{point}}$ and $\mathsf{Constr}_{\mathsf{path}}$ will be given in section 4.4.3 and section 4.4.4.

The forgetful functors U now do not map into $\mathcal{S}et$, but into the category of sorts $[\![\ \mathcal{S}\ ]\!]$.

4.4.3 Point constructors

In this section we will define three things:

- what the data for a point constructor is, i.e. we will define $\mathsf{Constr}_{\mathsf{point}}$,
- what, given these data, the category of algebras is,
- what the forgetful functor from this new category of algebras into the category of sorts is.

Suppose we have $S_i \in \mathcal{S}$ and $s : \mathsf{Spec}$, and we have already constructed

- $\mathsf{Alg}_s : \mathsf{Cat}$,
- $\mathsf{U}_s : \mathsf{Alg}_s \Rightarrow [\![\ \mathcal{S}\ ]\!]$ its forgetful functor, and
- the extension $\widehat{\mathsf{U}}_s : \mathsf{Alg}_s \Rightarrow S_i$ of U_s, which we get from $S_i \in \mathcal{S}$.

From the proof that $S_i \in \mathcal{S}$, we get some information about the category S_i. We know that we get a functor $t_i : S_i \Rightarrow S_{i-1}$ for some category S_{i-1} and that the objects of S_i are of the form $(X : |S_{i-1}|) \times (R_i X \to \mathsf{Set})$ for some functor $R_i : S_{i-1} \Rightarrow \mathcal{S}et$.

To make this more concrete: suppose $\mathcal{S}$ contains n sorts and s describes m constructors, then an object in $X : \mathsf{Alg}_s$ is a tuple $(X_0, \ldots, X_n, \theta_0, \ldots, \theta_m)$. The functor U_s gives us back all underlying carriers, i.e. $\mathsf{U}_s(X_0, \ldots, X_n, \theta_0, \ldots, \theta_m) = (X_0, \ldots, X_n)$ and the functor $\widehat{\mathsf{U}}_s$ further projects down to the i first carriers $X_0, \ldots, X_i$, with $(X_0, \ldots, X_{i-1}) : |S_{i-1}|$ and

$$X_i : R_i(X_0, \ldots, X_{i-1}) \to \mathsf{Set} \ .$$

Recall that $R_i : S_{i-1} \Rightarrow \mathcal{S}et$ is the functor that describes how the family X_i depends on the previous sorts $X_0, \ldots, X_{i-1}$. A point constructor structure

on X is of the form

$$\theta : (x : R_i(X_0, \ldots, X_{i-1})) \to F^1\ (X_0, \ldots, X_n, \theta_0, \ldots, \theta_m)\ x \to X_i\ x$$

where

$$F^1 : ((X_0, \ldots, X_n, \theta_0, \ldots, \theta_m) : |\mathsf{Alg}_s|) \to R_i(X_0, \ldots, X_{i-1}) \to \mathsf{Set}$$

Note that the functor $\widehat{\mathsf{U}}_s$ can be decomposed into two parts: we have $\widehat{\mathsf{U}}^0 : \mathsf{Alg}_s \to S_{i-1}$ and $\widehat{\mathsf{U}}^1 : (X : \mathsf{Alg}_s) \to R_i\ (\widehat{\mathsf{U}}^0\ X) \to \mathsf{Set}$. $\widehat{\mathsf{U}}^0$ can also be written as $t_i \circ \widehat{\mathsf{U}}_s$, where $t_i : S_i \to S_{i-1}$ is the forgetful functor of S_i. In order for the point constructor specified by F^1 to be well-defined, i.e. to make sense of its computation rule, we need $FX :\equiv (\widehat{\mathsf{U}}^0\ X, F^1\ X)$ to be a functor $\mathsf{Alg}_s \Rightarrow S_i$. This leads us to the definition of a point constructor specification:

Definition 4.4.2 (Data for a point constructor)**.** Given a specification $s : \mathsf{Spec}$, a point constructor of sort $S_i \in \mathcal{S}$ is specified by giving a functor $F : \mathsf{Alg}_s \Rightarrow S_i$ such that the following commutes:

$$\begin{array}{ccc} \mathsf{Alg}_s & \xrightarrow{F} & S_i \\ {\scriptstyle \widehat{\mathsf{U}}_s}\downarrow & & \downarrow{\scriptstyle t_i} \\ S_i & \xrightarrow{t_i} & S_{i-1} \end{array}$$

In other words, we have

$$\mathsf{Constr}_{\mathsf{point}}\ s\ S_i\ p :\equiv (F : \mathsf{Alg}_s \Rightarrow S_i) \times (p : t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s)\ .$$

The fact that F must satisfy the commutativity condition intuitively means that F leaves the sorts $(X_0, \ldots, X_{i-1})$ "untouched". We can think of F as describing a family of functors, fibred over the sorts $(X_0, \ldots, X_{i-1})$.

We now explain how to construct the category $\mathsf{Alg}_{s'}$ of algebras for a specification with an additional point constructor. Suppose the specification s' has been obtained as the extension of a specification s by a point constructor given by a functor $F : \mathsf{Alg}_s \to S_i$, which decomposes as $F = (\widehat{\mathsf{U}}^0, F^1)$. The previous characterisation of a point constructor algebra struc-

ture on an object $X : |\mathsf{Alg}_s|$ can be summarised as having a dependent function of morphisms

$$\theta : (x : R_i(\widehat{\mathsf{U}}^0\ X)) \to F^1\ X\ x \to \widehat{\mathsf{U}}^1\ X\ x$$

A morphism $(X, \theta) \to (Y, \rho)$ consists of a morphism $f : \mathsf{Alg}_s(X, Y)$ along with a "computation rule":

$$f_0 : (x : R_i\ (\widehat{\mathsf{U}}^0\ X))\ (y : F^1\ X\ x) \to \\ \widehat{\mathsf{U}}^1\ f\ x\ (\theta\ x\ y) = \rho\ (R_i(\widehat{\mathsf{U}}^0\ f\ x)\ (F^1\ f\ x\ y))$$

This category of algebras look similar to $(F, \widehat{\mathsf{U}}_s)$-dialg. However, as we have seen previously, the category of dialgebras contains too much: morphisms there also come with a superfluous endomorphism in S_{i-1}. We want this morphism to be an identity morphism, which we can achieve by taking an equaliser. The category $(F, \widehat{\mathsf{U}}_s)$-dialg comes with two forgetful functors: V that gives us the carrier of the algebra and V^I that gives us algebra structure on the carrier, which is a morphism in S_i or equivalently, an object in the arrow category S_i^I. The definition of the new category of algebras is as follows:

Definition 4.4.3 (Category of algebras for a point constructor)**.** Let $s' : \mathsf{Spec}$ be the specification s extended with a point constructor of sort S_i given by a functor $F : \mathsf{Alg}_s \to S_i$ satisfying $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$. The category of algebras $\mathsf{Alg}_{s'}$ is as the following equaliser in Cat:

$$\begin{array}{ccccccc} \mathsf{Alg}_{s'} & \xrightarrow{e} & (F, \widehat{\mathsf{U}}_s)\text{-dialg} & \xrightarrow{V} & \mathsf{Alg}_s & \xrightarrow{\widehat{\mathsf{U}}_s} S_i & \xrightarrow{t_i} S_{i-1} \\ & & \downarrow{\scriptstyle V^I} & & & & \downarrow{\scriptstyle \mathsf{id}} \\ & & S_i^I & & \xrightarrow{\qquad t_i^I \qquad} & & S_{i-1}^I \end{array}$$

where e is the projection map of the equaliser, t_i^I is the functor t_i lifted to a functor between the respective arrow categories and id is the functor that assigns to each object the identity morphism of that object.

The forgetful functor $\mathsf{U}_{s'} : \mathsf{Alg}'_s \to [\![\ \mathcal{S}\ ]\!]$ is defined as the composite:

$$\mathsf{Alg}_{s'} \xrightarrow{e} (F, \widehat{\mathsf{U}}_s)\text{-dialg} \xrightarrow{V} \mathsf{Alg}_s \xrightarrow{\mathsf{U}_s} [\![\ \mathcal{S}\ ]\!]$$

Note that usually the construction of equalisers in the category Cat is *evil* as it uses equality on objects. Since we are working with univalent categories, equality of objects coincides with isomorphism of objects, hence this is not an issue in our setting.

Remark 4.4.4. To explicate the phenomenon of a category of dialgebras containing "too much", we will look at how the category of algebras of the following inductive-inductive definitions is built up:

$$\begin{aligned}&\textbf{data}\ A : \mathsf{Set}\\ &\textbf{data}\ B : A \to \mathsf{Set}\end{aligned}$$

$$\begin{aligned}&\textbf{data}\ A\ \textbf{where}\\ &\quad c_0 : A\end{aligned}$$

$$\begin{aligned}&\textbf{data}\ B\ \textbf{where}\\ &\quad c_1 : (x : A) \to 0 \to B\ x\end{aligned}$$

We can see that A is equivalent to 1. The second constructor c_1 does not add anything to any $B\ x$: B is the constantly empty family.

The constructor c_0 can be described by the functor:

$$\begin{aligned}F_0 &: \mathcal{F}am \Rightarrow \mathcal{S}et\\ F_0(X, P) &:\equiv 1\end{aligned}$$

The category of algebras containing only the first constructor is (F_0, V)-dialg where $V : \mathcal{F}am \Rightarrow \mathcal{S}et$ is the forgetful functor. The second constructor c_1, which does not add anything to the inductive type, can be described by the

functor:

$$F_1 : (F_0, V)\text{-dialg} \Rightarrow \mathcal{F}am$$
$$F_1((X, P), \theta) :\equiv (X, \lambda x.0)$$

It is clear that this functor satisfies the condition that $V \circ F_1 = V \circ U_0$ where $U_0 : (F_0, V)$-dialg $\Rightarrow \mathcal{F}am$ is the forgetful functor.

If we then unfold the definition of (F_1, U_0)-dialg, we see that it has objects:

$$\begin{aligned}|(F_1, U_0)\text{-dialg}| &= (X : \mathsf{Set}) \times (P : X \to \mathsf{Set}) \\ &\quad \times (\theta_0 : 1 \to X) \\ &\quad \times (\theta_1 : X \to X) \times (\theta_2 : (x : X) \to 0 \to P\ (\theta_1\ x)) \\ &= (X : \mathsf{Set}) \times (P : X \to \mathsf{Set}) \times (\theta_0 : X) \times (\theta_1 : X \to X)\end{aligned}$$

As mentioned before: we see that we get a superfluous $X \to X$ function. For the inductive definition on its own, we are just interested in the initial object of the category of algebras. Here we see that the initial object of the dialgebra category is the natural numbers with a constantly empty family over it, which is different from what we want: a *unit type* with a constantly empty family over it.

4.4.4 Path constructors

Specifying the arguments for a path constructor is done in exactly the same way as for a point constructor, while the endpoints of the path are given by natural transformations. Suppose $s : \mathsf{Spec}$, then a path constructor structure of sort $S_i \in \mathcal{S}$ on an object $(X_0, \ldots, X_n, \theta_0, \ldots, \theta_m) : |\mathsf{Alg}_s|$ is of the form:

$$\begin{aligned}&\theta : (x : R_i(X_0, \ldots, X_{i-1})) \\ &\quad \to (y : F^1\ (X_0, \ldots, X_n, \theta_0, \ldots, \theta_m))\ x \\ &\quad \to \ell^1_{(X_0,\ldots,X_n,\theta_0,\ldots,\theta_m)}\ x\ y =_{X_i\ x} r^1_{(X_0,\ldots,X_n,\theta_0,\ldots,\theta_m)}\ x\ y\end{aligned}$$

where

- $F^1 : (X : |\mathsf{Alg}_s|) \to R_i(\widehat{\mathsf{U}}^0 X) \to \mathsf{Set}$, such that $FX :\equiv (\widehat{\mathsf{U}}^0 X, F^1 X)$ is a functor $\mathsf{Alg}_s \Rightarrow S_i$, and
- $\ell^1, r^1 : (X : |\mathcal{C}|)\ (x : R_i(\widehat{\mathsf{U}}^0\ X))\ (y : F^1\ X\ x) \to \widehat{\mathsf{U}}^1\ X\ x$ such that $\ell_X :\equiv (\mathsf{id}_{\widehat{\mathsf{U}}^0 X}, \ell^1)$ and $r_X :\equiv (\mathsf{id}_{\widehat{\mathsf{U}}^0 X}, r^1)$ are natural transformations $F \dot{\to} \widehat{\mathsf{U}}_s$.

Algebra morphisms $(X, \theta) \to (Y, \rho)$ are simply morphisms $X \to Y$ in Alg_s. As we are working with sets, we do not need computation rules for the paths: any equation between paths is trivial.

Summarising the above, we get to the following definition of path constructor specification:

Definition 4.4.5 (Data for a path constructor)**.** Given a specification $s : \mathsf{Spec}$, a path constructor of sort $S_i \in \mathcal{S}$ is specified by a functor $F : \mathsf{Alg}_s \Rightarrow S_i$ satisfying $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$ and two natural transformations $\ell, r : F \dot{\to} \widehat{\mathsf{U}}_s$, such that when whiskered[1] with t_i they are the identity natural transformation, i.e. they satisfy $t_i\ \ell = t_i\ r = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s}$:

$$\begin{aligned}\mathsf{Constr}_{\mathsf{path}}\ s\ S_i\ p :\equiv (F : \mathsf{Alg}_s \to S_i) \times (p : t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s)\\ \times (\ell : F \dot{\to} \widehat{\mathsf{U}}_s) \times (q_\ell : t_i\ \ell = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s})\\ \times (r : F \dot{\to} \widehat{\mathsf{U}}_s) \times (q_r : t_i\ r = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s})\ .\end{aligned}$$

The restriction on the natural transformations ℓ and r is the same kind of restriction as the restriction on F: the natural transformations must leave the sorts below S_i untouched.

Note that by function extensionality, the type of θ is equivalent to $\ell^1_X = r^1_X$. Since $t_i\ \ell_X = t_i\ r_X = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s}$ the equality $\ell^1_X = r^1_X$ of dependent functions in $\mathcal{S}et$ is equivalent to the equality $\ell_X = r_X$ of morphisms in S_i. We also observe that a natural transformation $\alpha : F \dot{\to} \widehat{\mathsf{U}}_s$ gives rise to a functor

[1] Whiskering a natural transformation $\alpha : F \dot{\to} G$, with $F, G : \mathcal{C} \Rightarrow \mathcal{D}$ with a functor $H : \mathcal{D} \Rightarrow \mathcal{E}$ yields a natural transformation $H\alpha : HF \dot{\to} HG$ by applying H on every component of α.

$\hat{\alpha} : \mathsf{Alg}_s \Rightarrow S_i^I$, mapping $X : |\mathsf{Alg}_s|$ to $\alpha_X : S_i(F\ X, \widehat{\mathsf{U}}_s\ X)$. Functoriality of this functor comes from the naturality of α. This leads us to the definition of the category of algebras for a path constructor:

Definition 4.4.6 (Category of algebras for a path constructor)**.** Let $s' : \mathsf{Spec}$ be the specification s extended with a path constructor of sort S_i given by a functor $F : \mathsf{Alg}_s \to S_i$ satisfying $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$ and natural transformations $l, r : F \to \widehat{\mathsf{U}}_s$ satisfying $t_i\ l = t_i\ r = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s}$. The category of algebras $\mathsf{Alg}_{s'}$ is defined as the following equaliser in Cat:

$$\mathsf{Alg}_{s'} \xrightarrow{e} \mathsf{Alg}_s \underset{\hat{r}}{\overset{\hat{\ell}}{\rightrightarrows}} S_i^I$$

where e is the projection map of the equaliser.

The forgetful functor $\mathsf{U}_{s'} : \mathsf{Alg}'_s \to [\![\ \mathcal{S}\ ]\!]$ is defined as the composite

$$\mathsf{Alg}_{s'} \xrightarrow{e} \mathsf{Alg}_s \xrightarrow{\mathsf{U}_s} [\![\ \mathcal{S}\ ]\!]$$

4.4.5 Worked example

In this section we will work out the description of the $(\mathsf{Con}, \mathsf{Ty})$ type family of contexts and types in a context as described in section 3.1.6. First of all, notice that it is a $\mathcal{F}am$-sorted definition. In example 4.3.4, we have seen that these can be described with the following functors:

- $R_{\mathsf{Con}} : \mathbb{1} \Rightarrow \mathcal{S}et$, $R_{\mathsf{Con}}\ x :\equiv 1$
- $R_{\mathsf{Ty}} : [\![\ R_{\mathsf{Con}}\ ]\!] \Rightarrow \mathcal{S}et$, $R_{\mathsf{Ty}}(x, A) :\equiv A\ x$

This gives us the sorts category $[\![\ R_0, R_1\ ]\!]$ with:

- objects: $(x : 1) \times (A : 1 \to \mathsf{Set}) \times (B : A\ x \to \mathsf{Set})$
- morphisms $(x, A, B) \to (y, C, D)$: $(f : 1 \to 1) \times (g : (x : 1) \to A\ x \to C\ (f\ x)) \times (h : (a : A\ x) \to B\ a \to D\ (g\ x\ a))$

To not clutter things too much with terms of type 1, we will leave these out of the sorts part of the definition. The inductive-inductive definition we are

trying to fit into the framework we just presented is as follows, massaged a bit to see more directly how it fits in the framework:

$$\begin{aligned}
&\textbf{data}\ \mathsf{Con} : 1 \to \mathsf{Set}\\
&\textbf{data}\ \mathsf{Ty} : \mathsf{Con}\ \mathsf{tt} \to \mathsf{Set}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \mathsf{Con}\ \textbf{where}\\
&\quad \epsilon : 1 \to \mathsf{Con}\ \mathsf{tt}\\
&\quad _,_ : (\Gamma : \mathsf{Con}\ \mathsf{tt}) \times (\tau : \mathsf{Ty}\ \Gamma) \to \mathsf{Con}\ \mathsf{tt}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \mathsf{Ty}\ \textbf{where}\\
&\quad \text{`}0 : (\Gamma : \mathsf{Con}\ \mathsf{tt}) \to 1 \to \mathsf{Ty}\ \Gamma\\
&\quad \text{`}1 : (\Gamma : \mathsf{Con}\ \mathsf{tt}) \to 1 \to \mathsf{Ty}\ \Gamma\\
&\quad \text{`}\Pi : (\Gamma : \mathsf{Con}\ \mathsf{tt}) \to (A : \mathsf{Ty}\ \Gamma) \times \mathsf{Ty}\ (\Gamma,A) \to \mathsf{Ty}\ \Gamma
\end{aligned}$$

The specification $s_{(\mathsf{Con},\mathsf{Ty})} : \mathsf{Spec}$ is a list of five elements, all containing a point constructor. Of the point constructors, the first two are of the first sort, $1 \to \mathsf{Set}$, and the last three of sort $\mathsf{Con}\ \mathsf{tt} \to \mathsf{Set}$. The following functors describe the point constructors:

- $F_{\epsilon}\ (A, B) :\equiv (\lambda\_.1)$
- $F_{\_,\_}\ ((A, B), \theta_{\epsilon}) :\equiv (\lambda\_.(\Gamma : A\ \mathsf{tt}) \times (\tau : B\ \Gamma))$
- $F_{\text{`}0}\ ((A, B), \theta_{\epsilon}, \theta_{\_,\_}) :\equiv (A, \lambda\_.1)$
- $F_{\text{`}1}\ ((A, B), \theta_{\epsilon}, \theta_{\_,\_}, \theta_{\text{`}0}) :\equiv (A, \lambda\_.1)$
- $F_{\text{`}\Pi}\ ((A, B), \theta_{\epsilon}, \theta_{\_,\_}, \theta_{\text{`}0}, \theta_{\text{`}1}) :\equiv (A, \lambda\Gamma.(X : B\ \Gamma) \times B\ (\theta_{\_,\_}\ (\Gamma, X)))$

The algebras we end up with, after singleton contraction, look as follows:

$$\begin{aligned}
&(A : 1 \to \mathsf{Set}) \times (B : A\ \mathsf{tt} \to \mathsf{Set}) \\
&\times (\theta_{\epsilon} : 1 \to A\ \mathsf{tt}) \\
&\times (\theta_{_,_} : (\Gamma : A\ \mathsf{tt}) \times B\ \Gamma \to A\ \mathsf{tt}) \\
&\times (\theta_{\mathsf{`0}} : (\Gamma : A\ \mathsf{tt}) \to 1 \to B\ \Gamma) \\
&\times (\theta_{\mathsf{`1}} : (\Gamma : A\ \mathsf{tt}) \to 1 \to B\ \Gamma) \\
&\times (\theta_{\mathsf{`\Pi}} : (\Gamma : A\ \mathsf{tt}) \to (X : B\ \Gamma) \times B\ (\theta_{_,_}\ (\Gamma, X)) \to B\ \Gamma)
\end{aligned}$$

If we squint our eyes a bit, we see that the algebras indeed coincide with the constructors we wanted to give a specification of.

The pieces of the specification we have left out here for notational clarity, are the sort membership proofs. An example of these proofs has been given in example 4.3.8, also for the same sorts specification as the one we use here.

4.5 Other forms of constructors

We have characterised quotient inductive-inductive definitions as (subcategories of) iterated dialgebras. For the dialgebras we have looked at, we only had the freedom to choose the left functor, i.e. if we look at the point constructors we have seen so far, they all are of the shape:

$$c : S_i(FX, UX)$$

where $s : \mathsf{Spec}$ and $F, U : \mathsf{Alg}_s \Rightarrow S_i$ for some sort category S_i. We have only considered the case where U is a forgetful functor. This begs the question if we generalise this to a larger class of dialgebras. For example, can we have constructors such as:

$$c : A \to \mathsf{List}\ A$$

It turns out that this particular example does not work out. To see this, let us define the following inductive definition:

$$\begin{aligned}&\textbf{data } A : \mathsf{Set} \textbf{ where}\\ &\quad c_0 : A\\ &\quad c_1 : A \to \mathsf{List}\ A\end{aligned}$$

Proposition 4.5.1. *A does not exist, i.e. the category of algebras that corresponds to the above inductive definition does not have an initial algebra.*

Proof. Suppose (X, x, θ) is an initial algebra, i.e. a pointed set (X, x) with a function $\theta : X \to \mathsf{List}\ X$. By pattern matching, we know that $\theta\ x$ is either nil or $\mathsf{cons}\ a\ as$ for some $a : X$ and $as : \mathsf{List}\ A$.

If $\theta\ x = \mathsf{nil}$, then we can define an algebra $(1, \mathsf{tt}, \lambda x.[\mathsf{tt}])$. We observe that there are no algebra morphisms from (X, x, θ) into $(1, \mathsf{tt}, \lambda x.[\mathsf{tt}])$: for $f : X \to 1$, which obviously satisfies $f\ x = \mathsf{tt}$, we have that $\mathsf{nil} = \mathsf{List}\ f\ (\theta\ x) \neq [\mathsf{tt}]$.

In the case that $\theta\ x = \mathsf{cons}\ a\ as$, we can do something similar: instead of having a singleton list as the algebra on $(1, \mathsf{tt})$, we have the empty list. We can then observe that there are no algebra morphisms from (X, x, θ) into $(1, \mathsf{tt}, \lambda x.\mathsf{nil})$. □

There are examples of constructors that do make sense. If we want to truncate our inductive type to be propositional, we can add a constructor:

$$c : \mathsf{is\text{-}prop}\ A$$

This works out in this case, as $\mathsf{is\text{-}prop}\ A$ is equivalent to $(x\ y : A) \to x = y$.

Other examples that are allowed, are constructors of the form $B \to A \times A$, as this is equivalent to have two constructors with type $B \to A$. Generalising this, the result type may be given by any representable functor. One notable example of such a representable functor is the stream functor, e.g. we can have a constructor, with $B : \mathsf{Set}$:

$$c : B \to \mathsf{Stream}\ A$$

as this is equivalent to $B \to \mathbb{N} \to A$.

4.5.1 Dependent dialgebras

As an earlier attempt to unifying the treatment of point and path constructors, we have considered *dependent dialgebras* [Alt+15], i.e. given $F : \mathcal{C} \Rightarrow \mathcal{S}et$ and $G : \int_{\mathcal{C}} F \Rightarrow \mathcal{S}et$ a dependent dialgebra is an object $X : |\mathcal{C}|$ along with:

$$\theta : (x : FX) \to G(X, x)$$

The category $\int_{\mathcal{C}} F$ is the *category of elements* or the *Grothendieck construction* of the functor F. It is defined as having:

- objects: $(X : |\mathcal{C}|) \times FX$
- morphisms $(X, x) \to (Y, y)$: $(f : \mathcal{C}(X, Y)) \times (F\ f\ x = y)$

In other words: the category of elements has as objects objects in X along with a point in FX. Morphisms are morphisms in $\mathcal{C}$ that preserve the points.

One nice feature of this approach is that it gives us a unified framework to talk about both path constructors as well as point constructors of various sorts.

However, formalising the category structure of dependent dialgebras turns out to be rather involved. In particular, when defining what morphisms between these algebras should be, we need to lift a morphism in $\mathcal{C}$ to one in $\int_{\mathcal{C}} F$. There is a canonical way of doing so, given an $x : FX$ for some $X : |\mathcal{C}|$: given $f : \mathcal{C}(X, Y)$ with $x : FX$, we define:

$$\hat{f}_x : \int_{\mathcal{C}} F((X, x), (Y, F\ f\ x))$$
$$\hat{f}_x :\equiv (f, \mathsf{refl})$$

This operation is not functorial in the untruncated setting: we do not have $\widehat{g \circ f}_x$ and $\hat{g}_{Ffx} \circ \hat{f}_x$ are equal. In fact, they have different types:

- $\widehat{g \circ f}_x : \int_{\mathcal{C}} F((X, x), (Z, F\ (g \circ f)\ x))$

- $\hat{g}_{Ffx} \circ \hat{f}_x : \int_{\mathcal{C}} F((X, x), (Z, F\ g\ (F\ f\ x)))$

While we do have a proof that $F\ (g \circ f)\ x = F\ g\ (F\ f\ x)$, we do not have in general that this equation holds definitionally. Writing down the category structure therefore involves lots of transporting across the proofs of functoriality of F and reasoning about these transports.

In the approach taken in this thesis, we build up the category of algebras using easier to define building blocks, i.e. equaliser categories and categories of (ordinary) dialgebras. While this approach still has its share of having to deal with transporting terms across proofs of functoriality, these can be solved in a more abstract setting with fewer moving parts.

Another reason why this thesis does not use dependent dialgebras is that they are too general for our purposes. If we try to prove properties about them, e.g. the existence of limits and initial objects, it is not immediately clear what restrictions to put on the target functor G to make things work. In this thesis we use a more bottom up approach, starting from the inductive definitions we know and what functors show up when defining them.

4.5.2 Currying

When a constructor has multiple arguments, we often write things down in its curried form, e.g. suppose we have $K, L : \mathsf{Set}$ and define the following constructor for type A:

$$c : K \to (L \to A)$$

In our presentation, we consider dialgebras where the functor describing the target of the constructor is usually a forgetful functor. As such, we are forced to have all our constructors in uncurried form, e.g. the above case would be $K \times L \to A$. To describe the curried constructor c, we can see it as a (F, G)-dialgebra with $FX :\equiv K$ and $GX :\equiv (L \to X)$. This approach does not always work, if we were to consider instead the constructor:

$$c : K \to (A \to A)$$

We cannot define $GX :\equiv (X \to X)$ as it is not functorial. So it seems that at this level of syntax, dialgebras are not the appropriate concept to describe what is going on.

4.6 Positivity

Not all expressions of the type Set $\to$ Set are functorial: the arguments may only be in positive positions in the result. As we have mentioned in chapter 1, $\lambda X.(X \to X)$ is problematic. Whilst problematic, such negative occurrences can be useful. One example of is the higher order abstract syntax embedding of untyped lambda calculus:

$$\begin{aligned}&\textbf{data}\ \mathsf{Tm} : \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{app} : \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm}\\ &\quad \mathsf{lam} : (\mathsf{Tm} \to \mathsf{Tm}) \to \mathsf{Tm}\end{aligned}$$

In the constructor lam, there is a recursive occurrence in both negative as well as a positive position. The map $\lambda X.(X \to X)$ is therefore not functorial: it is neither contravariant nor covariant.

This is problematic with pattern matching semantics. Having an inductive type such as Tm along with pattern matching (even when recursion is restricted to structurally smaller subterms), is unsound and allows us to write diverging terms.

Another example of this phenomenon is if we write down an inductive definition with as constructors the axioms of a field:

$$\begin{aligned}&\textbf{data}\ \mathsf{Field} : \mathsf{Set}\ \textbf{where}\\ &\quad 0 : \mathsf{Field}\\ &\quad \vdots\\ &\quad \mathsf{inv} : (x : \mathsf{Field}) \to ((x = 0) \to 0) \to \mathsf{Field}\\ &\quad \vdots\end{aligned}$$

The category of algebras of this specification should be equivalent to the category of fields. If the datatype Field were to exist, it then would be the initial object in this category. However, the category of fields does not have an initial object. The problem with the datatype is that the constructor inv has a recursive occurrence in a negative position: the constructor 0 occurs in a negative position.

4.7 Related work

4.7.1 Inductive-inductive definitions

The approach we have taken in our framework builds heavily on the work on inductive-inductive types [Alt+11]. In the article, the authors describe the categorical interpretation of an inductive-inductive definition with one $\mathcal{S}et$-sorted point constructor, followed by one $\mathcal{F}am$-sorted point constructor. We have slightly altered their idea of using an equaliser category for the $\mathcal{F}am$-sorted constructor to allow us to iterate the construction more easily. Furthermore, we have extended the framework to support a larger class of sorts rather than $\mathcal{F}am$ and have added support for path constructors.

4.7.2 Inductive definitions in Agda

Our framework allows us to specify most of the inductive definitions we can write down in Agda. The notable exception is that we do not support induction-recursion. However, some forms of induction-recursion can be simulated, using the methods described in [Han+13].

Apart from the absence of induction-recursion, there is another fundamental difference between our framework and that of Agda. In Agda we have to group all the constructors per sort, i.e. we cannot write down a definition with sorts A : Set, B : $A \to$ Set and constructors:

- c_0 : A
- c_1 : $B\ c_0$

- $c_2 : c_1 = c_1 \to A$

We cannot alternate between the sorts in Agda, which prevents us from defining types as the one given above. Our framework has no such restriction.

Furthermore, Agda allows for certain negative inductive definitions. Its positivity checker only checks whether the recursive occurrences referring to the type being defined are in strictly positive positions. It does not check whether references to previous *constructors* are in strictly positive positions. For example, the following definition is accepted:

$$\begin{aligned}&\textbf{data } A : \mathsf{Set} \textbf{ where}\\ &\quad c_0 : A\\ &\quad c_1 : ((c_0 = c_0) \to A) \to A\end{aligned}$$

The operation $\lambda(X, \theta).(\theta = \theta) \to A$ is not functorial, so we cannot represent this example in our framework. As of yet, it is unclear whether this is a bug in Agda's positivity checker, i.e. we have not found a way yet to use this to prove falsity.

Similarly, Agda allows for sorts which are not positive. For example, Agda accepts the following definition:

$$\begin{aligned}&\textbf{data } A : \mathsf{Set}\\ &\textbf{data } B : (A \to A) \to \mathsf{Set}\end{aligned}$$

$$\begin{aligned}&\textbf{data } A \textbf{ where}\\ &\quad c_0 : A\end{aligned}$$

$$\begin{aligned}&\textbf{data } B \textbf{ where}\\ &\quad c_1 : B\ (\lambda x.x)\end{aligned}$$

We cannot formalise definitions of this shape in our framework, as the oper-

ation $\lambda X.(X \to X)$ is not functorial, hence we cannot define its dependent sorts. As with the previous example, it is as of yet unclear to us whether definitions of this form may lead to inconsistencies.

4.7.3 Higher inductive types

Our quotient inductive-inductive definitions are a first approximation to a theory of higher inductive types. The presentation here has been inspired by work on the semantics of higher inductive types [LS13b] via monads in model categories. Furthermore, [Cap14] and [Alt+15], which also build upon that note, have influenced the design choices of this framework.

Chapter 5

Induction versus initiality

In the previous chapter we have seen how we can specify a quotient inductive-inductive definition by giving a list of sorts and a list of constructors: we have to give the type formation and introduction rules. From these rules, we can then derive the category of algebras. The notion of algebra morphism looks a lot like the recursion principle for the inductive definition. In fact, saying that an algebra is *weakly initial* is precisely the statement that the algebra satisfies the recursion principle. The link between full initiality and the induction principle is a bit more involved. Initiality talks about the morphism we get from the recursion principle being *unique*: its statement involves equality of morphisms. The induction principle is something that allows us to produce dependent functions into families over our inductive definitions.

Initial algebra semantics is a very attractive way of "explaining" inductive definitions: stating the property of being initial only requires us to have defined the objects and morphisms:

Definition 5.0.1 (Initiality)**.** An object X of a category $\mathcal{C}$ is *initial* if for every $Y : |\mathcal{C}|$ the set $\mathcal{C}(X, Y)$ is contractible.

From this definition it is also immediate that being initial is propositional.

5.1 Categorical characterisation of induction

The induction principle of an inductive type T gives us a way to construct dependent functions for families defined on T. The family P which we are eliminating into is also called the *motive* of the induction. By providing *methods* for every constructor for this motive, the induction principle gives us a function $(x : T) \to P\ x$. Recall that in the case of the natural numbers, the methods we have to supply have the following types:

- $m_{\mathsf{zero}} : P\ \mathsf{zero}$,
- $m_{\mathsf{succ}} : (n : \mathbb{N}) \to P\ n \to P\ (\mathsf{succ}\ n)$,

Given all this, the induction principle yields a dependent function $s : (x : \mathbb{N}) \to P\ x$ satisfying certain computation rules.

We can think of the triple $(P, m_{\mathsf{zero}}, m_{\mathsf{succ}})$ as a family of algebras defined over the algebra $(\mathbb{N}, \mathsf{zero}, \mathsf{succ})$:

Definition 5.1.1. We define a type of algebra families for this particular category of algebras as $\mathsf{Fam}_{\mathsf{Alg}_{\lambda X.1+X}} : |\mathsf{Alg}_{\lambda X.1+X}| \to \mathsf{Set}$ with:

$$\begin{aligned}\mathsf{Fam}_{\mathsf{Alg}_{\lambda X.1+X}}(X, \theta_0, \theta_1) :\equiv\ & (P : X \to \mathsf{Set}) \\ & \times (m_0 : P\ \theta_0) \\ & \times (m_1 : (x : X) \to P\ x \to P\ (\theta_1\ x))\end{aligned}$$

In order to see that this type does correspond to a notion of family, recall that families on a type can also be represented as functions into said type, i.e. as a *fibration*:

Proposition 5.1.2. *Given $X : \mathsf{Set}$, there is an equivalence:*

$$(X \to \mathsf{Set}) = (Y : \mathsf{Set}) \times (p : Y \to X)$$

Proof. Let $P : X \to \mathsf{Set}$ be a family on X, we can map this to the pair $((x : X) \times P\ x, \pi_0)$, i.e. the family's total space along with its projection map into its base space. In the other direction we map (Y, p) to the preimage

family $\lambda x.(y : Y) \times (p\, y = x)$. Checking that these two maps are each other's inverses can be done by using function extensionality and univalence. □

For the aforementioned algebra families, we have a similar equivalence:

Proposition 5.1.3. *Given $X : \mathsf{Alg}_{\lambda X.1+X}$, there is an equivalence:*

$$\mathsf{Fam}_{\mathsf{Alg}_{\lambda X.1+X}}\, X = (Y : |\mathsf{Alg}_{\lambda X.1+X}|) \times (p : \mathsf{Alg}_{\lambda X.1+X}(Y, X))$$

Proof. The proof follows the same structure as the Set case. Given an algebra family (P, m_0, m_1), we can define its "total algebra" as follows:

$$\mathsf{total}\ (P, m_0, m_1) :\equiv ((x : X) \times P\ x, (\theta_0, m_0), (\lambda(x, p).(\theta_1\ x, m_1\ x\ p)))$$

The projection function $\pi_0 : (x : X) \times P\ x \to X$ turns out to be an algebra morphism $\mathsf{total}\ (P, m_0, m_1) \to (X, \theta_0, \theta_1)$: it satisfies the computation rules definitionally. Let us denote this morphism as $\mathsf{proj}\ (P, m_0, m_1)$. The mapping from left to right maps (P, m_0, m_1) to the pair $(\mathsf{total}\ (P, m_0, m_1), \mathsf{proj}\ (P, m_0, m_1))$.

For the other direction we need to generalise the preimage family to algebras: given (Y, ρ_0, ρ_1) with $(p, p_0, p_1) : \mathsf{Alg}_{\lambda X.1+X}((Y, \rho_0, \rho_1), (X, \theta_0, \theta_1))$, we define the following family:

$$\begin{array}{ll}
(\ \lambda x.(y : Y) \times p\ y = x & : X \to \mathsf{Set} \\
,\ (\rho_0, p_0) & : (y : Y) \times p\ y = \theta_0 \\
,\ \lambda x(y, z).(\rho_1\ y, w) & : (x : X) \to (y : Y) \times (p\ y = x) \to (y' : Y) \times (p\ y' = \theta_1\ x) \\
)
\end{array}$$

where w is defined as the following path:

$$p\ (\rho_1\ y) \xrightarrow{p_1} \theta_1\ (p\ y) \xrightarrow{\mathsf{ap}\ \theta_1\ z} \theta_1\ x$$

□

Given a family $P : X \to \mathsf{Set}$, a dependent function $(x : X) \to P\ x$ corresponds to a *section* of the projection function $\pi_0 : (x : X) \times P\ x \to$

X. As it turns out, the corresponding notion of dependent function for an algebra family is a dependent function along with computation rules, i.e. everything we get from the induction principle:

Definition 5.1.4. Given an algebra family $(P, m_{\mathsf{zero}}, m_{\mathsf{succ}})$, a *dependent algebra morphism* is a dependent function $s : (x : X) \to P\ x$ equipped with the computation rules:

- $s_{\mathsf{zero}} : s\ \mathsf{zero} = m_{\mathsf{zero}}$
- $s_{\mathsf{succ}} : (n : \mathbb{N}) \to s\ (\mathsf{succ}\ n) = m_{\mathsf{succ}}\ n\ (s\ n)$

As the definitions of function into X and section only refer to the category structure, this generalises to any category. The induction principle that gives us a dependent morphism for any family can therefore be phrased abstractly as follows:

Definition 5.1.5. The *section principle* for an object X in a category $\mathcal{C}$ says that for every $Y : |\mathcal{C}|$ and $p : \mathcal{C}(Y, X)$, there exists $s : \mathcal{C}(X, Y)$ and a proof of $p \circ s = \mathsf{id}_X$, i.e. that there is a term of type:

$$(Y : |\mathcal{C}|) \times (p : \mathcal{C}(Y, X)) \to (s : \mathcal{C}(X, Y)) \times (p \circ s = \mathsf{id}_X)$$

5.2 The section principle is logically equivalent to initiality

Now we have a category theoretic characterisation of the induction principle, we have to show that it is logically equivalent to initiality. Assuming a bit more structure of the categories we are working with, namely that finite limits exist, we can show that an object satisfies the section principle if and only if it is an initial object.

To see that finite limits are exactly what we need, we will first sketch what the proof of the logical equivalence looks like for the natural numbers. Showing that initiality implies the induction principle means that we are given $X : \mathsf{Set}$ with $\theta_0 : X$ and $\theta_1 : X \to X$, forming an initial algebra.

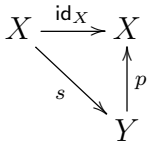
Given $P : X \to \mathsf{Set}$ with $m_0 : P\ \theta_0$ and $m_1 : (x : X) \to P\ x \to P\ (\theta_1\ x)$, we have to define a dependent function $s : (x : X) \to P\ x$. Using the function total we get an algebra and hence a unique algebra morphism $(X, \theta_0, \theta_1) \to \mathsf{total}\ (P, m_0, m_1)$. This algebra morphism gives us a function $s : X \to (x : X) \times P\ x$ and using uniqueness of the algebra morphism, we get that $\pi_0 \circ s = \mathsf{id}_X$, hence we get a dependent function $(x : X) \to P\ x$. We furthermore have that this dependent function satisfies the computation rules, which follow from the computation rules satisfied by the algebra morphism.

To establish that the induction principle gives us initiality, we will produce an algebra morphism and then show it is in fact unique. Suppose $X : \mathsf{Set}$ with $\theta_0 : X$ and $\theta_1 : X \to X$ satisfies the induction principle, then we have for any other algebra (Y, ρ_0, ρ_1) the algebra family $(\lambda x.Y, \rho_0, \lambda x.\rho_1)$. This family is the family that is "constantly (Y, ρ_0, ρ_1)". In the categorical proof, this corresponds to constructing the product of (X, θ_0, θ_1) with (Y, ρ_0, ρ_1). We then get a function $f : X \to Y$, which is an algebra morphism thanks to the computation rules we get from the induction principle. We have to show that this function is unique: for any other $g : X \to Y$ which is an algebra morphism, we need to show that $f = g$. By employing function extensionality, this is equivalent to showing $(x : X) \to f\ x = g\ x$, which means we can use the induction principle to prove this. The motive of the induction is the family $\lambda x\ .\ f\ x = g\ x$ with the methods being of the types $f\ \theta_0 = g\ \theta_0$ and $(x : X) \to f\ x = g\ x \to f\ (\theta_1\ x) = g\ (\theta_1\ x)$. These methods can be defined using the computation rules of f and g. In the categorical proof, constructing this algebra family corresponds to constructing the equaliser of the two algebra morphisms f and g, which gives us a proof of $f = g$.

Lemma 5.2.1. *Let $\mathcal{C} : \mathsf{Cat}$. If $X : |\mathcal{C}|$ is initial, then X satisfies the section principle.*

Proof. Assume X is initial. Given a morphism $p : Y \to X$, we need to produce a morphism $s : X \to Y$ such that $p \circ s = id_X$. Since X is initial, we

get a unique arrow $s : X \to Y$ such that:



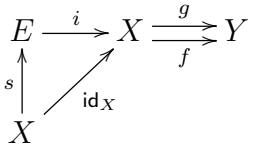
The composite has to be equal to the identity morphism on X, as that by initiality is the only endomorphism on X. □

Lemma 5.2.2. *Let $\mathcal{C}$: Cat and assume $\mathcal{C}$ has finite limits. If $X : |\mathcal{C}|$ satisfies the section principle, then X is initial.*

Proof. Given $Y : |\mathcal{C}|$, we need to provide a unique arrow $X \to Y$. Consider the projection $\pi_0 : X \times Y \to X$, which is an arrow into X and therefore has a section $s : X \to X \times Y$. Our candidate arrow is then the composite:

$$X \xrightarrow{s} X \times Y \xrightarrow{\pi_1} Y$$

which we have to show is unique. Using equalisers, we can show that any two arrows f, g out of X to some other object Y are equal:



Let E be the equaliser of f and g, then we get a projection map $i : E \to X$.

By the section principle, this map has a section $s : X \to E$, hence we have:

$$\begin{aligned} f &= \mathsf{id}_X \circ f \\ &= (s \circ i) \circ f \\ &= s \circ (i \circ f) \\ &= s \circ (i \circ g) \\ &= (s \circ i) \circ g \\ &= \mathsf{id}_X \circ g \\ &= g \end{aligned}$$

□

Stating that an object is initial only requires us to define the type of objects and type of morphisms of the category. This makes it an attractive notion to internalise and work with, as defining it will not give us any coherence issues. The section principle however, needs a bit more structure: we need composition and identity morphisms. In order to show that initiality and the section principle coincide, we need even more structure: we need the identity *laws* and associativity and the existence of certain limits.

5.3 Limits in categories of algebras

In this section we will show that the categories of algebras we are working with have products and equalisers, and hence satisfy the assumption of lemma 5.2.2. This is done by induction on the specification of the inductive definition, i.e. by induction on its number of constructors. We will see that we also need that the forgetful functors into the category of sorts preserve these limits, which we can prove simultaneously with the construction of the limits.

5.3.1 Sort categories

For the empty specification, an inductive definition with no constructors, the resulting category of algebras is the category of sorts. We will show that this category has the required limits:

Lemma 5.3.1. *For each sort* $\mathcal{S} : \mathsf{Sorts}$*, the category* $[\![\mathcal{S}]\!]$ *has binary products.*

Proof. We proceed by induction on the specification of sorts $\mathcal{S} : \mathsf{Sorts}$. If $\mathcal{S} = \mathsf{nil}$, then $[\![\mathcal{S}]\!] = \mathbb{1}$, which trivially satisfies our criteria.

In the induction step case, we have $[\![\mathcal{S}]\!] = S_i$ for a category S_i which is built out of the previous category of sorts $S_{i-1} : \mathsf{Cat}$ with $R_i : S_{i-1} \Rightarrow \mathcal{S}et$. By the induction hypothesis S_{i-1} has products and equalisers. We can then define products in S_i as follows: suppose $(X, P), (Y, Q) : |S_i|$, i.e. $X, Y : |S_{i-1}|$ and $P : R_i X \to \mathsf{Set}$, $Q : R_i Y \to \mathsf{Set}$, since S_{i-1} has products, we can define:

$$(X, P) \times (Y, Q) :\equiv (X \times Y, P \times Q)$$

where $P \times Q : R_i(X \times Y) \to \mathsf{Set}$ is defined pointwise as:

$$(P \times Q)\ x :\equiv P(R_i\ \pi_0\ x) \times Q(R_i\ \pi_1\ x)$$

This definition satisfies the universal property of products, which can be shown by appealing to the universal properties of products in S_{i-1} and $\mathcal{S}et$:

let $(Z, T) : |S_i|$, then we have:

$$
\begin{aligned}
& S_i((Z,T),(X \times Y, P \times Q)) \\
= & \{ \text{ definition of products in } S_i \} \\
& (f : S_{i-1}(Z, X \times Y)) \\
& \times (g : (x : R_i Z) \to T\ x \to (P \times Q)\ (R_i\ f\ x)) \\
= & \{ \text{ universal property of } X \times Y \text{ in } S_{i-1} \text{ and functoriality of } R_i \} \\
& (f_0 : S_{i-1}(Z, X)) \times (f_1 : S_{i-1}(Z, Y)) \\
& \times (g : (x : R_i Z) \to T\ x \to P\ (R_i\ f\ x) \times Q\ (R_i\ g\ x)) \\
= & \{ \text{ universal property of } P\ (R_i\ f\ x) \times Q\ (R_i\ g\ x) \text{ in } \mathcal{S}et \} \\
& (f_0 : S_{i-1}(Z, X)) \times (f_1 : S_{i-1}(Z, Y)) \\
& \times (g_0 : (x : R_i Z) \to T\ x \to P\ (R_i\ f\ x)) \\
& \times (g_1 : (x : R_i Z) \to T\ x \to Q\ (R_i\ g\ x)) \\
= & \{ \text{ definition of products in } S_i \} \\
& S_i((Z,T),(X,P)) \times S_i((Z,T),(Y,Q))
\end{aligned}
$$

□

Equalisers are constructed in similar way to products, however it involves equalities between morphisms, which we can simplify using the following proposition:

Proposition 5.3.2. *Suppose* $(f, f'), (g, g') : S_i((X, Y), (Z, W))$*, then:*

$$
\begin{aligned}
((f,f') = (g,g')) = (p : f = g) & \\
\times (p' : (x : X)\ (y : Y\ x) \to f'\ x\ y =^{W}_{R_i\ p\ x} g'\ x\ y) &
\end{aligned}
$$

where we denote the action of R_i *on a proof of equality* $p : f = g$ *by* $R_i\ p\ x : R_i\ f\ x = R_i\ g\ x$*. Since* $f'\ x\ y : W\ (R_i\ f\ x)$ *and* $g'\ x\ y : W\ (R_i\ g\ x)$*, we have to transport the left hand side of the equation along the equality* $R_i\ p\ x$*.*

Proof. This holds by function extensionality and using that an equality of pairs is equivalent a pair of equalities. □

Lemma 5.3.3. *For each sort $\mathcal{S}$: Sorts, the category $[\![\ \mathcal{S}\ ]\!]$ has equalisers.*

Proof. As before, we proceed by induction on the specification of sorts $\mathcal{S}$: Sorts. If $\mathcal{S} = \mathsf{nil}$, then $[\![\ \mathcal{S}\ ]\!] = \mathbb{1}$, which trivially satisfies our criteria.

Given $(f, f'), (g, g') : S_i((X, Y), (Z, W))$, by the induction hypothesis we get an equaliser $E : |S_{i-1}|$ with a projection map $e : S_{i-1}(E, X)$. This equaliser comes equipped with a proof $p : f \circ e = g \circ e$. The equaliser is then defined as (E, F) with:

$$\begin{aligned} F\ x :\equiv\ & (y : Y\ (R_i\ e\ x)) \\ & \times (f'\ (R_i\ e\ x)\ y =^{W}_{R_i\ p\ x} g'\ (R_i\ e\ x)\ y) \end{aligned}$$

with $(e, e') : S_i((E, F), (X, Y))$ the projection morphism where

$$e'\ x\ (y, p) :\equiv y$$

Showing that $(f, f') \circ (e, e') = (g, g') \circ (e, e')$ is then straightforward. The universal property can be shown similarly to that of the product: we have to appeal to the universal properties of equalisers in S_{i-1} and $\mathcal{S}et$. □

Remark 5.3.4. Let $\mathbb{1} \leftarrow S_0 \leftarrow S_1 \leftarrow \ldots \leftarrow S_n$ be a chain of sort categories. All the forgetful functors $t_i : S_i \to S_{i-1}$ (with $S_{-1} :\equiv \mathbb{1}$) preserve products and equalisers. This follows directly from the definition of the limits.

5.3.2 Categories of algebras

For the categories of algebras, we will perform induction on the number of constructors. We also need to simultaneously show that the forgetful functors into the category of sorts preserve the limits.

The assumptions for the following lemmata are:

- $\mathcal{S}$: Sorts is the definition of the sorts
- s : $\mathsf{Spec}_{\mathcal{S}}$ is the specification of the previous constructors, hence
- Alg_s will have products and equalisers

- $\mathsf{U}_s : \mathsf{Alg}_s \Rightarrow [\![\ \mathcal{S}\ ]\!]$ preserves these
- $S_i \in \mathcal{S}$ is the sort of the constructor we are adding

The forgetful functor U_s extends to $\widehat{\mathsf{U}}_s : \mathsf{Alg}_s \Rightarrow S_i$ since $S_i \in \mathcal{S}$. By remark 5.3.4 $\widehat{\mathsf{U}}_s$ also preserves products and equalisers.

Note that $S_i \neq \mathbb{1}$, as this is prevented by the assumption that $S_i \in [\![\ \mathcal{S}\ ]\!]$.

Products

Lemma 5.3.5 (Products of point constructors). *Let $s' : \mathsf{Spec}_{\mathcal{S}}$ be s extended with a point constructor of sort S_i as specified by:*

- $F : \mathsf{Alg}_s \Rightarrow S_i$, *satisfying:*
- $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$

If Alg_s has products and U_s preserves them, then the category $\mathsf{Alg}_{s'}$ has products and its forgetful functor $\mathsf{U}_{s'}$ preserves them.

Proof. Suppose we have two algebras in $\mathsf{Alg}_{s'}$, i.e. we have:

- $X, Y : |\mathsf{Alg}_s|$
- $\theta : S_i(FX, \widehat{\mathsf{U}}_s X)$, $\rho : S_i(FY, \widehat{\mathsf{U}}_s Y)$
- $t_i\theta = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s X)}$, $t_i\rho = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s Y)}$

We will proceed by showing that $X \times Y : \mathsf{Alg}_s$ has an algebra structure. Let us define for X, Y the morphism $\phi_{\widehat{\mathsf{U}}_s} : S_i(\widehat{\mathsf{U}}_s(X \times Y), \widehat{\mathsf{U}}_s X \times \widehat{\mathsf{U}}_s X)$ as $\phi_{\widehat{\mathsf{U}}_s} :\equiv \langle \widehat{\mathsf{U}}_s \pi_0, \widehat{\mathsf{U}}_s \pi_1 \rangle$. Per assumption $\phi_{\widehat{\mathsf{U}}_s}$ has an inverse $\phi^{-1}_{\widehat{\mathsf{U}}_s}$. We can then define the algebra structure ζ on $X \times Y$ as follows:

$$F(X \times Y) \xrightarrow{\phi_F} FX \times FY \xrightarrow{\theta \times \rho} \widehat{\mathsf{U}}_s X \times \widehat{\mathsf{U}}_s Y \xrightarrow{\phi^{-1}_{\widehat{\mathsf{U}}_s}} \widehat{\mathsf{U}}_s(X \times Y)$$

where $\phi_F :\equiv \langle F\pi_0, F\pi_1 \rangle$.

For this to be an algebra structure, we need to establish that $t_i\zeta = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s(X\times Y))}$. This holds as t_i preserves products definitionally. We have:

$$\begin{aligned} t_i(\theta \times \rho) &= t_i\langle \theta \circ \pi_0, \rho \circ \pi_1\rangle \\ &= \langle t_i(\theta \circ \pi_0), t_i(\rho \circ \pi_1)\rangle \\ &= \langle \pi_0, \pi_1\rangle \\ &= \mathsf{id}_{t_I(\widehat{\mathsf{U}}_s(X\times Y))} \end{aligned}$$

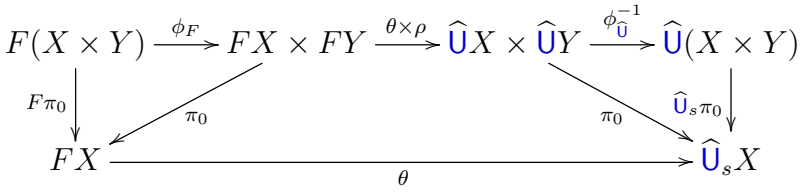
and

$$\begin{aligned} t_i\phi_F &= t_i\langle F\pi_0, F\pi_1\rangle \\ &= \langle t_iF\pi_0, t_iF\pi_1\rangle \\ &= \langle t_i\widehat{\mathsf{U}}_s\pi_0, t_i\widehat{\mathsf{U}}_s\pi_1\rangle \\ &= t_i\phi_{\widehat{\mathsf{U}}_s} \end{aligned}$$

Putting it all together, we get:

$$\begin{aligned} t_i\zeta &= t_i(\phi_u^{-1} \circ \theta \times \rho \circ \phi_F) \\ &= t_i\phi_u^{-1} \circ t_i(\theta \times \rho) \circ t_i\phi_F \\ &= t_i\phi_{\widehat{\mathsf{U}}_s}^{-1} \circ \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s(X\times Y))} \circ t_i\phi_{\widehat{\mathsf{U}}_s} \\ &= \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s(X\times Y))} \end{aligned}$$

We furthermore have to check whether $\pi_0 : S_i(X\times Y, X)$ and $\pi_1 : S_i(X\times Y, Y)$ are algebra morphisms. To this end, let us consider the following diagram:



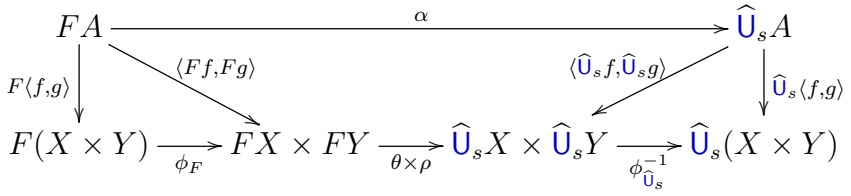
The triangle on the left holds by definition of ϕ_F. The square (trapezoid) in the middle holds by definition of $\theta \times \rho$. The triangle on the right can be

established as follows: we have:

$$\begin{aligned}\pi_0 \circ \phi_{\widehat{\mathsf{U}}_s} &= \widehat{\mathsf{U}}_s \pi_0 \\ &= \widehat{\mathsf{U}}_s \pi_0 \circ \phi^{-1}_{\widehat{\mathsf{U}}_s} \circ \phi_{\widehat{\mathsf{U}}_s}\end{aligned}$$

Since $\phi_{\widehat{\mathsf{U}}_s}$ is an isomorphism it is in particular a monomorphism, hence from this equation we get that $\pi_0 = \widehat{\mathsf{U}}_s \pi_0 \circ \phi_{\widehat{\mathsf{U}}_s}{}^{-}1$. This shows that π_0 is an algebra morphism $\zeta \to \theta$. The proof that π_1 is an algebra morphism $\zeta \to \rho$ goes along the same lines.

Finally we have to show that the universal property is satisfied. Suppose we have an algebra $(A, \alpha) : |\mathsf{Alg}_{s'}|$ with $f : \mathsf{Alg}_s(A, X)$ and $g : \mathsf{Alg}_s(A, Y)$ both algebra morphisms. We get a unique arrow $\langle f, g\rangle : \mathsf{Alg}_s(A, X \times Y)$. We then have to ascertain that this is an algebra morphism $\alpha \to \zeta$, i.e. we have to show that the following commutes:



The left triangle holds by definition. The middle trapezoid holds as f and g are algebra morphisms $\alpha \to \theta$ and $\alpha \to \rho$ respectively. The right triangle can be shown to hold by the following equational reasoning:

$$\begin{aligned}\phi_{\widehat{\mathsf{U}}_s} \circ \widehat{\mathsf{U}}_s\langle f, g\rangle &= \langle \widehat{\mathsf{U}}_s f, \widehat{\mathsf{U}}_s g\rangle \\ &= \phi_{\widehat{\mathsf{U}}_s} \circ \phi^{-1}_{\widehat{\mathsf{U}}_s} \circ \langle \widehat{\mathsf{U}}_s f, \widehat{\mathsf{U}}_s g\rangle\end{aligned}$$

Since $\phi_{\widehat{\mathsf{U}}_s}$ is an isomorphism, it is also an epimorphism, which means we get $\widehat{\mathsf{U}}_s\langle f, g\rangle = \phi^{-1}_{\widehat{\mathsf{U}}_s} \circ \langle \widehat{\mathsf{U}}_s f, \widehat{\mathsf{U}}_s g\rangle$.

Note that by construction, $\widehat{\mathsf{U}}_{s'}$ preserves products definitionally. □

Lemma 5.3.6 (Products of path constructors)**.** *Let s' : $\mathsf{Spec}_{\mathcal{S}}$ be s extended with a point constructor of sort S_i as specified by:*

- $F : \mathsf{Alg}_s \Rightarrow S_i$*, satisfying* $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$

- $l, r : F \dot{\to} \widehat{\mathsf{U}}_s$ *satisfying* $t_i\ l = t_i\ r = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s}$.

If Alg_s *has products and* U_s *preserves them, then the category* $\mathsf{Alg}_{s'}$ *has products and its forgetful functor* $\mathsf{U}_{s'}$ *preserves them.*

Proof. Suppose we have objects $X, Y : |\mathsf{Alg}_s|$ with algebra structures $\theta : \ell_X = r_X$ and $\rho : \ell_Y = r_Y$. We will proceed by constructing an algebra on the object $X \times Y$.

Note that by naturality of ℓ, the following commutes:

$$\begin{array}{ccc} F(X \times Y) & \xrightarrow{\phi_F} & FX \times FY \\ {\scriptstyle \ell_{X\times Y}}\downarrow & & \downarrow{\scriptstyle \ell_X \times \ell_Y} \\ \widehat{\mathsf{U}}_s(X \times Y) & \xrightarrow[\phi_{\widehat{\mathsf{U}}_s}]{} & \widehat{\mathsf{U}}_s X \times \widehat{\mathsf{U}}_s X \end{array}$$

Postcomposing with $\phi^{-1}_{\widehat{\mathsf{U}}_s}$, we get the equation:

$$\ell_{X\times Y} = \phi^{-1}_{\widehat{\mathsf{U}}_s} \circ \ell_X \times \ell_Y \circ \phi_F$$

Since we have $\ell_X = r_X$ and $\ell_Y = r_Y$, we also have $\ell_{X\times Y} = r_{X\times Y}$.

Now the category of algebras $\mathsf{Alg}_{s'}$ is a full subcategory of Alg_s, hence we get automatically that the projections are algebra morphisms and that the universal property is satisfied.

By construction of the products, the forgetful functor $\mathsf{U}_{s'}$ preserves products definitionally. □

Equalisers

To talk about equalisers in a category of algebras, we need to know what equality of algebra morphisms is. An algebra morphism is a pair of a morphism in the underlying category and an equality stating that this morphism preserves the algebra structure. If we have two pairs, since equality of pairs is a pair of equalities, an equality between them consists of an equality between the morphisms and an equality between the proofs that they preserve the algebra structure. Since we are working with types that

satisfy uniqueness of identity proofs, we can ignore the latter part, hence an equality between two algebra morphisms is just an equality of the two underlying morphisms. This reasoning applies to both point constructor algebra morphisms as well as path constructor algebra morphisms.

Lemma 5.3.7 (Equalisers of point constructors)**.** *Let s' : $\mathsf{Spec}_{\mathcal{S}}$ be s extended with a point constructor of sort S_i as specified by:*

- $F : \mathsf{Alg}_s \Rightarrow S_i$*, satisfying:*
- $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$

If Alg_s has equalisers and U_s preserves them, then the category $\mathsf{Alg}_{s'}$ has equalisers and its forgetful functor $\mathsf{U}_{s'}$ preserves them.

Proof. Suppose we have two algebras in $\mathsf{Alg}_{s'}$, i.e. we have:

- $X, Y : |\mathsf{Alg}_s|$
- $\theta : S_i(FX, \widehat{\mathsf{U}}_s X)$, $\rho : S_i(FY, \widehat{\mathsf{U}}_s Y)$
- $t_i\theta = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s X)}$, $t_i\rho = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s Y)}$

along with morphisms $f, g : \mathsf{Alg}_s(X, Y)$ satisfying:

- $\widehat{\mathsf{U}}_s f \circ \theta = \rho \circ Ff$
- $\widehat{\mathsf{U}}_s g \circ \theta = \rho \circ Fg$

i.e. they are algebra morphisms $\theta \to \rho$.

Define $E : \mathsf{Alg}_s$ to be the equaliser of f and g with inclusion $e : \mathsf{Alg}_s(E, X)$ which satisfies $f \circ e = g \circ e$. We need to show that E has an algebra structure and e is an algebra morphism of this structure into θ. The following diagram commutes by virtue of E being an equaliser and f and g being algebra morphisms:

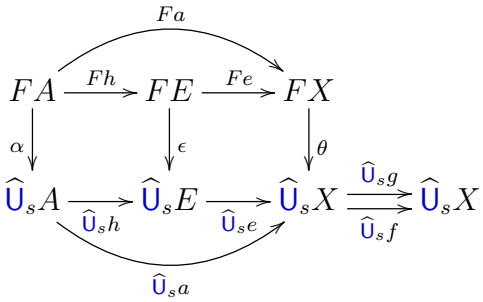
$$\begin{array}{ccccc} FE & \xrightarrow{Fe} & FX & \underset{Ff}{\overset{Fg}{\rightrightarrows}} & FY \\ & & \downarrow{\scriptstyle\theta} & & \downarrow{\scriptstyle\rho} \\ \widehat{\mathsf{U}}_s E & \xrightarrow{\widehat{\mathsf{U}}_s e} & \widehat{\mathsf{U}}_s X & \underset{\widehat{\mathsf{U}}_s f}{\overset{\widehat{\mathsf{U}}_s g}{\rightrightarrows}} & \widehat{\mathsf{U}}_s X \end{array}$$

Since $\widehat{\mathsf{U}}_s$ preserves equalisers, $\widehat{\mathsf{U}}_s E$ is an equaliser, hence we get a unique arrow $\epsilon : S_i(FE, \widehat{\mathsf{U}}_s E)$. We need to check that we have $t_i\epsilon = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s E)}$. We know that the following commutes:

$$\begin{array}{ccccc} t_i(FE) & \xrightarrow{t_i(Fe)} & t_i(FX) & & \\ \downarrow{\scriptstyle t_i\epsilon} & & \downarrow{\scriptstyle t_i\theta} & & \\ t_i(\widehat{\mathsf{U}}_s E) & \xrightarrow{t_i(\widehat{\mathsf{U}}_s e)} & t_i(\widehat{\mathsf{U}}_s X) & \underset{t_i\widehat{\mathsf{U}}_s f}{\overset{t_i\widehat{\mathsf{U}}_s g}{\rightrightarrows}} & t_i(\widehat{\mathsf{U}}_s X) \end{array}$$

Furthermore, $t_i\theta = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s X)}$ and $t_i(Fe) = t_i(\widehat{\mathsf{U}}_s e)$, and since t_i preserves equalisers, we necessarily have that $t_i\epsilon = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s E)}$.

We still have to establish that the universal property is satisfied. Suppose we have $(A, \alpha) : |\mathsf{Alg}_{s'}|$ along with a an algebra morphism $\alpha \to \theta$, by the universal property of E, we get a unique arrow $h : \mathsf{Alg}_s(A, E)$, satisfying $a = e \circ h$. We have to establish whether this is an algebra morphism $\alpha \to \epsilon$. Consider the following diagram:



We have yet to establish whether the left square commutes, the other subdiagrams have already been shown to commute. Observe that we have two cones $(FA, \theta \circ Fa)$ and $(FA, \widehat{\mathsf{U}}_s a \circ \alpha)$ for the fork $\widehat{\mathsf{U}}_s f, \widehat{\mathsf{U}}_s g$. Since a is an algebra morphism, these two cones are actually the same. Both $\epsilon \circ Fh$ and $\widehat{\mathsf{U}}_s h \circ \alpha$ are then cone morphisms from this cone into $(\widehat{\mathsf{U}}_s E, \widehat{\mathsf{U}}_s e)$, which by the universal property of $\widehat{\mathsf{U}}_s E$ means that $\widehat{\mathsf{U}}_s h \circ \alpha = \epsilon \circ Fh$, which establishes the commutativity of the left square and thereby the universal property of equalisers in $\mathsf{Alg}_{s'}$. □

Lemma 5.3.8 (Equalisers of path constructors)**.** *Let $s' : \mathsf{Spec}_S$ be s extended*

with a path constructor of sort S_i as specified by:

- $F : \mathsf{Alg}_s \Rightarrow S_i$, *satisfying* $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$
- $l, r : F \dashrightarrow \widehat{\mathsf{U}}_s$ *satisfying* $t_i\, l = t_i\, r = \mathsf{id}_{t_i \circ \widehat{\mathsf{U}}_s}$.

If Alg_s has equalisers and U_s preserves them, then the category $\mathsf{Alg}_{s'}$ has equalisers and its forgetful functor $\mathsf{U}_{s'}$ preserves them.

Proof. Suppose we have objects $X, Y : |\mathsf{Alg}_s|$ with algebra structures $\theta : \ell_X = r_X$ and $\rho : \ell_Y = r_Y$ and morphisms $f, g : \mathsf{Alg}_s(X, Y)$. We claim that the equaliser $E : \mathsf{Alg}_s$ of f and g has an algebra structure, i.e. we can show that $\ell_E = r_E$.

Since $\ell_X = r_X$ and the naturality of l and r, we have that the following commutes:

$$\begin{array}{ccccc} FE & \xrightarrow{Fe} & FX & & \\ {\scriptstyle \ell_E}\downarrow & & {\scriptstyle \ell_X}\downarrow\downarrow{\scriptstyle r_X} & & \\ \widehat{\mathsf{U}}_s E & \xrightarrow{\widehat{\mathsf{U}}_s e} & \widehat{\mathsf{U}}_s X & \underset{\widehat{\mathsf{U}}_s f}{\overset{\widehat{\mathsf{U}}_s g}{\rightrightarrows}} & \widehat{\mathsf{U}}_s X \end{array}$$

We observe that $(FE, \ell_X \circ Fe)$ is a cone for the fork $\widehat{\mathsf{U}}_s f, \widehat{\mathsf{U}}_s g$. By $\ell_X = r_X$, $(FE, r_X \circ Fe)$ is the same fork. We then have that by naturality, both ℓ_E and r_E are cone morphisms from $(FE, \ell_X \circ Fe)$ into the terminal cone $(\widehat{\mathsf{U}}_s E, \widehat{\mathsf{U}}_s e)$, hence by universality we get that $\ell_E = r_E$.

Since $\mathsf{Alg}_{s'}$ is a full subcategory of Alg_s, E with its proof $\epsilon : \ell_E = r_E$ inherits its universal property directly from Alg_s. □

Together with lemma 5.2.1 and lemma 5.2.2, this immediately gives the main theorem of this section:

Theorem 5.3.9. *For each sort $\mathcal{S} : \mathsf{Sorts}$ and specification $s : \mathsf{Spec}$, let X be an object in the category of algebras Alg_s. Then X is initial if and only if X satisfies the section principle.* □

In particular, this means that when implementing or formalising quotient inductive-inductive types, one can restrict attention to the conceptually simpler notion of initial algebra.

5.4 Deriving the induction principle

If we unfold the section principle for our categories of algebras, we will not end up with a very natural induction principle. One of the key features of type theory is that we have special machinery for dealing with type families/fibrations. If we want to define a predicate on a type X, the most natural way to do it is usually to define a family $X \to \mathsf{Set}$, not to define another type Y with a function $Y \to X$.

Since our categories of sorts and algebras are built out of types, we can try and derive a notion of families in these categories. In $\mathcal{S}et$, we have the equivalence, for any $X : \mathsf{Set}$:

$$(X \to \mathsf{Set}) = (Y : \mathsf{Set}) \times (p : Y \to X)$$

where the function left to right is defined as $\lambda P.((x : X) \times P\ x, \pi_0)$, i.e. we map the family to its "total space" and its projection function. The inverse operation is the preimage family: (Y, p) is mapped to $\lambda x.(y : Y) \times (p\ y = x)$.

Apart from having the machinery to deal with type families, we also have the machinery to deal with dependent functions, or equivalently, sections of fibrations. We have:

$$(x : X) \to P\ x = (s : X \to (x : X) \times P\ x) \times (\pi_0 \circ s = \mathsf{id}_X)$$

where the map from left to right is the map that gives the "graph" of the function: a dependent function s is mapped to the non-dependent function $\lambda x.(x, s\ x)$. The map in the other direction is the second projection π_1, which is well-typed due to s being a section.

We will start out with using this observation to make precise the derivation of algebra families and dependent algebra morphisms in F-alg for an endofunctor $F : \mathcal{S}et \Rightarrow \mathcal{S}et$ in section 5.4.1. We will then give a general framework containing the ingredients for derivations of the induction principle in section 5.4.2. Finally, in section 5.4.3, we give some examples of induction principles of some quotient inductive-inductive definitions and then employ the general framework to derive the induction principle for

arbitrary quotient inductive-inductive definitions in its full generality.

5.4.1 Induction for F-algebras

If we want to derive the notion of families of F-algebras for some endofunctor $F : \mathcal{S}et \Rightarrow \mathcal{S}et$, then we proceed as follows: let $(X, \theta) : |F\text{-alg}|$, we want to solve the equivalence for the type τ:

$$\begin{aligned}&(P : X \to \mathsf{Set}) \times (m : \tau)\\ =\ &((Y, \rho) : |F\text{-alg}|) \times ((p, p_0) : F\text{-alg}((Y, \rho), (X, \theta)))\end{aligned}$$

The idea here is that we want to determine the hypotheses needed to prove $(x : X) \to P\ x$. The type τ that we will derive below, is the type of hypotheses.

We set Y to $(x : X) \times P\ x$ and p to π_0. We then have to simplify the type $(\rho : F((x : X) \times P\ x)) \times (p_0 : \pi_0\ \circ \rho = \theta \circ F\ \pi_0)$, which leads us to:

$$m : (x : F((x : X) \times P\ x)) \to P\ (F\ \pi_0\ x)$$

This can be simplified even further if we define the operation $\Box_F : (X \to \mathsf{Set}) \to FX \to \mathsf{Set}$, i.e. the action of F on type families, which has the defining equation $F((x : X) \times P\ x) = (x : FX) \times \Box_F\ P\ x$. Rewriting the type of m using this notation gives us:

$$m : (x : FX) \times \Box_F\ P\ x \to P\ (\theta\ x)$$

Intuitively the type $\Box_F\ P\ x$ is the induction hypothesis. We can also read it as the modality "all": $\Box_F\ P\ x$ holds if P holds for all values of type X "contained" in x.

Now that we know what the algebra families are, we want to go on and derive the dependent algebra morphisms. Let $(X, \theta) : |F\text{-alg}|$ with an algebra family $P : X \to \mathsf{Set}$, $m : (x : FX) \times \Box_F\ P\ x \to P\ (\theta\ x)$. Our goal is

to find a type τ such that the following equivalence holds:

$$((s : (x : X) \to P\ x) \times \tau) = (f : F\text{-}\mathsf{alg}((X, \theta), (\Sigma X.P, \tilde{m}))) \times (\text{is-section}\ f)$$

where $(\Sigma X.P, \tilde{m})$ is the total space of the family of algebras (P, m), where $\tilde{m}$ is given as the composite:

$$F(\Sigma X.P) \xrightarrow{\phi} \Sigma(FX).\Box_F\ P \xrightarrow{\langle \theta \circ \pi_0, m \rangle} \Sigma X.P$$

where ϕ is the map given by the defining equivalence of $\Box_F$.

As $\Box_F$ gives the functorial action of F on *families*, we need a similar functorial actions on dependent functions, namely $\overline{F} : ((x : X) \to P\ x) \to (x : FX) \to \Box_F\ P\ x$. $\overline{F}$ has the defining property:

$$\text{graph}\ (\overline{F}\ s) = \phi \circ F(\text{graph}\ s)$$

where $\text{graph}\ s = \lambda x.(x, s\ x)$.

Using this, we can derive dependent morphisms for F-alg as follows:

$$
\begin{aligned}
&(f : F\text{-alg}((X,\theta),((x:X)\times P\ x,\tilde{m}))) \times (\text{is-section}_{F\text{-alg}}\ (\pi_0, \text{refl})\ f)\\
=\ &\{\ \text{definition of } F\text{-alg and is-section}_{F\text{-alg}}\ \}\\
&(f : X \to (x:X)\times P\ x) \times (f \circ \theta = \tilde{m} \circ Ff) \times (\pi_0 \circ f = \text{id}_X)\\
=\ &\{\ \text{dependent morphism structure on } \mathcal{S}et\ \}\\
&(s : (x:X) \to P\ x) \times (s_0 : \text{graph}\ s \circ \theta = \tilde{m} \circ F(\text{graph}\ s))\\
=\ &\{\ \text{equality of pairs is pair of equalities}\ \}\\
&(s : (x:X) \to P\ x)\\
&\times (s_0 : \pi_0 \circ \text{graph}\ s \circ \theta = \pi_0 \circ \tilde{m} \circ F(\text{graph}\ s))\\
&\times (s_1 : \pi_1 \circ \text{graph}\ s \circ \theta = \pi_1 \circ \tilde{m} \circ F(\text{graph}\ s))\\
=\ &\{\ \pi_0 \text{ is an algebra morphism } \tilde{m} \to \theta \text{ and } \pi_0 \circ \text{graph}\ s = \text{id}_X\ \}\\
&(s : (x:X) \to P\ x) \times (s_0 : \theta = \theta) \times (s_1 : \pi_1 \circ \text{graph}\ s \circ \theta = \pi_1 \circ \tilde{m} \circ F(\text{graph}\ s))\\
=\ &\{\ \text{unfold definition } \tilde{m},\ \theta = \theta \text{ is trivial by uniqueness of identity proofs}\ \}\\
&(s : (x:X) \to P\ x) \times (s_1 : \pi_1 \circ \text{graph}\ s \circ \theta = \pi_1 \circ \langle \theta \circ \pi_0, m\rangle \circ \phi \circ F(\text{graph}\ s))\\
=\ &\{\ \text{computation rule of } \langle \theta \circ \pi_0, m\rangle,\ \pi_1 \circ \text{graph}\ s = s, \text{ defining property of } \overline{F}\ \}\\
&(s : (x:X) \to P\ x) \times (s_1 : s \circ \theta = m \circ \text{graph}(\overline{F}\ s))
\end{aligned}
$$

If we η-expand and β-reduce the definitions, we see that a dependent algebra morphism on an algebra family (P, m) on an algebra (X, θ) consists of a dependent function $s : (x : X) \to P\ x$ with computation rule $s\ (\theta\ x) = m\ x\ (\overline{F}\ s\ x)$.

5.4.2 General framework

To derive notions of families and dependent morphism in the categories of algebras, we will introduce some notation, making precise what we have to generalise. We want to define a type of families over an object that is equivalent to the type of morphisms into that same object. We will call such a structure the *family structure* on that category. This definition on its own is not very useful: we can always trivially satisfy the definition by

letting the type of families exactly be the type of morphisms into an object. However, our categories are not completely arbitrary: they are always in some way built out of objects and morphisms in $\mathcal{S}et$. We can therefore use the notion of families in $\mathcal{S}et$ for those parts and see how the remaining parts simplify.

Definition 5.4.1 (Family structure on a category)**.** The operations we need from a category $\mathcal{C}$ to talk about families are as follows:

- $\mathsf{Fam}_{\mathcal{C}} : |\mathcal{C}| \to \mathsf{Set}$
- $\mathsf{total} : \{X : |\mathcal{C}|\} \to \mathsf{Fam}_{\mathcal{C}}\ X \to |\mathcal{C}|$
- $\mathsf{proj} : \{X : |\mathcal{C}|\}\ (P : \mathsf{Fam}_{\mathcal{C}}\ X) \to \mathcal{C}(\mathsf{total}\ P, X)$
- $\mathsf{preimage} : \{X : |\mathcal{C}|\}\ (Y : |\mathcal{C}|)\ (p : \mathcal{C}(Y, X)) \to \mathsf{Fam}_{\mathcal{C}}\ X$

The operations should also satisfy the following correctness conditions, for any $X : |\mathcal{C}|$:

- for any family $P : \mathsf{Fam}_{\mathcal{C}}\ X$,

$$\mathsf{preimage}\ X\ (\mathsf{total}\ P)\ (\mathsf{proj}\ P) = P$$

- for any object $Y : |\mathcal{C}|$ with $p : \mathcal{C}(Y, X)$,

$$\begin{aligned}&(\mathsf{total}\ X\ (\mathsf{preimage}\ Y\ p), \mathsf{proj}\ X\ (\mathsf{preimage}\ Y\ p))\\ &= (Y, p)\end{aligned}$$

A family structure as defined above gives us an equivalence $\mathsf{Fam}_{\mathcal{C}}\ X = (Y : |\mathcal{C}|) \times (p : \mathcal{C}(Y, X))$. We will sometimes explicitly define the operations total, proj and $\mathsf{preimage}$, but sometimes only some of them and will give the definition of $\mathsf{Fam}_{\mathcal{C}}$ along with the equivalence reasoning that lead up to that definition. From this equivalence proof, we can reconstruct the operations.

To define these operations for the sort and algebra categories, we can perform induction on the specification and apply the same techniques as we did to derive the notion of families in F-alg.

Given these operations, we can generalise the $\Box$ operator to any functor $F : \mathcal{C} \Rightarrow \mathcal{D}$.

Definition 5.4.2 (Functorial action on families)**.** Given family structures $\mathsf{Fam}_{\mathcal{C}}$ and $\mathsf{Fam}_{\mathcal{D}}$, and $X : |\mathcal{C}|$ and $P : \mathsf{Fam}_{\mathcal{C}}\ X$, $\Box_F : (X : |\mathcal{C}|) \to \mathsf{Fam}_{\mathcal{C}} \to \mathsf{Fam}_{\mathcal{D}}$ is defined as:

$$\Box_F\ P :\equiv \mathsf{preimage}_{\mathcal{D}}\ (F\ (\mathsf{total}_{\mathcal{C}}\ P), F\ (\mathsf{proj}_{\mathcal{C}}\ P))$$

Proposition 5.4.3. *Given family structures* $\mathsf{Fam}_{\mathcal{C}}$ *and* $\mathsf{Fam}_{\mathcal{D}}$*, and* $X : |\mathcal{C}|$ *and* $P : \mathsf{Fam}_{\mathcal{C}}\ X$*, we have*

$$F(\mathsf{total}_{\mathcal{C}}\ P) = \mathsf{total}_{\mathcal{D}}\ (\Box_F\ P)$$

Proof. This follows directly from the correctness condition of the family structure on $\mathcal{D}$:

$$\begin{aligned}
&F(\mathsf{total}_{\mathcal{C}}\ P) \\
=\ &\{\ \text{correctness condition of } \mathsf{Fam}_{\mathcal{D}}\ \} \\
&\mathsf{total}_{\mathcal{D}}\ (\mathsf{preimage}_{\mathcal{D}}\ (F(\mathsf{total}_{\mathcal{C}}\ P), F(\mathsf{proj}_{\mathcal{C}}\ P))) \\
=\ &\{\ \text{definition of } \Box_F\ \} \\
&\mathsf{total}_{\mathcal{D}}\ (\Box_F\ P)
\end{aligned}$$

□

Corollary 5.4.4. *Applying proposition 5.4.3 to endofunctors on* $\mathcal{S}et$*, we get, given* $F : \mathcal{S}et \Rightarrow \mathcal{S}et$*,* $X : \mathsf{Set}$*,* $P : X \to \mathsf{Set}$*:*

$$F((x : X) \times Px) = (x : FX) \times \Box_F\ P\ x$$

Having defined families, we can then go on to generalise the notion of dependent functions. In $\mathcal{S}et$, given a family $P : X \to \mathsf{Set}$, a dependent function $(x : X) \to P\ x$ corresponds to a function $s : X \to (x : X) \times P\ x$ along with $s_0 : \pi_0 \circ s = \mathsf{id}_X$. The function from left to right sends a dependent function s to its graph $\lambda x.(x, s\ x)$, which is trivially a section.

The other direction composes the section s with π_1, but we have to transport along the proof given by s_0 to make it typecheck.

Definition 5.4.5. The type of *sections* in a category $\mathcal{C}$ is defined as:

$$\mathsf{Sect}_{\mathcal{C}} : \{X : |\mathcal{C}|\}\ (Y : |\mathcal{C}|)\ (p : \mathcal{C}(Y, X)) \to \mathsf{Set}$$
$$\mathsf{Sect}_{\mathcal{C}}\ Y\ p :\equiv (s : \mathcal{C}(X, Y)) \times (s_0 : p \circ s = \mathsf{id}_X)$$

A dependent morphism structure on a category with a family structure is something that allows us to perform the same construction as in $\mathcal{S}et$:

Definition 5.4.6 (Dependent morphism structure)**.** To generalise dependent morphisms to a category $\mathcal{C}$, we need a family $\mathsf{DepHom}_{\mathcal{C}} : \{X : |\mathcal{C}|\}\ (P : \mathsf{Fam}_{\mathcal{C}}\ X) \to \mathsf{Set}$ along with the following operations, given an object $X : |\mathcal{C}|$ and a family $P : \mathsf{Fam}_{\mathcal{C}}\ X$:

- $\mathsf{graph} : \mathsf{DepHom}\ P \to \mathsf{Sect}\ (\mathsf{total}\ P, \mathsf{proj}\ P)$
- $\mathsf{snd} : \mathsf{Sect}\ (\mathsf{total}\ P, \mathsf{proj}\ P) \to \mathsf{DepHom}_{\mathcal{C}}\ X\ P$

The correctness conditions are:

- for any $f : \mathsf{DepHom}_{\mathcal{C}}\ P$,
$$\mathsf{snd}\ P\ s\ (\mathsf{graph}\ P\ f) = f$$
- for any $s : \mathsf{Sect}\ (\mathsf{total}\ P, \mathsf{proj}\ P)$,
$$\mathsf{graph}\ P\ (\mathsf{snd}\ P\ s) = s$$

Note that the correctness conditions give us an equivalence

$$\mathsf{DepHom}_{\mathcal{C}}\ P = \mathsf{Sect}_{\mathcal{C}}\ X\ (\mathsf{proj}_{\mathcal{C}}\ P)$$

Similar to the action of a functor F on families, we can use this structure to define the action of functors on dependent morphisms. Given a functor

$F : \mathcal{C} \Rightarrow \mathcal{D}$, we can define:

$$\overline{F} : (X : |\mathcal{C}|)\ (P : \mathsf{Fam}_{\mathcal{C}}\ X) \to \mathsf{DepHom}_{\mathcal{C}}\ P \to \mathsf{DepHom}_{\mathcal{C}}\ FX\ (\Box_F\ P)$$

If F happens to be an endofunctor on $\mathcal{S}et$, this simplifies to:

$$\overline{F} : (X : \mathsf{Set})\ (P : X \to \mathsf{Set}) \to ((x : X) \to P\ x) \to (x : FX) \to \Box_F\ P\ x$$

Before we define $\overline{F}$, observe that we can easily define a function:

$$\mathsf{Sect}_{\mathcal{C}}(s, p) \to \mathsf{Sect}_{\mathcal{D}}\ (Fs, Fp)$$

$\overline{F}$ can then be defined as the composite:

$$\begin{array}{c}
\mathsf{DepHom}_{\mathcal{C}}\ P \\
\downarrow{\scriptstyle \mathsf{graph}} \\
\mathsf{Sect}_{\mathcal{C}}\ (\mathsf{total}_{\mathcal{C}}\ P, \mathsf{proj}_{\mathcal{C}}\ P) \\
\downarrow \\
\mathsf{Sect}_{\mathcal{D}}\ (F(\mathsf{total}_{\mathcal{C}}\ P), F\ (\mathsf{proj}_{\mathcal{C}}\ P)) \\
\downarrow{\scriptstyle \phi} \\
\mathsf{Sect}_{\mathcal{D}}\ (\mathsf{total}_{\mathcal{D}}\ (\Box_F\ P), \mathsf{proj}_{\mathcal{D}}\ (\Box_F\ P)) \\
\downarrow{\scriptstyle \mathsf{snd}} \\
\mathsf{DepHom}_{\mathcal{D}}\ (\Box_F\ P)
\end{array}$$

The map ϕ we get from the equivalence given in proposition 5.4.3.

5.4.3 Induction for quotient inductive-inductive definitions

The family structure on $\mathcal{S}et$ is the usual one. We can derive definitions of the family structure on categories of algebras by induction. Since they are defined "on top of" $\mathcal{S}et$, we can use the specification and the family structure on $\mathcal{S}et$ to derive appropriate definitions. We will start out by considering the induction principles for the contexts and types example

and the interval type.

Induction principle for (Con, Ty)

The induction principle consists of four parts:

- the *motive*, which is intuitively the property that we are trying to prove holds for every element of the inductive definition
- the *methods*: we show for every constructor that the motive holds for that constructor, given the assumption that the motive holds for the recursive arguments
- given the motive and methods, we get a *dependent morphism* from the inductive type into the motive
- this dependent morphism comes with *computation rules*, telling us how this dependent morphism behaves in conjunction with the constructors.

Consider the inductive definition of the syntax of type theory, restricting ourselves to contexts and types in a context, i.e. we have sorts $\mathsf{Con} : \mathsf{Set}$ and $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}$ with constructors:

$$\begin{aligned}
&\epsilon : \mathsf{Con}\\
&_,_ : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\ \Gamma \to \mathsf{Con}\\
&\text{`}\Pi : (\Gamma : \mathsf{Con})\ (A : \mathsf{Ty}\ \Gamma)\ (B : \mathsf{Ty}\ (\Gamma,A)) \to \mathsf{Ty}\ \Gamma
\end{aligned}$$

The motives for this definition are:

$$\begin{aligned}
&P : \mathsf{Con} \to \mathsf{Set}\\
&Q : (\Gamma : \mathsf{Con}) \to P\ \Gamma \to \mathsf{Ty}\ \Gamma \to \mathsf{Set}
\end{aligned}$$

P is a family over Con and Q is a family over both P and Ty. In the motive for Ty we are allowed to refer to results of the induction principle applied to the context at hand.

The methods are as follows:

$$
\begin{aligned}
&m_{\epsilon} : P\ \epsilon \\
&m_{\_,\_} : (\Gamma : \mathsf{Con})\ (x : P\ \Gamma) \\
&\qquad (A : \mathsf{Ty}\ \Gamma)\ (y : Q\ \Gamma\ x\ A) \\
&\qquad \to P\ (\Gamma,A) \\
&m_{`\Pi} : (\Gamma : \mathsf{Con})\ (x : P\ \Gamma) \\
&\qquad (A : \mathsf{Ty}\ \Gamma)\ (y : Q\ \Gamma\ x\ A) \\
&\qquad (B : \mathsf{Ty}\ (\Gamma,A))\ (z : Q\ (\Gamma,A)\ (m_{\_,\_}\ \Gamma\ x\ A\ y)\ B) \\
&\qquad \to Q\ \Gamma\ x\ (`\Pi\ \Gamma\ A\ B)
\end{aligned}
$$

The method for the constructor ϵ is unsurprising. For $\_,\_$ we see that its method, just like the constructor itself, refers to the motive of Ty. The method for $`\Pi$ refers to a previous *method*, reflecting the reference to the corresponding previous constructor.

Given these motives and methods, the induction principle gives us the following dependent functions:

$$
\begin{aligned}
&\mathsf{Con\text{-}ind} : (\Gamma : \mathsf{Con}) \to P\ \Gamma \\
&\mathsf{Ty\text{-}ind} : (\Gamma : \mathsf{Con})\ (A : \mathsf{Ty}\ \Gamma) \to Q\ \Gamma\ (\mathsf{Con\text{-}ind}\ \Gamma)\ A
\end{aligned}
$$

with computation rules:

$$
\begin{aligned}
&\mathsf{Con\text{-}ind}\ \epsilon = m_{\epsilon} \\
&\mathsf{Con\text{-}ind}\ (\Gamma,A) = m_{\_,\_}\ \Gamma\ (\mathsf{Con\text{-}ind}\ \Gamma)\ A\ (\mathsf{Ty\text{-}ind}\ \Gamma\ A) \\
&\mathsf{Ty\text{-}ind}\ \Gamma\ (`\Pi\ \Gamma\ A\ B) = m_{`\Pi}\ \Gamma\ (\mathsf{Con\text{-}ind}\ \Gamma)\ A\ (\mathsf{Ty\text{-}ind}\ \Gamma\ A)\ B\ (\mathsf{Ty\text{-}ind}\ (\Gamma,A)\ B)
\end{aligned}
$$

Reading these equations as a recursive definition of the morphisms, we see that it is a mutual definition. Another thing to note is that the last equation, for the constructor $`\Pi$, has repeating variables on the left hand side, so we cannot read it strictly as a pattern matching definition, but instead it is a definition given by dependent pattern matching. Unifying the type of the

constructor ‘Π with the signature of Ty-ind, we see that the two variables of type Con have to be the same.

Induction principle for the interval type

Let us recall the inductive definition of the interval I with constructors:

$$\begin{aligned} &\mathsf{zero} : \mathsf{I} \\ &\mathsf{one} : \mathsf{I} \\ &\mathsf{seg} : \mathsf{zero} = \mathsf{one} \end{aligned}$$

Since it is just a $\mathcal{S}et$-sorted inductive definition, a motive for induction on I is a family $P : \mathsf{I} \to \mathsf{Set}$. The methods are:

$$\begin{aligned} &m_{\mathsf{zero}} : P\ \mathsf{zero} \\ &m_{\mathsf{one}} : P\ \mathsf{one} \\ &m_{\mathsf{seg}} : m_{\mathsf{zero}} =^{P}_{\mathsf{seg}} m_{\mathsf{one}} \end{aligned}$$

which gives us a dependent function $\mathsf{I\text{-}ind} : (x : \mathsf{I}) \to P\ x$ with computation rules:

$$\begin{aligned} &\mathsf{I\text{-}ind}\ \mathsf{zero} = m_{\mathsf{zero}} \\ &\mathsf{I\text{-}ind}\ \mathsf{one} = m_{\mathsf{one}} \end{aligned}$$

The induction principle for I is not very different from its recursion principle. The most interesting difference is that the method for seg is a *dependent equality*, as m_{zero} and m_{one} have definitionally different types.

Since we are working with sets, we do not need to add a computation rule for the path constructor seg, as that would introduce a path between equalities, i.e. it would not introduce anything new.

Family and dependent morphism structure on $\mathcal{F}am$

Before we move on to sort categories and categories of algebras, let us first consider the families and dependent morphisms in the category $\mathcal{F}am$. Recall that $\mathcal{F}am$ is the sort category for the inductive-inductive type $(\mathsf{Con}, \mathsf{Ty})$ of contexts and types in a context. We will use the intuition of the induction principle of that inductive-inductive definition to make sense of what is going on here.

Our first goal is to derive a family $\mathsf{Fam}_{\mathcal{F}am} : |\mathcal{F}am| \to \mathsf{Set}$ that satisfies, for any $X : |\mathcal{F}am|$:

$$\mathsf{Fam}_{\mathcal{F}am}\ X = (Y : |\mathcal{F}am|) \times \mathcal{F}am(Y, X)$$

We derive the definition by taking the right hand side of the above equation and performing the following equational reasoning:

$$\begin{aligned}
&(Y : |\mathcal{F}am|) \times \mathcal{F}am(Y, X) \\
&= \{ \text{ definition of objects and morphisms in } \mathcal{F}am \ \} \\
&\quad (Y : \mathsf{Set}) \times (Q : Y \to \mathsf{Set}) \times (p : Y \to X) \\
&\quad \times (q : (y : Y) \to Q\ y \to P\ (p\ y)) \\
&= \{ \text{ combining } Y \text{ and } P \text{ using the family structure on } \mathcal{S}et \ \} \\
&\quad (V : X \to \mathsf{Set}) \times (Q : (x : X) \times V\ x \to \mathsf{Set}) \\
&\quad \times (q : (y : (x : X) \times V\ x) \to Q\ y \to P\ (\pi_1\ y)) \\
&= \{ \text{ currying } \} \\
&\quad (V : X \to \mathsf{Set}) \times (Q : (x : X) \to V\ x \to \mathsf{Set}) \\
&\quad \times (q : (x : X) \to (y : V\ x) \to Q\ x\ y \to P\ x) \\
&= \{ \text{ combining } Q \text{ and } q \text{ using family structure on } \mathcal{S}et \ \} \\
&\quad (V : X \to \mathsf{Set}) \times (W : (x : X) \to V\ x \to P\ x \to \mathsf{Set})
\end{aligned}$$

What we ended up with does match with what a motive for $(\mathsf{Con}, \mathsf{Ty})$ is,

namely:

$$(P : \mathsf{Con} \to \mathsf{Set}) \times (Q : (\Gamma : \mathsf{Con}) \to P\ \Gamma \to \mathsf{Ty}\ \Gamma \to \mathsf{Set})$$

A motive consists of a motive on Con, which is just a family on it, along with a family on Ty, for every Γ : Con. Since we may also refer to results we get from doing induction on Γ, we also have a $P\ \Gamma$ in there.

We then give the following definitions:

$$\begin{aligned}
\mathsf{Fam}_{\mathcal{F}am}\ (X, P) &:\equiv (V : X \to \mathsf{Set}) \times (W : (x : X) \to P\ x \to V\ x \to \mathsf{Set}) \\
\mathsf{total}_{\mathcal{F}am}\ \{X, P\}\ (V, W) &:\equiv ((x : X) \times V\ x, (\lambda(x, z).(y : P\ x) \times W\ x\ y\ z)) \\
\mathsf{proj}_{\mathcal{F}am}\ \{X, P\}\ (V, W) &:\equiv (\lambda(x, y).x, \lambda(x, y)\ (z, w) \to z) \\
\mathsf{preimage}_{\mathcal{F}am}\ \{X, P\}\ ((C, D), (p, q)) &:\equiv (\lambda a.(c : C) \times p\ c = a, \lambda xy(z, z').(w : D\ z) \times (q\ z\ w =^{P}_{z'}
\end{aligned}$$

Now we have a family structure on $\mathcal{F}am$, we need to define what dependent morphisms of these families are. Returning to the (Con, Ty) example, given a motive P : Con $\to$ Set, Q : $(\Gamma$: Con$) \to P$ Con $\to$ Ty $\Gamma \to$ Set and the appropriate methods, the induction principle gives us two functions:

$$\begin{aligned}
&\mathsf{Con\text{-}ind} : (\Gamma : \mathsf{Con}) \to P\ \Gamma \\
&\mathsf{Ty\text{-}ind} : (\Gamma : \mathsf{Con})\ (\tau : \mathsf{Ty}\ \Gamma) \to Q\ \Gamma\ (\mathsf{Con\text{-}ind}\ \Gamma)\ \tau
\end{aligned}$$

The function Ty-ind refers to Con-ind in its type as well: the results we get from Con-ind may be used in the motive of Ty.

We can derive this formulation by the following equational reasoning:

$$
\begin{aligned}
& (s : \mathsf{Fam}_{\mathcal{F}am}((X,P),(\mathsf{total}_{\mathcal{F}am}\ (X,P)\ (V,W)))) \\
& \times (s_0 : \mathsf{proj}_{\mathcal{F}am}\ (X,P)\ (V,W) \circ_{\mathcal{F}am} s = \mathsf{id}_{\mathcal{F}am}\ (X,P)) \\
= \quad & \{ \text{ definition of morphisms in } \mathcal{F}am, \text{ equality of pairs is pair of equalities } \} \\
& (s : X \to (x : X) \times V\ x) \\
& \times (t : (x : X) \to P\ x \to (y : P(\pi_0\ (s\ x))) \times (W\ (\pi_0\ (s\ x))\ y\ (\pi_1\ (s\ x)))) \\
& \times (s_0 : \pi_0 \circ s = \mathsf{id}_{\mathcal{S}et}\ X) \times (t_0 : \lambda xz.\pi_0\ (t\ x\ z) = \lambda xz.z) \\
= \quad & \{ \text{ combining } s \text{ and } s_0 \text{ using the dependent morphism structure on } \mathcal{S}et \ \} \\
& (s : (x : X) \to V\ x) \times (t : (x : X) \to P\ x \to (y : P\ x) \times W\ x\ y\ (s\ x)) \\
& \times (t_0 : \lambda xz.\pi_0\ (t\ x\ z) = \lambda xz.z) \\
= \quad & \{ \text{ combining } t \text{ and } t_0 \text{ using the dependent morphism structure on } \mathcal{S}et \ \} \\
& (s : (x : X) \to V\ x) \times (t : (x : X)\ (y : Px) \to W\ x\ y\ (s\ x))
\end{aligned}
$$

The dependent morphism structure on $\mathcal{F}am$ is then:

$$
\begin{aligned}
\mathsf{DepHom}_{\mathcal{F}am}\ \{X,P\}\ (V,W) :\equiv\ & (f : (x : X) \to V\ x) \\
& \times (g : (x : X) \to (y : P\ x) \to W\ x\ y\ (f\ x))
\end{aligned}
$$

Sort categories

The family structure on a sort category can be given by induction on the specification. In the case of an empty specification, we define $\mathsf{Fam}_{\mathbb{1}}\ X :\equiv 1$ for any $X : |\mathbb{1}|$. The other definitions are equally trivial.

In the induction step case, we get a family structure on S_{i-1} and have to provide one on the category S_i which is built out of S_{i-1} with the functor $R_i : S_{i-1} \Rightarrow \mathcal{S}et$. Our goal is to find, given $X : |S_{i-1}|$ and $P : R_i X \to \mathsf{Set}$, a set $\mathsf{Fam}_{S_i}(X,P)$ such that:

$$\mathsf{Fam}_{S_i}(X,P) = ((Y,Q) : |S_i|) \times (p : S_i((Y,Q),(X,P)))$$

We can perform the following equational reasoning on the right hand side

of the equation above:

$$
\begin{aligned}
& (Y : |S_i|) \times (p : S_i(Y, (X, P))) \\
= & \{ \text{ definition of } S_i \ \} \\
& (Y : |S_{i-1}|) \times (Q : R_i Y \to \mathsf{Set}) \times (p : S_{i-1}(Y, X)) \times (q : (x : R_i Y) \to Q\ x \to P\ (R_i\ p\ x)) \\
= & \{ \text{ combining } Y \text{ and } p \text{ using family structure on } S_{i-1} \ \} \\
& (V : \mathsf{Fam}_{S_{i-1}}\ X) \times (Q : R_i(\mathsf{total}_{S_{i-1}}\ X\ V) \to \mathsf{Set}) \\
& \times (q : (x : R_i(\mathsf{total}_{S_{i-1}}\ X\ V)) \to Q\ x \to P\ (R_i(\mathsf{proj}_{S_{i-1}}\ X\ V)\ x)) \\
= & \{ \text{ rewrite } R_i(\mathsf{total}_{S_{i-1}}\ X\ V) \text{ with } \Box_{R_i} \ \} \\
& (V : \mathsf{Fam}_{S_{i-1}}\ X) \times (Q : (x : R_i X) \times \Box_{R_i}\ X\ V\ x \to \mathsf{Set}) \\
& \times (q : (x : R_i X) \times (y : \Box_{R_i}\ X\ V\ x) \to Q\ (x, y) \to P\ x)) \\
= & \{ \text{ combining } Q \text{ and } q \text{ using family structure on } \mathcal{S}et \ \} \\
& (V : \mathsf{Fam}_{S_{i-1}} X) \times (W : (x : R_i X) \times \Box_{R_i}\ X\ V\ x \to P\ x \to \mathsf{Set})
\end{aligned}
$$

The family structure on S_i is then:

$$\mathsf{Fam}_{S_i}\ (X, P) :\equiv (V : \mathsf{Fam}_{S_{i-1}} X) \times (W : (x : R_i X) \times \Box_{R_i}\ X\ V\ x \to P\ x \to \mathsf{Set})$$

with

$$
\begin{aligned}
\mathsf{total}_{S_i}\ \{X, P\}\ (V, W) &:\equiv (\mathsf{total}_{S_{i-1}}\ X\ V, \lambda x.(y : P\ (R_i\ (\mathsf{proj}_{S_{i-1}}\ X\ V)\ x)) \times (W(\phi\ x)\ y)) \\
\mathsf{proj}_{S_i}\ \{X, P\}\ (V, W) &:\equiv (\mathsf{proj}_{S_{i-1}}\ X\ V, \lambda x(y, z).y)
\end{aligned}
$$

where $\phi : (x : R_i X) \times \Box_{R_i}\ X\ V x \to R_i\ (\mathsf{total}_{S_{i-1}}\ X\ V)$ is the isomorphism we get from the definition of $\Box$.

The dependent morphism structure on S_i can be derived in a similar way that of $\mathsf{DepHom}_{\mathcal{F}am}$, arriving at:

$$
\begin{aligned}
\mathsf{DepHom}_{S_i}\ \{X, P\}\ (V, W) :\equiv & (f : \mathsf{DepHom}_{S_{i-1}}\ X\ V) \\
& \times (g : (x : R_i\ X)\ (y : P\ x) \to W\ (x, \bar{R}_i\ f\ x)\ y)
\end{aligned}
$$

Point constructors

Just as the sort and algebra categories are defined inductively as having objects from the previous sort/algebra categories along with some extra structures, the family structures on these categories reflect this pattern. We have seen this happen in the previous section when describing the family structure on sort categories.

Before we move on to the full generality of point constructors of arbitrary sort, we will consider $\mathcal{S}et$-sorted point constructors. Let Alg_s be a category of algebras of some specification $s : \mathsf{Spec}$. A $\mathcal{S}et$-sorted constructor on Alg_s is given by a functor $F : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$ and the new category of algebras is (F, U_s)-dialg with $\mathsf{U}_s : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$ the forgetful functor. Following a similar derivation to that of the family structure on F-alg for endofunctors F on $\mathcal{S}et$, we arrive at:

$$\mathsf{Fam}_{(F,\mathsf{U}_s)\text{-dialg}}\,(X,\theta) :\equiv (P : \mathsf{Fam}_{\mathsf{Alg}_s}\,X)\times(m : (x : FX)\times\Box_F\,P\,x \to \Box_{\mathsf{U}_s}\,P\,(\theta\,x))$$

We can recover $\mathsf{Fam}_{F\text{-alg}}$ from this if $\mathsf{Alg}_s = \mathcal{S}et$ and U_s is the identity functor, as $\Box_{\mathsf{id}_{\mathcal{S}et}}\,P = P$.

Moving on to the fully general point constructor case, we have the following data:

- $\mathcal{S} : \mathsf{Sorts}$ with $s : \mathsf{Spec}_{\mathcal{S}}$
- $S_i : \mathsf{Cat}$ with $p : S_i \in \mathcal{S}$
- $F : \mathsf{Alg}_s \Rightarrow S_i$ such that $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$

We start out by investigating what $\Box_F$ and $\Box_{\widehat{\mathsf{U}}_s}$ look like. Recall that we can split the functor $F : \mathsf{Alg}_s \Rightarrow S_i$ into two operations, which are both functorial:

- $F^0 : |\mathcal{C}| \to |S_{i-1}|$, which we know to be equal to $\widehat{\mathsf{U}}_s^0$
- $F^1 : (X : |\mathcal{C}|) \to (x : R_i(F^0 X)) \to F^1\ X\ x \to \mathsf{Set}$

Knowing what the family structure on S_i is, we know that for $P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X$ $\Box_F\ P$ has the following type:

$$\Box_F\ P : (V : \mathsf{Fam}_{S_{i-1}}\ F^0 X) \times (W : (x : R_i(F^0 X)) \to \Box_{R_i}\ V\ x \to F^1\ X\ x \to \mathsf{Set})$$

Splitting this into its constituents will turn out to be useful, just as we have done with the functors itself. Let us call the first and second projections of this $\Box_{F^0}$ and $\Box_{F^1}$ respectively:

$$\begin{aligned}
&\Box_{F^0}\ P : \mathsf{Fam}_{S_{i-1}}\ (F^0\ X) \\
&\Box_{F^1}\ P : (x : R_i(F^0 X)) \times \Box_{R_i}\ (\Box_{F^0}\ P)\ x \to F^1\ X\ x \to \mathsf{Set}
\end{aligned}$$

Since the $\Box$ operation is functorial, we will use $\Box_{R_i \circ F^0}$ instead of the nested variant. Also since $F^0 = \widehat{\mathsf{U}}^0_s$, we will use $\Box_{R_i \circ \widehat{\mathsf{U}}^0_s}$.

Define $s' :\equiv \mathsf{snoc}\ s\ (S_i, p, F) : \mathsf{Spec}$ to be the new specification, which is s extended with the point constructor given by the data above. We can now derive the family structure on this category $\mathsf{Alg}_{s'}$ (see fig. 5.1) and arrive at the following:

$$\begin{aligned}
\mathsf{Fam}_{\mathsf{Alg}_{s'}}\ (X, \theta) :\equiv (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X) \\
\times (m : (x : R_i(\widehat{\mathsf{U}}^0_s X)) \times (x' : \Box_{R_i \circ \widehat{\mathsf{U}}^0_s}\ P\ x) \\
\to (y : F^1\ X\ x) \times (y' : \Box_{F^1}\ P\ (x, x')\ y) \\
\to \Box_{\widehat{\mathsf{U}}^1_s}\ P\ (x, x')\ (\theta\ x\ y))
\end{aligned}$$

What we get out as the definition of algebra families for point constructors looks a bit daunting at first glance. However, we can also see how it is a generalisation of the original situation in F-alg for endofunctors on $\mathcal{S}et$. The first part, $(x : R_i(\widehat{\mathsf{U}}^0_s X)) \times (x' : \Box_{R_i \circ \widehat{\mathsf{U}}^0_s}\ P\ x)$, gives us the induction hypotheses for all the sorts "below" the sort of the current constructors. The second part, $(y : F^1\ X\ x) \times (y' : \Box_{F^1}\ P\ (x, x')\ y)$, gives us the induction hypothesis for the recursive arguments, which looks very similar to the F-alg case. Finally, the result type is $\Box_{\widehat{\mathsf{U}}^1_s}\ P\ (x, x')\ (\theta\ x\ y)$, which is the dialgebra

$$
\begin{aligned}
& (Y : |\mathsf{Alg}_s|) \\
& \times (\rho : S_i(FY, \widehat{\mathsf{U}}_s Y)) \times (t_i\rho = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s X)}) \\
& \times (\mathsf{Alg}_{s'}((Y,\rho),(X,\theta)) \\
= \quad & \{ \text{ unfold definition of } \mathsf{Alg}_{s'} \} \\
& (Y : |\mathsf{Alg}_s|) \times (\rho : S_i(FY, \widehat{\mathsf{U}}_s Y)) \times (t_i\rho = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s Y)}) \\
& \times (p : \mathsf{Alg}_s(Y,X)) \times (p_0 : \widehat{\mathsf{U}}_s p \circ \rho = \theta \circ Fp) \\
= \quad & \{ \text{ combine } Y \text{ and } p \text{ using Fam structure on } \mathsf{Alg}_s \} \\
& (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X) \times (\rho : S_i(F(\mathsf{total}\ P), \widehat{\mathsf{U}}_s(\mathsf{total}\ P))) \times (t_i\rho = \mathsf{id}_{t_i(\widehat{\mathsf{U}}_s(\mathsf{total}\ P))}) \\
& \times (p_0 : \widehat{\mathsf{U}}_s(\mathsf{proj}\ P) \circ \rho = \theta \circ F(\mathsf{proj}\ P)) \\
= \quad & \{ \text{ unfold definition } S_i \text{ and expand } p_0 \} \\
& (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X) \\
& \times (\rho : (x : R_i(\widehat{\mathsf{U}}^0_s(\mathsf{total}\ P))) \to F^1(\mathsf{total}\ P)\ x \to \widehat{\mathsf{U}}^1_s(\mathsf{total}\ P)\ x) \\
& \times (p_0 : (x : R_i(\widehat{\mathsf{U}}^0_s(\mathsf{total}\ P)))\ (y : F^1(\mathsf{total}\ P)\ x) \\
& \quad \to \widehat{\mathsf{U}}^1_s(\mathsf{proj}\ P)\ x(\rho\ x\ y) = \theta\ (R_i\ (\widehat{\mathsf{U}}^0_s(\mathsf{proj}\ P)x)\ (F^1\ (\mathsf{proj}\ P)\ x\ y))) \\
= \quad & \{ \text{ rewriting using } \Box \text{ notation } \} \\
& (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X) \\
& \times (\rho : (x : R_i(\widehat{\mathsf{U}}^0_s X)) \times (x' : \Box_{R_i \circ \widehat{\mathsf{U}}^0_s}\ P\ x) \\
& \quad \to (y : F^1\ X\ x) \times (y' : \Box_{F^1}\ P\ (x,x')\ y) \\
& \quad \to (z : \widehat{\mathsf{U}}^1_s\ X\ x) \times \Box_{\widehat{\mathsf{U}}^1_s}\ P\ (x,x')\ z) \\
& \times (p_0 : (x : R_i(\widehat{\mathsf{U}}^0_s X)) \times (x' : \Box_{R_i \circ \widehat{\mathsf{U}}^0_s}\ P\ x) \\
& \quad \to (y : F^1\ X\ x) \times (y' : \Box_{F^1}\ P\ (x,x')\ y) \\
& \quad \to \pi_0\ (\rho\ (x,x')\ (y,y')) = \theta\ x\ y) \\
= \quad & \{ \text{ singleton contraction } \} \\
& (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X) \\
& \times (m : (x : R_i(\widehat{\mathsf{U}}^0_s X)) \times (x' : \Box_{R_i \circ \widehat{\mathsf{U}}^0_s}\ P\ x) \\
& \quad \to (y : F^1\ X\ x) \times (y' : \Box_{F^1}\ P\ (x,x')\ y) \\
& \quad \to \Box_{\widehat{\mathsf{U}}^1_s}\ P\ (x,x')\ (\theta\ x\ y))
\end{aligned}
$$

Figure 5.1: Derivation of the family structure for point constructors

counterpart of what we had before.

For dependent morphisms, the $\mathcal{S}et$-sorted case again follows the F-alg for endofunctors F on $\mathcal{S}et$ in a straightforward way. Suppose we have $F : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$, along with $(X, \theta) : |(F, \mathsf{U}_s)\text{-dialg}|$ and $(P, m) : \mathsf{Fam}_{(F,\mathsf{U}_s)\text{-dialg}}\ (X, \theta)$, we have:

$$\mathsf{DepHom}_{(F,\mathsf{U}_s)\text{-dialg}}\ (P, m) :\equiv (\mathcal{s} : \mathsf{DepHom}_{\mathsf{Alg}_s}\ P) \times (\mathcal{s}_0 : (x : FX) \to \overline{\mathsf{U}_s}\ \mathcal{s}\ (\theta\ x) = m\ x\ (\overline{F}\ \mathcal{s}\ x))$$

Generalising to the arbitrarily sorted point constructors follows the same steps as we took to derive the notion of algebra family for point constructors. We have to find out, given $P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X$ and $X : |\mathsf{Alg}_s|$, what $\overline{F} : \mathsf{DepHom}_{\mathsf{Alg}_s}\ P \to \mathsf{DepHom}_{S_i}\ (\square_F\ P)$ gives us for a functor $F : \mathsf{Alg}_s \Rightarrow S_i$. We can split $\overline{F}$ into two parts:

$$\overline{F}^0 : \mathsf{DepHom}_{S_{i-1}}\ (\square_{F^0}\ P)$$
$$\overline{F}^1 : (\mathcal{s} : \mathsf{DepHom}_{\mathsf{Alg}_s}\ P) \to (x : R_i(F^0 X))\ (y : F^1\ X\ x) \to \square_{F^1}\ (x, \overline{R_i}\ \mathcal{s}\ x)\ y$$

The dependent morphism structure on $\mathsf{Alg}_{s'}$ is then, given $(X, \theta) : |\mathsf{Alg}_{s'}|$ and $(P, m) : \mathsf{Fam}_{\mathsf{Alg}_{s'}}\ (X, \theta)$:

$$\begin{aligned}
&\mathsf{DepHom}_{\mathsf{Alg}_{s'}}\ (P, m) :\equiv \\
&\quad (\mathcal{s} : \mathsf{DepHom}_{\mathsf{Alg}_s}\ P) \\
&\quad \times (\mathcal{s}_0 : (x : R_i(\widehat{\mathsf{U}}^0_s X))\ (y : F^1\ X\ x) \\
&\qquad \to \overline{\mathsf{U}}^1_s\ \mathcal{s}\ x\ (\theta\ x\ y) = m\ (x, \overline{R_i \circ \widehat{\mathsf{U}}^0_s}\ \mathcal{s}\ x)\ (y, \overline{F}^1\ \mathcal{s}\ x\ y))
\end{aligned}$$

Path constructors

Let us first consider $\mathcal{S}et$-sorted path constructors. Suppose we have Alg_s for some specification $s : \mathsf{Spec}$ and $F : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$ describing the arguments and $\ell, r : F \dashrightarrow \mathsf{U}_s$ the end points of the path constructor. Let $s' : \mathsf{Spec}$ be s extend with the path constructor described by these data. Since $\mathsf{Alg}_{s'}$ is a full subcategory of Alg_s, we have that $\mathsf{Fam}_{\mathsf{Alg}_{s'}}\ (X, \theta)$ for $X : \mathsf{Alg}_s$ and

$\theta : \ell_X = r_X$ is simply:

$$(P : \mathsf{Fam}_{\mathsf{Alg}_s}) \times (\rho : \ell_{\mathsf{total}\ P} = r_{\mathsf{total}\ P})$$

We can simplify this a bit further. We can define $\ell_P : (x : FX) \times \Box_F\ P\ x \to (y : \mathsf{U}_s X) \times \Box_{\mathsf{U}_s}\ P\ y$ as $\ell_{\mathsf{total}\ P}$ with the defining equivalences of $\Box_F$ and $\Box_u$ plugged in at the right places. We furthermore will use the notation ℓ_P^0 and ℓ_P^1 to indicate first and second projections of these functions respectively. Note that by naturality $\ell_P^0 = \ell_X$ and $r_P^0 = r_X$. An equality $\ell_P = r_P$ is then an equality of functions mapping into a Σ-type, we can then perform the following equational reasoning to simplify the expression:

$$
\begin{aligned}
& (\ell_{\mathsf{total}\ P} = r_{\mathsf{total}\ P}) \\
= \quad & \{ \text{ definition of } \ell_P \text{ and } r_P \ \} \\
& (\ell_P = r_P) \\
= \quad & \{ \text{ equality of pairs is a pair of equalities } \} \\
& (\rho_0 : \pi_0 \circ \ell_P = r_P) \times (\rho_1 : \pi_1 \circ \ell_P =^{\lambda f.(x:FX)\times\Box_F Px\to\Box_{\mathsf{U}_s} P(fx)}_{\rho_0} \pi_1 \circ r_P) \\
= \quad & \{ \text{ by definition and naturality of } \ell_P \text{ and } r_P \ \} \\
& (\rho_0 : \ell_X = r_X) \times (\rho_1 : \pi_1 \circ \ell_P =^{\lambda f.(x:FX)\times\Box_F Px\to\Box_{\mathsf{U}_s} P(fx)}_{\rho_0} \pi_1 \circ r_P) \\
= \quad & \{ \text{ by uniqueness of identity proofs, we can multiply by } (\rho_0 = \theta) \ \} \\
& (\rho_0 : \ell_X = r_X) \times (\rho_0 = \theta) \times (\rho_1 : \pi_1 \circ \ell_P =^{\lambda f.(x:FX)\times\Box_F Px\to\Box_{\mathsf{U}_s} P(fx)}_{\rho_0} \pi_1 \circ r_P) \\
= \quad & \{ \text{ singleton contraction } \} \\
& (\pi_1 \circ \ell_P =^{\lambda f.(x:FX)\times\Box_F Px\to\Box_{\mathsf{U}_s} P(fx)}_{\rho_0} \pi_1 \circ r_P) \\
= \quad & \{ \text{ function extensionality } \} \\
& (x : FX) \times (y : \Box_F\ P\ x) \to l_P^1\ (x, y) =^{\Box_{\mathsf{U}_s} P}_{\theta x} r_P^1\ (x, y)
\end{aligned}
$$

In the general case, we have the following data describing a path constructor:

- $\mathcal{S}$: Sorts with s : $\mathsf{Spec}_{\mathcal{S}}$
- S_i : Cat with $p : S_i \in \mathcal{S}$

- $F : \mathsf{Alg}_s \Rightarrow S_i$ such that $t_i \circ F = t_i \circ \widehat{\mathsf{U}}_s$
- $l, r : F \dot{\to} \widehat{\mathsf{U}}_s$ such that $t_i l = t_i r = \mathsf{id}$

Define $s' :\equiv \mathsf{snoc}\ s\ (S_i, p, F) : \mathsf{Spec}$ to be the new specification, which is s extended with the path constructor given by the data above. Let $(X, \theta) : |\mathsf{Alg}_{s'}|$, i.e.:

$$\theta : (x : R_i(\widehat{\mathsf{U}}_s^0 X)) \to (y : F^1\ X\ x) \to \ell_X^1\ x\ y = r_X^1\ x\ y$$

Deriving the family structure $\mathsf{Alg}_{s'}$ follows both our $\mathcal{S}et$-sorted situation described above as well as the derivation for arbitrarily sorted point constructors. The family structure on $\mathsf{Alg}_{s'}$ is then:

$$\begin{aligned}\mathsf{Fam}_{\mathsf{Alg}_{s'}}\ (X, \theta) :\equiv (P : \mathsf{Fam}_{\mathsf{Alg}_s}\ X)\\ (m : (x : R_i(\widehat{\mathsf{U}}_s^0 X)) \times (x' : \square_{R_i \circ \widehat{\mathsf{U}}_s^0}\ P\ x)\\ \to (y : F^1\ X\ x) \times (y' : \square_{F^1}\ P\ (x, x')\ y)\\ \to \ell_P^1\ (x, x')\ (y, y') =_{\theta\ x\ y}^{\square_{\widehat{\mathsf{U}}_s^1 P(x,x')}} r_P^1\ (x, x')\ (y, y'))\end{aligned}$$

Since $\mathsf{Alg}_{s'}$ is a full subcategory of Alg_s, the dependent morphisms are simply inherited from Alg_s:

$$\mathsf{DepHom}_{\mathsf{Alg}_{s'}}\ (P, m) :\equiv \mathsf{DepHom}_{\mathsf{Alg}_s}\ P$$

5.4.4 Putting it all together

In the last subsection, we have seen what the induction principles of quotient inductive-inductive definitions look like. Given a specification $s : \mathsf{Spec}$ with sorts $\mathcal{S} : \mathsf{Sorts}$, we can construct the family and dependent morphism structure on Alg_s by induction on s. We have seen that for $\mathcal{S}et$-sorted definitions it is a rather straightforward generalisation of the situation for F-algebras with F an endofunctor on $\mathcal{S}et$. The fully general case where we may have arbitrary sorts is a bit more involved, but if we squint our eyes we can still see that the induction principle is of the same form as the principle

for $\mathcal{S}et$-sorted definitions.

5.5 Related work

The treatment of induction for ordinary inductive types presented here is loosely based on the approach given in [GJF10].

In [Nor13], the induction principle for inductive-inductive definitions is constructed by first considering the induction principle for ordinary inductive types and using the categories with families [Dyb95] framework to generalise this construction to inductive-inductive definitions. In this chapter, we do not make use of the categories with families framework, but the constructions are quite similar nonetheless.

Chapter 6

Constructing quotient inductive-inductive definitions

We have given a formal definition of what constitutes a quotient inductive-inductive definition in terms of algebraic semantics. We have shown that the resulting categories of algebras are sensible in the sense that an algebra is initial if and only if it satisfies the induction principle. We have therefore shown that the initial algebras are well-behaved in this regard. To justify adding quotient inductive-inductive types to the theory, we still need to establish their existence in the usual models. After all, these results are all uninteresting if these models do not have quotient inductive-inductive types, which would make our results vacuously true.

To start out we will address the issue of strict positivity in inductive definitions, showing that our previous specification of quotient inductive-inductive definitions includes definitions of which the category of algebras does not have an initial object.

For definitions with no constructors, the category of algebras is the category of sorts. We show that the categories of sorts always have an initial object (theorem 6.2.3).

We discuss in section 6.3 the usual construction of initial algebras for an endofunctor as a sequential colimit. We explore how this approach can be both internalised as well as adapted to construct initial algebras for certain

classes of quotient inductive-inductive definitions. The key is here to not only construct an initial object but establish that there is a left adjoint to the forgetful functor $\mathsf{Alg}_{n+1} \Rightarrow \mathsf{Alg}_n$, which allows us to do the construction by induction on the length of the specification. We end up with a chain of adjunctions:

$$\mathcal{S}et \underset{U_0}{\overset{L_0}{\rightleftarrows}} \mathsf{Alg}_0 \underset{U_1}{\overset{L_1}{\rightleftarrows}} \mathsf{Alg}_1 \underset{U_2}{\overset{L_2}{\rightleftarrows}} \ldots \underset{U_n}{\overset{L_n}{\rightleftarrows}} \mathsf{Alg}_n$$

Since left adjoints preserve colimits, in particular initial objects, and $\mathcal{S}et$ has an initial object 0, $(L_n \circ \ldots \circ L_0)\ 0$ gives us the initial algebra in Alg_n we are after.

6.1 Strict positivity

For ordinary inductive types we have to have several syntactic restrictions on the point constructors in order for the inductive types to actually exist. We have seen in section 4.6 that the recursive positions may only occur in *positive positions* in order for the arguments to describe a covariant functor. This is not enough to guarantee the existence of an initial algebra. If we look at $\mathcal{S}et$ and consider the double powerset functor $P : \mathcal{S}et \Rightarrow \mathcal{S}et$:

$$P\ X :\equiv (X \to \mathsf{Bool}) \to \mathsf{Bool}$$

This defines a covariant functor on $\mathcal{S}et$, yet P-alg does not have an initial algebra. By Lambek's lemma, having an initial algebra $(X, \theta) : |P\text{-}\mathsf{alg}|$ would imply that $X \simeq P\ X$. By Cantor's theorem, we know that there is no set X which is isomorphic to its powerset, hence we arrive at a contradiction.

6.2 Initial objects in sort categories

If we have an inductive specification with no constructors, the category of algebras is the category of sorts. Constructing the initial object in these

categories is similar to the construction in $\mathcal{F}am$. We will construct the left adjoint to the forgetful functors of sort categories, similar to that of $\mathcal{F}am$. The forgetful functor $\mathcal{F}am \Rightarrow \mathcal{S}et$ has a left adjoint: the *truth functor* that maps a set X to the family $\lambda x.1 : X \to \mathsf{Set}$. This generalises to any functor in the abovementioned chain of sort categories: suppose we have a sort specification $\mathcal{S} : \mathsf{Sorts}$ giving rise to the following chain of sort categories:

$$\mathbb{1} \xleftarrow{t_0} S_0 \xleftarrow{t_1} S_1 \xleftarrow{t_2} \ldots \xleftarrow{t_n} S_n$$

Proposition 6.2.1. *Every forgetful functor $t_i : S_{i+1} \Rightarrow S_i$ has a left adjoint*

Proof. We define the functor $u_i : S_i \Rightarrow S_{i+1}$ as follows:

- on objects: $u_i X :\equiv (X, \lambda x.1)$
- on morphisms: $u_i f :\equiv (f, \lambda a\ x.x)$

We then have to check whether we have for any $X : |S_i|$ and $(Y, Q) : |S_{i+1}|$ that

$$S_{i+1}(u_i\ X, (Y, Q)) = S_i(X, Y)$$

which we can show by simple equational reasoning:

$$\begin{aligned} S_{i+1}(u_i\ X, (Y, Q)) &= S_{i+1}((X, \lambda x.1), (Y, Q)) \\ &= (f : S_i(X, Y) \times (g : (a : R_{i+1}X) \to 1 \to 1) \\ &= S_i(X, Y) \end{aligned}$$

□

Proposition 6.2.2. *There exists an adjunction between S_0 and $\mathcal{S}et$*

Proof. Note that S_0 is equivalent to a category with objects $A \to \mathsf{Set}$ for some $A : \mathsf{Set}$ and morphisms between $P, Q : A \to \mathsf{Set}$ being dependent functions $(x : A) \to P\ x \to Q\ x$. Usually $A = 1$, so we have a trivial adjunction between S_0 and $\mathcal{S}et$. Whatever the choice of A is, we can construct the

following pair of adjoint functors between $\mathcal{S}et$ and S_0:

$$\Pi : S_0 \Rightarrow \mathcal{S}et$$
$$\Pi\ P :\equiv (x : A) \to P\ x$$

$$K : \mathcal{S}et \Rightarrow S_0$$
$$K\ X :\equiv (\lambda x.X)$$

We then have for any X : Set and $P : A \to$ Set the following equality

$$\begin{aligned} S_0(LX, P) &= (x : A) \to X \to P\ x \\ &= X \to (x : A) \to P\ x \\ &= X \to \Pi\ P \end{aligned}$$

Therefore we have that $K \dashv \Pi$. □

We therefore get the following chain of adjunctions:

$$\mathcal{S}et \underset{\Pi}{\overset{K}{\underset{\top}{\rightleftarrows}}} S_0 \underset{t_1}{\overset{u_1}{\underset{\top}{\rightleftarrows}}} S_1 \underset{t_2}{\overset{u_2}{\underset{\top}{\rightleftarrows}}} \cdots \underset{t_n}{\overset{u_n}{\underset{\top}{\rightleftarrows}}} S_n$$

Theorem 6.2.3. *Given a sorts specification $\mathcal{S}$: Sorts, every sort category $S_i \in [\![\ \mathcal{S}\ ]\!]$ has an initial object.*

Proof. Since $\mathcal{S}et$ has an initial object, we get the initial object of any sort category S_i by following the chain of left adjoints constructed in proposition 6.2.1 and proposition 6.2.2. □

6.3 Initial objects via sequential colimits

Initial algebras of endofunctors can be constructed via sequential colimits, given some reasonable assumptions on the endofunctor. In this section we will review this result due to Adámek [AK79] and see how we can use it to

construct initial dialgebras. We will also discuss how these proofs can be performed inside type theory itself.

Definition 6.3.1 (ω-cochain)**.** An ω-cochain in a category $\mathcal{C}$ consists of:

- $X : \mathbb{N} \to |\mathcal{C}|$
- $x : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_{n+1})$

i.e., we have the following diagram:

$$X_0 \xrightarrow{x_0} X_1 \xrightarrow{x_1} X_2 \xrightarrow{x_2} X_3 \xrightarrow{x_3} X_4 \xrightarrow{x_4} \dots$$

Definition 6.3.2 (Sequential colimit)**.** Given an ω-cochain (X, x), the colimit of (X, x) consists of:

- An object $X_\omega : |\mathcal{C}|$
- $c : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_\omega)$ its constructors
- $g : (n : \mathbb{N}) \to c_n = c_{n+1} \circ x_n$

with satisfies the universal property that for any other cocone (Y, d, h) we get:

- a unique $f : \mathcal{C}(X_\omega, Y)$
- with computation rule $f \circ c_n = d_n$

Theorem 6.3.3 (Adámek)**.** *Let $\mathcal{C}$: Cat be a category with an endofunctor $F : \mathcal{C} \Rightarrow \mathcal{C}$. The category of algebras F-*alg *has an initial object if $\mathcal{C}$ has sequential colimits and an initial object and F preserves these colimits.*

Proof. Define X_ω as the colimit of the cochain:

$$\emptyset \xrightarrow{!} F\,\emptyset \xrightarrow{F!} F^2\,\emptyset \xrightarrow{F^2!} F^3\,\emptyset \xrightarrow{F^3!} F^4\,\emptyset \longrightarrow \dots$$

We get the following:

- $c_n : \mathcal{C}(F^n\emptyset, X_\omega)$

- $g_n : c_n = c_{n+1} \circ F^n!$

along with the universal property giving us a recursion principle. Similarly for FX_ω: since F preserves sequential colimits, we have that FX_ω is the colimit of the cochain formed by applying F to the one given above, hence we have:

- $Fc_n : \mathcal{C}(F^{n+1}\emptyset, FX_\omega)$
- $Fg_n : c'_n = Fc_n \circ F^{n+1}!$

We can define the algebra structure on X_ω by employing the recursion principle of FX_ω. We define θ to be the morphism with computation rule, for any $n : \mathbb{N}$:

$$\theta \circ Fc_n = c_{n+1}$$

Now that we have an algebra, we need to show that it is initial. Suppose (Y, ρ) is an F-algebra, then we first show that we get an algebra morphism $(X_\omega, \theta) \to (Y, \rho)$. Y comes equipped with a cocone by virtue of its algebra structure, defined as follows:

$$\begin{array}{ccccccccc}
\emptyset & \xrightarrow{!} & F\,\emptyset & \xrightarrow{F!} & F^2\,\emptyset & \xrightarrow{F^2!} & F^3\,\emptyset & \xrightarrow{F^3!} & \dots \\
\downarrow{\scriptstyle !} & & \downarrow{\scriptstyle F!} & & \downarrow{\scriptstyle F^2!} & & \downarrow{\scriptstyle F^3!} & & \\
Y & \xleftarrow[\rho]{} & F\,Y & \xleftarrow[F\rho]{} & F^2\,Y & \xleftarrow[F^2\rho]{} & F^3\,Y & \xleftarrow[F^3\rho]{} & \dots
\end{array}$$

We have the cocone (Y, y_n) where y_n is defined recursively:

$$\begin{aligned}
&y : (n : \mathbb{N}) \to \mathcal{C}(F^n\emptyset, Y) \\
&y\ 0 :\equiv ! \\
&y\ (n+1) :\equiv \rho \circ Fy_n
\end{aligned}$$

It is immediate from this definition that we have $y_n = y_{n+1} \circ F^n!$, hence (Y, y_n) is indeed a cocone.

We can now define $f : \mathcal{C}(X_\omega, Y)$ by recursion, satisfying the computation rule:

$$f \circ c_n = y_n$$

Now we need to establish whether $f \circ \theta = \rho \circ Ff$. Both the left hand and right hand side are cocone morphisms $(FX_\omega, Fc_n) \to (Y, y_n)$:

$$\begin{aligned} f \circ \theta \circ Fc_n &= f \circ c_{n+1} \\ &= y_{n+1} \end{aligned}$$

and

$$\begin{aligned} \rho \circ Ff \circ Fc_n &= \rho \circ F(f \circ c_n) \\ &= \rho \circ Fy_n \\ &= y_{n+1} \end{aligned}$$

By the universal property of FX_ω they are indeed equal. The same line of reasoning can be used to establish that if we have another algebra morphism $f' : (X_\omega, \theta) \to (Y, \rho)$, then $f = f'$, which shows initiality of (X_ω, θ). □

6.3.1 Internal sequential colimits

The definition of sequential colimits is presented in such a way that we can straightforwardly formalise it in type theory. The same goes for Adámek's theorem. We can also show internally that $\mathcal{S}et$ has sequential colimits, if we have an internal version of coequalisers at hand.

Proposition 6.3.4. *We can construct sequential colimits in $\mathcal{S}et$ from the natural numbers and coequalisers.*

Proof. Suppose we have $X : \mathbb{N} \to \mathsf{Set}$ with $x : (n : \mathbb{N}) \to X_n \to X_{n+1}$. Define the function f:

$$\begin{aligned} &f : (n : \mathbb{N}) \times X_n \to (n : \mathbb{N}) \times X_n \\ &f\ (n, a) :\equiv (n + 1, x_n\ a) \end{aligned}$$

We define X_ω to be the coequaliser:

$$(n : \mathbb{N}) \times X_n \underset{f}{\overset{\mathsf{id}}{\rightrightarrows}} (n : \mathbb{N}) \times X_n \xrightarrow{c} X_\omega$$

We therefore get for any $(n, a) : (n : \mathbb{N}) \times X_n$:

$$c\,(n, a) = c\,(n + 1, x_n\ a)$$

Hence we get a cocone for the ω-cochain. Its universal property follows directly from the universal property of the coequaliser. □

We can generalise the result to arbitrary categories easily, but this does not help us much as constructing coproducts in categories of algebras is rather involved, and already involves sequential colimits themselves.

Moving beyond ω

In section 3.1.4, we hinted at the possibility of constructing quotient inductive types where the recursive positions are in some sense *finitary*. In this situation it seems we can construct the ordinary inductive part first and then quotient it once and get what we want. When we have an inductive definition with *infinitary* recursive occurrences, such as the example of the infinitely branching trees given in section 3.1.4, then this construction does not work. In that situation it is essential that the quotienting happens at the same time as the construction of the ordinary inductive parts.

In this chapter we consider such a construction with sequential colimits. However, everything we present in this chapter is about sequential colimits of length ω and with all the functors involved preserving ω-colimits. This means that the functors are finitary, which would then imply that there exists a construction where we only need to quotient once. However, the constructions given here do not make essential use of the fact that the cochains are of length ω and therefore should carry over to cochains of arbitrary length. Making this precise is not a trivial matter however, as we would first have to come up with an appropriate formalisation of ordinals, which

we defer to future work.

6.3.2 Constructing $\mathcal{S}et$-sorted quotient inductive-inductive definitions

In this section we will look at the simplified setting of $\mathcal{S}et$-sorted quotient inductive-inductive definitions. Our ultimate goal is to show that for any $\mathcal{S}et$-sorted $s : \mathsf{Spec}$, where all the functors preserve sequential colimits, we have a left adjoint to the forgetful functor $U : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$. We can do this by induction on the specification s. To construct this left adjoint for a category Alg_s extended with a point or path constructor, we need to have sequential colimits, coequalisers and binary coproducts in Alg_s.

$\mathcal{S}et$ satisfies all these conditions, so the base case of the induction holds. In the next two sections we will do the inductive step for point and path constructors respectively.

Point constructors

If we have Alg_s with an adjunction $L \dashv U : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$ and a functor $F : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$, then the category of (F, U)-dialgebras is equivalent to the category of $(L \circ F)$-algebras. As such, we can focus our attention on the properties of categories of algebras of endofunctors. Also note that since L is a left adjoint, it preserves colimits, hence $L \circ F$ also preserves sequential colimits.

It suffices to show that if a category $\mathcal{C}$ has sequential colimits, binary coproducts and coequalisers and F is an endofunctor on $\mathcal{C}$ preserving colimits, then F-alg also has sequential colimits, binary coproducts and coequalisers and furthermore a left adjoint from $\mathcal{C}$ to the forgetful functor.

In the following we will assume that $\mathcal{C}$ is a category with sequential colimits, binary coproducts and coequalisers and that F is an endofunctor on $\mathcal{C}$ preserving sequential colimits.

Proposition 6.3.5. *The category F-alg has sequential colimits.*

Proof. Suppose we have the following cochain in F-alg:

$$(X_0, \theta_0) \xrightarrow{x_0} (X_1, \theta_1) \xrightarrow{x_1} (X_2, \theta_1) \xrightarrow{x_2} (X_3, \theta_1) \xrightarrow{x_3} (X_4, \theta_1) \longrightarrow \dots$$

This means we have the following diagram in $\mathcal{C}$:

$$\begin{array}{ccccccccccc}
FX_0 & \xrightarrow{Fx_0} & FX_1 & \xrightarrow{Fx_1} & FX_2 & \xrightarrow{Fx_2} & FX_3 & \xrightarrow{Fx_3} & FX_4 & \longrightarrow & \dots \\
\downarrow{\scriptstyle \theta_0} & & \downarrow{\scriptstyle \theta_1} & & \downarrow{\scriptstyle \theta_2} & & \downarrow{\scriptstyle \theta_3} & & \downarrow{\scriptstyle \theta_4} & & \\
X_0 & \xrightarrow[x_0]{} & X_1 & \xrightarrow[x_1]{} & X_2 & \xrightarrow[x_2]{} & X_3 & \xrightarrow[x_3]{} & X_4 & \longrightarrow & \dots
\end{array}$$

To be precise, we have:

- $X : \mathbb{N} \to |\mathcal{C}|$ with algebras $\theta : (n : \mathbb{N}) \to \mathcal{C}(FX_n, X_n)$
- $x : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_{n+1})$ satisfying $x_n \circ \theta_n = \theta_{n+1} \circ Fx_n$

We can take the colimit of the cochain (X, x) in $\mathcal{C}$ and we get $X_\omega : |\mathcal{C}|$ with $c : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_\omega)$ satisfying $c_n = c_{n+1} \circ x_n$, satisfying the universal property of sequential colimits. Similarly, as F preserves sequential colimits, we get that FX_ω with $Fc : (n : \mathbb{N}) \to \mathcal{C}(FX_n, FX_\omega)$ is a colimit of the cochain (FX, Fx). This allows us to construct an algebra structure on X_ω, which is the map θ_ω defined by the computation property:

$$\theta_\omega \circ Fc_n = c_n \circ \theta_n$$

For this to work, we have to make sure that $c_n \circ \theta_n$ does in fact form a cocone, i.e. we need to check whether $c_n \circ \theta_n = c_{n+1} \circ \theta_{n+1} \circ Fx_n$. This follows from the fact that X_ω is a colimit of (X, x), giving us $c_n = c_{n+1} \circ x_n$. Every x_n is an algebra morphism: $x_n \circ \theta_n = \theta_{n+1} \circ Fx_n$, so we are done.

By definition of θ_ω, every c_n is an algebra morphism.

Finally we have to check whether $(X_\omega, \theta_\omega)$ satisfies the universal property. This follows from the universal property of X_ω. □

Proposition 6.3.6. *The category F-*alg *has binary coproducts.*

Proof. Let $(X, \theta), (Y, \rho) : |F\text{-alg}|$. Observe that by assumption we have the coproduct $X + Y$ in $\mathcal{C}$, but there is no evidence that this will carry an algebra

structure, let alone one such that the inclusions $X \to X+Y$ and $Y \to X+Y$ will preserve this structure.

Instead, we will create a sequence of objects A_n in $\mathcal{C}$, which have an approximate algebra structure (a map $\alpha_n : FA_n \to A_{n+1}$). For every A_n, we also have $\mathsf{inl}_n : X \to A_n$ and $\mathsf{inr}_n : Y \to A_n$, such that $\mathsf{inl}_{n+1} \circ \theta = \alpha_n \circ F\mathsf{inl}_n$, and similarly for inr_n and ρ. In other words: the inclusions preserve the approximate algebra structure.

We define the objects A_n inductively:

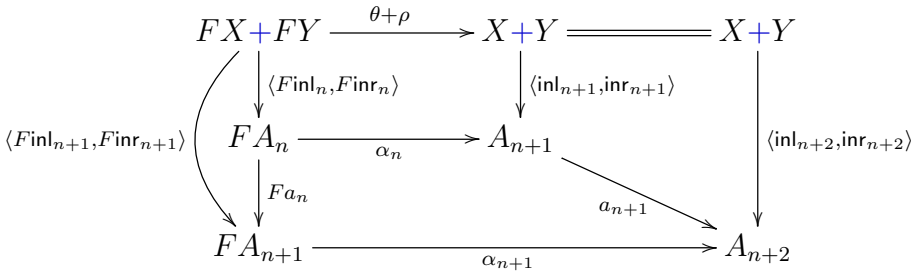
- $A_0 :\equiv X+Y$ with $\mathsf{inl}_0 :\equiv \mathsf{inl}$ and $\mathsf{inr}_n 0 :\equiv \mathsf{inr}$
- $A_n + 1 :\equiv$ pushout $\langle F\mathsf{inl}_n, F\mathsf{inr}_n\rangle\ (\theta + \rho)$ with $\langle \mathsf{inl}_{n+1}, \mathsf{inr}_{n+1}\rangle : X+Y \to A_{n+1}$ and $\alpha_n : FA_n \to A_{n+1}$ being its two inclusion maps. (We have pushouts in $\mathcal{C}$ as it has binary coproducts and coequalisers.)

The corresponding pushout diagram for A_{n+1} is the following:

$$\begin{array}{ccc} FX+FY & \xrightarrow{\theta+\rho} & X+Y \\ {\scriptstyle \langle F\mathsf{inl}_n, F\mathsf{inr}_n\rangle}\downarrow & & \downarrow{\scriptstyle \langle \mathsf{inl}_{n+1}, \mathsf{inr}_{n+1}\rangle} \\ FA_n & \xrightarrow[\alpha_n]{} & A_{n+1} \end{array}$$

The morphisms $a_n : A_n \to A_{n+1}$ are also defined inductively:

- $a_0 :\equiv \langle \mathsf{inl}_1, \mathsf{inr}_1\rangle$, which satisfies by definition $a_0 \circ \mathsf{inl}_0 = \mathsf{inl}_1$ and $a_0 \circ \mathsf{inr}_0 = \mathsf{inr}_1$.
- a_{n+1} is defined using the universal property of A_{n+1}:



Note that we can assume that the left "triangle" commutes by induction hypothesis. As we have seen, it holds for $n = 0$. We then have to show that

for the a_{n+1} we define here, it again holds. By the computation rules of a_{n+1} we have $a_{n+1} \circ \langle \mathsf{inl}_{n+1}, \mathsf{inr}_{n+1} \rangle = \langle \mathsf{inl}_{n+2}, \mathsf{inr}_{n+2} \rangle$.

If we take the colimit of the cochain (A, a), we get an object A_ω in $\mathcal{C}$ with constructors $c_n : A_n \to A_\omega$ satisfying $c_n = c_{n+1} \circ a_n$, such that it satisfies the universal property of sequential colimits. Similarly, since F preserves sequential colimits, we also have that FA_ω with Fc_n forms a colimiting cocone of the cochain (FA, Fa).

We now have to construct and check for several things on A_ω:

- an algebra structure $FA_\omega \to A_\omega$
- inclusions $\mathsf{inl}_\omega : X \to A_\omega$ and inr_ω, such that they are algebra morphisms $\theta \to \alpha_\omega$ and $\rho \to \alpha_\omega$ respectively
- $(A_\omega, \alpha_\omega)$ should have the universal property of being a coproduct of (X, θ) and (Y, ρ)

To construct the algebra structure on A_ω, we define a cocone with carrier A_ω on the cochain (FA, Fa) with a family of morphisms $d_n : FA_n \to A_\omega$ defined as the composite:

$$FA_n \xrightarrow{\alpha_n} A_{n+1} \xrightarrow{c_{n+1}} A_\omega$$

In order for this to be a cocone, we have to check whether $d_n = d_{n+1} \circ Fa_n$.

For $n = 0$, we have that it follows from the computation rule of a_1 and from the fact that c_1 and c_2 are constructors of the colimit A_ω:

$$c_1 \circ \alpha_0 = c_2 \circ a_1 \circ \alpha_0 = c_2 \circ \alpha_1 \circ Fa_0$$

From this we get a unique morphism $\alpha_\omega : FA_\omega \to A_\omega$ with computation rules $\alpha_\omega \circ Fc_n = c_{n+1} \circ \alpha_n$.

The inclusions maps $\mathsf{inl}_\omega, \mathsf{inr}_\omega$ from X and Y respectively into A_ω are defined as composing inl_0 and inr_0 with c_0. We then have to establish that

these maps are in fact algebra morphisms. We will show this here for inl_ω:

$$\begin{aligned}
\mathsf{inl}_\omega \circ \theta &= c_0 \circ \mathsf{inl}_0 \circ \theta \\
&= c_1 \circ a_0 \circ \theta \\
&= c_1 \circ \mathsf{inl}_1 \circ \theta \\
&= c_1 \circ \alpha_0 \circ F\mathsf{inl}_0 \\
&= \alpha_\omega \circ Fc_0 \circ F\mathsf{inl}_0 \\
&= \alpha_\omega \circ F(c_0 \circ \mathsf{inl}_0) \\
&= \alpha_\omega \circ F\mathsf{inl}_\omega
\end{aligned}$$

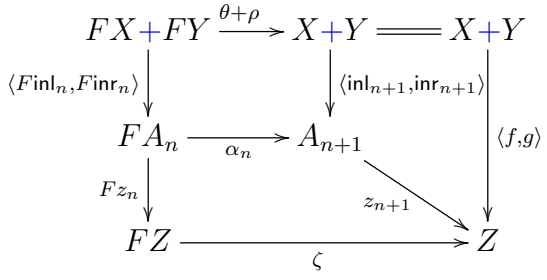
By the same reasoning inr_ω is an algebra morphism $\rho \to \alpha_\omega$.

Finally we have to show that given an algebra (Z, ζ) with algebra morphisms $f : \theta \to \zeta$ and $g : \rho \to \zeta$, we get a unique algebra morphism $h : \alpha_\omega \to \zeta$ satisfying $h \circ \mathsf{inl}_\omega = f$ and $h \circ \mathsf{inr}_\omega = g$.

Observe that Z comes with a cocone for the cochain (A, a) with $z_n : A_n \to Z$ defined inductively:

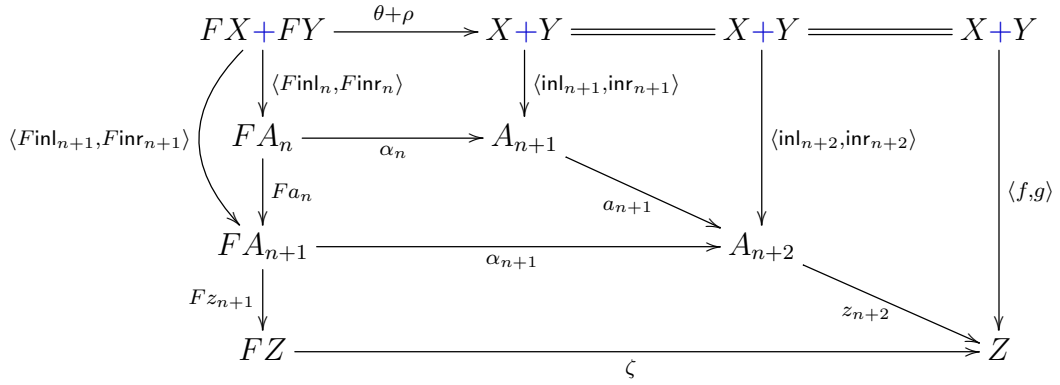
- $z_0 : A_0 \to Z$ defined as $\langle f, g \rangle$, as $A_0 :\equiv X+Y$.

- z_{n+1} is defined using the universal property of A_{n+1}:



In order for the definition of z_{n+1} to make sense, we have to check whether outer square commutes. For $n = 0$, it commutes as f and g are algebra morphisms. For the inductive step, it follows from the computation rule of z_{n+1} and the fact that f and g are algebra morphisms. As for the inductive step,

we consider the following diagram:



The triangle on the left commutes as before. This means that the outer square is the cocone on Z that defines z_{n+1}. By the universal property of A_{n+1} we then get that z_{n+1} and $z_{n+2} \circ a_{n+1}$ are equal.

We therefore get a unique morphism $z_\omega : A_\omega \to Z$ satisfying $z_\omega \circ c_n = z_n$. Showing that this is an algebra morphism $\alpha_\omega \to \zeta$ can be done by showing that we have a cocone of the cochain (FA, Fa) with carrier Z and that both $\zeta \circ Fz_\omega$ and $z_\omega \circ \alpha_\omega$ are cocone morphisms from FA_ω into Z, hence by the universal property of FA_ω they are equal.

The cocone on Z is defined with morphisms $\zeta \circ Fz_n : FA_n \to Z$. We have $\zeta \circ Fz_n = \zeta \circ Fz_{n+1} \circ Fa_n$, as z_n itself is a cocone on (A, a). The fact that $\zeta \circ Fz_\omega$ follows directly from the computation rules for z_ω. Showing that $z_{n+1} \circ \alpha_n$ is a cocone morphism also follows directly from the computation rules of z_ω and α_ω.

Lastly, we have to check whether composing z_ω with the inclusions from X and Y into Z gives us back f and g respectively. We can show this by performing the following computation:

$$\begin{aligned} z_\omega \circ \mathsf{inl}_\omega &= z_\omega \circ c_0 \circ \mathsf{inl}_0 \\ &= z_0 \circ \mathsf{inl}_0 \\ &= \langle f, g\rangle \circ \mathsf{inl} \\ &= f \end{aligned}$$

Showing that $z_\omega \circ \mathsf{inr}_\omega = g$ can be done analogously. □

Proposition 6.3.7. *The category F-alg has coequalisers.*

Proof. Let $(X, \theta), (Y, \rho) : |F\text{-alg}|$ with $f, g : \theta \to \rho$ algebra morphisms. The coequaliser of f and g can be defined using a construction similar to that of coproducts in proposition 6.3.6. We again use the idea of constructing a sequence of approximations to the coequaliser, using pushouts to ensure that the inclusion maps are algebra morphisms.

We define the objects A_n inductively:

- $A_0 :\equiv \mathsf{coeq}_{f,g}$
- $A_{n+1} :\equiv \mathsf{pushout}\ (F\beta_n \circ \langle Fin_{f,g} \rangle)\ \langle in_{f,g} \circ \rho \rangle$

i.e. the following is a pushout diagram:

$$\begin{array}{ccc} \mathsf{coeq}_{Ff,Fg} & \xrightarrow{\langle in_{f,g} \circ \rho \rangle} & \mathsf{coeq}_{f,g} \\ {\scriptstyle \langle Fin_{f,g} \rangle}\downarrow & & \downarrow{\scriptstyle \beta_{n+1}} \\ F(\mathsf{coeq}_{f,g}) & & \\ {\scriptstyle F\beta_n}\downarrow & & \\ FA_n & \xrightarrow[\alpha_n]{} & A_{n+1} \end{array}$$

where $in_{f,g} : Y \to \mathsf{coeq}_{f,g}$ is the constructor of $\mathsf{coeq}_{f,g}$ and $\beta_0 :\equiv \mathsf{id}_{\mathsf{coeq}_{f,g}}$.

We can define the morphisms $a_n : A_n \to A_{n+1}$ in the same way as before, using the universal property of pushouts. We can then proceed to define an algebra $\alpha_\omega : FA_\omega \to A_\omega$ on the colimit of (A, a). This algebra has an inclusion from (Y, ρ), which coequalises f and g and satisfies the universal property of coequalisers in F-alg. The details for these constructions are all analogous to the proof of proposition 6.3.6.

□

Proposition 6.3.8. *The forgetful functor $U : F\text{-alg} \Rightarrow \mathcal{C}$ has a left adjoint.*

Proof. This is a straightforward generalisation of proposition 4.1.4. Since F preserves sequential colimits, so does $\bar{F}_X$ for any $X : |\mathcal{C}|$, hence the initial algebra F^*X exists. □

Path constructors

Suppose we have a category $\mathcal{C} : \mathsf{Cat}$ with an adjunction $L \dashv U : \mathcal{C} \Rightarrow \mathcal{S}et$. To describe a path constructor on $\mathcal{C}$, we need the following data:

- $F : \mathcal{C} \Rightarrow \mathcal{S}et$, describing the arguments of the constructor
- $\ell, r : F \dot{\to} U$, giving the end points of the equations

In the following we will assume that $\mathcal{C}$ is a category with sequential colimits, binary coproducts and coequalisers and that F is an endofunctor on $\mathcal{C}$ preserving sequential colimits.

Let us define the category $\mathcal{C}'$ as the full subcategory of $\mathcal{C}$ with objects $X : |\mathcal{C}|$ satisfying $\ell_X = r_X$.

Proposition 6.3.9. *$\mathcal{C}'$ has an initial object.*

Proof. Since we have $L \dashv U$, we have isomorphisms

$$\phi_{X,Y} : \mathcal{C}(LX, Y) \to (X \to UY)$$

ϕ is natural in both X and Y, hence we have natural transformations:

$$\phi \circ \ell, \phi \circ r : L \circ F \dot{\to} \mathsf{id}_{\mathcal{C}}$$

It suffices to show that we have an initial object in the subcategory of $\mathcal{C}$ of objects $X : |\mathcal{C}|$ satisfying $\phi \circ \ell_X = \phi \circ r_X$.

We define X_ω to be the colimit of the following cochain in $\mathcal{C}$:

$$\begin{array}{ccccccccccccc} & & LFX_0 & & & & LFX_1 & & & & LFX_2 & & \\ & & {\scriptstyle \phi\circ r_{X_0}}\downarrow\downarrow{\scriptstyle \phi\circ \ell_{X_0}} & & & & {\scriptstyle \phi\circ r_{X_1}}\downarrow\downarrow{\scriptstyle \phi\circ \ell_{X_1}} & & & & {\scriptstyle \phi\circ r_{X_2}}\downarrow\downarrow{\scriptstyle \phi\circ \ell_{X_2}} & & \\ \emptyset & :\equiv & X_0 & \xrightarrow[x_0]{} & X_1 & & & \xrightarrow[x_1]{} & X_2 & & & \xrightarrow[x_2]{} & \cdots \end{array}$$

That is, the ω-cochain (X, x) is defined as:

$$\begin{aligned} X &: (n : \mathbb{N}) \to |\mathcal{C}| \\ X\ 0 &:\equiv \emptyset \\ X\ (n+1) &:\equiv \mathsf{coeq}_{\phi \circ \ell_{X_n}, \phi \circ r_{X_n}} \end{aligned}$$

with x_n the constructor of corresponding coequaliser.

We need to show that for X_ω, $\phi \circ \ell_{X_\omega} = \phi \circ r_{X_\omega}$. This we will achieve by showing that X_ω has a cocone structure for the cochain (LFX_n, LFx_n). Both $\phi \circ \ell_{X_\omega}$ and $\phi \circ r_{X_\omega}$ are cocone morphisms $LFX_\omega \to X_\omega$, which means that by the universal property of LFX_ω they must be equal.

For X_ω we have:

- constructors $d_n : \mathcal{C}(X_n, X_\omega)$
- satisfying $d_n = d_{n+1} \circ x_n$

F preserves sequential colimits and L preserves all colimits as it is a left adjoint, hence $L \circ F$ preserves sequential colimits. We then have that LFX_ω is the colimit of the ω-cochain (LFX_n, x_n) and therefore have:

- constructors $LFd_n : \mathcal{C}(LFX_n, LFX_\omega)$
- satisfying $LFd_n = LFd_{n+1} \circ LFx_n$

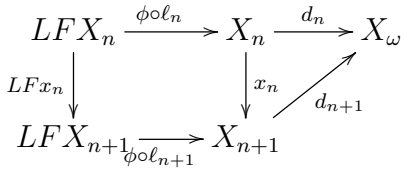
We define the (LFX_n, LFx_n)-cocone on X_ω with z_n as the composite:

$$LFX_n \xrightarrow{\phi \circ \ell_n} X_n \xrightarrow{d_n} X_\omega$$

Note that it does not matter whether we use l or r here, as $d_n = x_n \circ d_{n+1}$ and x_n is the coequaliser map, hence precomposing with $\phi \circ \ell$ or $\phi \circ r$ is going to yield the same result.

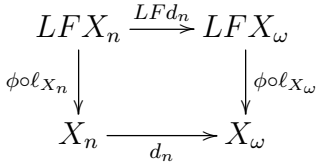
We have to check that $z_n = z_{n+1} \circ LFx_n$, which holds as we have the

following commutative diagram:



The left square holds by naturality of $\phi \circ \ell$, the right square holds as X_ω is a colimit with d_n its constructors.

Now we have constructed a (LFX_n, LFx_n)-cocone structure on X_ω, we need to check whether $\phi \circ \ell_{X_\omega}$ and $\phi \circ r_{X_\omega}$ are indeed cocone morphisms. This amounts to checking whether the following commutes:



This square commutes by naturality of $\phi \circ \ell$. By the same reasoning $\phi \circ r_{X_\omega}$ is a cocone morphism, hence by the universal property of LFX_ω we get that $\phi \circ \ell_{X_\omega} = \phi \circ r_{X_\omega}$. □

Proposition 6.3.10. *The inclusion/forgetful functor $U' : \mathcal{C}' \Rightarrow \mathcal{C}$ has a left adjoint.*

Proof. We can take the proof of initiality and replace the object X_0 with any $X : |\mathcal{C}|$. We define the left adjoint $L' : \mathcal{C} \Rightarrow \mathcal{C}'$ with $L'X$ being the colimit of this sequence. □

Proposition 6.3.11. *$\mathcal{C}'$ has coproducts and coequalisers.*

Proof. We claim that the left adjoint L' to the inclusion functor of the subcategory $\mathcal{C}'$ into $\mathcal{C}$ gives us coproducts and coequalisers.

Let $X, Y : |\mathcal{C}|$ with $\theta : \ell_X = \rho_X$ and $\rho : \ell_Y = \rho_Y$. By assumption, $\mathcal{C}$ has coproducts. The object $L'(X{+}Y)$ is the coproduct of (X, θ) and (Y, ρ), as

we have for any $(Z, \zeta) : |\mathcal{C}'|$:

$$\begin{aligned}\mathcal{C}'(L'(X+Y),(Z,\zeta)) &= \mathcal{C}(X+Y,Z)\\ &= \mathcal{C}(X,Z)\times\mathcal{C}(Y,Z)\\ &= \mathcal{C}'((X,\theta),(Z,\zeta))\times\mathcal{C}'((Y,\rho),(Z,\zeta))\end{aligned}$$

Similarly for coequalisers, suppose that we again have $(X,\theta),(Y,\rho) : |\mathcal{C}'|$ with $f, g : \mathcal{C}'((X,\theta),(Y,\rho))$, so we really just have $f, g : \mathcal{C}(X,Y)$, then taking the coequaliser to be $L'(\mathsf{coeq}_{f,g})$, we can calculate for any $(Z,\zeta) : |\mathcal{C}'|$:

$$\mathcal{C}'(L'(\mathsf{coeq}_{f,g}),(Z,\zeta)) = \mathcal{C}(\mathsf{coeq}_{f,g},Z) = (h : \mathcal{C}(Y,Z)) \times (h\circ f = h\circ g)$$

which again by $\mathcal{C}'$ being a full subcategory means that $L'(\mathsf{coeq}_{f,g})$ is indeed a coequaliser in $\mathcal{C}'$ of f and g. □

Proposition 6.3.12. *$\mathcal{C}'$ has sequential colimits.*

Proof. Suppose we have a cochain in $\mathcal{C}'$, i.e. we have:

- $X : \mathbb{N} \to |\mathcal{C}|$
- $\theta : (n : \mathbb{N}) \to \ell_{X_n} = r_{X_n}$
- $x : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_{n+1})$

By the assumptions on $\mathcal{C}$, we the colimit of the cochain (X, x) in $\mathcal{C}$: we have:

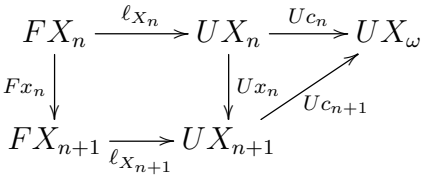
- $X_\omega : |\mathcal{C}|$
- $c : (n : \mathbb{N}) \to \mathcal{C}(X_n, X_\omega)$
- $g : c_n = c_{n+1} \circ x_n$

Similarly, as F preserves sequential colimits FX_ω with Fc is a colimiting cocone of the cochain (FX, Fx).

Note that the natural transformation ℓ gives rise to the following cocone of the cochain (FX, Fx):

- $UX_\omega : |\mathcal{C}|$
- $d : \mathcal{C}(FX_n, UX_\omega)$ defined as $Uc_n \circ \ell_{X_n}$
- $h : d_n = d_{n+1} \circ Fx_n$

The equality h holds by the considering the following diagram:



The left square holds by naturality of ℓ and the right triangle commutes due to the fact that X_ω is the colimit of the cochain (X, x).

We claim that both l_{X_ω} and r_{X_ω} are cocone morphisms from colimiting FX_ω with Fc into UX_ω with Uc, as we have $Uc_n \circ \ell_{X_n} = \ell_{X_\omega} \circ Fc_n$ by naturality. Since $\ell_{X_n} = r_{X_n}$ for every $n : \mathbb{N}$, l_{X_ω} and r_{X_ω} are cocone morphisms from the same cocone, hence $l_{X_\omega} = r_{X_\omega}$. This gives us that X_ω is in the subcategory $\mathcal{C}'$. Since it is a full subcategory, we also get the universality from $\mathcal{C}$. □

6.3.3 Putting it all together

We can now put all the results together and show that we have initial algebras for $\mathcal{S}et$-sorted quotient inductive-inductive definitions where all the arguments functors preserve sequential colimits.

Theorem 6.3.13. *Let s : Spec be a specification of a $\mathcal{S}et$-sorted quotient inductive-inductive definition. If all the arguments functors contained in the specification s preserve sequential colimits, then the category Alg_s has an initial object.*

Proof. Using the results from the previous section, we can create a left adjoint $L : \mathcal{S}et \Rightarrow \mathsf{Alg}_s$ to the forgetful functor $U : \mathsf{Alg}_s \Rightarrow \mathcal{S}et$ by induction on s. Since left adjoints preserve colimits, $L\,\emptyset$ is the initial object of Alg_s. □

Making the theorem work for arbitrary quotient inductive-inductive definitions means generalising in two dimensions. We have to be able to deal with:

- arbitrary sorts: the constructions given in this chapter only work for $\mathcal{S}et$-sorted definitions
- cochains of arbitrary length: our construction works with colimits of length ω. In general we will need longer cochains, e.g. for infinitely branching trees, we need to consider the colimit of a cochain of length $\omega + \omega$.

6.4 Related work

A well-known result from category theory on constructing left adjoints is Freyd's adjoint functor theorem. This theorem gives us a left adjoint for a functor, given that its domain is complete, that the functor is continuous and that the solution set condition is satisfied. In section 5.3, we have basically shown that the categories we are working with are finitely complete and that the forgetful functors are continuous. (We have not explicitly shown that the categories have a terminal object, but this is easy to show.) Extending this to completeness means that we have to show that they have terminal objects and generalise binary products to arbitrary ones. More difficult would be to show that the solution set condition is satisfied. This seems to be as difficult as constructing the left adjoint itself in our case.

As for showing cocompleteness of categories of monad algebras, there are several treatments of colimits of monad algebras [Lin69; BW85]. The main result in [Lin69] is that, given that $\mathcal{C}$ is cocomplete, the category of monad algebras M-Alg for a monad $M : \mathcal{C} \Rightarrow \mathcal{C}$ is cocomplete if and only if it has reflexive coequalisers. M-Alg has reflexive coequalisers if M preserves them.

In [LS13b], the authors use sequential colimits to construct the monads corresponding to the higher inductive types, using techniques from [Kel80]. Evaluating these monads at the initial object yields the carrier of

the initial algebra for the higher inductive type. The construction given by the authors is therefore not hugely dissimilar from ours. An important difference is that our construction is carried out in type theory itself.

The construction of the initial object for path constructors can also be seen as a generalisation of the construction of propositional truncation as a sequential colimit [Doo16]. Our construction is performed in a set truncated setting. However, seeing as the technique is very similar to the one used to construct propositional truncation, it seems that our result can be generalised to the untruncated setting.

Chapter 7

Concluding remarks

In this thesis we have given a formal specification of quotient inductive-inductive definitions, intended as a stepping stone towards a theory of higher inductive(-inductive) types. This theory has been presented in such a way that formalising it in type theory is straightforward.

After the first two introductory and preliminary chapters, chapter 3 was devoted to giving examples and intuition for quotient inductive-inductive types, before diving into the formal definition. We argued that even only considering higher inductive(-inductive) types truncated to sets is already a useful extension over ordinary inductive types. The set truncation also entails that we have to consider only the point and first path constructors, as anything higher will be trivial.

In the examples of chapter 3, we uncovered that quotient inductive-inductive definitions set themselves apart from ordinary inductive definitions in the following regards:

- instead of defining a single type, we may have a collection of *dependent sorts*
- any constructor may *refer to any previous constructor*
- the result type of a constructor may be an *equation* in any of the sorts, i.e. we allow for *path constructors*

Both inductive-inductive definitions as well as higher inductive types have the property of allowing for references to previous constructors, which prompted our investigations into a uniform treatment of both classes of inductive definitions.

In chapter 4 we give the formal specification of quotient inductive- inductive definitions, which is given simultaneously with its interpretation as categories of algebras: we characterise quotient inductive-inductive definitions roughly as iterated dialgebras (definition 4.4.1). Dealing with dependent sorts means that ordinary dialgebras do not suffice: the categories of dialgebras are fibred over all sorts below the sort of the current constructor. Dealing with this means we define the category of algebras as an equaliser category of a category of dialgebras. The category of algebras for a path constructor is simply the category of algebras of the previous constructors extended with an equation on those algebras given by natural transformations.

Since we have not focussed on our specifications of inductive definitions being sound in the sense of the categories of algebras having initial objects, the system can also be used to work with equational theories in type theory. While there are plenty of equational theories that are nicely behaved in the sense that the corresponding category of models has an initial object, this is not always the case. A notable example of this phenomenon is the category of fields, which can be described with our framework, but does not have an initial object.

As we have given a specification of quotient inductive-inductive definitions, we ought to prove properties about them. In this thesis we investigate several properties: the logical equivalence of initiality and induction and the construction of initial algebras via sequential colimits.

Chapter 5 is devoted to the correspondence between initiality and induction in the context of quotient inductive-inductive definitions (theorem 5.3.9). We first give a categorical characterisation of the induction principle as the section principle: every algebra fibration has a section. We then go on to show that in the presence of binary products and equalisers this principle is logically equivalent to initiality. This proof follows what one

would intuitively do in type theory if one shows that some induction principle implies initiality: you first show that weak initiality holds for which we need to produce a constant algebra family (this is the construction of products). Establishing that the resulting morphism is unique can be done by employing the induction principle again with the motive being the pointwise equivalence of the two morphisms. Giving the methods then amounts to giving the equaliser of the two morphisms. Since we already know the categorical structure of the algebras, proving that the section principle and initiality coincide then amounts to giving constructions of products and equalisers. This approach saves us from first having to come up with an induction principle.

Interesting to note is that while initiality is a property of an object that requires us only to have the objects and morphisms of the category at hand, the section principle requires the full categorical structure, i.e. we need composition, identity morphisms and laws and associativity. Initiality is an attractive property in the light of working in an untruncated setting, i.e. working with hom-types as opposed to hom-sets, as we do not have to bother with the category laws and hence do not have to deal with any further coherence laws. However, comparing it to the section principle requires us to use more category structure and laws.

The second part of chapter 5 gives a derivation of the induction principle for quotient inductive-inductive definitions. Since we know what display maps and sections amount to in $\mathcal{S}et$ in a type theoretic sense, i.e. they are type families and dependent functions, and given that all our categories are in some way built upon $\mathcal{S}et$, we can use this information to derive the type theoretic induction principle for our quotient inductive-inductive definitions.

In chapter 6 we consider the existence of inductive definitions, i.e. the existence of initial objects in the categories of algebras. The way we set things up in chapter 4 allows us to give inductive definitions of which the corresponding category of algebras does not necessarily have an initial object. As our quotient inductive-inductive definitions subsume ordinary inductive types in the sense of providing an endofunctor on $\mathcal{S}et$, we can con-

sider the double powerset functor which does not have an initial algebra. We therefore need to make sure that our definitions are *strictly positive*.

We give constructions of initial algebras for a class of $\mathcal{S}et$-sorted quotient inductive-inductive definitions where the functors are ω-cocontinuous (theorem 6.3.13). These constructions can be performed inside the type theory, which gives us the result that having natural numbers and coequalisers/quotients is enough to be able to construct a wide range of quotient inductive-inductive definitions.

In appendix A we present some preliminary work on characterising strictly positive functors for quotient inductive-inductive definitions, by generalising containers. We give a generalisation of these to functors from any category into $\mathcal{S}et$, which allows us to express the data needed for a $\mathcal{S}et$-sorted quotient inductive definition. We also present a generalisation of this to situations where the category of sorts is a presheaf category over $\mathcal{S}et$.

In appendix B we try to lift the restriction to sets: instead of considering hom-sets in our categories of sorts and algebras, we broaden this to hom-*types*. This would turn our theory of quotient inductive-inductive types into one of higher inductive-inductive types with point constructors and path constructors that construct paths between points, i.e. no higher path constructors. As opposed to moving to $(\infty, 1)$-categories straight away, we move from hom-sets to hom-types and go through all the constructions to see where we run into coherence issues. Somewhat surprisingly issues already show up when considering only point constructors for $\mathcal{T}ype$-sorted definitions. The category of F-algebras for an endofunctor on $\mathcal{T}ype$ is no longer a category that satisfies the category laws strictly, unlike $\mathcal{T}ype$ itself. Even if F happens to be a strict functor, we still do not end up with a strict $(\infty, 1)$-category. If we add a point constructor to this category of algebras, we increase the level of coherence needed. Therefore the number of coherence problems we have to deal with increases with the number of constructors, whether they are point constructors of path constructors.

7.1 Future work

7.1.1 Metaprogramming and generic programming

Given that our definition of quotient inductive-inductive types can be formulated inside type theory, one avenue for future work would be applying this definition to generic programming ideas. Having these definitions as the basis of the implementation of inductive definitions in your theory is useful when one wants to use metaprogramming techniques to define programs abstracting over data types. One aspect of our approach is that we stay with the idea of an inductive definition being given as a list of constructors, as opposed to simplifying the situation to being a code of a single endofunctor. Staying with the list of constructors idea also means that we could build a system for writing attribute grammars internally without needing any external tools, allowing for aspect oriented programming.

7.1.2 Invariance of descriptions under equivalence of constructors

An important property that should hold is that the definitions should be invariant under equivalence of constructors. If we have two specifications $s, s' : \mathsf{Spec}$ with the same dependent sorts, such that if $|\mathsf{Alg}_s| = |\mathsf{Alg}_{s'}|$, i.e. all the constructors combined of s are equivalent to those of s', then the initial object of Alg_s should have an isomorphic carrier to that of $\mathsf{Alg}_{s'}$. This is an important property that is used often to reason about equivalence of inductive definitions. For example, it implies that the definitions are invariant under reordering of constructors.

7.1.3 Generalised containers

We have given the definition of generalised containers as a means of describing functors into $\mathcal{S}et$ and presheaf categories. For descriptions of quotient inductive-inductive definitions we need to be able to handle functors

into sort categories. Generalising the containers to support this is an avenue of future work.

Along with this one should also establish that the usual properties of ordinary containers hold, i.e. container morphisms completely describe natural transformations between extensions of containers.

We have noted that we can use quotient inductive types to define functors on $\mathcal{S}et$ which are not representable as an ordinary container, namely propositional truncation. It would be interesting to see whether we can adjust the definition of container to subsume such functors.

7.1.4 Constructing initial algebras

We have given constructions of initial algebras for $\mathcal{S}et$-sorted definitions where the arguments functors were ω-cocontinuous/finitary. Future work would be to generalise this to other ordinals as well and make the construction work for arbitrarily sorted definitions.

One possible approach would be to have internally to the type theory a syntax for (strictly positive) quotient inductive-inductive definitions with ordinal annotations, so one could compute at what ordinal the colimits stabilise.

7.1.5 Generalising to higher inductive types

The ultimate goal of this project is to have a theory of higher inductive types. In appendix B we have shown what the kind of issues are we run into when trying to move our results from the category theoretic setting, where we work only with sets, to a higher category theoretic setting without any truncation. In the chapter we also argue that the naive approach gets unworkable very quickly. To adequately describe a theory of higher inductive types, one has to turn to $(\infty, 1)$-categories. Defining $(\infty, 1)$-categories in type theory is ongoing work and seems to require extending the type theory with an internal notion of strict equality, which allows us to talk about definitional equalities in the type theory itself, as well as propositional equality [ACK16b; ACK16a].

Appendix A

Containers for quotient inductive-inductive definitions

The inductive definitions therefore need to be *strictly* positive: positivity alone does not suffice. There are different ways to formally specify strictly positive functors. We can use a syntactic way to describe them as the class of functors that contains all constant functor, closed under sums and products of strictly positive functors, exponentiation with a constant on the left of the arrow, and taking fixpoints [Mor07]. A more compact way to characterise strictly positive functors on $\mathcal{S}et$ in type theory is as *containers* [AAG05]:

Definition A.0.1. A *container* on $\mathcal{S}et$ consists of:

- $S : \mathsf{Set}$, a type of *shapes*
- $P : S \to \mathsf{Set}$, a family of *position*, indexed by the shapes.

The container with shapes S and positions P is denoted as $S \lhd P$

The corresponding functor is called the *extension* of the container:

Definition A.0.2. Given a container $S \lhd P$, its *extension* is the functor $[\![S \lhd P]\!] : \mathcal{S}et \Rightarrow \mathcal{S}et$ with its action on objects defined as, for every $X : \mathsf{Set}$:

$$[\![S \lhd P]\!]\ X :\equiv (s : S) \times (P\ s \to X)$$

and its action on functions $f : X \to Y$:

$$[\![S \lhd P]\!]\ f :\equiv \lambda(s,t).(s, f \circ t)$$

A.1 Containers for $\mathcal{S}et$-sorted definitions

To give the data for a quotient inductive-inductive definition, we often need more than just endofunctors on $\mathcal{S}et$. We are generally working with functors $\mathsf{Alg}_s \Rightarrow S_i$ where $s : \mathsf{Spec}$ describes the previous constructors and S_i is sort category describing the sort of the constructor we are defining. Containers have a generalisation to *indexed containers* which describe functors between slice categories of $\mathcal{S}et$. This concept is again an instance of the more general notion of *polynomial functor* [Koc11], which describes strictly positive functors between slice categories of a locally cartesian closed category. We cannot expect Alg_s to be locally cartesian closed in general: if we take s to be the specification corresponding to setoids, then Alg_s is equivalent to the category setoids, which is not locally cartesian closed [AK12].

If we look at containers a bit more closely, we see that they are coproduct of a family of representable functors. This observation leads us to *generalised containers*, also known as *familially representable functors* [CJ95]:

Definition A.1.1. A *generalised container* on a category $\mathcal{C}$ consists of:

- $S : \mathsf{Set}$, a type of shapes,
- $P : S \to |\mathcal{C}|$, a family of representing objects, indexed by the shapes.

The extension generalises straightforwardly:

Definition A.1.2. Given a container $S \lhd P$ on a category $\mathcal{C}$, its extension is the functor $[\![S \lhd P]\!] : \mathcal{C} \Rightarrow \mathcal{S}et$ with its action on objects defined as, for every $X : |\mathcal{C}|$:

$$[\![S \lhd P]\!]\ X :\equiv (s : S) \times \mathcal{C}(P\ s, X)$$

and its action on functions $f : X \to Y$:

$$[\![S \lhd P]\!]\ f :\equiv \lambda(s,t).(s, f \circ t)$$

To describe the end points of path constructors, we use natural transformations. *Container morphisms* are used to represent natural transformations between containers. For generalised containers they are as follows:

Definition A.1.3. Given $\mathcal{C}$-containers $S \lhd P$ and $T \lhd Q$, a container morphism consists of:

- $f : S \to T$
- $g : (s : S) \to \mathcal{C}(Q\ (f\ s), P\ s)$

with its extension being the natural transformation:

$$\llbracket f, g \rrbracket : (X : |\mathcal{C}|) \to \llbracket S \lhd P \rrbracket\ X \to \llbracket T \lhd Q \rrbracket\ X$$
$$\llbracket f, g \rrbracket\ X\ (s, t) :\equiv (f\ s, t \circ (g\ s))$$

Naturality follows from the associativity law of $\mathcal{C}$.

A.2 Containers for arbitrarily sorted definitions

We have given a way to describe strictly positive functors and natural transformations needed to describe $\mathcal{S}et$-sorted quotient inductive-inductive definitions. However, the functors we work with are not generally functors into $\mathcal{S}et$, but may also be into any sort category.

In this section we will show how this can be done for the special case $\mathcal{F}am$. Suppose we have a category $\mathcal{C}$, which we can think of as being a category of $\mathcal{F}am$-sorted algebras. It is therefore equipped with a forgetful functor $U : \mathcal{C} \Rightarrow \mathcal{F}am$. Describing the arguments of a $\mathcal{F}am$-sorted constructor over $\mathcal{C}$ requires us to give a functor $F : \mathcal{C} \Rightarrow \mathcal{F}am$ such that $t_1 \circ F = t_1 \circ U$, where $t_1 : \mathcal{F}am \Rightarrow \mathcal{S}et$ is its forgetful functor.

Note that we have $\mathcal{F}am = \mathcal{S}et^I$, therefore by the cartesian-closedness of Cat, we have $\mathcal{C} \Rightarrow \mathcal{S}et^I = \mathcal{C} \times I \Rightarrow \mathcal{S}et$. To give a functor $F : \mathcal{C} \Rightarrow \mathcal{F}am$ is to give two functors $F^0, F^1 : \mathcal{C} \Rightarrow \mathcal{S}et$ along with a natural transformation $\alpha : F^1 \dot{\to} F^0$.

Furthermore, we have the requirement that $F^0 = t_1 \circ U$. If we assume that we have an adjunction $L \dashv U : \mathcal{C} \Rightarrow \mathcal{F}am$, $t_1 \circ U$ will also have a left adjoint (as t_1 also has a left adjoint). If $t_1 \circ U$ has a left adjoint, it is a representible functor, which means it is also a container.

A.3 Limitations of containers

While in the traditional setting, containers (on $\mathcal{S}et$) seem to be an adequate way to characterise strictly positive functors, it has its limitations. Let us consider the propositional truncation operation on Set: $||\_|| : \mathsf{Set} \to \mathsf{Set}$. Let $S \lhd P$ be its container representation, then the following holds:

$$1 = ||1|| = (s : S) \times (P\ s \to 1) = S$$

Therefore we know that that the shapes $S = 1$, hence $||\_||$ has to be a representable functor. Let $P : \mathsf{Set}$ be its representing object. P has to either be empty or inhabited. If it is empty, then we have $0 = ||0|| = 0 \to 0 = 1$, a contradiction. If it is inhabited, we have $1 = ||\mathsf{Bool}|| = P \to \mathsf{Bool}$, however $P \to \mathsf{Bool}$ has at least two distinct inhabitants: $\lambda x.\mathsf{true}$ and $\lambda x.\mathsf{false}$, also a contradiction. Hence $||\_||$ is not a container.

Now this fact is not necessarily bad for the expressiveness of our system. If we wanted to express a constructor of a type A such as $c : ||A|| \to A$, we could simply incorporate propositional truncation into our inductive definition, i.e. add another sort $B : \mathsf{Set}$ which has a constructor $d : A \to B$ and a constructor of type $(x\ y : B) \to x = y$.

Appendix B

Moving to an untruncated setting

In the previous chapters, we have mostly worked with sets. In this chapter we show what kind of issues one encounters if we work in an untruncated setting instead, motivating why we went with the choice to work in the set truncated setting in the first place. If we want to generalise the theory to higher inductive types, we have to lift this restriction and somehow deal with these issues.

The place where we had to ensure that certain types were sets, was in the definition of category: we worked with *sets* of morphisms. This means that the category laws are *propositions* and saves us from having to worry too much about reasoning about equality of morphisms: any two such proofs will be equal.

If we have categories with hom-*types*, then we have no guarantees that the category laws interact nicely. For example, if we have four composable morphisms:

$$X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} W \xrightarrow{i} V$$

If we want to show that the following equation holds:

$$((i \circ h) \circ g) \circ f = i \circ (h \circ (g \circ f))$$

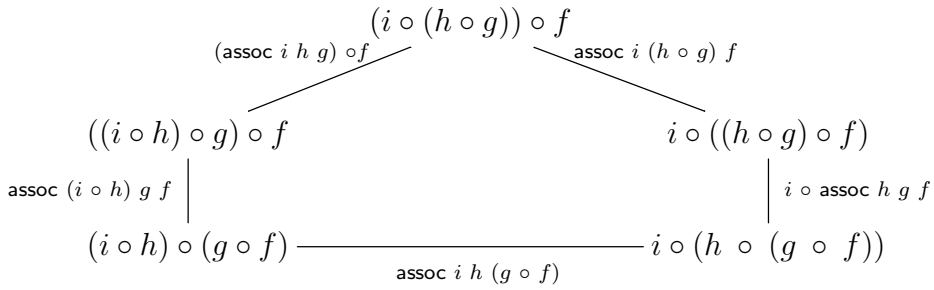
then we have a choice in what order we apply the category laws. There is a priori no guarantee that these choices yield equal proofs: we need to add

this assumption as an extra *coherence condition*. In the case of associativity interacting with itself, it is as follows:

Definition B.0.1 (Coherence condition for associativity)**.** The witness of associativity

$$\mathsf{assoc} : (h : \mathcal{C}(Z, W))\,(g : \mathcal{C}(Y, Z))\,(f : \mathcal{C}(X, Y)) \to ((h \circ g) \circ f) = (h \circ (g \circ f))$$

is *coherent* if for any composable arrows i, h, g, f the following commutes:



We can formulate similar conditions for the interactions between the identity laws and associativity. The story does not end there however: there is no reason to assume that the equality between equalities of morphism is propositional. The coherence condition of associativity needs to behave nicely in harmony with the other category laws and coherence conditions: the coherence conditions themselves require further coherence conditions. This phenomenon does not stop: we get an infinite tower of coherence conditions.

Sometimes we are lucky with the concrete categories we are working with, in that the category laws hold up to definitional equality: they are satisfied *strictly*. Examples of such categories are Type and Fam. In these cases, as the category laws themselves are satisfied trivially, any higher coherence law will also be satisfied trivially.

For categories of algebras, this is not the case. Even if we consider F-algebras with F a strict endofunctor on Type, F-alg will not satisfy the category laws strictly, with the usual definition. In [Cra13], the author shows essentially that we cannot find a definition of the category of pointed types

that satisfies the category laws strictly. Since pointed types are a special case of F-algebras, we therefore cannot hope to find a nice definition of F-alg that is strict.

The way to deal with coherences is by considering $(\infty, 1)$-categories [Cam13]. The usual definitions/models of $(\infty, 1)$-categories are given by using simplicial sets. This is not a practical approach for our purposes, as we cannot easily move between simplicial sets and types. Another approach is to consider simplicial *types*, which is as of yet an open problem in homotopy type theory. One attempt to solve this and give a definition of $(\infty, 1)$-categories is by extending the type theory with an internal notion of strict equality [ACK16b; ACK16a].

However, as we are only concerned with a finite amount of categorical structure: we are really only interested in the category of algebras satisfying the identity and associativity laws, we will in this chapter see how far we get by taking a *lazy* approach. We will only add those coherence conditions we actually need to show that the categorical laws are satisfied in all the categories of algebras.

B.1 Coherence laws for functors

Talking about the category laws in an untruncated setting also means that we now have to worry about what it means for a functor to preserve them. If we take for example the left identity law, we notice that there are multiple ways to produce an equality $F(idY \circ f) = Ff$. we can either use the left identity law of the domain of F or the one of its codomain, by first appealing to the fact that F preserves composition and identity morphisms. The functor F preserves the left identity law if these two approaches yield the same equality:

Definition B.1.1. A functor $F : \mathcal{C} \Rightarrow \mathcal{D}$ satisfies the coherence law for the

left identity law of $\mathcal{C}$ if the following commutes for any $f : \mathcal{C}(X, Y)$:

$$\begin{array}{ccc} F(\mathsf{id}_{\mathcal{C}} Y \circ_{\mathcal{C}} f) & \xrightarrow{F\ (\mathsf{left\text{-}id}_{\mathcal{C}}\ f)} & Ff \\ {\scriptstyle F\text{-}\circ\ (\mathsf{id}_{\mathcal{C}}\ Y)\ f}\downarrow & & \downarrow{\scriptstyle \mathsf{left\text{-}id}_{\mathcal{D}}\ F\ f} \\ F(\mathsf{id}_{\mathcal{C}} Y) \circ_{\mathcal{D}} f & \xrightarrow[F\text{-}\mathsf{id}\ Y\ \circ_{\mathcal{D}}\ f]{} & \mathsf{id}_{\mathcal{D}}(FY) \circ_{\mathcal{D}} f \end{array}$$

where F-id and F-$\circ$ are the witnesses of the functoriality of F.

The coherence law for the right identity law is defined similarly.

Definition B.1.2. A functor $F : \mathcal{C} \Rightarrow \mathcal{D}$ satisfies the coherence law for the associativity law of $\mathcal{C}$ if, given three composable arrows:

$$X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} W$$

the following commutes

$$\begin{array}{ccc} F((h \circ g) \circ f) & \xrightarrow{F\ (\mathsf{assoc}_{\mathcal{C}}\ h\ g\ f)} & F(h \circ (g \circ f)) \\ {\scriptstyle F\text{-}\circ\ (h\ \circ\ g)\ f}\downarrow & & \downarrow{\scriptstyle F\text{-}\circ\ h\ (g\ \circ\ f)} \\ F(h \circ g) \circ Ff & & Fh \circ F(g \circ f) \\ {\scriptstyle F\text{-}\circ\ h\ g\ \circ\ F\ f}\downarrow & & \downarrow{\scriptstyle F\ h\ \circ\ F\text{-}\circ\ g\ f} \\ (Fh \circ Fg) \circ Ff & \xrightarrow[\mathsf{assoc}_{\mathcal{D}}\ Fh\ Fg\ Ff]{} & Fh \circ (Fg \circ Ff) \end{array}$$

B.1.1 Generalised containers

Although we are not able to internally define the type of strict functors, there is a definable class of functors that happen to be strict: containers on Type (see definition A.0.1). As the functorial action of containers are defined in terms of composition of functions in Type, the functor laws reduce to the identity and associativity laws of Type, which are satisfied strictly. As such, containers on Type form a class of strict functors.

For generalised containers (definition A.1.1), the functors again inherit the functor laws from the category laws of the domain of the functor. If the generalised container maps out of a strict category, then it defines a strict

functor. However, in practice, these categories will not be strict as they will be categories of algebras.

B.2 Sort categories

The categories Type and Fam satisfy the category laws and their (higher) coherences definitionally. Whether a sorts category $[\![\mathcal{S}]\!]$ given by a $\mathcal{S}$: Sorts is strict, depends on the functors in the list $\mathcal{S}$. Looking at the definition of the sorts categories, if all the functors involved satisfy the functor laws definitionally then the resulting categories will be strict as well. This means that if we give all the functors in $\mathcal{S}$ as generalised containers, we end up with strict sort categories, which is quite a reasonable assumption to make.

B.3 Categories of dialgebras

So far we have not given a precise definition of (F, G)-dialg for some functors $F, G : \mathcal{C} \to \mathcal{D}$: we have not formally defined composition and so on. In definition 4.2.2, we have defined the category its objects and morphisms. The objects are defined in terms of objects from $\mathcal{C}$ and morphisms from $\mathcal{D}$. Morphisms are defined in terms of morphisms in $\mathcal{C}$ and $\mathcal{D}$ along with equalities between them. If we are only interested in objects and morphisms of (F, G)-dialg, we need to know what the objects and morphisms of $\mathcal{C}$ and $\mathcal{D}$ are and the actions of F and G on those. Once we go beyond morphisms, we run into trouble.

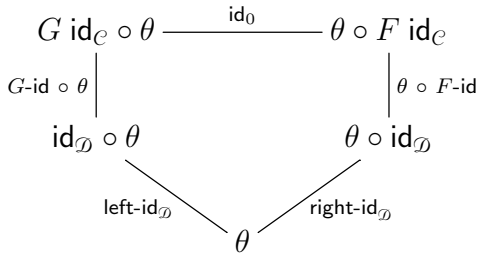
To illustrate these issues, we will look at the definition of the (F, G)-dialg in detail.

B.3.1 Identity morphisms

Given an object $(X, \theta) : |(F, G)\text{-dialg}|$, we define the identity morphism as

$$\mathsf{id}_{(F,G)\text{-dialg}}\ (X, \theta) :\equiv (\mathsf{id}_{\mathcal{C}}\ X, \mathsf{id}_0)$$

with id_0 defined as:

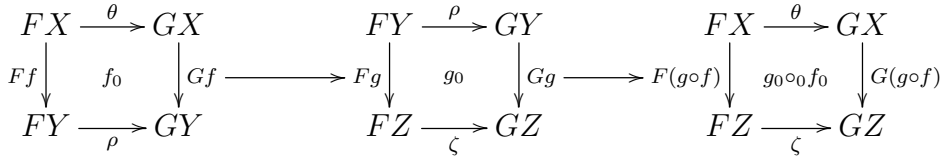


Unsurprisingly, the construction of identity morphisms relies on the functor F and G preserving identity morphisms. Perhaps more surprising is that the construction also relies on the identity *laws* of the category $\mathcal{D}$, i.e. a category structure "one level up" from identity morphisms.

In our cases, the codomain of functors F and G is always Type or some other category of sorts: it does not change with the number of constructors.

B.3.2 Composition

Suppose we are given algebras $(X, \theta), (Y, \rho)(Z, \zeta) : |(F, G)\text{-dialg}|$ and morphisms $(g, g_0) : (Y, \rho) \to (Z, \zeta)$ and $(f, f_0) : (X, \theta) \to (Y, \rho)$. If we want to compose the two morphisms, we need a way to glue the squares g_0 and f_0 together, i.e. we need an operation:



The (vertical) composition of the squares $g_0 \circ_0 f_0$ is defined as the composite:

$$\begin{array}{ccc}
G\ (g \circ f) \circ \theta & \xrightarrow{g_0\ \circ_0\ f_0} & \zeta \circ F(g \circ f) \\
{\scriptstyle G\text{-}\circ\ g\ f\ \circ\ \theta}\big\downarrow & & \big\downarrow{\scriptstyle \zeta\ \circ\ F\text{-}\circ\ g\ f} \\
(G\ g \circ G\ f) \circ \theta & & \zeta \circ (F\ g \circ F\ f) \\
{\scriptstyle \mathsf{assoc}_{\mathcal{D}}}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{assoc}_{\mathcal{D}}} \\
G\ g \circ (G\ f \circ \theta) & & (\zeta \circ F\ g) \circ F\ f \\
{\scriptstyle G\ g\ \circ\ f_0}\big\downarrow & & \big\downarrow{\scriptstyle g_0\ \circ\ F\ f} \\
G\ g \circ (\rho \circ F\ f) & \xrightarrow{\overline{\mathsf{assoc}_{\mathcal{D}}}} & (G\ g \circ \rho) \circ F\ f
\end{array}$$

Composition of dialgebra morphisms is then:

$$(g, g_0) \circ (f, f_0) :\equiv (g \circ f, g_0 \circ_0 f_0)$$

As with identity morphisms, we notice that we need to appeal to the functors preserving the same kind of structure we are defining here: they need to preserve composition. We also need categorical structure one level up from composition from the category $\mathcal{D}$, namely associativity.

Looking at this definition of composition, we notice that even when $\mathcal{C}$ and $\mathcal{D}$ are strict categories, with F and G being strict functors, e.g. when we consider F-algebras on Type with F given as a container, composition will not be strictly associative.

B.3.3 Category laws

If we want to talk about the identity laws in a category of dialgebras, we need to know what equality between dialgebra morphisms looks like. We can characterise it as follows:

Proposition B.3.1. *Let* $(f, f_0), (g, g_0)$ *be two dialgebra morphisms* $(X, \theta) \to$

(Y, ρ) *in* (F, G)-dialg*, then we have the following equivalence of equalities:*

$$\begin{aligned}((f, f_0) = (g, g_0)) = (p : f = g) \\ \times (p_0 : \quad G\ f \circ \theta \xrightarrow{f_0} \rho \circ F\ f \quad) \\ {\scriptstyle G\ p \circ \theta}\Big| \qquad\qquad \Big|{\scriptstyle \rho \circ F\ p} \\ G\ g \circ \theta \xrightarrow[g_0]{} \rho \circ F\ g\end{aligned}$$

Proof. Using the fact that an equality of dependent pairs is a dependent pair of equalities, we get that

$$((f, f_0) = (g, g_0)) = (p : f = g) \times (p_0 : f_0 =_p^{\lambda h. G\ h \circ \theta = \rho \circ F\ h} g_0$$

Having a path p_0 over a family of equalities is equivalent to the square by the usual reasoning. □

To give a witness for the left identity law, we need to show given:

- objects $(X, \theta), (Y, \rho) : |(F, G)\text{-dialg}|$
- with a morphism $(f, f_0) : (X, \theta) \to (Y, \rho)$

that $\mathsf{id}_{(F,G)\text{-dialg}}\ (Y, \rho) \circ_{(F,G)\text{-dialg}} (f, f_0) = (f, f_0)$. Unfolding definitions this reduces to having to show that:

$$(\mathsf{id}_{\mathcal{C}}\ Y \circ_{\mathcal{C}} f, \mathsf{id}_0\ \rho \circ_0 f_0) = (f, f_0)$$

Applying proposition B.3.1 this is the same as giving a proof $p : \mathsf{id}_{\mathcal{C}}\ Y \circ f = f$ along with a square:

$$\begin{aligned}G\ (\mathsf{id}_{\mathcal{C}}\ Y \circ f) \circ \theta \xrightarrow{\mathsf{id}_0\ \rho \circ_0 f_0} \rho \circ F\ (\mathsf{id}_{\mathcal{C}}\ Y \circ f) \\ {\scriptstyle G\ p \circ \theta}\Big| \qquad\qquad\qquad\qquad \Big|{\scriptstyle \rho \circ F\ p} \\ G\ f \circ \theta \xrightarrow[f_0]{} \rho \circ F\ f\end{aligned}$$

We will work this out in detail, assuming the category $\mathcal{D}$ is strict. This assumption is reasonable as per the reasons given in appendix B.2.

For p we can fill in $\mathsf{left\text{-}id}_{\mathcal{C}}\ f$. Furthermore we know that, assuming the functors F and G preserve the left identity laws:

$$F(\mathsf{left\text{-}id}_{\mathcal{C}}\ f) = F\text{-}\circ\ (\mathsf{id}_{\mathcal{C}}\ Y)\ f \cdot F\text{-}\mathsf{id}\ Y \circ Ff$$

and similarly for G. The diagram we end up with is the one shown in fig. B.1, where the double lines indicate a refl path. The non-trivial part of the diagram is the following square:

$$\begin{array}{ccc} G(\mathsf{id}_{\mathcal{C}}Y) \circ Gf \circ \theta & \xrightarrow{G\ (\mathsf{id}_{\mathcal{C}}\ Y)\ \circ\ f_0} & G(\mathsf{id}_{\mathcal{C}}Y) \circ \rho \circ Ff \\ {\scriptstyle G\text{-}\mathsf{id}\ Y\ \circ\ G\ f\ \circ\ \theta}\big\downarrow & & \big\downarrow{\scriptstyle G\text{-}\mathsf{id}\ Y\ \circ\ \rho\ \circ\ Ff} \\ Gf \circ \theta & \xrightarrow[f_0]{} & \rho \circ Ff \end{array}$$

Since we have $f_0 = \mathsf{id}_{\mathcal{D}}GY \circ f_0$, the above square follows from the naturality property enjoyed by homotopies.

As we see, even when we simplify the situation by assuming that $\mathcal{D}$ is strict, we end up with a rather involved calculation. Working out the associativity law in detail is even more strenuous. It seems that we can show that the category (F, G)-dialg has all the category laws that are preserved by the functors F and G.

B.4 Untruncated $\mathcal{T}ype$-sorted inductive-inductive definitions

In the previous section we have sketched how the category laws can be established in the untruncated dialgebra categories, assuming the codomain of the functors is a strict category. This means that it explains what we need from of a $\mathcal{T}ype$-sorted inductive-inductive definition to be able to show that the resulting category of algebras does in fact satisfy the category laws: all the arguments functors have to preserve the category laws.

While it seems that the coherence data is constant in the amount of constructors, in practice we see that this is not the case. To show that a functor

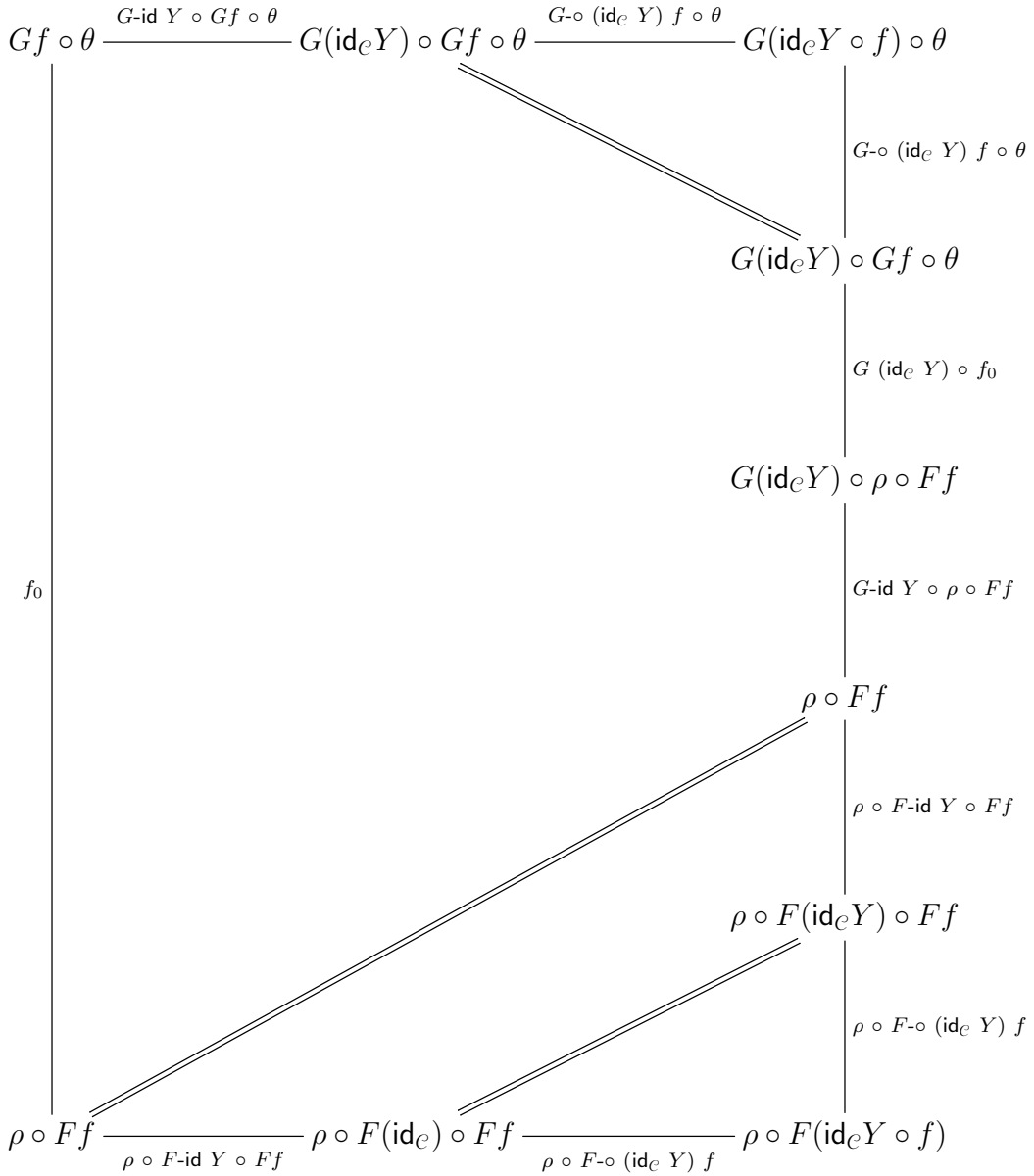


Figure B.1: Establishing the left identity law

preserves category laws, we may need more structure from the domain category. For generalised containers, we see that to show it preserves identity morphisms, we need to have identity laws in the domain category, i.e. we need coherence "one level up". To show that it preserves the identity laws, we need the higher coherence condition for identity laws from the domain category. This means that the amount of coherence levels needed stacks up every time we add a constructor to the definition.

What is somewhat surprising about this, is that we have not even considered path constructors yet. The coherence issues grow in the number of constructors no matter whether they are point or path constructors.

B.5 Path constructors and their computation rules

In the set truncated setting we could ignore the computation rules for path constructors, as all paths between equalities were trivial. In the untruncated setting they need to be accounted for. Suppose we have a category of algebras $\mathcal{C}$ with forgetful functor $U : \mathcal{C} \Rightarrow \mathcal{T}ype$. A path constructor defined on this category is given by a functor $F : \mathcal{C} \Rightarrow \mathcal{T}ype$ along with two natural transformations $l, r : F \dot{\to} U$. Let $X, Y : |\mathcal{C}|$, with $\theta : \ell_X = r_X$ and $\rho : \ell_Y = r_Y$. A morphism $f : X \to Y$ preserves the algebra structures θ and ρ, if "applying" the equality θ first and then the morphism yields the same equality as first applying the morphism and then the equality ρ. We would like to say:

$$Uf \circ \theta = \rho \circ Ff$$

but this does not type check as we have $Uf \circ \theta : Uf \circ \ell_X = Uf \circ r_X$ and $\rho \circ Ff : \ell_Y \circ Ff = r_Y \circ Ff$. We have to appeal to naturality in order for this equation to make sense. Let us denote for the witnesses of naturality as $\ell_f : Uf \circ \ell_X = \ell_Y \circ Ff$. For f to be an algebra morphism requires us to

have a witness of the following:

$$\begin{array}{ccc} Uf \circ \ell_X & \overline{}^{U\,f\,\circ\,\theta} & Uf \circ r_X \\ {\scriptstyle \ell_f}\Big| & & \Big|{\scriptstyle r_f} \\ \ell_Y \circ Ff & \overline{}_{\rho\,\circ\,Ff} & r_Y \circ Ff \end{array}$$

Bibliography

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 'Containers: constructing strictly positive types'. In: *Theoretical Computer Science* 342.1 (2005), pp. 3–27 (cit. on pp. 64, 173).

[AAL11] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. 'Definable quotients in type theory'. In: *Draft paper* (2011) (cit. on pp. 48, 49, 54, 60).

[Ace00] Fabio Acerbi. 'Plato: Parmenides 149a7-c3. a proof by complete induction?' In: *Archive for History of Exact Sciences* 55.1 (2000), pp. 57–76 (cit. on p. 1).

[ACK16a] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 'Extending Homotopy Type Theory with Strict Equality'. In: *CSL*. 2016 (cit. on pp. 12, 172, 179).

[ACK16b] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 'Higher Categories in Homotopy Type Theory'. In: *TYPES preproceedings*. 2016 (cit. on pp. 172, 179).

[Acz77] Peter Aczel. 'An introduction to inductive definitions'. In: *Studies in Logic and the Foundations of Mathematics* 90 (1977), pp. 739–782 (cit. on p. 5).

[AGS12] Steve Awodey, Nicola Gambino, and Kristina Sojakova. 'Inductive types in homotopy type theory'. In: *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. IEEE Computer Society. 2012, pp. 95–104 (cit. on pp. 12, 64).

[AK12] Thorsten Altenkirch and Nicolai Kraus. *Setoids are not an LCCC*. `http://www.cs.nott.ac.uk/~psznk/docs/setoids.pdf`. 2012 (cit. on p. 174).

[AK16] Thorsten Altenkirch and Ambrus Kaposi. 'Type theory in type theory using quotient inductive types'. In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 18–29 (cit. on pp. 12, 57).

[AK79] Ji Adámek and Václav Koubek. 'Least fixed point of a functor'. In: *Journal of Computer and System Sciences* 19.2 (1979), pp. 163–178 (cit. on p. 148).

[AKS15] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. 'Univalent categories and the Rezk completion'. In: *Mathematical Structures in Computer Science* 25.05 (2015), pp. 1010–1039 (cit. on p. 37).

[Alt+11] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. 'A categorical semantics for inductive-inductive definitions'. In: *CALCO 2011*. Ed. by Andrea Corradini, Bartek Klin, and Corina Cirstea. Vol. 6859. Lecture Notes in Computer Science. Springer, Heidelberg, 2011, pp. 70–84 (cit. on pp. 12, 60, 101).

[Alt+15] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. 'Towards a theory of higher inductive types'. In: *Presentation at TYPES* 15 (2015) (cit. on pp. 13, 15, 98, 103).

[Alt+16] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. 'Quotient inductive-inductive types'. In: *arXiv preprint arXiv:1612.02346* (2016) (cit. on p. 15).

[Bac+89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. 'Do-it-yourself type theory'. In: *Formal Aspects of Computing* 1.1 (1989), pp. 19–84 (cit. on pp. 6, 63).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: CoqArt: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2004 (cit. on p. 6).

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. 'A model of type theory in cubical sets'. In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Vol. 26. 2014, pp. 107–128 (cit. on p. 60).

[BDJ03] Marcin Benke, Peter Dybjer, and Patrik Jansson. 'Universes for generic programs and proofs in dependent type theory'. In: *Nord. J. Comput.* 10.4 (2003), pp. 265–289 (cit. on p. 11).

[BHW07] Frédéric Blanqui, Thérèse Hardin, and Pierre Weis. 'On the implementation of construction functions for non-free concrete data types'. In: *European Symposium on Programming*. Springer. 2007, pp. 95–109 (cit. on p. 60).

[Bru16] Guillaume Brunerie. 'On the homotopy groups of spheres in homotopy type theory'. PhD thesis. Université de Nice Sophia Antipolis, 2016 (cit. on p. 10).

[BW85] Michael Barr and Charles Wells. *Toposes, triples and theories*. Springer-Verlag New York, 1985 (cit. on p. 165).

[Cam13] Omar Antoln Camarena. *A whirlwind tour of the world of (, 1)-categories*. 2013 (cit. on pp. 12, 179).

[Cap14] Paolo Capriotti. *Mutual and Higher Inductive Types in Homotopy Type Theory*. 2014. URL: `http://cs.nott.ac.uk/~pvc/away-day-2014/mhit.pdf` (cit. on pp. 13, 103).

[Cav15] Evan Cavallo. 'Synthetic cohomology in homotopy type theory'. MA thesis. Carnegie Mellon University, 2015 (cit. on p. 10).

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. 'Pattern matching without K'. In: *International Conference on Functional Programming (ICFP 2014)*. 2014 (cit. on pp. 5, 22).

[Cha09] James Chapman. 'Type theory should eat itself'. In: *Electronic Notes in Theoretical Computer Science* 228 (2009), pp. 21–36 (cit. on p. 55).

[CJ95] Aurelio Carboni and Peter Johnstone. 'Connected limits, familial representability and Artin glueing'. In: *Mathematical Structures in Computer Science* 5.04 (1995), pp. 441–459 (cit. on p. 174).

[Coh+15] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 'Cubical type theory: a constructive interpretation of the univalence axiom'. In: *Preprint, December* (2015) (cit. on pp. 44, 60).

[Coq86] T Coquand. 'An analysis of Girard's paradox'. In: (1986) (cit. on p. 19).

[Coq92] Thierry Coquand. 'Pattern matching with dependent types'. In: *Informal proceedings of Logical Frameworks*. Vol. 92. 1992, pp. 66–79 (cit. on pp. 4, 22).

[Cra13] James Cranch. 'Concrete categories in homotopy type theory'. In: *arXiv preprint arXiv:1311.1852* (2013) (cit. on p. 178).

[Dan06] Nils Anders Danielsson. 'A formalisation of a dependently typed language as an inductive-recursive family'. In: *International Workshop on Types for Proofs and Programs*. Springer. 2006, pp. 93–109 (cit. on p. 55).

[Dia75] Radu Diaconescu. 'Axiom of choice and complementation'. In: *Proceedings of the American Mathematical Society* 51.1 (1975), pp. 176–178 (cit. on p. 53).

[Doo16] Floris van Doorn. 'Constructing the propositional truncation using non-recursive HITs'. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM. 2016, pp. 122–129 (cit. on pp. 50, 166).

[DS99] Peter Dybjer and Anton Setzer. 'A finite axiomatization of inductive-recursive definitions'. In: *Typed Lambda Calculi and Applications*. Springer, 1999, pp. 129–146 (cit. on p. 5).

[Dyb95] Peter Dybjer. 'Internal type theory'. In: *International Workshop on Types for Proofs and Programs*. Springer. 1995, pp. 120–134 (cit. on p. 143).

[Gir72] Jean-Yves Girard. 'Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur'. PhD thesis. PhD thesis, Université Paris VII, 1972 (cit. on p. 19).

[GJF10] Neil Ghani, Patricia Johann, and Clément Fumex. 'Fibrational induction rules for initial algebras'. In: *Computer Science Logic*. Springer. 2010, pp. 336–350 (cit. on p. 143).

[GK13] Nicola Gambino and Joachim Kock. 'Polynomial functors and polynomial monads'. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 154. 01. Cambridge Univ Press. 2013, pp. 153–192 (cit. on p. 70).

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. 'Eliminating dependent pattern matching'. In: *Algebra, Meaning, and Computation*. Springer, 2006, pp. 521–540 (cit. on pp. 5, 22).

[Hag87] Tatsuya Hagino. 'Category theoretic approach to data types'. PhD thesis. University of Edinburgh, 1987 (cit. on p. 72).

[Han+13] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. 'Small induction recursion'. In: *Typed Lambda Calculi and Applications*. Springer, 2013, pp. 156–172 (cit. on p. 101).

[Hof95] Martin Hofmann. 'Extensional concepts in intensional type theory'. PhD thesis. University of Edinburgh, 1995 (cit. on pp. 13, 46, 53, 60).

[Hou+16] Kuen-Bang Hou (Favonia), Eric Finster, Dan Licata, and Peter LeFanu Lumsdaine. 'A mechanization of the Blakers-Massey connectivity theorem in Homotopy Type Theory'. In: *arXiv preprint arXiv:1605.03227* (2016) (cit. on p. 10).

[HS98] Martin Hofmann and Thomas Streicher. 'The groupoid interpretation of type theory'. In: *Twenty-five years of constructive type theory (Venice, 1995)* 36 (1998), pp. 83–111 (cit. on p. 8).

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003 (cit. on p. 2).

[Kah01] Stefan Kahrs. 'Red-black trees with types'. In: *Journal of functional programming* 11.04 (2001), pp. 425–432 (cit. on p. 4).

[Kap16] Ambrus Kaposi. 'Type theory in a type theory with quotient inductive types'. PhD thesis. University of Nottingham, 2016 (cit. on p. 57).

[Kel80] G Max Kelly. 'A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on'. In: *Bulletin of the Australian Mathematical Society* 22.01 (1980), pp. 1–83 (cit. on p. 165).

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. 'The simplicial model of univalent foundations'. In: *arXiv preprint arXiv:1211.2851* (2012) (cit. on p. 9).

[Koc11] Joachim Kock. 'Polynomial functors and trees'. In: *International Mathematics Research Notices* 2011.3 (2011), pp. 609–673 (cit. on p. 174).

[Kra15] Nicolai Kraus. 'Truncation levels in homotopy type theory'. PhD thesis. University of Nottingham, 2015 (cit. on p. 50).

[KS15] Nicolai Kraus and Christian Sattler. 'Higher homotopies in a hierarchy of univalent universes'. In: *ACM Transactions on Computational Logic (TOCL)* 16.2 (2015), p. 18 (cit. on p. 9).

[LB13] Daniel R Licata and Guillaume Brunerie. 'π n (S n) in Homotopy Type Theory'. In: *International Conference on Certified Programs and Proofs*. Springer. 2013, pp. 1–16 (cit. on p. 10).

[Li15] Nuo Li. 'Quotient types in type theory'. PhD thesis. University of Nottingham, 2015 (cit. on pp. 13, 60).

[Lic11] Daniel R Licata. *Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute*. Apr. 2011. URL: `https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/` (cit. on p. 57).

[Lin69] Fred EJ Linton. 'Coequalizers in categories of algebras'. In: *Seminar on triples and categorical homology theory*. Springer. 1969, pp. 75–90 (cit. on p. 165).

[LS13a] Daniel R Licata and Michael Shulman. 'Calculating the fundamental group of the circle in homotopy type theory'. In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Computer Society. 2013, pp. 223–232 (cit. on p. 10).

[LS13b] Peter LeFanu Lumsdaine and Michael Shulman. *Semantics of higher inductive types*. https://uf-ias-2012.wikispaces.com/file/detail/semantics.pdf. 2013 (cit. on pp. 10, 13, 60, 70, 103, 165).

[Lum09] Peter LeFanu Lumsdaine. 'Weak ω-categories from intensional type theory'. In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2009, pp. 172–187 (cit. on pp. 9, 25).

[Mak95] Michael Makkai. 'First order logic with dependent sorts, with applications to category theory'. `http://www.math.mcgill.ca/makkai/folds/`. 1995 (cit. on p. 79).

[Mal+12] Lorenzo Malatesta, Thorsten Altenkirch, Neil Ghani, Peter Hancock, and Conor McBride. 'Small induction recursion, indexed containers and dependent polynomials are equivalent'. In: *Submitted for publication* 35 (2012) (cit. on p. 74).

[Mar71] Per Martin-Löf. 'Hauptsatz for the intuitionistic theory of iterated inductive definitions'. In: *Studies in Logic and the Foundations of Mathematics* 63 (1971), pp. 179–216 (cit. on p. 5).

[Mar72] Per Martin-Löf. *An intuitionistic theory of types*. 1972 (cit. on p. 5).

[McB06] Conor McBride. *On Berry's Majority Function*. June 2006. URL: `https://www.mail-archive.com/epigram@durham.ac.uk/msg00229.html` (cit. on p. 22).

[Mor07] Peter Morris. 'Constructing universes for generic programming'. In: *PhD thesis, The University of Nottingham* (2007) (cit. on pp. 22, 173).

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007 (cit. on p. 4).

[Nor13] Fredrik Nordvall Forsberg. 'Inductive-inductive definitions'. PhD thesis. Swansea University, 2013 (cit. on pp. 1, 5, 12, 55, 143).

[Pea89] Guiseppe Peano. 'Arithmetices Principia nova methodo exposita, Aug'. In: *Taurinorum (D Opere scelte, a cura dellUnione Matematica Italiana 2: 20–55)* (1889) (cit. on p. 1).

[PL04] Emir Paali and Nathan Linger. 'Meta-programming with typed object-language representations'. In: *International Conference on Generative Programming and Component Engineering*. Springer. 2004, pp. 136–167 (cit. on p. 4).

[Rij12] Egbert Rijke. *A type theoretical Yoneda lemma*. May 2012. URL: https://homotopytypetheory.org/2012/05/02/a-type-theoretical-yoneda-lemma/ (cit. on p. 36).

[Shu11a] Michael Shulman. *An Interval Type Implies Function Extensionality*. Apr. 2011. URL: https://homotopytypetheory.org/2011/04/04/an-interval-type-implies-function-extensionality/ (cit. on p. 46).

[Shu11b] Michael Shulman. *Re: Homotopy Type Theory, VI*. Apr. 2011. URL: https://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html#c041358 (cit. on pp. 7, 70).

[Soj14] K. Sojakova. 'Higher Inductive Types as Homotopy-Initial Algebras'. In: *ArXiv e-prints* (Feb. 2014) (cit. on p. 11).

[Tho86] Simon Thompson. 'Laws in miranda'. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM. 1986, pp. 1–12 (cit. on p. 60).

[Tho90] Simon Thompson. 'Lawful functions and program verification in Miranda'. In: *Science of Computer Programming* 13.2 (1990), pp. 181–218 (cit. on p. 60).

[Tur85] David A Turner. 'Miranda: A non-strict functional language with polymorphic types'. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 1–16 (cit. on p. 60).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`, 2013 (cit. on pp. 5, 7, 12, 17, 30, 33, 55).

[VG11] Benno Van Den Berg and Richard Garner. 'Types are weak ω-groupoids'. In: *Proceedings of the London Mathematical Society* 102.2 (2011), pp. 370–394 (cit. on pp. 9, 25).