Higher Inductive Types, Inductive Families, and Inductive-Inductive Types

Jakob von Raumer

September 30, 2019

Abstract

Martin-Löf type theory is a formal language which is used both as a foundation for mathematics and the theoretical basis of a range of functional programming languages. Inductive types are an important part of type theory which is necessary to express data types by giving a list of rules stating how to form this data. In this thesis we we tackle several questions about different classes of inductive types.

In the setting of homotopy type theory, we will take a look at higher inductive types based on homotopy coequalizers and characterize their path spaces with a recursive rule which looks like an induction principle. This encapsulates a proof technique known as "encode-decode method".

In an extensional meta-theory we will then explore the phenomenon of induction-induction, specify inductice families and discuss how we can reduce each instance of an inductive-inductive type to an inductive family. Our result suggests a way to show that each type theory which encompasses inductive families can also express all inductive-inductive types.

Acknowledgements

There are several people inside and outside university without whom my pursuit of a PhD as well as this thesis would have been impossible. I am grateful for everyone who supported and encouraged me during the past four years.

First of all, I would like to thank my supervisor Thorsten Altenkirch for giving me the maximal amount of freedom in the choice of the direction of my research, for being patient when progress was sluggish, for his good advice, and for being understanding when I had to interrupt my studies.

I am grateful for the support both professional and emotional that I received from the other members of the Functional Programming Lab at the University of Nottingham, especially Nicolai Kraus and Paolo Capriotti. The Lab has always been a place for good collaboration and broadening my horizon.

Furthermore, I want to thank Ambrus Kaposi for giving me the opportunity to work with him in Budapest and to help me get unstuck with the constructions on inductive-inductive types.

Contents

Chapter 1

Introduction

1.1 Background

This thesis explores several problems in the field of *type theory*. By type theory we will always mean various flavors of what is usually referred to as Martin-Löf type theory or dependent type theory. Martin-Löf type theory (MLTT) can serve as a foundational framework for mathematics as well as an organization principle for functional programming languages [Martin-Löf and Sambin, 1984].

In the field of type theory, many researchers either apply theoretical considerations to achieve cleaner formalizations of mathematical content, they create implementations of type theory which can be used as interactive theorem proving systems, and they try to extend type theory to improve its usability and justify these extensions with models.

In this spirit, this thesis will also explore ways to make certain kinds of formalizations and certain constructions in type theory smoother and easier to use. While it is *mainly theoretical work*, it has consequences for the application of interactive theorem provers, as they are used today. This thesis is broadly split in two halves. Both halfs will explore different classes of language elements both of which are generalizations of a language feature which is called *inductive types*. Inductive types are a common way to define collections of data in mathematics as well as computer science.

In the first half, we will encapsulate a common proof strategy which is often used in the field of homotopy type theory and especially in synthetic homotopy theory [Univalent Foundations Program, 2013]. We will do this by proving a very general result about *higher inductive types*. These are inductive types in which statements about *equality* between elements of these data types carry a *higher-dimensional structure*, making them on the one hand an interesting object of study in terms of their topology, but on the other hand they are sometimes hard to handle. Our theorem allows an easier way to prove propositions about these equalities of elements.

In the second half, we will explore the topic of induction-induction. *Inductive-inductive types* [Nordvall Forsberg, 2013] are a class of inductive types which allows us to define a data type simultaneously with data types depending on values of the former type. We will give an exact definition of what inductive-inductive types are, and, with a new definition of *inductive families* provide a point of comparison which allows us to represent each example of an inductive-inductive type as a construction based on a series of inductive families.

1.2 Homotopy Type Theory and Higher Inductive Types

Homotopy type theory is a relatively new field which connects the study of dependent types with the field of higher category theory and homotopy theory. This synthesis has since offered a new perspective on how to constructively have a formal representation of homotopy theory which is *synthetic*, i. e. builds the spaces which are considered out of just a few fundamental operations.

The connection to homotopy theory is based on the observation that one might consider equality types, which by the proposition-as-types interpretation of type theory, represent the statement that two elements x and y of a type A are equal, as representing the spaces of *paths modulo homotopy*. Extending on this equivalence, we can view types to represent spaces, and types depending on the data of other types as fibrations.

In this setting, we want to consider types, which are inductively defined, like for example the natural numbers, but which, besides the *points* of the type also allow the (free) generation of new paths between points and "higher" paths between other paths. These types are called *higher inductive types*.

To prove facts about higher inductive types, for example in order to get the type theoretic equivalent of the fact that the fundamental group of the circle is equivalent to the integers, or the type theoretic Seifert-van Kampen theorem, which characterizes the fundamental groupoid of a pushout of spaces, a proof strategy with the name "encode-decode method" is employed.

In this thesis we provide a theorem which can be seen a generalization of encode-decode proofs. The application of this theorem can help to reduce "boiler plate" overhead in formalizations and in reasoning about the equalities between points of higher inductive types.

1.3 The Concept of Induction-Induction

While the first half of this thesis is about ways to make inductive types carry higher-dimensional structure, the second half is about allowing for inductive types which are more *interdependent*: There are situations in which we might not only want to define one single type or type family, but instead we want to define a type A and a type family $B : A \to \mathcal{U}$, or even a whole system of new types, indexed over each other, mutually. "Mutually" here means that the point constructors, which specify which elements we can form can refer to any of the types being defined. And more than that, also the signature of the type families which we define can be indexed over other type families the definition of which is not finished.

To illustrate one main application, imagine we wanted to formalize and reason about the syntax of type theory in a type theoretic setting. The environment of variables which are at our disposal at a given point in a piece of type theoretic syntax are captured in what is called a *context*. We can model contexts as a type $Con : \mathcal{U}$ ($\mathcal{U}$ being the universe of all types). But at the same time we want to model types as existing in a context, so we want a simultaneous definition of a type family $Ty : Con \to \mathcal{U}$. To see that we can not first define Con and then move on to define Ty, consider the following data which Con and Ty should include:

Contexts can be seen as lists of types depending on previous entries in that list. In that sense, it is obvious that for a context $\Gamma : Con$ and a type $A : Ty(\Gamma)$ in that context, we want a context which represents the extension of Γ by A. This means that Con should have a constructor of the form

$$ext : (\Gamma : Con) \to Ty(\Gamma) \to Con,$$

and the fact that this constructor mentions Ty is already enough to exclude a sequential definition of Con and Ty.

Given the need for inductive-inductive types we can ask the question of whether this concept is really stronger than inductive families without dependencies between the sorts. At first glance it might seem as if they are more expressive, but on a closer look we can discover that we can actually reduce every example of an inductive-inductive type to an inductive family.

The second half of this thesis is about making this reduction, which can easily be seen to work on specific examples, more general. To make it a formal statement we will have to give precise definitions of what inductive-inductive types are and what, as our reference point, inductive families are. While we don't succeed at providing a full formal proof for the reduction, the essential steps of it are complete and formalized in Agda.

1.4 Contributions and Publications

While parts of this thesis consists of the review and introduction of constructions and knowledge which is already established, other parts offer new contributions which stand on the shoulders of these "giants". The contributions of this thesis include the following:

- The formalization of lots of homotopy theoretic notions in the theorem prover Lean as described in Section 2.4.1.
- The formulation of the characterization theorem for path spaces of homotopy Coequalizers Theorem 4.0.2, as well as its proof as given in Section 4.1.
- The adaptation of this theorem for pushouts as described in Theorem 4.2.1.
- The formulation of a possible higher Seifert-van Kampen theorem as stated in Theorem 4.4.1.
- The formalization of Section 4.1 and Theorem 4.2.1 in Lean.
- An adaptation of the syntax for higher inductive-inductive types by Kaposi and Kovács [2018a] to separate sort and point constructors, as described, together with its semantics, in Chapter 5.
- A syntax of signatures for inductive families as given in Chapter 6.

- A formal specification of type erasure, wellformedness relation and eliminator relation given as syntactic translations as described in Section 7.2, Section 7.3, and Section 7.5.
- A formal definition of the "sigma construction" for an initial algebra for inductive-inductive types as proposed in Section 7.4.

Parts of this thesis have been peer-reviewed and published already, while other parts, especially Chapter 6 and Chapter 7 are not yet published elsewhere.

- Together with Floris van Doorn and Ulrik Buchholtz, a more detailed description of our homotopy type theory formalizations was given in the proceedings of the conference *Interactive Theorem Proving – 8th International Conference* in 2017 [van Doorn et al., 2017].
- In joint work with Nicolai Kraus, the characterization of path spaces was published in the proceeding of the conference *Thirty-Fourth Annual ACM/IEEE Symposium on Logic in Computer Science* (*LICS*) in 2019 [Kraus and von Raumer, 2019].

1.5 Structure of this Thesis

We will start off this thesis by giving a more detailed exposition of the background in type theory, which is the basis for the further content. In this endeavour, Chapter 2 does not only serve to give the necessary background to readers unfamiliar with dependent types, it also sets notations and terminology which we will re-use in the later chapters. In Section 2.4, it furthermore gives a short characterization of the interactive theorem provers Lean and Agda.

Chapter 3 will then first introduce some examples of higher inductive types and will then go on to propose homotopy coequalizers as a fundamental higher inductive type which can serve as a common generalization of all of these examples. Sketches for two proofs using the "encode-decode" method will be given in Section 3.3.

Following the introduction of homotopy coequalizers we will then (Chapter 4) see how we can characterize their path spaces in order to replace encode-decode proofs. Apart from proving this characterization (Section 4.1) we will demonstrate this use in some examples (Section 4.3 and Section 4.4).

Switching not only the type theoretical setting from homotopy type theory to set-truncated (or even extensional) type theory but also the focus of the thesis, the second half will explore types which instead of higher dimensional structure carry intricate dependencies between multiple types to be defined.

We will start this second half by first giving a syntax for inductive-inductive types (Chapter 5) including its semantics, before comparing it to the simpler fragment of inductive families (Chapter 6). The latter will be reduced to indexed W-types in Section 6.4.

After reducing inductive families to indexed W-types we will try to reduce inductive-inductive types to inductive families: In Chapter 7 we will give a formal description about how to generate the inductive families which correspond to erasing the inductive-inductive typing information, recovering it with a wellformedness predicate, yielding a candidate for an inital object in the target type theory. Then we will present a binary relation which could be used in the future to prove its initiality.

Chapter 2

Basic Type Theory

This chapter shall serve to introduce the basic notions of type theory which we will need for the subsequent content of this thesis. At first (Chapter 2.1), we will have a general look on dependent type theory, its use and how it differs from a set theoretic foundation, afterwards (Chapter 2.2) we want to unify the notion of an inductive type using the concept of indexed W-types. Then (Chapter 2.3), we will explain how homotopy type theory was created to have a suitable language to reason about higher equalities and how it provides a synthetic way to formalize topological insights. Finally, in Chapter 2.4, we will give examples of two theorem provers based on dependent types, Agda and Lean, and point out some of their differences.

2.1 Dependent Type Theory

The term "type theory" stems from the early nineteenth centry, when Bertrand Russell sought to lay out an alternative form of set theory which did not suffer from the paradox which Russell discovered. Todays versions of type theory have little in common with Russell's attempts but rather rely on the considerations of Per Martin-Löf (Martin-Löf and Sambin [1984], Martin-Löf [1998]) who, starting in the 1970's, built a new mathematical foundation based on the λ-calculus, himself drawing inspiration from previous logicians and mathematicians like Alonzo Church and Haskell Curry. Often, dependent type theory is also referred to as Martin-Löf type theory (MLTT).

Based on a phenomenon known as the "Curry-Howard correspondence", type theory can serve both as a theoretical foundation for a formal representation of mathematics, as well as a principle for the specification for strongly typed functional programming languages. It was implemented in computer languages for programming and theorem proving which are massively used in the field of formal verification and in the formalization of mathematics. Among the most commonly used implementations are the theorem prover Coq (Barras et al. [1997]) which notably has a lot of users in the field of hardware verification, the prover Agda (Norell [2009]), which is popular amongst type theorists themselves, and the Microsoft Research based project Lean(de Moura et al. [2015]), which has drawn considerable attention from researching mathematicians as a tool to formally verify their proofs.

Type theory differs from a set theoretic mathematical foundation (let us, as a point of reference, consider set theories based on first-order predicate logic like Zermelo-Fraenkel set theory) in several important aspects:

- Type theory follows a paradigm called *"propositions-as-types"*. This means that statements like theorems and conjectures are represented using the same class of objects as other data like sets or (algebraic) structures. In contrast to this, most set theoretic foundations are built on a *dichotomy between the propositions and the objects* they describe: They first start out with a logical framework on which axiomatically a theory of sets is introduced. The coherence between these two levels must then be created using an axiom like the comprehension axiom in Zermelo-Fraenkel set theory.

- Type theory is *typed* while set theory is *untyped*. While in set theory, objects can be an element of different sets – consider the number two which is an element both of the set of even integers as well as the set of all integers – type theory is based on the principle that every piece of data (every term) is assigned a unique type which is known at the point of the creation of the data. This assignment, called typing, is decidable, and we consider it a judgment rather than a provable proposition that a given term t has type A.

- Type theory is inherently *constructive* while many set theoretic foundations, such as Zermelo-Fraenkel, are *non-constructive*. This has great consequences for the computational use of the mathematics represented: Every type theoretic function the codomain of which are the natural numbers, can compute a numeral for any given input.

- While in set theory, *sets are the only primitives* and all data is encoded as sets, type theory provides often-used objects like *functions and inductively defined types as primitives*. This means that we can use these without having to care about how to assemble them from other primitives.

Let us now fix some notation and some basic constructions which will occur throughout this text. As mentioned, we will, whenever we talk about a certain piece of type theoretic data, accompany it with its type, we need a notation for this kind of **type judgment**: We write $t : A$ to state that the term t is of type A. This is similar to the element relation of a set, but it also represents the fact that t is a proof for a proposition A. Sometimes, we want to express that two terms s and t of the same type A only differ by an unfolding or folding of a definition, or that the application of a reduction rule to s results in t. We will notate this by $s \equiv t$. Our type theory will then make no distinction between s and t. This means that if A and B are types with $A \equiv B$, then $s : A$ implies $s : B$. To keep type checking decidable it is easy to see that it is important to keep the question whether two terms are what we call **definitionally equal** decidable itself.

We said above that each piece of data, so each term, has a unique type (up to definitional equality). This statement also holds true for types itself. Types which themselves contain types, are called **universes** and we will denote them using $\mathcal{U}$. The universe itself also needs a type, but assuming $\mathcal{U} : \mathcal{U}$ is inconsistent. This is often referred to as "Girard's paradox" [Girard, 1972, Hurkens, 1995] which can be seen as the type theoretic equivalent to Russell's paradox. The solution we assume for the remainder of this text is to assume that we have an infinite chain of universes

$$\mathcal{U}_0 : \mathcal{U}_1,\ \mathcal{U}_1 : \mathcal{U}_2,\ \mathcal{U}_2 : \mathcal{U}_3,\ \ \ldots,$$

each contained in the next one. Most often, we will chose to leave the index implicit and regard our constructions as being *universe polymorphic*, meaning that they are valid in any chosen universe. Some type theories are constructed to be *cumulative* in the sense that whenever we have a type $A : \mathcal{U}_i$, it is a type in the succeeding universe, so $A : \mathcal{U}_{i+1}$. Our constructions will not rely on cumulativity and they are formalized in non-cumulative type theories.

Until now, we have not talked about any way to form types. In the following, we will get to know the non-dependent version of some basic types, some of which will later be generalized to a dependent form. These type formers will be presented in a very similar way: We will give a rule on how to form the type itself (called formation rule), rules on how to construct elements of the type (called introduction rules), rules on how to use the elements of the type (called elimination rules), and some of them will be followed by some reduction rules offering definitional equalities used to simplify terms. As we have mentioned in the comparison to set theory, functions are basic building blocks in type theory. The non-dependent functions form what is known as *simply typed λ-calculus*. The formation, introduction, and elimination rules correspond to the fact that we can form the function type of any type for its domain and any type for its codomain, the fact that we can build a function by specifying its output for any given input, and the fact that we can apply a function to any term of its domain:

$$\to\text{-}\textsc{Form}\ \frac{A, B : \mathcal{U}}{A \to B : \mathcal{U}} \qquad \to\text{-}\textsc{Intro}\ \frac{a : A \vdash \Phi[a/x] : B}{(\lambda x.\Phi) : A \to B}$$

$$\to\text{-}\textsc{Elim}\ \frac{f : A \to B \qquad a : A}{f(a) : B}$$

Here, Φ is a term that may have x as a free variable. $\Phi[a/x]$ denotes the replacement of every free appearance of x by a. Additional to these rules we also have η-conversion and β-reduction:

$$\eta\ \frac{f : A \to B}{(\lambda x.f(x)) \equiv f} \qquad \beta\ \frac{(\lambda x.\Phi) : A \to B \qquad a : A}{(\lambda x.\Phi)(a) \equiv \Phi[a/x]}$$

In type theory, these functions are used to represent both functions between sets as well as implications between propositions. A special kind of function are those of the form $B : A \to \mathcal{U}$. These so called **type families** are used to represent set-valued functions as well as propositions with a free variable in A.

Remark 2.1.1. When giving inference rules as the ones above, we always assume an arbitrary *context* of variables for their premises as well as their conclusion. Often, when presenting the syntax of a type theory, rules on how to form these contexts $\vdash \Gamma$ are given together with rules for the formation of types $\Gamma \vdash A$ in a given context Γ and the treatment of terms $\Gamma \vdash t : A$ of a given type A in context Γ. We will choose to be implicit about the variable context in this chapter, while being more explicit about them when we introduce other syntaxes in later chapter, for which a more formal treatment is appropriate.

Since we want a representation of logics in type theory, and so far the only logical connective we introduced are implications, we might next ask for types representing the propositions “true” and “false”. Since the elements of this type should correspond to the proofs of “true” and “false”, we can conclude that the type corresponding to “true” should contain one canonical element, while the one corresponding to “false” should contain none. Thus, we will call these two types the **empty type** and the **unit type**, respectively and denote them with **0** and **1**. The formation and introduction rules for these are not very surprising:

$$\textbf{0}\text{-}\textsc{Form}\ \frac{}{\mathbf{0} : \mathcal{U}} \qquad \textbf{1}\text{-}\textsc{Form}\ \frac{}{\mathbf{1} : \mathcal{U}} \qquad \textbf{1}\text{-}\textsc{Intro}\ \frac{}{\star : \mathbf{1}}$$

The elimination rule for **0** says the we can derive a proof for any statement, given a proof of “false” (“ex falso quodlibet”), while the elimination rule for **1** specifies that when showing a statement about the elements of the unit type, it suffices to consider its canonical element $\star$:

$$\textbf{0}\text{-}\textsc{Elim}\ \frac{C : \mathbf{0} \to \mathcal{U} \qquad x : \mathbf{0}}{\mathsf{elim}_{\mathbf{0}}(C, x) : C(x)} \qquad \textbf{1}\text{-}\textsc{Elim}\ \frac{C : \mathbf{1} \to \mathcal{U} \qquad p : C(\star) \qquad x : \mathbf{1}}{\mathsf{elim}_{\mathbf{1}}(C, p, x) : C(x)}$$

with the reduction rule $\mathsf{elim}_{\mathbf{1}}(C, p, \star) \equiv p$. Some type theories assume so called η-rules for types like **1**, which say that for every $x : \mathbf{1}$, we have $x \equiv \star$. We will *not* assume these rules to hold, but instead we will later be able to express the fact that all instances of **1** are equal to $\star$ using propositional equality.

To give an example for a type which contains more than just one element, we can consider the type of booleans **2**, containing elements $0_{\mathbf{2}}$ and $1_{\mathbf{2}}$. This time, we have more than just one constructor, and the elimination rule requires us to give proofs for both of the two elements:

$$\textbf{2}\text{-}\textsc{Form}\ \frac{}{\mathbf{2} : \mathcal{U}} \qquad \textbf{2}\text{-}\textsc{Intro1}\ \frac{}{0_{\mathbf{2}} : \mathbf{2}} \qquad \textbf{2}\text{-}\textsc{Intro2}\ \frac{}{1_{\mathbf{2}} : \mathbf{2}}$$

$$\textbf{2}\text{-}\textsc{Elim}\ \frac{C : \mathbf{2} \to \mathcal{U} \qquad p_0 : C(0_{\mathbf{2}}) \qquad p_1 : C(1_{\mathbf{2}}) \qquad x : \mathbf{2}}{\mathsf{elim}_{\mathbf{2}}(C, p_0, p_1, x) : C(x)}$$

with the reduction rules $\mathsf{elim}_{\mathbf{2}}(C, p_0, p_1, 0_{\mathbf{2}}) \equiv p_0$ and $\mathsf{elim}_{\mathbf{2}}(C, p_0, p_1, 1_{\mathbf{2}}) \equiv p_1$.

As a minimal example of an infinite type, the set theoretic equivalent of which would be introduced axiomatically, we can take a look at the type of natural numbers. It is generated inductively by the zero element and the successor function. The eliminator provides the principle of induction on the natural numbers, which also makes sure that, when proving properties about the natural numbers, we can assume that every element of the natural numbers is equal to a repeated applying the successor function to zero:

$$\textsc{$\mathbb{N}$-Form}\ \frac{}{\mathbb{N} : \mathcal{U}} \qquad \textsc{$\mathbb{N}$-Intro}\ \frac{}{0 : \mathcal{U} \qquad S : \mathbb{N} \to \mathbb{N}}$$

$$\textsc{$\mathbb{N}$-Elim}\ \frac{\begin{array}{c} C : \mathbb{N} \to \mathcal{U} \\ p : C(0) \qquad q_n : C(n) \to C(S(n)) \text{ for all } n : \mathbb{N} \qquad x : \mathbb{N} \end{array}}{\mathsf{elim}_{\mathbb{N}}(C, p, q, x) : C(x)}$$

with the reduction rules

$$\begin{aligned} \mathsf{elim}_{\mathbb{N}}(C, p, q, 0) &\equiv p \text{ and} \\ \mathsf{elim}_{\mathbb{N}}(C, p, q, S(x)) &\equiv q(x, \mathsf{ind}_{\mathbb{N}}(C, p, q, x)). \end{aligned}$$

Note that we regain, by restricting elim_N to constant type families, the *non-dependent eliminator* or *recursor*, corresponding to the usual way to recursively define functions $f : \mathbb{N} \to A$ for a type $A : \mathcal{U}$ by providing the value $f(0)$ and, for each $n : \mathbb{N}$ the recursive definition $f(n) \to f(S(n))$. It has the following type:

$$\mathsf{rec}_{\mathbb{N}}(A) \equiv \mathsf{ind}(\lambda x.A) : A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A.$$

So far we did not do the word "dependent" in the name of the type theory much justice, but now we will move on to introduce some dependent type formers. The first of these will be **Π-types** or dependent function types. When considering non-dependent functions, the codomain was a fixed type B such that for all inputs $a : A$, the output $f(a)$ is an element of B. For dependent functions, however, the codomain is a type family $B : A \to \mathcal{U}$, and for each input $a : A$ the output $f(a)$ is of type $B(a)$. Considering that we want to use type families to represent propositions depending on a free variable, these functions represent the universal quantification over this free variable! While for application and λ-abstraction we don't introduce new notation for dependent functions, we will denote the type of all dependent functions on a type family $B : A \to \mathcal{U}$ as $\prod_{(a:A)} B(a)$, or, alternatively, as $\Pi(B)$ (as a shorter variant) or $(a : A) \to B(a)$ (often called

"Agda-notation" in referral to the theorem prover of that name). The rules to form the type of Π-types and to introduce and apply dependent functions generalize the rules for non-dependent functions as follows:

$$\Pi\text{-}\textsc{Form}\ \frac{A : \mathcal{U}_i \qquad B : A \to \mathcal{U}_j}{(\prod_{(a:A)} B(a)) : \mathcal{U}_{\max\{i,j\}}} \qquad \Pi\text{-}\textsc{Intro}\ \frac{a : A \vdash \Phi[a/x] : B(a)}{(\lambda x.\Phi) : \prod_{(a:A)} B(a)}$$

$$\Pi\text{-}\textsc{Elim}\ \frac{f : \prod_{(a:A)} B(a) \qquad a : A}{f(a) : B(a)}$$

Again, we have the rules for β-reduction and η-conversion like in the non-dependent case, yielding reduction rules in the form of judgmental equalities $(\lambda x.f(x)) \equiv f$ and $(\lambda x.\Phi)(a) \equiv \Phi[a/x]$. Having Π-types at our disposal allows us to state the rules governing a couple of further essential type formers. Note that, once we have added dependent functions, we can rediscover non-dependent functions as the special case of dependent functions over a constant type family. When iterating Π-types, we will often find that the argument of a dependent function is already determined by an earlier argument, as in $f : (a : A)(b : B(a)) \to C(a,b)$. In this case, we borrow the notation used by many theorem provers and use curly brackets to denote arguments which will be left implicit: If $f : \{a : A\}(b : B(a)) \to C(a,b)$ and $b : B(a)$, we write $f(b) : C(a,b)$. If we later want to state these explicitly we will re-use curly brackets to denote that we reintroduce them: $f\{a\}(b) : C(a,b)$.

Two important logical connectives are still missing: Conjunction and disjunction. In type theory these coincide with the product and disjoint union (sum) of types. The rules for the product type are as follows:

$$\times\text{-}\textsc{Form}\ \frac{A, B : \mathcal{U}}{A \times B : \mathcal{U}} \qquad \times\text{-}\textsc{Intro}\ \frac{a : A \qquad b : B}{(a,b) : A \times B}$$

$$\times\text{-}\textsc{Elim}\ \frac{C : A \times B \to \mathcal{U} \qquad p : (a : A)(b : B) \to C(a,b) \qquad x : A \times B}{\mathsf{elim}_{A\times B}(C, p, x) : C(x)}$$

with the reduction rule $\mathsf{elim}_{A\times B}(C, p, (a,b)) \equiv p(a,b)$. The projections of an instance $x : A \times B$ are then defined by induction:

$$\mathsf{pr}_1(x) :\equiv \mathsf{elim}_{A\times B}((\lambda y.A), (\lambda a.\lambda b.a), x) : A \text{ and}$$
$$\mathsf{pr}_2(x) :\equiv \mathsf{elim}_{A\times B}((\lambda y.A), (\lambda a.\lambda b.b), x) : B,$$

yielding $\mathsf{pr}_1((a,b)) \equiv a$ and $\mathsf{pr}_2((a,b)) \equiv b$ judgmentally. The type representing disjunction has two constructors determining whether we provide proof for its left or its right type:

$$+\text{-}\textsc{Form}\ \frac{A : \mathcal{U} \qquad B : \mathcal{U}}{A + B : \mathcal{U}}$$

$$+\text{-}\textsc{Intro1}\ \frac{a : A}{\mathsf{inl}(a) : A + B} \qquad +\text{-}\textsc{Intro2}\ \frac{b : B}{\mathsf{inr}(b) : A + B}$$

$$+\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : (A + B) \to \mathcal{U} \\ p : (a : A) \to C(\mathsf{inl}(a)) \qquad q : (b : B) \to C(\mathsf{inr}(b)) \qquad x : A + B \end{array}}{\mathsf{elim}_{A+B}(C, p, q, x) : C(x)}$$

For the sum type, we have the reduction rules

$$\begin{aligned} \mathsf{elim}_{A+B}(C, p, q, \mathsf{inl}(a)) &\equiv p(a) \text{ and} \\ \mathsf{elim}_{A+B}(C, p, q, \mathsf{inr}(b)) &\equiv q(b). \end{aligned}$$

Looking at the product type, we can find a generalization which is very useful when we want to model existential quantification of a type family $B : A \to \mathcal{U}$ in type theory: A version of the product type where the type of the second component of a pair (a, b) may depend on $a : A$ via the type family, i. e. $b : B(a)$. The type holding this kind of pair for a fixed type family $B : A \to \mathcal{U}$ is called the **Σ-type** over B. We will again have three different notations for this type, of which in this text we will mostly prefer the latter one: The type is usually [Homotopy Type Theory, 2013] denoted by $\sum_{(a:A)} B(a)$, mimicing mathematical notation for sums. Furthermore there is the short variant of writing $\Sigma(B)$ and an Agda-inspired notation $(a : A) \times B(a)$. The inference rules for this type are just a slight generalization of the rules we have already seen for the non-dependent product type:

$$\Sigma\text{-}\textsc{Form}\ \frac{A : \mathcal{U} \qquad B : A \to \mathcal{U}}{(a : A) \times B(a) : \mathcal{U}} \qquad \Sigma\text{-}\textsc{Intro}\ \frac{a : A \qquad b : B(a)}{(a, b) : (a : A) \times B(a)}$$

$$\Sigma\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : (a : A) \times B(a) \to \mathcal{U} \\ p : (a : A)(b : B(a)) \to C((a, b)) \qquad x : (a : A) \times B(a) \end{array}}{\mathsf{elim}_{(a:A) \times B(a)}(C, p, x) : C(x)}$$

As our notation already suggests, we can view the product $A \times B$ as a special case of a Σ-type where B is a constant type family. Note that also the sum type $A + B$ can be defined as a special case of the sigma type over $C : \mathbf{2} \to \mathcal{U}$ with $C(0_\mathbf{2}) :\equiv A$ and $C(1_\mathbf{2}) :\equiv B$. This is why Σ-types are sometimes also referred to as *dependent sum types*. By induction we can define the **projections**

$$\begin{aligned}&\mathsf{pr}_1 : ((a : A) \times B(a)) \to A \text{ and}\\&\mathsf{pr}_2 : (x : (a : A) \times B(a)) \to B(\mathsf{pr}_1(x)).\end{aligned}$$

which return the components of the dependent pairs. When we iterate Σ-types an products we will be liberal with the notation and allow notation like $\mathsf{pr}_3, \ldots$ as well.

With the type formers we met so far, we can already represent a lot of mathematical definitions and knowledge. For example, if we wanted to define what it means for a natural number to be odd, we could set

$$\mathsf{isodd} :\equiv \mathsf{rec}_\mathbb{N}(\mathcal{U}, \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0})) : \mathbb{N} \to \mathcal{U}.$$

For example, this gives us the statement that the number one is odd, witnessed by the following term:

$$\begin{aligned}(\lambda x.x) :\ & \mathbf{0} \to \mathbf{0}\\&\equiv \mathsf{elim}_\mathbb{N}((\lambda x.\mathcal{U}), \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0}), 0) \to \mathbf{0}\\&\equiv \mathsf{elim}_\mathbb{N}((\lambda x.\mathcal{U}), \mathbf{0}, (\lambda n.\lambda A.A \to \mathbf{0}), S(0))\\&\equiv \mathsf{isodd}(S(0)).\end{aligned}$$

2.2 Inductive Types

So far, the presented type formers may seem like a zoo of unrelated, random examples. Some are generalization of others (like Σ-types generalize sums), but the question one might ask is if there is any overarching principle behind the choice of those type formers. The feature that all of them have in common is that, rather than by an enumeration of their elements, they are defined by their formation, introduction, and elimination rules – their elements are those which are generated *inductively* by their introduction rules, also called *constructors*.

Some of the type formers we have seen were *parameterized* by other types, but more than that, dependent types allow us to specify *indexed* inductive types. One example for this is the type of *vectors* of a type A. Vectors are a variant of lists, where we use the typing to keep track of the length of its elements: The type of vectors on A is not a type but a type family:

$$\mathsf{Vec}_A : \mathbb{N} \to \mathcal{U}.$$

It has two constructors and an eliminator of the following form:

$$\frac{}{\mathsf{nil} : \mathsf{Vec}(0)} \qquad \frac{n : \mathbb{N} \qquad a : A \qquad v : \mathsf{Vec}(n)}{\mathsf{cons}(a, v) : \mathsf{Vec}(n+1)}$$

$$\frac{\begin{array}{c} C : \{n : \mathbb{N}\} \to \mathsf{Vec}(n) \to \mathcal{U} \qquad p_{\mathsf{nil}} : C(\mathsf{nil}) \\ p_{\mathsf{cons}} : \{n : \mathbb{N}\}(a : A)(v : \mathsf{Vec}(n)) \to C(v) \to C(\mathsf{cons}(a, v)) \\ n : \mathbb{N} \qquad v : \mathsf{Vec}(n) \end{array}}{\mathsf{elim}_{\mathsf{Vec}}(C, p_{\mathsf{nil}}, p_{\mathsf{cons}}, v) : C(v)}$$

with two reduction rules

$$\begin{aligned} \mathsf{elim}_{\mathsf{Vec}}(C, p_{\mathsf{nil}}, p_{\mathsf{cons}}, \mathsf{nil}) &\equiv p_{\mathsf{nil}} \text{ and} \\ \mathsf{elim}_{\mathsf{Vec}}(C, p_{\mathsf{nil}}, p_{\mathsf{cons}}, \mathsf{cons}(a, v)) &\equiv p_{\mathsf{cons}}(a, v, \mathsf{elim}_{\mathsf{Vec}}(C, p_{\mathsf{nil}}, p_{\mathsf{cons}}, v)). \end{aligned}$$

In natural language, the eliminator says that to show a statement about vectors it is sufficient to prove it about the empty vector and that the statement is stable under extending an arbitrary vector. Note that the natural number in the conclusion of both introduction rules is not a variable, and the cons even modifies the natural number. As such, we call the dependency on the length in the type of vectors an *index* instead of a parameter.

While dependent and non-dependent functions were primitives needed to even talk about other type formers, all other examples which we have seen, can be captured by the concept of *indexed inductive type*. But since we want to be more formal than to just assume the presence of indexed inductive types based on giving a few examples, we will present different attempts to make precise a definition of what an indexed inductive type is. One schematic approach was made by Dybjer [1994], while the approach which we want to introduce here requires an encoding of the constructors in a very specific way to be able to present the inductive type as an instance of a very general form of a *tree*. The types covered by this approach are called *indexed W-Types*, and they cover a bigger fragment of inductive types than the non-indexed version, which are also known as "Petersson-Synek trees" [Petersson and Synek, 1989], and which are in a topos theoretic setting, discussed by Moerdijk and Palmgren [2000]. The generalization to indexed W-Types, as presented here, was found by Altenkirch et al. [2015].

Definition 2.2.1. Assume that we are given the following data:

- A type $I : \mathcal{U}$ of *indices*,
- a type $A : \mathcal{U}$ encoding the number and non-recursive input *data* or *shapes* for contructors,
- a function $o : A \to I$ assigning to each piece of input data the corresponding *output index* of the constructor,
- a type $B : A \to \mathcal{U}$ of *recursive occurrences* or *positions* for each bit of input data, and
- a function $r : (a : A) \to B(a) \to I$ of indices of *recursive occurrences*.

Then, we assume that we have a type $\mathsf{IW}^{o,r}_{A,B} : I \to \mathcal{U}$ with the following introduction and elimination rules:

$$\text{IW-Intro}\ \frac{a : A \qquad c : (b : B(a)) \to \mathsf{IW}^{o,r}_{A,B}(r(a,b))}{\mathsf{sup}(a,c) : \mathsf{IW}^{o,r}_{A,B}(o(a))}$$

$$\text{IW-Elim}\ \frac{\begin{array}{c} C : \{i : I\} \to \mathsf{IW}^{o,r}_{A,B}(i) \to \mathcal{U} \\ p : (a : A)\Big(c : (b : B(a)) \to \mathsf{IW}^{o,r}_{A,B}(r(a,b))\Big) \\ \to \Big((b : B(a)) \to C(c(b))\Big) \to C(\mathsf{sup}(a,c)) \end{array}}{\mathsf{elim}_{\mathsf{IW}}(C,p) : (i : I)(w : \mathsf{IW}^{o,r}_{A,B}(i)) \to C(w)}$$

We furthermore assume that we are provided the reduction rule

$$\mathsf{elim}_{\mathsf{IW}}(C,p,o(a),\mathsf{sup}(a,c)) \equiv p(a,c,\lambda b : B(a).\, \mathsf{elim}_{\mathsf{IW}}(C,p,r(a,b),c(b))).$$

This definition may seem very confusing and overly complicated, but this is necessary to capture all possible indexed inductive types in full generality. In words, the constructor describes that we chose a constructor and giving possible non-recursive input data by providing $a : A$, and then, based on this data, give data for all recursive occurrences of the type in a constructor in the form of c, we get a new element $\mathsf{sup}(a,c)$ which is reminiscent of the *supremum* of the given data. Conversely the elimination rule describes that to proof a statement C about this tree like structure it is enough to prove C for $\mathsf{sup}(a, c)$ under the hypothesis that it holds for all recursive input data $c(b)$.

Assuming that we already have $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$ and Σ-types at our disposal (allowing us to define $A + B$), indexed W-types live up to our expectations of capturing all previously mentioned inductive types. To provide a translation of these, including indexed inductive types still to be defined in the next chapter, we will provide all the arguments for the respective indexed W-types in Table 2.1, while often, in the case of $I \equiv \mathbf{1}$ describe $\mathbf{1} \to A$ in place of a type A itself.

Remark 2.2.2 (Plain W-Types)**.** Indexed W-types can be reduced to the simpler class of *W-types*, which only are specified by giving a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, without having a type of indices $I : \mathcal{U}$, together with o and r. We can think of them as the class of indexed W-types where $I \equiv \mathbf{1}$.

The fact that we can present each indexed W-types by a non-indexed one was shown by Altenkirch et al. [2015] using the K-rule, and by Sattler [2015] without presence of the K-rule and under assumption of functional extensionality.

In Chapters 5 and 6 we will introduce two more calculi to replace indexed W-types.

2.3 Typal Equality and Homotopy Type Theory

Until now, the only equations we encountered were judgmental equalities which aren't "visible" internally in the type theory. But since we want to follow the paradigm of propositions-as-types, we also want to have a way to represent the statement that two terms of a type $A : \mathcal{U}$ are equal *inside* the type theory. To correct this, we introduce what is usually called **propositional equality** or **typal equality**. For each type $A : \mathcal{U}$ and each two elements $a, b : A$ we want to have a type $(a =_A b) : \mathcal{U}$ of proofs that a and b are equal. We can view this as an inductive definition, giving only

Type Former	$I : \mathcal{U}$	$A : \mathcal{U}$	$B : A \to \mathcal{U}$	$o : A \to I$	$r : (a : A) \to B(a) \to I$
$\mathbb{N}$	$\mathbf{1}$	$\mathbf{1}$ $+\mathbf{1}$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(\star) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$
Vec_A	$\mathbb{N}$	$\mathbf{1}$ $+A \times \mathbb{N}$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(a, n) \mapsto \mathbf{1}$	$\mathsf{inl}(\star) \mapsto 0$ $\mathsf{inr}(a, n) \mapsto n + 1$	$-$ $\mathsf{inr}(a, n) \mapsto (\star) \mapsto n$
$x =_A \_$	A	$\mathbf{1}$	$\star \mapsto \mathbf{0}$	$\star \mapsto x$	$-$

Table 2.1: The input data for the indexed W-types corresponding to the type formers given in Chapter 2.1.

the witness for this relation to be reflexive as a constructor:

$$\text{=-F\small{ORM}}\ \frac{A : \mathcal{U} \qquad a, b : A}{a =_A b : \mathcal{U}} \qquad \text{=-I\small{NTRO}}\ \frac{A : \mathcal{U} \qquad a : A}{\mathsf{refl}_a : a =_A a}$$

$$\text{=-E\small{LIM}}\ \frac{\begin{array}{c} a : A \qquad C : (b : A) \to a =_A b \to \mathcal{U} \\ c : C(a, \mathsf{refl}_a) \qquad b : A \qquad p : a =_A b \end{array}}{\mathsf{elim}_{=_A}(a, C, c, b, p) : C(b, p)}$$

with the reduction rule $\mathsf{elim}_{=_A}(C, c, a, p, \mathsf{refl}_a) \equiv c(a)$. It can be defined as an indexed W-type as per Table 2.1. The elimination rule says that to prove a statement indexed over varying right hand sides of an equality and corresponding equality proofs, we can assume it to be the reflexivity witness refl.

Remark 2.3.1. Apart from the above elimination rule, which is sometimes referred to as **J-rule**, some type theories also assume the so called **K-rule**, which in proofs also allow us to simplify a given equality proof to refl if the type family which we want to inhabit varies both endpoints of the equality simultaneously, as in the following:

$$\text{=-K}\ \frac{\begin{array}{c} C : (a : A) \to a =_A a \to \mathcal{U} \\ c : (a : A) \to C(a, \mathsf{refl}_a) \qquad a : A \qquad p : a =_A a \end{array}}{\mathsf{elim}'_{=_A}(C, c, a) : C(a)}$$

We will assume this rule to hold in Chapter 5 and later chapters, but explicitly assume its absence in this and the following two chapters.

In literature, our version of =-Elim is often called the *based* J-rule, because the equation's left hand side is fixed, or Paulin-Mohring J-rule, after Paulin-Mohring [1993]. An alternative, but provably equivalent version is the so-called **unbased J-rule** in which the type family varies over both sides of the equation:

$$\text{=-E\small{LIM}}'\ \frac{\begin{array}{c} C : (a, b : A) \to a =_A b \to \mathcal{U} \\ c : C(a, a, \mathsf{refl}_a) \qquad a, b : A \qquad p : a =_A b \end{array}}{\mathsf{elim}'_{=_A}(C, c, a, b, p) : C(b, p)}$$

While we introduced types as a means to represent sets and propositions, passing on the K-rule also gives rise to a further interpretation: We can use types to model the homotopy types of topological spaces. This principle, together with the univalence axiom which we will introduce in the next chapter, is the underlying insight of the field of *homotopy type theory*. This correspondence was first explored by Awodey and Warren [2009] and made precise by Kapulkin and Lumsdaine [2012], who, after an idea by Vladimir Voevodsky, modelled homotopy type theory using simplicial sets. In this correspondence – a succinct list of which can be found in Table 2.2 – equality proofs correspond to **paths**, a name which we will use as a synonym from using from now on.

But what does equality, as defined here, entail? The most important observation is, that two equal objects are *indiscernable* by any attribute in the sense that for any type family $C : A \to \mathcal{U}$ if $C(a)$ and $a = b$, then we can prove $C(b)$, as we can prove in the following lemma which defines an operation to achieve this:

Lemma 2.3.2 (Transport)**.** *Let $A : \mathcal{U}$ and $C : A \to \mathcal{U}$. For any two elements $a, b : A$ with $p : a = b$ we can define the* ***transport*** *operation on p:*

$$p^* : C(a) \to C(b)$$

Proof. By induction, we can assume p to be refl_a, on which we can define $(\mathsf{refl}_a)^*(c) :\equiv c$. □

While it is obvious that equality should be an equivalence relation, we only assume a witness for reflexivity. But what about its symmetry and transitivity? It turns out that these can be proven by induction without the need to add them as constructors:

Lemma 2.3.3. *Let again be $A : \mathcal{U}$, $a, b : A$, and $p : a = b$. Then, there is an equality proof $p^{-1} : b = a$, called the* ***inverse*** *of p.*

Proof. Applying the eliminator to p, we can reduce this problem to finding a path $(\mathsf{refl}_a)^{-1} : a = a$, which we can define to be refl_a itself. □

Lemma 2.3.4. *For $A : \mathcal{U}$, elements $a, b, c : A$ and paths $p : a = b$ and $q : b = c$, there is a path $p \cdot q : a = c$, proving that propositional equality is transitive.*

Proof. Here it suffices to apply induction on the second path and give the definition

$$p \cdot \mathsf{refl}_b :\equiv p.$$

□

	Logical Interpretation	Set Interpretation	Homotopical Interpretation
(closed) Type $A : \mathcal{U}$	Proposition	Set	Topological Space
Function $f : A \to B$	Proof of implication	Function between sets	Continuous map
Type family $B : A \to \mathcal{U}$	Proposition with free variable in A	Family of sets indexed by A	Fibration over base A
Pair $(a,b) : A \times B$	Proof of conjunction	Pair in cartesian product	Point in product space
Element $\mathsf{inl}(a) : A + B$	Proof of disjunction	Element of disjoint union	Point in disjoint union
Dep. fn. $f : \Pi(B)$	Proof of universal quantification	Dep. set valued function	Section of fibration
Dep. pair $(a,b) : \Sigma(B)$	Proof of existential quantification	Dependent product	Point in total space of fibration
Equality $p : a =_A b$	–	Equality in A	Path from x to y
Equivalence $f : A \simeq B$	Proof of biconditional	Bijection	Homotopy equivalence

Table 2.2: Three interpretations

Having provided the proof that equality is an equivalence relation, we can already conclude that it makes each type a *setoid*. But more than that we can also prove that it carries the structure of a *groupoid*:

Lemma 2.3.5 (Groupoid laws). *Let $A : \mathcal{U}$, $a, b, c, d : A$ and $p : a = b$, $q : b = c$ and $r : c = d$. Then,*

- $p = p \cdot \mathsf{refl}_b = \mathsf{refl}_a \cdot p$,
- $p^{-1} \cdot p = \mathsf{refl}_b$, $p \cdot p^{-1} = \mathsf{refl}_a$,
- $(p^{-1})^{-1} = p$ *and*
- $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

Proof. The first three laws can be proven by induction on the path p, the last one by induction on r. □

As a suitable notion of equality, propositional equality is respected by functions:

Lemma 2.3.6. *Let $A, B : \mathcal{U}$, $f : A \to B$, and $a, b : A$. Then, there is a function $\mathsf{ap}_f : (a = b) \to (f(a) = f(b))$ such that $\mathsf{ap}_f(\mathsf{refl}_a) \equiv \mathsf{refl}_{f(a)}$.* □

Under this notion ap_f, every function is functorial with respect to equality. Besides this, it is also functorial with respect to function composition. The following laws can be proved by induction on the paths involved:

Lemma 2.3.7. *Let $A, B, C : \mathcal{U}$, $f : A \to B$ and $g : B \to C$. For paths $p : a =_A b$ and $q : b =_A c$ we have equalities*

- $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$,
- $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$,
- $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$, *and*
- $\mathsf{ap}_{\mathsf{id}_A}(p) = p$. □

If in the above considerations f is instead a dependent function, we can not necessarily consider the type $f(a) = f(b)$ since the left and right hand side need not have the same type. For this situation, there is a dependent version of the above defined function ap_f:

Lemma 2.3.8. *Let $A : \mathcal{U}$, $B : A \to \mathcal{U}$ and $f : (a : A)B(a)$. Then, we can construct a dependent function*

$$\mathsf{apd}_f : \{a, b\}(p : a =_A b) \to p_*(f(a)) =_{B(b)} f(b),$$

such that $\mathsf{apd}_f(a, a, \mathsf{refl}_a) \equiv \mathsf{refl}_{f(a)}$. □

We continue by considering one of the representation of one of the most import notions of homotopy theory: Homotopies between functions.

Definition 2.3.9 (Homotopy of functions)**.** Two maps $f, g : A \to B$ are called **homotopic** or **pointwise equal** if for all $a : A$ we have $f(a) = g(a)$. We define the notation

$$f \sim g :\equiv (a : A) \to f(a) = g(a).$$

We can define the same for two dependent functions $f, g : \prod_{(a:A)} B(a)$ over the same type family.

Remark 2.3.10 (Function Extensionality)**.** Since in homotopy theory, we can find a path $p : f = g$ in the space of functions, whenever $f \sim g$, we might ask if in homotopy type theory this holds as well – are two functions equal as soon as they are pointwise equal? In the setting of homotopy type theory we will be able to derive this from univalence, while in settings where we assume the K-rule, we will usually assume this property of the function space, which is called **function extensionality**, axiomatically.

So far, we have not considered what it means for two *types to be equal*. Of course, our definition of equality is valid for the universe itself as well, but how does it relate to how we represent biconditionals of propositions, bijections between sets, and homotopy equivalences between spaces? One common representation of all of these three concepts is the notion of equivalence of types:

Definition 2.3.11 (Equivalences)**.** Let $A, B : \mathcal{U}$. A function $f : A \to B$ is called an **equivalence** between A and B, if there is a $g : B \to A$ such that $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$ and furthermore

$$\tau : \prod_{a:A} \mathsf{ap}_f(\eta(a)) =_{f(g(f(a)))=a} \epsilon(f(a)) \equiv \mathsf{ap}_f \circ \eta \sim \epsilon \circ f.$$

We need τ to make sure that each two witnesses for the fact that f is an equivalence are equal. Since τ is one of the two commutativity conditions for pairs of adjoint functors, this kind of equivalence is also called a **half adjoint equivalence**. The type of all equivalences between two types $A, B : \mathcal{U}$ is denoted by

$$A \simeq B :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f).$$

It is easy to show that the equivalence of types is indeed an equivalence relation and that it behaves as we expect it to on easy examples such as $(\mathbf{1} \to A) \simeq A$, $(\mathbf{0} + A) \simeq A$, and $(\mathbf{0} \times A) \simeq \mathbf{0}$.

But how can we now compare the equivalence and equality on types? By induction the reflexivity proof for equivalence is enough to show that equality implies equivalence: For each $A, B : \mathcal{U}$ there is

$$\mathsf{idtoeqv}_{A,B} : (A =_{\mathcal{U}} B) \to (A \simeq B).$$

But how about the other direction? It turns out that there is no way to obtain equality from equivalence, so one solution is to assume axiomatically, that $\mathsf{idtoeqv}_{A,B}$ has an inverse function. The consistency of this axiom has been shown by Kapulkin and Lumsdaine [2012], after an idea by Vladimir Voevodsky.

Axiom 2.3.12 (Univalence)**.** *For every $A, B : \mathcal{U}$ we asume that $\mathsf{idtoeqv}_{A,B}$ is itself an equivalence. This implies that*

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B)$$

and yields an inverse to $\mathsf{idtoeqv}_{A,B}$ which we call

$$\mathsf{ua}_{A,B} : (A \simeq B) \to (A =_{\mathcal{U}} B).$$

Remark 2.3.13. Postulating axioms is avoided as often as possible in type theory, since it is detrimental to one of the desirable properties of the type theory: normalization. To give an example, with an axiom like ua, not every closed term of the natural numbers can be reduced to a numeral any more. It is for this reason that type theorists have been looking for ways to achieve univalence without postulating it as an axiom. One approach is to not consider equality as an inductive type any more but to have it as a primitive in the language. This approach is used in a variety of type theory called "cubical type theory" [Cohen et al., 2016].

Another instance of equalities we have not explored so far are *iterated equalities*: Equalities between equality proofs. Since for types $A : \mathcal{U}$ representing propositions we might only care about whether or not we have a proof for it, it might not make sense to consider multiple elements, so we might want that for all $a, b : A$ we have $a = b$. For types that should represent sets we *do* want multiple elements, but we might not be interested in having different equality proofs – so for $a, b : A$ and $p, q : a =_A b$ we might want to have a proof for $p =_{a=b} q$. This effect, that at a certain level of iteration, all equality proofs should become equal is called **truncation**:

Definition 2.3.14 (n-types)**.** For a type $A : \mathcal{U}$ we set

$$\begin{aligned}\mathsf{is\text{-}(-1)\text{-}type}(A) \equiv \mathsf{isProp}(A) &:\equiv (a, b : A) \to a = b,\\ \mathsf{is\text{-}0\text{-}type}(A) \equiv \mathsf{isSet}(A) &:\equiv (a, b : A)(p, q : a = b) \to p = q, \text{ and}\\ \mathsf{is\text{-}}(n+1)\mathsf{\text{-}type}(A) &:\equiv (a, b : A) \to \mathsf{is\text{-}}n\mathsf{\text{-}type}(a = b) \text{ for } n : \mathbb{N}.\end{aligned}$$

and call A a **(mere) proposition** if $\mathsf{isProp}(A)$, a **set** if $\mathsf{isSet}(A)$ and more general an n**-type** or n-truncated if $\mathsf{is\text{-}}n\mathsf{\text{-}type}(A)$. We also extend this definition to include what it means for a type to represent a contractible space, a singleton, or a true proposition:

$$\mathsf{is\text{-}(-2)\text{-}type}(A) \equiv \mathsf{isContr}(A) :\equiv (a : A) \times ((b : B) \to a = b).$$

There are some important, but easy to prove facts about truncatedness:

Lemma 2.3.15.

- *If $A : \mathcal{U}$ is an n-type, then A is an $(n+1)$-type.*
- *For each $n \geq -2$ and $A : \mathcal{U}$, the type $\mathsf{is\text{-}}n\mathsf{\text{-}type}(A)$ is a mere proposition.*
- *If $A : \mathcal{U}$ is an n-type and $B : A \to \mathcal{U}$ such that for each $a : A$, $B(a)$ is an n-type, then $(a : A) \times B(a)$ is an n-type as well.*
- *If $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ are such that $B(a)$ is an n-type for each $a : A$, then $(a : A) \to B(a)$ is an n-type as well.*
- *Let $n \geq -1$. Then, $A : \mathcal{U}$ is an $(n+1)$-type if and only if for all $a : A$, the equality type $a =_A a$ is an n-type.*

- *Let $n \geq 0$. Then, $A : \mathcal{U}$ is an n-type, if and only if for all $a : A$, the n-*fold iterated loop space $\Omega^{n+1}(A, a)$ *is contractible. The n-fold iterated loop space is defined by recursion on n by*

$$\begin{aligned} \Omega^1(A, a) &:\equiv (a =_A a) \textit{ and} \\ \Omega^{n+1}(A, a) &:\equiv (\mathsf{refl}_{\cdot^{\cdot^{\cdot}}_a} =_{\Omega^n(A,a)} \mathsf{refl}_{\cdot^{\cdot^{\cdot}}_a}). \end{aligned}$$

□

Remark 2.3.16 (Uniqueness of Identity Proofs)**.** Truncation levels are a notion which behaves quite differently depending on whether we assume presence of the K-rule or not.

If, as in homotopy type theory, we assume univalence and only the J-rule, it is an easy exercise to find types which are not sets. In fact the universe $\mathcal{U}$ itself is not a set [Homotopy Type Theory, 2013, Example 3.1.9]: We can construct two different automorphisms of $\mathbf{2}$: The identity function and the map $f : \mathbf{2} \to \mathbf{2}$ with $f(0_\mathbf{2}) :\equiv 1_\mathbf{2}$ and $f(1_\mathbf{2}) :\equiv 0_\mathbf{2}$. It is easy to prove that these are non-equal and as such, using univalence, yield different instances of the type $\mathbf{2} = \mathbf{2}$.

If on the other hand we *do* assume the K-rule, then it is immediate – by first applying the J-rule to p and then the K-rule to q – that for every type A we can use the rule to show the inhabitedness of the type family

$$(a, b : A)(p, q : a = b) \to p = q,$$

and so A is a set.

2.4 Theorem Provers Based on Type Theory

As remarked at the beginning of Chapter 2, we value type theory not only for its capability to represent mathematics and logics, but also for its good computational behaviour. This suggests to have a look at how type theory is implemented in different languages the focus of which can be either to be a functional programming language based on dependent types or a theorem proving tool, or to be useful in both of these cases. With regards to this thesis, introducing different implementations is relevant in two ways:

1. A good part of the new results in the form of definitions, lemmas, and theorems, has been formalized in a one of these languages, and

2. both the higher Seifert-van Kampen theorem presented in Section 4.4 as well as the attempted reduction from inductive-inductive type to inductive families in Chapter 7 suggest useful new features or extensions of implementations of type theory.

We will thus concentrate on the two languages we have used in this respect: Lean [de Moura et al., 2015] and Agda [Norell, 2009]. This does not mean that the content discussed is irrelevant to other implementations or that these are less important. Au contraire – the theorem prover Coq [Barras et al., 1997] is not only the implementation with the biggest user base but also has a lot of distinguishing features.

Before we delve into the differences between Lean and Agda, we will first take a look at what these provers have *in common*. The following list contains both fundamental design principles as well as several features which are important to improve the usability of the implementation:

- Implementation of *Martin-Löf type theory* with an unlimited chain of universes. Construction can be made *universe polymorphic* by the use of universe variables which are used as variable indices for the unvierses.

- Function types distinguish between explicit and *implicit arguments*. The latter are useful to make definitions more succinct.

- Both languages are equipped with a system of *modules* or *name spaces* used to organize definitions and their scope.

- There is a way to customize the implementation by the means of *options*. This enables us to switch between a set-truncated setting and homotopy type theory.

- The implementations have an source code which is open and thus reviewable and customizable by everybody.

- There are a one or several text editors which support an interactive and well-integrated use of the language.

Let us next concentrate on Lean as our first specimen.

2.4.1 Lean

The development of the theorem prover Lean was started by Leonardo de Moura at Microsoft Research in 2013. It aims to be both a useful tool for the formalization of mathematics and the verification of programs written in other languages, as well as to be a useful and performant programming language itself. Another goal of Lean is to make automated theorem proving (such as the solution for SMT-style problems) available in a dependently typed theorem prover. As of now, Lean is in its third major version, after a lot of the system was overhauled in the transition from Lean 2. The fourth version is under development but not yet ready to be used as a prover.

Lean – as its name suggests – relies on a relatively small trusted kernel to which the user code is compiled to. This makes it easier to make claims about the consistency of the prover. Let us first take a look at some of the specifics of Lean's type theory.

Lean is based on *inductive families* as described by Dybjer [1994]. Whenever the user writes down the definition of such an inductive family, Lean generates its constructors and its dependent eliminator and makes it available to the user. For example, the following snippet shows a definition of the natural numbers:

```
inductive nat : Type 0
| zero : nat
| succ : nat → nat

#check nat.succ nat.zero -- Prints nat.succ nat.zero : nat
#check @nat.rec_on -- Π {C : nat → Type u} (n : nat),
                   --   C nat.zero →
                   --   (Π (a : nat), C a → C (nat.succ a)) → C n
```

A widely used subclass of inductive types are **structures**, which are similar to what is often referred to as "records". Structures are inductive types which are non-recursive and only have one constructor. That means that they are equivalent to iterated sigma types but, among other advantages, have named projections. Structures provide a basic inheritance mechanism as they can extend other structures:

```
structure graph (V : Type) :=
  (E : V → V → Type)

structure refl_graph (V : Type) extends graph V :=
  (refl : Π (v : V), E v v)

structure trans_graph (V : Type) extends graph V :=
  (trans : Π (u v w : V), E u v → E v w → E u w)
```

Lean features a strict and impredicative universe `Prop` of propositions. Since this universe is incompatible with homotopy type theory, we have to be careful to exclude its use whenever we want formalizations to hold in a setting of homotopy type theory instead of truncated type theory. *Strict* here means that if we have a proposition `p : Prop` and two proofs `a, b : P`, these are considered judgmentally equal, thus for every type family `P → Type`, the fibers `P a` and `P b` are judgmentally equal as well. *Impredicative* means, that for every type family valued in `Prop`, its universal quantification `∀ a, P a` is in `Prop` as well, even if `A` is not.

Lean makes heavy use of *tactics*. These are commands which can modify a given proof goal or a series of proof goals. They are accesible once the user has switched from a mode where she can enter literal proof terms to a *tactic-mode*, which allows her to give proofs and definitions as a sequence of tactic applications, which are then desugared as the do-notation of a *tactic monad*. Here is an example for a tactic proof in lean, using a simplification tactic, a rewriting tactic, and a tactic calling previously defined theorems:

```
example : ∀ (n : ℕ), n = 0 ∨ n > 0 :=
begin
  intro n,
  induction n with n' IH,
  simp,
  right,
  cases IH,
  rw IH, constructor,
  apply nat.zero_lt_succ
end
```

Lean has its own *meta-language* which allows the user to define new language features like commands and tactics. It does so by providing a way to reflect arbitrary terms into an inductive *type of expressions* `expr`, such that by recursion on this type, structural recursion on the term can be emulated.

Lean has several big libraries, the biggest one, which provides basic data types, tactics, and formalization in different fields of mathematics, is called *Mathlib* (`https://github.com/leanprover-community/mathlib`). It is not recommended to use Lean without using Mathlib. Another big formalization effort it Lean's homotopy type theory library, the development of which has been described by van Doorn et al. [2017]. It was originally written in Lean 2 and then ported to Lean 3. A good introduction to the language can be found online [Avigad et al., 2015].

2.4.2 Agda

Agda is a theorem prover. Its first version was written by Catarina Coquand, while the development of the current major version, Agda 2, was initiated by Andreas Abel and Ulf Norell in 2005. There are biannual "Agda Implementors Meetings" to discuss new ideas on how to improve Agda.

Compared to Lean, Agda implements a wider range of inductive types, also allowing *inductive-inductive types*. In contrast to Lean it does not automatically produce the eliminator for an inductive definition but instead provides an induction principle based on *dependent pattern matching*: The user can write (dependent) functions with an inductive domain, by pattern matching on its input. In the following example, an indexed type of vectors and its head function are defined. Agda's pattern matching algorithm recognizes that it can exclude the case of the first constructor in the definition of the head function.

```
data Vec (A : Set) : N → Set where
  nil : Vec A Z
  cons : ∀{n} → A → Vec A n → Vec A (S n)

hd : ∀{A n} → Vec A (S n) → A
hd (cons a v) = a
```

Agda has several options which make it possible to use it with a range of different type theories: Homotopy type theory is supported by declaring via an option `--without-K` that it should reject any use of the K-rule. Agda also has an option `--rewriting` whic makes it possible to declare many equalities as strict by declaring their proofs to be *rewrite rules* [Cockx and Abel, 2016], giving the type theory the flavour of extensional type theory.

Agda has also been modified to support a type theory which provides a constructive, non-axiomatic version of univalence. Based on the geometry of its identity types, this type theory is called *cubical type theory*, and the modified version has the name “Cubical Agda” [Vezzosi, 2018].

Chapter 3

Higher Inductive Types

3.1 Examples of Higher Inductive Types

The propositional equality discussed in the last chapter works just fine if we want to *prove* things to be equal. But in mathematics as well as computer science, we also often want to *make things equal*, in the sense that we might want to consider a type A and two of its elements $a, b : A$ and want to obtain another type which only differs from A in that a and b are equal. In short, we want to take *a quotient*. In this chapter we will present different ways to achieve quotients in the setting of homotopy type theory. Afterwards we will show how we can derive all of these from the basic notion of a *homotopy coequalizer*.

The general way to achieve quotients is to go beyond the inductive types which we encountered in the last chapters – and which were all examples of indexed W-types – and also allow for constructors which, in contrast to the *point constructors* which we have seen so far, are *path constructors*. Instead of adding new elements to the inductive type, these constructors are there to make instances of other constructors equal. In a setting where the K-rule is present, and every type is a set (see Remark 2.3.16), this bigger class of inductive types is called **quotient inductive types**, while, as we will see, in homotopy type theory it is more fitting to call them **higher inductive types** (HITs) given the fact that equality types carry not only proofs but data. Higher inductive types allow the development of a synthetic version of homotopy theory inside HoTT (cf. Buchholtz et al. [2018], Buchholtz and Hou (Favonia) [2018], Buchholtz and Rijke [2016, 2017], Hou (Favonia) and Shulman [2016], Licata and Finster [2014], Licata and Brunerie [2013], Brunerie [2017], Rijke [2017]). A main objective of this line of research is to describe, classify, and compare path spaces (i. e. equality types) or homotopy groups (i. e. truncated path spaces) of higher inductive types such as circles and spheres.

A minimal example for a higher inductive type could be the following definition of a type theoretic representation of the space of the interval I. Its constructors are two points, but furthermore also a path which connects these two points:

$$I\text{-}\textsc{Intro1}\ \frac{}{0_I : I} \qquad I\text{-}\textsc{Intro2}\ \frac{}{1_I : I} \qquad I\text{-}\textsc{Intro3}\ \frac{}{\mathsf{seg} : 0_I = 1_I}$$

$$I\text{-}\textsc{Elim}\ \frac{\begin{array}{c} C : I \to \mathcal{U} \\ c_0 : C(0_I) \qquad c_1 : C(1_I) \qquad p : \mathsf{seg}_*(c_0) = c_1 \qquad x : I \end{array}}{\mathsf{elim}_I(C, c_0, c_1, p, x) : C(x)}$$

It is easy to check that the unique map $I \to \mathbf{1}$ is an equivalence, and so I is contractible and thus a set. But what if instead of two point constructors we only had one, as in the higher inductive types governed by the following rules?

$$\mathbb{S}^1\text{-}\textsc{Intro1}\ \frac{}{\mathsf{base} : \mathbb{S}^1} \qquad \mathbb{S}^1\text{-}\textsc{Intro2}\ \frac{}{\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}}$$

$$\mathbb{S}^1\text{-}\textsc{Elim}\ \frac{C : \mathbb{S}^1 \to \mathcal{U} \qquad c : C(\mathsf{base}) \qquad p : \mathsf{loop}^*(c) = c \qquad x : \mathbb{S}^1}{\mathsf{elim}_{\mathbb{S}^1}(C, c, p, x) : C(x)}$$

We can see that loop introduces a new path from base to itself which cannot be reduced to $\mathsf{refl}_{\mathsf{base}}$. This means that we can interpret it as a loop which is not homotopic to the identity, and so the type represents the circle (cf. Figure 3.1). The fact that loop is not equal to $\mathsf{refl}_{\mathsf{base}}$ also makes it clear that this type is not a set.

In the same way in which we can add arbitrary paths between constructors, we can also use iterated equality types to express the addition of arbitrary higher dimensional cells (surfaces, volumes). An example for this is the definition of a twodimensional sphere, where we have one basepoint and one surface:

$$\mathbb{S}^2\text{-}\textsc{Intro1}\ \frac{}{\mathsf{base} : \mathbb{S}^2} \qquad \mathbb{S}^2\text{-}\textsc{Intro2}\ \frac{}{\mathsf{surf} : \mathsf{refl}_{\mathsf{base}} =_{\mathsf{base}=\mathsf{base}} \mathsf{refl}_{\mathsf{base}}}$$

Besides these closed examples for higher inductive types, we can also have important constructions which are parametrized over an arbitrary

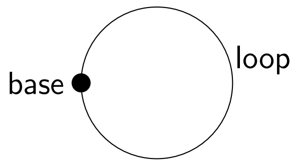


Figure 3.1: A circle with a base point.

type. One important operation in topology, especially in the field of homologies, is the one of the **suspension** $\mathsf{Susp}(A)$ of spaces A which turns an n-dimensional type into a $(n+1)$-dimensional type. Suspensions will also provide a way to conveniently define all higher-dimensional spheres by setting $\mathbb{S}^{n+1} :\equiv \mathsf{Susp}(\mathbb{S}^n)$. The suspension has two point constructors (sometimes called the north and south pole) and for each point in A a path between those points:

$$\frac{}{\mathsf{N} : \mathsf{Susp}(A)} \qquad \frac{}{\mathsf{S} : \mathsf{Susp}(A)} \qquad \frac{a : A}{\mathsf{merid}(a) : \mathsf{N} = \mathsf{S}}$$

$$\frac{\begin{array}{c} C : \mathsf{Susp}(A) \to \mathcal{U} \\ c_{\mathsf{N}} : C(\mathsf{N}) \qquad c_{\mathsf{S}} : C(\mathsf{S}) \qquad c_{\mathsf{merid}} : (a : A) \to \mathsf{merid}(a)^*(\mathsf{N}) = \mathsf{S} \end{array}}{\mathsf{elim}_{\mathsf{Susp}(A)}(C, c_{\mathsf{N}}, c_{\mathsf{S}}, c_{\mathsf{merid}}) : (x : \mathsf{Susp}(A)) \to C(x)}$$

Another very general construction is the **pushout of types**. It does not only depend on one a single type but instead has as input a whole *span* of types, meaning three types L, M, and N, and functions $f : L \to M$ and $g : L \to N$. It consists of a type $P \equiv M \sqcup^L N$ the point constructors of which are to functions inl and inr as in the following diagram:

$$\begin{tikzcd}
L \arrow[r, "g"] \arrow[d, "f"'] & N \arrow[d, dashed, "\mathsf{inr}"] \\
M \arrow[r, dashed, "\mathsf{inl}"'] & P
\end{tikzcd}$$

The introduction rules are the same as for the sum type, but each instance of L contributes a new path in the resulting type:

$$\frac{m : M}{\mathsf{inl}(m) : M \sqcup^L N} \qquad \frac{n : N}{\mathsf{inr}(n) : M \sqcup^L N}$$

$$\frac{l : L}{\mathsf{glue}(l) : \mathsf{inl}(f(l)) = \mathsf{inr}(g(l))}$$

One can try to visualize the resulting type as the sum of M and N which was glued along an L-shaped overlapping. The pushout is a very general construction. In fact, it is easy to check that all the previous higher inductive types we presented were just special cases of a pushout, for example the suspension $\mathsf{Susp}(A)$ can also be defined as the pushout where both f and g are the unique map $A \to \mathbf{1}$.

An example of a higher inductive type, the importance of which is easily recognizable even we only care for propositions and sets is the one of the **truncation operator**. It is a tool to bring types to the desired level of truncation, as we can see in the following example: Remember that we represented the disjunctive logical connective with the sum type. But the sum of two propositional types is not necessarily a proposition itself: For example we have $\mathbf{1} + \mathbf{1} \simeq \mathbf{2}$. So how can we make it a proposition? The solution is to add equations between *all* the points in the sum type and additionally make sure that no trivial higher equality proofs exist – the second introduction rule below is a short way of saying that all these paths should exist. Similar situations can happen for every truncation level. For example, a collection of sets is generally not a set itself. The elimination principle for this operation has to make sure that only functions on the truncated type, which represent continuous maps, can be created, and so it has to require that the fibers of the type family which we want to inhabit are truncated:

$$\textsc{Trunc-Intro1}\ \frac{a : A}{|a|_n : \|A\|_n} \qquad \textsc{Trunc-Intro2}\ \frac{}{\mathsf{is}\text{-}n\text{-}\mathsf{type}(\|A\|_n)}$$

$$\textsc{Trunc-Elim}\ \frac{\begin{array}{c} C : \|A\|_n \to \mathcal{U} \\ p : (x : \|A\|_n) \to \mathsf{is}\text{-}n\text{-}\mathsf{type}(C(x)) \qquad q : (a : A) \to C(|a|_n) \end{array}}{\mathsf{elim}_{\|A\|_n}(C, p, q) : (x : \|A\|_n) \to C(x)}$$

Next, we will discuss how – parallel to how we unified the examples for indexed inductive types using indexed W-types – find a general way to express all higher inductive types which we need.

3.2 Coequalizers as a Fundamental HIT

For a long time after homotopy type theory became its own field, the question about what could be a suitable syntax and semantics for higher inductive types remained open. To make up for the lack of core support for higher inductive types in interactive theorem provers, users of Coq, Agda, and Lean decided to come up with pragmatic definitions and implementations. In Agda and Coq, the common solution was to apply a trick first documented by Licata [2011] which uses *private inductive types* to hide the elimination principle generated by the prover and replace it by a manually defined one. Another strategy which the homotopy type theory formalizations in Lean 2 are based on, and which are described by van Doorn et al. [2017], is to find a single higher inductive types which can serve as a *universal* example which can be used to derive a big range of other instances. This is similar to the generalized type of trees which make up indexed W-types.

The type we use as such a fundamental higher inductive type is much simpler than the definition of indexed W-types, and it is a straightforward generalization of **quotients** in set-based type theory. Suppose that we have a type $A : \mathcal{U}$ and and equivalence relation $\sim : A \to A \to \mathcal{U}$. For a type to be justifiably called the quotient of A by $\sim$, we want a projection map from A into the quotient and we need to make sure that any $a, b : A$ with $a \sim b$ are projected to equal elements of the quotient. An elimination principle should say that maps out of the quotient are the same as maps out of A respecting the relation $\sim$. And these are exactly the features of the type of **homotopy coequalizers** which we will now introduce. The reason why we refrain from calling them quotients – even though van Doorn et al. [2017] call them *typal* quotients – will become apparent in the remarks and examples we will study after giving the inference rules and is due to the fact that we differ from several common assumptions about quotients:

- The base type $A : \mathcal{U}$ does not need to be a set,
- the relation $\sim : A \to A \to \mathcal{U}$ does not need to be an equivalence relation, and
- it does not have to be a relation at all, because for $a, b : A$, the type of relatedness witnesses $a \sim b : \mathcal{U}$ does not need to be a proposition but can contain multiple elements and even structure in higher equalities. (It might be reasonable to speak of a quotient by a higher relation, which is the approach taken by Boulier et al. [2017].)

The formation, introduction, and elimination rules for the coequalizer are the following:

$$\textsc{Coeq-Form}\ \frac{A : \mathcal{U} \qquad \_ \sim \_ : A \to A \to \mathcal{U}}{A/\!\!/\!\sim\ : \mathcal{U}}$$

$$\textsc{Coeq-Intro1}\ \frac{a : A}{[a] : A/\!\!/\!\sim} \qquad \textsc{Coeq-Intro2}\ \frac{a, b : A \qquad s : a \sim b}{\mathsf{glue}(s) : [a] = [b]}$$

$$\textsc{Coeq-Elim}\ \frac{\begin{array}{c} P : A/\!\!/\!\sim\ \to \mathcal{U} \qquad f : (a : A) \to P([a]) \\ e : \{a, b : A\}(s : a \sim b) \to \mathsf{glue}(s)^*(f(a)) = f(b) \end{array}}{\mathsf{elim}_{A/\!\!/\sim}(P, f, e) : (a : A/\!\!/\!\sim) \to P(x)}$$

Remark 3.2.1 (Pathovers)**.** Since we will often encounter equality types with a transported term on one side, it merits its own name and notation: Whenever we have a type family $B : A \to \mathcal{U}$, two points $a, a' : A$, and a path $p : a = a'$, we will for $b : B(a)$ and $b' : B(a')$ define the type of **paths over** or **dependent paths over** p to be

$$(b =_p b') :\equiv (p_*(b) = b').$$

These path-overs have proven to be useful for many formalizations and their attributes, as well as their generalizations to square-overs and cube-overs have been described by Licata and Brunerie [2015].

Remark 3.2.2 (Coequalizer of Functions)**.** In category theory as well as homotopy theory, (homotopy) coequalizers are not defined on an object and a relation but instead are a universal construction on two functions $f, g : X \to A$. It can be thought as an object in which for each $x : X$, $f(x)$ and $g(x)$ are identified. This can be expressed in our variant of a coequalizers by setting the relation on A to be

$$a \sim b :\equiv (x : X) \times (f(x) = a) \times (g(x) = b),$$

or, alternatively, by defining $\_ \sim \_$ inductively by $f(x) \sim g(x)$ for each $x : X$. It is then easy to see that the coequalizer of f and g is $A/\!\!/\!\sim$.

Vice versa, we can view $A/\!\!/\!\sim$ for an arbitrary $A : \mathcal{U}$ and $\sim : A \to A \to \mathcal{U}$ as the coequalizer on the maps

$$\mathsf{pr}_1, \mathsf{pr}_2 : (\Sigma(a, b : A).a \sim b) \rightrightarrows A.$$

Another reason why we might avoid to conflate homotopy coequalizers with ordinary quotients is, that $A/\!\!/{\sim}$ is not always a set, even if A is: We can express the circle which we have defined earlier as a coequalizer in the following simple way: Set $A :\equiv \mathbf{1}$ and for $a, b : A$, set $a \sim b :\equiv \mathbf{1}$. While this, at first glance, might seem to yield a trivial coequalizer, it actually add an *additional* path to the unit type, and it is easy to check that $A/\!\!/{\sim} \simeq \mathbb{S}^1$.

To see the importance of not requiring the relation to be propositional, consider the case of the pushout. It can be written as a coequalizer on $A :\equiv M + N$, but it is easy that the following definition for $\_ \sim \_$ is not necessarily a proposition, and in general it is neither reflexive, nor symmetric, nor transitive:

$$\begin{aligned}(\mathsf{inl}(m) \sim \mathsf{inl}(m')) &:\equiv \mathbf{0},\\ (\mathsf{inl}(m) \sim \mathsf{inr}(n)) &:\equiv (l : L) \times (f(l) = m) \times (g(l) = n), \text{ and}\\ (\mathsf{inr}(m) \sim x) &:\equiv \mathbf{0}.\end{aligned}$$

With the pushouts (and sequential colimits which are defined similarly), and given that there are more intricate constructions giving propositional truncations [van Doorn, 2016, Kraus, 2016] and higher truncations [Rijke, 2017], we have presented ways to encode all the higher inductive types which we have seen so far as coequalizers.

Remark 3.2.3 (Coequalizers as Pushouts)**.** Basing all necessary higher inductive types on homotopy coequalizers is a somewhat arbitrary decision based on aesthetical considerations, since we can also derive coequalizers from pushouts in the following way:

$$\begin{tikzcd} A + (a, b : A) \times a \sim b \arrow[r, "g"] \arrow[d, "f"'] & A \arrow[d, dashed, "\mathsf{inr}"] \\ A \arrow[r, dashed, "\mathsf{inl}"'] & P \end{tikzcd}$$

with

$$\begin{aligned}f(\mathsf{inl}(a)) &:\equiv a,\\ f(\mathsf{inr}(a, b, s)) &:\equiv a,\\ g(\mathsf{inl}(a)) &:\equiv a, \text{ and}\\ g(\mathsf{inr}(a, b, s)) &:\equiv b.\end{aligned}$$

It is then easy to check that the pushout P is equivalent to $A/\!\!/\!\sim$. As a result, we can conclude that pushouts and homotopy coequalizers are *interderivable* in homotopy type theory.

3.3 Encode-Decode Proofs

To motivate our results, which we will present in the next chapter, we will now look at some important problems which occur when we want to prove facts about higher inductive types. Often, we want to find out what specific equality types of higher inductive types look like. For a very concrete example, one of the most basic results in homotopy theory is the proof of the statement that *the fundamental group of the circle is isomorphic to the group $\mathbb{Z}$ of integers*. It is an example of a problem which in a classical, set theoretic setting, is solved in a way which is very different from the approach which is needed in its type theoretic counterpart. Licata and Shulman [2013] were the first to translate this fact into the setting of homotopy type theory. Another fundamental theorem in homotopy theory is the *theorem of Seifert-van Kampen*. It provides proof that the fundamental groupoid of a pushout of two spaces is isomorphic to the pushout of the fundamental groupoid of these spaces. This is an important help when trying to calculate homotopy groups of spaces which were "glued" together from more primitive spaces using pushouts. Hou (Favonia) and Shulman [2016] documented and formalized in Agda a type theoretic equivalent of the Seifert-van Kampen theorem. In this chapter, we will revisit the proof ideas for both of these results and discover, what they have in common.

The intuitive way to compare the integers with the loop space $\Omega(\mathbb{S}^1)$ of the circle is the following: Each loop can be reached by following the constructor loop a number of times either in its actual direction or by reversing the direction, i. e. concatenating with loop^{-1}. Following an inverted loop after a non-inverted one or vice versa cancels out since we have

$$\mathsf{loop} \cdot \mathsf{loop}^{-1} = \mathsf{refl} = \mathsf{loop}^{-1} \cdot \mathsf{loop}.$$

The integer corresponding to a loop would then be the "exponent" of the number of times we concatenate loop (with $\mathsf{loop}^0 = \mathsf{refl}$). But translating this intuition into a formal proof, especially in order to define a function from the loop space to the integers, requires some tricks.

Theorem 3.3.1 (Loop Space of the Circle, Licata and Shulman [2013]). *The loop space of the circle is equivalent to the type of the integers:* $\Omega(\mathbb{S}^1, \mathsf{base}) \simeq \mathbb{Z}$

Proof. Instead of directly defining a map $\Omega(\mathbb{S}^1, \mathsf{base}) \to \mathbb{Z}$, we perform the type theoretic equivalent of constructing the *universal cover* for the circle. Over each point of the circle, we will have a whole type, i. e. a type family $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ and by the end we will be able to prove that this type family satisfies the following equivalences:

$$(\mathsf{base} = x) \simeq \mathsf{code}(x) \simeq \mathbb{Z}, \tag{3.1}$$

after which we can get the desired result by plugging in $x \equiv \mathsf{base}$. The definition of this type family is determined by the following two equations:

$$\begin{aligned} \mathsf{code}(\mathsf{base}) &\equiv \mathbb{Z} \\ \mathsf{ap}_{\mathsf{code}}(\mathsf{loop}) &= \mathsf{ua}(S), \end{aligned}$$

with $S : \mathbb{Z} \to \mathbb{Z}$ being the successor function on the integers, which is obviously an equivalence. Formally speaking, we can achieve this by setting

$$\mathsf{code} :\equiv \mathsf{elim}_{\mathbb{S}^1}(\lambda\_.\mathcal{U}, \mathbb{Z}, \mathsf{ua}(S)).$$

We can then calculate that

$$\mathsf{loop}^*(z) \equiv (\mathsf{ap}_{\mathsf{code}}(\mathsf{loop}))^*(z) \equiv (\mathsf{ua}(S))^*(z) \equiv S(z), \tag{3.2}$$

and that, by the same reasoning, $(\mathsf{loop}^{-1})^*(z) = P(z)$ where $P = S^{-1}$ is the predecessor function on $\mathbb{Z}$. After we gave the definition of code, the second equivalence of (3.1) is straightforward to prove by induction on x, while the first one is more involved. We call the map into $\mathsf{code}(x)$, which we have to define, the *encoding* map and the inverse the *decoding*.

The encoding works by, for each path, transporting along this type in the type family code, starting from zero:

$$\begin{aligned} \mathsf{encode} &: \{x : \mathbb{S}^1\}(p : \mathsf{base} = x) \to \mathsf{code}(x) \\ \mathsf{encode}(p) &:\equiv p^*(0) \end{aligned}$$

This is the same as defining encode as 0 induction on p. It is easy to check via (3.2) that $\mathsf{encode}(\mathsf{loop})$ calculates to $S(z)$, $\mathsf{encode}(\mathsf{loop} \cdot \mathsf{loop})$ to $S(S(z))$ etc.

Defining the decoding function $\mathsf{decode}(x) : \mathsf{code}(x) \to (\mathsf{base} = x)$ goes by induction on the circle element x involved. For the induction we first have to give the base case $\mathsf{decode}(\mathsf{base}) : \mathsf{code}(\mathsf{base}) \to (\mathsf{base} = \mathsf{base})$ which we set to the function which is constantly loop. We then have to prove that

$$\mathsf{loop}^*(\lambda\_.\,\mathsf{loop}) = (\lambda\_.\,\mathsf{loop}),$$

where the transport is over the type family $\lambda x'.\,\mathsf{code}(x') \to (\mathsf{base} = x')$, a calculation we don't want to detail here. This defines decode. It is then left to prove by induction that

$$\mathsf{encode}(\mathsf{decode})(c) = c \text{ for all } c : \mathsf{code}(x) \text{ and}$$
$$\mathsf{decode}(\mathsf{encode})(p) = p \text{ for all } x : \mathbb{S}^1 \text{ and } p : \mathsf{base} = x.$$

completing the proof of the theorem. □

The proof for the Seifert-van Kampen theorem employs the same method of providing a type family acting as a universal cover. It can be seen as a generalization of the above theorem. Recall that in a classical topological setting the statement of the theorem is the following:

> The fundamental groupoid of the pushout of two spaces is equivalent to the pushout of its fundamental groupoids.

So while the for the theorem about the circle the "right-hand side" of the equivalence was rather clearly defined, here the question is whether we find a good definition of the pushout of groupoids. Since we attempt another example of an encode-decode proof, we might as well first define the morphisms of the pushout of groupoids as the code itself. Intuitively, the morphisms the pushout $P :\equiv M \sqcup^L N$ are "zig-zags" consisting of morphisms alternatingly in M and N, separated by morphisms in L.

Definition 3.3.2 (Codes for Groupoid Pushouts, Hou (Favonia) and Shulman [2016])**.** We define a type family $\mathsf{code} : P \to P \to \mathcal{U}$, by recursion on both arguments. That means that it suffices to give families $M \to M \to \mathcal{U}$, $M \to N \to \mathcal{U}$, $N \to M \to \mathcal{U}$, and $N \to N \to \mathcal{U}$, which agree on $f(l)$ and $g(l)$ for all $l : L$ in either argument. We will omit proof of this consistency.

For the first case, i. e. if we have $m, m' : N$, we want $\mathsf{code}(\mathsf{inr}(m), \mathsf{inr}(m'))$ to be the inductive type where elements are characterized as lists

$$(p_0, x_1, q_1, y_1, p_1, x_2, q_2, y_2, p_2, \ldots, y_k, p_k),$$

$$\begin{tikzcd}
m \arrow[r, "p_0"] & g(x_1) \arrow[d] & g(y_1) \arrow[r, "p_1"] & g(x_2) \arrow[d] & g(y_2) \arrow[r, "p_2"] & m' \\
 & f(x_1) \arrow[r, "q_1"] & f(y_1) \arrow[u] & f(x_2) \arrow[r, "q_2"] & f(y_2) \arrow[u] &
\end{tikzcd}$$

Figure 3.2: “Zig-zag paths” in the pushout codes, vertical arrows correspond to applications of glue, horizontal arrows to applications of $\mathsf{ap}_{\mathsf{inl}}$ and $\mathsf{ap}_{\mathsf{inr}}$.

the entries of which (cf. Figure 3.2) have the following type for each $i : \mathbb{N}$ with $0 < i \leq k$:

$$\begin{aligned}
x_i, y_i &: L \\
p_0 &: \|m = f(x_1)\|_0 \\
p_i &: \begin{cases} \|f(y_i) = f(x_{k+1})\|_0 & \text{for } i < k \\ \|f(y_k) = m'\|_0 & \text{for } i = k \end{cases} \\
q_i &: \|g(x_k) = g(y_k)\|_0.
\end{aligned}$$

These lists have to be quotiented by the path constructors

$$(\ldots, q_i, y_i, \mathsf{refl}_{f(y_i)}, y_i, q_{i+1}, \ldots) = (\ldots, q_i \cdot q_{i+1}, \ldots) \text{ and}$$
$$(\ldots, p_i, x_i, \mathsf{refl}_{g(x_i)}, x_i, p_{i+1}, \ldots) = (\ldots, p_i \cdot p_{i+1}, \ldots),$$

and the resulting type is made a set by the means of truncation. The type families for the three remaining cases are likewise defined “zig-zags” in the form of quotiented lists, where the parity of the length is changed and/or the position of p_i and q_i are swapped.

With this definition, the statement of the theorem can be stated as the following:

Theorem 3.3.3 (Seifert-van Kampen, Hou (Favonia) and Shulman [2016]). *For each $x, y : P$ we have*

$$\|x = y\|_0 \simeq \mathsf{code}(x, y).$$

Proof. Again, we define functions encode and decode constituting the equivalence. The strategy is generally the same as for the characterization of equalities in the circle: The encoding function

$$\mathsf{encode} : \{u, v : P\} \to \|u = v\|_0 \to \mathsf{code}(u, v)$$

can be defined by induction on the truncated type (since we truncated $\mathsf{code}(u, v)$) then on the path $p : u = v$ involved, and setting the function to be the trivial element in the respective type of lists:

$$\mathsf{encode}(|\mathsf{refl}|) :\equiv (\mathsf{refl}).$$

This is the same as transporting p along the type family $\lambda v.\, \mathsf{code}(u, v)$. The decoding function has to take a list and "glue together" the paths in the pushout which it represents, as for example

$$\begin{aligned}&\mathsf{decode}(p_0, x_1, q_1, y_1, p_1, \ldots)\\ :\equiv&\mathsf{ap}_{\mathsf{inl}}(p_0) \cdot \mathsf{glue}(x_1) \cdot \mathsf{ap}_{\mathsf{inr}}(q_1) \cdot \mathsf{glue}(y_1)^{-1} \cdot \mathsf{ap}_{\mathsf{inl}}(p_1) \cdot \ldots\end{aligned}$$

The proof that these functions are inverse to each other is again done by induction. □

Remark 3.3.4 (The classical Seifert-van Kampen Theorem)**.** How does the statement of Theorem 3.3.3 relate to the classical version of the theorem? The left side of the equivalence clearly correponds to morphisms between x and y in the fundamental groupoid (i. e. the equalities) of the pushout of the spaces, but what about the right-hand side? We can show that $\mathsf{code}(x, y)$ is not some arbitrary construction but that its definition as a quotient of lists can be straightforwardly generalized to a characterization of morphisms in the pushout of any two groupoids.

Chapter 4

Path Spaces of Higher Inductive Types

We have seen in the previous chapter that the encode-decode method can be used in a variety of cases when we want to make statements about the equality types – or *path spaces* – of higher inductive types. Going through all necessary steps of such a proof can be somewhat tedious, but part of it is very mechanical work. One main goal of this chapter is to present a different method to directly work with equality types of coequalizers and pushouts (and constructions based on these): Since elimination rules such as the one for coequalizers characterize only the points of the type, but in the constructors we create points and equalities simultaneously, we believe that it is natural to hope for an "induction principle for equalities" which is reminiscent of an elimination rule. More concretely, for our case of a coequalizer $A/\!\!/\!\sim : \mathcal{U}$ of a type $A : \mathcal{U}$ and typal relation $\_ \sim \_ : A \to A \to \mathcal{U}$, let us assume we are given a type family

$$Q : \{a, b : A\} \to [a] = [b] \to \mathcal{U}.$$

Is it possible to have simple-to-check conditions which are sufficient to conclude $Q(q)$ for a general q (instead of just $glue(s)$ for some $s : a \sim b$)?

Remark 4.0.1. Note that Q above quantifies over two elements of A and an equality of $A/\!\!/\!\sim$. If instead we asked the same question for a type family

$$S : (x, y : A/\!\!/\!\sim) \to x = y \to \mathcal{U},$$

the answer would be that we could use the J-rule to populate this family by giving $S(\mathsf{refl}_x)$. The principle we want for the applications we presented is the version where endpoints are "restricted" as above.

It turns out that, like for the J-rule, there is a generalization of the above question. We get this generalization by switching from an *unbased* (or *global*) type family to a *based* (or *local*) one: We can fix one of the two endpoints to be $[a_0] : A /\!/ \sim$ and replace Q by a family which is indexed only *once* over A:

$$P : (b : A) \to [a_0] = [b] \to \mathcal{U}. \tag{4.1}$$

Like for the two versions of the J-rule, a principle answering the based version of the question also answers the unbased one, which is why we will focus exclusively on the former, and we will be able to easily derive the latter from it.

In order to get some intuition for the subtleties of the equality types, let us first look at a hypothetical principle which turns out to be wrong. Usually, induction principles contain one case for every constructor, the standard equality constructor is refl and with Coeq-Intro2, we have one further path constructor glue. Thus, we might try whether it is sufficient to assume terms

$$\begin{aligned} &r : P(a_0, \mathsf{refl}_{[a_0]}) \text{ and} \\ &p : (b : A)(s : a_0 \sim b) \to P(b, \mathsf{glue}(s)) \end{aligned}$$

to conclude that $(b : A)(q : [a_0] = [b]) \to P(b, q)$? It turns out that this attempt fails: Consider the relation $\sim$ on the natural numbers defined by

$$(m \sim n) :\equiv m + 1 = n.$$

We can look at the coequalizer $\mathbb{N} /\!/ \sim$. Let us take $1 : \mathbb{N}$ as the base point and $P : (n : \mathbb{N}) \to ([0] = [n]) \to \mathcal{U}$ defined by $P(n, q) :\equiv (n \geq 1)$. The terms r and p are constructed easily, but at the same time, it is clear that $P(0, \mathsf{glue}(k)^{-1})$ is empty (here, k is a proof for $0 + 1 = 1$).

The above naïve suggestion was easy to disprove, but let us try to understand why it was insufficient. Equalities that come from A can, by the J-rule, be assumed to be refl; these are sufficiently covered. However this is not true for equalities that are generated using the glue constructor. The counterexample uses the fact that we have not explicitely closed them under symmetry and similarity – we could have also used that we have not closed them under transitivity.

How could we fix this? Given an equality q in $A /\!/ \sim$, we can compose it with $\mathsf{glue}(s)$ assuming the endpoints match. This suggests that the induction principle we are looking for should assume $Q(q) \to Q(q \cdot \mathsf{glue}(s))$. But we can also compose with $\mathsf{glue}(s)^{-1}$, suggesting that we also need a function $Q(q) \to Q(q \cdot \mathsf{glue}(s)^{-1})$. The operations of composing with $\mathsf{glue}(s)$ and its inverse should furthermore be inverse to each other, wich motivates us to ask for only *one* of them and require this one to be an equivalence, i. e. $Q(q) \simeq Q(q \cdot \mathsf{glue}(s))$. This finally leads us to a valid induction principle, which is short, useful, and comes with two β-rules. Proving this princple is the main result of this chapter:

Theorem 4.0.2 (Induction for Coequalizer Equality)**.** *Assume A and $\sim$ as before, a point $a_0 : A$, and we are further given a type family*

$$P : (b : A) \to [a_0] = [b] \to \mathcal{U},$$

together with terms

$$r : P(\mathsf{refl}_{[a_0]}) \textit{ and}$$
$$e : \{b, c : A\}(q : [a_0] = [b])(s : b \sim c) \to P(q) \simeq P(q \cdot \mathsf{glue}(s)).$$

The we can construct a dependent function

$$\mathsf{ind}_{r,e} : \{b : A\}(q : [a_0] = [b]) \to P(q)$$

with the following equalities reminiscent of β-rules:

$$\mathsf{ind}_{r,e}(\mathsf{refl}_{[a_0]}) = r \tag{4.2}$$
$$\mathsf{ind}_{r,e}(q \cdot \mathsf{glue}(s)) = e(q, s, \mathsf{ind}_{r,e}(q)). \tag{4.3}$$

Remark 4.0.3. The theorem can be proved in a way which makes the first β-rule hold judgmentally. This is what we have done in our formalization, but we will refrain from checking whether equalities hold strictly in this chapter.

Remark 4.0.4 (Incorrect Principle)**.** foo

In the following sections we will first prove this main result (Chapter 4.1), then modify it to obtain a version for pushouts (Chapter 4.2), and present a few smaller applications (Chapter 4.3) by characterizing the loop space of the circle and proving that pushouts preserve embeddings, before applying the approach to state a version of the Seifert-van Kampen theorem which instead of groupoids refers to *higher* fundamental groupoids (Chapter 4.4). Most of the contents have been formalized in Lean, an effort which we will comment on in Chapter 4.5.

4.1 The Main Theorem: Path Spaces in Coequalizers

We will first formulate and prove the non-dependent version of the main result by developing the corresponding categorical framework inside type theory. This then allows us to derive the induction principle as stated in Theorem 4.0.2.

Using categorical ideas to structure constructions and reason inside type theory is standard. The dependent elimination principle can usually equivalently be formulated as a recursion (or *non-dependent* elimination) principle together with a uniqueness principle, often phrased as a *universal property*. A principled way of doing this is to define objects and morphisms of a category; the statement then is that the inductive type in question is (homotopy) initial in this category. For the specific case of homotopy type theory, the connection between induction and initiality has been shown by Awodey et al. [2017] for inductive types, and by Sojakova [2015] for some higher inductive types.

However, category theory in homotopy type theory is subtle. The "obvious" naïve definition of a category without imposing any truncation levels on objects and morphisms (sometimes called a *wild category*) is not always a well-behaved notion. For example the slice of a wild category is not a wild category anymore. The underlying reason is that the identity and associativity equalities do not behave like laws (or properties) but like higher morphisms in a *higher category* where additional coherences are required. One approach to higher categories in homotopy type theory is discussed by Capriotti and Kraus [2017]. Alternatively, the *univalent categories* by Ahrens et al. [2015] restrict the truncation levels to avoid the issue. For us, truncating is not a suitable strategy since it would not allow us to prove our general result.

Although not well-behaved in general, wild categories are still a useful tool for us. We do *not* think of them as "bad ordinary categories" but instead as an approximation to $(\infty, 1)$-categories, where most of the higher data is omitted. However, since none of our constructions require us to actually *use* the omitted data, we are able to get away with this. Most importantly, we can talk about the concept of homotopy initiality without ever referring to higher morphisms. Technically, we do not even need associativity – it could be excluded from the following definition without consequences for the rest of the section.

Definition 4.1.1 (Wild Categories)**.** A **wild category** $\mathcal{A}$, for simplicity henceforth simply **category**, consists of a type $|\mathcal{A}| : \mathcal{U}$ of objects; for objects $X, Y : |\mathcal{A}|$ a type $\mathcal{A}(X, Y)$ of morphisms; a composition operator $\circ$ and identities of the following obvious types

$$\begin{aligned} \_\circ\_ &: \{X, Y, Z : |\mathcal{A}|\} \to \mathcal{A}(Y, Z) \to \mathcal{A}(X, Y) \to \mathcal{A}(X, Z), \\ \mathsf{id} &: \{X : |\mathcal{A}|\} \to \mathcal{A}(X, X), \end{aligned}$$

together with the two standard equalities for the identities and one equality which states that $\circ$ is associative:

$$\begin{aligned} \mathsf{id} \circ f &= f, \\ f \circ \mathsf{id} &= f, \text{ and} \\ (f \circ g) \circ h &= f \circ (g \circ h). \end{aligned}$$

Initiality is one of the few notions which are still well-behaved in wild categories: We can define it in terms of contractability.

Definition 4.1.2 (Initiality)**.** An object X of a category $\mathcal{A}$ is called **initial** if for every object Y the type of morphisms $\mathcal{A}(X, Y)$ is contractible.

For the whole remainder of the section, let us assume that a type $A : \mathcal{U}$ together with a point $a_0 : A$ and a relation $\_ \sim \_ : A \to A \to \mathcal{U}$ are given. Our main category of interest is the following:

Definition 4.1.3 (Category of Coherent, Pointed Families)**.** The category $\mathcal{C}$ is defined as follows: Objects in $|\mathcal{C}|$ are "pointed type families respecting $\sim$", by which we mean triples (K, r, e) of the types

$$\begin{aligned} &K : A \to \mathcal{U}, \\ &r : K(a_0), \text{ and} \\ &e : \{b, c : A\}(b \sim c) \to K(b) \simeq K(c). \end{aligned}$$

Morphisms are "pointed fibrewise functions". Explicitely, a morphism in $\mathcal{C}((K, r, e), (K', r', e'))$ is a triple (f, γ, δ) with

$$\begin{aligned} &f : (b : A) \to K(b) \to K'(b), \\ &\gamma : f_{a_0}(r) = r', \text{ and} \\ &\delta : \{b, c : A\}(s : b \sim c) \to e'(s) \circ f_b = f_c \circ e(s). \end{aligned}$$

It might be helpful to think of γ as an equality witnessing that, for any $s : b \sim c$ the following square commutes:

$$\begin{tikzcd}
K(b) \arrow[r, "e(s)"] \arrow[d, "f_b"'] & K(c) \arrow[d, "f_c"] \\
K'(b) \arrow[r, "e'(s)"] & K'(c)
\end{tikzcd} \tag{4.4}$$

The remaining components (identities, composition and their equations) are straightforward to define. For example identities are given as

$$(\lambda b.\, \mathsf{id}, \mathsf{refl}_r, \lambda s.\, \mathsf{refl}_{e(s)}),$$

and composition is given by

$$(f', \delta', \gamma') \circ (f, \delta, \gamma) :\equiv (\lambda b.(f_b{}' \circ f_b), \mathsf{ap}_{f'_{a_0}}(\delta) \cdot \delta', \gamma' \circ \gamma),$$

where the last bit is given by pasting two vertically neighboring squares (4.4).

A variation of Theorem 4.0.2, this time not as dependent elimination principle but as a non-dependent recursor, together with uniqueness can now be stated as follows:

Theorem 4.1.4 (Initiality of Coequalizer Equality). *Consider the object (K^i, p^i, e^i) of $\mathcal{C}$, where the first part is given by*

$$K^i(b) :\equiv ([a_0] = [b]),$$

i.e. K^i is given by equality in the coequalizer $A /\!/ \sim$. The point is given by

$$r^i :\equiv \mathsf{refl}_{[a_0]}.$$

For every $s : b \sim c$, the component $e^i(s)$ is the equivalence between $([a_0] = [b])$ and $([a_0] = [c])$ which is given by composition with $\mathsf{glue}(s)$*; we simply write*

$$e^i(s) :\equiv \_ \cdot \mathsf{glue}(s).$$

Then, our statement is: The object (K^i, p^i, e^i) is initial in the category $\mathcal{C}$.

In the following we will first provide a proof of this theorem, requiring various constructions and lemmas, before then deriving the dependent version. In order to prove Theorem 4.1.4, we consider a second category which we call $\mathcal{D}$. We will then show that $\mathcal{C}$ and $\mathcal{D}$ are isomorphic. The point of this is that it is very easy to find the initial object in $\mathcal{D}$, and, via the isomorphism, it can easily be transported to $\mathcal{C}$. A useful technical tool is the formulation of coequalizer induction as an equivalence which is what we start with:

Lemma 4.1.5 (Coequalizer Induction as Equivalence). *Given a type family $P : A /\!\!/ \!\sim\, \to \mathcal{U}$, there is a canonical map from the type*

$$(x : A /\!\!/ \!\sim) \to P(x) \tag{4.5}$$

to the type

$$(f : (a : A) \to P([a])) \times \Big(\{a, b : A\}(s : a \sim b) \to f(a) =_{\mathsf{glue}(s)} f(b)\Big) \tag{4.6}$$

mapping g to the pair $(g \circ [-], \lambda s.\mathsf{apd}_g(\mathsf{glue}(s)))$. This canonical map is an equivalence.

Proof. The standard induction principle for the coequalizer states that there is a function from (4.6) to (4.5) with β-rules that essentially amount to stating that this function is a section of the canonical map above. Lemma 4.1.5 replaces "section" by "inverse". This easily follows from the standard induction principle. We are not the first to make this observation – a small variation of the lemma is already present in the Lean library, formalized by van Doorn and Buchholtz [2017]. □

Remark 4.1.6. Note that Lemma 4.1.5 crucially depends on the "non-recursiveness" of the coequalizer $A /\!\!/ \!\sim$. For example, the analogous statement about the natural numbers $\mathbb{N}$ is false (i. e. assuming it leads to a contradiction).

In line with the strategy outlined above, we further consider the following category $\mathcal{D}$:

Definition 4.1.7 (Category $\mathcal{D}$). $\mathcal{D}$ is the category of pointed families over $A /\!\!/ \!\sim$. Explicitly, objects in $|\mathcal{D}|$ are pairs (L, p) as in

$$\begin{aligned} &L : A /\!\!/ \!\sim\, \to \mathcal{U} \\ &p : L([a_0]), \end{aligned}$$

and morphisms in $\mathcal{D}((L,p),(L',p'))$ are pairs (g,ϵ) of types

$$g : (x : A/\!\!/\!\sim) \to L(x) \to L'(x) \tag{4.7}$$

$$\epsilon : g(p) = p' \tag{4.8}$$

Again, the remaining components of the category are defined in the straightforward way.

The connection between $\mathcal{C}$ and $\mathcal{D}$ is as follows:

Lemma 4.1.8. *The two categories are isomorphic, in the following sense. There is a map*

$$\Phi_0 : |\mathcal{D}| \to |\mathcal{C}| \tag{4.9}$$

which is an equivalence, and there is also a map

$$\Phi_1 : \Pi(X,Y : |\mathcal{D}|).\mathcal{D}(X,Y) \to \mathcal{C}(\Phi_0(X),\Phi_0(Y)) \tag{4.10}$$

such that each $\Phi_1(X,Y)$ is an equivalence. Moreover, identities and compositions are preserved by the equivalence. This corresponds to the two wild categories being isomorphic objects in the category of wild category *which as morphisms contains the obvious notion of "wild functors".*

Proof. Let us unfold the type in (4.9); this is the type of the equivalence that we *want* to construct:

$$\begin{aligned} &(L : A/\!\!/\!\sim \to \mathcal{U}) \times L([a_0]) \\ \simeq\ &(K : A \to \mathcal{U}) \times K(a_0) \times (\{b,c : A\}(s : b \sim c) \to K(b) \simeq K(c)) \end{aligned} \tag{4.11}$$

Lemma 4.1.5 gives us a tool to construct equivalences. Let us use that lemma with the constant type family $P(x) :\equiv \mathcal{U}$; this makes use of the fact that the lemma works on all universe levels. The lemma then gives us, simply by replacing $P(x)$ by $\mathcal{U}$, renaming variables, and using the fact that we are now in the non-dependent special case, the following equivalence φ_0:

$$\begin{aligned} &(A/\!\!/\!\sim \to \mathcal{U}) \\ \simeq\ &(K : A \to \mathcal{U}) \times (\{b,c : A\}(s : b \sim c) \to K(b) = K(c)). \end{aligned} \tag{4.12}$$

Moreover, we know how φ_0 is defined, namely by

$$\varphi_0 :\equiv (L \circ [-], \lambda s.\mathsf{ap}_L(\mathsf{glue}(s))) \tag{4.13}$$

(since we are in the non-dependent case, apd became ap).

We claim that the function Φ_0 of type (4.9) can be constructed from φ_0 via two small modifications:

- First, if we compare the domain of Φ_0 with the domain of φ_0, and furthermore their codomains, we see that the "point-component" is missing from φ_0, i. e. the component of the Σ-type $L([a_0])$ is missing in its domain and $p : K(a_0)$ is missing in its codomain. However, we can ust extend domain and codomain with this component. The Equation (4.13) tells us that this extension is completely trivial, since $K \equiv L \circ [-]$, i. e. we extend φ_0 with the identity on one additional new component.

- The codomain of this extended φ_0 only differes from the codomain of Φ_0 in that the component e in (4.12) concludes $K(b) = K(c)$ instead of $K(b) \simeq K(c)$. To close this gap we can use the canonical function idtoeqv which turns an equality between types into an equivalence. The univalence axiom ensures that it is an equivalence itself.

This concludes the construction of the equivalence Φ_0, and, using (4.13), we can write down how the function part of it computes when applied to a pair (L, p):

$$\Phi_0(L, p) \equiv \big(L \circ [-], p, \lambda s.\mathsf{idtoeqv}(\mathsf{ap}_L(\mathsf{glue}(s)))\big) \tag{4.14}$$

The construction of Φ_1 as in (4.10) is slightly more subtle, since it depends on Φ_0, but it works in essentially the same way. Assume we are given (L, p) and (L', p') in $|\mathcal{D}|$. We unfold the desired type of $\Phi_1((L, p), (L', p'))$, making use of equation (4.14). This gives us the following type which we want to inhabit:

$$\begin{aligned}
& \big(g : (x : A/\!\!/\!\sim) \to L(x) \to L'(x)\big) \times \big(g(p) = p'\big) \\
\simeq \quad & (f : (b : A) \to L([b]) \to L'([b])) \\
& \times (\delta : f(p) = p') \\
& \times \Big(\{b, c : A\}(s : b \sim c) \to \\
& \qquad\qquad \mathsf{idtoeqv}(\mathsf{ap}_L(\mathsf{glue}(s))) \circ f(b) \\
& \qquad\qquad = f(c) \circ \mathsf{idtoeqv}(\mathsf{ap}_{L'}(\mathsf{glue}(s)))\Big)
\end{aligned} \tag{4.15}$$

Let us use Lemma 4.1.5 again, this time with the type family $P(x) :\equiv (L(x) \to L'(x))$. Simply by plugging this into Lemma 4.1.5 and renaming variables we obtain the following equivalence φ_1:

$$\begin{gathered}((x : A/\!\!/\!\sim) \to L(x) \to L'(x)) \\ \simeq \\ (f : (b : A) \to L([b]) \to L'([b])) \times (\{b, c : A\}(s : b \sim c) \to f(b) =_{\mathsf{glue}(s)} f(c))\end{gathered} \tag{4.16}$$

Similar to what we have done before, we have to use (4.16) to derive (4.15); and as before, there are two steps. First, we need to add the equation for the points (i. e. the components ϵ and δ), but this is as simple and direct as before; we will not spell out the details here. Second, and more interestingly, we have to show that the last components of the right hand side of (4.15) and (4.16) coincide in the sense that their types are equivalent. As very often in homotopy type theory when we want to prove something for a specific equality (here $\mathsf{glue}(s)$), the easiest way to do it is to generalize the statement and formulate it in terms of an *arbitrary* equality, which then allows for path induction. The only red herring here is that f is a family of functions. But since it is indexed over A and the equality in question lives in $A/\!\!/\!\sim$, we cannot make use of this. The equivalence follows from Lemma 4.1.9 below, by using $f(b)$ for h, $f(c)$ for k, and $\mathsf{glue}(s)$ for q. It is easy to check that Φ_1 preserves identities and composition of morphisms. □

Lemma 4.1.9. *Let Z be a type, $F, G : Z \to \mathcal{U}$ two type families, $x, y : Z$, and $q : x = y$ a path. Assume we have functions $h : F(x) \to G(x)$ and $k : F(y) \to G(y)$. Then, the path-over type $h =_q k$ is equivalent to the type*

$$\mathsf{idtoeqv}(\mathsf{ap}_G(q)) \circ h = k \circ \mathsf{idtoeqv}(\mathsf{ap}_F(q)).$$

Having shown Lemma 4.1.8, which constitutes the main technical difficulty of the proof of theorem Theorem 4.1.4, we can work with $\mathcal{D}$ instead of $\mathcal{C}$. The benefit is that it is easy to find the initial object of $\mathcal{D}$.

Lemma 4.1.10. *Let us consider the object (L^i, p^i) of $\mathcal{D}$, given as follows:*

$$\begin{aligned} L^i(x) &:\equiv ([a_0] = x) \\ p^i &:\equiv \mathsf{refl}_{[a_0]}. \end{aligned}$$

This object is initial in $\mathcal{D}$.

Proof. Let (L, p) be any other object. After unfolding the definition of morphisms in $\mathcal{D}$, the type $\mathcal{D}((L^i, p^i), (L, p))$ is given by

$$\big(g : (x : A/\!\!/\!\sim) \to ([a_0] = x) \to L(x)\big) \times \big(g([a_0], \mathsf{refl}) = p\big).$$

This type is contractible by applying "singleton contraction" twice: first, we use that an element x together with an equality $[a_0] = x$ form a contractible pair, simplifying the above type to $(g : L([a_0])) \times (g = p)$; and this type is clearly contractible. □

Having found the initial object in $\mathcal{D}$, we transport it to $\mathcal{C}$ in order to prove the categorical version of our main result, which is Theorem 4.1.4.

Proof of Theorem 4.1.4. Since Φ_1, as constructed in Lemma 4.1.8, preserves morphism spaces, Φ_0 preserves the initial object. Thus, all we need to do is to use the object found in Lemma 4.1.10 and compute using (4.14). This gives us K^i and r^i immediately. We obtain the last component e^i by a standard "path induction"-argument. □

As a last step we can now derive the dependent eliminator from the non-dependent one. The main part of this derivation of Theorem 4.0.2 is completely standard and follows known principles, which for example can be found in the work by Awodey et al. [2017]. We use the "total space" construction to turn the dependent case into the non-dependent one. Afterwards, we still need to derive the β-rules, and this is trickier; we use a small trick to "strictify" equations.

Proof of Theorem 4.0.2. Assume that P, r and e are given. The "total space" versions of these three components of an object $(\overline{P}, \overline{r}, \overline{e})$ of the category $\mathcal{C}$, and they are defined as follows:

$$\begin{aligned}
\overline{P} &: A \to \mathcal{U} \\
\overline{P}(b) &:\equiv (q : [a_0] = [b]) \times P(q) \\[1ex]
\overline{r} &: \overline{P}(a_0) \\
\overline{r} &:\equiv (\mathsf{refl}_{[a_0]}, r) \\[1ex]
\overline{e} &: \{b, c : A\} \to (b \sim c) \to \overline{P}(b) \simeq \overline{P}(c) \\
\overline{e}(s) &:\equiv (\_ \cdot \mathsf{glue}(s), e(\_, s, \_))\,.
\end{aligned}$$

Note that the definition of $\overline{e}(s)$ implicitely uses the fact that an equivalence between Σ-types can be constructed from a pair of equivalences for the first and second component. Explicitely, the function part of $\overline{e}(s)$ maps a given pair (q, x) with $q : [a_0] = [b]$ and $x : P(q)$ to the pair $(q \cdot \mathsf{glue}(s), e(q, s, x))$. We have a morphism from the initial object of $\mathcal{C}$ to this newly constructed object which from now on we will call (f, γ, δ), but we also have the "first projection" in the other direction:

$$(f, \delta, \gamma) : \mathcal{C}\big((K^i, p^i, e^i), (\overline{P}, \overline{r}, \overline{e})\big) \tag{4.17}$$

$$(\lambda b.\mathsf{pr}_1, \mathsf{refl}_{r^i}, \lambda s.\mathsf{refl}_{e^i(s)}) : \mathcal{C}\big((\overline{P}, \overline{r}, \overline{e}), (K^i, p^i, e^i)\big) \tag{4.18}$$

It follows from initiality that the composition of these morphisms is the identity on the object (K^i, r^i, e^i), i. e. we have a ψ of the following type:

$$\psi : (\lambda b.\mathsf{pr}_1, \mathsf{refl}_{r^i}, \lambda s.\mathsf{refl}_{e^i(s)}) \circ (f, \delta, \gamma) = (\lambda b.\mathsf{id}, \mathsf{refl}_{r^i}, \lambda s.\mathsf{refl}_{e^i(s)}). \tag{4.19}$$

In particular, given any $q : [a_0] = [b]$, we get an equality

$$\psi^1_q : \mathsf{pr}_1(f_b(q)) = q, \tag{4.20}$$

and we can define $\mathsf{ind}_{r,e}(q) : P(q)$ by

$$\mathsf{ind}_{r,e}(q) :\equiv (\psi^1_q)^*(\mathsf{pr}_2(f_b(q))). \tag{4.21}$$

This defines the induction principle, but the two β-rules still need to be justified. The equality ψ in (4.19) consists of three parts, one for each component; let us write (ψ^1, ψ^2, ψ^3) for them. The general idea is that, just as ψ^1 has allowed us to construct the induction principle (4.21), ψ^2 allows us to show the first β-equation and ψ^3 gives us the second one. The main difficulty here are the many *transports* or *pathovers* which are involved, since the types of ψ^2 and ψ^3 depend on ψ^1. The trick is to split f into (f_1, f_2) by setting $f^1_b :\equiv \mathsf{pr}_1 \circ f_b$ and $f^2_b :\equiv \mathsf{pr}_2 \circ f_b$, and similarly split γ and δ. Using this, and calculating the left side of (4.19), we get

$$(\psi^1, \psi^2, \psi^3) : \ (f^1, \delta^1, \gamma^1) = (\lambda b.\mathsf{id}, \mathsf{refl}_{r^i}, \lambda s.\mathsf{refl}_{e^i(s)}).$$

Now, we can generalize the situation: we claim that, *for all* $(\psi^1, f^1, \ldots)$, we can derive the induction principle plus two β-equalities. This formulation allows us to use based path induction on (f^1, ψ^1) and assume that $f^1 \equiv \lambda b.\mathsf{id}$, $\psi_1 \equiv \mathsf{refl}_{\lambda b.\mathsf{id}}$. This lets the mentioned dependencies disappear and we get $\psi^2 : \delta^1 = \mathsf{refl}_{r^i}$ as well as $\psi^3 : \gamma^1 = \lambda s.\mathsf{refl}_{e^i(s)}$. In addition, (4.21) simplifies to $\mathsf{ind}_{r,e}(q) :\equiv \mathsf{pr}_2(f_b(q))$.

For the first β-equality, we unfold the type of δ:

$$\delta : (\mathsf{refl}_{a_0}, \mathsf{ind}_{r,e}(\mathsf{refl}_{a_0})) = (\mathsf{refl}_{a_0}, r)$$

We need to show that the second components are equal. From δ, we get that the second components are equal when one is transported along the δ^1, and from ψ^2, we get that this is a transport along refl.

The procedure for the second β-equation is similar. The details are best seen by considering the following diagram:

$$\begin{tikzcd}
{[a_0] = [b]} \arrow[r, "f_b"] \arrow[d, "\_ \cdot \mathsf{glue}(s)"'] \arrow[dr, phantom, "\gamma"] & (q : [a_0] = [b]) \times P(q) \arrow[d, "\_ \cdot \mathsf{glue}(s){,} e(\_{,}s{,}\_)"] \\
{[a_0] = [c]} \arrow[r, "f_b"'] & (q : [a_0] = [c]) \times P(q)
\end{tikzcd}$$

γ says that this square commutes. Let us take some $q : [a_0] = [b]$ and see how it is mapped (using $f_1 \equiv \mathsf{id}$ and so on):

$$\begin{tikzcd}
q \arrow[r, mapsto] \arrow[dd, mapsto] & (q, \mathsf{ind}_{r,e}(q)) \arrow[d, mapsto] \\
& (q \cdot \mathsf{glue}(s), e(q, s, \mathsf{ind}_{r,e}(q))) \\
q \cdot \mathsf{glue}(s) \arrow[r, mapsto] & (q \cdot \mathsf{glue}(s), \mathsf{ind}(q \cdot \mathsf{glue}(s)))
\end{tikzcd}$$

Here, γ tells us that the two pairs at the bottom right are equal. As before, we need that their second components are equal; and analogously to what we did before, we use ψ^3 to see that this is the case. □

4.2 Equality in Pushouts

We already learned that pushouts and coequalizers are interderivable, so it is an obvious question what our main theorem translates to when regarded for pushouts instead of coequalizers. We write $\mathsf{in} : (M + N) \to M \sqcup^L N$ for the map given by $(\mathsf{inl}, \mathsf{inr})$. To simplify notation, we keep the inclusions

$\mathsf{inl} : M \to (M + N)$ and $\mathsf{inl} : N \to (M + N)$ implicit and only mention the inclusions into the pushout.

Since pushouts are used a lot and play a vital role in the Seifert-van Kampen theorem (cf. Section 4.4), we want to state our main result explicitly for pushouts instead of coequalizers. The proofs can straightforwardly be obtained by expressing the pushouts as coequalizers, as described in the introduction. (In Lean, this is simply a specialization).

Theorem 4.2.1 (Induction for Pushout Equality). *Assume $L, M, N : \mathcal{U}$, $f : L \to M$, $g : L \to N$ are given as in the definition of the pushout, together with a point $n_0 : N$. Assume we are given families P, Q and terms r, e as follows:*

$$P : \{m : M\} \to (\mathsf{inr}(n_0) = \mathsf{inl}(m)) \to \mathcal{U} \tag{4.22}$$
$$Q : \{n : N\} \to (\mathsf{inr}(n_0) = \mathsf{inr}(n)) \to \mathcal{U} \tag{4.23}$$
$$r : Q(\mathsf{refl}_{\mathsf{inr}(n_0)}) \tag{4.24}$$
$$e : (l : L) \to (q : \mathsf{inr}(n_0) = \mathsf{inl}(f(l))) \to P(q) \simeq Q(q \cdot \mathsf{glue}(l)). \tag{4.25}$$

Then, we can construct terms

$$\mathsf{ind}^P_{r,e} : \{m : M\}(q : \mathsf{inr}(n_0) = \mathsf{inl}(m)) \to P(q)$$
$$\mathsf{ind}^Q_{r,e} : \{n : N\}(q : \mathsf{inr}(n_0) = \mathsf{inr}(n)) \to Q(q)$$

with the following β-rules:

$$\mathsf{ind}^P_{r,e}(\mathsf{refl}_{\mathsf{inr}(n_0)}) = r$$
$$\mathsf{ind}^Q_{r,e}(q \cdot \mathsf{glue}(l)) = e(l, q, \mathsf{ind}^P_{r,e}(q))$$

□

Remark 4.2.2. As before, the first β-rule holds judgmentally in our formalization.

Theorem 4.2.3 (Initiality of Pushout Equality). *Given the same data as in the previous theorem, we can consider the category $\mathcal{P}$, whose definition mirrors that of $\mathcal{C}$. Objects are quadruples (J, K, r, e),*

$$J : M \to \mathcal{U}$$
$$K : N \to \mathcal{U}$$
$$r : K(n_0)$$
$$e : (l : L) \to J(f(l)) \simeq K(g(l))$$

and a morphism between (J, K, r, e) and (J', K', r', e') consists of fiberwise functions which preserve r and commute with e. Then, the object (J^i, K^i, r^i, e^i) defined by

$$\begin{aligned} J^i(m) &:\equiv (\mathsf{inr}(n_0) = \mathsf{inl}(m)) \\ K^i(n) &:\equiv (\mathsf{inr}(n_0) = \mathsf{inr}(n)) \\ r^i &:\equiv \mathsf{refl}_{\mathsf{inr}(n_0)} \\ e^i(l) &:\equiv \_ \cdot \mathsf{glue}(l) \end{aligned}$$

is initial in $\mathcal{P}$. □

4.3 First Applications

We anticipate that our main result, especially in the formulations of Theorem 4.0.2 and Theorem 4.2.1, will be a very useful tool for a variety of constructions in homotopy type theory. In this chapter, we will present two short applications.

The **loop space** $\Omega(X)$ of a type X with an (implicitly given) point $x_0 : X$ is defined to be $x_0 = x_0$. Thus, the loop space of the circle $\mathbb{S}^1$ is simply $\mathsf{base} = \mathsf{base}$. We can use our main result to reprove Theorem 3.3.1 stating that

$$\Omega(\mathbb{S}^1) \simeq \mathbb{Z}.$$

New proof for Theorem 3.3.1. As discussed in Section 3.2, $\mathbb{S}^1$ can be expressed as the coequalizer of **1** and the relation which has **1** as its value. This allows us to apply Theorem 4.1.4 and, since all quantifications are now quantifications over the unit type, we can safely ignore them. Thus, $\big(\Omega(\mathbb{S}^1), \mathsf{refl}, \_ \cdot \mathsf{loop}\big)$ is the initial object in the category of pointed types with an automorphism. Due to the uniqueness of initial objects, all we need is that $(\mathbb{Z}, 0, S)$ is initial in this category. This statement is completely removed from the higher inductive type $\mathbb{S}^1$; it is a basic property of the integers, analogous to the fact that $(\mathbb{N}, 0, S)$ is initial in the category of pointed types with an endofunction. □

Of course, the difficulty of a concrete proof for the initiality property depends on the concrete definition of $\mathbb{Z}$ that one uses. With the definition used by Licata and Shulman (essentially $\mathbb{N} + \mathbf{1} + \mathbb{N}$), this is easy albeit some work. We will come back to definitions of the integers in Remark 4.4.2.

As a second application we will prove that a certain class of functions, called embeddings, is preserved under pushouts.

Definition 4.3.1. An **embedding** is a map $h : X \to Y$ whose fibers are propositions, i. e. where, for each $y : Y$, the type $h^{-1}(y) :\equiv (x : X) \times (y = h(x))$ is a proposition. Equivalently, h is an embedding if and only if

$$\mathsf{ap}_h : \{x, x' : X\} \to (x = x') \to (h(x) = h(x')) \tag{4.26}$$

is a family of equivalences between path spaces.

As formalized by Finster [2017] via an encode-decode construction, embeddings are closed under pushout. In the following, we present an alternative (and significantly shorter) argument.

Theorem 4.3.2 (Pushouts preserve Embeddings)**.** *Embeddings are closed under pushout. Explicitly, if f in following the diagram is an embedding, then so is* inr*:*

$$\begin{tikzcd} L \arrow[r, "g"] \arrow[d, hook, "f"'] & N \arrow[d, hook, dashed, "\mathsf{inr}"] \\ M \arrow[r, dashed, "\mathsf{inl}"'] & M \sqcup^L N \end{tikzcd}$$

Proof. Using (4.26), we need to show that $\mathsf{ap}_{\mathsf{inr}} : (n_0 = n) \to (\mathsf{inr}(n_0) = \mathsf{inr}(n))$ is an equivalence for all points n_0, n. Thus, for any $q : \mathsf{inr}(n_0) = \mathsf{inr}(n)$, we want to find something in the preimage of q. This tells us how we need to choose the type family Q of Theorem 4.2.1: We fix n_0 and define

$$\begin{aligned} Q &: (n : N) \to (\mathsf{inr}(n_0) = \mathsf{inr}(n)) \to \mathcal{U} \\ Q(n, q) &:\equiv (p : n_0 = n) \times \mathsf{ap}_{\mathsf{inr}}(p) = q. \end{aligned}$$

We also need to define the type family P. Given something in M, we "move" it back to N by going via the fiber, which allows us to define P using Q:

$$\begin{aligned} P &: (m : M) \to (\mathsf{inr}(n_0) = \mathsf{inl}(m)) \to \mathcal{U} \\ P(m, q) &:\equiv \Big((l_0, q_0) : f^{-1}(m)\Big) \times Q\big(g(l_0), q \cdot \mathsf{ap}_{\mathsf{inl}}(q_0) \cdot \mathsf{glue}(l_0)\big). \end{aligned}$$

The component r is the obvious one, $r :\equiv (\mathsf{refl}, \mathsf{refl})$. For a given $l : L$ we know that, since f is an embedding, the type $f^{-1}(f(l))$ is contractible and we can assume $(l_0, q_0) \equiv (l, \mathsf{refl})$. This implies $P(f(l), q) \simeq Q(g(l), q \cdot$

$\mathsf{glue}(l))$, which is exactly what we need in order to define the component e. Thus, we have

$$\mathsf{ind}^{Q}_{r,e} : \{n : N\}(q : \mathsf{inr}(n_0) = \mathsf{inr}(n)) \to (p : n_0 = n) \times \mathsf{ap}_{\mathsf{inr}}(p) = q,$$

i. e. a section s of $\mathsf{ap}_{\mathsf{inr}}$ (a function such that $\mathsf{ap}_{\mathsf{inr}} \circ s = \mathsf{id}$). To show that $s \circ \mathsf{ap}_{\mathsf{inr}} : (n_0 = n) \to (n_0 = n)$ is the identity, we do path induction and use the first β-rule. □

4.4 Free Groupoids and a Higher Seifert-van Kampen Theorem

Fundamental groups in topology are *quotients* of spaces – in homotopy type theory we represented them as 0-truncations of equality types. Thus, it is natural to ask for a *higher dimensional* version of the theorem which does not quotient or truncate. In homotopy theory, different versions have been proved by Lurie [2009] and Brown et al. [2011]. In homotopy type theory, it is an open problem how this could be done. The results of this chapter suggest one possible such higher Seifert-van Kampen theorem (Theorem 3.3.3), which we present in this section. Note that the precise formulation of a theorem is part of the open question how to generalize the Seifert-van Kampen theorem in homotopy type theory, since the analogue of the code family by Favonia and Shulman has to be defined (and a trivial solution exists: define this analogue to be the equality). Our justification for why the analogue we suggest is reasonable is that, by 0-truncating, the Favonia-Shulman theorem can be recovered relatively easily. As in Section 4.2, let as assume $L, M, N : \mathcal{U}$, $f : L \to M$, and $g : L \to N$. We write $P :\equiv M \sqcup^L N$. A caveat is in order. In this section, we make used of *indexed higher inductive types*, of which we expect that they can be encoded in terms of higher inductive types, analogously to the fact that indexed inductive types can always be encoded via inductive types, as proven by Altenkirch et al. [2015], Sattler [2015].

To recall, the Seifert-van Kampen theorem in the version of Hou (Favonia) and Shulman [2016] states that for $x, y : M + N$ there is an equivalence

$$\|\mathsf{in}(x) =_P \mathsf{in}(y)\|_0 \simeq \mathsf{code}(x, y).$$

The central difficulty of a higher version of the theorem is, of course, avoiding the set-truncation. Note that, in our description of the the lists used to define code, the set-truncations in

$$p_i : \|g(l_i) = g(k_{i+1})\|_0 \text{ and}$$
$$q_i : \|p(k_i) = p(l_i)\|_0$$

can be removed since we set-truncate later when taking the set-quotient. This is essentially a repeated application of the equivalence

$$\|(a : A) \times \|B(a)\|_n\|_n \simeq \|(a : A) \times B(a)\|_n.$$

This unnecessary set-truncation *does* make sense in the formulation of the Seifert-van Kampen theorem, where all equality types are set-truncated, but removing it makes it easier to motivate our *higher* Seifert-van Kampen theorem.

Next, we suggest an alternative definition for the type of lists (before quotienting/truncation). To simplify things further, let us fix $n_0 : N$ and consider lists starting at this point. Let us now look at the following *indexed* inductive type $C_0 : (M + N) \to \mathcal{U}$ with three constructors, where $C_0(x)$ should be understood as a type of lists from n_0 to x. Recall that we keep the embeddings $i_1 : M \to (M + N)$ and $i_2 : N \to (M + N)$ implicit, and that the data we are given are maps $f : L \to M$ and $g : L \to N$.

$$C_0\text{-}\textsc{Intro1}\ \frac{}{\mathsf{nil} : C_0(n_0)}$$

$$C_0\text{-}\textsc{Intro2}\ \frac{l : L \qquad c : C_0(f(l))}{\mathsf{gl}(l, c) : C_0(g(l))} \qquad C_0\text{-}\textsc{Intro3}\ \frac{l : L \qquad c : C_0(g(l))}{\mathsf{gl}'(l, c) : C_0(f(l))}$$

Clearly, nil gives us the empty list. The other two constructors allow us to switch between lists ending in a point in M to lists ending in a point in N and vice versa. Intuitively, this is done simply by adding a glue at the end of the list. This explains how to add the *vertical* lines of a list as drawn in Figure 3.2. It may be surprising that we do not add the *horizontal* components p_i and q_i explicitly. The reason is that they are automatically and implicitly present in this encoding: the map $(\_)^*$ of type

$$\{l, l' : L\} \to (g(l) = g(l')) \to \big(C_0(g(l)) \to C_0(g(l'))\big) \tag{4.27}$$

allows us to "insert" the upper horizontal components in Figure 3.2 and (exchanging g by f) also the lower horizontal components.

The type $C_0(x)$ encodes lists from n_0 to x, but we have not done the quotienting, i.e. lists that should be the same are still different. To remedy this, we can turn C_0 into an indexed *higher* inductive type and add constructors ensuring that $\mathsf{gl}(l, \mathsf{gl}'(l, x)) = x$ and $\mathsf{gl}'(l, \mathsf{gl}(l, x)) = x$. If we set-truncate, this would give us the correct type, namely something equivalent to the $\mathsf{code}(n_0, x)$ by Favonia and Shulman. Since we do not want to set-truncate, we have to be more careful. $\mathsf{gl}(l)$ and $\mathsf{gl}'(l)$ together with the equality constructors will form a pair of *quasi-inverses* [Homotopy Type Theory, 2013], and it is known that this type is not well-behaved. Instead, we mirror the components that form an actual equivalence. Although there are several formulations that would work, we use those that turn gl into a *bi-invertible map* [Homotopy Type Theory, 2013], as with the following introduction rules for a type family $C : (M + N) \to \mathcal{U}$:

$$C\text{-}\textsc{Intro}1\ \frac{}{\mathsf{nil} : C(n_0)} \qquad C\text{-}\textsc{Intro}2\ \frac{l : L \qquad c : C(f(l))}{\mathsf{gl}(l, c) : C(g(l))}$$

$$C\text{-}\textsc{Intro}3\ \frac{l : L \qquad c : C(g(l))}{\mathsf{linv}(l, c) : C(f(l))} \qquad C\text{-}\textsc{Intro}4\ \frac{l : L \qquad c : C(g(l))}{\mathsf{rinv}(l, c) : C(f(l))}$$

$$C\text{-}\textsc{Intro}5\ \frac{l : L \qquad c : C(f(l))}{\mathsf{linv}(l, \mathsf{gl}(l, x)) = x} \qquad C\text{-}\textsc{Intro}6\ \frac{l : L \qquad c : C(g(l))}{\mathsf{gl}(l, \mathsf{rinv}(l, y)) = y}$$

This definition of C does certainly not look very appealing, and we only give this presentation because it is the "standard" way of presenting higher inductive types. If we allow ourselves to fold the last five constructors into a single one, its introduction rule could look as follows:

$$C\text{-}\textsc{Intro}2\text{–}6\ \frac{l : L}{C(f(l)) \simeq C(g(l))}$$

It may also be interesting to do this in the formulation for a coequalizer instead of a pushout. As explained in Section 4.1, this is a completely mechanical translation. Thus, assume A with $a_0 : A$ and $\_ \sim \_$. Then, the corresponding type G in the "folded" form looks as follows:

$$\begin{aligned}
&\mathsf{data}\ G : A \to \mathcal{U} \\
&\quad \mathsf{nil} : G(a_0) \\
&\quad \mathsf{cons} : \{b, c : A\} \to (b \sim c) \to G(b) \simeq G(c)
\end{aligned} \tag{4.28}$$

Let us write $\mathfrak{G}(a_0, \_)$ instead of $G(\_)$, in order to explicitly mention the point a_0. We can call $\mathfrak{G}$ the *free higher groupoid* generated by $\sim$. This construction generalizes the explicit construction of a free higher group (based on an idea by Kraus and Altenkirch [2018]). It also generalizes the "integer type as a higher inductive type" (itself a special case of the free higher group) which was independently suggested by Pinyo and Altenkirch [2018] (based on Capriotti's idea), by van der Weide et al. in unpublished work, and in a formalization by Cavallo and Mörtberg [Dec 2018]. This example is discussed further in Remark 4.4.2 below.

The type family C depends on the chosen point n_0. To remove this dependency, let us consider a version of C which is indexed twice over $(M + N)$: we write $\mathfrak{C}(x, y)$ for the type which is defined by induction on x to be $\mathfrak{C}(n_0, y) \equiv C(y)$ for $n_0 : N$ or the obvious modification of the first introduction rule of C for $\mathfrak{C}(m_0, y)$ for $m_0 : M$. This expression plays the role of code in our higher analogue of the Favonia-Shulman result, Theorem 3.3.3. While it can be extended to a family $P \to P \to \mathcal{U}$ in a straightforward way, we choose the following formulation for simplicity (and to match Theorem 3.3.3 more closely):

Theorem 4.4.1 (Higher Seifert-van Kampen Theorem)**.** *For $x, y : M + N$, we have an equivalence:*

$$(\mathsf{in}(x) =_P \mathsf{in}(y)) \simeq \mathfrak{C}(x, y). \tag{4.29}$$

Proof. Like all (indexed/higher/ordinary) inductive types, our type family C is initial in an appropriately formulated category of algebras (see Awodey et al. [2017], Coquand et al. [2018], and others). Here is where we draw the connection with the main result of the paper: The category in which C is initial is the category $\mathcal{P}$ from Theorem 4.2.3.[1] This is easy to see when we use the general specification and definition of higher inductive-inductive types given by Kaposi and Kovács [2018b, 2019], but see Remark 4.4.2 below.

By the uniqueness of the initial object and by Theorem 4.2.3, $C(x)$ is equivalent to $\mathsf{inr}(n_0) =_P \mathsf{in}(x)$. Letting n_0 vary, we get the statement of the theorem. □

It is relatively straightforward to recover the set-truncated SvK statement from the higher version (Theorem 4.4.1). We can simply set-truncate both sides in (4.29) and then prove that $\|\mathfrak{C}(x, y)\|_0$ is equivalent to $\mathsf{code}(x, y)$ by constructing maps in both directions.

[1]To be precise, the object $(\mathsf{C} \circ i_1, \mathsf{C} \circ i_2, \mathsf{nil}, \mathsf{gl})$ is initial in P.

Remark 4.4.2. Theorem 4.4.1 and its proof deserve additional comments. We think it is fair to say that the formal theory of indexed higher inductive types is not yet well-established, but it is under very active development. Kaposi and Kovács [2018b, 2019] have suggested a definition for general higher inductive-inductive types which captures the case we need. Indexed higher inductive types are considered in some of the cubical settings; cf. Cavallo and Harper [2019], and there are plans to extend cubical Agda [Vezzosi, 2018, Mörtberg, 2018, Mörtberg and Vezzosi, 2018] and redtt [Angiuli et al., 2018] with the concept (at the time of writing, a possibly not final version is available in cubical Agda). Our definition of C would be covered by any potential implementation of indexed higher inductive types. We think it would be desirable to also allow a direct syntactical representation as in C-Intro2-6, even if only as syntactic sugar for the rules it replaces. Note that Kaposi and Kovács allow equalities between types, which is very similar to allowing this family of equivalences.

The critical step in the above proof of Theorem 4.4.1 is to establish C as the initial object of the category $\mathcal{P}$. With the specification suggested by Kaposi and Kovács allowing C-Intro2-6, with equalities instead of equivalences, this part is easy. However, we want to emphasize that the initiality of C using C-Intro2-6 is not immediate at all if we use what we could call the *direct induction principle* [2]. The direct induction principle is the "standard" principle one derives by giving one case for each constructor, as done in the book [Homotopy Type Theory, 2013] and by current proof assistants such as cubical Agda. Unfortunately, due to the type dependency in the direct induction principle, it becomes very hard to "fold" the components for the type C in order to achieve the principle one would expect from the constructor C-Intro2-6. We expect that implementing Theorem 4.4.1 in cubical Agda would be extremely tedious for this reason.

The core of the problem with the direct induction principle is that it does not allow us to "reason on the level of constructors". As an example, let us consider the interval with two point constructors and one path constructor. If we can reason on the level of constructors, it is by "singleton contraction" clear that one point and the path constructor form a contractible pair, and that the interval is therefore equivalent to the type generated by a single point. With the direct induction principle, this style of reasoning is not possible. It turns out to be easy enough to prove the

[2]The terminology was suggested by Anders Mörtberg.

interval contractible, but in other cases, the situation is less fortunate.

As an example, proposals by Pinyo and Altenkirch [2018], unpublished work by van der Weide et al., and a formalization by Cavallo based on a remark by Mörtberg [Cavallo and Mörtberg, Dec 2018] suggest to define $\mathbb{Z}$ as a higher inductive type, and their very definition is chosen such that $\mathbb{Z}$ should become the initial object of the category of pointed types with automorphism (cf. Section 4.3). Their definitions are versions of (4.28) with A and $\sim$ replaced by the unit type and the relation constantly unit. Crucially, they have to "unfold" the constructor cons, since this is what the current cubical proof assistants require. It turns out that this makes it extremely tedious to prove the resulting type equivalent to other definitions of the integers.

4.5 Formalization in Lean

Not all, but most results of this chapter have been formalized in the theorem prover Lean:

- We formalized the two wild categories $\mathcal{C}$ and $\mathcal{D}$ of Definition 4.1.3 and Definition 4.1.7 as structures,
- we proved their equivalence as per Lemma 4.1.8,
- we provided the initial element of $\mathcal{C}$ as in Lemma 4.1.10,
- we use it to prove the non-dependent version of the main result, Theorem 4.1.4,
- and then derive the dependent eliminator from the uniqueness of the non-dependent one, as in the proof of Theorem 4.0.2,
- we specialize the result to pushouts, which in Lean, are a special case of coequalizers,
- we formalize the two applications of the theorem as given in Section 4.3.

The higher Seifert-van Kampen was not formalized, since we don't have indexed higher inductive types at our disposal in Lean.

We will now give a few code snippets as examples for how the contents are represented in Lean. The formalization makes heavy use of Lean's homotopy type theory library [van Doorn et al., 2017]. The snippets contain more syntax than what we introduced in Section 2.4.1, but we chose not to "sanitize" the actual formalization and refer the reader to Lean's complete introduction [Avigad et al., 2015].

The wild categories $\mathcal{C}$ and $\mathcal{D}$ are encoded as structures, the definition of objects and morphisms in the latter looking as follows:

```
@[hott] protected structure CatD_ob :=
  (L : D → Type w')
  (n : L (ι x))

@[hott] protected structure CatD_mor (X Y : CatD_ob) :=
  (L : Π d, X.L d → Y.L d)
  (n : L _ X.n = Y.n)
```

Note, that `@[hott]` is a user-define command which makes sure that the strict universe of propositions is not used in any way, and thus our formalization is indeed consistent with homotopy type theory.

The equivalence of $\mathcal{C}$ and $\mathcal{D}$ is, on objects defined and stated as follows (we omit the proof itself here):

```
@[hott] private def CatCD_ob (X : CatC_ob) : CatD_ob :=
⟨hott.quotient.elim X.K X.c, X.n⟩

@[hott] private def CatDC'_ob (X : CatD_ob) : CatC_ob :=
⟨X.L ∘ ι, X.n, λ y z r, ap X.L (eq_of_rel R r)⟩

@[hott] private def CatCD_ob_equiv : CatC_ob ≃ CatD_ob := ...

@[hott] private def CatCD_mor_equiv (X Y : CatD_ob)
  : CatD_mor X Y ≃ CatC_mor (CatDC'_ob X) (CatDC'_ob Y) := ...
```

The initial object of $\mathcal{C}$ is then defined by first giving the initial object of $\mathcal{D}$ and then mapping it to $\mathcal{C}$:

```
@[hott] private def CatD_init : CatD_ob := ⟨λ d, ι x = d, idp⟩
```

The non-dependent eliminator is manifested in the following:

```
section elim'
parameters {Q : A → Type (max u v)}
  (x : A)
  (Qrefl : Q x)
  (Qeq : Π y z (r : R y z), Q y = Q z)
include Qrefl Qeq

@[hott] def Q_obj' : CatC_ob x := ⟨Q, Qrefl, Qeq⟩

@[hott, elab_as_eliminator] protected def path_elim' (y : A) (p : ι x = ι y)
  : Q y := ...
```

The version of the dependent eliminator for pushouts is obtained by a simple specialization, as visible in the following snippet:

```
section
  parameters (x : B ⊎ C)
    (Q : Π (y : B ⊎ C), ι x = ι y →  Type w')
    (Qrefl : Q x idp)
    (Qcons : Π (a : A) (p : ι x = inl (f a)),
      Q (sum.inl (f a)) p ≃ Q (sum.inr (g a)) (p ⬝ glue a))
  include Qrefl Qcons

@[hott] private def prel : B ⊎ C → B ⊎ C → Type _ :=
@hott.pushout.pushout_rel A B C f g

@[hott] private def Qcons' (y z : B ⊎ C) (p : ι x = ι y) (r)
  : Q y p ≃ Q z (p ⬝ eq_of_rel prel r) :=
by { hinduction r with a; apply Qcons }

@[hott] protected def path_rec (y : B ⊎ C) (p : ι x = ι y) : Q y p :=
begin
  refine quotient.path_rec x Qrefl _ y p,
  exact Qcons' f g x Q Qrefl Qcons,
end
```

To emphasize the benefits of our theorem when it comes to allowing for short proofs of homotopy theoretic statements, let us look at the proof of the theorem stating that pushouts preserve embeddings (Theorem 4.3.2) in full:

```
parameters {A : Type u} {B : Type v} {C : Type w} (f : A → B) (g : A → C)
  [is_embedding f]
```

```
include f g

@[hott] def motive {c₀} :
  Π x (q : (inr c₀ : pushout f g) = quotient.class_of _ x), Type _
| (sum.inl b) q := Σ (a : fiber f b),
                     fiber (ap inr) (q ⬝ ap inl a.2⁻¹ ⬝ (glue a.1))
| (sum.inr c) q := fiber (ap inr) q

@[hott] def fib_rec (c₀ c q) : @motive c₀ (sum.inr c) q :=
let Qcons : Π a (p : inr c₀ = inl (f a)),
            motive (sum.inl (f a)) p ≃ motive (sum.inr (g a)) (p ⬝ glue a) :=
 λ a p, @sigma_equiv_of_is_contr_left _
    (λ (a : fiber f (f a)), fiber (ap inr) (p ⬝ ap inl (a.2)⁻¹ ⬝ glue (a.1)))
    (is_contr_of_inhabited_prop ⟨a, idp⟩) in
pushout.path_rec f g _ motive ⟨idp, idp⟩ Qcons _ q

@[hott] protected def preserves_embedding : is_embedding inr :=
λ c c', adjointify _ (λ q, (fib_rec c c' q).1)
  (λ p, (fib_rec c c' p).2) (λ p, by { hinduction p, refl })
```

Chapter 5

Specification of Inductive-Inductive Types

As we have mentioned in Section 1.3, we want to make a sharp pivot at this point of the thesis and want to explore *inductive-inductive types*. This chapter will first introduce inductive-inductive types by a few illustrative examples, before we will start with its main purpose: Giving a formal specification of what inductive-inductive types are.

The treatment of inductive-inductive types, which includes all the subsequent chapters, will happen in a type theory which admits the K-rule (cf. Remark 2.3.1) and thus we will not have any meaningful higher equalities in types. While some of the constructions are easily transferable to homotopy type theory, others rely on the K-rule and it is not obvious how to make them *coherent* with respect to higher equalities. Further than having the K-rule we will also take the liberties in regarding some equalities as *strict*. We will use $\equiv$ instead of $=$ to denote strict equality.

Inductive-inductive types are a feature of type theories which comes naturally in provers like Agda, which are based on dependent pattern matching. A formal treatment of their syntax and semantics is still interesting for several reasons: We like to give grounds to their soundness, we might want to implement them in theorem provers which are not based on dependent pattern matching but on a derivation of elimination rules, or we might use them as a tool to prove the consistency of concepts like dependent pattern matching.

The use case for inductive-inductive types is always of the following nature: We want to define some data by the means of an inductive types, but we want its point constructors to refer to data from another type family which is indexed over the very type we are just defining. In the following, we will take a look at a few examples which we are going to revisit at various steps throughout this presentation:

Example 5.0.1 (Type Theory Syntax)**.** Internalising type theory in itself has been a useful tool for many insights about type theory. Danielsson [2006] used induction to achieve this, while later Altenkirch and Kaposi [2016] showed how to internalise the syntax of type theory inside type theory itself using a quotient inductive-inductive type. Leaving out terms and substitutions, we arrive at a fragment of type theoretical syntax specifying a type of contexts and a type of types over a certain contexts, together with type formers for a unit type and a Π-type: We want $\mathit{Con} : \mathcal{U}$ to be inductively defined by

$$\begin{aligned} \mathit{nil} &: \mathit{Con} \text{ and} \\ \mathit{ext} &: (\Gamma : \mathit{Con})(A : \mathit{Ty}(\Gamma)) \to \mathit{Con}\,, \end{aligned}$$

while simultaneously defining a family $\mathit{Ty} : \mathit{Con} \to \mathcal{U}$ with constructors

$$\begin{aligned} \mathit{unit} &: (\Gamma : \mathit{Con}) \to \mathit{Ty}(\Gamma) \text{ and} \\ \mathit{pi} &: (\Gamma : \mathit{Con})(A : \mathit{Ty}(\Gamma))(B : \mathit{Ty}(\mathit{ext}(\Gamma, A))) \to \mathit{Ty}(\Gamma). \end{aligned}$$

Example 5.0.2 (Free Dense Completion)**.** Nordvall Forsberg [2013] proposed the example of a "free dense completion" of an order (or, more general, any relation) which for any type $A : \mathcal{U}$ and any type valued relation $\_ < \_ : A \to A \to \mathcal{U}$ on A freely adds midpoints to all pairs of related elements by $\_ < \_$. It does so by introducing a new type $A' : \mathcal{U}$ inductively generated by the original points and their midpoints:

$$\begin{aligned} \iota_A &: A \to A' \text{ and} \\ \mathit{mid} &: \{x, y : A'\}(p : x <' y) \to A'. \end{aligned}$$

But since our relation was only defined on A, we have to extend it to A' by postulating

$$\begin{aligned} \iota_< &: \{a, b : A\}(p : a < b) \to \iota_A(a) <' \iota_A(b), \\ \underset{l}{\mathit{mid}} &: \{x, y : A'\}(p : x <' y) \to x < \mathit{mid}(p), \text{ and} \\ \underset{r}{\mathit{mid}} &: \{x, y : A'\}(p : x <' y) \to \mathit{mid}(p) < y. \end{aligned}$$

5.1 Signatures for Inductive-Inductive Types

Inductive-Inductive Types are specified by giving a context in a small type theoretic syntax which we will refer to as *source type theory*. This idea originates from the work by Kaposi and Kovács [2018a] on the syntax of *higher* inductive-inductive types, which we adapt and rid of equality constructors to only allow for inductive-inductive types. In contrast to their presentation we will leave the context of the ambient type theory implicit and, instead of highlighting syntax of the ambient type theory, mark elements of the source type theory in green. The only technical difference between our type theory and this proposed source type theory is that we want an explicit separation of sort and point constructors, which prohibit some of the possible types which are possible in Kaposi and Kovács [2018a]. This separation is achieved by introducing *kinds* (see below).

We assume that the source type theory makes use of the standard syntax of type theory, using contexts, types, terms, and variables. We regard the presentation to be *intrinsic* meaning that we will only ever consider wellformed contexts, types, and terms. We will nevertheless use the turnstile notation of syntax, and, for example write $\vdash \Gamma$, to say that Γ is a context – instead of writing something like $\Gamma : Con$. We hope that this makes it less confusing to the reader since we will also have type theoretical syntax as one of our *main examples* for inductive-inductive types.

Types and terms are uniquely ascribed to one of two *kinds*: Either their kind is S which indicates that the type contains sort constructors, or their kind is P because elements of it describe point constructors. We will write $\Gamma \vdash A :: k$ to say that A is a type of kind k and $\Gamma \vdash t : A :: k$ to state that t is a term of the type A which in turn has kind k. Often, we will omit the annotation of the sort, meaning that a judgment is to hold true for both S and P, or that the kind of a term's type has already been specified.

It's important that contexts can be extended by sort and point types in any order to be able to capture sorts which depend on previously defined point constructors. So we have the usual two rules for context formation:

$$\vdash \cdot \qquad \frac{\Gamma \vdash A :: k}{\vdash \Gamma, A}$$

We need one atomic building block for sort types: For plain types and the codomain of function types we need a type $\mathcal{U}$ which serves as a token for the *universe*. We will call terms of this universe "small types". Positiviy requires that these are the only (internal) types which are allowed in the domain of functions. An operation El reifies these small types to big types, making our version of universe what is commonly referred to as “Tarski-style universe” (cf. Luo [2012]):

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U} :: \mathsf{S}} \qquad \frac{\Gamma \vdash a : \mathcal{U}}{\Gamma \vdash \mathsf{El}(a) :: \mathsf{P}}$$

For sorts which are type families over other sorts that we seek to define, and for constructors which recursively refer to other constructors, we need Π-types which have a small type as their codomain. One example for this is the successor constructor of the natural number, which we will see in Example 5.1.4. To distinguish them from the other function types, which we will introduce below, we will often refer to them as *recursive* Π-*types*. Note that whether we want to build a sort or a point type only depends on the kind of the *codomain* of such a Π-type. To eliminate from Π-types we want a rule for its *application* which turns a term of a Π-type into a term of its codomain. Note that there is no introduction rule in the form of λ terms for these functions, since they are not needed in the description of inductive-inductive types.

$$\frac{\Gamma \vdash a : \mathcal{U} \qquad \Gamma, \mathsf{El}(a) \vdash B :: k}{\Gamma \vdash \Pi(a, B) :: k} \qquad \frac{\Gamma \vdash f : \Pi(a, B)}{\Gamma, \mathsf{El}(a) \vdash \mathsf{app}(f) : B}$$

Additionally, we want sorts to be able to be parametrised by previously defined types which are not part of the signature itself. The same goes for point constructors. Since this cannot be captured using the previous Π-type, we will do the obvious and just introduce another type former for this occasion. We will usually call it *external* or *non-recursive* function type. Note that external functions must have a fixed kind. This is to prevent a function which, depending on the input returns sometimes a sort and sometimes a point constructor. An example for an external Π-type can be seen when considering the inductive family of vectors, depending on an external type of natural numbers (see Example 5.1.5).

$$\frac{T : \mathcal{U} \qquad (\tau : T) \to \Gamma \vdash B(\tau) :: k}{\Gamma \vdash \hat{\Pi}(T, B) :: k} \qquad \frac{\Gamma \vdash f : \hat{\Pi}(T, B) \qquad \tau : T}{\Gamma \vdash f(\tau) : B(\tau)}$$

Since we are working with explicit substitutions, we need to postulate a calculus for substitutions $\Gamma \xrightarrow{\sigma} \Delta$ between any two contexts Γ and Δ. The substitutions should form a category as postulated by the following rules:

$$\frac{\vdash \Gamma}{\Gamma \xrightarrow{\mathsf{id}} \Gamma} \qquad \frac{\Delta \xrightarrow{\sigma} \Sigma \qquad \Gamma \xrightarrow{\delta} \Delta}{\Gamma \xrightarrow{\sigma \circ \delta} \Sigma}$$

$$\begin{aligned} \mathsf{id} \circ \sigma &= \sigma \\ \sigma \circ \mathsf{id} &= \sigma \\ (\sigma \circ \delta) \circ \gamma &= \sigma \circ (\delta \circ \gamma) \end{aligned}$$

We can pull back types and terms along substitutions, and these pull-backs are functorial in the categorical structure:

$$\frac{\Delta \vdash A :: k \qquad \Gamma \xrightarrow{\sigma} \Delta}{\Gamma \vdash A[\sigma] :: k} \qquad \frac{\Delta \vdash t : A \qquad \Gamma \xrightarrow{\sigma} \Delta}{\Gamma \vdash t[\sigma] : A[\sigma]}$$

$$\begin{aligned} A[\mathsf{id}] &= A \\ A[\sigma \circ \delta] &= A[\sigma][\delta] \\ t[\mathsf{id}] &= t \\ t[\sigma \circ \delta] &= t[\sigma][\delta] \end{aligned}$$

We have a canonical substitution into the empty context and we can extend substitutions by giving a term in a type pulled back to their domain.

$$\frac{\vdash \Gamma}{\Gamma \xrightarrow{\epsilon} \cdot} \qquad \frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \Delta \vdash A \qquad \Gamma \vdash t : A[\sigma]}{\Gamma, A \xrightarrow{\sigma, t} \Delta}$$

Empty substituition and extension simplify by the following laws:

$$\begin{aligned} \sigma &= \epsilon && \text{for all } \Gamma \xrightarrow{\sigma} \cdot\text{, and} \\ (\delta, t) \circ \sigma &= (\delta \circ \sigma), t[\sigma] && \text{for } \Gamma \xrightarrow{\delta} \Delta, \Sigma \xrightarrow{\sigma} \Gamma. \end{aligned}$$

A substitution into an extended context allows us to project out a “shorter” substitution and a term as a terminal component:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta, A}{\Gamma \xrightarrow{\pi_1(\sigma)} \Delta} \qquad \frac{\Gamma \xrightarrow{\sigma} \Delta, A}{\Gamma \vdash \pi_2(\sigma) : A[\pi_1(\sigma)]}, \text{ with}$$

$$\begin{aligned}\pi_1(\sigma, t) &= \sigma,\\ \pi_2(\sigma, t) &= t, \text{ and}\\ (\pi_1(\sigma), \pi_2(\sigma)) &= \sigma.\end{aligned}$$

Finally, we also need rules that tell us, how the constructors of the universe and the Π-types behave when pulled back along an arbitrary substitution $\Gamma \xrightarrow{\sigma} \Delta$:

$$\begin{aligned}\mathcal{U}[\sigma] &= \mathcal{U},\\ \mathsf{El}(a)[\sigma] &= \mathsf{El}(a[\sigma]),\\ \Pi(a, B)[\sigma] &= \Pi(a[\sigma], B[\sigma \wedge \mathsf{El}(a)]),\\ \mathsf{app}(f)[\sigma \wedge El(a)] &= \mathsf{app}(f[\sigma]),\\ \hat{\Pi}(T,\, B)[\sigma] &= \hat{\Pi}(T,\, \lambda\tau.B(\tau)[\sigma]), \text{ and}\\ f(\tau)[\sigma] &= f[\sigma](\tau).\end{aligned}$$

This concludes the specification of the syntax for inductive-inductive types.

Definition 5.1.1. Above, $\sigma \wedge A$ is one of several auxiliary constructions on the syntax which are helpful when dealing with substitutions and which can be derived from the other rules.

The first one is the operation known as *weakening* which for any $\Gamma \vdash A$ gives a substitution $\Gamma, A \xrightarrow{\mathsf{wk}} \Gamma$ from the extended into the original context by $\mathsf{wk} := \pi_1(\mathsf{id})$.

Likewise, we can apply the second projection to the identity substitution such that whenever $\Gamma \vdash A$, we have the first *variable* of the context $\mathsf{vz} := \pi_2(\mathsf{id})$ with $\Gamma, A \vdash \mathsf{vz} : A[\mathsf{wk}]$. Transporting a term $\Gamma \vdash t : A$ along the weakening substitution defined above for any $\Gamma \vdash B$, we get $\Gamma, B : t[\mathsf{wk}] : A[\mathsf{wk}]$. We will write $\mathsf{vs}(t) := t[\mathsf{wk}]$ for this variable. Together, vz and vs form *typed de Bruijn indices* to select variables from a context via numbering them with a zero (vz) and a successor (vs).

For $\Gamma \xrightarrow{\sigma} \Delta$ and $\Gamma \vdash A$ we can "lift" σ along A to get a substitution $\Gamma, A[\sigma] \xrightarrow{\sigma \wedge A} \Delta, A$. This operation can be defined by

$$\sigma \wedge A := \sigma \circ \mathsf{wk}, \mathsf{vz}(A[\sigma]).$$

At last, every term $\Gamma \vdash t : A$ gives rise to a substitution $\Gamma \xrightarrow{\langle t \rangle} \Gamma, A$, representing the extension of Γ by t, via

$$\langle t \rangle := id, t.$$

Definition 5.1.2 (Application)**.** Most often, we want to use the application of the inductive function type in the form where we give the function term and its input separately, as in the following rule:

$$\frac{\Gamma \vdash f : \Pi(a, B) \qquad \Gamma \vdash u : \mathsf{El}(a)}{\Gamma \vdash f(u) : B[\sigma]},$$

for some substitution σ. Now we know that we can define this substitution by

$$f(u) := \mathsf{app}(f)[\langle u \rangle],$$

and $\sigma = \langle u \rangle$.

Remark 5.1.3. The syntax with its postulated equations itself on the one hand shows features of induction-induction itself – types are dependent on contexts, etc. – but the equations turn it what is called a *quotient inductive-inductive type*, as Altenkirch et al. [2018] and Kaposi et al. [2019a] define it. Quotient inductive-inductive types are a very useful generalization of inductive-inductive types, featuring *path constructors* beside sort and point constructors. They can be used, for example to represent the type of real numbers in type theory [Univalent Foundations Program, 2013].

We will now look at a few example to make it clearer on how to encode inductive-inductive declarations in the syntax presented above. To see what the different function types are used for, consider the following two examples:

Example 5.1.4 (Natural numbers)**.** The encoding of the natural numbers as would correspond to the following source type theory context using the first :

$$\cdot, \mathcal{U}, \mathsf{El}(\mathsf{vz}), \Pi(\mathsf{vs}(\mathsf{vz}), \mathsf{El}(\mathsf{vs}(\mathsf{vs}(\mathsf{vz})))).$$

Often, we will, instead of denoting variables using de Bruijn indices, use names as binders in contexts and domains of Π-types to make example contexts more legible. Assuming we always use fresh names, this is not any more imprecise than restricting ourselves to use vz and vz instead:

$$\cdot, \mathbb{N} : \mathcal{U}, 0 : \mathsf{El}(\mathbb{N}), S : \Pi(n : \mathbb{N}, \mathsf{El}(\mathbb{N})).$$

Example 5.1.5 (Vectors)**.** The type of vectors over a type $A : \mathcal{U}$ can be represented using the "external" natural numbers $\mathbb{N}$. In the following, the constructor *cons* uses both the non-inductive and the inductive function type:

$$\begin{gathered}\cdot,\ vec : \hat{\Pi}(\mathbb{N},\ \lambda n.\mathcal{U}),\ nil : \mathsf{El}(vec(0)),\\ cons : \hat{\Pi}(A,\ \lambda a.\hat{\Pi}(\mathbb{N},\ \lambda n.\Pi(v : vec(n),\ \mathsf{El}(vec(n+1)))))\end{gathered}$$

With de-Bruijn indices instead of names the signature Γ_{vec} would be

$$\begin{gathered}\cdot,\ \hat{\Pi}(\mathbb{N},\ \lambda n.\mathcal{U}),\ \mathsf{El}(\mathsf{vz}(0)),\\ \hat{\Pi}(A,\ \lambda a.\hat{\Pi}(\mathbb{N},\ \lambda n.\Pi(\mathsf{vs}(\mathsf{vz})(n),\ \mathsf{El}(\mathsf{vs}(\mathsf{vs}(\mathsf{vz}))(n+1)))))\end{gathered}$$

Example 5.1.6 (Type Theory Syntax)**.** The example of the syntax of type theory 5.0.1 is represented by the following signature Γ_{ConTy}:

$$\begin{gathered}\cdot,\ Con : \mathcal{U},\ Ty : \Pi(\Gamma : Con,\ \mathcal{U}),\\ nil : \mathsf{El}(Con),\\ ext : \Pi(\Gamma : Con,\ \Pi(A : Ty,\ \mathsf{El}(Con))),\\ unit : \Pi(\Gamma : Con,\ \mathsf{El}(\mathsf{app}(Ty))),\\ pi : \Pi(\Gamma : Con,\ \Pi(A : \mathsf{app}(Ty),\ \Pi(B : Ty(ext(\Gamma, A)),\ Ty(\Gamma))))\end{gathered}$$

This is where the more general form of application, which we have defined in Definition 5.1.2 comes handy to make the notation of signatures lighter.

5.2 Algebras of Inductive-Inductive Types

To give meaning to the codes expressed in the source type theory, we need to interpret the contexts as a type in the ambient type theory whose elements are the algebras of of the specified inductive-inductive type. This means that the interpretation of our contexts must give the types of the sort and point constructors they specify.

Definition 5.2.1 (Algebra Operator)**.** By structural recursion over the source syntax, we define an operation $-^{\mathsf{A}}$ which assigns types to source contexts, fibrations over those to types, sections of these fibrations to terms, and maps between types to substitutions:

$$\frac{\vdash \Gamma}{\Gamma^{\mathsf{A}} : \mathcal{U}_1} \qquad \frac{\Gamma \vdash A :: \mathsf{S}}{A^{\mathsf{A}} : \Gamma^{\mathsf{A}} \to \mathcal{U}_1} \qquad \frac{\Gamma \vdash A :: \mathsf{P}}{A^{\mathsf{A}} : \Gamma^{\mathsf{A}} \to \mathcal{U}_0}$$

$$\frac{\Gamma \vdash t : A}{t^{\mathsf{A}} : (\gamma : \Gamma^{\mathsf{A}}) \to A^{\mathsf{A}}(\gamma)} \qquad \frac{\Gamma \xrightarrow{\sigma} \Delta}{\sigma^{\mathsf{A}} : \Gamma^{\mathsf{A}} \to \Delta^{\mathsf{A}}}$$

We will give the construction on contexts, types, subsitutions and terms in the same order as there were presented in Section 5.1. On contexts, the operation is defined by iterated Σ-types:

$$\cdot^{\mathsf{A}} :\equiv \mathbf{1} \text{ and}$$
$$(\Gamma,\, A)^{\mathsf{A}} :\equiv (\gamma : \Gamma^{\mathsf{A}}) \times A^{\mathsf{A}}(\gamma)$$

The universe in the syntax needs of course be mapped to the metatheoretic universe. We chose $\mathcal{U}_1$ as a target for context interpretation to make sure $\mathcal{U}_0$ fits in there. The operation El is just there to make the conversation between small and big types, which in turn is needed to ensure positivity of the constructors. Since this distinction doesn't have any semantic meaning, El will just be ignored by the algebra operator:

$$\mathcal{U}^{\mathsf{A}}(\gamma) :\equiv \mathcal{U}_0$$
$$(\mathsf{El}(a))^{\mathsf{A}}(\gamma) :\equiv a^{\mathsf{A}}(\gamma)$$

Recursive Π-types become metatheoretic dependent function spaces, with app the usual function application:

$$\Pi(a, B)^{\mathsf{A}}(\gamma) :\equiv (\alpha : a^{\mathsf{A}}(\gamma)) \to B^{\mathsf{A}}(\gamma, \alpha)$$
$$\mathsf{app}(t)^{\mathsf{A}}(\gamma, \alpha) :\equiv t^{\mathsf{A}}(\gamma, \alpha)$$

Non-recursive Π-types also become functions, but here we have to apply the external argument to the codomain to be able to evaluate its interpretation:

$$\hat{\Pi}(T,\, B)^{\mathsf{A}}(\gamma) :\equiv (\tau : T) \to B(\tau)^{\mathsf{A}}(\gamma)$$
$$f(\tau)^{\mathsf{A}}(\gamma) :\equiv f^{\mathsf{A}}(\gamma, \tau)$$

Unsurprisingly the category structure of substitution is achieved by interpreting it into the one of metatheoretic functions between context interpretations:

$$\mathsf{id}^{\mathsf{A}}(\gamma) :\equiv \gamma$$
$$(\sigma \circ \delta)^{\mathsf{A}}(\gamma) :\equiv \sigma^{\mathsf{A}}(\delta^{\mathsf{A}}(\gamma))$$

Pulling back a type or a term along a substitution means interpreting it after applying the function which we get from interpreting the substitution:

$$A[\sigma]^{\mathsf{A}}(\gamma) :\equiv A^{\mathsf{A}}(\sigma^{\mathsf{A}}(\gamma))$$
$$t[\sigma]^{\mathsf{A}}(\gamma) :\equiv t^{\mathsf{A}}(\sigma^{\mathsf{A}}(\gamma))$$

The interpretation of the empty substitution is the unique map into the interpretation of the empty context. Extension of and projections from a substitution now justify their name by being interpreted as the extension of a function and the projections of a σ-type:

$$\begin{aligned}
\epsilon^{\mathsf{A}}(\gamma) &:\equiv \star \\
(\sigma, t)^{\mathsf{A}}(\gamma) &:\equiv (\sigma^{\mathsf{A}}(\gamma), t^{\mathsf{A}}(\gamma)) \\
\pi_1(\sigma)^{\mathsf{A}}(\gamma) &:\equiv \mathsf{pr}_1(\sigma^{\mathsf{A}}(\gamma)) \\
\pi_2(\sigma)^{\mathsf{A}}(\gamma) &:\equiv \mathsf{pr}_2(\sigma^{\mathsf{A}}(\gamma))
\end{aligned}$$

All the rules of the substitution calculus which were mentioned in Section 5.1 are preserved *definitionally*, which, in the end, is due to types and functions forming a *strict* category.

Example 5.2.2 (Natural numbers)**.** Strictly speaking, the algebra interpretation of the context from our example of natural numbers (Example 5.1.4) would compute to the following iterated Σ-type:

$$(N' : (N : \top \times \mathcal{U}) \times \mathsf{pr}_2(N)) \times \big(\mathsf{pr}_2(\mathsf{pr}_1(N')) \to \mathsf{pr}_2(\mathsf{pr}_1(N'))\big).$$

Obviously, this is unnecessarily complicated and we can easily transform this type using equivalences to see that we can also express the algebras as being elements of the type

$$(N : \mathcal{U}) \times N \times (N \to N).$$

5.3 Morphisms of Algebras

When we talk about the questions whether a language support inductive-inductive types, we obviously don't only want their constructors – in fact there are many algebras which don't meet our expectation of what the "realization" of an inductive-inductive definition should be. Referring to Example 5.2.2 above, any type N with a point $z : N$ and a function $s : N \to N$ is an algebra for the definition of the natural numbers, even types which are either "too large" and contain much more points, think for example of the real numbers, or too "too small" – the unit type **1** is an algebra for the natural numbers as well. What we want is an algebra which is just large enough that all constructors are injective. We will express this as a categorical property: We will equip the algebras over a context with a notion of a *morphism of algebra* that turns them into a category. The fact that the type is equipped with a non-dependent eliminator will then correspond to the fact that we can find a morphism from its realization to any other algebra – the type is weakly initial. The fact that it also comes with η-rules is expressed by the fact that this morphism is unique, making the realization the *initial algebra*. In this section we first give the definition of morphisms and then define initiality.

Definition 5.3.1 (Morphism Operator)**.** Like for the algebra operator we again perform structural recursion over the syntax to define, what the type of morphisms between two of its algebras should be. Apart from contexts we will also give the operation on types, terms and substitutions, consisting of data indexed over two algebras:

$$\frac{\vdash \Gamma \qquad \gamma, \delta : \Gamma^{\mathsf{A}}}{\Gamma^{\mathsf{M}}(\gamma, \delta) : \mathcal{U}} \qquad \frac{\Gamma \vdash A :: k \qquad \gamma, \delta : \Gamma^{\mathsf{A}} \qquad \mu : \Gamma^{\mathsf{M}}(\gamma, \delta)}{A^{\mathsf{M}}(\mu) : A^{\mathsf{A}}(\gamma) \to A^{\mathsf{A}}(\delta) \to \mathcal{U}}$$

$$\frac{\Gamma \vdash t : A :: k \qquad \gamma, \delta : \Gamma^{\mathsf{A}}}{t^{\mathsf{M}} : (\mu : \Gamma^{\mathsf{M}}(\gamma, \delta)) \to A^{\mathsf{M}}(\mu, t^{\mathsf{A}}(\gamma), t^{\mathsf{A}}(\delta))}$$

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma, \delta : \Gamma^{\mathsf{A}}}{\sigma^{\mathsf{M}} : \Gamma^{\mathsf{M}}(\gamma, \delta) \to \Delta^{\mathsf{M}}(\sigma^{\mathsf{A}}(\gamma), \sigma^{\mathsf{A}}(\delta))}$$

We will again proceed in the order we presented the type theory in Section 5.1, starting with the different ways we can form contexts. There are no surprises here since we can define it by recursion and because the empty context does not contain any information:

$$\cdot^{\mathsf{M}}(\gamma, \delta) :\equiv \mathbf{1} \text{ and}$$
$$(\Gamma,\ A)^{\mathsf{M}}((\gamma, \alpha), (\delta, \beta)) :\equiv (\mu : \Gamma^{\mathsf{M}}(\gamma, \delta)) \times A^{\mathsf{M}}(\mu, \alpha, \beta).$$

For the universe, the definition makes sure that morphisms between sorts are indeed functions between their corresponding realizations. The element operator requires a proof that the two ways we can obtain an algebra of the term in the codomain of the morphism match:

$$\mathcal{U}^{\mathsf{M}}(\mu, \gamma, \delta) :\equiv \gamma^{\mathsf{A}} \to \delta^{\mathsf{A}} \text{ and}$$
$$\mathsf{El}(a)^{\mathsf{M}}(\mu, \alpha, \beta) :\equiv \left(a^{\mathsf{M}}(\mu, \alpha) = \beta\right).$$

Recursive Π-types are mapped to dependent function spaces on the set which is the algebra of the universe term. For the application we can perform induction on the equality proof and thus we can assume it to be refl.

$$\begin{aligned}\Pi(a, B)^{\mathsf{M}}(\mu, \pi, \phi) &:\equiv \left(\alpha : a^{\mathsf{A}}(\gamma)\right) \to B^{\mathsf{M}}\left((\mu, \mathsf{refl}), \pi(\alpha), \phi(a^{\mathsf{M}}(\mu, \alpha))\right)\\ \mathsf{app}(f)^{\mathsf{M}}\{\gamma, \alpha\}(\mu, \mathsf{refl}) &:\equiv f^{\mathsf{M}}(\mu, \alpha)\end{aligned}$$

Morphisms between interpretations of non-recursive Π-types are functions over the external parameter as well:

$$\begin{aligned}\hat{\Pi}(T, B)^{\mathsf{M}}(\mu, \pi, \phi) &:\equiv (\tau : T) \to B(\tau)^{\mathsf{M}}(\mu, \pi(\tau), \phi(\tau))\\ f(\tau)^{\mathsf{M}}(\mu) &:\equiv f^{\mathsf{M}}(\mu, \tau)\end{aligned}$$

The treatment of substitutions looks almost identical to the one in the algebra operator in that the cateogorical structure of subsitution gets interpreted as the strict category of functions in the outer type theory:

$$\begin{aligned}\mathsf{id}^{\mathsf{M}}(\mu) &:\equiv \mu\\ \sigma \circ \delta^{\mathsf{M}}(\mu) &:\equiv \sigma^{\mathsf{M}}(\delta^{\mathsf{M}}(\mu))\end{aligned}$$

$$\begin{aligned}A[\sigma]^{\mathsf{M}}(\mu, \alpha, \beta) &:\equiv A^{\mathsf{M}}(\sigma^{\mathsf{M}}(\mu), \alpha, \beta)\\ t[\sigma]^{\mathsf{M}}(\mu) &:\equiv t^{\mathsf{M}}(\sigma^{\mathsf{M}}(\mu))\end{aligned}$$

$$\begin{aligned}\epsilon^{\mathsf{M}}(\mu) &:\equiv \star\\ (\sigma, t)^{\mathsf{M}}(\mu) &:\equiv \left(\sigma^{\mathsf{M}}(\mu), t^{\mathsf{M}}(\mu)\right)\\ \pi_1(\sigma)^{\mathsf{M}}(\mu) &:\equiv \mathsf{pr}_1(\sigma^{\mathsf{M}}(\mu))\\ \pi_2(\sigma)^{\mathsf{M}}(\mu) &:\equiv \mathsf{pr}_2(\sigma^{\mathsf{M}}(\mu))\end{aligned}$$

Substitution laws hold strictly again.

The definition of morphisms doesn't yet make the algebras over a given signature Γ a category: For this we would still need to define identity and composition and prove the category laws. But these constructions are trivial: In last consequence they are identity and composition of functions in the outer type theory. Besides this, they are not necessary to state the most important categorical attribute is definable without identity and composition: Initiality.

Definition 5.3.2 (Initial Algebra)**.** An algebra $\gamma : \Gamma^{\mathsf{A}}$ is called **weakly initial** if for every other algebra $\delta : \Gamma^{\mathsf{A}}$ we can find a morphism

$$\mu : \Gamma^{\mathsf{M}}(\gamma, \delta).$$

It is (strongly) **initial**, if this morphism is unique for every δ.

The existence of initial algebras for every possible context will be the main topic of Chapter 7.

Remark 5.3.3. We now defined what it means for an algebra to admit a non-dependent eliminator and its β-rules, but what about the dependent eliminator? It is usually defined as the sections of an *dependent morphism* or a *displayed algebra*, over the inital algebra – we will see this notion for the inductive families in the next chapter. But, as Kaposi et al. [2019a] have proven for their version of syntax for higher inductive-inductive types, the dependent version can be recovered from the non-dependent one, representing the dependent morphism as a non-dependent one by taking its Σ-type.

Chapter 6

Specification of Inductive Families

As we have already seen in the last chapter, sometimes it is helpful to allow point constructors of a collection of inductive types to be *mutually dependent*. This means that to define several sorts simultaneously whose point constructors may refer to the other types being defined. We will refer to this class as *inductive families*, though others might call them, for example, mutual inductive types. Inductive families are a class of inductive types which at first glance seems more powerful than indexed W-types but less than inductive-inductive types – sorts are *not* allowed to depend on other sorts but only point constructors.

Previous specifications of mutual inductive families have taken different approaches: Some are based on the notion of a polynomial functor [Altenkirch et al., 2015, Dybjer and Setzer, 1999] while others, like the original Dybjer [1994] description are based on a schematic description.

6.1 Signatures for Inductive Families

Applying the same principle as in the case of inductive-inductive types we want to create a specification based on the contexts of type theory syntax. We could imagine that we can obtain such a specification by just restricting the syntax for inductive-inductive types to not use the recursive Π-type for sorts, but this approach doesn't capture the full extent of inductive families being a much simpler concept than inductive-inductive types. Given the strategy of our reduction we want the specification to capture at least the following features of inductive families:

- Sorts are either types or functions over existing types.
- Point constructors can also be indexed over existing ("external") types.
- Point constructors can refer to any sort being defined.

We will use blue font to distinguish the new syntax from the ambient type theory. The first point above says that we want the *sort types* S to be inductively generated by a *universe* token and a constructor of external functions for sorts which are meant to be *type families*:

$$\frac{}{\mathcal{U} :: \mathsf{S}} \qquad \frac{T : \mathcal{U} \qquad B : T \to \mathsf{S}}{\hat{\Pi}_{\mathsf{S}}(T, B :: \mathsf{S})}$$

For example, the sort of vectors over a type $A : \mathcal{U}$ would be described by $\hat{\Pi}_{\mathsf{S}}(A, \hat{\Pi}_{\mathsf{S}}(\mathbb{N}, \mathcal{U}))$. Note that in contrast to the sort types of inductive-inductive definitions these do not depend on a context.

Instead, we say that a *sort context* is just a list of sort types without any interdependencies:

$$\vdash_{\mathsf{S}} \cdot_{\mathsf{S}} \qquad \frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}} \qquad B :: \mathsf{S}}{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}, B}$$

In order to refer to sorts we introduce a simplified term calculus based on typed de Bruijn indices for bound variables and an application operation for type families:

$$\frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}} \qquad B :: \mathsf{S}}{\Gamma_{\mathsf{S}}, B \vdash_{\mathsf{S}} \mathsf{var}(\mathsf{vz}) : B} \qquad \frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} \mathsf{var}(v) : B}{\Gamma_{\mathsf{S}}, B' \vdash_{\mathsf{S}} \mathsf{var}(\mathsf{vs}(v)) : B}$$

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : \hat{\Pi}_{\mathsf{S}}(T, B) \qquad \tau : T}{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t(\tau) : B(\tau)}$$

Point constructors will be represented by *point types* over a given sort context. This means that in contrast to inductive-inductive types, they cannot depend on other point types. The type formers we need are the element type for the universe $\mathcal{U}$, an external, non-recursive function type like the one we have for sorts, and an internal function type used for recursive point constructors – which are *non-dependent* since point constructors only depend on the *sort context*:

$$\frac{\Gamma_S \vdash_S a : \mathcal{U}}{\Gamma_S \vdash_S \mathsf{El}(a)} \qquad \frac{T : \mathcal{U} \qquad (\tau : T) \to \Gamma_S \vdash_S B(\tau)}{\Gamma_S \vdash_S \hat{\Pi}_P(T,\, B)}$$

$$\frac{\Gamma_S \vdash_S a : \mathcal{U} \qquad \Gamma_S \vdash_S A}{\Gamma_S \vdash_S a \Rightarrow_P A}$$

As a last building block of the syntax, we can now form contexts consisting of point constructors over a given sort context. Such a context Γ can be formed over a given sort context Γ_S which we will denote as a subscript to the turnstile or omit when inferrable. The empty context can be formed over any sort context, and an extension by a point constructor leaves the sort context fixed:

$$\frac{\vdash_S \Gamma_S}{\vdash_{\Gamma_S} \Gamma} \qquad \frac{\vdash_{\Gamma_S} \Gamma \qquad \Gamma_S \vdash_S A}{\vdash_{\Gamma_S} \Gamma, A}$$

Example 6.1.1 (Natural numbers, Vectors)**.** A common example for inductive types, the natural numbers, with one constructor for zero and one for the successor function, are represented by the sort context $\cdot_S, \mathcal{U}$ and the points

$$\mathsf{El}(\mathsf{var}(\mathsf{vz})), \mathsf{var}(\mathsf{vz}) \Rightarrow_P \mathsf{El}(\mathsf{var}(\mathsf{vz})).$$

An example of a real indexed type would be the type family of vectors over a fixed type $A : \mathcal{U}$ which is defined over the sort context $\cdot_S,\, \hat{\Pi}_S(n : \mathbb{N},\, \mathcal{U})$ by

$$\begin{gathered}\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})(0)),\\ \hat{\Pi}_P(a : A,\, \hat{\Pi}_P(n : \mathbb{N},\, \mathsf{var}(\mathsf{vz})(n) \Rightarrow_P \mathsf{El}(\mathsf{var}(\mathsf{vz})(n+1)))).\end{gathered}$$

An easy example with non-trivial mutual dependency between the point constructors is the predicate of evenness and oddness on natural numbers: The sorts are represented by $\cdot_S, \Pi_S(\mathbb{N}, \lambda n.\mathcal{U}), \Pi_S(\mathbb{N}, \lambda n.\mathcal{U})$ and the point constructors by

$$\begin{gathered}\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vs}(\mathsf{vz}))(0)),\\ \hat{\Pi}_P(n : \mathbb{N},\, \mathsf{var}(\mathsf{vz})(n) \Rightarrow_P \mathsf{El}(\mathsf{var}(\mathsf{vs}(\mathsf{vz}))(n+1))),\\ \hat{\Pi}_P(n : \mathbb{N},\, \mathsf{var}(\mathsf{vs}(\mathsf{vz}))(n) \Rightarrow_P \mathsf{El}(\mathsf{var}(\mathsf{vz})(n+1))).\end{gathered}$$

Here, the first sort constructor represents evenness, the second one oddness, the first point constructor the proof that 0 is even and the other two the proof that evenness implies oddness of the successor and vice versa.

Definition 6.1.2 (Sort Substitutions)**.** One component of the syntax which has completely gone missing are substitutions. Since we cant refer to previous point constructors, we certainly don't need them for the point contexts. But since we also got rid of sort interdependencies, we could reduce the recursive function types on points to a non-depenent one and thus don't need to use substitutions in the definition of application. It will still be helpful for syntax transformations to use substitutions of the sort contexts which we define as generated by

$$\frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}}{\Gamma_{\mathsf{S}} \xrightarrow{\epsilon} \cdot_{\mathsf{S}}} \quad \text{and} \quad \frac{\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}} \qquad \Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : B}{\Gamma_{\mathsf{S}} \xrightarrow{\sigma, t} (\Delta_{\mathsf{S}}, B)}.$$

These then allow us to substitute point types, point contexts, and sort terms via the following "pullback" operations:

$$\frac{\Delta_{\mathsf{S}} \vdash_{\mathsf{S}} A \qquad \Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A[\sigma]} \qquad \frac{\Delta_{\mathsf{S}} \vdash_{\mathsf{S}} t : B \qquad \Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t[\sigma] : B}$$

$$\frac{\vdash_{\Delta_{\mathsf{S}}} \Delta \qquad \Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\vdash_{\Gamma_{\mathsf{S}}} \Delta[\sigma]}$$

given by the defining rules for substitution

$$\begin{aligned}
\hat{\Pi}_{\mathsf{P}}(T,\, A)[\sigma] &= \hat{\Pi}_{\mathsf{P}}(T,\, \lambda\tau.\, A(\tau)[\sigma]), \\
\mathsf{El}(a)[\sigma] &= \mathsf{El}(a[\sigma]), \\
(a \Rightarrow_{\mathsf{P}} A)[\sigma] &= a[\sigma] \Rightarrow_{\mathsf{P}} A[\sigma], \\
\mathsf{var}(\mathsf{vz})[\sigma, t] &= t, \\
\mathsf{var}(\mathsf{vs}(t))[\sigma, s] &= \mathsf{var}(t)[\sigma] && \text{for } \Delta_{\mathsf{S}} \vdash_{\mathsf{S}} \mathsf{var}(t) : B, \\
f(\tau)[\sigma] &= f[\sigma](\tau) && \text{for } \Delta_{\mathsf{S}} \vdash_{\mathsf{S}} f : \hat{\Pi}_{\mathsf{S}}(T,\, B), \\
\cdot[\sigma] &= \cdot, \text{ and} \\
(\Gamma,\, A)[\sigma] &= (\Gamma[\sigma],\, A[\sigma]).
\end{aligned}$$

We can derive from this the gadgets of the substitutional calculus which we are already acquainted with from the syntax of inductive-inductive Types: We can define the *weakening* of a subsitution $\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}$ to $\Gamma_{\mathsf{S}}, B \xrightarrow{\mathsf{wk}_\sigma} \Delta_{\mathsf{S}}$ via recursion on σ by

$$\begin{aligned}
\mathsf{wk}_\epsilon &:\equiv \epsilon \text{ and} \\
\mathsf{wk}_{\sigma,t} &:\equiv (\mathsf{wk}_\sigma, \mathsf{vs}(t)).
\end{aligned}$$

Using wk, we can then recover the categorical structure of the substitutions by defining the identity $\Gamma_{\mathsf{S}} \xrightarrow{\mathsf{id}_{\Gamma_{\mathsf{S}}}} \Gamma_{\mathsf{S}}$ by recursion of the context Γ_{S}:

$$\mathsf{id}_{(\cdot_{\mathsf{S}})} :\equiv \epsilon \text{ and}$$
$$\mathsf{id}_{\Gamma_{\mathsf{S}},B} :\equiv (\mathsf{wk}_{\mathsf{id}_{\Gamma_{\mathsf{S}}}}, \mathsf{vz}).$$

Composition $\Gamma_{\mathsf{S}} \xrightarrow{\sigma\circ\delta} \Sigma_{\mathsf{S}}$ of substitutions $\Delta_{\mathsf{S}} \xrightarrow{\sigma} \Sigma_{\mathsf{S}}$ and $\Gamma_{\mathsf{S}} \xrightarrow{\delta} \Delta_{\mathsf{S}}$ is defined by recursion on the first context:

$$\epsilon \circ \delta :\equiv \epsilon,$$
$$(\sigma, t) \circ \delta :\equiv (\sigma \circ \delta, t[\delta]).$$

Projections $\Gamma_{\mathsf{S}} \xrightarrow{\pi_1(\sigma)} \Delta_{\mathsf{S}}$ and $\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} \pi_2(\sigma) : B$ of a substitution $\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}, B$ can be defined as just that – projections. Any substitution between Γ_{S} and Δ_{S}, B is of the form σ, t and we can just set

$$\pi_1(\sigma, t) :\equiv \sigma \text{ and}$$
$$\pi_2(\sigma, t) :\equiv t.$$

6.2 Algebras of Inductive Families

Like for inductive-inductive types, we have to give a way to semantify the signatures by stating what kind of data they should represent.

Definition 6.2.1 (Algebra operator)**.** Again, sort contexts will be mapped to types, sort constructors to families over these types, their terms to sections of these families. Point contexts will give the same data, but depending on an interpretation of the sort contexts:

$$\frac{B :: \mathsf{S}}{B^{\mathsf{A}} : \mathcal{U}} \qquad \frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}}{\Gamma_{\mathsf{S}}{}^{\mathsf{A}} : \mathcal{U}} \qquad \frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S}}{t^{\mathsf{A}} : \Gamma_{\mathsf{S}}{}^{\mathsf{A}} \to B^{\mathsf{A}}}$$

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A :: \mathsf{P}}{A^{\mathsf{A}} : \Gamma_{\mathsf{S}}{}^{\mathsf{A}} \to \mathcal{U}} \qquad \frac{\vdash_{\Gamma_{\mathsf{S}}} \Gamma}{\Gamma^{\mathsf{A}} : \mathcal{U}}$$

Going through all of these translation in order, we first define the algebras of sorts to be interpreted into functions over the universe:

$$\mathcal{U}^{\mathsf{A}} :\equiv \mathcal{U}$$
$$\hat{\Pi}_{\mathsf{S}}(T, B)^{\mathsf{A}} :\equiv (\tau : T) \to B(\tau)^{\mathsf{A}}$$

Sort contexts become iterated product types – note that we don't even need to use Σ-types since there are no dependencies between sorts:

$$
\begin{aligned}
{\cdot_{\mathsf{S}}}^{\mathsf{A}} &:\equiv \mathbf{1} \\
(\Gamma_{\mathsf{S}}, B)^{\mathsf{A}} &:\equiv {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \times B^{\mathsf{A}}
\end{aligned}
$$

We use terms to navigate these iterated product via iterated projects, and to apply function sorts:

$$
\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{A}}(\gamma, \alpha) &:\equiv \alpha \\
\mathsf{var}(\mathsf{vs}(t))^{\mathsf{A}}(\gamma, \alpha) &:\equiv \mathsf{var}(t)^{\mathsf{A}}(\gamma) \\
t(\tau)^{\mathsf{A}}(\gamma) &:\equiv t^{\mathsf{A}}(\gamma)(\tau)
\end{aligned}
$$

For point constructors, we need to interpret both types of functions into functions while erasing the element operator, since it does not have any semantic meaning:

$$
\begin{aligned}
\mathsf{El}(a)^{\mathsf{A}}(\gamma) &:\equiv a^{\mathsf{A}}(\gamma) \\
\hat{\Pi}_{\mathsf{P}}(T, A)^{\mathsf{A}}(\gamma) &:\equiv (\tau : T) \to A(\tau)^{\mathsf{A}}(\gamma) \\
(a \Rightarrow_{\mathsf{P}} A)^{\mathsf{A}}(\gamma) &:\equiv a^{\mathsf{A}}(\gamma) \to A^{\mathsf{A}}(\gamma)
\end{aligned}
$$

Just like for the sort contexts, point contexts are interdependency-free lists of point constructors and as such can be interpreted as simple products instead of Σ-types:

$$
\begin{aligned}
\cdot^{\mathsf{A}}(\gamma) &:\equiv \mathbf{1} \\
(\Gamma, A)^{\mathsf{A}}(\gamma) &:\equiv \Gamma^{\mathsf{A}}(\gamma) \times A^{\mathsf{A}}(\gamma)
\end{aligned}
$$

Example 6.2.2 (Natural numbers)**.** Looking at the signature of the natural numbers from Example 6.1.1, we see that the algebra interpretation of its sort context evaluates to

$$\mathbf{1} \times \mathcal{U}$$

and given an element $(\star, N) : \mathbf{1} \times \mathcal{U}$, the algebras of its point contexts, evaluated at this point result in

$$N \times (N \to N).$$

In the previous section, we introduced a substition calculus for the sort contexts. Obviously, we might also want to consider algebras over these substitutions.

Definition 6.2.3 (Algebras of substitutions)**.** We can extend the algebra operator by defining it on substitutions by functions between the interpretations of sort contexts:

$$\frac{\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\sigma^{\mathsf{A}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \to {\Gamma_{\mathsf{S}}}^{\mathsf{A}}}$$

This is done by setting

$$\begin{aligned} \epsilon^{\mathsf{A}} &:\equiv \star \text{ and} \\ (\sigma, t)^{\mathsf{A}} &:\equiv (\sigma^{\mathsf{A}}, t^{\mathsf{A}}). \end{aligned}$$

Lemma 6.2.4. *It's easy to check that this definition of algebras of a subtitution respects the substitution calculus given in Definition 6.1.2 in the following sense:*

$$\begin{aligned} A[\sigma]^{\mathsf{A}}(\gamma) &= A^{\mathsf{A}}(\sigma^{\mathsf{A}}(\gamma)), \\ t[\sigma]^{\mathsf{A}}(\gamma) &= t^{\mathsf{A}}(\sigma^{\mathsf{A}}(\gamma)), \\ \mathsf{id}^{\mathsf{A}}(\gamma) &= \gamma, \\ (\sigma \circ \delta)^{\mathsf{A}}(\gamma) &= \sigma^{\mathsf{A}}(\delta^{\mathsf{A}}(\gamma)), \\ {\mathsf{wk}_{\sigma}}^{\mathsf{A}}(\gamma, \alpha) &= \sigma^{\mathsf{A}}(\gamma), \\ \pi_1(\sigma)^{\mathsf{A}}(\gamma) &= \mathsf{pr}_1(\sigma^{\mathsf{A}}(\gamma)), \textit{ and} \\ \pi_2(\sigma)^{\mathsf{A}}(\gamma) &= \mathsf{pr}_2(\sigma^{\mathsf{A}}(\gamma)). \end{aligned}$$

Proof. We can prove the first rule by recursion on the point type $\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A :: \mathsf{P}$, the second rule by recursing on the term $\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S}$, the third by induction on the context, and all other by induction by the substitution. □

6.3 Displayed Algebras and their Sections

To represent the dependent eliminator, we need algebras which vary over other algebras. To get a feeling about what these should look like, let us first look at our usual simplest example:

Example 6.3.1. Take algebras $(\star, N) : \mathbf{1} \times \mathcal{U}$ and $(\star, z, s) : \mathbf{1} \times N \times (N \to N)$ of the the signature of natural numbers (Example 6.1.1). A *displayed algebra* over this should contain the data which the dependent eliminator of the natural numbers takes as input: A type family $P : N \to \mathcal{U}$ together with a point $p_z : P(z)$ and a family of functions $p_s : (n : N) \to P(n) \to P(s(n))$.

A *section* of this algebra would be a section $f : (n : N) \to P(n)$ of P respecting the other data by ensuring that $f(z) = p_z$ and that for all $n : N$, we have $f(s(n)) = p_s(f(n))$.

Let's first concentrate on the first piece of data:

Definition 6.3.2 (Displayed Algebra Operator)**.** As seen above, we want to map sorts to type families over the given algebra. Sort context will likewise be type families over an algebra:

$$\frac{B :: \mathsf{S}}{B^{\mathsf{D}} : B^{\mathsf{A}} \to \mathcal{U}} \qquad \frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}}{{\Gamma_{\mathsf{S}}}^{\mathsf{D}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \to \mathcal{U}}$$

Since sorts can themselves be interpreted as functions, we have to apply them whenever we encounter a sort function. Sort contexts will again be interpreted as iterated products.

$$\begin{aligned}
\mathcal{U}^{\mathsf{D}}(\alpha) &:\equiv \alpha \to \mathcal{U} \\
\hat{\Pi}_{\mathsf{S}}(T, B)^{\mathsf{D}}(\alpha) &:\equiv (\tau : T) \to B(\tau)^{\mathsf{D}}(\alpha(\tau)) \\
{\cdot_{\mathsf{S}}}^{\mathsf{D}}(\star) &:\equiv \mathbf{1} \\
(\Gamma_{\mathsf{S}}, B)^{\mathsf{D}}(\gamma, \alpha) &:\equiv \Gamma^{\mathsf{D}}(\gamma) \times B^{\mathsf{D}}(\alpha)
\end{aligned}$$

The interpretation of point constructors and of point contexts now not only depends on the algebra, but also on the interpretation of the underlying sorts:

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A :: \mathsf{P}}{A^{\mathsf{D}} : \{\gamma : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma) \to A^{\mathsf{A}}(\gamma) \to \mathcal{U}}$$

$$\frac{\vdash_{\Gamma_{\mathsf{S}}} \Gamma}{\Gamma^{\mathsf{D}} : \{\gamma : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma) \to \Gamma^{\mathsf{A}}(\gamma) \to \mathcal{U}}$$

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S}}{t^{\mathsf{D}} : \{\gamma : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma) \to B^{\mathsf{D}}(t^{\mathsf{A}}(\gamma))}$$

The definition on terms is almost the same as for fixed algebras:

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &:\equiv \alpha^{\mathsf{D}}, \\
\mathsf{var}(\mathsf{vs}(t))^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &:\equiv \mathsf{var}(t)^{\mathsf{D}}(\gamma^{\mathsf{D}}), \text{ and} \\
f(\tau)^{\mathsf{D}}(\gamma^{\mathsf{D}}) &:\equiv f^{\mathsf{D}}(\gamma^{\mathsf{D}})(\tau).
\end{aligned}$$

Displayed algebras on point constructors are defined fiberwise, like the ones for sorts:

$$\begin{aligned}
\mathsf{El}(a)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha) &:\equiv a^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha)\\
\hat{\Pi}_{\mathsf{P}}(T,\, A)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\pi) &:\equiv (\tau : T) \to A(\tau)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\pi(\tau))\\
(a \Rightarrow_{\mathsf{P}} A)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\pi) &:\equiv \{\alpha : a^{\mathsf{A}}(\gamma_{\mathsf{S}})\} \to a^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha) \to A^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\pi(\alpha))
\end{aligned}$$

Finally, point contexts are interpreted as iterated products again:

$$\begin{aligned}
\cdot^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\gamma) &:\equiv \mathbf{1}\\
(\Gamma,\, A)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},(\gamma,\alpha)) &:\equiv \Gamma^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\gamma) \times A^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha)
\end{aligned}$$

Definition 6.3.3 (Section Operator)**.** For sorts and sort contexts, we want the sections of a displayed algebra to be the sections of the type family they represent:

$$\frac{B :: \mathsf{S}}{B^{\mathsf{S}} : \{\alpha : B^{\mathsf{A}}\} \to B^{\mathsf{D}}(\alpha) \to \mathcal{U}}$$

$$\frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}}{{\Gamma_{\mathsf{S}}}^{\mathsf{S}} : \{\gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}}) \to \mathcal{U}}$$

Both follow the structure of the underlying displayed algebra – fibrewise for sort functions and by iterated products on sort contexts:

$$\begin{aligned}
\mathcal{U}^{\mathsf{S}}(\alpha^{\mathsf{D}}) &:\equiv (x : \alpha) \to \alpha^{\mathsf{D}}(x)\\
\hat{\Pi}_{\mathsf{S}}(T,\, B)^{\mathsf{S}}(\pi^{\mathsf{D}}) &:\equiv (\tau : T) \to B(\tau)^{\mathsf{S}}(\pi^{\mathsf{D}}(\tau))\\
\cdot_{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}}) &:\equiv \mathbf{1}\\
(\Gamma_{\mathsf{S}}, B)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha^{\mathsf{D}}) &:\equiv {\Gamma_{\mathsf{S}}}^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}}) \times B^{\mathsf{S}}(\alpha^{\mathsf{D}})
\end{aligned}$$

Sections of point constructors, point contexts, and sort terms will clearly have to depend on a section of the underlying sort interpretation:

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A :: k \qquad \gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \qquad \gamma_{\mathsf{S}}^{\mathsf{D}} : {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}}) \qquad \gamma_{\mathsf{S}}^{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}})}{A^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}) : \left\{\alpha : A^{\mathsf{A}}(\gamma_{\mathsf{S}})\right\} \left(\alpha^{\mathsf{D}} : A^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\alpha)\right) \to \mathcal{U}}$$

$$\frac{\vdash_{\Gamma_{\mathsf{S}}} \Gamma \qquad \gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \qquad \gamma_{\mathsf{S}}^{\mathsf{D}} : {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}}) \qquad \gamma_{\mathsf{S}}^{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}})}{\Gamma^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}) : \left\{\gamma : \Gamma^{\mathsf{A}}(\gamma_{\mathsf{S}})\right\} \left(\gamma^{\mathsf{D}} : \Gamma^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}},\gamma) \to \mathcal{U}\right)}$$

$$\frac{\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}} \qquad \gamma_{\mathsf{S}}^{\mathsf{D}} : {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}}) \qquad \gamma_{\mathsf{S}}^{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{D}})}{t^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}) : B^{\mathsf{S}}\left(t^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}})\right)}$$

For point types we again descent fibrewise, but what to do about the element operator? This is where the equations which we have seen in Example 6.3.1 come into play: The element which we get out of the interpretation of the section must coincide with the one we provided by giving the displayed algebra:

$$\begin{aligned}
\mathsf{El}(a)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \alpha^{\mathsf{D}}) &:\equiv \left(a^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \alpha) = \alpha^{\mathsf{D}}\right) \\
\hat{\Pi}_{\mathsf{P}}(T,\, A)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \pi^{\mathsf{D}}) &:\equiv (\tau : T) \to A(\tau)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \pi^{\mathsf{D}}(\tau)) \\
(a \Rightarrow_{\mathsf{P}} A)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \pi^{\mathsf{D}}) &:\equiv (\alpha : a^{\mathsf{A}}(\gamma_{\mathsf{S}})) \to A^{\mathsf{S}}\left(\gamma_{\mathsf{S}}^{\mathsf{S}}, \pi^{\mathsf{D}}(a^{\mathsf{S}}(a, \gamma_{\mathsf{S}}^{\mathsf{S}})(\alpha))\right)
\end{aligned}$$

The definition of sections of point contexts is easier as it is, again, just an iteration of products:

$$\begin{aligned}
\cdot^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \gamma^{\mathsf{D}}) &:\equiv \mathbf{1} \\
(\Gamma,\, A)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, (\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}})) &:\equiv \Gamma^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \gamma^{\mathsf{D}}) \times A^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \alpha^{\mathsf{D}})
\end{aligned}$$

At last, also terms follow the usual pattern of variables selecting sort interpretations via projections of products and interpreting the application by metatheoretic application:

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \alpha^{\mathsf{S}}) &:\equiv \alpha^{\mathsf{S}} \\
\mathsf{var}(\mathsf{vs}(t))^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}, \alpha^{\mathsf{S}}) &:\equiv \mathsf{var}(t)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}) \\
f(\tau)^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}}) &:\equiv f^{\mathsf{S}}(\gamma_{\mathsf{S}}^{\mathsf{S}})(\tau)
\end{aligned}$$

Later on, we will need that, following Definition 6.2.3, we can interpret sort substitutions with the means of displayed algebras, for which we also need a definition of a section:

Definition 6.3.4 (Displayed Algebras of Substitutions)**.** Given a sort substitution, its type of displayed algebras should be the type of function between the displayed algebras of its domain and codomain, where in the latter we have to apply the function which we get from the *algebra* over the substitution:

$$\frac{\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\sigma^{\mathsf{D}} : \left\{\gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\right\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}}) \to {\Delta_{\mathsf{S}}}^{\mathsf{D}}(\sigma^{\mathsf{A}}(\gamma_{\mathsf{S}}))}$$

These are defined, like in the non-displayed case, by

$$\begin{aligned}
\epsilon^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}) &:\equiv \star \text{ and} \\
(\sigma,\, t)^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}) &:\equiv \left(\sigma^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}), t^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}})\right).
\end{aligned}$$

Definition 6.3.5 (Sections of Substitutions)**.** A section of a displayed algebra of a sort substitution is supposed to map sections of its domain to sections of its codomain:

$$\frac{\Gamma_{\mathsf{S}} \xrightarrow{\sigma} \Delta_{\mathsf{S}}}{\sigma^{\mathsf{S}} : \left\{\gamma_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}\right\} \left\{\gamma^{\mathsf{D}}_{\mathsf{S}} : {\Gamma_{\mathsf{S}}}^{\mathsf{D}}(\gamma_{\mathsf{S}})\right\} \to {\Gamma_{\mathsf{S}}}^{\mathsf{S}}(\gamma^{\mathsf{D}}_{\mathsf{S}}) \to {\Delta_{\mathsf{S}}}^{\mathsf{S}}(\sigma^{\mathsf{D}}(\gamma^{\mathsf{D}}_{\mathsf{S}}))}$$

Again, this is happening componentwise:

$$\begin{aligned} \epsilon^{\mathsf{S}}(\gamma^{\mathsf{S}}_{\mathsf{S}}) &:\equiv \star \text{ and} \\ (\sigma, t)^{\mathsf{S}}(\gamma^{\mathsf{S}}_{\mathsf{S}}) &:\equiv \left(\sigma^{\mathsf{S}}(\gamma^{\mathsf{S}}_{\mathsf{S}}), t^{\mathsf{S}}(\gamma^{\mathsf{S}}_{\mathsf{S}})\right). \end{aligned}$$

6.4 Existence of Inductive Families

Having a specification for Inductive Families is not worth much if there is no way to know what it means for a type theory to actually "support" types of this specification. The intended meaning of the signatures is clear from the definition of their algebras as seen in Section 6.2 and as discussed in Section 6.3, candidates for their eliminators and computation rules are specified in the definition of sections displayed algebras. This means that we can formally say what it means for inductive families to exist in a type theory. In this section, we will prove that any metatheory as premised in Section 6.4 actually supports inductive families as specified here. Since we make heavy use of indexed W-types, we can also see this endeavour as *reducing* inductive families to indexed W-types.

Theorem 6.4.1 (Existence of Inductive Families)**.** *For every signature of inductive families given by a sort context $\vdash_{\mathsf{S}} \Omega_{\mathsf{S}}$ and a point context $\vdash_{\Omega_{\mathsf{S}}} \Omega$, there are are sort and point* constructors *in the form of*

$$\begin{aligned} \mathsf{cons}_{\mathsf{S}}(\Omega) &: {\Omega_{\mathsf{S}}}^{\mathsf{A}} \textit{ and} \\ \mathsf{con}(\Omega) &: \Omega^{\mathsf{A}}(\mathsf{cons}_{\mathsf{S}}(\Omega)) \end{aligned}$$

such that for each displayed algebra given by motives $\omega^{\mathsf{D}}_{\mathsf{S}} : {\Omega_{\mathsf{S}}}^{\mathsf{D}}(\mathsf{cons}_{\mathsf{S}}(\Omega))$ and methods $\omega^{\mathsf{D}} : \Omega^{\mathsf{D}}(\omega^{\mathsf{D}}_{\mathsf{S}}, \mathsf{con}(\Omega))$ we can prove an eliminator *by the means of giving sections*

$$\begin{aligned} \mathsf{elim}_{\mathsf{S}}(\Omega, \omega^{\mathsf{D}}) &: {\Omega_{\mathsf{S}}}^{\mathsf{S}}(\omega^{\mathsf{D}}_{\mathsf{S}}) \textit{ with} \\ \mathsf{elim}(\Omega, \omega^{\mathsf{D}}) &: \Omega^{\mathsf{S}}(\mathsf{elim}_{\mathsf{S}}(\Omega, \omega^{\mathsf{D}}), \omega^{\mathsf{D}}). \end{aligned}$$

Our strategy to prove this theorem is to first extend our syntax with elements that have been missing: terms and substitutions for point types. For the extended syntax, we will than show that indexed W-types allow us to find an *internal representation* of the syntax (Section 6.4.1) and then construct a *term model* using the internalization, which we can then show to be the initial algebra (Section 6.4.2).

6.4.1 Internalization of the Syntax

At first, we will need to make up for some of the short cuts and simplifications in our definition of signatures. In the theory of semantics of type theory, which studies various models of different type theories, the model which is initial in the category of all models is usually called the *term model*. This is because in this model, a type get interpreted as the set of all of its terms. Since our signatures form – or are at least strongly inspired by – a type theoretic syntax as well, we might hope to deploy the same strategy for inductive families. In the core of this interpretation is the issue of how to find an interpretation for a given sort term a of the universe token $\mathcal{U}$. The interpretation of this ought to be the terms of the *point type* $\mathsf{El}(a)$ associated with this sort term. But our syntax does not mention terms of point types at all, since point constructor are not interdependent! So our solution is to retrofit the theory with terms, as well as substitutions for the point contexts:

Definition 6.4.2 (Point Substitution Calculus)**.** Let us fix a sort context $\vdash_S \Gamma_S$. In total, there are four ways to construct reasonable terms of point types in Γ_S: Via two constructors for de-Bruijn indices to navigate point contexts and by an application constructor for each of the two kinds of Π-type present in the syntax.

$$\frac{\vdash_{\Gamma_S} \Gamma \qquad \Gamma_S \vdash_S A}{\Gamma, A \vdash \mathsf{var}(\mathsf{vz}) : A :: \mathsf{P}} \qquad \frac{\Gamma_S \vdash_S A \qquad \Gamma_S \vdash_S A' \qquad \Gamma \vdash \mathsf{var}(t) : A :: \mathsf{P}}{\Gamma, A' \vdash \mathsf{var}(\mathsf{vs}(t)) : A :: \mathsf{P}}$$

$$\frac{\Gamma \vdash f : (a \Rightarrow_{\mathsf{P}} A) \qquad \Gamma \vdash t : \mathsf{El}(a)}{\Gamma \vdash f(t) : A :: \mathsf{P}}$$

$$\frac{\Gamma \vdash f : \hat{\Pi}_{\mathsf{P}}(T, A) \qquad \tau : T}{\Gamma \vdash f(\tau) : A(\tau) :: \mathsf{P}}$$

Like with the sort substitutions defined in Definition 6.1.2, we define substitutions between point contexts over a fixed sort context $\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}$ to be lists of point terms:

$$\frac{\vdash_{\Gamma_{\mathsf{S}}} \Gamma}{\Gamma \xrightarrow{\epsilon_{\mathsf{P}}} \cdot} \qquad \frac{\Gamma \xrightarrow{\sigma_{\mathsf{P}}} \Delta \qquad \Gamma \vdash t : A :: \mathsf{P}}{\Gamma \xrightarrow{\sigma_{\mathsf{P}}, t} \Delta, A}$$

We can again define a pullback operation for terms – this time for point terms – along substitutions in the form of

$$\frac{\Delta \vdash_{\Gamma_{\mathsf{S}}} t : A :: \mathsf{P} \qquad \Gamma \xrightarrow{\sigma_{\mathsf{P}}} \Delta}{\Gamma \vdash_{\Gamma_{\mathsf{S}}} t[\sigma] : A :: \mathsf{P}}$$

which is recursively defined by

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})[\sigma_{\mathsf{P}}, t_{\mathsf{P}}] &:\equiv t_{\mathsf{P}}, \\
\mathsf{var}(\mathsf{vs}(v_{\mathsf{P}}))[\sigma_{\mathsf{P}}, t_{\mathsf{P}}] &:\equiv \mathsf{var}(v_{\mathsf{P}})[\sigma_{\mathsf{P}}], \\
f(t)[\sigma_{\mathsf{P}}] &:\equiv f[\sigma_{\mathsf{P}}](t[\sigma_{\mathsf{P}}]), \text{ and} \\
f(\tau)[\sigma_{\mathsf{P}}] &:\equiv f[\sigma_{\mathsf{P}}](\tau).
\end{aligned}$$

Analogously to 6.1.2 we can define the weakening $\Gamma, A \xrightarrow{\mathsf{wk}_{\sigma_{\mathsf{P}}}} \Delta$ of a point substitution $\Gamma \xrightarrow{\sigma_{\mathsf{P}}} \Delta$ along a point type $\Gamma_{\mathsf{S}} \vdash_{\mathsf{S}} A :: \mathsf{P}$, the identity substitution $\Gamma \xrightarrow{\mathsf{id}_{\mathsf{P}}} \Gamma$, and the Composition $\Gamma \xrightarrow{\sigma_{\mathsf{P}} \circ \delta_{\mathsf{P}}} \Sigma$ of substitutions $\Delta \xrightarrow{\sigma_{\mathsf{P}}} \Sigma$ and $\Gamma \xrightarrow{\delta_{\mathsf{P}}} \Delta$ causing the analogous effect when being used to pullback point terms:

$$\begin{aligned}
t_{\mathsf{P}}[\mathsf{wk}_{\sigma_{\mathsf{P}}}] &= \mathsf{vs}(t_{\mathsf{P}}[\sigma_{\mathsf{P}}]) \\
t_{\mathsf{P}}[\mathsf{id}] &= t_{\mathsf{P}} \\
t_{\mathsf{P}}[\sigma_{\mathsf{P}} \circ \delta_{\mathsf{P}}] &= t_{\mathsf{P}}[\sigma_{\mathsf{P}}][\delta_{\mathsf{P}}]
\end{aligned}$$

As an auxiliary construction for our existence proof we will furthermore need notions of algebra, displayed algebras, and sections for the point terms and point substitutions:

Definition 6.4.3 (Algebras of Point Substitutions & Terms)**.** We can give semantic meaning to point types and point substitution by extending the algebra operator with the following components, all over a fixed sort context $\vdash_S \Gamma_S$:

$$\frac{\Gamma \vdash_{\Gamma_S} t_P : A :: \mathsf{P}}{{t_P}^{\mathsf{A}} : \left\{\gamma_S : {\Gamma_S}^{\mathsf{A}}\right\} \to \Gamma^{\mathsf{A}}(\gamma_S) \to A^{\mathsf{A}}(\gamma_S)}$$

$$\frac{\Gamma \xrightarrow{\sigma_P} \Delta}{{\sigma_P}^{\mathsf{A}} : \left\{\gamma_S : {\Gamma_S}^{\mathsf{A}}\right\} \to \Gamma^{\mathsf{A}}(\gamma_S) \to \Delta^{\mathsf{A}}(\gamma_S)}$$

These components are, in essence, defined as iterated tuples and projections. For point terms, these defining equations are

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{A}}(\gamma, \alpha) &:\equiv \alpha, \\
\mathsf{var}(\mathsf{vs}(t))^{\mathsf{A}}(\gamma, \alpha) &:\equiv \mathsf{var}(t)^{\mathsf{A}}(\gamma), \\
f(t)^{\mathsf{A}}(\gamma) &:\equiv f^{\mathsf{A}}(\gamma)(t^{\mathsf{A}}(\gamma)), \text{ and} \\
f(\tau)^{\mathsf{A}}(\gamma) &:\equiv f^{\mathsf{A}}(\gamma)(\tau),
\end{aligned}$$

while for point contexts we have the usual

$$\begin{aligned}
{\epsilon_P}^{\mathsf{A}}(\gamma) &:\equiv \star \text{ and} \\
(\Gamma, A)^{\mathsf{A}}(\gamma) &:\equiv \left(\Gamma^{\mathsf{A}}(\gamma), A^{\mathsf{A}}(\gamma)\right).
\end{aligned}$$

Of course, apart from these defining equations, this definition of algebras is also well-behaved under the other components of substitutional calculus:

$$\begin{aligned}
t_P[\sigma_P]^{\mathsf{A}}(\gamma) &= {t_P}^{\mathsf{A}}({\sigma_P}^{\mathsf{A}}(\gamma)), \\
\mathsf{vs}(t_P)^{\mathsf{A}}(\gamma, \alpha) &= {t_P}^{\mathsf{A}}(\gamma), \\
{\mathsf{wk}_{\sigma_P}}^{\mathsf{A}}(\gamma, \alpha) &= {\sigma_P}^{\mathsf{A}}(\gamma), \\
{\mathsf{id}_P}^{\mathsf{A}}(\gamma) &= \gamma, \text{ and} \\
(\sigma_P \circ \delta_P)^{\mathsf{A}}(\gamma) &= {\sigma_P}^{\mathsf{A}}({\delta_P}^{\mathsf{A}}(\gamma)).
\end{aligned}$$

Definition 6.4.4 (Displayed Algebras of Point Terms & Subsitutions)**.** Let us for this definition fix a sort context $\vdash_S \Gamma_S$ with an algebra $\gamma_S : {\Gamma_S}^{\mathsf{A}}$ as well as a displayed algebra $\gamma_S^{\mathsf{D}} : {\Gamma_S}^{\mathsf{D}}(\gamma_S)$ over γ_S. For the displayed version of these algebras, the interpretation of point terms and of point substitutions needs to depend on these and, additionally, on an algebra and displayed algebra of the underlying point context. This leads to the addition of the following rules:

$$\frac{\Gamma \vdash_{\Gamma_{\mathsf{S}}} t_{\mathsf{P}} : A :: \mathsf{P}}{{t_{\mathsf{P}}}^{\mathsf{D}} : \left\{\gamma : \Gamma^{\mathsf{A}}(\gamma_{\mathsf{S}})\right\} \to \Gamma^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}, \gamma) \to A^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}, {t_{\mathsf{P}}}^{\mathsf{A}}(\gamma))}$$

$$\frac{\Gamma \xrightarrow{\sigma_{\mathsf{P}}} \Delta}{{\sigma_{\mathsf{P}}}^{\mathsf{D}} : \left\{\gamma : \Gamma^{\mathsf{A}}(\gamma_{\mathsf{S}})\right\} \to \Gamma^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}, \gamma) \to \Delta^{\mathsf{D}}(\gamma_{\mathsf{S}}^{\mathsf{D}}, {\sigma_{\mathsf{P}}}^{\mathsf{A}}(\gamma))}$$

We define them by setting

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &:\equiv \alpha^{\mathsf{D}}, \\
\mathsf{var}(\mathsf{vs}(t_{\mathsf{P}}))^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &:\equiv \mathsf{var}(t_{\mathsf{P}})^{\mathsf{D}}(\gamma^{\mathsf{D}}), \\
f_{\mathsf{P}}(t_{\mathsf{P}})^{\mathsf{D}}(\gamma^{\mathsf{D}}) &:\equiv {f_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}) \left({t_{\mathsf{P}}}^{\mathsf{A}}(\gamma), {t_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}})\right), \text{ and} \\
f_{\mathsf{P}}(\tau)^{\mathsf{D}}(\gamma^{\mathsf{D}}) &:\equiv {f_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}})(\tau) \text{ for terms, and} \\
{\epsilon_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}) &:\equiv \star \text{ and} \\
(\sigma_{\mathsf{P}}, t_{\mathsf{P}})^{\mathsf{D}}(\gamma^{\mathsf{D}}) &:\equiv \left({\sigma_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}), {t_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}})\right) \text{ for point substitutions.}
\end{aligned}$$

Again, substitution rules analogous to the ones in 6.4.3 hold:

$$\begin{aligned}
t_{\mathsf{P}}[\sigma_{\mathsf{P}}]^{\mathsf{D}}(\gamma^{\mathsf{D}}) &= {t_{\mathsf{P}}}^{\mathsf{D}}({\sigma_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}})), \\
\mathsf{vs}(t_{\mathsf{P}})^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &= {t_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}), \\
{\mathsf{wk}_{\sigma_{\mathsf{P}}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}, \alpha^{\mathsf{D}}) &= {\sigma_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}), \\
{\mathsf{id}_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}}) &= \gamma^{\mathsf{D}}, \text{ and} \\
(\sigma_{\mathsf{P}} \circ \delta_{\mathsf{P}})^{\mathsf{D}}(\gamma^{\mathsf{D}}) &= {\sigma_{\mathsf{P}}}^{\mathsf{D}}({\delta_{\mathsf{P}}}^{\mathsf{D}}(\gamma^{\mathsf{D}})).
\end{aligned}$$

As a next step after having extended our syntax and defined the semantics of this extension, we will show that any type theory with indexed W-types is able to represent the whole syntax for inductive families internally.

Remark 6.4.5. While for the signatures of inductive-inductive types, contexts, types, and terms depend on each other, we can here define sort types, sort contexts, terms, point types, and contexts in the presented order without referring to later constructions. This means that unlike mentioned in Remark 5.1.3, we can internalize this syntax just using inductive families, as shown in the following agda implementation:

```
data TyS : Set₁ where
  U  : TyS
  Π̂S : (T : Set) → (T → TyS) → TyS

data ConS : Set₁ where
  ·c    : ConS
  _▶c_ : ConS → TyS → ConS

data VarS : ConS → TyS → Set₁ where
  vvz : ∀{Γc B} → Var (Γc ▶c B) B
  vvs : ∀{Γc B B'} → Var Γc B → Var (Γc ▶c B') B

data TmS (Γc : ConS) : TyS → Set₁ where
  var   : ∀{A} → Var Γc A → TmS Γc A
  _@S_ : ∀{T B} → TmS Γc (Π̂S T B) → (τ : T) → TmS Γc (B τ)

data TyP (Γc : ConS) : Set₁ where
  El    : TmS Γc U → TyP Γc
  Π̂P    : (T : Set) → (T → TyP Γc) → TyP Γc
  _⇒P_ : TmS Γc U → TyP Γc → TyP Γc

data Con (Γc : ConS) : Set₁ where
  ·     : Con Γc
  _▶P_ : Con Γc → TyP Γc → Con Γc
```

Note, that in the implementaion, variables and terms are defined in separate types to allow for var(v) to appear as a premise for the introduction rule for vs(v). The extension of the syntax by sort substitutions of Definition 6.1.2 as well as the subsequent extension by point terms and point substitutions as presented in Definition 6.4.2 is implementable as well:

```
data SubS : ConS → ConS → Set₁ where
  ε   : ∀{Γc} → SubS Γc ·c
  _,_ : ∀{Γc Δc B} → SubS Γc Δc → TmS Γc B → SubS Γc (Δc ▶c B)

data VarP {Γc} : Con Γc → TyP Γc → Set₁ where
  vvzP : ∀{Γ A} → VarP (Γ ▶P A) A
  vvsP : ∀{Γ A B} → VarP Γ A → VarP (Γ ▶P B) A

data TmP {Γc}(Γ : Con Γc) : TyP Γc → Set₁ where
  varP : ∀{A} → VarP Γ A → TmP Γ A
  _@P_ : ∀{a A} → TmP Γ (a ⇒P A) → TmP Γ (El a) → TmP Γ A
  _^@P_ : ∀{T A} → TmP Γ (Π̂P T A) → (τ : T) → TmP Γ (A τ)

```

```
data SubP {Γc} : Con Γc → Con Γc → Set₁ where
  εP   : ∀{Γ} → SubP Γ ·
  _,P_ : ∀{Γ Δ A} → SubP Γ Δ → TmP Γ A → SubP Γ (Δ ▸P A)
```

To make this proof of internalizability formal, we will present the exact definition of all the parts of the extended syntax as indexed W-types. This means giving a type $Con_S : \mathcal{U}$ of sort contexts, a type $Ty_S : \mathcal{U}$ of sort types, a family $Var_S : Con_S \to Ty_S \to \mathcal{U}$ of variables of a given sort context and sort type, extending the latter, a family $W_{Tm_S} : Con_S \to Ty_S \to \mathcal{U}$ of sort terms, a family $Ty_P : Con_S \to \mathcal{U}$ of point types in a given sort context, and finally a type $Con_P : Con_S \to \mathcal{U}$ of point contexts over a given sort context. Afterwards, we will give the same treatment to the extensions with sort substitutions between two sort contexts, with variables of a point type, terms of a point type and point substitution between two point contexts over the same sort context in the form of

$$\begin{aligned}
Sub_S &: Con_S \to Con_S \to \mathcal{U}, \\
Var_P &: \{\Gamma_S : Con_S\} \to Con_P(\Gamma_S) \to Ty_P(\Gamma_S) \to \mathcal{U}, \\
Tm_P &: \{\Gamma_S : Con_S\} \to Con_P(\Gamma_S) \to Ty_P(\Gamma_S) \to \mathcal{U}, \text{ and} \\
Sub_P &: \{\Gamma_S : Con_S\} \to Con_P(\Gamma_S) \to Con_P(\Gamma_S) \to \mathcal{U}.
\end{aligned}$$

In the following definition we will give all of these ten types and type families in general by giving the respective input data for indexed W-types as described in Chapter 2.2.

Definition 6.4.6 (IF-Syntax as W-Types)**.** We define the types mentioned above as follows:

$$\begin{aligned}
Ty_S &:\equiv \mathsf{IW}^{o_{Ty_S}, r_{Ty_S}}_{A_{Ty_S}, B_{Ty_S}}(\star), & Ty_P(\Gamma_S) &:\equiv \mathsf{IW}^{o_{Ty_P}, r_{Ty_P}}_{A_{Ty_P}, B_{Ty_P}}(\star), \\
Con_S &:\equiv \mathsf{IW}^{o_{Con_S}, r_{Con_S}}_{A_{Con_S}, B_{Con_S}}(\star), & Con_P(\Gamma_S) &:\equiv \mathsf{IW}^{o_{Con_P}, r_{Con_P}}_{A_{Con_P}, B_{Con_P}}(\star), \\
Var_S(\_, B) &:\equiv \mathsf{IW}^{o_{Var_S}(B), r_{Var_S}(B)}_{A_{Var_S}(B), B_{Var_S}(B)}, & Var_P(\_, A) &:\equiv \mathsf{IW}^{o_{Var_P}(A), r_{Var_P}(A)}_{A_{Var_P}(A), B_{Var_P}(A)}, \\
W_{Tm_S}(\Gamma_S) &:\equiv \mathsf{IW}^{o_{Tm_S}(\Gamma_S), r_{Tm_S}(\Gamma_S)}_{A_{Tm_S}(\Gamma_S), B_{Tm_S}(\Gamma_S)}, & Tm_P(\Gamma) &:\equiv \mathsf{IW}^{o_{Tm_P}(\Gamma), r_{Tm_P}(\Gamma)}_{A_{Tm_P}(\Gamma), B_{Tm_P}(\Gamma)}, \\
Sub_S(\Gamma_S) &:\equiv \mathsf{IW}^{o_{Sub_S}(\Gamma_S), r_{Sub_S}(\Gamma_S)}_{A_{Sub_S}(\Gamma_S), B_{Sub_S}(\Gamma_S)}, & Sub_P(\Gamma) &:\equiv \mathsf{IW}^{o_{Sub_P}(\Gamma), r_{Sub_P}(\Gamma)}_{A_{Sub_P}(\Gamma), B_{Sub_P}(\Gamma)},
\end{aligned}$$

where the respective indices for the indexed W-types are given in Table 6.1.

i	$I_i : \mathcal{U}$	$A_i : \mathcal{U}$	$B_i : A_i \to \mathcal{U}$	$o_i : A_i \to I_i$	$r_i : (a : A_i) \to B_i(a) \to I_i$
Ty_S	$\mathbf{1}$	$\mathbf{1}$ $+\mathcal{U}$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(T) \mapsto T$	$\_ \mapsto \star$	$\_ \mapsto \star$
Con_S	$\mathbf{1}$	$\mathbf{1}$ $+\mathit{Ty}_\mathsf{S}$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(B) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$
Var_S	Con_S	Con_S $+\mathit{Con}_\mathsf{S} \times W_{\mathit{Ty}_\mathsf{S}}$	$\mathsf{inl}(\Gamma_\mathsf{S}) \mapsto \mathbf{0}$ $\mathsf{inr}(\Gamma_\mathsf{S}, B') \mapsto \mathbf{1}$	$\mathsf{inl}(\Gamma_\mathsf{S}) \mapsto (\Gamma_\mathsf{S}, B)$ $\mathsf{inr}(\Gamma_\mathsf{S}, B') \mapsto (\Gamma_\mathsf{S}, B')$	$-$ $\mathsf{inr}(\Gamma_\mathsf{S}, B')(\star) \mapsto \Gamma_\mathsf{S}$
$\mathit{Tm}_\mathsf{S}(\Gamma_\mathsf{S})$	Ty_S	Ty_S $+(T : \mathcal{U}) \times (T \to \mathit{Ty}_\mathsf{S}) \times T$	$\mathsf{inl}(B) \mapsto \mathbf{0}$ $\mathsf{inr}(\_) \mapsto \mathbf{1}$	$\mathsf{inl}(B) \mapsto \mathbf{0}$ $\mathsf{inr}(T, B, \tau) \mapsto B(\tau)$	$-$ $\mathsf{inr}(T, B, \tau)(\star) \mapsto \hat{\Pi}_\mathsf{S}(T, B)$
$\mathit{Sub}_\mathsf{S}(\Gamma_\mathsf{S})$	$\mathbf{1}$	$\mathbf{1}$ $+(B : \mathit{Ty}_\mathsf{S}) \times \mathit{Tm}_\mathsf{S}(\Gamma_\mathsf{S}, B)$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(B, t) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$
$\mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})$	$\mathbf{1}$	$\mathit{Tm}_\mathsf{S}(\Gamma, \mathcal{U})$ $+\mathcal{U}$ $+W_{\mathit{Tm}_\mathsf{S}}(\Gamma, \mathcal{U})$	$\mathsf{inl}(a) \mapsto \mathbf{0}$ $\mathsf{inr}(\mathsf{inl}(\tau)) \mapsto T$ $\mathsf{inr}(\mathsf{inr}(a)) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$
$\mathit{Con}_\mathsf{P}(\Gamma_\mathsf{S})$	$\mathbf{1}$	$\mathbf{1}$ $+\mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(A) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$
$\mathit{Var}_\mathsf{P}(A)$	$\mathit{Con}_\mathsf{P}(\Gamma_\mathsf{S})$	$\mathit{Con}_\mathsf{P}(\Gamma_\mathsf{S})$ $+\mathit{Con}_\mathsf{P}(\Gamma_\mathsf{S}) \times \mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})$	$\mathsf{inl}(\Gamma) \mapsto \mathbf{0}$ $\mathsf{inr}(\Gamma, A') \mapsto \mathbf{1}$	$\mathsf{inl}(\Gamma) \mapsto (\Gamma, A)$ $\mathsf{inr}(\Gamma, A') \mapsto (\Gamma, A')$	$-$ $\mathsf{inr}(\Gamma, A')(\star) \mapsto \Gamma$
$\mathit{Tm}_\mathsf{P}(\Gamma)$	$\mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})$	$(A : \mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})) \times \mathit{Var}_\mathsf{P}(\Gamma, A)$	$\mathsf{inl}(A, v) \mapsto \mathbf{0}$ $\mathsf{inr}(\mathsf{inl}(\_)) \mapsto \mathbf{2}$ $\mathsf{inr}(\mathsf{inr}(\_)) \mapsto \mathbf{1}$	$\mathsf{inl}(A, v) \mapsto A$ $\mathsf{inr}(\mathsf{inl}(A, a)) \mapsto A$ $\mathsf{inr}(\mathsf{inr}(T, A, \tau)) \mapsto A(\tau)$	$-$ $\mathsf{inr}(\mathsf{inl}(A, a))(0) \mapsto (a \Rightarrow_\mathsf{P} A)$ $\mathsf{inr}(\mathsf{inl}(A, a))(1) \mapsto \mathsf{El}(a)$ $\mathsf{inr}(\mathsf{inr}(T, A, \tau))(\star) \mapsto \hat{\Pi}_\mathsf{P}(T, A)$
$\mathit{Sub}_\mathsf{P}(\Gamma)$	$\mathbf{1}$	$\mathbf{1}$ $+(A : \mathit{Ty}_\mathsf{P}(\Gamma_\mathsf{S})) \times \mathit{Tm}_\mathsf{P}(\Gamma, A)$	$\mathsf{inl}(\star) \mapsto \mathbf{0}$ $\mathsf{inr}(A, t) \mapsto \mathbf{1}$	$\_ \mapsto \star$	$\_ \mapsto \star$

Table 6.1: The input data for the indexed W-types representing the internalized syntax for inductive families.

6.4.2 Constructing the Term Model

For the remainder of this section, let us fix the sort context $\vdash_{\mathsf{S}} \Omega_{\mathsf{S}}$ and the point context $\vdash_{\Omega_{\mathsf{S}}} \Omega$ which we want to construct by giving

$$\begin{aligned} \mathsf{con}_{\mathsf{S}}(\Omega) &: {\Omega_{\mathsf{S}}}^{\mathsf{A}} \text{ and} \\ \mathsf{con}(\Omega) &: \Omega^{\mathsf{A}}(\mathsf{con}_{\mathsf{S}}(\Omega)). \end{aligned}$$

Our definition of the constructor uses the trick to index several of the constructions by a second sort or point context together with a sort or point substitution from Ω_{S} or Ω. We can think of this second context as some sort of a "sub-context" of a fixed context.

Definition 6.4.7 (The Sort Constructor)**.** The generalized sort constructor consists of the following data:

$$\frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}} \qquad \Omega_{\mathsf{S}} \xrightarrow{\sigma} \Gamma_{\mathsf{S}}}{\mathsf{con}_{\mathsf{S}}(\sigma) : {\Gamma_{\mathsf{S}}}^{\mathsf{A}}}$$

We can define this recursively via

$$\begin{aligned} \mathsf{con}_{\mathsf{S}}(\epsilon) &:\equiv \star \text{ and} \\ \mathsf{con}_{\mathsf{S}}(\sigma, t) &:\equiv (\mathsf{con}_{\mathsf{S}}(\sigma), \mathsf{con}_{\mathsf{S}}(t)), \end{aligned}$$

where on sort terms we will define a constructor operation yielding an algebra of the respective sort type:

$$\frac{\Omega_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S}}{\mathsf{con}_{\mathsf{S}}(t) : B^{\mathsf{A}}}$$

This operation will on universe terms consist of the type of point terms, while on external sort functions, it will return a function with constructor of the applied term:

$$\begin{aligned} \mathsf{con}_{\mathsf{S}}(a) &:\equiv \mathit{Tm}_{\mathsf{P}}(\Omega, \mathsf{El}(a)) && \text{for } \Omega_{\mathsf{S}} \vdash_{\mathsf{S}} a : \mathcal{U} :: \mathsf{S} \text{ and} \\ \mathsf{con}_{\mathsf{S}}(f) &:\equiv \lambda\tau.\, \mathsf{con}_{\mathsf{S}}(f(\tau)) && \text{for } \Omega_{\mathsf{S}} \vdash_{\mathsf{S}} f : \hat{\Pi}_{\mathsf{S}}(T, B) :: \mathsf{S}. \end{aligned}$$

This construction is already enough to give the sort constructor required in Theorem 6.4.1 by pinning the substitution to be the identity:

$$\mathsf{con}_{\mathsf{S}}(\Omega) :\equiv \mathsf{con}_{\mathsf{S}}(\mathsf{id}_{\Omega_{\mathsf{S}}}) : {\Omega_{\mathsf{S}}}^{\mathsf{A}} \tag{6.1}$$

It is not immediately clear that the operation on substitutions and the operation on sort terms is well-behaved under the pullback along substitutions. We can, however, show that this is indeed the case.

Lemma 6.4.8 (Coherence of the Sort Constructor)**.** *For all subsitutions $\Gamma_\mathsf{S} \xrightarrow{\sigma} \Delta_\mathsf{S}$ and sort terms $\Gamma_\mathsf{S} \vdash_\mathsf{S} t : B :: \mathsf{S}$, taking a constructor of t pulled back along σ has the same effect as taking the term algebra over the context algebra generated by the constructor on σ, i. e.*

$$t^\mathsf{A}(\mathsf{con_S}(\sigma)) = \mathsf{con_S}(t[\sigma]).$$

Proof. Let us first do a case distinction on the substitution. If it is ϵ, then $\Gamma_\mathsf{S} = \cdot_\mathsf{S}$, and it is easy to see that there are no terms in the empty sort context. Thus, we can assume the substitution to be of the form (σ, s). In this case, lets recurse on the term and see that

$$\begin{aligned}\mathsf{var}(\mathsf{vz})^\mathsf{A}(\mathsf{con_S}(\sigma, s)) &= \mathsf{var}(\mathsf{vz})^\mathsf{A}(\mathsf{con_S}(\sigma), \mathsf{con_S}(s)) \\ &= \mathsf{con_S}(s) \\ &= \mathsf{con_S}(\mathsf{var}(\mathsf{vz})[\sigma, s]),\end{aligned}$$

$$\begin{aligned}\mathsf{var}(\mathsf{vs}(t))^\mathsf{A}(\mathsf{con_S}(\sigma, s)) &= \mathsf{var}(\mathsf{vs}(t))^\mathsf{A}(\mathsf{con_S}(\sigma), \mathsf{con_S}(s)) \\ &= \mathsf{var}(t)^\mathsf{A}(\mathsf{con_S}(\sigma)) \\ &= \mathsf{con_S}(\mathsf{var}(t)[\sigma]) && \text{by induction} \\ &= \mathsf{con_S}(\mathsf{var}(\mathsf{vs}(t))[\sigma, s]), && \text{and lastly}\end{aligned}$$

$$\begin{aligned}f(\tau)^\mathsf{A}(\mathsf{con_S}(\sigma, s)) &= f^\mathsf{A}(\mathsf{con_S}(\sigma, s))(\tau) \\ &= \mathsf{con_S}(f[\sigma, s])(\tau) && \text{by induction} \\ &= \mathsf{con_S}(f(\tau)[\sigma, s]) && \text{for } f : \hat{\Pi}_\mathsf{S}(T, B).\end{aligned}$$

□

We can now use this lemma to be able to do a trick with $\mathsf{con}(\Omega)$ similar to the trick we did for $\mathsf{con_S}(\Omega)$: Replace the fixed point context with a variable one, together with a substitution from Ω, and define the constructor recursively on point types.

Definition 6.4.9 (The Point Constructor)**.** We define operations on point contexts and point terms, resulting in algebras, in the form of the following:

$$\frac{\vdash_{\Gamma_\mathsf{S}} \Gamma \qquad \Omega \xrightarrow{\sigma_\mathsf{P}} \Gamma}{\mathsf{con}(\sigma_\mathsf{P}) : \Gamma^\mathsf{A}(\mathsf{con_S}(\Omega))} \qquad \frac{\Omega \vdash_{\Omega_\mathsf{S}} t_\mathsf{P} : A :: \mathsf{P}}{\mathsf{con}(t_\mathsf{P}) : A^\mathsf{A}(\mathsf{con_S}(\Omega))}$$

The operation on point substitutions is defined recursively by

$$\begin{aligned}\mathsf{con}(\epsilon_\mathsf{P}) &:\equiv \star \text{ and} \\ \mathsf{con}(\sigma_\mathsf{P}, t_\mathsf{P}) &:\equiv (\mathsf{con}(\sigma_\mathsf{P}), \mathsf{con}(t_\mathsf{P})),\end{aligned}$$

wheres for point terms, note that if $\Omega \vdash t_{\mathsf{P}} : \mathsf{El}(a) :: \mathsf{P}$, then by Lemma 6.4.8

$$t_{\mathsf{P}} : \mathsf{con}_{\mathsf{S}}(a) \equiv \mathsf{con}_{\mathsf{S}}(a[\mathsf{id}]) = a^{\mathsf{A}}(\mathsf{con}_{\mathsf{S}}(\mathsf{id}_{\Omega_{\mathsf{S}}})) \equiv \mathsf{El}(a)^{\mathsf{A}}(\mathsf{con}_{\mathsf{S}}(\Omega)),$$

which allows us to define the constructor operator by

$$\begin{aligned} \mathsf{con}(t_{\mathsf{P}}) &:\equiv t_{\mathsf{P}} && \text{for } \Omega \vdash t_{\mathsf{P}} : \mathsf{El}(a), \\ \mathsf{con}(f_{\mathsf{P}}) &:\equiv \lambda t_{\mathsf{P}}.\,\mathsf{con}(f_{\mathsf{P}}(t_{\mathsf{P}})) && \text{for } \Omega \vdash f_{\mathsf{P}} : a \Rightarrow_{\mathsf{P}} A, \text{ and} \\ \mathsf{con}(f_{\mathsf{P}}) &:\equiv \lambda \tau.\,\mathsf{con}(f_{\mathsf{P}}(\tau)) && \text{for } \Omega \vdash f_{\mathsf{P}} : \hat{\Pi}_{\mathsf{P}}(T, A). \end{aligned}$$

This concludes the definition of the constructors, since we can set, like for the sort constructor

$$\mathsf{con}(\Omega) :\equiv \mathsf{con}(\mathsf{id}_{\Omega}) : \Omega^{\mathsf{A}}(\mathsf{con}_{\mathsf{S}}(\Omega)). \tag{6.2}$$

Again, the construction comes with a property that makes it coherent under pulled back point terms. Analogously to Lemma 6.4.8, this coherence looks as follows:

Lemma 6.4.10 (Coherence of the Point Constructor)**.** *For all point subsitutions $\Omega \xrightarrow{\sigma_{\mathsf{P}}} \Delta$ and point terms $\Gamma \vdash_{\Omega_{\mathsf{S}}} t_{\mathsf{P}} : A :: \mathsf{P}$, pulling back has the same effect as the point constructor as in*

$$t_{\mathsf{P}}{}^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}})) = \mathsf{con}(t_{\mathsf{P}}[\sigma_{\mathsf{P}}]). \tag{6.3}$$

Proof. Repeating the strategy of the proof of Lemma 6.4.8, we again see that we can assume the substitution to be of an extended form $(\sigma_{\mathsf{P}}, s_{\mathsf{P}})$, since there are no point terms in the empty point context. Now, by recursion on the term we see that

$$\begin{aligned}
\mathsf{var}(\mathsf{vz})^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}, s_{\mathsf{P}})) &= \mathsf{var}(\mathsf{vz})^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}), \mathsf{con}(s_{\mathsf{P}})) \\
&= \mathsf{con}(s_{\mathsf{P}}) \\
&= \mathsf{con}(\mathsf{var}(\mathsf{vz})[\sigma_{\mathsf{P}}, s_{\mathsf{P}}]),
\end{aligned}$$

$$\begin{aligned}
\mathsf{var}(\mathsf{vs}(t_{\mathsf{P}}))^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}, s_{\mathsf{P}})) &= \mathsf{var}(\mathsf{vs}(t_{\mathsf{P}}))^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}), \mathsf{con}(s_{\mathsf{P}})) \\
&= \mathsf{var}(t_{\mathsf{P}})^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}})) \\
&= \mathsf{con}(\mathsf{var}(t_{\mathsf{P}})[\sigma_{\mathsf{P}}]) && \text{by induction} \\
&= \mathsf{con}(\mathsf{var}(\mathsf{vs}(t_{\mathsf{P}}))[\sigma_{\mathsf{P}}, s_{\mathsf{P}}]),
\end{aligned}$$

$$\begin{aligned}
f_{\mathsf{P}}(t_{\mathsf{P}})^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}})) &= f_{\mathsf{P}}{}^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}))\left(t_{\mathsf{P}}{}^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}))\right) \\
&= \mathsf{con}(f_{\mathsf{P}}[\sigma_{\mathsf{P}}])(\mathsf{con}(t_{\mathsf{P}}[\sigma_{\mathsf{P}}])) && \text{by induction} \\
&= \mathsf{con}(f_{\mathsf{P}}(t_{\mathsf{P}})[\sigma_{\mathsf{P}}]), \text{ and}
\end{aligned}$$

$$\begin{aligned}
f_{\mathsf{P}}(\tau)^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}})) &= f_{\mathsf{P}}{}^{\mathsf{A}}(\mathsf{con}(\sigma_{\mathsf{P}}))(\tau) \\
&= \mathsf{con}(f_{\mathsf{P}}[\sigma_{\mathsf{P}}])(\tau) && \text{by induction} \\
&= \mathsf{con}(f_{\mathsf{P}}(\tau)[\sigma_{\mathsf{P}}]).
\end{aligned}$$

□

With the constructors defined let us move on the construction of the eliminator. Let us from now on fix displayed algebras $\omega_{\mathsf{S}}^{\mathsf{D}} : \Omega_{\mathsf{S}}{}^{\mathsf{D}}(\mathsf{cons}_{\mathsf{S}}(\Omega))$ and $\omega^{\mathsf{D}} : \Omega^{\mathsf{D}}(\omega_{\mathsf{S}}^{\mathsf{D}}, \mathsf{con}(\Omega))$. We will proceed in the same order as for the constructors and start by generalizing $\mathsf{elim}_{\mathsf{S}}(\Omega, \omega^{\mathsf{D}})$ to arbitrary subcontexts of Ω by giving constructions on sort substitutions and sort terms.

Definition 6.4.11 (The Eliminator)**.** The generalized eliminator will take substitutions or sort terms to give sections of sort types or sort contexts, respectively:

$$\frac{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}} \qquad \Omega_{\mathsf{S}} \xrightarrow{\sigma} \Gamma_{\mathsf{S}}}{\mathsf{elim}_{\mathsf{S}}(\sigma) : \Gamma_{\mathsf{S}}{}^{\mathsf{S}}(\sigma^{\mathsf{D}}(\omega_{\mathsf{S}}^{\mathsf{D}}))} \qquad \frac{\Omega_{\mathsf{S}} \vdash_{\mathsf{S}} t : B :: \mathsf{S}}{\mathsf{elim}_{\mathsf{S}}(t) : B^{\mathsf{S}}(t^{\mathsf{D}}(\omega_{\mathsf{S}}^{\mathsf{D}}))}$$

The first rule is defined by recursion using the second construction as usual:

$$\begin{aligned}
\mathsf{elim}_{\mathsf{S}}(\epsilon) &:\equiv \star \text{ and} \\
\mathsf{elim}_{\mathsf{S}}(\sigma, t) &:\equiv (\mathsf{elim}_{\mathsf{S}}(\sigma), \mathsf{elim}_{\mathsf{S}}(t)).
\end{aligned}$$

For the sort terms, we observe that, by Lemmas 6.4.8 and 6.4.10, for $\Omega_\mathsf{S} \vdash_\mathsf{S} a : \mathcal{U}$ and $t_\mathsf{P} : a^\mathsf{A}(\mathsf{con}_\mathsf{S}(\Omega))$ we have

$$\mathcal{U}^\mathsf{S}(a^\mathsf{D}(\omega_\mathsf{S}^\mathsf{D}), {t_\mathsf{P}}^\mathsf{A}(\mathsf{con}(\Omega))) = a^\mathsf{D}(\omega_\mathsf{S}^\mathsf{D}, t_\mathsf{P})$$

and thus we can set, disregarding transports,

$$\begin{aligned} \mathsf{elim}_\mathsf{S}(a) &:\equiv \lambda t_\mathsf{P}.\, {t_\mathsf{P}}^\mathsf{D}(\omega^\mathsf{D}) && \text{for } \Omega_\mathsf{S} \vdash_\mathsf{S} a : \mathcal{U} \text{ and} \\ \mathsf{elim}_\mathsf{S}(f) &:\equiv \lambda \tau.\, \mathsf{elim}_\mathsf{S}(f(\tau)) && \text{for } \Omega_\mathsf{S} \vdash_\mathsf{S} f : \hat{\Pi}_\mathsf{S}(T, B). \end{aligned}$$

Similar to Lemma 6.4.8, these definitions are coherent in the following form:

Lemma 6.4.12. *Given a sort substitution* $\Omega_\mathsf{S} \xrightarrow{\sigma} \Gamma_\mathsf{S}$ *and a sort term* $\Gamma_\mathsf{S} \vdash_\mathsf{S} t : B :: \mathsf{S}$, *the eliminator of a pulled back term is the section of the term, evaluated at the eliminator on a substitution:*

$$\mathsf{elim}_\mathsf{S}(t[\sigma]) = t^\mathsf{S}(\mathsf{elim}_\mathsf{S}(\sigma)).$$

Proof. The proof strategy is exactly the same as for Lemma 6.4.8. □

As a last step, we still need to prove the computation rules for the eliminator, consisting of section of given point contexts. Consistent with 6.4.7, we generalize them to arbitrary point substitutions and point terms.

Lemma 6.4.13 (Computation Rules). *We prove the computation rule for our eliminator* $\mathsf{elim}_\mathsf{S}(\Omega)$ *to be a section of subcontexts of* Ω *and on point terms of* Ω*:*

$$\frac{\vdash_{\Omega_\mathsf{S}} \Gamma \qquad \Omega \xrightarrow{\sigma_\mathsf{P}} \Gamma}{\mathsf{elim}(\sigma_\mathsf{P}) : \Gamma^\mathsf{S}(\mathsf{elim}_\mathsf{S}(\Omega), {\sigma_\mathsf{P}}^\mathsf{D}(\omega^\mathsf{D}))}$$

$$\frac{\Omega \vdash_{\Omega_\mathsf{S}} t_\mathsf{P} : A :: \mathsf{P}}{\mathsf{elim}(t_\mathsf{P}) : A^\mathsf{S}(\mathsf{elim}_\mathsf{S}(\Omega), {t_\mathsf{P}}^\mathsf{D}(\omega^\mathsf{D}))}$$

Proof. Using the second rule, the first one can be proved in a straightforward way by recursion on the point substitution:

$$\begin{aligned} \mathsf{elim}(\epsilon_\mathsf{P}) &:\equiv \star \text{ and} \\ \mathsf{elim}(\sigma_\mathsf{P}, t_\mathsf{P}) &:\equiv (\mathsf{elim}(\sigma_\mathsf{P}), \mathsf{elim}(t_\mathsf{P}))\,. \end{aligned}$$

For the second rule we again need to consider the types needed for the element case. The previous lemmas tell us that for $\Omega \vdash t_{\mathsf{P}} : \mathsf{El}(a) :: \mathsf{P}$ we can prove the required rule by

$$\begin{aligned}
&a^{\mathsf{S}}\left(\mathsf{elim}_{\mathsf{S}}(\mathsf{id}_{\Omega_{\mathsf{S}}}), {t_{\mathsf{P}}}^{\mathsf{A}}(\mathsf{con}(\Omega))\right) \\
=&a^{\mathsf{S}}(\mathsf{elim}_{\mathsf{S}}(\mathsf{id}_{\Omega_{\mathsf{S}}}), t_{\mathsf{P}}) && \text{by Lemma 6.4.10} \\
=&\mathsf{elim}_{\mathsf{S}}(a)(t_{\mathsf{P}}) && \text{by Lemma 6.4.12} \\
=&{t_{\mathsf{P}}}^{\mathsf{D}}(\omega^{\mathsf{D}}).
\end{aligned}$$

For the case of $\Omega \vdash f_{\mathsf{P}} : \hat{\Pi}_{\mathsf{P}}(T, A)$, we see that we can recursively define $\mathsf{elim}(f_{\mathsf{P}})$ by proving $\mathsf{elim}(f_{\mathsf{P}}(\tau))$ for all $\tau : T$. Likewise in the case of a recursive function term $\Omega \vdash f_{\mathsf{P}} : a \Rightarrow_{\mathsf{P}} A$, we prove $\mathsf{elim}(f_{\mathsf{P}})$ recursively by $\mathsf{elim}(f_{\mathsf{P}}(t_{\mathsf{P}}))$. $\square$

Proof of Theorem 6.4.1. Lemma 6.4.13 completes the construction of the eliminator and setting

$$\begin{aligned}
\mathsf{elim}_{\mathsf{S}}(\Omega, \omega^{\mathsf{D}}) &:\equiv \mathsf{elim}_{\mathsf{S}}(\mathsf{id}_{\Omega_{\mathsf{S}}}) \text{ and} \\
\mathsf{elim}(\Omega, \omega^{\mathsf{D}}) &:\equiv \mathsf{elim}(\mathsf{id}_{\Omega})
\end{aligned}$$

completes the existence proofs for our specification of inductive families. $\square$

Example 6.4.14 (Natural numbers)**.** To give insight on how this construction works for a concrete example, let us look at the type of natural numbers, which we have represented by a signature in Example 6.1.1: Let us look at the construction at the following sort and point context:

$$\begin{aligned}
\Omega_{\mathsf{S}} &\equiv (\cdot_{\mathsf{S}}, \mathcal{U}) \\
\Omega &\equiv \mathsf{El}(\mathsf{var}(\mathsf{vz})), \mathsf{var}(\mathsf{vz}) \Rightarrow_{\mathsf{P}} \mathsf{El}(\mathsf{var}(\mathsf{vz})).
\end{aligned}$$

The sort constructor computes as follows

$$\begin{aligned}
\mathsf{con}_{\mathsf{S}}(\Omega) &\equiv \mathsf{con}_{\mathsf{S}}(\epsilon, \mathsf{var}(\mathsf{vz})) \\
&\equiv (\star, \mathsf{con}_{\mathsf{S}}(\mathsf{var}(\mathsf{vz}))) \\
&\equiv (\star, Tm_{\mathsf{P}}(\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})))) \\
&\equiv (\star, \{\mathsf{var}_{\mathsf{P}}(\mathsf{vs}(\mathsf{vz})), \mathsf{var}_{\mathsf{P}}(\mathsf{vz})(\mathsf{var}_{\mathsf{P}}(\mathsf{vs}(\mathsf{vz}))), \ldots\}) \\
&= (\star, \{\mathsf{var}_{\mathsf{P}}(\mathsf{vz})^{n}(\mathsf{var}_{\mathsf{P}}(\mathsf{vs}(\mathsf{vz}))) \mid n : \mathbb{N}\}),
\end{aligned}$$

while the point constructor gives the following:

$$\begin{aligned}
\mathsf{con}(\Omega) &\equiv \mathsf{con}(\epsilon, \mathsf{var}(\mathsf{vz}), \mathsf{var}(\mathsf{vs}(\mathsf{vz}))) \\
&\equiv (\star, \mathsf{var}_{\mathsf{P}}(\mathsf{vs}(\mathsf{vz})), \lambda n.\, \mathsf{var}_{\mathsf{P}}(\mathsf{vz})(n)).
\end{aligned}$$

Chapter 7

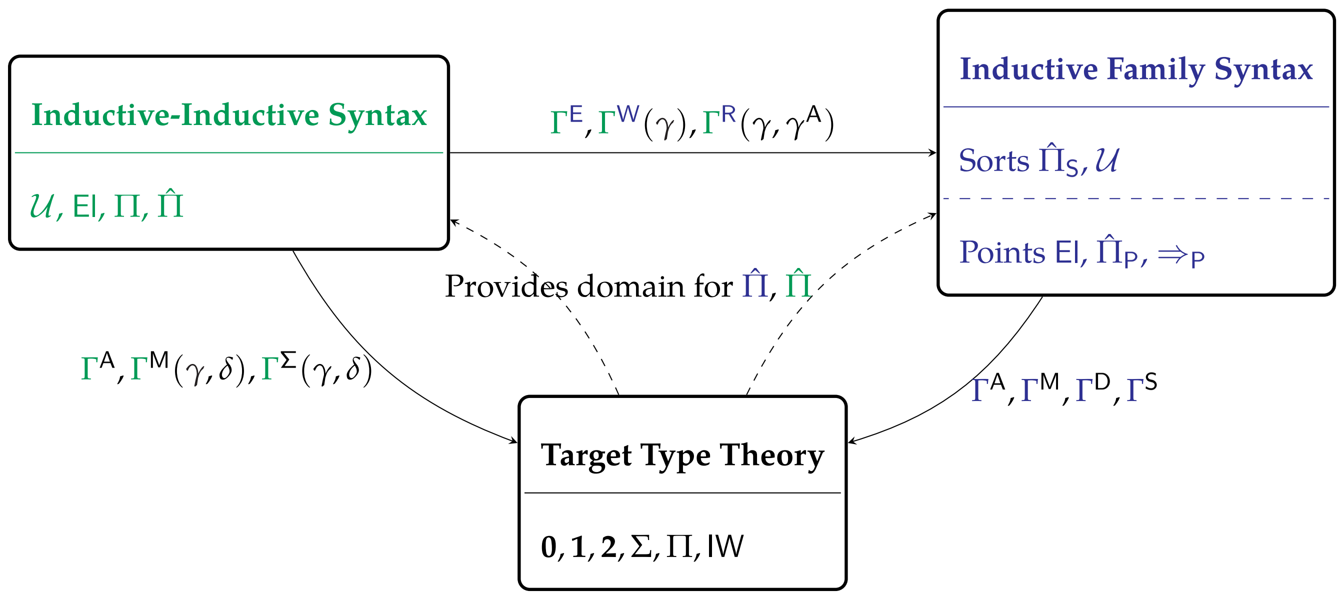
Reducing Inductive-Inductive Types to Inductive Families

We have now learned how we can express all inductive-inductive types and all inductive families by signatures consisting of a context in a type theory which is made specifically for this purpose. Now we want to pursue the question of whether every inductive-inductive type can be represented in a type theory that only supports (indexed) W-types, and thus inductive families.

To explore how we can transform an inductive-inductive signature into a sequence of constructions of inductive families, we will have to deal with four type theories: We imagine that we live in an *ambient type theory* in which all of our constructions will take place. This type theory must be powerful enough to represent the syntaxes of the other type theory we use, and thus should support quotient inductive types. Then, we have what we call *target type theory*, which is the language in which the reduced types should be available. This language must at least contain indexed W-types. At last, we also have the two "domain specific" type theories which we use to encode inductive-inductive types and inductive families. A graphical overview of the relations between these last three type theories can be seen in Figure 7.1.

The end goal of the reduction is to construct for any given signature Γ of an inductive-inductive type an object $\mathsf{con}(\Gamma)$, such that it is *initial* as defined in Definition 5.3.2: For any other given algebra $\gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}$ of the signature we get a morphism

$$\mathsf{elim}(\Gamma, \gamma^{\mathsf{A}}) : \Gamma^{\mathsf{M}}(\mathsf{con}(\Gamma), \gamma^{\mathsf{A}}).$$



Inductive-Inductive Syntax
$\mathcal{U}$, El, Π, $\hat{\Pi}$
Inductive Family Syntax
Sorts $\hat{\Pi}_{\mathsf{S}}, \mathcal{U}$
Points El, $\hat{\Pi}_{\mathsf{P}}$, $\Rightarrow_{\mathsf{P}}$
Target Type Theory
$\mathbf{0}, \mathbf{1}, \mathbf{2}, \Sigma, \Pi, \mathsf{IW}$
$\Gamma^{\mathsf{E}}, \Gamma^{\mathsf{W}}(\gamma), \Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})$
Provides domain for $\hat{\Pi}$, $\hat{\Pi}$
$\Gamma^{\mathsf{A}}, \Gamma^{\mathsf{M}}(\gamma, \delta), \Gamma^{\Sigma}(\gamma, \delta)$
$\Gamma^{\mathsf{A}}, \Gamma^{\mathsf{M}}, \Gamma^{\mathsf{D}}, \Gamma^{\mathsf{S}}$

Figure 7.1: The Type Theories used in this Chapter.

Before we attempt the reduction in the general case, it is useful to first look at how it works in a special case.

7.1 Example: Type Theory Syntax

As a prime example we chose Example 5.0.1 which describes the contexts and types of a type theoretic syntax with a base type and Π-types. To recall the specifics of the example: We want to define a type $Con : \mathcal{U}$ of contexts and a type family $Ty : Con \to \mathcal{U}$ which gives the type of types over a context. These are populated by constructors, providing the empty context, context extension, the base type former and the type former for the Π-types:

$$\begin{aligned}
nil &: Con, \\
ext &: (\Gamma : Con) \to Ty(\Gamma) \to Con, \\
unit &: (\Gamma : Con) \to Ty(\Gamma), \text{ and} \\
pi &: (\Gamma : Con)(A : Ty(\Gamma)) \to Ty(ext(\Gamma, A)) \to Ty(\gamma).
\end{aligned}$$

Since the dependency between the two sorts *Con* and *Ty* can not be represented directly with inductive families, we might, as a first approximation, simply forget about all the indices of the sort – that is, the *Con*-index in *Ty* – and adapt the point constructors accordingly: Let $Con' : \mathcal{U}$ and $Ty' : \mathcal{U}$ be plain types generated by the following four mutually dependent constructors:

$$\begin{aligned}
nil' &: Con', \\
ext' &: Con' \to Ty' \to Con', \\
unit' &: Con' \to Ty', \text{ and} \\
pi' &: Con' \to Ty' \to Ty' \to Ty'.
\end{aligned}$$

But this transformation, which we will call *type erasure* loses important information about the constructed types: In the syntax generated by Con' and Ty', all types exist in the same context. There is no way to tell that the codomain of the Π-types may depend on its domain, and that the Π-type itself exists in the same context as its domain. This justifies that we might call the above types the *presyntax* associated to the syntax given by *Con* and *Ty*, consisting of *precontexts* and *pretypes*.

To counteract this shortcoming, we reintroduce the *typing relation* as a pair of predicates over the presyntax. These inductively defined predicates capture whether an instance of Con' or Ty' is *wellformed* according to the original typing. For the contexts, this is a simple property $w_{Con} : Con' \to \mathcal{U}$, while for types, it needs to state what precontext a pretype is wellformed in: $w_{Ty} : Con' \to Ty' \to \mathcal{U}$. Note that these are inductive families since the definition of all indexing types is concluded at the point of the definition of W_{Con} and W_{Ty}. The point constructors for the wellformedness predicates simply state that is preserved by all constructors of Con' and Ty', in the case of Ty' given the correct index:

$$\begin{aligned}
w_{nil} &: W_{Con}(nil'), \\
w_{ext} &: \{\Gamma : Con'\}\{A : Ty'\} \to W_{Con}(\Gamma) \to W_{Ty}(\Gamma, A) \\
&\qquad \to W_{Con}(ext'(\Gamma, A)), \\
w_{unit} &: \{\Gamma : Con'\} \to W_{Con}(\Gamma) \to W_{Ty}(unit'(\Gamma)), \text{ and} \\
w_{pi} &: \{\Gamma : Con'\}\{A, B : Ty'\} \to W_{Con}(\Gamma) \\
&\qquad \to W_{Ty}(\Gamma, A) \to W_{Ty}(ext'(\Gamma, A), B) \to W_{Ty}(\Gamma, pi'(\Gamma, A, B)).
\end{aligned}$$

Now we can use the predicates to cut out the correct subset of Con' and Ty': A context is a precontext together with a proof of its wellformedness just as a type is a pretype together with a welltypedness witness:

$$\begin{aligned}
Con &:\equiv (\Gamma : Con') \times W_{Con}(\Gamma) \text{ and} \\
Ty(\Gamma) &:\equiv (A : Ty') \times W_{Ty}(\mathsf{pr}_1(\Gamma), A).
\end{aligned}$$

The four point constructors are then easy to define as pairs:

$$\begin{aligned}
nil &:\equiv (nil', w_{nil}), \\
ext(\Gamma, A) &:\equiv \left(ext'(\mathsf{pr}_1(\Gamma), \mathsf{pr}_1(A)), w_{ext}(\mathsf{pr}_2(\Gamma), \mathsf{pr}_2(A))\right), \\
unit(\Gamma) &:\equiv \left(unit'(\mathsf{pr}_1(\Gamma)), w_{unit}(\mathsf{pr}_2(\Gamma))\right), \text{ and} \\
pi(\Gamma, A, B) &:\equiv \left(pi'(\mathsf{pr}_1(\Gamma), \mathsf{pr}_1(A), \mathsf{pr}_1(B)), w_{pi}(\mathsf{pr}_2(\Gamma), \mathsf{pr}_2(A), \mathsf{pr}_2(B))\right).
\end{aligned}$$

This definition clearly has the correct type signature but for it to be the correct replacement for the intended inductive-inductive type, we also need to construct its eliminator: For any given $C : \mathcal{U}$ and $T : C \to \mathcal{U}$ with

$$\begin{aligned}
&n : C, \\
&e : (\gamma : C) \to T(\gamma) \to C, \\
&u : (\gamma : C) \to T(\gamma), \text{ and} \\
&p : (\gamma : C)(a : T(\gamma)) \to T(e(\gamma, a)) \to T(\gamma),
\end{aligned}$$

we need to construct functions $\mathsf{rec}^{Con} : Con \to C$ and $\mathsf{rec}^{Ty} : \{\Gamma : Con\} \to Ty(\Gamma) \to T(\mathsf{rec}^{Con}(\Gamma))$ such that the preservation of the point constructors is manifested in the following β-rules:

$$\begin{aligned}
\mathsf{rec}^{Con}(nil) &= n, \\
\mathsf{rec}^{Con}(ext(\Gamma, A)) &= e(\mathsf{rec}^{Con}(\Gamma), \mathsf{rec}^{Ty}(A)), \\
\mathsf{rec}^{Ty}(unit(\Gamma)) &= u(\mathsf{rec}^{Con}(\Gamma)), \text{ and} \\
\mathsf{rec}^{Ty}(pi(\Gamma, A, B)) &= p(\mathsf{rec}^{Con}(\Gamma), \mathsf{rec}^{Ty}(A), \mathsf{rec}^{Ty}(B)).
\end{aligned}$$

The intricate dependencies between the types make it difficult to define these functions straight away, but it turns out that we will be able to define an *eliminator relation* between the presyntax (Con', Ty') and the motive (C, T) which we can show restricts to the graph of a function on the wellformed parts of the syntax. Just like the wellformedness predicate, this relation is defined inductively as a type family over the presyntax. The signature of this relation is $R_{Con} : Con' \to C \to \mathcal{U}$ for contexts and $R_{Ty} : (\Gamma : Ty')\{\gamma : C\} \to T(\gamma) \to \mathcal{U}$ and the constructors for the relation state that relatedness is preserved by each constructor of the presyntax:

$$\begin{aligned}
r_{nil} &: R_{Con}(nil', n), \\
r_{ext}(\Gamma, A, \gamma, a) &: R_{Con}(\Gamma, \gamma) \to R_{Ty}(A, a) \to R_{Con}(ext'(\Gamma, A), e(\gamma, a)), \\
r_{unit}(\Gamma, \gamma) &: R_{Con}(\Gamma, \gamma) \to R_{Ty}(unit'(\Gamma), u(\gamma)), \text{ and} \\
r_{pi}(\Gamma, A, B, \gamma, a, b) &: R_{Con}(\Gamma, \gamma) \to R_{Ty}(A, a) \to R_{Ty}(B, b) \\
&\qquad \to R_{Ty}(pi'(\Gamma, A, B), p(\gamma, a, b)).
\end{aligned}$$

Since we want a morphism to the model (C, T, n, e, u, p) instead of a relation, we now need to prove that the relation is in fact the graph of a function – i. e. it is right-unique and left-total.

Lemma 7.1.1. *The relation is right-unique on contexts and types. That is, for $\gamma, \gamma' : C$ with $R_{Con}(\Gamma, \gamma)$ and $R_{Con}(\Gamma, \gamma')$, we have $\gamma = \gamma'$, and, regarding types, for $\gamma : C$ and $a, a' : T(\gamma)$, with $R_{Ty}(A, a)$ and $R_{Ty}(A, a')$, we have $a = a'$.*

Proof. Let us first apply induction on the presyntactic variables Γ and A, respectively. This leaves us to consider the cases of the four constructors of Con' and Ty'. For the case of nil', we observe that the only constructor resulting in $R_{Con}(nil', \gamma)$ for some γ is $r_{nil} : R_{Con}(nil, n)$ and we can conclude that both γ and γ' must be equal to n. The reasoning analogously applies to the other cases as well: There is only one relatedness constructor for each of the constructors of the presyntax, so we can always obtain the right-uniqueness for all arguments via the induction hypothesis and, by congruence, infer that the uniqueness carries over to the constructor in consideration. As an example, in the case of δ and δ' with $R_{Con}(ext'(\Gamma, A), \delta)$ and $R_{Con}(ext'(\Gamma, A), \delta')$, we first conclude that $\delta = e(\gamma, a)$ and $\delta' = e(\gamma', a')$ for some γ, γ', a, and a', we see that for these $R_{Con}(\Gamma, \gamma)$, $R_{Con}(\Gamma, \gamma')$, $R_{Ty}(A, a)$, and $R_{Ty}(A, a')$ have to hold and from this we infer that $\gamma = \gamma'$ as well as $a = a'$ and thus $\delta = \delta'$. □

Lemma 7.1.2. *The eliminator relation is left-total on wellformed presyntax: For $\Gamma : Con'$ with $W_{Con}(\Gamma)$ there is $\gamma : C$ such that $R_{Con}(\Gamma, \gamma)$. Analogously, for $A : Ty'(\Gamma)$ with $W_{Ty}(\Gamma, A)$ and $\gamma : C$ with $R_{Con}(\Gamma, \gamma)$ there is $a : T(\gamma)$ such that $R_{Ty}(A, a)$.*

Proof. Again, we first perform induction on the presyntactic argument to the statement – that is, Γ or A. The case of nil' is trivial by providing n and r_{nil}. So let us look at the case of $ext'(\Gamma, A)$. From the induction hypothesis, we get witnesses for the wellformedness of the arguments in the form of $W_{Con}(\Gamma)$ and $W_{Ty}(\Gamma, A)$, as well as related data from the algebra: $\gamma : C$ with $R_{Con}(\Gamma, \gamma)$, and $a : T(\gamma)$ with $R_{Ty}(A, a)$. But this is all the input to use r_{ext} to obtain $R_{Con}(ext'(\Gamma, A), e(\gamma, a))$. The other two cases can be proved analogously. □

The left-totality will suffice to define the recursor functions by simply setting $\mathsf{rec}^{Con}(\Gamma)$ and $\mathsf{rec}^{Ty}(A)$ to be the respective witnesses gained from Lemma 7.1.2. This means that the β-rule for the non-recursive constructor nil will be definitional, while to prove β-rules for the recursive constructors will require the use of Lemma 7.1.1:

To prove, for example, that $\mathsf{rec}^{Con}(ext(\Gamma, A)) = e(\mathsf{rec}^{Con}(\Gamma), \mathsf{rec}^{Ty}(A))$ holds, we observe that both the left-hand side and the right-hand side provide elements in C which by R_{Con} are related to $ext'(\Gamma, A)$, so the lemma give us the desired equality.

7.2 Type Erasure

As seen in the example, the first step to prove the reducability is to formally define the operation which we will call *flattening* or – inspired by the syntax example – *type erasure*. This operation strips away any dependencies between the sorts of a signature as well as all external indices to sorts. The operation should take arbitrary inductive-inductive signatures (contexts) and return signatures for inductive families. Let us look at what type erasure should do with our running examples:

Example 7.2.1 (Natural Numbers)**.** Since the inductive-inductive signature of the *natural numbers* 5.1.4 doesn't contain any indexed sorts, type erasure should "do nothing" with it. That is, returning the sort context and point context of the inductive family syntax which looks like a obvious correspondence to it (cf. Example 6.1.1) while ignoring all entries of the other kind: Let

$$\Gamma_{nat} :\equiv (\cdot, \mathcal{U}, \mathsf{El}(\mathsf{vz}), \Pi\,(\mathsf{vs}(\mathsf{vz}), \mathsf{El}(\mathsf{vs}(\mathsf{vs}(\mathsf{vz}))))).$$

We want to have the following split into sort types and point types:

$$\begin{aligned}&{\Gamma_{nat}}^{\mathsf{E}}_{\mathsf{S}} = (\cdot_{\mathsf{S}}, \mathcal{U}) \text{ and}\\ &{\Gamma_{nat}}^{\mathsf{E}} = (\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})), \mathsf{var}(\mathsf{vz}) \Rightarrow_{\mathsf{P}} \mathsf{El}(\mathsf{var}(\mathsf{vz}))).\end{aligned}$$

Example 7.2.2 (Vectors)**.** In the example of vectors 5.1.5 we need to erase the natural numbers index of the only sort under consideration:

$$\begin{aligned}&{\Gamma_{vec}}^{\mathsf{E}}_{\mathsf{S}} = (\cdot_{\mathsf{S}}, \mathcal{U}) \text{ and}\\ &{\Gamma_{vec}}^{\mathsf{E}} = (\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})), \hat{\Pi}_{\mathsf{P}}(a : A, \hat{\Pi}_{\mathsf{P}}(n : \mathbb{N}, \mathsf{var}(\mathsf{vz}) \Rightarrow_{\mathsf{P}} \mathsf{El}(\mathsf{var}(\mathsf{vz}))))).\end{aligned}$$

Note that the erasure of the vectors does not coincide with the vectors represented as an inductive family (Example 6.1.1), because its sort lacks the indexing over the natural numbers. In fact, it's easy to see that the algebras of this signature would be isomorphic to the type of lists over the type $A \times \mathbb{N}$.

To go from examples to the general case, we will present the different components of the type erasure operation in roughly the same order in which they appear in Section 5.1, most often needing to distinguish between sort and point constructors.

Definition 7.2.3 (Type Erasure)**.** First of all, each context will need to be split into a sort context and a point context:

$$\frac{\vdash \Gamma}{\vdash_{\mathsf{S}} \Gamma^{\mathsf{E}}_{\mathsf{S}}} \qquad \frac{\vdash \Gamma}{\vdash_{\Gamma^{\mathsf{E}}_{\mathsf{S}}} \Gamma^{\mathsf{E}}}$$

To descent down the components of the contexts, we will need to define the operation on types as well. Since we are erasing all information from the sorts, we will only need this for point types, though. Unsurprisingly, we want them to be translated to point types in the appropriate sort context:

$$\frac{\Gamma \vdash A :: \mathsf{P}}{\Gamma^{\mathsf{E}}_{\mathsf{S}} \vdash_{\mathsf{S}} A^{\mathsf{E}} :: \mathsf{P}}$$

Using this we will be able to define the operation creating sort contexts by

$$\begin{aligned} \cdot^{\mathsf{E}}_{\mathsf{S}} &:\equiv \cdot_{\mathsf{S}}, \\ (\Gamma,\ B)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \left(\Gamma^{\mathsf{E}}_{\mathsf{S}}, \mathcal{U}\right) \text{ for } B :: \mathsf{S}, \text{ and} \\ (\Gamma,\ A)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \Gamma^{\mathsf{E}}_{\mathsf{S}} \text{ for } A :: \mathsf{P}. \end{aligned}$$

The generated point context over this sort context has to be extended in the case where the input is an extension by a point type. In the case where it is an extension by a sort type, we want to return the unextended context, but to make up for the definition above, we need to weaken to account for the extension of the resulting sort context:

$$\begin{aligned} \cdot^{\mathsf{E}} &:\equiv \cdot, \\ (\Gamma,\ B)^{\mathsf{E}} &:\equiv \Gamma^{\mathsf{E}}[\mathsf{wk}_{\mathsf{id}}] \text{ for } B :: \mathsf{S}, \text{ and} \\ (\Gamma,\ A)^{\mathsf{E}} &:\equiv \left(\Gamma^{\mathsf{E}},\ A^{\mathsf{E}}\right) \text{ for } A :: \mathsf{P}. \end{aligned}$$

So how do we define A^{E} for a point type A? The fact the we have to recurse on $\mathsf{El}(a)$ makes it clear that we will have to extend our operation to terms of sort types at least. That is, together with A^{E} we also need the following:

$$\frac{\Gamma \vdash t : B :: \mathsf{S}}{\Gamma^{\mathsf{E}}_{\mathsf{S}} \vdash_{\mathsf{S}} t^{\mathsf{E}} : \mathcal{U}}$$

And indeed, with this we can set

$$\mathsf{El}(a)^{\mathsf{E}} :\equiv \mathsf{El}(a^{\mathsf{E}}).$$

For recursive Π-types, we need only care about the ones yielding point types. Note that the operation turns a Π-type into a non-dependent function type!

$$\Pi(a, A)^{\mathsf{E}} :\equiv a^{\mathsf{E}} \Rightarrow_{\mathsf{P}} A^{\mathsf{E}}$$

Since we forgot about the indexing of sort types, erasure of sort-kinded application terms is just erasure of its Π-type term:

$$\mathsf{app}(f)^{\mathsf{E}} :\equiv f^{\mathsf{E}} \text{ for } \Gamma \vdash f : \Pi(a, B) :: \mathsf{S}.$$

External Π-types convert directly into their respective counterparts in the syntax of inductive families. For application of terms of sort-kinded Π-types we need to erase the argument since we erased the Π-type itself.

$$\begin{aligned} \hat{\Pi}(T, A)^{\mathsf{E}} &:\equiv \hat{\Pi}_{\mathsf{P}}(T, \lambda\tau. A(\tau)^{\mathsf{E}}), \text{ and} \\ f(\tau)^{\mathsf{E}} &:\equiv f^{\mathsf{E}} \text{ for } \Gamma \vdash f : \hat{\Pi}(T, B) : \mathsf{S} \end{aligned}$$

Defining the erasure on point types and sort terms pulled back along a substitution, we see that we will also need to erase entire sort substitutions. This is achieved by extending the operation as follows:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta}{\Gamma^{\mathsf{E}}_{\mathsf{S}} \xrightarrow{\sigma^{\mathsf{E}}_{\mathsf{S}}} \Delta^{\mathsf{E}}_{\mathsf{S}}}$$

We will then be able to use this in a straight forward way to define the pullbacks:

$$\begin{aligned} A[\sigma]^{\mathsf{E}} &:\equiv A^{\mathsf{E}}[\sigma^{\mathsf{E}}_{\mathsf{S}}] && \text{for } \Gamma \vdash A :: \mathsf{P} \text{ and} \\ t[\sigma]^{\mathsf{E}} &:\equiv t^{\mathsf{E}}[\sigma^{\mathsf{E}}_{\mathsf{S}}] && \text{for } \Gamma \vdash t : B :: \mathsf{S}. \end{aligned}$$

Erasure of substitutions is built recursively, ignoring point types. Likewise, the first projection will ignore point types:

$$\begin{aligned} \mathsf{id}^{\mathsf{E}}_{\mathsf{S}} &:\equiv \mathsf{id}, \\ (\sigma \circ \delta)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \sigma^{\mathsf{E}}_{\mathsf{S}} \circ \delta^{\mathsf{E}}_{\mathsf{S}}, \\ \epsilon^{\mathsf{E}}_{\mathsf{S}} &:\equiv \epsilon, \\ (\sigma, t)^{\mathsf{E}}_{\mathsf{S}} &:\equiv (\sigma^{\mathsf{E}}_{\mathsf{S}}, t^{\mathsf{E}}) && \text{for } \Gamma \vdash t : B[\sigma] :: \mathsf{S}, \\ (\sigma, t)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \sigma^{\mathsf{E}}_{\mathsf{S}} && \text{for } \Gamma \vdash t : A[\sigma] :: \mathsf{P}, \\ \pi_1(\sigma)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \pi_1(\sigma^{\mathsf{E}}_{\mathsf{S}}) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta, B :: \mathsf{S}), \\ \pi_1(\sigma)^{\mathsf{E}}_{\mathsf{S}} &:\equiv \sigma^{\mathsf{E}}_{\mathsf{S}} && \text{for } \Gamma \xrightarrow{\sigma} (\Delta, A :: \mathsf{P}), \text{ and} \\ \pi_2(\sigma)^{\mathsf{E}} &:\equiv \pi_2(\sigma^{\mathsf{E}}_{\mathsf{S}}). \end{aligned}$$

This concludes the definition of the erasure operation.

For the steps that follow it will be necessary to equip the *algebras* of the resulting signatures with a substitution calculus that also considers point contexts instead of only sort contexts. To this end, we extend the operation of type erasure by assigning a map between the types of algebras of the erasure to each substitution. To be able to build these maps, we furthermore need to find a way how to get an element of the algebra of point type for any given term of this type.

Definition 7.2.4 (Erasure for Point Substitutions)**.** We define the following operation on substitutions and terms of point types:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}}}{\sigma^{\mathsf{E}} : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \to \Delta^{\mathsf{EA}}\left(\sigma^{\mathsf{E}}_{\mathsf{S}}(\gamma_{\mathsf{S}})\right)}$$

$$\frac{\Gamma \vdash t : A :: \mathsf{P} \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}}}{t^{\mathsf{E}} : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \to A^{\mathsf{EA}}(\gamma_{\mathsf{S}})}$$

While in for $\sigma^{\mathsf{E}}_{\mathsf{S}}$ we ignored point constructors, this time we will to the opposite and ignore all sort constructors:

$$\begin{aligned}
\mathsf{id}^{\mathsf{E}}(\gamma) &:\equiv \gamma, \\
\sigma \circ \delta^{\mathsf{E}}(\gamma) &:\equiv \sigma^{\mathsf{E}}\left(\delta^{\mathsf{E}}(\gamma)\right), \\
\epsilon^{\mathsf{E}}(\gamma) &:\equiv \star, \\
(\sigma,\, t)^{\mathsf{E}}(\gamma) &:\equiv \sigma^{\mathsf{E}}(\gamma) && \text{for } \Gamma \vdash t : B[\sigma] :: \mathsf{S}, \\
(\sigma,\, t)^{\mathsf{E}}(\gamma) &:\equiv \left(\sigma^{\mathsf{E}}(\gamma), t^{\mathsf{E}}(\gamma)\right) && \text{for } \Gamma \vdash t : A[\sigma] :: \mathsf{P}, \\
\pi_1(\sigma)^{\mathsf{E}}(\gamma) &:\equiv \sigma^{\mathsf{E}}(\gamma) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, B :: \mathsf{S}), \\
\pi_1(\sigma)^{\mathsf{E}}(\gamma, \alpha) &:\equiv \sigma^{\mathsf{E}}(\gamma) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, A :: \mathsf{P}).
\end{aligned}$$

On point constructors we descend recursively by following the structure of the respective algebra:

$$\begin{aligned}
\mathsf{app}(f)^{\mathsf{E}}(\gamma, \alpha) &:\equiv f^{\mathsf{E}}(\gamma)(\alpha) && \text{for } \Gamma \vdash f : \Pi(a, A :: \mathsf{P}), \\
f(\tau)^{\mathsf{E}}(\gamma) &:\equiv f^{\mathsf{E}}(\gamma)(\tau) && \text{for } \Gamma \vdash f : \hat{\Pi}_{\mathsf{P}}(T,\, A), \\
t[\sigma]^{\mathsf{E}}(\gamma) &:\equiv t^{\mathsf{E}}(\sigma^{\mathsf{E}}(\gamma)) && \text{for } \Gamma \xrightarrow{\sigma} \Delta, \text{ and} \\
\pi_2(\sigma)(\gamma) &:\equiv \mathsf{pr}_2(\sigma^{\mathsf{E}}(\gamma)) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, A :: \mathsf{P}).
\end{aligned}$$

7.3 The Wellformedness Predicate

To remove the ambiguity created by the type erasure we will now have to find a way to select those instances of the types which are "wellformed" in the sense that they lie in the correct fibers of dependent sorts. This predicate will be a proposition dependent on a realization of the erased signature, i. e. on contexts, it will be a function on the type of algebras of the erasure. It is important keep this dependencies and not only to use the initial such algebra, since when we will recursively define this wellformedness predicate, the corresponding piece of signature will not always be initial – in the same way in which a projection of an initial algebra is not necessarily initial anymore.

Example 7.3.1 (Natural Numbers)**.** Taking up the example of Γ_{nat} from 7.2.1, we observe that algebras of ${\Gamma_{nat}}_{\mathsf{S}}^{\mathsf{E}}$ take the form of $(\star, N)$ with $N : \mathcal{U}$ and, given N, those of ${\Gamma_{nat}}^{\mathsf{E}}$ are of the form $(\star, z, s)$ with $z : N$ and $s : N \to N$. Our wellformedness predicate in this case will encode a type family on N, inductively populated by elements "over" z and n. The code for its sort and point constructors looks as follows:

${\Gamma_{nat}}_{\mathsf{S}}^{\mathsf{W}}(\star, z, s) = (\cdot_{\mathsf{S}}, \hat{\Pi}_{\mathsf{S}}(N, \mathcal{U}))$ and
${\Gamma_{nat}}^{\mathsf{W}}(\star, z, s) = (\cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})(z)), \hat{\Pi}_{\mathsf{P}}(n : N, \mathsf{var}(\mathsf{vz})(n) \Rightarrow_{\mathsf{P}} \mathsf{El}(\mathsf{var}(\mathsf{vz})(s(n)))))$

Example 7.3.2 (Vectors)**.** For vectors on a type $A : \mathcal{U}$, the duties of the wellformedness predicate are less trivial: We have to add back the length information which we erased, as described in 7.2.2: Empty vectors should have length zero and appending an element should increase its length by one. This can be achieved by, given the data from an erasure algebra in the form of $V : \mathcal{U}$, $n : V$, and $c : A \to \mathbb{N} \to V \to V$, having a predicate encoded by

$$ {\Gamma_{vec}}_{\mathsf{S}}^{\mathsf{W}}(\star, n, c) = (\cdot_{\mathsf{S}}, \hat{\Pi}_{\mathsf{S}}(n : \mathbb{N}, \hat{\Pi}_{\mathsf{S}}(v : V, \mathcal{U}))), $$

with point constructors that ensure the correct lengh by setting ${\Gamma_{vec}}^{\mathsf{W}}(\star, n, c)$ to be the point context

$$ \begin{gathered} \cdot, \mathsf{El}(\mathsf{var}(\mathsf{vz})(0, n)), \\ \hat{\Pi}_{\mathsf{P}}(a : A, \hat{\Pi}_{\mathsf{P}}(n : \mathbb{N}, \hat{\Pi}_{\mathsf{P}}(v : V, \mathsf{var}(\mathsf{vz})(n, v) \Rightarrow_{\mathsf{P}} \mathsf{El}(\mathsf{var}(\mathsf{vz})(n+1, c(a, n, v)))))). \end{gathered} $$

Like for the type erasure, we will now proceed to generalize this to arbitrary inductive-inductive types.

Definition 7.3.3 (Wellformedness Predicates)**.** Again, we start by considering the resulting type on contexts. Clearly, we want the operation to result in the sort context and the point context of another signature of an inductive family. As we have alredy seen in the previous exapmles, there needs to be a dependency on an erasure algebra which leads to the following rules:

$$\frac{\vdash \Gamma \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}})}{\vdash_{\mathsf{S}} \Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma)}$$

$$\frac{\vdash \Gamma \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}})}{\vdash_{\Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma)} \Gamma^{\mathsf{W}}(\gamma)}$$

To be able to do recursion we will again need to provide a suitable operation on types. We need to distinguish between sort and point types. For sort types, note that we don't have an erasure operation of which we could take an algebra, but since, implicitly, every input sort turns into the inductive-family universe token $\mathcal{U}$, we know that we can act as if its universe is a plain type. Also, we need to know the interpretation of the erasure of the context the type is based on.

$$\frac{\Gamma \vdash B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \alpha : \mathcal{U}}{\vdash_{\mathsf{S}} B^{\mathsf{W}}(\gamma, \alpha) :: \mathsf{S}}$$

$$\frac{\Gamma \vdash A :: \mathsf{P} \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \alpha : A^{\mathsf{EA}}(\gamma_{\mathsf{S}})}{\Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma) \vdash_{\mathsf{S}} A^{\mathsf{W}}(\gamma, \alpha) :: \mathsf{P}}$$

The recursion of the context then looks very much like the one in the definition of type erasure: Extending the sort context whenever we encounter a sort type in the inductive-inductive signature and extending the point case for each point type. Again, we can not leave the point context fixed "on the nose" when encountering a sort type since we need to weaken it to account for the new sort:

$$\begin{aligned}
\cdot_{\mathsf{S}}^{\mathsf{W}}(\gamma) &:\equiv \cdot_{\mathsf{S}} \\
(\Gamma,\, B :: \mathsf{S})_{\mathsf{S}}^{\mathsf{W}}\{\gamma_{\mathsf{S}}, \alpha\}(\gamma) &:\equiv \left(\Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma),\, B^{\mathsf{W}}(\gamma, \alpha)\right) \\
(\Gamma,\, A :: \mathsf{P})_{\mathsf{S}}^{\mathsf{W}}(\gamma, \alpha) &:\equiv \Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma)
\end{aligned}$$

$$\begin{aligned}
\cdot^{\mathsf{W}}(\gamma) &:\equiv \cdot \\
(\Gamma,\, B :: \mathsf{S})^{\mathsf{W}}(\gamma) &:\equiv \Gamma^{\mathsf{W}}(\gamma)[\mathsf{wk}_{\mathsf{id}}] \\
(\Gamma,\, A :: \mathsf{P})^{\mathsf{W}}(\gamma, \alpha) &:\equiv \left(\Gamma^{\mathsf{W}}(\gamma),\, A^{\mathsf{W}}(\gamma, \alpha)\right)
\end{aligned}$$

Like in the definition of type erasure, recursing on $\mathsf{El}(a)$ makes it necessary to extend the definition at least to sort types. So we will also give an operation producing the following data:

$$\frac{\Gamma \vdash t : B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}})}{\Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma) \vdash_{\mathsf{S}} t^{\mathsf{W}}(\gamma) : B^{\mathsf{W}}(\gamma, t^{\mathsf{EA}}(\gamma_{\mathsf{S}}))}$$

Let us now proceed to give the definition on all type formers. Then each sort of the input signature should become a predicate. Since a predicate is the same as a type family with propositional values, we set the wellformedness on the universe to be a type family, the domain of which is given by the set we obtain from the algebra of the erased context. Note that this type family is a non-dependent, *non-recursive* Π-type. The interpretation of $\mathsf{El}(a)$ has to make up for this shift by applying to the wellformedness predicate corresponding the sort term a the element we get from the erasure of $\mathsf{El}(a)$:

$$\begin{aligned}
\mathcal{U}^{\mathsf{W}}(\gamma, \alpha) &:\equiv \hat{\Pi}_{\mathsf{S}}(x : \alpha, \mathcal{U}) \text{ and} \\
\mathsf{El}(a)^{\mathsf{W}}(\gamma, \alpha) &:\equiv \mathsf{El}\left(a^{\mathsf{W}}(\gamma)(\alpha)\right).
\end{aligned}$$

For sort-kinded, recursive Π-types, we again need to remember that in the definition of type erasure, we turned them into instances of $\mathcal{U}$, so to add the information back which we erased, the wellformedness has to turn them into non-recursive Π-types over the erasure of sort term which is the domain of the Π-type we started with. The interpretation of application terms has to follow this step accordingly:

$$\begin{aligned}
\Pi(a, B :: \mathsf{S})^{\mathsf{W}}\{\gamma_{\mathsf{S}}\}(\gamma, \phi) &:\equiv \hat{\Pi}_{\mathsf{S}}(\alpha : a^{\mathsf{EA}}(\gamma_{\mathsf{S}}),\, B^{\mathsf{W}}((\gamma, \alpha), \phi)) \text{ and} \\
\mathsf{app}(f)^{\mathsf{W}}(\gamma, \alpha) &:\equiv f^{\mathsf{W}}(\gamma)(\alpha) \text{ for } \Gamma \vdash f : \Pi(a, B :: \mathsf{S}).
\end{aligned}$$

The treatment of Π-types in point constructors is arguably the trickiest part of the definition. A non-technical description of the effect of the wellformedness operation on these Π-types is the following: For each bit of input data from an algebra of the erasure, wellformedness of this input data should imply wellformedness of the result.

$$\Pi(a, A :: \mathsf{P})^{\mathsf{W}}\{\gamma_{\mathsf{S}}\}(\gamma, \phi) :\equiv \hat{\Pi}_{\mathsf{P}}(\alpha : a^{\mathsf{EA}}(\gamma_{\mathsf{S}}),\, a^{\mathsf{W}}(\gamma)(\alpha) \Rightarrow_{\mathsf{P}} A^{\mathsf{W}}((\gamma, \alpha), \phi(\alpha)))$$

Let us next look at the non-recursive function types. Since we erased them just like the recursive ones, they are processed similar to the definitions above, with the difference that for point constructors, there is no wellformedness of the domain that we have to presuppose to infer wellformedness of the codomain:

$$\begin{aligned}
\hat{\Pi}(T,\, B :: \mathsf{S})^{\mathsf{W}}(\gamma, \phi) &:\equiv \hat{\Pi}_{\mathsf{S}}(\tau : T,\, B(\tau)^{\mathsf{W}}(\gamma, \phi)),\\
\hat{\Pi}(T,\, A :: \mathsf{P})^{\mathsf{W}}(\gamma, \phi) &:\equiv \hat{\Pi}_{\mathsf{P}}(\tau : T,\, A(\tau)^{\mathsf{W}}(\gamma, \phi(\tau))), \text{ and}\\
f(\tau)^{\mathsf{W}}(\gamma) &:\equiv f^{\mathsf{W}}(\gamma)(\tau).
\end{aligned}$$

Again, we need to extend the definition to substitutions to be able to specify it on pulled back types and terms:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}})}{\Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma) \xrightarrow{\sigma_{\mathsf{S}}^{\mathsf{W}}(\gamma)} \Delta_{\mathsf{S}}^{\mathsf{W}}(\gamma)}$$

Their category structure is a direct translation to the sort substitutions of the inductive family syntax. Note that here, we need to refer to 7.2.4 to carry the algebra of the erase point context along the substitution:

$$\begin{aligned}
\mathsf{id}_{\mathsf{S}}^{\mathsf{W}}(\gamma) &:\equiv \mathsf{id} \text{ and}\\
(\sigma \circ \delta)_{\mathsf{S}}^{\mathsf{W}}(\gamma) &:\equiv \sigma_{\mathsf{S}}^{\mathsf{W}}(\delta^{\mathsf{E}}(\gamma)) \circ \delta_{\mathsf{S}}^{\mathsf{W}}(\gamma).
\end{aligned}$$

The pullback operations can afterwards defined by

$$\begin{aligned}
B[\sigma]^{\mathsf{W}}(\gamma, \alpha) &:\equiv B^{\mathsf{W}}(\sigma^{\mathsf{E}}(\gamma), \alpha),\\
A[\sigma]^{\mathsf{W}}(\gamma) &:\equiv A^{\mathsf{W}}(\gamma, \alpha)[\sigma_{\mathsf{S}}^{\mathsf{W}}(\gamma)], \text{ and}\\
t[\sigma]^{\mathsf{W}}(\gamma) &:\equiv t^{\mathsf{W}}(\sigma^{\mathsf{E}}(\gamma))[\sigma_{\mathsf{S}}^{\mathsf{W}}(\gamma)] \text{ for } \Gamma \vdash t : B :: \mathsf{S}.
\end{aligned}$$

The remaining pieces of substitutional calculus are straightforward and look the same as for the type erasure:

$$\begin{aligned}
\epsilon^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \epsilon, \\
(\sigma, t)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \left(\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma), t^{\mathsf{W}}(\gamma)\right) \text{ for } t : B :: \mathsf{S}, \\
(\sigma, t)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma) \\
\pi_1(\sigma)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \pi_1(\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma)) \text{ for } \Gamma \xrightarrow{\sigma} \Delta, B :: \mathsf{S}, \\
\pi_1(\sigma)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma) \text{ for } \Gamma \xrightarrow{\sigma} \Delta, A :: \mathsf{P}, \\
\pi_2(\sigma)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \pi_2(\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma)) \text{ for } \Gamma \xrightarrow{\sigma} \Delta, B :: \mathsf{S}, \\
\pi_2(\sigma)^{\mathsf{W}}_{\mathsf{S}}(\gamma) &:\equiv \sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma) \text{ for } \Gamma \xrightarrow{\sigma} \Delta, A :: \mathsf{P},
\end{aligned}$$

For the next step – using the wellformedness predicate to define the initial object itself – we will need data which provides evidence that the point contexts of the wellformedness predicate behave as well as the sort substitutions. Since we don't have point substitutions as part of the syntax of inductive families, we will proceed like in Definition 7.2.4 and work directly on algebras. To be able to give the definition we will also need a corresponding operation on the terms of point types.

Definition 7.3.4 (Wellformedness for Point Substitutions)**.** We give a wellformedness predicate operation on substitutions and point terms in the following form:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}}}{\sigma^{\mathsf{W}}(\gamma) : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}}) \to \Delta^{\mathsf{W}}(\sigma^{\mathsf{E}}(\gamma))^{\mathsf{A}}\left(\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})\right)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : A :: \mathsf{P} \\ \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})\end{array}}{t^{\mathsf{W}}(\gamma, \delta) : A^{\mathsf{W}}(\gamma, t^{\mathsf{E}}(\gamma))^{\mathsf{A}}(\delta_{\mathsf{S}})}$$

The definition will follow the structure of the algebra, ignoring the occurence of sort terms:

$$\begin{aligned}
\mathsf{id}^{\mathsf{W}}(\gamma,\delta) &:\equiv \delta, \\
\sigma \circ \delta^{\mathsf{W}}(\gamma,\delta') &:\equiv \sigma^{\mathsf{W}}(\delta^{\mathsf{E}}(\gamma),\delta^{\mathsf{W}}(\gamma,\delta')), \\
\epsilon^{\mathsf{W}}(\gamma,\delta) &:\equiv \star, \\
(\sigma,\, t)^{\mathsf{W}}(\gamma,\delta) &:\equiv \sigma^{\mathsf{W}}(\gamma,\delta) && \text{for } \Gamma \vdash t : B[\sigma] :: \mathsf{S}, \\
(\sigma,\, t)^{\mathsf{W}}(\gamma,\delta) &:\equiv \big(\sigma^{\mathsf{W}}(\gamma,\delta), t^{\mathsf{W}}(\delta)\big) && \text{for } \Gamma \vdash t : A[\sigma] :: \mathsf{P}, \\
\pi_1(\sigma)^{\mathsf{W}}(\gamma,\delta) &:\equiv \sigma^{\mathsf{W}}(\gamma,\delta) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, B :: \mathsf{S}), \text{ and} \\
\pi_1(\sigma)^{\mathsf{W}}(\gamma,\delta) &:\equiv \mathsf{pr}_1(\sigma^{\mathsf{W}}(\gamma,\delta)) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, A :: \mathsf{P}), \\
\mathsf{app}(f)^{\mathsf{W}}(\gamma,\alpha)(\delta,\omega) &:\equiv f^{\mathsf{W}}(\delta)(\alpha)(\omega) && \text{for } \Gamma \vdash f : \Pi(a, A :: \mathsf{P}), \\
f(\tau)^{\mathsf{W}}(\gamma,\delta) &:\equiv f^{\mathsf{W}}(\gamma,\delta)(\tau) && \text{for } \Gamma \vdash f : \hat{\Pi}_{\mathsf{P}}(T,\, A), \\
f[\sigma]^{\mathsf{W}}(\gamma,\delta) &:\equiv f^{\mathsf{W}}(\sigma^{\mathsf{E}}(\gamma),\sigma^{\mathsf{W}}(\gamma,\delta)) && \text{for } \Gamma \xrightarrow{\sigma} \Delta, \text{ and} \\
\pi_2(\sigma)^{\mathsf{W}}(\gamma,\delta) &:\equiv \mathsf{pr}_2(\sigma^{\mathsf{W}}(\gamma,\delta)) && \text{for } \Gamma \xrightarrow{\sigma} (\Delta,\, A :: \mathsf{P}).
\end{aligned}$$

7.4 The Initial Object

Since we now have a way to “carve out” the wellformed elements from the types we created via type erasure, we can now define our desired inductive-inductive types itself. In this section, this will amount to defining just one specific algebra over the given inductive-inductive signature. This corresponds to giving sorts with the correct *point constructors*. What distinguishes this algebra from others is that we have strong reasons to believe that, besides constructors, it also admits a dependent *eliminator*, or, equivalently, that it is initial among all algebras.

The construction of the initial object obviously presupposes the existence of initial algebras of inductive families. Nevertheless, we need to apply the same strategy as in the definition of the wellformedness predicate: The construction will depend on arbitrary algebras of type erasure and wellformedness instead of just depending on the initial one. This allows us to descend recursively and still refer to the correct algebra of the respective inductive families.

Like in the last two steps of the construction, let us again start off by taking a look at our set of running examples:

Example 7.4.1 (Natural Numbers)**.** Continuing from Example 7.3.1, we again assume sort and point algebras $(\star, N')$ and $(\star, z', s')$ of the erasure of natural numbers $\Gamma_{nat}{}^{\mathsf{E}}_{\mathsf{S}}$ and $\Gamma_{nat}{}^{\mathsf{E}}$. Given this data, the algebras of the well-formedness predicate take the form of $(\star, W_N)$ and $(\star, w_z, w_s)$ with types

$$\begin{aligned}
W_N &: N' \to \mathcal{U}, \\
w_z &: W(z'), \text{ and} \\
w_s &: (n' : N') \to W_N(n') \to W_N(s'(n')).
\end{aligned}$$

Then, we want the inductive-inductive algebra $\mathsf{con}(\Gamma_{nat}) : \Gamma_{nat}{}^{\mathsf{A}}$ to consist of the subsets of erased types which (in this case trivially) fulfil the well-formedness condition, with the point constructors lifted to these subsets: $\mathsf{con}(\Gamma_{nat}) = (\star, N, z, s)$ with

$$\begin{aligned}
N &= (n' : N') \times W_N(n'), \\
z &= (z', w_z), \text{ and} \\
s &= \lambda((n', w_n) : N).\,(s'(n'), w_s(n', w_n)).
\end{aligned}$$

Example 7.4.2 (Vectors)**.** Let us next consider the type of vectors on a type $A : \mathcal{U}$. The assumed algebras of the type erasure give us V', n', and c' as in 7.3.2. With those as input, algebras of the sort and point part of the wellformedness predicate $\Gamma_{vec}{}^{\mathsf{W}}_{\mathsf{S}}$ and $\Gamma_{vec}{}^{\mathsf{W}}$ look like $(\star, W_V)$ and $(\star, w_n, w_c)$ with

$$\begin{aligned}
W_V &: \mathbb{N} \to V' \to \mathcal{U}, \\
w_n &: W_V(0, n'), \text{ and} \\
w_c &: (a : A)(m : \mathbb{N})(v' : V) \to W_V(m, v') \to W_V(m+1, c'(a, m, v')).
\end{aligned}$$

This suggests that we will have an algebra $\mathsf{con}(\Gamma_{vec}) : \Gamma^{\mathsf{A}}_{vec}$ defined by $\mathsf{con}(\Gamma_{vec}) = (\star, V, n, c)$ with

$$\begin{aligned}
V &= \lambda(n : \mathbb{N}).\,(v' : V') \times W_V(n, v'), \\
n &= (n', w_n), \text{ and} \\
c &= \lambda(a : A)(m : \mathbb{N})((v', w_{v'}) : V).\,(c'(a, m, v'), w_c(a, m, v', w_{v'})).
\end{aligned}$$

Let us now consider the case of an arbitrary signature $\vdash \Gamma$. The form which our operation will take is clear – for each signature we need to return an algebra of that signature:

$$\frac{\vdash \Gamma}{\mathsf{con}(\Gamma) : \Gamma^{\mathsf{A}}}$$

But as we saw before, recursion is easier when we make the dependent on arbitrary algebras of the previous steps – that is, arbitrary algebras over type erasure and the wellformedness predicate. After we succeed in defining this more general construction Γ^{Σ}, we will eliminate this dependency by fixing these algebras to be the initial ones which we assume to exist in this chapter.

Definition 7.4.3 (Sigma Construction)**.** As mentioned, the more general construction will depend on both the type erasure and the wellformedness, so that the operation will take the following form:

$$\frac{\vdash \Gamma \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})}{\Gamma^{\Sigma}(\gamma, \delta) : \Gamma^{\mathsf{A}}}$$

To recurse on the contexts, we again need to extend the operations to types, distinguishing between sort and point constructors, resulting in the following two rules:

$$\frac{\begin{array}{c}\Gamma \vdash B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \\ \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}}) \qquad \alpha : \mathcal{U} \qquad \omega : B^{\mathsf{W}}(\gamma, \alpha)^{\mathsf{A}}\end{array}}{B^{\Sigma}(\gamma, \delta, \omega) : B^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma, \delta)\right)}$$

$$\frac{\begin{array}{c}\Gamma \vdash A :: \mathsf{P} \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \\ \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}}) \qquad \alpha : A^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \omega : A^{\mathsf{W}}(\gamma, \alpha)^{\mathsf{A}}(\delta_{\mathsf{S}})\end{array}}{A^{\Sigma}(\gamma, \delta, \omega) : A^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma, \delta)\right)}$$

These operations allow us two define the sigma construction as straightforward as we have seen in the previous constructions:

$$\begin{aligned} \cdot^{\Sigma}(\gamma, \delta) &:\equiv \star \\ (\Gamma,\ B :: \mathsf{S})^{\Sigma}(\gamma)\{\delta_{\mathsf{S}}, \omega\}(\delta) &:\equiv \left(\Gamma^{\Sigma}(\gamma, \delta), B^{\Sigma}(\gamma, \delta, \omega)\right) \\ (\Gamma,\ A :: \mathsf{P})^{\Sigma}((\gamma, \alpha), (\delta, \omega)) &:\equiv \left(\Gamma^{\Sigma}(\gamma, \delta), A^{\Sigma}(\gamma, \delta, \omega)\right) \end{aligned}$$

Again, the necessity to define $\mathsf{El}(a)^{\Sigma}$ forces us to extend the definition on terms as well. The treatment for sort and point terms differs because type erasure and wellformedness predicate are defined as maps between algebras of inductive family syntax instead of syntax itself: We do not need to give data, but instead we have to make sure that, when given a term, the two ways of getting an element of the interpretation of its type – via the Σ-construction on contexts and via the Σ-construction on types – coincide:

$$\frac{\Gamma \vdash t : B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})}{t^{\Sigma}(\gamma,\delta) : t^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta)\right) = B^{\Sigma}\left(\gamma,\delta,t^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})\right)}$$

$$\frac{\Gamma \vdash t : A :: \mathsf{P} \qquad \gamma_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{EA}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma_{\mathsf{S}}^{\mathsf{W}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})}{t^{\Sigma}(\gamma,\delta) : t^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta)\right) = A^{\Sigma}\left(\gamma,\delta,t^{\mathsf{W}}(\delta)\right)}$$

We will now go through all the type formers in order, starting with the universe. It justifies the name of the construction, producing a sigma type of the erasure and its wellformedness. For the element operator, we need the above equation for terms to be able to populate these sigma types accordingly:

$$\begin{aligned} \mathcal{U}^{\Sigma}(\gamma,\delta,\omega) &:\equiv (x : \alpha) \times \omega(x) \text{ and} \\ \mathsf{El}(a)^{\Sigma}(\gamma,\delta,\omega) &:\equiv \left(a^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}(\alpha,\omega) \end{aligned}$$

Let us next look at the recursive Π-types and their application: Let (α,ω) be the result of $\left(a^{\Sigma}(\gamma,\delta)\right)^{*}(\xi)$, then we can set

$$\Pi(a, B :: \mathsf{S})^{\Sigma}(\gamma,\delta,\phi)(\xi) :\equiv p^{*}\left(B^{\Sigma}((\gamma,\alpha),(\delta,\omega),\phi(\alpha))\right) \text{ and}$$
$$\Pi(a, A :: \mathsf{P})^{\Sigma}(\gamma,\delta,\phi)(\xi) :\equiv p^{*}\left(B^{\Sigma}((\gamma,\alpha),(\delta,\omega),\phi(\alpha,\omega))\right),$$

where p is a proof for

$$B^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta),\left(a^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}(\alpha,\omega)\right) = B^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta),\xi\right).$$

For the application we provide $\mathsf{app}(f)^{\Sigma}((\gamma,\alpha),(\delta,\omega))$ for $\Gamma \vdash f : \Pi(a,B)$ by the following identity proofs:

$$\begin{aligned} &\mathsf{app}(f)^{\mathsf{A}}\left((\Gamma,\,\mathsf{El}(a))^{\Sigma}((\gamma,\alpha),(\delta,\omega))\right) \\ &\equiv f^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma),\left(a^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}(\alpha,\omega)\right) \\ &= \Pi(a,B)^{\Sigma}\left(\gamma,\delta,\left(a^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}(\alpha,\omega)\right) \text{ by } f^{\Sigma} \\ &\equiv \begin{cases} B^{\Sigma}\left((\gamma,\alpha),(\delta,\omega),f^{\mathsf{EA}}(\gamma_{\mathsf{S}})\left(f^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})\right)\right) & \text{for } B :: \mathsf{S} \text{ and} \\ B^{\Sigma}\left((\gamma,\alpha),(\delta,\omega),f^{\mathsf{E}}(\gamma)\left(f^{\mathsf{W}}(\delta)\right)\right) & \text{for } B :: \mathsf{P}. \end{cases} \end{aligned}$$

The case for non-recursive Π-types is a rather straightforward descent, compared to the recursive ones:

$$\begin{aligned} \Pi(a,B)^{\Sigma}(\gamma,\delta,\phi,\tau) &:\equiv B^{\Sigma}(\gamma,\delta,\phi(\tau)) \text{ and} \\ f(\tau)^{\Sigma}(\gamma,\delta) &:\equiv \mathsf{happly}\left(f^{\Sigma}(\gamma,\delta),\tau\right) \end{aligned}$$

This concludes all type formers, though we still need to have a definition on types which are the result of pullback along a substitution, and thus need to extend the operation to substitution with the following rule, which introduces equalities similar to the ones that we already saw for terms:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}}{\sigma^{\Sigma}(\gamma,\delta) : \sigma^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta)\right) = \Delta^{\Sigma}\left(\sigma^{\mathsf{E}}(\gamma),\sigma^{\mathsf{W}}(\delta)\right)}$$

With this rule we can provide the correct operations on pulled back types and terms:

$$B[\sigma]^{\Sigma}(\gamma,\delta,\omega) :\equiv \left(\sigma^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}\left(B^{\Sigma}(\sigma^{\mathsf{E}}(\gamma),\sigma^{\mathsf{W}}(\delta),\omega)\right)$$

for $\Gamma \vdash B :: \mathsf{S}$, and for a term $\Gamma \vdash t[\sigma] : B[\sigma]$, we use σ^{Σ} in the proof of the equality $t[\sigma]^{\Sigma}$:

$$\begin{aligned} & t[\sigma]^{\mathsf{A}}(\Gamma^{\Sigma}(\gamma,\delta)) \\ & \equiv t^{\mathsf{A}}(\sigma^{\mathsf{A}}(\Gamma^{\Sigma}(\gamma,\delta))) \\ & = \left(\sigma^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}\left(t^{\mathsf{A}}(\Delta^{\Sigma}(\sigma^{\mathsf{E}}(\gamma),\sigma^{\mathsf{W}}(\delta)))\right) \\ & = \left(\sigma^{\Sigma}(\gamma,\delta)^{-1}\right)^{*}\left(B^{\Sigma}(\sigma^{\mathsf{E}}(\gamma),\sigma^{\mathsf{W}}(\gamma,\delta),t^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}})))\right) \\ & \equiv B[\sigma]^{\Sigma}(\gamma,\delta,t^{\mathsf{W}}[\sigma^{\mathsf{W}}_{\mathsf{S}}(\gamma)]^{\mathsf{A}}(\delta_{\mathsf{S}})) \end{aligned}$$

and similarly for $\Gamma \vdash A :: \mathsf{P}$.

The substitutional calculus for this rule is easily defined since $\mathsf{id}^{\Sigma}(\gamma,\delta)$ and $\epsilon^{\Sigma}(\gamma,\delta)$ hold definitionally, the composition rule $\sigma \circ \delta^{\Sigma}(\gamma,\delta')$ for $\Delta \xrightarrow{\sigma} \Sigma$ and $\Gamma \xrightarrow{\delta} \Delta$ is justified by

$$\begin{aligned} \sigma^{\mathsf{A}}\left(\delta^{\mathsf{A}}\left(\Gamma^{\Sigma}(\gamma,\delta')\right)\right) &= \sigma^{\mathsf{A}}\left(\Delta^{\Sigma}(\delta^{\mathsf{E}}(\gamma),\delta^{\mathsf{W}}(\gamma,\delta'))\right) && \text{by } \delta^{\Sigma}(\gamma,\delta') \\ &= \Sigma^{\Sigma}(\sigma^{\mathsf{E}}(\delta^{\mathsf{E}}(\gamma)),\sigma^{\mathsf{W}}(\delta^{\mathsf{E}}(\gamma),\delta^{\mathsf{W}}(\gamma,\delta'))) && \text{by } \sigma^{\Sigma}(\ldots). \end{aligned}$$

To prove the coherence of a substitution extended by a term (σ, t) with $\Gamma \xrightarrow{\sigma} \Delta$ and $\Gamma \vdash B[\sigma] :: \mathsf{S}$, we use $\sigma^{\Sigma}(\gamma, \delta)$ and $t^{\Sigma}(\gamma, \delta)$ simultaneously to obtain the required equation (the variant for point terms follows in similar fashion):

$$\begin{aligned}
&(\sigma, t)^{\mathsf{A}}(\Gamma^{\Sigma}(\gamma, \delta)) \\
&\equiv \Big(\sigma^{\mathsf{A}}(\Gamma^{\Sigma}(\gamma, \delta)), t^{\mathsf{A}}(\Gamma^{\Sigma}(\gamma, \delta))\Big) \\
&= \Big(\Delta^{\Sigma}(\sigma^{\mathsf{E}}(\gamma), \sigma^{\mathsf{W}}(\gamma, \delta)), B^{\Sigma}(\sigma^{\mathsf{E}}(\gamma), \sigma^{\mathsf{W}}(\gamma, \delta), t^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}}))\Big) \\
&\equiv (\Gamma, B)^{\Sigma}(\sigma^{\mathsf{E}}(\gamma), \sigma^{\mathsf{W}}(\gamma, \delta)).
\end{aligned}$$

The remaining equations follow in similarly obvious way.

Definition 7.4.4 (Initial Object)**.** Using the generalized sigma construction we are now able to define the initial object by plugging in the respective initial objects of the inductive families:

$$\mathsf{con}(\Gamma) :\equiv \Gamma^{\Sigma}(\mathsf{con}(\Gamma^{\mathsf{E}}), \mathsf{con}(\Gamma^{\mathsf{W}}(\mathsf{con}(\Gamma^{\mathsf{E}})))) : \Gamma^{\mathsf{A}}$$

7.5 The Eliminator Relation

Now that we have defined a candidate for the initial object in the category of algebras, the obvious next step is to prove its initiality. The strategy for this is, as we have seen in Chapter 7.1, to first define a *relation* for the non-dependent eliminator, before showing that this relation is right-unique and left-total and as such, a function. Like the wellformedness predicate, the construction of this relation relies on the fact that we are provided with an initial algebra for the type erasure.

Again, we start by taking on the running examples of natural numbers, vectors, and type theoretic syntax, to get a feeling for what the construction is supposed to look like.

Example 7.5.1 (Natural Numbers)**.** Continuing the construction from Example 7.2.1 and parallel to Example 7.3.1, we assume sort and point algebras $(\star, N') : {\Gamma_{nat}}^{\mathsf{A}}$ and $(\star, z', s') : {\Gamma_{nat}}^{\mathsf{A}}(\star, N')$. Furthermore, our relation should relate these algebras to an arbitray algebra of Γ_{nat}, so we assume that we are given

$$\begin{aligned}(\star, N, z, s) : {\Gamma_{nat}}^{\mathsf{A}} &\text{ with}\\ N &: \mathcal{U},\\ z &: N, \text{ and}\\ s &: N \to N.\end{aligned}$$

Since, as we observed in Example 7.2.1, no erasure takes place, this algebra already contains the same amount of information as $((\star, N'), (\star, z', s'))$. Like type erasure and wellformedness predicate, the eliminator relation will come in the form of the sort and point context of an inductive family. The sort context simply describes the one of a type family indexed both over N and N' (we will use variable names instead of de-Bruijn indices):

$${\Gamma_{nat}}^{\mathsf{R}}_{\mathsf{S}} = (\cdot_{\mathsf{S}},\ R_N : \hat{\Pi}_{\mathsf{S}}(N,\ \hat{\Pi}_{\mathsf{S}}(N', \mathcal{U}))).$$

The point context will now populate this relation by witnesses for the fact that the corresponding point constructors are related, or, for s, that they preserve relatedness:

$$\begin{aligned}{\Gamma_{nat}}^{\mathsf{R}} = \Big(&\cdot,\ r_z : \mathsf{El}(R_N(z)(z'),\\ &\hat{\Pi}_{\mathsf{P}}(n : N,\ \hat{\Pi}_{\mathsf{P}}(n' : N',\ R_N(n)(n') \Rightarrow_{\mathsf{P}} \mathsf{El}(R_N(s(n))(s'(n')))))\Big).\end{aligned}$$

Since, as we remarked earlier, the type erasure on Γ_{nat} is without effect, we can expect this relation to be the same as the graph of the non-dependent eliminator on N' already.

Example 7.5.2 (Vectors)**.** To move on to an example where on the one hand no real induction-induction is happening, but we still have a non-trivial effect by type erasure, let us consider the type of vectors over an external type $A : \mathcal{U}$. Again, we assume that we already have constructed point and sort algebras $(\star, V')$ and $(\star, n', c')$ as in Example 7.3.2. We also assume an arbitrary algebra of the vectors in Γ_{vec}, which comes in the form of $(\star, V, n, c)$ where

$$\begin{aligned}&V : \mathbb{N} \to \mathcal{U},\\ &n : V(0), \text{ and}\\ &c : (a : A)(n : \mathbb{N}) \to V(n) \to V(n+1).\end{aligned}$$

Since V is a type family, the relation will also need to be indexed by its codomain and so we obtain the following for the sort context describing the relation:

$$\Gamma_{vec}{}^{\mathsf{R}}(\ldots) = \Big(\cdot_{\mathsf{S}},\ R_V : \hat{\Pi}_{\mathsf{S}}(n : \mathbb{N},\ \hat{\Pi}_{\mathsf{S}}(V(n),\ \hat{\Pi}_{\mathsf{S}}(V', \mathcal{U})))\Big).$$

The point context again describes that relatedness is preserved by all point constructors:

$$\Gamma_{vec}{}^{\mathsf{R}} = \Big(\cdot, \mathsf{El}(R_V(0)(n)(n')),$$
$$\hat{\Pi}_{\mathsf{P}}(a : A, n : \mathbb{N}, v : V, v' : V', R_V(n)(v)(v') \Rightarrow_{\mathsf{P}} R_V(n+1)(c(a,v))(c'(a,v')))\Big).$$

Definition 7.5.3 (Eliminator Relation)**.** As we have noticed before, the specification of the eliminator relation bears many similarities with the one of the wellformedness predicate, with the difference that we now also depend on an arbitrary algebra over the inductive-inductive signature under consideration. The definition on contexts again produces both a sort and a point context:

$$\frac{\vdash \Gamma \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}}{\vdash_{\mathsf{S}} \Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})}$$

$$\frac{\vdash \Gamma \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}}{\vdash_{\Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma,\gamma^{\mathsf{A}})} \Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})}$$

For the definition on types, we can make a a big simplification on sort types: The definition does not depend on the erasure of the remaining context, but only on the erasure of the type itself. For point types, we are not able to make this simplification.

$$\frac{\Gamma \vdash B :: \mathsf{S} \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}} \qquad \alpha : \mathcal{U} \qquad \alpha^{\mathsf{A}} : B^{\mathsf{A}}(\gamma^{\mathsf{A}})}{B^{\mathsf{R}}(\alpha, \alpha^{\mathsf{A}}) :: \mathsf{S}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A :: \mathsf{P} \\ \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}} \qquad \alpha : A^{\mathsf{E}}(\gamma_{\mathsf{S}}) \qquad \alpha^{\mathsf{A}} : A^{\mathsf{A}}(\gamma^{\mathsf{A}})\end{array}}{\Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) \vdash_{\mathsf{S}} A^{\mathsf{R}}(\gamma, \alpha, \alpha^{\mathsf{A}})}$$

As usual, we define the operation on contexts by a simple recursion, using the definition on types, ignoring extensions by types of the respective other kind:

$$\begin{aligned}
\cdot^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \cdot_{\mathsf{S}} \\
(\Gamma,\, B :: \mathsf{S})^{\mathsf{R}}_{\mathsf{S}}\{\gamma_{\mathsf{S}}, \alpha\}(\gamma, (\gamma^{\mathsf{A}}, \alpha^{\mathsf{A}})) &:\equiv \left(\Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}),\, B^{\mathsf{R}}(\alpha, \alpha^{\mathsf{A}})\right) \\
(\Gamma,\, A :: \mathsf{P})^{\mathsf{R}}_{\mathsf{S}}((\gamma, \alpha), (\gamma^{\mathsf{A}}, \alpha^{\mathsf{A}})) &:\equiv \Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})
\end{aligned}$$

$$\begin{aligned}
\cdot^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \cdot \\
(\Gamma,\, B :: \mathsf{S})^{\mathsf{R}}\{\gamma_{\mathsf{S}}, \alpha\}(\gamma, (\gamma^{\mathsf{A}}, \alpha^{\mathsf{A}})) &:\equiv \Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})[\mathsf{wk}_{\mathsf{id}}] \\
(\Gamma,\, A :: \mathsf{P})^{\mathsf{R}}((\gamma, \alpha), (\gamma^{\mathsf{A}}, \alpha^{\mathsf{A}})) &:\equiv \left(\Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}}),\, A^{\mathsf{R}}(\gamma, \alpha, \alpha^{\mathsf{A}})\right)
\end{aligned}$$

Before giving the definition on type formers, we need again care about how to handle at least sort terms. For those, we introduce a construction of the following type:

$$\frac{\Gamma \vdash t : B :: \mathsf{S} \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}}{\Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) \vdash_{\mathsf{S}} t^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}}) : B^{\mathsf{R}}(t^{\mathsf{EA}}(\gamma_{\mathsf{S}}), t^{\mathsf{A}}(\gamma^{\mathsf{A}}))}$$

Using this we can give the definition on the universe – producing the relation itself – and the element operator, populating the relation via the following definitions:

$$\begin{aligned}
\mathcal{U}^{\mathsf{R}}(\alpha, \alpha^{\mathsf{A}}) &:\equiv \hat{\Pi}_{\mathsf{S}}(\alpha^{\mathsf{A}}, \hat{\Pi}_{\mathsf{S}}(\alpha, \mathcal{U})) \\
\mathsf{El}(a)^{\mathsf{R}}(\gamma, \alpha, \alpha^{\mathsf{A}}) &:\equiv \mathsf{El}(a^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})(\alpha^{\mathsf{A}})(\alpha))
\end{aligned}$$

To continue to recursive Π-types, we observe that we need sort-kinded Π-types to be translated to Π-types which the generated relation is indexed over, including the appropriate application, while point-kinded Π-types are translated to the fact that relatedness is preserved for all point constructors.

$$\begin{aligned}
\Pi(a, B :: \mathsf{S})^{\mathsf{R}}(\phi, \phi^{\mathsf{A}}) &:\equiv \hat{\Pi}_{\mathsf{S}}(\alpha^{\mathsf{A}} : a^{\mathsf{A}}(\gamma^{\mathsf{A}}),\, B^{\mathsf{R}}(\phi, \phi^{\mathsf{A}}(\alpha^{\mathsf{A}}))) \\
\Pi(a, A :: \mathsf{P})^{\mathsf{R}}(\gamma, \phi, \phi^{\mathsf{A}}) &:\equiv \hat{\Pi}_{\mathsf{P}}\Big(\alpha^{\mathsf{A}} : a^{\mathsf{A}}(\gamma^{\mathsf{A}}), \alpha : a^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \\
&\qquad a^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})(\alpha^{\mathsf{A}})(\alpha) \Rightarrow_{\mathsf{P}} B^{\mathsf{R}}((\gamma, \alpha), \phi(\alpha), \phi^{\mathsf{A}}(\alpha^{\mathsf{A}}))\Big)
\end{aligned}$$

$$\mathsf{app}(f)^{\mathsf{R}}((\gamma, \alpha), (\gamma^{\mathsf{A}}, \alpha^{\mathsf{A}})) :\equiv f^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})(\alpha^{\mathsf{A}})$$

The treatment of external Π-types is a simple one-to-one translation using the respective external Π-types in the generated inductive-family:

$$\begin{aligned}
\hat{\Pi}(T,\ B :: \mathsf{S})^{\mathsf{R}}(\phi, \phi^{\mathsf{A}}) &:\equiv \hat{\Pi}_{\mathsf{S}}(\tau : T,\ B(\tau)^{\mathsf{R}}(\phi, \phi^{\mathsf{A}}(\tau)))\\
\hat{\Pi}(T,\ A :: \mathsf{P})^{\mathsf{R}}(\phi, \phi^{\mathsf{A}}) &:\equiv \hat{\Pi}_{\mathsf{P}}(\tau : T,\ A(\tau)^{\mathsf{R}}(\phi(\tau), \phi^{\mathsf{A}}(\tau)))\\
f(\tau)^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv f^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})(\tau)
\end{aligned}$$

To provide the relation for types pulled along substitutions, we again have to provide data for substitutions, in the form of sort substitutions:

$$\frac{\Gamma \xrightarrow{\sigma} \Delta \qquad \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}}{\Gamma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) \xrightarrow{\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})} \Delta^{\mathsf{R}}_{\mathsf{S}}(\sigma^{\mathsf{E}}(\gamma), \sigma^{\mathsf{A}}(\gamma^{\mathsf{A}}))}$$

As the sort substitutions of inductive families, these follow the categorical structure trivially:

$$\begin{aligned}
\mathsf{id}^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \mathsf{id} \text{ and}\\
\sigma \circ \delta^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \sigma^{\mathsf{R}}_{\mathsf{S}}(\delta^{\mathsf{E}}(\gamma), \delta^{\mathsf{A}}(\gamma^{\mathsf{A}})) \circ \delta^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}).
\end{aligned}$$

Pulled back sort types are invariant under pullback while point types and sort terms are pulled back with the operation we gave in Definition 6.1.2:

$$\begin{aligned}
B[\sigma]^{\mathsf{R}}(\alpha, \alpha^{\mathsf{A}}) &:\equiv B^{\mathsf{R}}(\alpha, \alpha^{\mathsf{A}}) \text{ for } \Gamma \vdash B :: \mathsf{S},\\
A[\sigma]^{\mathsf{R}}(\gamma, \alpha, \alpha^{\mathsf{A}}) &:\equiv A^{\mathsf{R}}(\sigma^{\mathsf{E}}(\gamma), \alpha, \alpha^{\mathsf{A}})[\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})] \text{ for } \Gamma \vdash A :: \mathsf{P}, \text{ and}\\
t[\sigma]^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv t^{\mathsf{R}}(\sigma^{\mathsf{E}}(\gamma), \sigma^{\mathsf{A}}(\gamma^{\mathsf{A}}))[\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})].
\end{aligned}$$

The remaining parts of the substitutional calculus are given by their counterparts in the sort substitutions of inductive families as they were defined in Definition 6.1.2:

$$\begin{aligned}
\epsilon^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \epsilon,\\
(\sigma,\ t)^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv (\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}), t^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})) \text{ for } \Gamma \vdash t : B :: \mathsf{S},\\
(\sigma,\ t)^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) \text{ for } \Gamma \vdash t : A :: \mathsf{P},\\
\pi_1(\sigma)^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \pi_1(\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})) \text{ for } \Gamma \xrightarrow{\sigma} (\Delta,\ B :: \mathsf{S}),\\
\pi_1(\sigma)^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) \text{ for } \Gamma \xrightarrow{\sigma} (\Delta,\ A :: \mathsf{P}),\\
\pi_2(\sigma)^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}}) &:\equiv \pi_2(\sigma^{\mathsf{R}}_{\mathsf{S}}(\gamma, \gamma^{\mathsf{A}})) \text{ for } \Gamma \xrightarrow{\sigma} (\Delta,\ B :: \mathsf{S})\ .
\end{aligned}$$

This concludes our construction of the eliminator relation.

7.6 Formalization in Agda

Most parts of this chapter have been formalized in the theorem prover Agda. In the following, we will describe this formalization including the problems we faced. The formalizations can be found online at `https://github.com/javra/indind-agda`.

The syntax of inductive families has been straightforward to internalize in Agda, using indexed inductive types. One trick we had to apply was to define variables and terms as separate types, with an inclusion map from variables to types (cf. Remark 6.4.5).

Algebras of inductive families including their morphisms, and furthermore their variant of displayed algebras and sections have been formalized by recursion on the syntax. We want the algebras to be as strict as possible, so we do not refrain from turning certain rules from the substitution calculus into rewrite rules, like in the following example of the mapping an algebra over a sort context along the identity substitution is without effect:

```
idᵃ : ∀{ℓ Γc} → (γc : _ᵃc {ℓ} Γc) → (id ᵃs) γc ≡ γc
idᵃ {ℓ}{•c}      γc       = refl
idᵃ {ℓ}{Γc ▶c x} (γc , α) = ,≡ (idᵃ γc) refl
{-# REWRITE idᵃ #-}
```

The syntax of inductive-inductive types is, as mentioned in Remark 5.1.3, only representable directly as a quotient inductive-inductive type. This is why we only postulate it like in the following excerpt:

```
postulate
  Con : Set
  Ty  : Con → PS → Set
  Tm  : ∀ Γ → ∀ {k} → Ty Γ k → Set
  Sub : Con → Con → Set

  •     : Con
  _▶_   : ∀{k}(Γ : Con) → Ty Γ k → Con
...
  ass   : ∀{Γ Δ Σ Ω}{σ : Sub Σ Ω}{δ : Sub Δ Σ}{ν : Sub Γ Δ}
        → (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)
```

Algebras and morphisms of inductive-inductive types are fomalized in record which follow the definition of the syntax:

```
record Con : Set₂ where
  field
    ᴬ    : Set₁
    ᴹ    : ᴬ → ᴬ → Set₁
...
_▸S_ : (Γ : Con) → TyS Γ → Con
Γ ▸S B = record {
  ᴬ    = Σ Γ.ᴬ B.ᴬ ;
  ᴹ    = λ { (γᴬ , αᴬ) (δᴬ , βᴬ) → Σ (Γ.ᴹ γᴬ δᴬ) λ γᴹ → B.ᴹ γᴹ αᴬ βᴬ } ;
...
```

We do not define a "map" from the postulated syntax to these records, but instead we work directly with the "stack" of construction *on* the syntax, in the style of a shallow embedding (cf. Kaposi et al. [2019b]).

The construction steps of Γ^{E}, Γ^{W} Γ^{Σ}, and Γ^{R} are just further entries in the above-mentioned records. At the time of writing a formalization of the fact that these constructions do respect the substitutional equalities in the syntax of inductive-inductive types, is still incomplete, since it is quite tedious without the presence of a proper extensional ambient type theory. The following code snippet shows the specification of all of these operations on sort types. The interpretation of the universe then is the core of the definitions as presented in the last chapter:

```
record TyS (Γ : Con) : Set₂ where
  module Γ = Con Γ
  field
    ᴬ    : Γ.ᴬ → Set₁
    ᴹ    : ∀{γᴬ δᴬ} → Γ.ᴹ γᴬ δᴬ → ᴬ γᴬ → ᴬ δᴬ → Set
    w    : ∀(γc : Γ.Ec ᵃc) → Set → S.TyS
    R    : ∀{γᴬ}(α : Set)(αᴬ : ᴬ γᴬ) → S.TyS
    sg   : ∀{γc}(γ : (Γ.E ᵃC) γc){δc}(δ : (Γ.w γ ᵃC) δc)
               (α : Set)(ω : _ᵃS {zero} (w γc α)) → ᴬ (Γ.sg γc γ δc δ)

U : {Γ : Con} → TyS Γ
U {Γ} = record { ᴬ    = λ γ → Set ;
                 ᴹ    = λ γᴹ γᴬ δᴬ → γᴬ → δᴬ ;
                 w    = λ γ α → α S.⇒̂S S.U ;
                 R    = λ T Tᴬ → Tᴬ S.⇒̂S (T S.⇒̂S S.U) ;
                 sg   = λ γ δ α ω → Σ α ω }
```

Several Examples are formalized as a "sanity check" for the constructions. It might be helpful to the reader to check those for a good piece of insight into how the derivation of the running examples in this thesis work. Here is the encoding of the vector example:

```
module TestVec (A : Set) (N : Set) (z : N) (s : N → N) where

Γ : Con
Γ = · ▸S Π̂S N (λ _ → U)
      ▸P El (âppS vz z)
      ▸P Π̂P A (λ a → Π̂P N (λ n' → ΠP (âppS (vs{S}{P} vz) n')
           (El (âppS (vs{S}{P} (vs{S}{P} vz)) (s n')))))
```

7.7 Conclusions and Future Work

It is obvious that the work presented in this chapter is incomplete. This section shall serve to address what is missing to obtain a complete reduction of inductive-inductive types to inductive families, and to discuss how the constructions can be applied to implement inductive-inductive types.

Remark 7.7.1 (Functionality of the Relation)**.** While we succeeded in defining the eliminator relation, we do not show that it is actually the graph of a function. In Section 7.1, we saw that the next step in the construction of the eliminator would be to show that, for each algebra $\gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}$ the initial algebra of the relation $\Gamma^{\mathsf{R}}(\mathsf{con}(\Gamma^{\mathsf{E}}), \gamma^{\mathsf{A}})$ is right-total and left-unique.

While it is easy to *state* this property as a predicate on the algebra, proving it poses major problems that still need to be solved, and the solution of which we consider to be beyond the scope of this thesis.

As we can see in Lemma 7.1.1, the proof of uniqueness relies on the fact that if we are given two inhabitants $x_0^{\mathsf{A}}, x_1^{\mathsf{A}} : \alpha^{\mathsf{A}}$ where $\alpha^{\mathsf{A}} : \mathcal{U}^{\mathsf{A}} \equiv \mathcal{U}$ is some interpretation of a sort in the signature, and if we assume their relatedness to the same presyntactic datum in the form of $r_0 : \alpha^{\mathsf{R}}(x_0^{\mathsf{A}}, x)$ and $r_1 : \alpha^{\mathsf{R}}(x_1^{\mathsf{A}}, x)$, where $\alpha^{\mathsf{R}} : \mathcal{U}^{\mathsf{R}}(\ldots)^{\mathsf{A}} \equiv (\alpha^{\mathsf{A}} \to \alpha \to \mathcal{U})$ is an interpretation of the relation itself, then we can conclude that $x_0 = x_1$. But to achieve this, we need to apply induction over *two witnesses* of relatedness. But this *double induction* over the inductive family which represents the relation is special in the sense that it only needs to be supplied with linearly many inductive cases since the constructors of the relation are over different of its presyntactical indices. Capturing this phenomenon has not been done yet for our notion of inductive families and it is unclear how to formally prove the necessary properties for the general case, even though they are intuitively fulfilled in every conceivable example.

An alternative to the double induction would be to prove the uniqueness by *inversion* on the relation: In the context of Section 7.1 this means that whenever we have $r : R_{Con}(ext'(\Gamma, a), \gamma)$ we determine the form of γ by finding an instance of the following Σ-type:

$$(\gamma' : C) \times (\alpha : T(\gamma)) \times (\gamma = e(\gamma', \alpha)).$$

This inversion can obviously replace the double induction needed in the proof of Lemma 7.1.1, but it turns out that it is also difficult to define and prove. Instead of defining inversions, the same might be achieved by defining wellformedness predicate and eliminator relation *recursively* instead of an inductive definition, though our version of the syntax for inductive-inductive types makes this this recursive approach difficult as well, and we have not suceeded in generalizing this construction.

Remark 7.7.2 (From Relations to Morphisms)**.** Assuming that we have succeeded in proving that the initial algebra for $\Gamma^{\mathsf{R}}(\mathsf{con}(\Gamma^{\mathsf{E}}), \gamma^{\mathsf{A}})$ is the graph of a function, how do we turn it into an actual morphism in $\Gamma^{\mathsf{M}}(\mathsf{con}(\Gamma), \gamma^{\mathsf{A}})$? It turns out that while it is difficult to generalize a proof for the functionality of the eliminator relation, we can easily *state* this property by defining a type $\Gamma^{\mathsf{F}}(\gamma, \gamma^{\mathsf{A}}, \delta, \rho) : \mathcal{U}$ for each algebra γ of the type erasure, an arbitrary algebra $\gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}}$, an algebra over the wellformedness predicate δ and one over the relation itself ρ. Using this, we can define an *embedding* into the sets of morphisms in the form of an following operation with the following signature:

$$\frac{\begin{array}{c}\vdash \Gamma \\ \gamma_{\mathsf{S}} : \Gamma^{\mathsf{EA}}_{\mathsf{S}} \qquad \gamma : \Gamma^{\mathsf{EA}}(\gamma_{\mathsf{S}}) \qquad \gamma^{\mathsf{A}} : \Gamma^{\mathsf{A}} \qquad \delta_{\mathsf{S}} : \Gamma^{\mathsf{W}}_{\mathsf{S}}(\gamma)^{\mathsf{A}} \qquad \delta : \Gamma^{\mathsf{W}}(\gamma)^{\mathsf{A}}(\delta_{\mathsf{S}}) \\ \rho_{\mathsf{S}} : \Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})^{\mathsf{A}} \qquad \rho : \Gamma^{\mathsf{R}}(\gamma, \gamma^{\mathsf{A}})^{\mathsf{A}}(\rho_{\mathsf{S}}) \qquad \varphi : \Gamma^{\mathsf{F}}(\gamma, \gamma^{\mathsf{A}}, \delta, \rho)\end{array}}{\Gamma^{\mathsf{m}}(\gamma, \gamma^{\mathsf{A}}, \delta, \rho, \varphi) : \Gamma^{\mathsf{M}}\left(\Gamma^{\Sigma}(\gamma, \delta), \gamma^{\mathsf{A}}\right)}$$

This then gives us the desired non-dependent eliminator.

Remark 7.7.3 (Implementation of IITs)**.** Another bit of future work on the topic of reduction is to implement it in a suitable theorem prover. While Agda already provides inductive-inductive types, Coq and Lean both do not come equipped with a way to define them, and they are both extensible enough to use this reduction strategy to define a command for the definition of inductive-inductive types as “syntactic sugar”. In Lean 3, such an implementation has been attempted but it is still far away from being usable. Lean 4 will come with a more reasonable interface to automatically add definitions of inductive families to the environment, suggesting that we could process the use input of an inductive-inductive type as follows:

1. Reflect the input from Lean’s own expression type `expr` into a representation of the syntax of inductive-inductive types minus the substitutional equalities. This is where we implicitly check the input for positivity.

2. Add inductive families corresponding to type erasure, wellformedness predicate and eliminator relation to the environment.

3. Define the constructors using the Σ-construction.

4. Define the non-dependent eliminator using the eliminator relation.

5. Prove its uniqueness using a proof of the relation’s right-uniqueness.

6. Derive the dependent eliminator from the non-dependent one with the strategy laid out in [Kaposi et al., 2019a].

Remark 7.7.4 (Internalization-Based Approach)**.** It is important to note that there is an unpublished alternative approach to the reduction which uses an internalization of the syntax for inductive-inductive types and a term model, similar to our construction in Section 6.4. Ambroise Lafont formalized this approach in Agda. The drawbacks of this approach might be that there is a bigger overhead in a potential implementation based on this idea, due to the indirect nature of the construction.

Bibliography

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984. (Cited on pages 1 and 7.)

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. (Cited on pages 1, 14, 27, 63, 65, and 77.)

Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013. (Cited on pages 2 and 72.)

Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 108. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018a. (Cited on pages 4 and 73.)

Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy type theory in lean. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 479–495. Springer, 2017. ISBN 978-3-319-66106-3. (Cited on pages 5, 31, 37, and 67.)

Nicolai Kraus and Jakob von Raumer. Path spaces of higher inductive types in homotopy type theory. *arXiv preprint arXiv:1901.06022*, 2019. (Cited on page 5.)

Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory* (*Venice, 1995*), volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998. (Cited on page 7.)

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997. (Cited on pages 8 and 28.)

Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009. (Cited on pages 8 and 28.)

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015. (Cited on pages 8 and 28.)

J. Y. Girard. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. 1972. (Cited on page 9.)

Antonius JC Hurkens. A simplification of girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995. (Cited on page 9.)

Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994. (Cited on pages 16, 29, and 85.)

Kent Petersson and Dan Synek. A set constructor for inductive sets in martin-löf's type theory. In *Category Theory and Computer Science*, pages 128–140. Springer, 1989. (Cited on page 17.)

Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1-3):189–218, 2000. (Cited on page 17.)

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. (Cited on pages 17, 18, 61, and 85.)

Christian Sattler. On relating indexed w-types with ordinary ones, 2015. Abstract, presented at TYPES'15. (Cited on pages 18 and 61.)

Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed*

Lambda Calculi and Applications (TLCA), number 664 in Lecture Notes in Computer Science, 1993. (Cited on page 20.)

Steve Awodey and Michael A Warren. Homotopy theoretic models of identity types. In *Mathematical proceedings of the cambridge philosophical society*, volume 146, pages 45–55. Cambridge University Press, 2009. (Cited on page 21.)

Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after voevodsky). *arXiv preprint arXiv:1211.2851*, 2012. (Cited on pages 21 and 25.)

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016. (Cited on page 25.)

Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem proving in lean, 2015. URL `https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf`. (Cited on pages 31 and 67.)

Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. *Abstract of a talk at TYPES*, 2016. (Cited on page 31.)

Andrea Vezzosi. Cubical Agda, 2018. Extension to Agda, available in the main Agda repository at `https://github.com/agda/agda`. (Cited on pages 32 and 65.)

Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher groups in homotopy type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 205–214, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. (Cited on page 33.)

Ulrik Buchholtz and Kuen-Bang Hou (Favonia). Cellular cohomology in homotopy type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, 2018. (Cited on page 33.)

Ulrik Buchholtz and Egbert Rijke. The Cayley-Dickson construction in homotopy type theory. *arXiv preprint arXiv:1610.01134*, 2016. (Cited on page 33.)

Ulrik Buchholtz and Egbert Rijke. The real projective spaces in homotopy type theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–8, 06 2017. (Cited on page 33.)

Kuen-Bang Hou (Favonia) and Michael Shulman. The Seifert-van Kampen theorem in homotopy type theory. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-022-4. URL `http://drops.dagstuhl.de/opus/volltexte/2016/6562`. (Cited on pages 33, 40, 42, 43, and 61.)

Daniel Licata and Eric Finster. Eilenberg-MacLane spaces in homotopy type theory. In *Proceedings of the 29th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 66–74. ACM, 2014. (Cited on page 33.)

Daniel Licata and Guillaume Brunerie. $\pi_n(S^n)$ in homotopy type theory. volume 8307 of *LNCS*, pages 1–16. Springer, 2013. (Cited on page 33.)

Guillaume Brunerie. The James construction and $\pi_4(\mathrm{s}^3)$ in homotopy type theory. *CoRR*, 2017. URL `http://arxiv.org/abs/1710.10307`. (Cited on page 34.)

Egbert Rijke. The join construction. arXiv:1701.07538, 2017. (Cited on pages 34 and 39.)

Dan Licata. Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute, 2011. URL `https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant`. (Cited on page 37.)

Simon Boulier, Egbert Rijke, and Nicolas Tabareau. A coinductive approach to type valued equivalence relations. 2017. Abstract presented at the workshop on HoTT/UF in Oxford. (Cited on page 37.)

Daniel R Licata and Guillaume Brunerie. A cubical approach to synthetic homotopy theory. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 92–103. IEEE, 2015. (Cited on page 38.)

Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 122–129, 2016. (Cited on page 39.)

Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 595–604, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4391-6. (Cited on page 39.)

Daniel Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 223–232, 2013. (Cited on pages 40 and 41.)

Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. *J. ACM*, 63(6):51:1–51:45, January 2017. ISSN 0004-5411. (Cited on pages 48, 55, and 64.)

Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Principles of Programming Languages (POPL)*, pages 31–42, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. (Cited on page 48.)

Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal types. *Proc. ACM Program. Lang.*, 2(POPL):44:1–44:29, December 2017. ISSN 2475-1421. (Cited on page 48.)

Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science (MSCS)*, pages 1–30, Jan 2015. ISSN 1469-8072. (Cited on page 48.)

Floris van Doorn and Ulrik Buchholtz. The dependent universal property, 2017. Lean library file, available on GitHub at https://github.com/gebner/hott3/. (Cited on page 51.)

Eric Finster. Agda library file pushoutmono, 2017. Available on GitHub at https://github.com/HoTT/HoTT-Agda/blob/master/theorems/stash/modalities/gbm/PushoutMono.agda. (Cited on page 60.)

Jacob Lurie. Derived algebraic geometry vi: Ek algebras. *arXiv preprint arXiv:0911.0018*, 2009. (Cited on page 61.)

Ronald Brown, Philip J Higgins, and Rafael Sivera. *Nonabelian algebraic topology*. 2011. (Cited on page 61.)

Nicolai Kraus and Thorsten Altenkirch. Free higher groups in homotopy type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 599–608, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. (Cited on page 64.)

Gun Pinyo and Thorsten Altenkirch. Integers as a higher inductive type, 2018. Abstract, presented at TYPES'18. (Cited on pages 64 and 66.)

Evan Cavallo and Anders Mörtberg. Successor on biinv-int which cancels pred exactly, Dec 2018. Redtt implementation, available online at `https://github.com/RedPRL/redtt/blob/master/library/cool/biinv-int.red`. (Cited on pages 64 and 66.)

Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 255–264, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. (Cited on page 64.)

Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2018b. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-077-4. URL `http://drops.dagstuhl.de/opus/volltexte/2018/9190`. (Cited on pages 64 and 65.)

Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *arXiv preprint arXiv:1902.00297*, 2019. (Cited on pages 64 and 65.)

Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1, 2019. (Cited on page 65.)

Anders Mörtberg. Cubical agda, 2018. Blog post at https://homotopytypetheory.org/2018/12/06/cubical-agda/. (Cited on page 65.)

Anders Mörtberg and Andrea Vezzosi. An experimental library for cubical agda, 2018. Online at https://github.com/agda/cubical. (Cited on page 65.)

Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, Anders Mörtberg, and Jon Sterling. redtt – cartesian cubical proof assistant, 2018. Talk available online at `http://www.jonmsterling.com/pdfs/dagstuhl.pdf`, implementation at `https://github.com/RedPRL/redtt`. (Cited on page 65.)

Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *International Workshop on Types for Proofs and Programs*, pages 93–109. Springer, 2006. (Cited on page 72.)

Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Notices*, volume 51, pages 18–29. ACM, 2016. (Cited on page 72.)

Zhaohui Luo. Notes on universes in type theory. `http://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf`, November 2012. Accessed: March 22, 2019. (Cited on page 74.)

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 293–310. Springer, Cham, 2018. (Cited on page 77.)

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019a. (Cited on pages 77, 83, and 140.)

Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *International Conference on Typed Lambda Calculi and Applications*, pages 129–146. Springer, 1999. (Cited on page 85.)

Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. *arXiv preprint arXiv:1907.07562*, 2019b. (Cited on page 137.)