Radboud Universiteit Nijmegen

RADBOUD UNIVERSITY

MASTER THESIS COMPUTER SCIENCE

Higher Inductive Types

Niels van der Weide

supervised by
Herman GEUVERS and Henning BASOLD

July 3, 2016

Abstract

In type theories like Martin-Löf type theory one often assumes that there is only one proof for an equality, namely reflexivity. In homotopy type theory this assumption is dropped and non-trivial identity proofs are allowed. This becomes especially powerful if one can add identities to the theory, as then, for example, quotient types can be represented. A controlled way to add identity proofs are higher inductive types. These are like inductive types in classical type theories, only that we are not just allowed to specify constructors for elements of a type but also for identity proofs on that type.

Currently, many examples of higher inductive types are known but a general definition is missing. In this paper, we rectify this situation and introduce a general syntax for higher inductive types, which also allows constructors for higher paths. Moreover, we give the corresponding elimination and computation rules, and show that computations preserve types. Finally, we show how to interpret a subclass of these higher inductive types over categories, an instance of which is the category of topological spaces.

Acknowledgements

Some people have helped me making this thesis. First of all, I would like to thank Herman Geuvers for being my supervisor. Herman helped me with forming the text and the structure of the thesis. Also, he discussed the material with me and helped me writing it down clearly. Secondly, I would like to thank Henning Basold. He discussed most of the material with me and provided several good ideas.

Contents

CHAPTER 1

Introduction

Type theory is a form of logic that is naturally connected with intuitionistic first order predicate logic. Rather than propositions and proofs one talks about types and terms in type theory. There is even a deep connection: propositions and proofs correspond with types and terms respectively. This way one can use type theory to reason about mathematical statements.

In logic we describe a connective by giving introduction rules, which tell us how to prove the statement, and elimination rules, which tell us what we can prove with the statement. Similarly, types come with introduction rules, which tell us how to build terms of that type, and elimination rules, which tells us how to map elements of that type to some other type. However, a difference is that types also come with computation rules which say how terms can be simplified. The elimination rule also has another way of using it, namely it can be used as an induction principle. This allows us to prove properties of the type. All in all, just like in logic one can talk about the construction of new types from other types.

In addition type theory is naturally connected with computer science, and in particular with functional programming. There one makes programs of a certain type, and here types correspond with types and terms with programs. This is why in type theory computational interpretations are important, but type theory should also be able to describe everything used in functional programming. So, there should be function types and algebraic data types, but also the axioms should make sense in a computational setting.

However, for algebraic data types the usual connectives from logic are not sufficient, because they only give finite types. A possible way to talk about algebraic types is given by inductive types. To describe such a type, one gives constructors with their arities, and then the terms are built from the constructors. Maps out of an inductive type can be defined via recursion, and we clarify this using examples. One can represent the natural numbers as an inductive type $\mathbb{N}$ which has a constructor 0 with arity 0 and a constructor S with arity 1. The terms of this type are 0, $S(0)$, $S(S(0))$, and so on, and thus the terms correspond with natural numbers. To make a map from the type $\mathbb{N}$ to some other type Y, we need some point z of Y, to which 0 is mapped, and a map $s : Y \to Y$. If we know that n is mapped to y, then $S(n)$ is mapped to $s(y)$, and this way we can determine the complete map.

Another example would be the booleans, which has two constructors True and False, both of arity 0. This type only has two terms, namely True and False. One can make a map from the booleans to another type Y by giving two terms y_{True} and y_{False} of Y, which will be the images of True and False respectively. Both the natural numbers and the booleans have an elimination rule and with this rule, we can give an induction principle for these types. This way we can prove some property for all natural numbers or all booleans.

Recently, *homotopy type theory* has emerged as a new direction of type theory. In type theory we can talk about the equality of terms. For example, one can define the type $A \to B$, which is the type of functions from A to B. Suppose that we have some term f that maps every x to x. Then we can reduce the term $f(x)$ to x, and thus the terms x and $f(x)$ are *definitionally equal*. In general, terms can be reduced to other terms, and reduction gives the notion of definitional equality. However, it is also possible to define another notion of equality called *propositional equality*, and homotopy type theory is about that notion. Given two terms x and y of the same type, we can define a type whose terms are equality proofs of x and y. More concretely, it is defined as an inductive type, which has a constructor for reflexivity proofs. Homotopy type theory is different, because types are seen as topological spaces identified up to homotopy equivalence and equalities are seen as paths. In topology there can be multiple paths between x and x, so there can be multiple proofs which show that x is equal to itself. Hence, even though we may assume in type theory that identity proofs are unique, this is not sensible in homotopy type theory.

However, in topology one often speaks about spaces like the intervals. These do not have natural analogues in type theory, and to solve that one needs a way to construct spaces in homotopy type theory. For example, the interval should have two points 0 and 1, and an equality proof of 0 and 1. On the other hand, the circle has one point base and an equality proof loop between base and base. Note that the circle type would just be a point if identity proofs would be unique. For homotopy type theory we need a general framework to define such types called *higher inductive types*.

In addition such a framework allows us to talk about quotient types. To construct the integers modulo 2 one can take a quotient of the type $\mathbb{N}$ by saying that 0 is equal to 2. This can easily be generalized to give a data type for the integers modulo, because one just takes the quotient of $\mathbb{N}$. Similarly, a data type of symmetric trees can be defined, and thus higher inductive types give a wide range of new possible types in programming.

More generally, higher inductive types allow both point and path constructors. So, for inductive types we give constructors with their arities which give the points, but for higher inductive types also give constructors which are equalities between these points. Our goal is to give a formal definition and interpretation of higher inductive types. More concretely, we give a syntax of higher inductive types with introduction rules, elimination rules and computation rules, and we show how to interpret this in category theory.

This thesis is structured as follows. In Chapter 2 we start by giving a basic introduction to Type Theory, and then we discuss examples of types like product types, sum types, dependent types and such. Next we discuss the basics of homotopy type theory. We give some properties of paths, and we discuss the Univalence Axiom. In the last section we look at some known and elementary examples of higher inductive types. These are the sphere, and the interval. In Chapter 4 we give our original work on higher inductive types. We start with CW-complexes on which our interpretation of higher inductive types are based. Then we give several definitions of higher inductive types. These are increasing in complexity, and step by step the definition becomes more general and allows more examples.

CHAPTER 2

Type Theory

Martin-Löf type theory is a form of logic, in which types and terms form the elementary notions. In first orderlogic, propositions and proofs are seen as the basic notions instead. Formulas are built from connectives, so $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\forall$, $\exists$, $\perp$, $\top$. For all these connectives there are certain introduction and elimination rules which tell us how we can use them. In type theory we can only talk about types and terms, so we need to define them in a different way. We need formation rules for these types, and we need rules which tell us how to make functions between types.

One important feature of type theory is that we formulate logic in type theory. For this we need types that model the connectives. For example, with product types we model conjunctions and with sum types we model disjunctions. The introduction and elimination rules of the sums correspond with those of conjunction. This way we can make a correspondence between type theory and intuitionistic logic, and this is called the *Brouwer-Heyting-Kolmogorov* interpretation. With these predicates we can do intuitionistic logic, and this is why type theory is a constructive theory.

The last important feature of type theory is that we have *inductive types*. In the metalanguage we can define some concrete types like the natural numbers or product types, but one would also like to have general ways to define types. This can be compared to programming languages where users can define their own types instead of just using the basic types. These inductive types have a computational interpretation, which is also desired in type theory.

In this section we give a description of Martin-Löf type theory and a beginning of homotopy type theory. It is based on [**Uni13**].

2.1. Basic Type Theory

As said before, types are one of the elementary notions of type theory, and we start with basic type formation rules and contexts. We start with a formal introduction of the rules, and explain them afterwards. In our formal language we have types, denoted by A, B, $\ldots$, and variables $x, y, z, \ldots$. Next we define *contexts* inductively. The empty context $\cdot$ is a context, and if we have a context Γ, a type symbol A and a variable symbol x, then $\Gamma, x : A$ is another context. There are a number of statements in type theory, namely $\Gamma \vdash A : \textsc{Type}$, $\Gamma \vdash a : A$ and $\Gamma \vdash a = b$. However, when we work with type theory, we need to restrict the contexts we use. This is because we do not want a variable to occur twice in the context, so we do not want to allow $x : A, x : B \vdash f[x] : C$ where we use the notation $f[x]$ for a term with free variable x.

To do this, we define a function Var which takes the variables of a context Γ. We say that $\text{Var}(\cdot) = \emptyset$, and that $\text{Var}(\Gamma, x : A)$ is $\{x\} \cup \text{Var}(\Gamma)$. A *nice* context is one in which no variable occurs twice. So, the empty context is nice, and if Γ is nice and $x \notin \text{Var}(\Gamma)$, then $\Gamma, x : A$ is nice. The next notion we define is *well-formed*, and we notate this by Γ ctx. We have an axiom $\cdot$ ctx, and for every nice context $\Gamma, x : A$ a reasoning rule

$$\frac{\Gamma \text{ ctx} \qquad \Gamma \vdash A : \textsc{Type} \qquad x \notin \mathrm{Var}(\Gamma)}{\Gamma, x : A \text{ ctx}}$$

From now on all contexts are assumed to be well-formed.

The statement $\Gamma \vdash A : \textsc{Type}$ is read as 'A is a type in context Γ', and $\Gamma \vdash a : A$ is interpreted as 'Γ proves that a is a term of type A'. In the construction of types and terms only the variables in the corresponding context may be used. The reason we only use well-formed contexts, is because the involved types must indeed be types. This motivates the reasoning rules since this gives that every type symbol in the context must indeed be a type. Also, we exclude contexts like $x : A, x : B$ for convenience, because then the type of x is ambiguous.

Now we explain how to define new types in the metalanguage. We add rules, and we require that the universe of types is closed under these rules. There are several kind of rules which we need to give. First of all, we have *formation rules* which say when the type exist. The second kind of rules are *introduction rules* which are used to make terms of the type. *Elimination rules* say how we can use elements of the type, and the *computation rules* say the result of the elimination rules on the terms of the type.

2.2. Examples of Types

Let us discuss some examples of types, and we will introduce them via the system discussed in the end of the previous section.

2.2.1. Function Types. *Function types* are very important in type theory, and that is why we discuss them first. The formation rule is

$$\frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A \to B : \textsc{Type}}$$

Hence, if Γ is a well-defined context and both A and B are types, then we have a product type $A \to B$. The introduction rule is called λ-abstraction, and is formulated as

$$\frac{\Gamma, x : A \vdash f[x] : B}{\Gamma \vdash \lambda x.f[x] : A \to B}$$

Here we assume that $\Gamma, x : A$ is a well-formed context, and $f[x]$ is a term with x as free variable. So, we can make elements of the function type with λ-abstraction. Our elimination rule is function-application.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f\, a : B}$$

The computation rule says that $(\lambda x.f[x])\, a = f[a/x]$, where $f[a/x]$ is the term $f[x]$ with a substituted for x. This is called β-reduction in λ-calculus. With this type we can construct functions between types, and note that this is similar to implication in logic.

Note that we always have the identity function, namely $\mathrm{Id}_A = \lambda x.x$. By the computation rule we have $\mathrm{Id}_A\, a = a$, so it is indeed the identity function. Also, we can compose functions, so suppose that $\Gamma \vdash f : A \to B$ and that $g : B \to C$. Then we define $g \circ f = \lambda x.g\,(f\,x)$.

2.2.2. Product Types. Next we discuss product types, and the formation rule is

$$\frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A \times B : \textsc{Type}}$$

The introduction rule is

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

We have two elimination rules.

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash p_1\, x : A}$$

This is the first projection, and we have a second projection

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash p_2\, x : B}$$

The computation rule says that $p_1\,(a, b) = a$ and that $p_2\,(a, b) = b$. Note the similarities between the induction and elimination rules of the conjunction and the product type.

Products types are functorial meaning that if we have $f : A \to C$ and $g : B \to D$, then we get a map $f \times g : A \times B \to C \times D$. Define

$$f \times g = \lambda x.(f\,(p_1\,x), g\,(p_2\,x)).$$

2.2.3. Sum Types. Another possible type is the *sum type*, and the formation rule is the same.

$$\frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A + B : \textsc{Type}}$$

However, now we have two introduction rules

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \iota_1\, a : A + B}$$

This is the first inclusion, and we also have a second inclusion

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash \iota_2\, b : A + B}$$

The elimination rule says how to map $A+B$ to some type C assuming that $\Gamma \vdash C : \textsc{Type}$.

$$\frac{\Gamma \vdash p : A + B \qquad \Gamma, x : A \vdash f[x] : C \qquad \Gamma, y : B \vdash g[y] : C}{\Gamma \vdash (\textbf{case}\ p\ \textbf{of}\ x : A\ \textbf{then}\ f[x],\ \textbf{of}\ y : B\ \textbf{then}\ g[y]) : C}$$

The context $\Gamma, C : \textsc{Type}, x : A$ must be well-formed, so we must have $\Gamma \vdash C : \textsc{Type}$. The computation rule says that

$$\textbf{case}\ \iota_1\, a\ \textbf{of}\ x : A\ \textbf{then}\ f[x],\ \textbf{of}\ y : B\ \textbf{then}\ g[x] = f[a/x]$$

and that

$$\textbf{case}\ \iota_2\, b\ \textbf{of}\ x : A\ \textbf{then}\ f[x],\ \textbf{of}\ y : B\ \textbf{then}\ g[x] = g[b/y].$$

Note that the function is defined by matching: inhabitants of type A are mapped via f and inhabitants of B are mapped via g. Sum types are similar to disjunctions in logic.

Sum types are functorial as well. Suppose that we have $f : A \to C$ and $g : B \to D$, and now our goal is to make $f + g : A + B \to C + D$. Define

$$f + g = \lambda p.\,\textbf{case}\ p\ \textbf{of}\ x : A\ \textbf{then}\ f\,x,\ \textbf{of}\ y : B\ \textbf{then}\ g\,y$$

2.2.4. The Empty Type and the Unit Type. There are types which can be defined in the empty context. One example of this is the type $\bot$. The formation rule is easy, namely $\cdot \vdash \bot : \textsc{Type}$. There are no introduction rules or computation rules, but we have the elimination rule $x : \bot : \textsc{Type} \vdash !_C(x) : C$ if we can proof $C : \textsc{Type}$. Another example is the type $\top$, and again we have a formation rule $\cdot \vdash \top : \textsc{Type}$. We have an introduction rule $\cdot \vdash * : \top$, and we have an elimination rule

$$\frac{\Gamma, C : \textsc{Type} \vdash c : C}{\Gamma, x : \top \vdash \top\text{-rec}(c, x) : C}$$

with the computation $\top\text{-rec}(c, *) = c$. In the interpretation of intuitionistic logic we interpret $\top$ and $\bot$ as $\top$ and $\bot$ respectively

2.2.5. Dependent Products. The next step will be defining the dependent product and sum. The function type only allows maps which map every inhabitant to the same type. However, we would like to generalize this. Inhabitants can be mapped to different types, and this still gives a function. Such functions are called *dependent functions*, and if we work in a type theory with those, we must also give dependent elimination rules. The type of dependent functions is called the dependent products, and the formation rule is as follows

$$\frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma, x : A \vdash B[x] : \textsc{Type}}{\Gamma \vdash \prod x : A.B[x] : \textsc{Type}}$$

The introduction rule is called λ-abstraction, and is similar to λ-abstraction of nondependent functions.

$$\frac{\Gamma, x : A \vdash f[x] : B[x]}{\Gamma \vdash \lambda x.f[x] : \prod x : A.B[x]}$$

Our elimination rule is function-application.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : \prod x : A.B[x]}{\Gamma \vdash f\ a : B[a/x]}$$

and the computation rule says that $(\lambda x.f[x])\,a = f[a/x]$. With this type we can construct functions between types, and note that this is similar to the universal quantification in logic.

2.2.6. Dependent Sums. For dependent sums we have the same formation rule

$$\frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma, x : A \vdash B[x] : \textsc{Type}}{\Gamma \vdash \sum x : A.B[x] : \textsc{Type}}$$

The introduction rule is written as

$$\frac{\Gamma, a : A \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum x : A.B[x]}$$

Our elimination rule is similar to the one for sum types, and is given as follows.

$$\frac{\Gamma \vdash p : \sum x : A.B[x] \qquad \Gamma \vdash C : \textsc{Type} \qquad \Gamma, x : A, y : B[x] \vdash f[x, y] : C}{\Gamma \vdash \textbf{case}\ p\ \textbf{of}\ (x, y)\ \textbf{then}\ f[x, y] : C}$$

where the computation rules says that

$$\textbf{case } (a,b) \textbf{ of } B[x] \textbf{ then } f[x,y] = f[a/x, b/y]$$

whenever $y : B[x]$.

Dependent types generalize some types discussed before. For example, function types are a special case of dependent types if every $B[x]$ is equal. Also, product types are dependent types if every $B[x]$ is the same type. Looking at the rules, one can see the similarity between dependent products and universal quantifiers, and dependent sums and existential quantifiers. This connects dependent type theory with predicate logic.

With these rules we can make projection functions. We define

$$\pi_1 = \lambda p.\textbf{case } p \textbf{ of } B[x] \textbf{ then } x$$

which gives $x = \pi_1(x,y)$. However, we are unable to define the second projection which should have the following definition

$$\pi_2 = \lambda p.\textbf{case } p \textbf{ of } B[x] \textbf{ then } y$$

To solve this we need the dependent elimination rule.

If we work in a type theory with dependent types, then we also want a dependent elimination rule. These are based on the original elimination rule, and we make it dependent by making the type depend on the variable. For example, the dependent elimination rule for the dependent sum is given that we can prove $\Gamma, z : \sum x : A.B[x] \vdash Y[z] : \textsc{Type}$,

$$\frac{\Gamma \vdash p : \sum x : A.B[x] \qquad \Gamma, a : A, b : B[x] \vdash f[a,b] : Y[(a,b)/z]}{\Gamma \vdash \textbf{case } p \textbf{ of } (x,y) \textbf{ then } f : Y[p/z]}$$

With this rule we can the define the second projection of the dependent sum. For the other rules we can give dependent elimination rules as well.

2.2.7. W-types. The next types we discuss are the so-called W-types, and these are very general, because it is a way to describe inductive types. So far we only discussed concrete types, but W-types generalize many types. Before diving into technical formalisms, let us try to understand what a W-type means. In functional programming language we define a type by giving constructors which all have an arity. For example, the type $\mathbb{N}$ has two constructors, namely 0 with arity 0 and S with arity 1. Now we informally take the free algebra with these constructors, and then we get all terms of the form $S(\dots(S(0)))$. Also, one can make a type of unlabeled binary trees. Here we again have two constructors, namely the leaf constructor with arity 0 and a constructor node with arity 2. Again the actual type is made by taking the free algebra, and then we build trees with these constructors. This motivates how we construct inhabitants of a general W-type: we make trees.

We let A be the type which has all the constructors and the type $\prod a : A.B[a]$ has the arities of every constructor a. For the natural numbers we can let A be the type $\top + \top$, and for the arities we need to give two types $B[0]$ and $B[1]$. Define $B[0] = \bot$ and $B[1] = \top$. The inhabitants of this W-type, which we call $W_{x:A}B[x]$, are trees built from the constructors. So, if we have a constructor a and a function b of type $B[a] \to W_{x:A}B[x]$, then we should get some $\sup(a,b)$ of type $W_{x:A}B[x]$. We thus have a node a and its children are given by the function b. Such trees are *well-founded trees*, and that is why it is known as W-types.

How can we make a function from a W-type to some dependent type $Y[x]$? Suppose we have some node a with children b. If we know how to map the children to $Y[x]$, and we know how to 'combine' these into one inhabitant of $Y[x]$, then we are done. We can use induction to construct the map. So, for inhabitants $a : A$ and $b : B[a] \to W_{x:A}B[x]$, we need to give an inhabitant of $Y[\text{sup}(a,b)/x]$ if we have a map $g : \prod y : B[x].Y[b(y)/x]$. If $y : B[a/x]$, then $b\,y$ is a child of $\text{sup}(a,b)$, and all the children are formed this way, which is why we need the map to be of type $\prod y : B[x].Y[b\,y/x]$.

Now we describe W-types formally, and in this description we abbreviate $W_{x:A}B(x)$ to W. We start with the formation rule

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B[x] : \text{Type}}{\Gamma \vdash W_{x:A}B[x] : \text{Type}}$$

Hence, W-types must always exist. The formation rule has already been described and looks as follows

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : B[a] \to W}{\Gamma \vdash \text{sup}(a,b) : W}$$

Also, the elimination rule was discussed as well given that $\Gamma, w : W \vdash Y[w] : \text{Type}$

$$\frac{\Gamma, a : A, f : B[a] \to W, g : \prod b : B[x].Y[f(b)/a] \vdash c[a,f,g] : Y[\text{sup}(a,b)/x]}{\Gamma, w : W \vdash \text{wrec}(w,c) : Y[w/x]}$$

For the computation rule we need to tell what $\text{wrec}(\text{sup}(a,f),c)$ is, and we say that this is equal to $c(a,f,\lambda y.\,\text{wrec}(u\,y,c))$ This means that the function is defined by recursion. It is given that $a : A$ and $f : B[a] \to W$. Hence, using that $\text{wrec}(f\,y,c) : Y[f\,y/x]$, we see that this equality makes sense, because both $\text{wrec}(\text{sup}(a,f),c)$ and $c(a,f,\lambda y.w(u\,y,c)$ inhabit the type $Y[\text{sup}(a,f)/x]$.

For W-type we need an axiom called *function extensionality* which says that for $f,g : \prod x : A.B(x)$ we have $f = g$ iff $f\,x = g\,x$ for all $x : A$. This does not follow from the rules so far. Function extensionality is a nice way to determine whether two functions are equal, but without it, proving equality of functions is difficult. To determine whether inhabitants of W-types are equal, it is needed to determine whether two functions are equal, so function extensionality is needed.

2.2.8. Inductive Schemes. Because W-types require extensionality, it is not convenient to use in practice. Therefore, we need an alternative way to describe inductive type, and this is given by *inductive schemes*. Since we need to give constructors for an inductive type, we start by saying how to specify those.

Definition 2.2.1 (Polynomial Functor)**.** A *polynomial functor* is a map on types, and we define it by induction. If B is any type, then the map $H(X) = B$ is polynomial. The identity functor $H(X) = X$ is also polynomial. Lastly, if F and G are polynomial functors, then $F \times G$ and $F + G$ where $(F \times G)(X) = F(X) \times G(X)$ and $(F + G)(X) = F(X) + G(X)$ are polynomial functors as well. ⌟

More formally, a polynomial functor can be seen as a map which maps type symbols to type symbols. To give the elimination rule, we will need a lifting property. This says that for any family of types $x : A \vdash Y[x]$, we get a family $\overline{H}(Y)[x]$ on $H(A)$. We define it using induction, and the definition is as follows.

Definition 2.2.2 (Lifting)**.** Let $x : A \vdash Y[x]$ be a family of types and let H be a polynomial functor. We define the lifting $\overline{H}$ of H using induction where $\overline{H}(Y)$ is a family of types on $H(A)$. For $x : H(A)$ we define

- If $H(X) = B$, then $\overline{H}(Y)[x] = B$.
- If $H(X) = X$, then $\overline{H}(Y)[x] = Y[x]$.
- If $H(X) = F(X) \times G(X)$, then $\overline{H}(Y)[x] = \overline{F}(Y)[p_1\, x] \times \overline{G}(Y)[p_2\, x]$.
- If $H(X) = F(X) + G(X)$, then

$$\overline{H}(Y)[x] = \textbf{case}\ x\ \textbf{of}\ y : F(X)\ \textbf{then}\ \overline{F}(Y)[y],\ \textbf{of}\ z : G(X)\ \textbf{then}\ \overline{G}(Y)[z]. \quad \lrcorner$$

Now we can give the syntax of an inductive scheme.

Definition 2.2.3 (Inductive Scheme)**.** We define an *inductive scheme* according to the following syntax

$\texttt{Inductive}\ T\ (B_1 : \text{TYPE}) \dots (B_\ell : \text{TYPE}) :=$
$\quad |\ c_1 : H_1(T) \to T$
$\quad \dots$
$\quad |\ c_k : H_k(T) \to T$

We require that every H_i is polynomial in T and uses parameters $B_1, \dots, B_\ell$. Also, we denote $T(B_1, \dots, B_\ell)$ by T. The introduction rules for the points are

$$\frac{\Gamma \vdash B_i : \text{TYPE}\ \text{ for } i = 1, \dots, \ell \qquad \Gamma \vdash x : H_i(T)}{\Gamma \vdash c_i\, x : T}$$

and the elimination rule is

$$\frac{\Gamma, x : T \vdash Y[x] : \text{TYPE} \qquad \Gamma \vdash z_i : \prod x : H_i(T).\overline{H_i}(Y)(x) \to Y(c_i\, x)\ \text{ for } i = 1, \dots, k}{\Gamma \vdash T\text{-elim}(z_1, \dots, z_k) : \prod x : T.Y[x]}$$

The computation rules are for $t : H_i(T)$

$$T\text{-elim}(z_1, \dots, z_k)(c_i\, t) = (z_i\, t)(H_i(T\text{-elim}(z_1, \dots, z_k))). \quad \lrcorner$$

There are many examples of inductive types, and among them is the natural numbers which is defined as follows

$\texttt{Inductive}\ \mathbb{N} :=$
$\quad |\ 0 : \mathbb{N}$
$\quad |\ S : \mathbb{N} \to \mathbb{N}$

Similarly, one can define an inductive type for lists, booleans, and many more types.

2.2.9. Identity Type. In type theory there is an 'internal' notion of equality called *propositonal equality*, and this is defined using *Identity Types* $x \rightsquigarrow y$. These can be formed by the following rule.

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x \rightsquigarrow y : \text{TYPE}}$$

One identity we expect to hold is x equals x, so we have the introduction rule.

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \text{refl}_x : x \rightsquigarrow x}$$

Lastly, the elimination rule, also known as the J-rule, says that a map from $x \rightsquigarrow y$ to Y is determined by what it does on the reflexivity path. So, this is formulated as

$$\frac{\Gamma, p : x \rightsquigarrow y \vdash Y[x, y, p] : \text{TYPE} \qquad \Gamma, a : A \vdash t(a) : Y[a/x, a/x, \text{refl}_x /p]}{\Gamma, q : a \rightsquigarrow b \vdash J(t, x, y, p) : Y[a/x, b/y, q/p]}$$

with $J(t,x,x,\text{refl}_x) = t$. Applying the J-rule is also known as path induction.

Up to now we have given many computation rules, and the smallest equivalence relation containing these is called *definitional equality*. If two terms are definitionally equal, then they are propositionally equal. However, propositional equality might not imply definitional equality. If this is the case, then the type theory is *extensional*. Otherwise, the type theory is called *intensional*. An example of intensional type theory is homotopy type theory.

2.3. Homotopy Type Theory

So far we have seen many types, and all of these have computation rules. Reducing these terms give a form of equality called *definitional equality*. In type theory it is also possible to define an internal notion of equality, called *propositional equality*, and that is given by the identity types.

In homotopy type theory we interpret inhabitants of $x \rightsquigarrow y$ as paths from x to y. We have actual equalities between functions given by the computation rules However, there is no reason to assume that $x = y$ whenever we have an inhabitant of $x \rightsquigarrow y$. Therefore, we can interpret $p : x \rightsquigarrow y$ as something other than an equality between x and y. One possibility would be paths, but then we should also have some structure. In algebraic topology we have the functor Π sending X to the category whose objects are elements of x and whose arrows are paths. This is called the *fundamental groupoid*, and that is because $\Pi(X)$ is always a groupoid. Paths can be inverted, we have identity paths, paths can be composed. Also, it is a functor, so if we have $f : X \to Y$ then we get $\Pi(f) : \Pi(X) \to \Pi(Y)$. We would like that $x \rightsquigarrow y$ has the same structure as the fundamental groupoid.

To show all these properties, we use path induction. We start by showing that paths can be inverted.

Proposition 2.3.1. *For all contexts Γ, types A and terms $x, y : A$ we have a function $x \rightsquigarrow y \to y \rightsquigarrow x$. More formally, in type theory we can prove*

$$\Gamma, p : x \rightsquigarrow y \vdash p^{-1} : y \rightsquigarrow x$$

PROOF. We apply the J-rule with $Y[x,y,p] = y \rightsquigarrow x$ and $t[x] = \text{refl}_x$. Define $p^{-1} = J(t,x,y,p)$, and note that $p^{-1} : Y[x,y,p]$ so $p^{-1} : y \rightsquigarrow x$. □

The computation rule says that $\text{refl}_x^{-1} = \text{refl}_x$. Proposition 2.3.1 says that equality is symmetrical. The next property says that equality is transitive, and this can also be seen as the composition of paths.

Proposition 2.3.2. *Let Γ be a context, A be a type and let $x, y, z : A$ be terms. Then we have a function $(x \rightsquigarrow y) \times (y \rightsquigarrow z) \to x \rightsquigarrow z$. More formally, in type theory we can prove*

$$\Gamma, p : x \rightsquigarrow y, q : y \rightsquigarrow z \vdash p \circ q : x \rightsquigarrow z$$

PROOF. Let $p : x \rightsquigarrow y$, and now we apply the J-rule. Take $Y[y,z,q] = x \rightsquigarrow z$ and let $t[x']$ be p. Define $p \circ q = J(t,y,z,q)$ which is of type $Y[y,z,p] = x \rightsquigarrow z$. □

Note that the computation rule gives $p \circ \text{refl}_y = p$. Normally transitivity is seen as a property of a relation, but now we see it as an operation on paths. We can prove some properties of that operation

Proposition 2.3.3. *For $p : x \rightsquigarrow y, q : y \rightsquigarrow z, r : z \rightsquigarrow z'$ we have inhabitants for the following types*

(1) $p \rightsquigarrow p \circ \mathrm{refl}_y$ *and* $p \rightsquigarrow \mathrm{refl}_x \circ p$*;*
(2) $\mathrm{refl}_x \rightsquigarrow p \circ p^{-1}$ *and* $\mathrm{refl}_y \rightsquigarrow p^{-1} \circ p$*;*
(3) $p \rightsquigarrow (p^{-1})^{-1}$*;*
(4) $p \circ (q \circ r) \rightsquigarrow (p \circ q) \circ r$*.*

The first statement is written formally in type theory as

$$\Gamma, p : x \rightsquigarrow y \vdash \mathrm{NeutralRight}(p) : p \rightsquigarrow p \circ \mathrm{refl}_y$$

and this can be proven using path induction. Another important property of paths is that functions preserve paths.

Proposition 2.3.4. *We have a term* $\mathrm{ap}(f, p)$ *of the following type*

$$\Gamma, f : A \to B, x : A, y : A, p : x \rightsquigarrow y \vdash \mathrm{ap}(f, p) : f\ x \rightsquigarrow f\ y$$

PROOF. Again we use path induction but now with $Y[x, y, p] = f\ x \rightsquigarrow f\ y$. Define $t[x] = \mathrm{refl}_{f\ x}$, and then we take $\mathrm{ap}(f, p) = J(t, x, y, p)$. □

The last property we need, is called *transport* which is needed when we talk about dependent types. Later we talk about higher inductive types which are inductive types with extra paths. So, we add constructors and also inhabitants of identity types between the constructor. For example, one could add two constructors 0 and 1 and an inhabitant of $0 \rightsquigarrow 1$. To make a dependent function from this type to some other type, one would also want to say where the inhabitant of $0 \rightsquigarrow 1$ is mapped to. So, we first need to make types $Y[0]$ and $Y[1]$, but now we have a problem: 0 and 1 get mapped to different types, so we cannot make a path between their images. To solve this, one needs transport.

Proposition 2.3.5. *Suppose, that* $\Gamma, x : A \vdash Y[x]$ *and that we have* $p : a \rightsquigarrow b$ *for* $a, b : A$*. Then we have a function* $p_* : Y[a/x] \to Y[b/x]$*.*

Again this is proved by path induction, and we leave details to the reader.

Lemma 2.3.6. *For all* $a : A$ *and* $p : a \rightsquigarrow y$ *we have the definitional equality* $p_*(\mathrm{refl}_a) = p$*.*

PROOF. Let $a : A$, and take $Y[x] = a \rightsquigarrow x$. Now for $p : x \rightsquigarrow y$ we get a map$p_* : a \rightsquigarrow x \to a \rightsquigarrow y$. In particular, for $x = a$, we get $p_* : a \rightsquigarrow a \to a \rightsquigarrow y$, and then we have the definitional equality $p_*(\mathrm{refl}_a) = p$.. □

Definition 2.3.7 (Dependent Identity Type)**.** Let $x : A \vdash Y[x]$ be a family of type, and let $p : a \rightsquigarrow b$ be a path with $a, b : A$. Then for $y : Y[a/x]$ and $z : Y[b/x]$ we define $y \rightsquigarrow^Y_p z = p_*(y) \rightsquigarrow z$. ⌟

With this we can prove the following proposition

Proposition 2.3.8. *We have a term* $\mathrm{apd}(f, p)$ *of the type*

$$\Gamma, f : \prod x : A.Y[x], a : A, b : A, p : a \rightsquigarrow b \vdash \mathrm{apd}(f, p) : f\ a \rightsquigarrow^Y_p f\ b$$

Note that the type

$$\Gamma, f : \prod x : A.Y[x], a : A, b : A, p : a \rightsquigarrow b \vdash \mathrm{apd}(f, p) : f\ a \rightsquigarrow f\ b$$

is not well-formed. This is because $f\ a$ is of type $Y[a/x]$ and $f\ b$ is of type $Y[b/y]$.

Up to now everything we discussed in this section is just a part of normal type theory, and we have not defined a distinguished aspect of homotopy type theory yet. There are two such aspects, the first being the *univalence axiom* and the second being *higher inductive types*. We discuss the first now, and in the following section we shall discuss examples of higher inductive types. In Chapter 4 we formally describe higher inductive types. For types X and Y we can define the type $X \simeq Y$ as

$$\sum f : X \to Y \sum g : Y \to X (\prod x : X . g\,(f\,x) \rightsquigarrow x) + (\prod y : Y . f\,(g\,x) \rightsquigarrow y)$$

Intuitively, this means that there are functions $f : X \to Y$ and $g : Y \to X$ such that for all $x : X$ we have a path $g\,(f\,x) \rightsquigarrow x$ and for all $y : Y$ we have a path $f\,(g\,x) \rightsquigarrow y$. This means that there is a homotopy equivalence between X and Y. On the other hand, one can talk about $X \rightsquigarrow Y$ which means that 'there is a path between the types'. The second notion is stronger meaning that we can always make a function $u : X \rightsquigarrow Y \to X \simeq Y$. This is done via path induction where we note that we can always find an inhabitant of $X \simeq X$.

However, in general there might not be a function from $X \simeq Y$ to $X \rightsquigarrow Y$. The univalence axiom says that this is possible. Let us define a type $\mathrm{Eq}(f)$ for $f : A \to B$ as follows

$$\mathrm{eq}(f) = \sum g : B \to A (\prod x : X . g\,(f\,x) \rightsquigarrow x) + (\prod y : Y . f\,(g\,x) \rightsquigarrow y)$$

which means that f is an equivalence.

Definition 2.3.9 (Univalence Axiom)**.** We say that the *univalence axiom* holds iff for all types X and Y there is an inhabitant h of the type $\mathrm{eq}(u)$. ⌟

2.4. Elementary Examples of Higher Inductive Types

Before we dwell into a theory of higher inductive types, let us consider some examples and try to explain what they mean.

2.4.1. Interval Type. One can form an interval type I. We have two constructors $0 : I$ and $1 : I$, and we must have a path $\mathrm{seg} : 0 \rightsquigarrow 1$. This means that we have two introduction rules $0 : I$ and $1 : I$. Furthermore, we must have the introduction rule $\mathrm{seg} : 0 \rightsquigarrow 1.$, and we have the elimination rule

$$\frac{\Gamma, x : I \vdash Y[x] : \text{TYPE} \qquad \Gamma \vdash p_0 : Y[0/x] \qquad \Gamma \vdash p_1 : Y[1/x] \qquad \Gamma \vdash h : p_0 \rightsquigarrow^{Y}_{\mathrm{seg}} p_1}{\Gamma \vdash \mathrm{Irec}(p_0, p_1, h) : \prod x : I . Y[x]}$$

with the computation rule $\mathrm{Irec}(p_0, p_1, h)\,0 = p_0$ and $\mathrm{Irec}(p_0, p_1, h)\,1 = p_1$. Also, it is required that $\mathrm{apd}(\mathrm{Irec}(p_0, p_1, h), \mathrm{seg}) = h$. We see 0 and 1 as the endpoints of I, and seg is the line between 0 and 1, and that is why we call this type the interval.

Definition 2.4.1 (Contractible Type)**.** Suppose $\Gamma \vdash A : \text{TYPE}$. Then A is called *contractible* iff there is a term c of the type

$$\Gamma \vdash c : \sum x : A \prod y : A . x \rightsquigarrow y. \quad ⌟$$

Just as in topology, the interval is contractible, and it can be contracted to 0.

Lemma 2.4.2. *The type I is contractible.*

PROOF. We say that we can contract the interval to 0, so we need to make a function of $f : \prod y : A . 0 \rightsquigarrow y$ which is made using Irec. First, we need to give $p_0 : 0 \rightsquigarrow 0$ and $p_1 : 0 \rightsquigarrow 1$, and we take $p_0 = \mathrm{refl}_0$ and $p_1 = \mathrm{seg}$. Now we need a path $h : \mathrm{refl}_0 \rightsquigarrow^{Y}_{\mathrm{seg}} \mathrm{seg}$, so we need a path $\mathrm{seg}_*(\mathrm{refl}_0) \rightsquigarrow \mathrm{seg}$. Recall that by Lemma 2.3.6 we have $\mathrm{seg}_*(\mathrm{refl}_0) = \mathrm{seg}$, and thus we can take the path $\mathrm{refl}_{\mathrm{seg}}$. The desired inhabitant is $(0, \mathrm{Irec}(\mathrm{refl}_0, \mathrm{seg}, \mathrm{refl}_{\mathrm{seg}}))$. □

With a similar proof one can show that the interval can be contracted to 1 for which the contraction is $(1, \mathrm{Irec}(\mathrm{refl}_1, \mathrm{seg}, \mathrm{refl}_{\mathrm{seg}}))$. Another similarity is that paths can be seen as functions from I to the type. Suppose, we have a type A and $x, y : A$. Then we have a map $I \to A$ sending 0 to x and 1 to y iff we have an inhabitant of $x \rightsquigarrow y$. From left to right one uses $\mathrm{ap}(f, \mathrm{seg})$, and from right to left one can use induction on the interval.

2.4.2. Circle Type. Another type is the circle S^1, and now we have only one constructor base and one equality loop : base $\rightsquigarrow$ base. So, in this case we have one introduction rule base : S^1, and we have an elimination rule

$$\frac{\Gamma, x : S^1 \vdash Y[x] : \textsc{Type} \qquad \Gamma \vdash p : Y[\mathrm{base}/x] \qquad \Gamma \vdash h : p \rightsquigarrow^{Y}_{\mathrm{loop}} p}{\Gamma \vdash \mathrm{Srec}(p, h) : \prod x : S^1 . Y[x]}$$

with Srec base $= p$ and $\mathrm{apd}(\mathrm{Srec}(p, h), \mathrm{loop}) = h$.

This type must also have an equality base $\rightsquigarrow$ base, and one might wonder why not every type is a circle type. One always has such an equality, namely reflexivity, so it seems to be trivial. However, if there is a path loop $\rightsquigarrow$ refl, then all identity proofs are equal to refl. This is because $\mathrm{refl}_{\mathrm{base}}$ must be mapped to refl_p, so one cannot map this equality to a nontrivial path. If we have $x : A$ and a path $p : x \rightsquigarrow x$, then we have $\mathrm{Srec}(x, p)$ which sends loop to p. A path between loop and refl gives a path between p and refl, and therefore every identity proof is equal to refl.

CHAPTER 3

Categorical Interpretation

If logic only had syntax, then it would be a rather sterile field of mathematics. Therefore, instead of just studying the syntax one should also look at the semantics. One possible way of giving type theory semantics is by using *category theory*. The notion of a *category* was invented by Eilenberg and MacLane to give a formal language for homology and cohomology theories [**EM45**]. In later years it was developed to be used for more fields of mathematics. For example, Kan gave the definitions of *adjoint functors* [**Kan58**] which are crucial for category theory. In this section we will recall the basic definitions of category theory, and say how type theory can be interpreted in category theory. All proofs are left to the reader. Our presentation is based on [**ML78, MLM92**].

3.1. Basic Definitions and Examples

Definition 3.1.1 (Category)**.** A *category* consists of a collection $\mathscr{C}_0$ of *objects* and a collection $\mathscr{C}_1$ of morphisms. Furthermore, we require that we have operations $\mathrm{dom}, \mathrm{cod} : \mathscr{C}_1 \to \mathscr{C}_0$, an operation $\mathrm{id}_\cdot : \mathscr{C}_0 \to \mathscr{C}_1$ and an operation $\circ : \{(f,g) : \mathscr{C}_1 \times \mathscr{C}_1 : \mathrm{cod}(g) = \mathrm{dom}(f)\}$ written as $f \circ g$. Let us write $\mathrm{Hom}(A,B) = \{f \in \mathscr{C}_1 \mid \mathrm{dom}(f) = A \text{ and } \mathrm{cod}(f) = B\}$. All this data is required to satisfy the following requirements

(1) $f \circ (g \circ h) = (f \circ g) \circ h$;
(2) $f \circ \mathrm{id}_A = \mathrm{id}_A \circ f = f$;
(3) Each $\mathrm{Hom}(A,B)$ is a set. ⌟

Elements of $\mathscr{C}_1$ are called *arrows* or *morphism*, and we use the notation $f : A \to B$ or $A \xrightarrow{f} B$ for an arrow f with $\mathrm{dom}(f) = A$ and $\mathrm{cod}(f) = B$. A *diagram* in $\mathscr{C}$ is a graph of arrows of $\mathscr{C}$, and we say that it commutes iff all paths with the same start and endpoint lead to the same composition. We call an arrow $f : A \to B$ an *isomorphism* iff there is $g : B \to A$ such that $f \circ g = \mathrm{id}_B$ and $g \circ f = \mathrm{id}_A$. A *section* of an arrow $f : A \to B$ is an arrow $s : B \to A$ such that $f \circ s = \mathrm{id}_A$.

Examples of categories are the category **Top** of topological spaces with continuous functions or **Sets** with sets as object and functions as morphism. Many other examples can be found including the *slice category*.

Definition 3.1.2 (Slice Category)**.** Given a category $\mathscr{C}$ and an object $A \in \mathscr{C}_0$, we can define the *slice category* $\mathscr{C}/A$ to be the category with objects arrows $C \to A$ and the morphisms are commutative triangles

$$\begin{tikzcd}
C \arrow[rr] \arrow[dr] & & D \arrow[dl] \\
& A &
\end{tikzcd}$$

⌟

Maps between categories are called *functors*. These should preserve the structure of the category.

Definition 3.1.3 (Functor)**.** Given two categories $\mathscr{C}$ and $\mathscr{D}$ a *functor* consists of a mapping $F_0 : \mathscr{C}_0 \to \mathscr{D}_0$ and a mapping $F_1 : \mathscr{C}_1 \to \mathscr{D}_1$ such that

(1) For $f : A \to B$ we have $F_1(f) : F_0(A) \to F_0(B)$;
(2) $F_1(f \circ g) = F_1(f) \circ F_1(g)$;
(3) $F_1(\mathrm{id}_A) = \mathrm{id}_(F_1(A))$. ⌟

We will write $F(A)$ instead of $F_0(A)$ and $F(f)$ instead of $F_1(f)$. In the next section we will give many examples of functors. Using functors we can give another important example of a category.

Definition 3.1.4 (Algebras of a Functor)**.** Let F be a functor. Then an algebra for F is an arrow $F(A) \to A$. ⌟

The algebras of a functor F form a category where the morphisms are commutative squares

$$\begin{tikzcd}
F(A) \arrow[r, "F(f)"] \arrow[d] & F(B) \arrow[d] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

3.2. Constructions

Let us start with *products* and *coproducts*.

Definition 3.2.1 (Product)**.** Let $\mathscr{C}$ be a category and let A and B be objects. Then we say C is a *product* of A and B iff we have $p_1 : C \to A$ and $p_2 : C \to B$ such that for arrows $f : X \to A$ and $g : X \to B$ there is a unique arrow $h : X \to C$ such that $p_1 \circ h = f$ and $p_2 \circ h = g$. In a diagram this is written as

$$\begin{tikzcd}
 & X \arrow[dl, "g"'] \arrow[d, dashed, "h"] \arrow[dr, "f"] & \\
A & C \arrow[l, "p_1"] \arrow[r, "p_2"'] & B
\end{tikzcd}$$

⌟

Note that products are unique up to unique isomorphism meaning that whenever C and D are both products of A and B, then there is a unique isomorphism $C \to D$. If A and B have a product, then we denote it by $A \times B$.

Definition 3.2.2 (Coproduct)**.** Let $\mathscr{C}$ be a category and let A and B be objects. Then we say C is a *coproduct* of A and B iff we have $\iota_1 : A \to C$ and $\iota_2 : B \to C$ such that for arrows $f : A \to X$ and $g : B \to X$ there is a unique arrow $h : C \to X$ such that $h \circ \iota_1 = f$ and $h \circ \iota_2 = g$. In a diagram this is written as

$$\begin{tikzcd}
 & X & \\
A \arrow[ur, "f"] \arrow[r, "\iota_1"'] & C \arrow[u, dashed, "h"] & B \arrow[ul, "g"'] \arrow[l, "\iota_2"]
\end{tikzcd}$$

⌟

Coproducts are unique up to unique isomorphism as well, and the coproduct of A and B is denoted by $A+B$ whenever it exists. In **Sets** every two sets have a product, namely the product of sets, and a coproduct which is the disjoint sum. Now we can discuss *initial objects* and *terminal objects*.

Definition 3.2.3 (Initial Object)**.** Let $\mathscr{C}$ be a category, and let A be an object of $\mathscr{C}$. Then we say that A is an initial object iff for all objects C we have $\#(\mathrm{Hom}(A,C)) = 1$. $\lrcorner$

Definition 3.2.4 (Terminal Object)**.** Let $\mathscr{C}$ be a category, and let A be an object of $\mathscr{C}$. Then we say that A is an initial object iff for all objects C we have $\#(\mathrm{Hom}(A,C)) = 1$. $\lrcorner$

Initial objects and terminal objects are unique up to unique isomorphism as well. We denote the initial object by 0 and the terminal object by 1 whenever they exist. Let us now discuss an important example of initial objects, namely *initial algebras*.

Definition 3.2.5 (Initial Algebra)**.** Let F be a functor. Then we say some algebra $F(A) \to A$ is the *initial algebra* of F iff it is the initial object of the category of algebras of F. $\lrcorner$

Suppose, $\mathscr{C}$ is a category with an initial object and coproducts. Define $F : \mathscr{C} \to \mathscr{C}$ to be the functor with $F(A) = 1+A$. An algebra of F is an arrow $1+A \to A$, so we have an arrow $0_A : 1 \to A$ and an arrow $S_A : A \to A$. If F has an initial algebra, then this algebra is called a *natural numbers object* of $\mathscr{C}$. Let us assume that F indeed has an initial algebra, and let us denote it by N. Since N is an algebra, we have arrows $0_N : 1 \to N$ and $S_N : N \to N$. To map N to any object X, we need to give $0_X : 1 \to X$ and $S_X : X \to X$. In **Sets** the natural numbers object is the natural numbers $\mathbb{N}$, and the induction is given by the fact that the algebra is initial.

Another construction is called the *pullback* or *fiber product*.

Definition 3.2.6 (Pullback)**.** Let $\mathscr{C}$ be a category and suppose we have arrows $f : A \to C$ and $g : B \to C$. Then we say P is a *pullback* iff we have $p_1 : P \to A$ and $p_2 : P \to B$ such that

- $f \circ p_1 = g \circ p_2$.
- For all $h_1 : X \to A$ and $h_2 : X \to B$ such that $f \circ h_1 = g \circ h_2$, there is a unique k such that $k \circ p_1 = h_1$ and $k \circ p_2 = h_2$.

This can be depicted as follows

$$\begin{tikzcd}
X \arrow[rd, dashed, "k"] \arrow[rrd, "h_2"] \arrow[rdd, "h_1"'] & & \\
& P \arrow[r, "p_2"'] \arrow[d, "p_1"] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & C
\end{tikzcd}$$

$\lrcorner$

Often we indicate pullbacks in a square as follows

$$\begin{tikzcd}
P \arrow[r] \arrow[d] \arrow[rd, phantom, very near start, "\lrcorner"] & B \arrow[d] \\
A \arrow[r] & C
\end{tikzcd}$$

or we write $P = A \times_C B$. It is more logical to call it the fiber product, because it can be used to give the fiber (preimage) of a map. However, the terminology 'pullback' is more common, so we will call it that. Next we can define pushouts.

Definition 3.2.7 (Pushout)**.** Let $\mathscr{C}$ be a category and suppose we have arrows $f : C \to A$ and $g : C \to A$. Then we say P is a *pushout* iff we have $\iota_1 : A \to P$ and $\iota_2 : B \to P$ such that

- $\iota_1 \circ f = \iota_2 \circ g$.
- For all $h_1 : A \to X$ and $h_2 : B \to X$ such that $h_1 \circ f = h_2 \circ g$, there is a unique k such that $\iota_1 \circ k = h_1$ and $\iota_2 \circ k = h_2$.

In a diagram this looks like

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "\iota_1"'] \arrow[ddr, "h_1"] & \\
B \arrow[r, "\iota_2"] \arrow[drr, "h_2"'] & P \arrow[dr, dashed, "k"] & \\
 & & X
\end{tikzcd}$$

⌟

In a diagram we indicate a pushout as follows

$$\begin{tikzcd}
A \arrow[r] \arrow[d] & C \arrow[d] \\
B \arrow[r] & P \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

and we write $P = B \coprod_A C$. So, if the maps are clear from the context, then we denote the pullback of A and B by $A \times_C B$ and the pushout by $A \coprod_C B$. Another construction generalizes quotients and it is called the *coequalizer*.

Definition 3.2.8 (Coequalizer)**.** Let $\mathscr{C}$ be a category and suppose we have arrows $f, g : A \to B$. Then we say $c : C \to A$ is a *coequalizer* of f and g iff for all $h : X \to A$ there is a unique $k : X \to C$ such that $h = c \circ k$. ⌟

Up to now we have described constructions by giving a universal mapping property. *Adjoint functors* allow different constructions. For this we first need the definition of a natural transformation.

Definition 3.2.9 (Natural Transformation)**.** Let $F, G : \mathscr{C} \to \mathscr{D}$ be two functors, and suppose that for each object A of $\mathscr{C}$ we have a map $\eta_A : F(A) \to G(A)$. Then we say that $\eta_\cdot$ is a natural transformation iff for all arrows $f : A \to B$ the following diagram commutes

$$\begin{tikzcd}
F(A) \arrow[r, "\eta_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\
F(B) \arrow[r, "\eta_B"'] & G(B)
\end{tikzcd}$$

⌟

All constructions so far can be said in the language of *limits* and *colimits*. We will only discuss colimits, and for that we first need to define *cocones*.

Definition 3.2.10 (Cocone)**.** Let I be a category and let $F : I \to \mathscr{C}$ be a functor. A *cocone* on I is a diagram consisting of an object X of $\mathscr{C}$ and morphisms $g_i : F(i) \to X$ for every object i of I such that the following diagram commutes for every arrow $f : i \to j$

$$\begin{tikzcd}
F(i) \arrow[rr, "F(f)"] \arrow[dr, "g_i"'] & & F(j) \arrow[dl, "g_j"] \\
& X &
\end{tikzcd}$$

⌟

Note that we can talk about a category of cocones on F.

Definition 3.2.11 (Colimit)**.** Let I be a category and let $F : I \to \mathscr{C}$ be a functor. Then the *colimit* of F is defined as the initial object in the category of cocones on F if it exists. ⌟

In a similar way limits can be defined. Coproducts, pushouts, coequalizers and initial objects are examples of colimits. Some constructions cannot be defined as limits or limits, but instead one needs the notion of an *adjoint functor*.

Definition 3.2.12 (Adjoint)**.** Let $\mathscr{C}$ and $\mathscr{D}$ be categories, and let $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ be functors. Then we say that F and G are an *adjoint pair*, denoted by $F \dashv G$, iff for all objects A of $\mathscr{C}$ and all objects B of $\mathscr{D}$ the sets $\mathrm{Hom}(A, G(B))$ and $\mathrm{Hom}(F(A), B)$ are isomorphic. This isomorphism is required to be a natural transformation. ⌟

Adjoints do not have to exist. In this situation F is called the *left adjoint* and G is called the *right adjoint*.

Definition 3.2.13 (Exponential)**.** Let $\mathscr{C}$ be a category with products, and let A be an object of $\mathscr{C}$. Then the *exponential* $(\cdot)^A$ is the right adjoint of the functor $\cdot \times A$ whenever it exists. ⌟

The exponential gives the functions from A to C.

Definition 3.2.14 (Cartesian Closed)**.** A category $\mathscr{C}$ is called *cartesian closed* iff it has all products and for all A the exponential $(\cdot)^A$ exists. ⌟

Definition 3.2.15 (Locally Cartesian Closed)**.** A category $\mathscr{C}$ is called *locally cartesian closed* iff for all objects A the slice category $\mathscr{C}/A$ is cartesian closed. ⌟

There are many examples of locally cartesian closed categories. For example, every *topos* is locally cartesian closed [**MLM92**], and thus **Sets** is locally cartesian closed.

Let us discuss a condition which shows the existence of initial algebras, and for that we start with a requirement on functors.

Definition 3.2.16 (Polynomial Functor)**.** Let $\mathscr{C}$ be a category with all products and coproducts. The notion *polynomial functor* on $\mathscr{C}$ is defined inductively. First of all, every constant functor $F(X) = A$ is a polynomial functor, and the identity functor $F(X) = X$ is a polynomial functor. Furthermore, if we have two polynomial functors F and G, then both $F \times G$ and $F \coprod G$ are polynomial functors as well. Here $(F \times G)(X) = F(X) \times G(X)$ and $(F \coprod G)(X) = F(X) \coprod G(X)$. ⌟

Polynomial functors have initial algebras if the category satisfies some conditions.

Theorem 3.2.17. *If $\mathscr{C}$ is a cartesian closed category which has all limits and colimits, then every polynomial functor $F : \mathscr{C} \to \mathscr{C}$ has an initial algebra.*

This can be proven as follows. Using the fact that $\mathscr{C}$ is cartesian closed, one can prove that F commutes with colimits and limits. The initial algebra is then defined as the colimit of the diagram

$$0 \longrightarrow F(0) \longrightarrow \ldots \longrightarrow F^{(n)}(0) \longrightarrow \ldots$$

Now we can give the definitions in which we will interpret type theory.

Definition 3.2.18 (Martin-Löf Category)**.** A category $\mathscr{C}$ is called a *Martin Löf category* iff the following requirements are satisfied

(1) It is locally cartesian closed;
(2) It has all limits;
(3) It has all colimits. ⌟

3.3. Interpreting Type Theory

Our goal is to give interpretations of type theory in the language of category theory, and our presentation is based on [**AW09**]. For this we need to interpret dependent sums, dependent products and inductive types in a category. So, in this complete section we will assume that $\mathscr{C}$ is a fixed Martin-Löf category, and our goal is to interpret type theory in $\mathscr{C}$. In this interpretation types are interpreted as objects and terms as morphisms.

To do this, we first need to recall the basic statements, and then give an interpretation for those When these have an interpretation, we can continue by interpreting more complicated types like $\perp$, $\top$, product types and so on. However, for these types certain rules need to be satisfied. For example, the product type has an introduction rule

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

To interpret this, we assume that we have interpretations of $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$. Using this data, we need to make an interpretation of $\Gamma \vdash (a, b) : A \times B$. In this fashion we can make interpretations of all desired types. Most arguments are similar, so we give a sketch and leave the details as an exercise for the reader.

In type theory we have three basic statements. First, we can say that A is a type which is written as $\Gamma \vdash A : \text{Type}$. Secondly, we can say that t is a term of type A and that is denoted as $\Gamma \vdash t : A$. This statement only makes sense if we have $\Gamma \vdash A$. Lastly, we can say that two terms t and t' are equal if they are of the same type, and we denote this by $\Gamma \vdash t = t'$.

Let us start by interpreting $\Gamma \vdash A : \text{Type}$, and we will interpret this as a chain of arrows. To do this, we need to use induction on Γ. The statement $\cdot \vdash A : \text{Type}$ is interpreted as the arrow $[\![A]\!] \to 1$ to the terminal object. Recall that we work in a category which has a terminal object, so this makes sense. Now suppose that Γ is $x_1 : A_1, \ldots, x_n : A_n$, and that $\Gamma \vdash A : \text{Type}$ is interpreted as a chain of arrows $B_n \to \ldots \to B_0$. Then we interpret $\Gamma, x_{n+1} : A_{n+1} \vdash A : \text{Type}$ as the chain $[\![A_{n+1}]\!] \to B_n \to \ldots \to B_0$. So, in particular the statement $x_1 : A_1, \ldots, x_n : A_n \vdash A : \text{Type}$ is interpreted as the chain $[\![A]\!] \to [\![A_n]\!] \to \ldots \to [\![A_1]\!]$. We will write the interpretation of $\Gamma \vdash A : \text{Type}$ often as $[\![A]\!] \to [\![\Gamma]\!]$.

Let us have a short intermezzo to explain why this makes sense, and for that we briefly describe how to interpret these statements in the category **Sets**. Then we see

$[\![A_2]\!]$ as a family $\{X_i\}_{i \in [\![A_1]\!]}$, and we have a projection sending $x \in X_i$ to i. Intuitively this explains why this interpretation makes sense, because it has the right dependencies.

Secondly, if we have $x_1 : A_1, \ldots, x_n : A_n \vdash A : \text{TYPE}$, then $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is interpreted as an arrow $[\![t]\!]$ which is a section of the arrow $[\![A]\!] \to [\![A_n]\!]$ given by the interpretation of $x_1 : A_1, \ldots, x_n : A_n \vdash A : \text{TYPE}$. We interpret $x_1 : A_1, \ldots, x_n : A_n \vdash A : \text{TYPE}$ as a chain of arrows $[\![A]\!] \to [\![A_n]\!] \to \ldots \to [\![A_1]\!]$, so we have an arrow $[\![A]\!] \to [\![A_n]\!]$, and a term of A is a section of this arrow. This means that $\vdash t : A$ is interpreted as an arrow $1 \to [\![A]\!]$. Definitional equality is interpreted as equality between arrows.

Now we get to the more interesting part: how can we interpret more complicated types. This will also clarify the definitions. Let us start with the zero type.

Definition 3.3.1 (Interpretation of the Zero Type). The zero type $\perp$ is interpreted as the initial object of the category. ⌟

Why does this make sense? The formation rule says $\vdash \perp : \text{TYPE}$. This needs to hold, so we need an arrow $0 \to 1$ which we have if the category has an initial and a terminal object. There are no introduction rules, but the elimination rule says $x : \perp \vdash !_C(x) : C$. Since we have $x : \perp \vdash C : \text{TYPE}$, we can assume we have an arrow $C \to 0$. We need to make a section, so an arrow $0 \to C$, and since 0 is initial, we have such an arrow. For it to be a section we need that the composition $0 \to C \to 0$ is the identity, and that is the case.

Definition 3.3.2 (Interpretation of the Unit Type). The unit type $\top$ is interpreted as the terminal object of the category. ⌟

The reasoning that this makes sense, is similar. Again the formation rule holds, because the category is assumed to have a terminal object. Now we only need to verify the introduction rule

Definition 3.3.3 (Interpretation of Product Types). The type $A \times B$ is interpreted as the product of $[\![A]\!]$ and $[\![B]\!]$. ⌟

The formation rule holds, because we have products in our category. The elimination rules hold as well. Suppose, we have an arrow $[\![A]\!] \times [\![B]\!] \to [\![\Gamma]\!]$ with a section $[\![x]\!] : [\![\Gamma]\!] \to [\![A]\!] \times [\![B]\!]$. Then we get arrows $[\![\Gamma]\!] \to [\![A]\!]$ and $[\![\Gamma]\!] \to [\![B]\!]$, because we have projections $p_1 : [\![A]\!] \times [\![B]\!] \to [\![A]\!]$ and $p_2 : [\![A]\!] \times [\![B]\!] \to [\![B]\!]$. Lastly, for the introduction rule we assume that we have sections $[\![a]\!] : [\![\Gamma]\!] \to [\![A]\!]$ and $[\![B]\!] : [\![\Gamma]\!] \to [\![B]\!]$. By the universal property we then get an arrow $[\![(a, b)]\!] : [\![\Gamma]\!] \to [\![A]\!] \times [\![B]\!]$ of which the first projection is $[\![a]\!]$ and the second projection is $[\![b]\!]$. Hence, all the given rules are satisfied, and thus $[\![A]\!] \times [\![B]\!]$ is the product type. Similarly, we can interpret the sum type as coproduct.

Definition 3.3.4 (Interpretation of Sum Types). The type $A + B$ is interpreted as the coproduct of $[\![A]\!]$ and $[\![B]\!]$. ⌟

Another interesting example are function types, because there are interpreted as exponentials. So, we interpret these as the right adjoint of the product.

Definition 3.3.5 (Interpretation of Function Types). The type $A \Rightarrow B$ is interpreted as the exponential B^A. ⌟

Because we assume the category has all exponentials, the formation rule holds. Let us now show that the introduction rule holds. Since we have the premise $\Gamma, x : A \vdash$

$f(x) : B$ we have an arrow $A \to B$. Note that this gives an arrow $\Gamma \times A \to A \to B$ by composing with the second projection. By adjunction we have

$$\mathrm{Hom}(\Gamma \times A, B) \cong \mathrm{Hom}(\Gamma, B^A),$$

and thus we get an arrow $\mathrm{Hom}(\Gamma, B^A)$ which is what we needed.

For the elimination rule we assume to have arrows $\Gamma \to A$ and $\Gamma \to B^A$. Note that this gives an arrow $\Gamma \to B^A \times A$. We have an arrow $\mathrm{id}_{B^A} : B^A \to B^A$, and we have by adjunction that

$$\mathrm{Hom}(B^A, B^A) \cong \mathrm{Hom}(B^A \times A, B).$$

So, the arrow id_{B^A} gives an arrow $\varepsilon : B^A \times A \to B$. Since we already have an arrow $\Gamma \to B^A \times A$, we get an arrow $\Gamma \to B$ which is what we wanted.

In **Sets** the set B^A consists of all functions $A \to B$, and the arrow ε sends a pair (f, a) to $f(a)$. This means that here it is indeed the application. Also, if we have a function $f : A \to B$, then the isomorphism sends it to the map f in the function set.

3.3.1. Dependent Products and Dependent Sums. So far we did not use that the category is locally cartesian closed. We only used that it is cartesian closed. However, to interpret dependent products and dependent sums we will need that the category is locally cartesian closed. Let us discuss some notation first. Given an arrow $f : B \to A$, we can make a functor $f^* : \mathscr{C}/A \to \mathscr{C}/B$, called the *pullback along f* or *reindexing along f*. An arrow $g : C \to A$ is sent to h where we have the following pullback square

$$\begin{tikzcd} P \arrow[r] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & C \arrow[d, "g"] \\ B \arrow[r, "f"'] & A \end{tikzcd}$$

Here we assume that we have a pullback functor meaning that we need to choose the pullback for each diagram. With this notation we can give the following theorem.

Theorem 3.3.6. *A category is locally cartesian closed iff for all arrows f the functor f^* has both a left adjoint and a right adjoint.*

We are working in a category $\mathscr{C}$ which is locally cartesian closed. Hence, every pullback functor has both a left adjoint and a right adjoint. Note that to show that $\prod x : A.B[x]$ or $\sum x : A.B[x]$ is a well-founded type, we need to show that $x : A \vdash B[x] : \textsc{Type}$. So, we have an arrow $f : [\![B]\!] \to [\![A]\!]$.

Definition 3.3.7 (Interpretation of Dependent Product)**.** We interpret the dependent product $\prod x : A.B[x]$ is interpreted as the right adjoint of the pullback along f where $f : [\![B]\!] \to [\![A]\!]$ is the interpretation of $x : A \vdash B[x] : \textsc{Type}$. ⌟

Definition 3.3.8 (Interpretation of Dependent Sum)**.** The dependent sum $\sum x : A.B[x]$ is interpreted as the left adjoint of the pullback along f where $f : [\![B]\!] \to [\![A]\!]$ is the interpretation of $x : A \vdash B[x] : \textsc{Type}$. ⌟

We leave it as an exercise to the reader to verify that this definition makes sense. Most of the rules can be checked, but for the computation rules one needs the *Beck-Chevalley* condition for which we refer the reader to [**MLM92**]. Examples of possible universes in which all these conditions hold, are toposes like **Sets** or presheaf categories.

3.3.2. Inductive Types. Inductive Types are interpreted using initial algebras. We can either describe an inductive type as a W-type or by giving constructors, and in this section we chose the latter approach. So, we define an inductive type T as follows

```
Inductive T :=
  | c_1 : H_1(T) → T
  ...
  | c_n : H_n(T) → T
```

where every H_i is a polynomial functor in T. The introduction rules are $c_i(x) : T$ where $x : H_i(T)$. Also, we have an elimination rule, namely

$$\frac{\Gamma, x : T \vdash Y(x) : \textsc{Type} \qquad \Gamma \vdash z_i : \prod x : H_i(T).\overline{H_i}(Y)[x] \to Y(c_i\, x) \;\; \text{for } i = 1, \ldots, k}{\Gamma \vdash T\text{-elim}(z_1, \ldots, z_k) : \prod x : T.Y(x)}$$

To give an interpretation of T, we start by defining an endofunctor F_T on $\mathscr{C}$. Note that every H_i is a polynomial functor, so we can define a polynomial functor $F_T(X) = H_1(X) \coprod \ldots \coprod H_n(X)$. We interpret T as the initial algebra of F_T and we denote it by $[\![T]\!]$.

Since $[\![T]\!]$ is an algebra of F_T, we have the required introduction rules. Also, the elimination rule is satisfied, because T is initial. The elimination rule basically says that if we have any other algebra of F_T, then we can map T into it, and this follows from the fact that $[\![T]\!]$ is initial.

CHAPTER 4

Higher Inductive Types

4.1. CW-Complexes

In topology, the notion of a *CW-complex* is very important [**May99, Hat02**]. These are the topological spaces that are built from the cells $I^n = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid x_i \in [0,1]\}$. To build CW-complexes, we need the notion of gluing, which can be defined using pushouts. The idea of constructing a CW-complex X is as follows. We start with a number of copies of I^0, so a discrete space. Afterwards we glue copies of I^1, I^2 and so on, which are glued increasing in dimension. This gives approximations $X_0, X_1, \ldots$ with $X = \bigcup_{i \in \mathbb{N}} X_i$.

Let us describe this more formally for which first we recall some notation. In the remainder of this section we work in the category of topological spaces with continuous maps. The boundary of an interval $\partial(I^1)$ is defined by $\{0,1\}$, and the boundary $\partial(I^n)$ of an n-cube is defined by $\{(x_1, \ldots, x_n) \mid x_i \in \{0,1\}$ for some $i\}$. If we have a map $d : \partial(I^n) \to X$, then we can glue I^n to X via the following pushout.

$$\begin{tikzcd}
\partial(I^n) \arrow[r, "d"] \arrow[d, hook] & X \arrow[d] \\
I^n \arrow[r] & P \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

The pushout P is the space $X \coprod I^n$ where x and $d(x)$ are identified for all $x \in \partial(I^n)$. This defines the gluing required in the construction.

To build a CW-complex, we start with a collection J_0 of points, and we define X_0 by $\coprod x \in J_0.I^0$. The next space X_1 is obtained from X_0: we need a collection J_1 and for each $j \in J_1$ a boundary map $d_j^1 : \partial(I^1) \to X_0$. We define X_1 as the following pushout.

$$\begin{tikzcd}
\coprod j \in J_1.\partial(I^1) \arrow[r, "\coprod j \in J_1.d_j^1"] \arrow[d, hook] & X_0 \arrow[d] \\
\coprod j \in J_1.I^1 \arrow[r, "f"'] & X_1
\end{tikzcd}$$

Note that in X_1 we have lines with endpoints given by the d_j^1: for each $j \in J$ we have the map $f \circ \iota_j$ where $\iota_j : I^1 \to \coprod j \in J_1.I^1$ sends I^1 to the jth copy of I^1. The endpoints of this map are given by d_j^1, because the square commutes. This means that for every $j \in J_1$ there is a line in X_1 whose endpoints are given by d_j^1. So, the construction says that we add lines with endpoints given by the d_j^1.

For X_2 we add squares to X_1 in a similar way. We need to give a collection of squares, and describe their desired 'endpoints'. For each square we thus need to describe what the boundary is mapped to, and that is given by a map $\partial(I^2) \to X_1$. So, we have a collection J_2 and for each $j \in J_2$ a boundary map $d_j^2 : \partial(I^2) \to X_1$, and then X_2 is the following pushout

$$\begin{tikzcd}
\coprod j \in J_2 . \partial(I^2) \arrow[r, "\coprod j \in J_2 . d_j^2"] \arrow[d, hook] & X_1 \arrow[d] \\
\coprod j \in J_2 . I^2 \arrow[r] & X_2
\end{tikzcd}$$

This can be done for every dimension which is captured in the following definition

Definition 4.1.1 (CW-Complex)**.** Suppose, X is a topological space. Then we say X is a *CW-complex* iff we have a sequence of inclusions $X_0 \subseteq X_1 \subseteq X_2 \subseteq \ldots \subseteq X$, sets J_n for each $n \in \mathbb{N}$, and continuous maps $d_j^n : \partial(I^n) \to X_n$ such that

- $X = \bigcup_{n \in \mathbb{N}} X_n$.
- For each n there are maps $I^n \to X_{n+1}$ for each $j \in J_n$ such that the following diagram is a pushout

$$\begin{tikzcd}
\coprod j \in J_n . \partial(I^n) \arrow[r, "\coprod j \in J_n . d_j^n"] \arrow[d, hook] & X_n \arrow[d] \\
\coprod j \in J_n . I^n \arrow[r] & X_{n+1} \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

This means that for each $j \in J_n$ we added a n-cube to X_{n+1} with the endpoints given by the d_j^n. ⌟

4.2. Intervals

The main idea is that higher inductive types are constructed in the same way as CW-complexes in topology. For CW-complexes we start with a set of points, then we glue some intervals, then we glue some squares and so on. Higher inductive types are made by adding equalities to some type. Since $x \rightsquigarrow y$ in a type T corresponds with a map $I^1 \to T$ which sends 0 to x and 1 to y, we see that higher inductive types are made by gluing intervals. Note that for CW-complexes the I^n are added in order of their dimension, but for a higher inductive type we will not require that.

4.2.1. Intervals in Type Theory. The interval types are thus fundamental in our presentation, and we start by defining them. These are particularly simple higher inductive types, and we are able to give their introduction, elimination and computation rules directly. We define I^n inductively, so we start by defining I^0 and then we define I^{n+1} from I^n.

Definition 4.2.1. We define I^0 as the type

```
Inductive I⁰ :=
  | ∗ : I⁰
```

This type has one introduction rule

$$* : I^0.$$

Also, it has one elimination rule

$$\frac{\Gamma, x : I^0 \vdash Y[x] : \textsc{Type} \qquad \Gamma \vdash z : Y[*]}{\Gamma \vdash I^0\text{-elim}(z) : \prod x : I^0 . Y[x]}$$

and a computation rule, namely

$$I^0\text{-elim}(z) * = z. \qquad \lrcorner$$

Recall that I^1 was defined as follows

```
Inductive $I^1$ :=
  | $0 : I^1$
  | $1 : I^1$
  | seg : $0 \rightsquigarrow 1$
```

This type has three introduction rules, namely $0 : I^1$, $1 : I^1$ and $\text{seg} : 0 \rightsquigarrow 1$. Using the notation of Definition 2.3.7 and Proposition 2.3.8, we can formulate the elimination rule of I^1

$$\frac{\Gamma, x : I^1 \vdash Y[x] : \textsc{Type} \qquad \Gamma \vdash z : Y[0] \qquad \Gamma \vdash o : Y[1] \qquad \Gamma \vdash s : z \rightsquigarrow^{Y}_{\text{seg}} o}{\Gamma \vdash I^1\text{-elim}(z, o, s) : \prod x : I^1 . Y[x]}$$

together three computation rules

$$I^1\text{-elim}(z, o, s)\, 0 = z,$$
$$I^1\text{-elim}(z, o, s)\, 1 = o,$$
$$\text{apd}(I^1\text{-elim}(z, o, s), \text{seg}) = s.$$

Note that I^1-elim(z, o, s) is of type $\prod x : I^1 . Y[x]$, so I^1-elim$(z, o, s)\, 0$ is of type $Y[0]$ and I^1-elim$(z, o, s)\, 1$ is of type $Y[1]$. Since $z : Y[0]$ and $o : Y[1]$, the first two computation rules preserve types. Also, if $f : \prod x : A . B[x]$ and $p : x \rightsquigarrow y$ with $x : A$ and $y : A$, then we have $\text{apd}(f, p) : f\, x \rightsquigarrow^{Y}_{\text{seg}} f\, y$. From this and the first two computation rules we can deduce that $\text{apd}(I^1\text{-elim}(z, o, s), \text{seg}) : z \rightsquigarrow^{Y}_{\text{seg}} o$, so the third rule preserves types as well.

To define I^{n+1}, we imitate the definition of I^1.

Definition 4.2.2. We define the type I^{n+1} as follows

```
Inductive $I^{n+1}$ :=
  | $0 : I^n \to I^{n+1}$
  | $1 : I^n \to I^{n+1}$
  | seg : $\prod x : I^n . 0\, x \rightsquigarrow 1\, x$
```

The elimination rule is $\Gamma, x : I^{n+1} \vdash Y[x] : \textsc{Type}$ where the terms z, o and s depend on $x : I^n$.

$$\frac{\Gamma, x : I^n \vdash z : Y[0\, x] \qquad \Gamma, x : I^n \vdash o : Y[1\, x] \qquad \Gamma, x : I^n \vdash s : z[x] \rightsquigarrow^{Y}_{\text{seg}\, x} o[x]}{\Gamma \vdash I^{n+1}\text{-elim}(z, o, s) : \prod x : I^{n+1} . Y[x]}$$

and we have three computation rules for a term $t : I^n$

$$I^{n+1}\text{-elim}(z, o, s)(0\, t) = z[t],$$
$$I^{n+1}\text{-elim}(z, o, s)(1\, t) = o[t],$$
$$\text{apd}(I^{n+1}\text{-elim}(z, o, s), \text{seg}\ t) = s[t]. \qquad \lrcorner$$

Again we have that I^{n+1}-elim(z, o, s) is of type $\prod x : I^{n+1} . Y[x]$, and thus for all $t : I^n$ the terms I^{n+1}-elim$(z, o, s)\,(0\, t)$ and I^{n+1}-elim$(z, o, s)\,(1\, t)$ are of types $Y[0\, t]$ and $Y[1\, t]$ respectively. For all $t : I^n$ we have $z[t] : Y[0\, t]$ and $o[t] : Y[1\, t]$, so the first two rules preserve types. Since $\text{apd}(I^{n+1}\text{-elim}(z, o, s), \text{seg}\ t)$ and $s\, t$ are of types $z[t] \rightsquigarrow^{Y}_{\text{seg}\ t} o[t]$ and $z[t] \rightsquigarrow^{Y}_{\text{seg}\ t} o[t]$ respectively, we can conclude using the first two rules that the third rule preserves types as well.

4.2.2. Interpretation of Intervals. Our goal is to define when a Martin-Löf category $\mathscr{C}$ with all inductive types has an interpretation of all interval types. For this we first define inductively how we interpret I^n for all n. Note that I^0 has an interpretation $[\![I^0]\!]$, because I^0 is an inductive type. Also, for $x : A$ and $y : A$ we can find an interpretation $[\![x \rightsquigarrow y]\!]$ of $x \rightsquigarrow y$. A map $h : A \to B$ gives a map $[\![\mathrm{ap}(h, -)]\!] : [\![x \rightsquigarrow y]\!] \to [\![h\,x \rightsquigarrow h\,y]\!]$. This is because $x \rightsquigarrow y$ is an inductive type which always have an interpretation in $\mathscr{C}$.

For the inductive step we assume that we have an interpretation $[\![I^n]\!]$ of I^n, and our goal is to make an interpretation $[\![I^{n+1}]\!]$ of I^{n+1}. So, suppose that we have an object $[\![I^n]\!]$ which satisfies all the rules of the type I^n. To make an interpretation of I^{n+1} we need maps $\iota_1, \iota_1 : [\![I^n]\!] \to [\![I^{n+1}]\!]$, because these give the introduction rules. For the elimination and computation rules, we require the commutativity and existence of maps, and this is given in the following definition.

Definition 4.2.3. Suppose, $\mathscr{C}$ is a Martin-Löf category, and let n be a natural number greater than 0. Then we say that $\mathscr{C}$ has an *interpretation of* I^{n+1} iff

- $\mathscr{C}$ has an interpretation $[\![I^n]\!]$ of I^n.
- $\mathscr{C}$ has an object $[\![I^{n+1}]\!]$ with maps $\iota_0, \iota_1 : [\![I^n]\!] \to [\![I^{n+1}]\!]$, and $s : 1 \to [\![\iota_0 \rightsquigarrow \iota_1]\!]$.
- For each $f, g : [\![I^n]\!] \to Y$ and $p : [\![f \rightsquigarrow g]\!]$ there is a map $h : [\![I^{n+1}]\!] \to Y$ such that the following two diagrams commute

$$\begin{tikzcd}
{[\![I^n]\!] \coprod [\![I^n]\!]} \arrow[r, "f \coprod g"] \arrow[dr, "\iota_0 \coprod \iota_1"'] & Y \\
& {[\![I^{n+1}]\!]} \arrow[u, "h"']
\end{tikzcd}$$

$$\begin{tikzcd}
1 \arrow[r, "s"] \arrow[dr, "p"'] & {[\![\iota_0 \rightsquigarrow \iota_1]\!]} \arrow[d, "{[\![\mathrm{ap}(h,-)]\!]}"] \\
& {[\![f \rightsquigarrow g]\!]}
\end{tikzcd}$$

Note that we have an interpretation of I^0, because it is an inductive type. ⌟

Now we can say when all intervals are interpreted in some Martin-Löf category.

Definition 4.2.4. Suppose, $\mathscr{C}$ is a Martin-Löf category that interprets every inductive type. Then we say that $\mathscr{C}$ *interprets every interval* iff it has an interpretation of I^n for every $n \geq 1$ as in the preceding definition. ⌟

4.3. Basic Introduction of Higher Inductive Types

Higher inductive types are interpreted by gluing intervals, and our next step is to give a formal definition. However, before diving into formalities, let us first discuss the basic idea and the challenges. To define a higher inductive type T, one first needs to give an inductive type using constructors $c : H(T) \to T$. Then we add paths of types like $\prod x : A \prod y : B.f\,x \rightsquigarrow g\,y$ which says that for every $x : A$ and $y : B$ we can construct a path $p(x, y)$ between $f\,x$ and $g\,y$. Here A and B are types, and $f\,x$ and $g\,x$ are terms of type T. Since we will allow parameters in our type definitions, A and B can depend on the parameters.

However, what is allowed in these rules? There are several issues, and the first one is recursion. If we are making a type T, then we might want to add the rule $p : \prod x : T \prod y : T.x \rightsquigarrow y$ Types with such rules will be called *recursive higher inductive types*. An example of such a type is the truncation

Inductive $||A|| \ (A : \text{TYPE}) :=$
  $| \ \iota : A \to ||A||$
  $| \ p : \prod x : ||A|| \prod y : ||A||.x \rightsquigarrow y$

This definition says that for every $x : ||A||$ and $y : ||A||$ we have a path $x \rightsquigarrow y$. However, gluing an interval might add new points in some interpretations, and this gives the first problem. We need to make sure that for every $x : ||A||$ and $y : ||A||$ we have a path $x \rightsquigarrow y$, and if we just add the intervals between the points of A, then we are not sure about the new points.

Higher inductive types without recursion are called *nonrecursive higher inductive types*. Even for these types there are complications, and these come from the constructors. To understand why, we consider the following example.

Inductive $\mathbb{N}/2\mathbb{N} :=$
  $| \ 0 : \mathbb{N}/2\mathbb{N}$
  $| \ S : \mathbb{N}/2\mathbb{N} \to \mathbb{N}/2\mathbb{N}$
  $| \ p : 0 \rightsquigarrow S\,(S\,0)$

With just the first two constructors we get the type $\mathbb{N}$ of the natural numbers. The third rule says that we get a path p between 0 and $S\,(S\,0)$. Since we can apply functions on paths via ap, we get a path $\text{ap}(S, p)$ between $S\,0$ and $S\,(S\,(S\,0))$. So, for the interpretation we need to take that in consideration, because otherwise we do not have enough paths. Furthermore, after all these paths are added, there still needs to be a map $S : \mathbb{N}/2\mathbb{N}$. In the category of topological spaces, adding a path adds new points, and somehow we need to determine where these are mapped to.

We will deal with these problems step by step. We start by giving higher inductive types where the only inductive constructors are point constructors. Furthermore, recursion in the paths will not be allowed. Next we allow more inductive constructors. Finally, we give a definition for recursive higher inductive types. Also, we give an interpretation of all nonrecursive higher inductive types, but not for recursive higher inductive types.

Let us give some intuition about the interpretation. We work in a Martin-Löf category $\mathscr{C}$ which interprets all inductive types and all intervals, and we will denote the interpretation of I^n by I^n to simplify the notation. To interpret a higher inductive type, we start with interpreting an inductive type, and then we need to add equalities. Since paths between n-paths correspond with maps $I^{n+1} \to T$, adding a path is the same as adding a map $I^n \to T$. So, to add an equality between two maps $x, y : I^n \to T$, we consider the following pushout

$$\begin{tikzcd}
I^n \coprod I^n \arrow[r, "x \coprod y"] \arrow[d, "\iota_0 \coprod \iota_1"'] & T \arrow[d, "h"] \\
I^{n+1} \arrow[r, "e"'] & P \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

Recall that a map $I^0 \to T$ corresponds with an inhabitant of T, and that a map $I^{n+1} \to T$ corresponds with an equality between two maps $I^n \to T$. Note that we have $T \to P$, so every constructor of T is also a constructor of P. Also, since we have a map $e : I^{n+1} \to P$ with $e \circ \iota_0 = h(x)$ and $e \circ \iota_1 = h(y)$, we have an equality between $h(x)$ and $h(y)$ in P, so P has the required constructors.

For higher inductive types the introduction rules are not difficult, but the elimination rule is. However, using this intuition we will give an elimination rule. The universal property of the pushout can be used to give an elimination rule for P, because a map $P \to Y$ corresponds with two maps $f : I^{n+1} \to Y$ and $g : X \to Y$ such that the following diagram commutes

$$\begin{tikzcd}
I^n \coprod I^n \arrow[r, "x \coprod y"] \arrow[d, "\iota_0 \coprod \iota_1"'] & X \arrow[d] \arrow[ddr, "g"] & \\
I^{n+1} \arrow[r] \arrow[drr, "f"'] & P \arrow[ul, phantom, "\ulcorner", very near start] \arrow[dr] & \\
 & & Y
\end{tikzcd}$$

4.4. Higher Inductive Types from Points

In this section we give a first kind of higher inductive types. These do not allow recursive, and in 0-constructors we only allow point constructors. For the definition we need the following notation.

Definition 4.4.1. Let T be a type and let $x_1 : A_1, \dots, x_n : A_n$ be variables. We define $T(x_1, \dots, x_n)$ to be the collection of terms with type T using $x_1, \dots x_n$ as free variables. So, to make a term t from $T(x_1, \dots, x_n)$, we need to prove the judgement

$$x_1 : A_1, \dots, x_n : A_n \vdash t : T. \qquad \lrcorner$$

Now we formally define the syntax of these higher inductive types, and give their introduction and elimination rule.

Definition 4.4.2 (Higher Inductive Type from Points). A *higher inductive type from points* is defined according to the following syntax

```
Inductive $T$ $(B_1 : \mathrm{TYPE}) \dots (B_\ell : \mathrm{TYPE}) :=$
  | $*_1 : T$
  ...
  | $*_k : T$
  | $p_1 : \prod x : A_1.F_1[x, *_1, \dots, *_k] \rightsquigarrow G_1[x, *_1, \dots, *_k]$
  ...
  | $p_n : \prod x : A_n.F_n[x, *_1, \dots, *_k, p_1, \dots, p_{n-1}] \rightsquigarrow G_n[x, *_1, \dots, *_k, p_1, \dots, p_{n-1}]$
```

Let us write $\overline{F_i} = F_i[x, *_1, \dots, *_k, p_1, \dots, p_{i-1}]$ and $\overline{G_i} = G_i[x, *_1, \dots, *_k, p_1, \dots, p_{i-1}]$. Also, we assume that we have types $B_1, \dots, B_\ell : \mathrm{TYPE}$, and we denote $T(B_1, \dots, B_\ell)$ by T. We require that $\overline{F_i}$ and $\overline{G_i}$ are terms in $(I^{d_i} \to T)(x, *_1, \dots, *_k, p_1, \dots, p_{i-1})$ with variables $x : A_i$, $*_j : T$ and

$$p_j : \prod x : A_j.F_j[x, *_1, \dots, *_k, p_1, \dots, p_{j-1}] \rightsquigarrow G_j[x, *_1, \dots, *_k, p_1 \dots, p_{j-1}].$$

The fact that $\overline{F_i}$ and $\overline{G_i}$ are terms in $(I^{d_i} \to T)(x, *_1, \dots, *_k, p_1, \dots, p_{i-1})$, means that we have the judgements with $x : A_i$

$$*_1 : T, \dots, *_k : T, p_1 : \prod y : A_1.\overline{F_1} \rightsquigarrow \overline{G_1}, \dots, p_{i-1} \prod y : A_{i-1}.\overline{F_{i-1}} \rightsquigarrow \overline{G_{i-1}} \vdash \overline{F_i} : I^{d_i} \to T,$$

$*_1 : T, \dots, *_k : T, p_1 : \prod y : A_1.\overline{F_1} \rightsquigarrow \overline{G_1}, \dots, p_{i-1} \prod y : A_{i-1}.\overline{F_{i-1}} \rightsquigarrow \overline{G_{i-1}} \vdash \overline{G_i} : I^{d_i} \to T,$

Furthermore, we require that A_i is any type depending on $B_1, \dots B_\ell$ in Martin-Löf type theory with higher inductive types.

The introduction rules for the points are

$$\Gamma \vdash *_i : T,$$

and the introduction rules for the paths are

$$\vdash p_i : \prod x : A_i(B_1, \dots, B_\ell).\overline{F_i} \rightsquigarrow \overline{G_i}.$$

Given $\Gamma, x : T \vdash Y[x] : \textsc{Type}$, the elimination rule is

$$\frac{\Gamma \vdash z_i : Y[*_i] \text{ for } i = 1, \dots, k \qquad \Gamma \vdash q_i : \prod x : A_i.F_i' \rightsquigarrow^{Y}_{p_i x} G_i' \text{ for } i = 1, \dots, n}{\Gamma \vdash T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n) : \prod x : T.Y[x]}$$

where $F_i' = F_i[x, z_1, \dots, z_k, q_1, \dots, q_{i-1}]$ and $G_i' = G_i[x, z_1, \dots, z_k, q_1, \dots, q_{i-1}]$. Finally, the computation rules are

$$T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n) *_i = z_i$$

and for $t : A_i$

$$\text{apd}(T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n), p_i\, t) = q_i\, t. \qquad \lrcorner$$

Note that the computation rules preserve types.

Lemma 4.4.3. *The computations rules preserve types.*

Proof. Since $T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n) : \prod x : T.Y[x]$, we have that both z_i and $T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n) *_i$ are of type $Y[*_i]$. This means that the first computation rule preserves types. Furthermore, using the first computation rule we see that $\text{apd}(T\text{-elim}(z_1, \dots, z_k, q_1, \dots, q_n), p_i\, t)$ is of type $F_i' \rightsquigarrow^{Y}_{p_i t} G_i'$, and thus the last rule preserves types as well. □

4.4.1. Interpretation. To give an interpretation of higher inductive types from points, we use pushouts. This means that we want to prove the following theorem

Theorem 4.4.4. *If $\mathscr{C}$ is a Martin-Löf category in which we can interpret all intervals, then we can interpret higher inductive types from points in $\mathscr{C}$.*

Proof. Basically, our approach is as follows. We make approximations of the type T step by step. In every step we add a path, and eventually we have everything. The interpretation is thus given as a sequence of pushouts, one for each path.

Let $\mathscr{C}$ be a Martin-Löf category with an interpretation of every I^n as discussed in Section 4.2.2, and we denote this interpretation by I^n. The first approximation will be T_0 defined as $\coprod_{i=1}^{k} I^0$, and note that for every i we have a map $*_i : \top \to T_0$ which is the ith inclusion. Also, a map $T_0 \to Y$ corresponds with maps $z_i : \top \to Y$ for $i = 1, \dots, k$.

To continue, we make T_i for $i = 1, \dots, n$ such that in T_i we have interpretations of $*_1, \dots, *_k, p_1, \dots, p_{i-1}$ and T_i satisfies the right elimination rule and computation rules. These rules for T_i is the elimination rule and the computation rules of the higher inductive type generated from $*_1, \dots, *_k, p_1, \dots, p_{i-1}$. We use induction, so assume that we have constructed T_i for $i = 1, \dots, n$, in which we can interpret $*_1, \dots, *_k, p_1, \dots, p_{i-1}$. Recall that p_i has type $F_i[x, *_1, \dots, *_k, p_1, \dots, p_{i-1}] \rightsquigarrow G_i[x, *_1, \dots, *_k, p_1 \dots, p_{i-1}]$, and that $\overline{F_i}$ and $\overline{G_i}$ are terms of type $I^{d_i} \to T$ using $x, *_1, \dots, *_k, p_1, \dots, p_{i-1}$ as free variables.

Since we have interpretations of $*_1, \dots, *_k$, $p_1, \dots, p_{i-1}$, we can interpret $\overline{F_i}$ and $\overline{G_i}$ as maps $f_i, g_i : \prod x : A_i . I^{d_i} \to T_i$. Now we construct T_{i+1} as the following pushout

$$\begin{tikzcd}
\prod x : A_i . I^{d_i} \coprod I^{d_i} \arrow[r, "f_i \coprod g_i"] \arrow[d] & T_i \arrow[d] \\
\prod x : A_i I^{d_i+1} \arrow[r, "p_i"'] & T_{i+1} \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

Note that for $j = 1, \dots, i-1$ we have a map $p_{i-1} : \prod x : A_i . I^{d_j+1} \to T_i \to T_{i+1}$, and for $j = 1, \dots, k$ we have a map $*_j : \top \to T_i \to T_{i+1}$, and thus T_{i+1} has the desired constructors. To make a map from T_{i+1} to some object Y, we need a map $T_i \to Y$ and a map $q_i : \prod x : A_i I^{d_i+1} \to Y$ such that the following diagram commutes

$$\begin{tikzcd}
\prod x : A_i . I^{d_i} \coprod I^{d_i} \arrow[r, "f_i \coprod g_i"] \arrow[d] & T_i \arrow[d] \\
\prod x : A_i I^{d_i+1} \arrow[r, "p_i"'] & Y
\end{tikzcd}$$

To make a map $T_i \to Y$, we can use the elimination rule of T_i. In addition, we need to give $I^{d_i+1} \to Y$, and the requirement that the diagram commutes gives the right requirement on the endpoints. Hence, we get a map $T_{i+1} \to Y$ which interpret the elimination rule. The computation rules can easily be checked by diagram chasing, and we show the right diagram for $*_j$

$$\begin{tikzcd}
 & \top \arrow[d, "*_j"] & \\
 & T_0 \arrow[d] \arrow[ddddr, "z_j"] & \\
 & \cdots \arrow[d] & \\
\prod x : A_i . I^{d_i} \coprod I^{d_i} \arrow[r, "f_i \coprod g_i"] \arrow[d] & T_i \arrow[d] \arrow[ddr] & \\
\prod x : A_i . I^{d_i+1} \arrow[r, "p_i"'] \arrow[drr] & T_{i+1} \arrow[dr] & \\
 & & Y
\end{tikzcd}$$

Using the universal property of the coproduct and pushout one can show that all triangles commutes, and this gives the right computation rule for $*_j$. For the paths the same argument can be given.

All in all, the type T_n satisfies the right rules, and therefore it is an interpretation of the higher inductive type T. □

4.4.2. Examples. There are several higher inductive types which can be defined this way. We discuss the circle and the torus, but one can consider other CW-complexes with a finite presentation as well.

Example 4.4.5 (Circle)**.** Let us start with the circle.

Inductive $S^1 :=$
 | base : S^1
 | loop : base $\rightsquigarrow$ base

We can then deduce the following elimination rule

$$\frac{\Gamma, x : S^1 \vdash Y[x] : \text{Type} \qquad \Gamma \vdash z : Y[\text{base}] \qquad \Gamma \vdash q : z \rightsquigarrow z}{\Gamma \vdash S^1\text{-elim}(z, q) : \prod x : T.Y[x]}$$

and the computation rules say that

$$S^1\text{-elim}(z, q)(\text{base}) = z,$$
$$\text{apd}(S^1\text{-elim}(z, q), \text{loop}) = q.$$

Note that these rules are the same as those in [**Uni13**].

Example 4.4.6 (Torus)**.** Another example is the torus $\mathbb{T}$.

Inductive $\mathbb{T} :=$
 | base : $\mathbb{T}$
 | p : base $\rightsquigarrow$ base
 | q : base $\rightsquigarrow$ base
 | s : Irec(base, base, $p \circ q$) $\rightsquigarrow$ Irec(base, base, $q \circ p$)

Again we can deduce the elimination rule given that we have $\Gamma, x : \mathbb{T} \vdash Y[x] : \text{Type}$

$$\frac{\Gamma \vdash z : Y[\text{base}] \qquad \Gamma \vdash p' : z \rightsquigarrow z \qquad \Gamma \vdash q' : z \rightsquigarrow z \qquad \Gamma \vdash s' : p' \circ q' \rightsquigarrow^{Y}_{s} q' \circ p'}{\Gamma \vdash \mathbb{T}\text{-elim}(z, p', q', s') : \prod x : T.Y[x]}$$

and the computation rules are

$$\mathbb{T}\text{-elim}(z, p', q', s')(\text{base}) = z,$$
$$\text{apd}(\mathbb{T}\text{-elim}(z, p', q', s')(\text{base}), p) = p',$$
$$\text{apd}(\mathbb{T}\text{-elim}(z, p', q', s')(\text{base}), q) = q',$$
$$\text{apd}(\mathbb{T}\text{-elim}(z, p', q', s')(\text{base}), s) = s'.$$

Again these rules are the same as those in [**Uni13**].

Interpreting S^1 gives the circle. It glues the endpoints of a line I^1 to some point which indeed gives the circle. Also, the type $\mathbb{T}$ gives the torus. Note that s gives a path from $p \circ q \circ p^{-1} \circ q^{-1}$ which is what is needed for the torus.

Example 4.4.7 (Cylinder)**.** Lastly, let us briefly show a higher inductive type which shows the advantage of working with intervals

Inductive Cyl :=
| base_1 : Cyl
| base_2 : Cyl
| p : $\text{base}_1 \rightsquigarrow \text{base}_1$
| q : $\text{base}_2 \rightsquigarrow \text{base}_2$
| s : Irec($\text{base}_1, \text{base}_1, p$) $\rightsquigarrow$ Irec($\text{base}_2, \text{base}_2, q$)

We cannot add an equality between p and q, because they have different types, but this way we can make this a cylinder.

4.5. Nonrecursive Higher Inductive Types

Recall the type

Inductive $\mathbb{N}/2\mathbb{N} :=$
  $\mid\ 0 : \mathbb{N}/2\mathbb{N}$
  $\mid\ S : \mathbb{N}/2\mathbb{N} \to \mathbb{N}/2\mathbb{N}$
  $\mid\ p : 0 \rightsquigarrow S\,(S\,0)$

This type is not allowed in the syntax of Definition 4.4.2, and we cannot interpret it in the same way as those types. The problem is that this would only add an equality between 0 and $S\,(S\,0)$, but not between $S(0)$ and $S\,(S\,(S\,0))$. To solve this problem we need to add more equalities in the interpretation. Also, after adding all the paths, there still should be a function $S : \mathbb{N}/2\mathbb{N} \to \mathbb{N}/2\mathbb{N}$. Thus, our goal is that S gets interpreted as a function in the construction, and that there are enough paths.

Recall that a polynomial functor is built from constant functors, the identity, products and coproducts. The constructors of inductive types are given using polynomial functors. By Theorem 3.2.17 all polynomial functors have an initial algebra which is the interpretation of the inductive type. We only need to extend this by giving the path constructors. Using the notation of Definition 2.2.2 we define

Definition 4.5.1 (Nonrecursive Higher Inductive Type)**.** A *nonrecursive higher inductive type* is given by the following syntax

Inductive $T\ (B_1 : \text{Type}) \ldots (B_\ell : \text{Type}) :=$
  $\mid\ c_1 : H_1(T) \to T$
  $\ldots$
  $\mid\ c_k : H_k(T) \to T$
  $\mid\ p_1 : \prod x : A_1.F_1[x, c_1, \ldots, c_k] \rightsquigarrow G_1[x, c_1, \ldots, c_k]$
  $\ldots$
  $\mid\ p_n : \prod x : A_n.F_n[x, c_1, \ldots, c_k, p_1, \ldots, p_{n-1}] \rightsquigarrow G_n[x, c_1, \ldots, c_k, p_1, \ldots, p_{n-1}]$

Again we write $\overline{F_i} = F_i[x, c_1, \ldots, c_k, p_1, \ldots, p_{i-1}]$ and $\overline{G_i} = G_i[x, c_1, \ldots, c_k, p_1, \ldots, p_{i-1}]$, and again we require that $\overline{F_i}$ and $\overline{G_i}$ are terms in $(I^{d_i} \to T)(x, c_1, \ldots, c_k, p_1, \ldots, p_{i-1})$ with variables $x : A_i$, $c_j : H_j(T) \to T$ and

$$p_j : \prod x : A_j.F_j[x, c_1, \ldots, c_k, p_1, \ldots, p_{j-1}] \rightsquigarrow G_j[x, c_1, \ldots, c_k, p_1 \ldots, p_{j-1}]$$

using the notation of Definition 4.4.1. Furthermore, we require that A_i is any type depending on $B_1, \ldots B_\ell$ in Martin-Löf type theory (higher inductive types are allowed), and that every H_i is polynomial.

We denote $T(B_1, \ldots, B_\ell)$ by T. The introduction rules for the points are

$$\frac{\Gamma \vdash x : H_i(T)}{\Gamma \vdash c_i\, x : T}$$

and the introduction rules for the paths are

$$\vdash p_i : \prod x : A_i(B_1, \ldots, B_\ell).\overline{F_i} \rightsquigarrow \overline{G_i}.$$

Also, the elimination rule is similar to the one in Definition 4.4.2, and we need to have a family of types Y satisfying $\Gamma, x : T \vdash Y[x] : \text{Type}$.

$$\frac{\begin{array}{cc} \text{For } i = 1, \ldots, k & \text{For } i = 1, \ldots, n \\ \Gamma \vdash z_i : \prod x : H_i(T).\overline{H_i}(Y)[x] \to Y[c_i(x)] & \Gamma \vdash q_i : \prod x : A_i.F_i' \rightsquigarrow^{Y}_{p_i\, x} G_i' \end{array}}{\Gamma \vdash T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n) : \prod x : T.Y[x]}$$

where $F_i' = F_i[x, z_1, \ldots, z_k, q_1, \ldots, q_{i-1}]$ and $G_i' = G_i[x, z_1, \ldots, z_k, q_1, \ldots, q_{i-1}]$. Lastly, we have similar computation rules for $t : H_i(T)$

$$T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)(c_i\, t) = (z_i\, t)(H_i(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n))\, t)$$

and for all $t : A_i$

$$\text{apd}(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n), p_i\, t) = q_i\, t. \qquad \lrcorner$$

Again we would like that the computation rules preserve types.

Lemma 4.5.2. *The computation rules preserve types.*

PROOF. We start with the rule

$$T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)(c_i\, x) = z_i\, (H_i(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n))(x))$$

Note that $T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)(c_i\, t)$ has type $Y(c_i\, t)$. On the other hand, the term $T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)$ has type $\prod x : T.Y(x)$. Because H_i is a polynomial functor, we have a function

$$H_i(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)) : \prod x : H_i(T).\overline{H_i}(Y)(x).$$

So, for all $t : H_i(T)$ the term $H_i(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)\, t$ has type $H_i(Y(t))$. Since z_i has type $\prod x : H_i(T).\overline{H_i}(Y)(x) \to Y(c_i(x))$, we see that $z_i\, t$ has type $\overline{H_i}(Y)(x) \to Y(c_i(x))$. Also, note that $T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)(c_i(x))$ is of type $Y(c_i(x))$, and thus this rule is typed correctly. All in all, we can conclude that the term

$$(z_i\, t)(H_i(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n))\, t)$$

has type $Y(c_i\, x)$, and thus this computation rule preserves types.

For $t : A_i$ we check the rule

$$\text{apd}(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n), p_i\, t) = q_i\, t.$$

Since q_i has type $\prod x : A_i.F_i' \rightsquigarrow^Y_{p_i\, x} G_i'$, we see that $q_i\, t$ has type $F_i' \rightsquigarrow^Y_{p_i\, t} G_i'$. On the other hand, since p_i has type $\prod x : A_i(B_1, \ldots, B_\ell).\overline{F_i} \rightsquigarrow \overline{G_i}$, the term $p_i\, t$ must have type $\overline{F_i} \rightsquigarrow \overline{G_i}$. By the computation rules we have that $T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)\,\overline{F_i} = F_i'$ and $T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n)\,\overline{G_i} = G_i'$. Using this and Proposition 2.3.8 we conclude that $\text{apd}(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n), p_i\, t)$ has type $F_i' \rightsquigarrow^Y_{p_i\, x} G_i'$. Hence, the types of $\text{apd}(T\text{-elim}(z_1, \ldots, z_k, q_1, \ldots, q_n), p_i\, t)$ and $q_i\, t$ coincide, and thus this rules preserve types as well. □

4.5.1. Interpretation. Our goal is to give an interpretation of the type

```
Inductive $T$ $(B_1 : \text{TYPE}) \ldots (B_\ell : \text{TYPE})$ :=
  | $c_1 : H_1(T) \to T$
  ...
  | $c_k : H_k(T) \to T$
  | $p_1 : \prod x : A_1.F_1[x, *_1, \ldots, *_m] \rightsquigarrow G_1[x, *_1, \ldots, *_m]$
  ...
  | $p_n : \prod x : A_n.F_n[x, *_1, \ldots, *_m, p_1, \ldots, p_{n-1}] \rightsquigarrow G_n[x, *_1, \ldots, *_m, p_1, \ldots, p_{n-1}]$
```

More specifically, we want to prove the following theorem

Theorem 4.5.3. *If $\mathscr{C}$ is a Martin-Löf category in which we can interpret all intervals, then we can interpret higher inductive types from points in $\mathscr{C}$. We can give interpretations of the introduction rule, computation rules and the nondependent elimination rule.*

This theorem is more difficult than Theorem 4.4.4, because we also need to add equalities $\mathrm{ap}(c_i, p_j)$. This can be solved by gluing more equalities to the inductive type.

More specifically, we start by interpreting just the constructors $c_1, \ldots, c_k$, and that can be done by interpreting inductive types. Now we can glue the paths $p_1, \ldots, p_n$ to the type, and this gives the first approximation T_0. To construct T_1, we add for every path p in T_0 a new path $\mathrm{ap}(c_i, p)$ which is done using a pushout. Several equalities should hold, namely $\mathrm{ap}(c_i, \mathrm{refl}) = \mathrm{refl}$, and using a coequalizer we get T_1.

Now we construct T_2 from T_1, and this is done in the same way. However, there is one issue. If we have a path p in T_0, then we get $\mathrm{ap}(c_i, p)$ twice in T_2. We already have this path, and we add $\mathrm{ap}(c_i, p)$ for every path p in T_1. Since p is also a path in T_0, we thus get $\mathrm{ap}(c_i, p)$ again, and these two need to be identified. Hence, we can construct T_2 from T_1 in the same way, but we need to take one extra coequalizer.

All in all, the construction is basically as follows. We start by giving an inductive type, and to that type we add the paths given in the definition. Then we approximate T step by step, and in every step we add new paths $\mathrm{ap}(c_i, p)$ where p is any path in the previous approximation. However, there might be duplicates and we need the ensure that $\mathrm{ap}(c_i, \mathrm{refl}) = \mathrm{refl}$, and thus we need to take some coequalizers in the construction. This gives $T_0 \to T_1 \to \ldots$, and we define T as the colimit of all the T_i.

PROOF. Let us start by defining the type T_{-1}:

```
Inductive $T_{-1}$ ($B_1$ : TYPE)...($B_\ell$ : TYPE) :=
  | $c_1 : H_1(T) \to T_{-1}$
  ...
  | $c_k : H_k(T) \to T_{-1}$
```

This type can be interpreted in $\mathscr{C}$ as an object T_{-1}.

Next we define T_0 from T_{-1}, and we add all the paths $p_1, \ldots, p_n$. This is done in the same way as in the previous section. So, we give a chain $T_0^0 \to T_0^1 \to \ldots \to T_0^n$ of arrows where $T_0^0 = T_{-1}$ and T_0^{i+1} is

$$\begin{tikzcd}
\prod x : A_i . I^{d_i} \coprod I^{d_i} \arrow[r, "f_i \coprod g_i"] \arrow[d] & T_0^i \arrow[d] \\
\prod x : A_i . I^{d_i+1} \arrow[r, "p_i"'] & T_0^{i+1} \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

Here f_i and g_i are the interpretations of $\overline{F_i}$ and $\overline{G_i}$ respectively.

Before we describe T_i let us explain why constructions will give the right constructors. For the inductive construction we will need that we always have a map $H_i(T_j) \to T_{j+1}$. If we have such a map and we define $T = \mathrm{colim}_{j \in \mathbb{N}} T_j$, then we get a map $H_i(T) \to T$. This is because a map $H_i(T) \to T$ is the same as a map $\mathrm{colim}_{j \in \mathbb{N}} H_i(T_j) \to T$, and thus it suffices to give maps $H_i(T_j) \to T$ for all j. Hence, if we have maps $H_i(T_j) \to T_{j+1}$, then we get a map $c_i : H_i(T) \to T$, and this gives the interpretation of the constructors.

Also, note that we have an interpretation of the paths p_i. This is because an interpretation of the path p_i is a given as a map $\prod x : A_i . I^{d_i+1} \to T$. We have a map $\prod x : A_i . I^{d_i+1} \to T_0$ with the right endpoints by construction. Since we have an inclusion map $T_0 \to T$, we thus get a map $\prod x : A_i . I^{d_i+1} \to T$ by composition. Therefore, T will have the right constructors.

Now let us describe the construction in more detail. Suppose, we have a constructed $T_1, \ldots, T_j$ such that we have maps $c_j^i : H_j(T_l) \to T_{l+1}$ for $l = 1, \ldots, i-1$, we have inclusions $\iota_i : T_{i-1} \to T_i$, and we have a maps $\prod_{j=1}^k \prod n : \mathbb{N} \prod p : I^{n+1} \to H_j(T_{l-1}).I^{n+1} \to T_l$ for $l = 1, \ldots, i-1$. We want to construct T_{i+1} such that we have a map $T_i \to T_{i+1}$ and maps $H_j(T_i) \to T_{i+1}$. This is done in several steps.

First we construct an object S such that we have maps $H_j(T_i) \to S_{i+1}$ for every j, and we define S as the following pushout

$$\begin{tikzcd}
\prod_{j=1}^k H_j(T_{i-1}) \arrow[r, "\prod_{j=1}^k H_j(\iota_i)"] \arrow[d, "\prod_{j=1}^k c_j^i"'] & \prod_{j=1}^k H_j(T_i) \arrow[d, "\prod_{j=1}^k c_j^{i+1}"] \\
\prod_{j=1}^k T_i \arrow[r, "\iota_{i+1}"'] & S \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

This gives maps $c_j^{i+1} : H_j(T_i) \to S$ such that for every inhabitant $x : H_j(T_{i-1})$ we have $c_j^{i+1}(\iota_i(x)) = \iota_{i+1}(c_j^i(x))$.

In the next step we define an object U such that we have an arrow $S \to U$, and to this object U we add $\mathrm{ap}(c_j, p)$ for paths p. We define U as the pushout

$$\begin{tikzcd}
\sum_{j=1}^k \sum n : \mathbb{N} \sum (p : I^{n+1} \to H_j(S)).I^n \coprod I^n \arrow[r, "f \coprod g"] \arrow[d, "\iota_0 \coprod \iota_1"'] & S \arrow[d] \\
\sum_{j=1}^k \sum n : \mathbb{N} \sum (p : I^{n+1} \to H_j(S)).I^{n+1} \arrow[r] & U \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

where $f : I^n \to S$ and $g : I^n \to S$ are the maps $c_j^{i+1} \circ p \circ \iota_0$ and $c_j^{i+1} \circ p \circ \iota_1$ respectively where $n : \mathbb{N}$ and $p : I^{n+1} \to H_{m+1}(S)$ and $j = 1, \ldots, k$. Note that the arrow $\sum_{j=1}^k \sum n : \mathbb{N} \sum p : (I^{n+1} \to H_j(S)).I^{n+1} \to U$ gives for every j and $p : I^{n+1} \to H_j(S)$ a path $\mathrm{ap}(c_j^{i+1}, p) : I^{n+1} \to U$. Hence, in U we added the desired paths.

Now we need to make two extra identifications. The first one says that $\mathrm{ap}(c_j^{i+1}, \mathrm{refl}) = \mathrm{refl}$, and for this we use coequalizers. Note that for every $n : \mathbb{N}$ and every j there is a path $\mathrm{ap}(c_j, \mathrm{refl})$ by construction, and we define V to be the following coequalizer.

$$\begin{tikzcd}
\prod_{j=1}^k I^{n+1} \arrow[r, shift left, "\mathrm{ap}(c_j,\mathrm{refl})"] \arrow[r, shift right, "\mathrm{refl}"'] & U \arrow[r] & V
\end{tikzcd}$$

We always have a map $\mathrm{refl} : I^{n+1} \to U$, and thus this coequalizer makes sense. It identifies $\mathrm{ap}(c_j, \mathrm{refl})$ with refl which is what we want.

The second identification says that $\mathrm{ap}(c_j^{i+1}, p)$ and $\mathrm{ap}(c_j^i, p)$ are the same for all paths p. Note that we have maps $T_{i-1} \to T_i \to S$, and this gives maps $h : H_j(T_{i-1}) \to H_j(T_i) \to H_j(S)$ for all j. Define the map $f : \prod_{j=1}^k \prod n : \mathbb{N} \prod p : I^{n+1} \to H_j(T_{i-1}).U$ as $\lambda j \lambda n \lambda p.\,\mathrm{ap}(c_j^{i+1}, h \circ p)$. Recall that the induction hypothesis says that we have a map $\prod_{j=1}^k \prod n : \mathbb{N} \prod p : I^{n+1} \to H_j(T_{l-1}).I^{n+1} \to T_l$ for $l = 1, \ldots, i-1$. Define $g : \prod_{j=1}^k \prod n : \mathbb{N} \prod p : I^{n+1} \to H_j(T_{i+1}).U$ to be the map $\lambda j \lambda n \lambda p.\iota \circ \mathrm{ap}(c_j^i, p)$ where ι is the map $T_i \to S \to U$. Again we take a coequalizer

$$\begin{tikzcd}\prod_{j=1}^{k} I^{n+1} \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & V \arrow[r] & T_{i+1}\end{tikzcd}$$

Note that T_1 can be defined from T_0 in the same way, but we do not need to identify $\mathrm{ap}(c_j^1, p)$ and $\mathrm{ap}(c_j^0, p)$, and thus we get a chain of arrows $T_0 \to T_1 \to T_2 \to \ldots$. We say that the interpretation of T is $\mathrm{colim}_{n \in \mathbb{N}} T_n$. Only the elimination and computation rules are left, because we already explain that it has the right constructors.

It is not difficult to verify the elimination rule. Let $Y[x]$ be a family of types on T. We can make a map $T_0 \to Y[x]$ by using induction. If we have a map $\prod x : T_i . Y[x]$, then we can extend it to a map $\prod x : T_{i+1} . Y[x]$. This is because every family of types $Y[x]$ with maps $z_i : \overline{H}(Y)(x) \to Y(c_i\, x)$ has all $\mathrm{ap}(z_i, p)$ for $p : I^n \to Y(x)$. Also, these satisfy $\mathrm{ap}(z_i, \mathrm{refl}) = \mathrm{refl}$, and the maps $\lambda j \lambda n \lambda p. \mathrm{ap}(c_j^{i+1}, h \circ p)$ and $\lambda j \lambda n \lambda p. \iota \circ \mathrm{ap}(c_j^i, p)$ are sent to $\lambda j \lambda n \lambda p. \mathrm{ap}(z_j, q)$ where q is the image of p. Hence, we get a map $\prod x : T . Y[x]$ by using the universal property of the colimit which is the interpretation of the elimination rule.

Lastly, the computation rules hold, because all involved diagrams commute. □

Note that if we do not require that $\mathrm{ap}(c_j, \mathrm{refl}) = \mathrm{refl}$, then we need to specify more to give a map $T \to Y$. This is because the images of all $\mathrm{ap}(c_j, \mathrm{refl})$ needs to be given as well. The same argument can be given for the requirement that $\mathrm{ap}(c_j^{i+1}, p)$ and $\mathrm{ap}(c_j^i, p)$.

4.5.2. Examples.

Now we can finally discuss the example we defined before.

Example 4.5.4 (Integers Modulo 2)**.** To define the integers modulo 2, we start with the natural numbers, and we add a path $p : 0 \rightsquigarrow S\,(S\,0)$. This can be done using a non higher inductive type, and thus we define

```
Inductive ℕ/2ℕ :=
  | 0 : ℕ/2ℕ
  | S : ℕ/2ℕ → ℕ/2ℕ
  | p : 0 ⇝ S (S 0)
```

We can easily form the elimination rule assuming $\Gamma, x : \mathbb{N}/2\mathbb{N} \vdash Y[x] : \mathrm{TYPE}$

$$\frac{\Gamma \vdash z : Y[0] \qquad \Gamma, n : \mathbb{N}/2\mathbb{N} \vdash s : Y[n] \to Y[S\,n] \qquad \Gamma \vdash q : z \rightsquigarrow_p^Y s\,(s\,z)}{\Gamma \vdash \mathbb{N}/2\mathbb{N}\text{-elim}(z, s, q) : \prod_{x : \mathbb{N}/2\mathbb{N}} Y[x]}$$

and the computation rules are

$$\mathbb{N}/2\mathbb{N}\text{-elim}(z, s, q)\, 0 = z,$$

$$\mathbb{N}/2\mathbb{N}\text{-elim}(z, s, q)\,(S\,n) = s\,(\mathbb{N}/2\mathbb{N}\text{-elim}(z, s, q)\, n),$$

$$\mathrm{apd}(\mathbb{N}/2\mathbb{N}\text{-elim}(z, s, q), p) = q.$$

Also, we have an inhabitant of $S\,0 \rightsquigarrow S^3\,0$, namely $\mathrm{apd}(S, p)$. In a similar way we can define the integers modulo n. We only have to change the last rule into $p : 0 \rightsquigarrow S^n\,0$.

It is insightful to work out the construction for this example. We start by making T_0 which is $\mathbb{N}$ with an equality between 0 and 2. To make T_1 we add paths $\mathrm{ap}(S, p)$ for $p : T_0^{I^1}$, and we say that $\mathrm{ap}(S, \mathrm{refl}) = \mathrm{refl}$. The only nontrivial path is $p : 0 \rightsquigarrow 2$, so now we get a path $\mathrm{ap}(S, p) : 1 \rightsquigarrow 3$. For T_2 we do the same, and now we get a path $\mathrm{ap}(S, \mathrm{ap}(S, p)) : 2 \rightsquigarrow 4$, but now again we get another copy of $\mathrm{ap}(S, p)$. The coequalizer says that the copy of $\mathrm{ap}(S,p)$ we just made is the same as the first one. This construction can be continued to get T_i for every i. Taking the colimit of all T_i we get a type T such that $n \rightsquigarrow n+2$ always has an inhabitant.

4.6. Recursive Higher Inductive Types

An important example of a higher inductive type is the truncation of a type. If we have a type A, then for the truncation $||A||$ we need a map $A \to ||A||$ and a map $\prod x : ||A|| \prod y : ||A||.x \rightsquigarrow y$, so everything in the truncation must be equal. This is defined in the following way

Inductive $||A||$ $(A : \textsc{Type}) :=$
  $|\ \iota : A \to ||A||$
  $|\ p : \prod x : ||A|| \prod y : ||A||.x \rightsquigarrow y$

Up to now we are unable to interpret this one, because the path p has $||A||$ in the quantifiers. So, we can say that this is a recursive higher inductive type, because the quantifiers uses the original type.

For these types we do not have a concrete interpretation, but instead we will give a suggestion of a possible definition.

Definition 4.6.1 (Recursive Higher Inductive Type)**.** A *recursive higher inductive type* is defined according to the following syntax

Inductive T $(B_1 : \textsc{Type})\dots(B_\ell : \textsc{Type}):=$
  $|\ c_1 : H_1(T) \to T$
  $\dots$
  $|\ c_k : H_k(T) \to T$
  $|\ p_1 : \prod x : A_1(T).F_1[x, c_1, \dots, c_k] \rightsquigarrow G_1[x, c_1, \dots, c_k]$
  $\dots$
  $|\ p_n : \prod x : A_n(T).F_n[x, c_1, \dots, c_k, p_1, \dots, p_{n-1}] \rightsquigarrow G_n[x, c_1, \dots, c_k, p_1, \dots, p_{n-1}]$

Again we write $\overline{F_i} = F_i[x, c_1, \dots, c_k, p_1, \dots, p_{i-1}]$ and $\overline{G_i} = G_i[x, c_1, \dots, c_k, p_1, \dots, p_{i-1}]$, and again we require that $\overline{F_i}$ and $\overline{G_i}$ are terms in $(I^{d_i} \to T)(x, c_1, \dots, c_k, p_1, \dots, p_{i-1})$ with variables $x : A_i(T)$, $c_j : H_j(T) \to T$ and

$$p_j : \prod x : A_j(T).F_j[x, c_1, \dots, c_k, p_1, \dots, p_{j-1}] \rightsquigarrow G_j[x, c_1, \dots, c_k, p_1 \dots, p_{j-1}].$$

Furthermore, we require that $A_i(T)$ is any type depending on $B_1, \dots B_\ell, T$, which is polynomial in T, in Martin-Löf type theory with higher inductive types, and that every H_i is polynomial. Also, we write $T = T(B_1, \dots, B_\ell)$, and using that notation we give the introduction rules for the points

$$\frac{\Gamma \vdash x : H_i(T)}{\Gamma \vdash c_i\, x : T}$$

and the introduction rules for the paths

$$\Gamma \vdash p_i : \prod x : A_i(T).\overline{F_i} \rightsquigarrow \overline{G_i}$$

However, in contrast to the previous versions, we will only give a nondependent elimination rule, because we were unable to formulate a dependent one. We assume that we have a type Y such that $\Gamma \vdash Y : \textsc{Type}$, and then the elimination rule is

$$\frac{\Gamma \vdash z_i : H_i(Y) \to Y \;\text{ for } i = 1,\dots,k \qquad \Gamma \vdash q_i : \prod x : A_i(Y).F_i' \rightsquigarrow G_i' \;\text{ for } i = 1,\dots,n}{\Gamma \vdash T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n) : T \to Y}$$

where $F_i' = F_i[x, z_1,\dots,z_k,q_1,\dots,q_{i-1}]$ and $G_i' = G_i[x,z_1,\dots,z_k,q_1,\dots,q_{i-1}]$. Lastly, we have similar computation rules for $t : H_i(T)$

$$T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n)\,(c_i\,t) = z_i\,(H_i(T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n))\,t)$$

and for all $t : A_i(T)$

$$\text{apd}(T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n), p_i\,x) = q_i\,(A_i(T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n))\,t). \quad ⌟$$

Note that A_i gives a map $A_i(T\text{-elim}(z_1,\dots,z_k,q_1,\dots,q_n)) : A_i(T) \to A_i(Y)$. In the same way as in Definition 4.5.1 we can check that the computation rules preserve types. Now let us discuss some examples of recursive higher inductive types.

Example 4.6.2 (Truncation)**.** Recall that the truncation which is defined as follows

```
Inductive $||A||$ $(A : \text{Type})$ :=
  | $\iota : A \to ||A||$
  | $p : \prod x : ||A|| \prod y : ||A||.x \rightsquigarrow y$
```

From Definition 4.6.1 we can already see the introduction rules, namely we have $\iota\,x : ||A||$ for $x : A$, and for $x : ||A||$ and $y : ||A||$ we have $p\,x\,y : x \rightsquigarrow y$. The elimination rule is as follows

$$\frac{\Gamma \vdash Y : \text{Type} \qquad \Gamma \vdash i : A \to Y \qquad \Gamma \vdash q : \prod x : Y \prod y : Y.x \rightsquigarrow y}{\Gamma \vdash ||A||\text{-elim}(i,q) : ||A|| \to Y}$$

and we have two computation rules which say that for $t : A$

$$||A||\text{-elim}(i,q)\,(\iota\,x) = i\,x$$

and for all $x : ||A||$ and $y : ||A||$ we have

$$\text{apd}(||A||\text{-elim}(i,q), p\,x\,y) = q\;(||A||\text{-elim}(i,q)\,x)(||A||\text{-elim}(i,q)\,y).$$

Note that the axiom $\Gamma \vdash q : \prod x : Y \prod y : Y.x \rightsquigarrow y$ says that Y is a mere proposition. So, in words the elimination rule says that $||A||$ is the smallest mere proposition containing A.

We can also define higher truncations.

Example 4.6.3 (Higher Truncation)**.** Let $n : \mathbb{N}$ be any fixed natural number. The definition of the higher truncation is similar to the definition of the truncation.

```
Inductive $||A||_n$ $(A : \text{Type})$ :=
  | $\iota : A \to ||A||$
  | $p : \prod x : ||A||^{I^n} \prod y : ||A||^{I^n}.x \rightsquigarrow y$
```

Again the introduction rules can be seen easily, namely we have $\iota\,x : ||A||$ for $x : A$, and for $x : ||A||^{I^n}$ and $y : ||A||^{I^n}$ we have $p\,x\,y : x \rightsquigarrow y$. The elimination rule is as follows

$$\frac{\Gamma \vdash Y : \text{Type} \qquad \Gamma \vdash i : A \to Y \qquad \Gamma \vdash q : \prod x : Y^{I^n} \prod y : Y^{I^n}.x \rightsquigarrow y}{\Gamma \vdash ||A||\text{-elim}(i,q) : ||A|| \to Y}$$

and we have two computation rules which say that for $x : A$

$$||A||\text{-elim}(i,q)\,(\iota\,x) = i\,x$$

and for all $x : ||A||^{I^n}$ and $y : ||A||^{I^n}$ we have

$$\text{apd}(||A||\text{-elim}(i,q), p\,x\,y) = q\,(||A||\text{-elim}(i,q)\,x)\,(||A||\text{-elim}(i,q)\,y).$$

There are several issues when interpreting recursive higher inductive types, and to explain them we look at the truncation of $1+1+1$. Note that $1+1+1$ has three points $*_1, *_2, *_3$. In the truncation there should be new paths $p_{ij} : *_i \rightsquigarrow *_j$ for $i, j \in \{1, 2, 3\}$. Now if we interpret this in topological spaces, then new points are added, because a path contains uncountably many points. So, on p_{12} we can find a point x and on p_{13} we can find a point y such neither x nor y is an endpoint. Because in the truncation there should be a path between every two points, there should be a path p_{xy} between x and y. However, we have a choice. We have $x \rightsquigarrow *_1$ and $y \rightsquigarrow *_1$, but we also have $x \rightsquigarrow *_2$ and $y \rightsquigarrow *_3$ giving $x \rightsquigarrow y$, because we have a path $*_2 \rightsquigarrow *_3$. For the truncation we need to choose one of these paths, and this is the first difficulty. This can be solved by again adding paths between every x and y, and this can be continued. However, the main problem we had was when to stop adding. At some point the construction should be completed, and this has to be given in a general fashion.

CHAPTER 5

Applications

In this chapter we discuss some examples of higher inductive types. These examples, including modular arithmetic, finite sets and integers, are often difficult or impossible to define using normal inductive types. With inductive definitions free data types are created in which no equations are satisfied. Instead of defining them directly, one gives a coding of these in some other data type like lists. Using higher inductive types it is possible to define them and reason about them in a direct way.

5.1. Modular Arithmetic

The first example we discuss is modular arithmetic. We define the type $\mathbb{N}/n\mathbb{N}$ which are the integers modulo n.

```
Inductive $\mathbb{N}/n\mathbb{N}$ :=
  | $0 : \mathbb{N}/n\mathbb{N}$
  | $S : \mathbb{N}/n\mathbb{N} \to \mathbb{N}/n\mathbb{N}$
  | $p : 0 \rightsquigarrow S^n\, 0$
```

Note that this is a nonrecursive type, because the path $0 \rightsquigarrow S^n\, 0$ does not use the type $\mathbb{N}/n\mathbb{N}$. This type has the following elimination rule for $\Gamma, x : \mathbb{N}/n\mathbb{N} \vdash Y[x] : \textsc{Type}$

$$\frac{\Gamma \vdash z : Y[0] \qquad \Gamma, x : \mathbb{N}/n\mathbb{N} \vdash s : Y[x] \to Y[S\,x] \qquad \Gamma \vdash q : z \rightsquigarrow^{Y}_{p} s^n\, z}{\Gamma \vdash \mathbb{N}/n\mathbb{N}\text{-elim}(z,s,q) : \prod x : \mathbb{N}/n\mathbb{N}. Y[x]}$$

and we have the following computation rules

$$\mathbb{N}/n\mathbb{N}\text{-elim}(z,s,q)\, 0 = z,$$

$$\mathbb{N}/n\mathbb{N}\text{-elim}(z,s,q)\,(S\,n) = s\,(\mathbb{N}/n\mathbb{N}\text{-elim}(z,s,q)\,n),$$

$$\text{apd}(\mathbb{N}/n\mathbb{N}\text{-elim}(z,s,q), p) = q.$$

For this type we can define addition, which uses the following proposition.

Proposition 5.1.1. *For every $m : \mathbb{N}/n\mathbb{N}$ we have an inhabitant q_m of $m \rightsquigarrow S^n\, m$.*

Proof. Define $Y[x]$ to be the type $x \rightsquigarrow S^n\, x$. Now we need to give an inhabitant z of $Y[0] = 0 \rightsquigarrow S^n\, 0$ for which we can take p. Next, we have to give an inhabitant of $Y[x] \to Y[S\,x]$, so a map $s : x \rightsquigarrow S^n\, x \to S\,x \rightsquigarrow S^n\,(S\,x)$. We define $s = \lambda q.\, \text{apd}(S,q) : x \rightsquigarrow S^n\, x \to S\,x \rightsquigarrow S^n\,(S\,x)$.

Finally, we need to give an inhabitant of $p \rightsquigarrow^{Y}_{p} s^n\, p$, so we need to give a path $p_*(p) \rightsquigarrow s^n\, p$ according to Definition 2.3.7. We will give a map $\prod x : \mathbb{N}/n\mathbb{N} \prod y : \mathbb{N}/n\mathbb{N} \prod q : x \rightsquigarrow y. p_*(q) \rightsquigarrow s^n\, q$ instead, and for that we use path induction. Taking q to be refl_x, we see that both $p_*(q)$ and $s^n\, q$ are refl, and thus reflexivity is a path between them. Hence, we can specialize this to the case $p_*(p) \rightsquigarrow s^n\, p$, and thus we get the desired path. We then apply the elimination rule to a map $\prod m : \mathbb{N}/n\mathbb{N}. m \rightsquigarrow S^n\, m$ □

For addition we basically define for every $m : \mathbb{N}/n\mathbb{N}$ a function f_m, which represents $\lambda x.x + m$, and f_m can be defined using the induction principle for $\mathbb{N}/2\mathbb{N}$. The environment Γ consists just of the declaration $m : \mathbb{N}/n\mathbb{N}$. To apply the elimination rule, we first need to give a type $Y[x]$ for every $x : \mathbb{N}/2\mathbb{N}$, and for $Y[x]$ we take $\mathbb{N}/n\mathbb{N}$. Next, we need to give an inhabitant z of $\mathbb{N}/n\mathbb{N}$, which needs to be m. Also, we need to give a function $s : \mathbb{N}/n\mathbb{N} \to \mathbb{N}/n\mathbb{N}$ which will be S. Lastly, we need to give a path between m and $S^n(m)$, which we can take to be q_m by Proposition 5.1.1. This gives us the desired function $f_m = \mathbb{N}/n\mathbb{N}\text{-elim}(S^m(0), S, q_m) : \mathbb{N}/n\mathbb{N} \to \mathbb{N}/n\mathbb{N}$. The computation rules give

$$f_m\,0 = m,$$
$$f_m\,(S\,x) = S\,(f_m\,x),$$
$$\mathrm{apd}(f_m, p) = q_m.$$

Hence, we can define $+ : \mathbb{N}/n\mathbb{N} \to \mathbb{N}/n\mathbb{N} \to \mathbb{N}/n\mathbb{N}$ by sending m to f_m.

Now let us show that $\mathbb{N}/2\mathbb{N}$ has two inhabitants, namely 0 and 1.

Proposition 5.1.2. *We have* $\mathbb{N}/2\mathbb{N} \simeq \top + \top$.

PROOF. We denote the inhabitants of $\top + \top$ by $*_1$ and $*_2$. First, we construct a map $f : \top + \top \to \mathbb{N}/2\mathbb{N}$ by sending $*_1$ to 0 and $*_2$ to $S\,0$.

Next, we define a map $g : \mathbb{N}/2\mathbb{N} \to \top + \top$ as follows. We send 0 to $*_1$, and for the map $h : \top + \top \to \top + \top$ we take the permutation which sends $*_1$ to $*_2$ and $*_2$ to $*_1$. Lastly, we need a path $q : *_1 \rightsquigarrow h(h *_1)$, and since $h(h *_1) = *_1$ by definition, we can take $q = \mathrm{refl}$. Now $\mathbb{N}/n\mathbb{N}$-elimination gives a function $g : \mathbb{N}/2\mathbb{N} \to \top + \top$.

It is not difficult to check that these maps are mutual inverses, and thus we have $\mathbb{N}/2\mathbb{N} \cong \top + \top$. □

5.2. Truncations

Suppose, we have a type A, and we would like to make a type B such that we have a maps $i : A \to B$ and $p : \prod x : B \prod y : B.x \rightsquigarrow y$. To define such a type B, we have two candidates.

```
Inductive ||A|| (A : Type) :=
  | ι : A → ||A||
  | p : ∏x : ||A|| ∏y : ||A||.x ⇝ y
```

```
Inductive |A| (A : Type) :=
  | ι : A → |A|
  | p : ∏x : A ∏y : A.ι x ⇝ ι y
```

It is not difficult to check that $||A||$ satisfies the requirements by the introduction rules. However, one might expect tht $|A|$ also satisfies the requirements, and the question arises whether $||A||$ and $|A|$ are isomorphic. This is not the case if the univalence axiom holds. From [**LS13**] we have the following proposition

Proposition 5.2.1. *From the univalence axiom follows that the fundamental group of the circle is* $\mathbb{Z}$.

We will not need the precise definition of the fundamental group here, but we do need some properties. These are that isomorphic types have isomorphic fundamental groups and that contractible types have a trivial fundamental group, namely 0. Also, we need that $\mathbb{Z}$ and 0 are not isomorphic. Since the fundamental group of contractible types is trivial, we can conclude from univalence that the circle is not contractible.

From the definition one can easily see that $||A||$ is contractible for every A. Our goal is to show that $|A|$ is not always contractible which allows us to conclude that $|A|$ and $||A||$ do not have to be isomorphic. For that we show that S^1 and $|\top|$ are isomorphic, where $\top$ is the type with just one point.

Theorem 5.2.2. *The types* $|\top|$ *and* S^1 *are isomorphic.*

PROOF. First, we make a function $f : |\top| \to S^1$. For this we need to give a map $i : \top \to S^1$ and a map $q : \prod x : \top \prod y : \top . i\, x \rightsquigarrow i\, y$. We define $i : \top \to S^1$ by sending the point $*$ to base. Also, q can be defined using induction, and if we take $x = *$ and $y = *$, then the path is loop. Now $|\top|$-recursion gives the map f. Next, we make a map $S^1 \to |\top|$. The point base is sent to $\iota *$, and loop is sent to $(p *) *$. Using S^1-recursion we get a map $g : S^1 \to |\top|$.

Lastly, we need to verify that these maps are mutual inverses. Note that

$$g\,(f\,(\iota *)) = g\,(\text{base}) = \iota(*),$$

$$\text{ap}(g, \text{ap}(f, (p *) *)) = \text{ap}(g, \text{loop}) = (p *) *,$$

and this shows the first requirement. For the second requirement, we note that

$$f\,(g\ \text{base}) = f\,(\iota *) = \text{base},$$

$$\text{ap}(f, \text{ap}(g, \text{loop})) = \text{ap}(f, ((p *) *)) = \text{loop},$$

and thus $f \circ g$ is the identity as well. Hence, the types are indeed isomorphic. □

We can thus conclude that $|\top|$ is not contractible if univalence holds. Hence, we get that $|A|$ and $||A||$ are not isomorphic.

For higher truncations we can do something similar.

Inductive $||A||_n$ $(A : \text{TYPE}) :=$
  | $\iota : A \to ||A||_n$
  | $p : \prod f : I^n \to ||A||_n \prod g : I^n \to ||A||_n . x \rightsquigarrow y$

Inductive $|A|_n$ $(A : \text{TYPE}) :=$
  | $\iota : A \to |A|_n$
  | $p : \prod f : I^n \to A \prod g : I^n \to A . \iota \circ f \rightsquigarrow \iota \circ g$

We can prove that $|*| = S^n$, and the nth homotopy group of S^n is nontrivial [**LB13**]. Hence, since the nth truncation has a trivial nth homotopy group, these types must be different.

5.3. Data Types in Functional Programming

In this section we will discuss some data types in functional programming which we can define using higher inductive types. These are the integers and finite sets, and they cannot easily be defined using just inductive types. Furthermore, we prove some properties of finite sets.

Let us start with a simple example of a data type in functional programming we can make using higher inductive types. After that example we look at finite sets in detail. This example is the integers, and the main idea is that instead of just saying that we have the operation $S : \mathbb{Z} \to \mathbb{Z}$, we say that S has an inverse $P : \mathbb{Z} \to \mathbb{Z}$. Hence, we need to require two equalities, namely $\prod x : \mathbb{Z}. P\,(S\,x) \rightsquigarrow x$ and $\prod x : \mathbb{Z}. S\,(P\,x) \rightsquigarrow x$. In a data type this is written as

Inductive $\mathbb{Z} :=$
$\mid 0 : \mathbb{Z}$
$\mid S : \mathbb{Z} \to \mathbb{Z}$
$\mid P : \mathbb{Z} \to \mathbb{Z}$
$\mid q_1 : \prod x : \mathbb{Z}.P\,(S\,x) \rightsquigarrow x$
$\mid q_2 : \prod x : \mathbb{Z}.S\,(P\,x) \rightsquigarrow x$

In this case we can prove that

$$\vdash \prod x : \mathbb{Z} \prod y : \mathbb{Z}.S\,x \rightsquigarrow S\,y \to x \rightsquigarrow y$$

by using that S has an inverse. More properties can be proved, but we will not discuss these.

Next we discuss finite sets, and we will prove some properties of them. To do that, we need some other data types first. First, we define a data type Bool as follows

Inductive $\textsc{Bool} :=$
$\mid \text{True} : \textsc{Bool}$
$\mid \text{False} : \textsc{Bool}$

Using the elimination principle we can define operations $\vee$ and $\wedge$. Furthermore, we assume that $\text{True} \rightsquigarrow \text{False} \simeq \bot$, meaning that True and False are not equal. Some other terminology that we will need are the notions of *mere propositions* and *sets*.

Definition 5.3.1 (Mere Proposition)**.** A type A is called a *mere proposition* iff the type $\prod x : A \prod y : A.x \rightsquigarrow y$ is inhabited. ⌟

It is easy to prove that two mere propositions are isomorphic if there are functions between them.

Lemma 5.3.2 (Lemma 3.3.3 in [**Uni13**])**.** *Let P and Q be mere propositions. Then P and Q are isomorphic iff we have $f : P \to Q$ and $g : Q \to P$.*

Furthermore, mere propositions are closed under products.

Lemma 5.3.3 (Lemma 3.6.1 in [**Uni13**])**.** *If P and Q be mere propositions, then $P \times Q$ is a mere proposition.*

From Lemma 5.3.3 we can conclude the following lemma.

Lemma 5.3.4. *If $A : \textsc{Type}$ and $B : \textsc{Type}$, then $||A \times B|| \simeq ||A|| \times ||B||$.*

We will use Lemma 5.3.4 frequently, so we will not give explicit references to it. Some types have unique identity proofs, and such types are called *sets*. This means that every $x \rightsquigarrow y$ is a mere proposition, or more formally.

Definition 5.3.5 (Set)**.** A type A is called a *set* if for all $x, y : A$ the type $x \rightsquigarrow y$ is a mere proposition. ⌟

An example of a set is Bool. This can be proven using Hedberg's Theorem [**Hed98, Uni13**].

Proposition 5.3.6. *The type* Bool *is a set.*

Now we will define a data type finite sets. Given a type A, we define the type of finite subsets of A as the free $\vee$-semilattice over A.

Definition 5.3.7 (Finite Sets)**.** We define the inductive type of finite sets $S(A)$ on A as follows.

```
Inductive S(A) (A : TYPE) :=
  | 1 : S(A)
  | L : A → S(A)
  | ∪ : S(A) × S(A) → S(A)
  | a : ∏x : S(A) ∏y : S(A) ∏z : S(A).x ∪ (y ∪ z) ⇝ (x ∪ y) ∪ z
  | n₁ : ∏x : S(A).x ∪ 1 ⇝ x
  | n₂ : ∏x : S(A).1 ∪ x ⇝ x
  | c : ∏x : S(A) ∏y : S(A).x ∪ y ⇝ y ∪ x
  | i : ∏x : A.L(x) ∪ L(x) ⇝ L(x)
```

So, we have $A : \text{TYPE} \vdash S(A) : \text{TYPE}$. ⌟

The constructor 1 gives the empty set, and the constructor L sends x to the singleton set $\{x\}$. We use the notation $\{x\}$ for $L\,x$. Also, we need to have an operator $\cup$ which gives the union of two finite sets. This operator should be associative, commutative, and we require that 1 is a neutral element for this operator. Furthermore, the union of $\{x\}$ and $\{x\}$ has to be $\{x\}$ so that multiplicity of the element does not matter. In technical terms, $S(A)$ is the free commutative idempotent monoid on A. If we remove c and i, then we just have the free monoid, and that is the data type of lists. If we only remove i, then this data type is the free commutative monoid, and this gives the data type of bags.

Note that we define this data type in several ways. For example, now we can find two inhabitants of $1 \cup x \rightsquigarrow x$. The first one is given by n_2, and the other one by $n_1 \circ c$. There is no requirement about the equality of those, so they might be different. It depends on what we want from the type: if we want this type to be a 1-type, then we need some extra requirement saying that we have an inhabitant of $p \rightsquigarrow q$ for any two paths p and q. This has one major disadvantage, namely that in such cases we can only map it to other 1-types, and for more general types it is more difficult.

Let us start with a simple property.

Proposition 5.3.8. *We can find a term of type* $\prod x : S(A).||x \cup x \rightsquigarrow x||$.

PROOF. We use $S(A)$-elimination. If $x = 1$, then n_1 gives a path $1 \cup 1 \rightsquigarrow x$. In the case that $x = \{a\}$, then i gives a path $\{a\} \cup \{a\} \rightsquigarrow \{a\}$.

For the union, if $x = y \cup z$ and if we have paths $p : ||y \cup y \rightsquigarrow y||$ and $q : ||z \cup z \rightsquigarrow z||$. Note that from a and c we get a path $y \cup z \cup y \cup z \rightsquigarrow y \cup y \cup z \cup z$. Also, p and q give paths $y \cup y \cup z \cup z \rightsquigarrow y \cup z \cup z$ and $y \cup z \cup z \rightsquigarrow y \cup z$. Together all these paths give a path $y \cup z \cup y \cup z \rightsquigarrow y \cup z$.

Finally, we need to find images of a, n_1, n_2, c and i. For example, let us show how we can find the image of i. Note that $\{a\} \cup \{a\}$ gets sent to a path p of type $||\{a\}\cup\{a\}\cup\{a\}\cup\{a\} \rightsquigarrow \{a\}\cup\{a\}||$ and that $\{a\}$ gets sent to a path q of type $||\{a\}\cup\{a\} \rightsquigarrow \{a\}||$. The image of i must be a path of type $p \rightsquigarrow_i^Y q$. To find the inhabitant of that type, we use that $||y \cup y \rightsquigarrow y||$ is a mere proposition. Between any two inhabitants of a mere proposition there is always a path, and thus we can find a path between $i_*(p)$ and q. For the other paths similar arguments can be given, and thus we get a term of type $\prod x : S(A).||x \cup x \rightsquigarrow x||$. □

Note the truncation in this proposition. Normally, we have to give the images of paths in higher inductive type explicitly, but now it is east due to the truncation. This is because we can always find the paths in truncations.

Next we show that some of the usual set theoretic axioms hold for $S(A)$. For that we must define a relation $\in : A \times S(A) \to \textsc{Bool}$, which says whether some element is in some finite set.

Let us start by defining the elements of a set. For that we need to know when two elements of A are equal, so first we need the notion of decidable equality.

Definition 5.3.9 (Merely Decidable Equality)**.** A type A has *merely decidable equality* iff we have a map $== : A \times A \to \textsc{Bool}$ such that for all $a, b : A$ the types $\|a \rightsquigarrow b\|$ and $a == b \rightsquigarrow \text{True}$ are isomorphic. ⌟

Since we will only talk about merely decidable equality, we will just say decidable equality from now on. This definition is equivalent to the definition of merely decidable equality in [**Uni13**].

Definition 5.3.10 (Elements)**.** Suppose, A is a type with decidable equality. Then we define a function $\in : A \times S(A) \to \textsc{Bool}$ by induction on $S(A)$ as follows.

$$\begin{gathered} \in (a, 1) = \text{False}, \\ \in (a, \{b\}) = a == b, \\ \in (a, x \cup y) = \in (a, x) \vee \in (a, y). \end{gathered}$$

Note that by definition we have

$$\in (a, x \cup 1) = \in (a, x) \vee \in (a, 1),$$

and that we have a path $\in (a, x) \vee \in (a, 1) \rightsquigarrow \in (a, 1)$. In the same way we can show that we have a path $\in (a, 1 \cup x) \rightsquigarrow \in (a, x)$. Also, by writing out the definitions we get a path $\in (a, x \cup y) \rightsquigarrow \in (a, y \cup x)$. Lastly, we have by definition

$$\in (a, \{b\} \cup \{b\}) = \in (a, \{b\}) \vee \in (a, \{b\}),$$

and we have a path $\in (a, \{b\}) \vee \in (a, \{b\}) \rightsquigarrow \in (a, \{b\})$. ⌟

We denote $\in (a, x)$ by $a \in x$. Note that by definition we have $a \in x \cup y$ iff $a \in x \vee a \in y$, and thus $\cup$ is indeed the union. Let us next define the comprehension. We write $\{a : \varphi\}$ to denote the set of elements in a satisfying some predicate φ.

Definition 5.3.11 (Comprehension)**.** We define $\{\} : S(A) \times (A \to \textsc{Bool}) \to S(A)$ using induction to $S(A)$.

$$\begin{gathered} \{1 : \varphi\} = 1, \\ \{\{a\} : \varphi\} = \text{if } \varphi\, a \text{ then } \{a\} \text{ else } 1, \\ \{x \cup y : \varphi\} = \{x : \varphi\} \cup \{y : \varphi\}. \end{gathered}$$

The required paths can easily be verified to exist, thus we get the desired map. ⌟

For the comprehension we need that $a \in \{x : \varphi\}$ iff $a \in x \wedge \varphi\, a$. To prove this we use induction on $S(A)$.

Theorem 5.3.12. *We have an inhabitant of* $a \in \{x : \varphi\} \rightsquigarrow a \in x \wedge \varphi\, a$.

Proof. Taking x to be 1, we need to find an inhabitant of $a \in \{1 : \varphi\} \rightsquigarrow a \in 1 \wedge \varphi\, a$. Note that $\{1 : \varphi\} = 1$, and that we have a path $a \in 1 \rightsquigarrow \text{False}$. Since we have a path $\text{False} \wedge b \rightsquigarrow b$, we need an inhabitant of $\text{False} \rightsquigarrow \text{False}$ which is refl.

Next, we take x to be $\{b\}$, which means that we need to find an inhabitant of $a \in \{\{b\} : \varphi\} \rightsquigarrow a \in \{b\} \wedge \varphi\, a$. Note that $\{\{b\} : \varphi\} = \text{if } \varphi\, b \text{ then } \{b\} \text{ else } 1$. There are four cases now, because we can make case distinctions on $a == b$ and $\varphi\, b$. If $\varphi\, b$ is false, then $\{\{b\} : \varphi\} = 1$, and if $\varphi\, b$ happens to be true, then $\{\{b\} : \varphi\} = \{b\}$.

Furthermore, $a \in \{b\}$ is equivalent to $a == b$. So, if $a == b$ is False or if $\varphi\, b$ is False, then one can check that both $a \in \{\{b\} : \varphi\}$ and $a \in \{b\} \wedge \varphi\, a$ are False.

If both $a == b$ is False and $\varphi\, b$ are True, then we proceed in a different way. Note that $a == b \rightsquigarrow$ True is isomorphic to $a \rightsquigarrow b$, because A has decidable equality. Hence, if we assume that $a == b$ is True, then we have a path $a \rightsquigarrow b$ which gives a path $\varphi\, a \rightsquigarrow \varphi\, b$. From this one can give a path $a \in \{\{b\} : \varphi\} \rightsquigarrow a \in \{b\} \wedge \varphi\, a$, and that finishes this case.

The last point constructor is the union, so we take x to be $y \cup z$. We assume that we have inhabitants $a \in \{y : \varphi\} \rightsquigarrow a \in y \wedge \varphi\, a$ and $a \in \{z : \varphi\} \rightsquigarrow a \in z \wedge \varphi\, a$. Writing out the definition of $\{y \cup z : \varphi\}$, we get an inhabitant of

$$a \in \{y \cup z : \varphi\} \rightsquigarrow a \in \{y : \varphi\} \cup \{z : \varphi\}.$$

This can be simplified further to get an inhabitant of

$$a \in \{y \cup z : \varphi\} \rightsquigarrow a \in \{y : \varphi\} \vee a \in \{z : \varphi\}.$$

Next we use the induction hypothesis, which says that we have inhabitants of $a \in \{y : \varphi\} \rightsquigarrow a \in y \wedge \varphi\, a$ and $a \in \{z : \varphi\} \rightsquigarrow a \in z \wedge \varphi\, a$, to obtain

$$a \in \{y \cup z : \varphi\} \rightsquigarrow (a \in y \wedge \varphi\, a) \vee (a \in z \wedge \varphi\, a).$$

Simplifying the right hand side gives an inhabitant of

$$a \in \{y \cup z : \varphi\} \rightsquigarrow (a \in y \vee a \in z) \wedge \varphi\, a.$$

Noting that $a \in y \cup z \rightsquigarrow a \in y \vee a \in z$, we get an inhabitant of

$$a \in \{y \cup z : \varphi\} \rightsquigarrow a \in y \cup z \wedge \varphi\, a.$$

This is what we wanted, so the property holds for the union.

Lastly, for the paths one needs to verify that both the left and right hand side gets sent to the same element. We do this for the path a Note that $(x \cup y) \cup z$ gets sent to a path $p : b \in \{(x \cup y) \cup z : \varphi\} \rightsquigarrow b \in (x \cup y) \cup z \wedge \varphi\, b$ and $x \cup (y \cup z)$ gets sent to a path $q : b \in \{x \cup (y \cup z) : \varphi\} \rightsquigarrow b \in x \cup (y \cup z) \wedge \varphi\, b$. Hence, the image of a must be a path of type $p \rightsquigarrow_a q$. Using that Bool is a set, it suffices to show that these get sent to the same boolean, which can easily be verified. □

We can define more operations like the intersection

$$x \cap y = \{x : \lambda a.a \in y\},$$

and the difference

$$x - y = \{x : \lambda a.\neg(a \in y)\}.$$

From Theorem 5.3.12 we can now easily deduce the following proposition.

Proposition 5.3.13. *There is an inhabitant of* $a \in x \cap y \rightsquigarrow a \in x \wedge a \in y$.

With this setup we show that every finite set has a size by defining an operator $\# : S(A) \to \mathbb{N}$. This is straightforward, except in the case $\cup$ for which we use the principle of inclusion and exclusion.

Definition 5.3.14 (Size)**.** Define $\# : S(A) \to \mathbb{N}$ by

$$\#(\mathbb{1}) = 0,$$
$$\#(\{a\}) = 1,$$
$$\#(x \cup y) = \#(x) + \#(y) - \#(x \cap y).$$

Note that a, n_1, n_2, c and i all can be mapped to refl by writing out the definitions. ⌟

The next important property is extensionality, which says that sets are equal iff they have the same elements. Our goal is now to show that this holds if we have decidable equality on the elements. First, we need to define when two sets are equal.

Definition 5.3.15 (Subsets). Let A be a type with decidable equality. We define a function $\subseteq : S(A) \times S(A) \to \textsc{Bool}$ by induction as follows

$$\subseteq (1, x) = \text{True},$$
$$\subseteq (\{a\}, x) = a \in x,$$
$$\subseteq (y \cup z, x) = (\subseteq (y, x)) \wedge (\subseteq (z, x)).$$

Note again that the needed equalities hold, and thus we get the function $\subseteq : S(A) \times S(A) \to \textsc{Bool}$. ⌟

We will denote $\subseteq (x, y)$ by $x \subseteq y$. This relation says that x is a subset of y. It can also be defined in an alternative way by saying that every element of x is an element of y.

Definition 5.3.16 (Subsets). Let A be a type with decidable equality. Then we define the type $x \subseteq' y$ as $\prod a : A.a \in x \rightsquigarrow \text{True} \to a \in y \rightsquigarrow \text{True}$. ⌟

Now we show that these definitions coincide

Proposition 5.3.17. *For a type A with decidable equality the types $x \subseteq y \rightsquigarrow$ True and $x \subseteq' y$ are isomorphic.*

Proof. Since both $x \subseteq y \rightsquigarrow$ True and $x \subseteq' y$ are mere propositions, it suffices to make maps $x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y$ and $x \subseteq' y \to x \subseteq y \rightsquigarrow \text{True}$ by Lemma 5.3.2. We start with making a map $x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y$ and for that we use the elimination rule of $S(A)$ on x. More specifically, for all x we make an inhabitant of the type $\prod y : A.(x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y)$, and that will be done using the elimination rule. If $x = 1$, then the type $a \in x \rightsquigarrow$ True is always $\perp$, so we have a map $\prod a : A.a \in x \rightsquigarrow \text{True} \to a \in y \rightsquigarrow \text{True}$.

In the second case we have $x = \{b\}$, and then the statements $x \subseteq y$ and $a \in x$ are equal to $b \in y$ and $a == b$ respectively. This means we need to make a map $b \in y \rightsquigarrow \text{True} \to \prod a : A.a == b \rightsquigarrow \text{True} \to a \in y \rightsquigarrow \text{True}$ Given a path $p : b \in y \rightsquigarrow$ True, an inhabitant $a : A$ and a path $q : a == b \rightsquigarrow$ True, we get make a path $r : a \in y \rightsquigarrow$ True. Note that q gives a path $q' : a \rightsquigarrow b$ that can be used to make a path $q'' : a \in y \rightsquigarrow b \in y$. Now we can make the path $r : a \in y \rightsquigarrow$ True. from p and q''.

In the third case we assume that $x = z_1 \cup z_2$ and that we have maps $f : z_1 \subseteq y \rightsquigarrow \text{True} \to z_1 \subseteq' y$ and $g : z_2 \subseteq y \rightsquigarrow \text{True} \to z_2 \subseteq' y$. By definition we have $x \subseteq y = z_1 \subseteq y \wedge z_2 \subseteq y$, and we have an isomorphism $x \subseteq y \rightsquigarrow \text{True} \simeq z_1 \subseteq y \rightsquigarrow \text{True} \times z_2 \subseteq y \rightsquigarrow$ True. Also, by definition we have $a \in x = a \in z_1 \vee a \in z_2$, and this gives $a \in x \rightsquigarrow \text{True} \simeq a \in z_1 \rightsquigarrow \text{True} + a \in z_2 \rightsquigarrow$ True. Given are inhabitants $p_1 \times p_2 : z_1 \subseteq y \rightsquigarrow \text{True} \times z_2 \subseteq y \rightsquigarrow$ True, $a : A$, and $q : a \in z_1 \rightsquigarrow \text{True} + a \in z_2 \rightsquigarrow$ True, we need to make an inhabitant $a \in y \rightsquigarrow$ True. Note that $f\, p_1\, a : a \in z_1 \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True and $g\, p_2\, a : a \in z_2 \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True. Using a case distinction on q and the maps $f\, p_1\, a$ and $g\, p_2\, a$, we can thus find an inhabitant of $a \in y \rightsquigarrow$ True, and that finishes this case.

Lastly, we need to say where the paths a, c, n_1, n_2 and i are mapped to. Again note that the type $x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y$ is a mere proposition, and thus we can always find the desired paths. Hence, the elimination rule gives a map $\prod x : A \prod y : A.(x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y)$ which is what we wanted.

To finish the proof we need to make a map $\prod x : A \prod y : A.(x \subseteq' y) \to x \subseteq y \rightsquigarrow$ True. For that we use the elimination rule of $S(A)$ on x and we let $y : A$ be arbitrary. First, we take x to be 1, and then we always have an inhabitant refl of $x \subseteq y \rightsquigarrow$ True. Hence, we can send everything of $(x \subseteq' y)$ to refl.

Next we assume $x = \{b\}$, and in that case $x \subseteq y = b \in y$. Suppose that we have $f : x \subseteq' y$. Since $x = \{b\}$, we always have an inhabitant refl of $b \in x \rightsquigarrow$ True. Now $f\ b$ refl is an inhabitant of $b \in y \rightsquigarrow$ True which is isomorphic to $x \subseteq y \rightsquigarrow$ True.

For the last case we assume that $x = z_1 \cup z_2$ and that we have $f : \prod a : A.z_1 \subseteq' y \to z_1 \subseteq y \rightsquigarrow$ True and $g : z_2 \subseteq' y \to z_2 \subseteq y \rightsquigarrow$ True. Given a map $h : \prod a : A.a \in x \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True, our goal is to make an inhabitant of $x \subseteq y \rightsquigarrow$ True, and note that $x \subseteq y \rightsquigarrow \text{True} \simeq (z_1 \subseteq y \rightsquigarrow \text{True}) \times (z_2 \subseteq y \rightsquigarrow \text{True})$. By definition we have $a \in x = a \in z_1 \vee z_2$, and therefore $a \in x \rightsquigarrow \text{True} \simeq (a \in z_1 \rightsquigarrow \text{True}) + (a \in z_2 \rightsquigarrow \text{True})$. Since we have a map $h : \prod a : A.a \in x \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True, we get maps $h_1 : \prod a : A.a \in z_1 \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True and $h_2 : \prod a : A.a \in z_2 \rightsquigarrow \text{True} \to a \in y \rightsquigarrow$ True. Hence, we can use the maps f, g, h_1, and h_2 to make an inhabitant of $x \subseteq y \rightsquigarrow \text{True} \simeq (z_1 \subseteq y \rightsquigarrow \text{True}) \times (z_2 \subseteq y \rightsquigarrow \text{True})$, and that finishes this case.

Again we have to say where the paths are mapped to, and this is easy because the type $\prod y : A.x \subseteq' y \to x \subseteq y \rightsquigarrow$ True is mere proposition. By the elimination rule we thus get a map $\prod x : A \prod y : A.x \subseteq' y \to x \subseteq y \rightsquigarrow$ True.

All in all, we have constructed maps $x \subseteq y \rightsquigarrow \text{True} \to x \subseteq' y$ and $x \subseteq' y \to x \subseteq y \rightsquigarrow$ True, and because these types are mere propositions, they are isomorphic. □

Note that equality can be defined using the subset relation in the following way.

Definition 5.3.18 (Equality of Sets)**.** Let A be a set with decidable equality. We define a function $== : S(A) \times S(A) \to \text{BOOL}$ sending (s, t) to $s \subseteq t \wedge t \subseteq s$. ⌟

Because of Proposition 5.3.17, two sets x and y are equal iff they have the same elements. Our goal is now to show that $S(A)$ has decidable equality given by this relation, which means that we need to show that $||x \rightsquigarrow y||$ and $x == y \rightsquigarrow$ True are isomorphic. Before we do so, we need some lemmas.

Lemma 5.3.19. *We have a term of type* $\prod x : S(A) \prod a : A.a \in x \rightsquigarrow \text{True} \to ||x \cup \{a\} \rightsquigarrow x||$.

PROOF. Again we use $S(A)$-elimination. If $x = 1$, then $a \in x \rightsquigarrow$ True is always the type $\perp$, and thus we get a map $a \in x \rightsquigarrow \text{True} \to ||x \cup \{a\} \rightsquigarrow x||$. Next we assume case that $x = \{b\}$, that we have an inhabitant $a : A$, and that we have a path $p : a \in x \rightsquigarrow$ True. Note that $a \in x \rightsquigarrow$ True and $a == b \rightsquigarrow$ True are isomorphic, and thus we get a path $p' : a \rightsquigarrow b$, because $==$ is the equality on A. The path p' gives a path $q : \{a\} \rightsquigarrow \{b\}$, and thus we get a path $\{b\} \cup \{a\} \rightsquigarrow \{b\} \cup \{b\}$. Because i gives a path $\{b\} \cup \{b\} \rightsquigarrow \{b\}$, we thus get a path $x \cup \{a\} \rightsquigarrow x$.

Lastly, we take $x = y \cup z$ and we assume that we have maps $f : \prod a : A.a \in y \rightsquigarrow \text{True} \to ||y \cup \{a\} \rightsquigarrow y||$ and $g : \prod a : A.a \in z \rightsquigarrow \text{True} \to ||z \cup \{a\} \rightsquigarrow z||$. Let $a : A$ be arbitrary. Note that $a \in x = a \in y \vee a \in z$, so $a \in x \rightsquigarrow \text{True} \simeq (a \in y \rightsquigarrow \text{True}) + (a \in z \rightsquigarrow \text{True})$. Using the elimination rule of $+$, we thus get a map h of type $\prod a : A.a \in x \rightsquigarrow \text{True} \to ((||y \cup \{a\} \rightsquigarrow y||) + (||z \cup \{a\} \rightsquigarrow z||))$. Given a path $y \cup \{a\} \rightsquigarrow y$, we get a path $y \cup \{a\} \cup z \rightsquigarrow y \cup z$, and this gives a path $y \cup z \cup \{a\} \rightsquigarrow y \cup z$. Similarly, a path $z \cup \{a\} \rightsquigarrow z$ gives a path $y \cup z \cup \{a\} \rightsquigarrow y \cup z$. We can thus conclude that we have a term of type $\prod a : A.a \in x \rightsquigarrow \text{True} \to ||x \cup \{a\} \rightsquigarrow x||$, and this finishes the argument. □

Lemma 5.3.20. *We always have an inhabitant of* $\prod x : S(A) \prod y : S(A) \prod z : S(A).||x \cup (y \cup z) \rightsquigarrow (x \cup y) \cup (y \cup z)||$.

PROOF. Note that we always have an inhabitant of $||x \cup x \rightsquigarrow x||$ by Proposition 5.3.8. Hence, we have an inhabitant of $||x \cup (y \cup z) \rightsquigarrow x \cup x \cup (y \cup z)||$. Now we can use the associative and the commutative property to find an inhabitant of $||x \cup (y \cup z) \rightsquigarrow (x \cup y) \cup (y \cup z)||$. □

These properties we need to show extensionality. We need one more lemma about subsets before we prove give the theorem.

Lemma 5.3.21. *We have an isomorphism between* $||x \cup y \rightsquigarrow x||$ *and* $y \subseteq x \rightsquigarrow \text{True}$.

PROOF. To prove this lemma, we use Lemma 5.3.2, so it suffices to make maps $||x \cup y \rightsquigarrow x|| \to y \subseteq x \rightsquigarrow \text{True}$ and $y \subseteq x \rightsquigarrow \text{True} \to ||x \cup y \rightsquigarrow x||$. We start with the map $||x \cup y \rightsquigarrow x|| \to y \subseteq x \rightsquigarrow \text{True}$. By Proposition 5.3.17 the types $y \subseteq x \rightsquigarrow \text{True}$ and $\prod a : A.a \in y \rightsquigarrow \text{True} \to a \in x \rightsquigarrow \text{True}$ are isomorphic, so instead we make a map $||x \cup y \rightsquigarrow x|| \to y \subseteq' x$. To make the map $||x \cup y \rightsquigarrow x|| \to y \subseteq x \rightsquigarrow \text{True}$, we use the elimination rule of the truncation, and thus it suffices to make a map $x \cup y \rightsquigarrow x \to y \subseteq x \rightsquigarrow \text{True}$. So, let $p : x \cup y \rightsquigarrow x$, let $a : A$ be any inhabitant and suppose we have a path $q : a \in y \rightsquigarrow \text{True}$. Since $a \in x \cup y = a \in x \vee a \in y$, we have a path $q' : a \in x \cup y \rightsquigarrow \text{True}$. From p and q we now get a path of type $a \in x \rightsquigarrow \text{True}$ which is what we wanted.

Next we make the map $y \subseteq x \rightsquigarrow \text{True} \to ||x \cup y \rightsquigarrow x||$. For this we use induction on y. If $y = 1$, then it is trivially true, because we always have the inhabitant refl, because $x \cup y = x$. Next we assume that $y = \{a\}$, and then $\{a\} \subseteq x \rightsquigarrow \text{True}$ is isomorphic to $a \in x \rightsquigarrow \text{True}$. in that case it follows from Lemma 5.3.19.

Lastly, we take $y = z_1 \cup z_2$, and we assume that we have maps $f : z_1 \subseteq x \rightsquigarrow \text{True} \to ||x \cup z_1 \rightsquigarrow x||$ and $g : z_2 \subseteq x \rightsquigarrow \text{True} \to ||x \cup z_2 \rightsquigarrow x||$. Our goal is to make a map $h : z_1 \cup z_2 \subseteq x \rightsquigarrow \text{True} \to ||x \cup z_1 \cup z_2 \rightsquigarrow x||$. By definition we have $z_1 \cup z_2 \subseteq x = z_1 \subseteq x \wedge z_2 \subseteq x$, so there is an isomorphism $z_1 \cup z_2 \subseteq x \rightsquigarrow \text{True} \simeq (z_1 \subseteq x \rightsquigarrow \text{True}) \times (z_2 \subseteq x \rightsquigarrow \text{True})$. From f and g we now get an inhabitant of $||x \cup z_1 \rightsquigarrow x|| \times ||x \cup z_2 \rightsquigarrow x||$. Note that we have a path $x \cup (z_1 \cup z_2) \rightsquigarrow (x \cup z_1) \cup (x \cup z_2)$ by Lemma 5.3.20, and thus we get an inhabitant of the type

$$||(x \cup (z_1 \cup z_2) \rightsquigarrow (x \cup z_1) \cup (x \cup z_2)) \times (x \cup z_1 \rightsquigarrow x) \times (x \cup z_2 \rightsquigarrow x)||.$$

We can map $(x \cup (z_1 \cup z_2) \rightsquigarrow (x \cup z_1) \cup (x \cup z_2)) \times (x \cup z_1 \rightsquigarrow x) \times (x \cup z_2 \rightsquigarrow x)$ to the type $z \cup (z_1 \cup z_2) \rightsquigarrow x$, and thus we get an inhabitant of $||z \cup (z_1 \cup z_2) \rightsquigarrow x||$ which is what we wanted. □

Now we can prove the following theorem which gives the extensionality and says that $==$ is decidable equality on $S(A)$.

Theorem 5.3.22. *Let A be a type with decidable equality. For all $x, y : S(A)$ the types* $||x \rightsquigarrow y||$ *and* $x == y \rightsquigarrow \text{True}$ *are isomorphic.*

PROOF. Note that $x == y \rightsquigarrow \text{True}$ and $x \subseteq y \wedge y \subseteq x \rightsquigarrow \text{True}$ are isomorphic by definition of $x == y$. Also, since both $x \subseteq y$ and $y \subseteq x$ are booleans, this type is isomorphic to $(x \subseteq y \rightsquigarrow \text{True}) \times (y \subseteq x \rightsquigarrow \text{True})$. By Lemma 5.3.21 this is isomorphic to $||x \cup y \rightsquigarrow x|| \times ||x \cup y \rightsquigarrow y||$. This is isomorphic to $||x \rightsquigarrow y||$, because we have functions $||x \rightsquigarrow y|| \to ||x \cup y \rightsquigarrow x|| \times ||x \cup y \rightsquigarrow y||$ and $||x \cup y \rightsquigarrow x|| \times ||x \cup y \rightsquigarrow y|| \to ||x \rightsquigarrow y||$ which means that they are isomorphic by Lemma 5.3.2. □

Other axioms for finite sets can be proven as well, for example, the power set axiom. We can make a map $\mathscr{P} : S(A) \to S(S(A))$ which gives the power set of a finite set. For this construction we will not give the details. However, note that the power set goes from $S(A)$ to $S(S(A))$.

To make a universe of set theory, we need to have all the axioms. This means that we have finite sets with elements from A, but also the finite sets with elements from $S(A)$. Let us consider the sequence of maps

$$S(A) \to S(A + S(A)) \to S(A + S(A) + S(S(A))) \to \dots.$$

We define a type $X_{n+1} = S(X_1 + \ldots + X_n)$. This means that inhabitants of X_{n+1} are finite sets with elements from $X_1, X_2, \ldots$, or X_n. Furthermore, we have inclusions $f_n : X_n \to X_{n+1}$. Now we take the colimit and for that we define the type

```
Inductive Sets (A : Type) :=
  | $\iota : \prod n : \mathbb{N}.X_n \to \mathbf{Sets}(A)$
  | $p : \prod n : \mathbb{N} \prod x : X_n.\iota_n\, x \rightsquigarrow (f_n \circ \iota_{n+1})\, x$
```

This gives a universe of sets.

Concluding, we have constructed a type $S(A)$ of finite subsets of A. This type has the desired properties of finite sets, and thus we can prove all usual properties of finite sets. For example, to prove that $\cap$ is associative, we can now just copy the usual proof from set theory. We can also give the notion of a function f from A to B which is an inhabitant of $S(A \times B)$, and with this notion we can prove the Cantor-Schröder-Bernstein Theorem.

CHAPTER 6

Conclusion and Related Work

6.1. Conclusion

As we have seen, there are several approaches to define higher inductive types. These differ in how much they allow: recursion might be allowed or not, and there could be restrictions on the inductive constructors. We have to give different interpretations for all of these, because if we allow functions in the point constructors, then we have to do a more to ensure that it will remain an interpretation. With these definitions one can extend proof assistants to allow formal definitions of S^1, $\mathbb{N}/2\mathbb{N}$, and the truncation $||A||$.

However, we have not given an interpretation of recursive higher inductive types. If we interpret them in the same way as nonrecursive higher inductive types, then there might be problems, because the recursion principle might not be satisfied. So, we would like to make a type T with a path $p : \prod x : T.s \rightsquigarrow t$ where s and t are terms. Our idea was to interpret the type by approximating it step by step starting with T_0 and making T_1 by gluing the path to T_0. We repeat this process to get $T_2, T_3, \ldots, T_\omega, T_{\omega+1}, \ldots$, and then we take the colimit. However, we could not get the proof to work. A point of research would thus be to give an interpretation of the recursive higher inductive types.

6.2. Required Improvements

There are some issues in the presented definitions which require a solution. First of all, the F_i and G_i in the definitions of higher inductive types should have better formation rules, so that their type can be controlled in a better way. The types of F_i' and G_i' can be better controlled if F_i and G_i are built in a certain way. Only a specified list of term constructors should be allowed, and then F_i' and G_i' can be made using a lifting property.

Secondly, maps $I^n \to A$ are equal too often, and for simplicity we restrict to the case for $n = 1$. Since I^1 is contractible, we can treat it as the type with one point. So, if $f, g : I^1$ are equal iff we have a path between $f\,0$ and $g\,0$. If we consider the type

```
Inductive $D :=$
| $* : D$
| $p : * \rightsquigarrow *$
| $s : \mathrm{Irec}(*, *, p) \rightsquigarrow \mathrm{Irec}(*, *, \mathrm{refl})$
```

We can make a map $D \to S^1$ sending $*$ to base, p to loop, and we have an image for s, because the maps $\mathrm{Irec}(*, *, p)$ and $\mathrm{Irec}(*, *, \mathrm{refl})$ are equal in 0. This should not be possible, because we cannot make such a map in topology. A possible solution would be to work with circles instead of cubes.

Also, this can also be solved by using cubical type theory. There one was $\mathbb{I}$, which is not considered to be a type, but one can use it to make paths. In a higher inductive type one would specify F_i and G_i using judgments $x : \mathbb{I} \vdash F_i$ and $x : \mathbb{I} \vdash G_i$, and that gives two paths. This gives a path between two paths.

Last, but not least, in the interpretation one should pay attention to fibrant objects. This is because in interpretations of homotopy type theory the objects are taken to be the fibrant objects. For it to be sensible, the construction needs to preserve fibrant objects.

6.3. Related Work

There is already a variety of work on higher inductive types. One approach is based on *homotopy-initial algebras* [**AGS12, Soj15**]. In the syntax of an inductive type we describe the constructors using functors, and the interpretation of the inductive type is given by an initial algebra of these. Higher inductive types are the homotopical analogue of these types, so a possible way to interpret these would be *homotopy initial algebras*. For that we first interpret inductive types in intensional type theory using homotopy initial algebras [**AGS12**]. Let F be a functor defining an inductive type, then we define

Definition 6.3.1 (F-Algebra). An *F-algebra* consists of a type A and a map $g : F(A) \to A$. We denote this by (A, g). ⌟

Definition 6.3.2 (Morphisms between F-Algebras). Let (C, g) and (D, h) be F-algebras. We define the type

$$\text{F-Alg}((C,g),(D,h)) = \sum f : C \to D. g \circ f \rightsquigarrow h \circ F(g) \qquad ⌟$$

This resembles the definitions of category theory. Now we can define

Definition 6.3.3 (Homotopy-Initial Algebra). Let (C, g) be an F-algebra. Then we say (C, g) is a *homotopy-initial algebra* iff for all F-algebras (D, h) the type $\text{F-Alg}((C,g),(D,h))$ is contractible. ⌟

This means that all inhabitants of the type $\text{F-Alg}((C,g),(D,h))$ are equal, and that there indeed is an inhabitant. The main theorem of [**AGS12**] says that a type satisfies the rules of an inductive type if and only if it is a homotopy initial algebra. So, this gives a characterization of inductive types using homotopy initial algebras, and a characterization of inductive types in categories.

This can be extended to some higher inductive types [**Soj15**]. Here the definition of *W-suspensions* is given.

Definition 6.3.4 (W-suspension). Suppose, we have types A, C : Type, and suppose that $x : A \vdash B[x]$: Type, and suppose that we have functions $l, r : C \to A$. Then we define $W(A, B, C, l, r)$, which we abbreviate by W, to be the higher inductive type generated by the constructors

$$\text{sup} : \prod a : A.(B[a/x] \to W) \to W,$$

$$\text{cell} : \prod c : C \prod t : B[l\,c] \to W \prod s : B[r\,c] \to W. \text{sup}(l\,c, t) \rightsquigarrow \text{sup}(r\,c, s). \qquad ⌟$$

This is based on W-types as defined in Section 2.2.7. The constructors are inhabitants of A, and the arity and parameters are from B. So, if we have a constructor a, and we know all the arguments and parameters, then we can make a new inhabitant. This is said by the rule of sup. Equalities between inhabitants can be added with cell. The type C contains the names of paths, and the functions l and r give the constructor of the left and right endpoint of the paths respectively. For these types the introduction rule, elimination rule and computation rules can be given. Furthermore, one can prove that the rules are satisfied by some type if and only if it is a homotopy initial algebra.

This definition and theorem are very beautiful and insightful. It provides a sensible argument for the fact that higher inductive types are indeed the homotopical analogue of inductive types. However, they only allow 1-constructors, so it cannot be used to give a definition of the torus. Furthermore, recursion is not allowed, but for truncations a similar result can be proved. The semantics of these types are given as homotopy initial algebras, and that does not explicitly construct interpretations of higher inductive types.

Another important article discussing higher inductive types is [**LS12**]. This article discusses the semantics of higher inductive types, and shows that these exist in a wide class of model categories. However, again they only allow 0-constructors and 1-constructors. Furthermore, they specify what the syntax should give for the semantics to work, but not what the syntax should be.

Related to higher inductive types are *hubs and spokes* [**Uni13**]. This construction allows us to specify higher inductive types by just giving 1-constructors. To give a path $p \rightsquigarrow q$ in a type T, it is sufficient to give a path $p \circ q^{-1} \rightsquigarrow \text{refl}$. For simplicity, we assume that p and q are paths between terms x and y. Note that we have a map $f : S^1 \to T$ sending base to x and loop to $p \circ q^{-1}$. Instead of adding the rule $p \rightsquigarrow q$, we add a constructor $*$ and the rule

$$\prod x : S^1 . f\ x \rightsquigarrow *$$

We fill the circle, and this gives an equality between p and q. Similarly, this can be done for higher paths if we have spheres S^n. The advantage is that higher inductive types can be specified using just 1-constructors, but the disadvantage is that only propositional computation rules are preserved and not definitional ones.

Some recursive higher inductive types can be reduced to nonrecursive higher inductive types [**Kra16, vD15**]. One example would be the propositional truncation [**vD15**], and the argument in generalized in [**Kra16**]. To get the truncation of A as a nonrecursive higher inductive type, we need two constructions. The first one is similar to the truncation.

```
Inductive $|A|$ $(A : \textsc{Type})$ :=
  | $\iota : A \to |A|$
  | $p : \prod x : A \prod y : A . \iota\ x \rightsquigarrow \iota\ y$
```

The other one is a colimit, and for that we assume we can prove $x : \mathbb{N} \vdash A[x] : \textsc{Type}$. In that case we define

```
Inductive $\mathrm{colim}_{n:\mathbb{N}} A[n/x]$ :=
  | $\iota : \prod n : \mathbb{N} . A[n/x] \to \mathrm{colim}_{n:\mathbb{N}} A[n/x]$
  | $p : \prod n : \mathbb{N} \prod a : A[n/x] . (\iota\ n+1)\ a \rightsquigarrow (\iota\ n)\ a$
```

Then the truncation is the colimit of $|A|, |(|A|)|$, and so on. Note that this gives a clear semantics of the truncation in our interpretation of higher inductive types. It is unclear whether this is possible for all recursive higher inductive types.

We think our interpretation is related to cubical sets [**BCH14**], and that the interpretation of higher inductive types can easily be computed in cubical sets. The category of cubical sets is a presheaf model of type theory, and there are explicit definitions of the intervals in that model. Also, colimits can be computed, and that would give a formula of higher inductive types in cubical sets. It is possible to define the geometric realization of cubical sets, and then one can compare higher inductive types with their topological interpretation. A question is whether this procedure where one first interprets the higher inductive type in cubical sets and then takes the geometric realization, gives the 'right' space for concrete examples.

Bibliography

[AGS12] Steve Awodey, Nicola Gambino, and Kristina Sojakova, *Inductive Types in Homotopy Type Theory*, Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science, IEEE Computer Society, 2012, pp. 95–104.

[AW09] Steve Awodey and Michael A Warren, *Homotopy Theoretic Models of Identity Types*, Mathematical Proceedings of the Cambridge Philosophical Society, vol. 146, Cambridge Univ Press, 2009, pp. 45–55.

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber, *A Model of Type Theory in Cubical Sets*, 19th International Conference on Types for Proofs and Programs (TYPES 2013), vol. 26, 2014, pp. 107–128.

[EM45] Samuel Eilenberg and Saunders MacLane, *General Theory of Natural Equivalences*, Transactions of the American Mathematical Society **58** (1945), no. 2, 231–294.

[Hat02] Allen Hatcher, *Algebraic Topology*, Cambridge University Press, Cambridge, New York, 2002.

[Hed98] Michael Hedberg, *A Coherence Theorem for Martin-Löf's Type Theory*, Journal of Functional Programming **8** (1998), no. 04, 413–436.

[Kan58] Daniel M Kan, *Adjoint Functors*, Transactions of the American Mathematical Society **87** (1958), no. 2, 294–329.

[Kra16] Nicolai Kraus, *Constructions with Non-Recursive Higher Inductive Types*, ACM/IEEE Symposium on Logic in Computer Science (LICS), 2016.

[LB13] Daniel R Licata and Guillaume Brunerie, *$\pi_n(S^n)$ in Homotopy Type Theory*, Certified Programs and Proofs, Springer, 2013, pp. 1–16.

[LS12] Peter LeFanu Lumsdaine and Michael Shulman, *Semantics of Higher Inductive Types*, Preprint (2012).

[LS13] Daniel R Licata and Michael Shulman, *Calculating the Fundamental Group of the Circle in Homotopy Type Theory*, Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2013, pp. 223–232.

[May99] Jon Peter May, *A Concise Course in Algebraic Topology*, University of Chicago Press, 1999.

[ML78] Saunders Mac Lane, *Categories for the Working Mathematician*, vol. 5, Springer Science & Business Media, 1978.

[MLM92] Saunders Mac Lane and Ieke Moerdijk, *Sheaves in Geometry and Logic*, Springer Science & Business Media, 1992.

[Soj15] Kristina Sojakova, *Higher Inductive Types as Homotopy-Initial Algebras*, POPL, ACM, 2015, pp. 31–42.

[Uni13] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[vD15] Floris van Doorn, *Constructing the Propositional Truncation using Non-Recursive HITs*, arXiv preprint arXiv:1512.02274 (2015).

APPENDIX A

List of Rules in Type Theory

$$\Rightarrow\mathrm{F}\ \frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A \Rightarrow B : \textsc{Type}}$$

$$\Rightarrow\mathrm{I}\ \frac{\Gamma, x : A \vdash f(x) : B}{\Gamma \vdash \lambda(x : A).f(x) : A \Rightarrow B}$$

$$\Rightarrow\mathrm{E}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \Rightarrow B}{\Gamma \vdash \mathrm{apply}(f, a) : B}$$

$$\mathrm{apply}(\lambda(x : A).f(x), a) = f(a)$$

$$\times\mathrm{F}\ \frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A \times B : \textsc{Type}}$$

$$\times\mathrm{I}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

$$\times\mathrm{I1}\ \frac{\Gamma \vdash x : A \times B}{\Gamma \vdash p_1(x) : A}$$

$$\times\mathrm{I2}\ \frac{\Gamma \vdash x : A \times B}{\Gamma \vdash p_2(x) : B}$$

$$p_1(a, b) = a,\ p_2(a, b) = b$$

$$+\mathrm{F}\ \frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash B : \textsc{Type}}{\Gamma \vdash A + B : \textsc{Type}}$$

$$+\mathrm{I1}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \iota_1(a) : A + B}$$

$$+\mathrm{I2}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \iota_2(b) : A + B}$$

$$+\mathrm{E}\ \frac{\Gamma \vdash p : A + B \qquad \Gamma, x : A \vdash f(x) : C \qquad \Gamma, y : B \vdash g(y) : C}{\Gamma \vdash (\textbf{case}\ p\ \textbf{of}\ x : A\ \textbf{then}\ f(x),\ \textbf{of}\ x : B\ \textbf{then}\ g(x)) : C}$$

$$f(a) = \textbf{case } \iota_1(a) \textbf{ of } x : A \textbf{ then } f(\iota_1(a)), \textbf{ of } x : B \textbf{ then } g(\iota_1(a))$$
$$g(b) = \textbf{case } \iota_2(b) \textbf{ of } x : A \textbf{ then } f(\iota_2(b)), \textbf{ of } x : B \textbf{ then } g(\iota_2(b))$$

$$\bot\text{F}: \quad \cdot \vdash \bot : \text{Type}$$
$$\bot\text{E}: \quad C, x : \bot : \text{Type} \vdash !_C(x) : C$$

$$\top\text{F}: \quad \cdot \vdash \top : \text{Type}$$
$$\top\text{I}: \quad \cdot \vdash * : \top$$
$$\top\text{E}\ \frac{\Gamma, C : \text{Type} \vdash c : C}{\Gamma, x : \top \vdash \top\text{-rec}(c, x) : C}$$
$$\top\text{-rec}(c, *) = c$$

$$\mathbb{N}\text{F}: \quad \cdot \vdash \mathbb{N} : \text{Type}$$
$$\mathbb{N}\text{I}0: \quad \cdot \vdash 0 : \mathbb{N}$$
$$\mathbb{N}\text{I}S: \quad n : \mathbb{N} \vdash S(n) : \mathbb{N}$$
$$\mathbb{N}\text{E}\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma, x : A \vdash f(x) : A}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{rec}(a, f, n) : A}$$
$$\mathbb{N}\text{rec}(a, f, 0) = a$$
$$\mathbb{N}\text{rec}(a, f, S(n)) = f(g(n))$$

$$\textstyle\prod\text{F}\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B(x) : \text{Type}}{\Gamma \vdash \prod_{x:A} B(x) : \text{Type}}$$
$$\textstyle\prod\text{I}\ \frac{\Gamma, x : A \vdash f(x) : B(x)}{\Gamma \vdash \lambda(x : A).f(x) : \prod_{x:A} B(x)}$$
$$\textstyle\prod\text{E}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : \prod_{x:A} B(x)}{\Gamma \vdash \text{apply}(f, a) : B(a)}$$
$$\text{apply}(\lambda(x : A).f(x), a) = f(a)$$

$$\textstyle\sum\text{F}\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B(x) : \text{Type}}{\Gamma \vdash \sum_{x:A} B(x) : \text{Type}}$$
$$\textstyle\sum\text{I}\ \frac{\Gamma, a : A \vdash b : B(a)}{\Gamma \vdash (a, b) : \sum_{x:A} B(x)}$$

$$\textstyle\sum\mathrm{E}\ \frac{\Gamma \vdash p : \sum_{x:A} B(x) \qquad \Gamma \vdash C : \textsc{Type} \qquad \Gamma, x : A, y : B(x) \vdash f(x,y) : C}{\Gamma \vdash \textbf{case}\ p\ \textbf{of}\ B(x)\ \textbf{then}\ f(x,p) : C}$$

$$f(x,y) = \textbf{case}\ (x,y)\ \textbf{of}\ B(x)\ \textbf{then}\ f(x,y)$$

$$W\mathrm{F}\ \frac{\Gamma \vdash A : \textsc{Type} \qquad \Gamma, a : A \vdash B(a) : \textsc{Type}}{\Gamma \vdash W_{x:A} B(x) : \textsc{Type}}$$

$$W\mathrm{I}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : B(a) \to W_{x:A} B(x)}{\Gamma \vdash \mathrm{sup}(a,b) : W_{x:A} B(x)}$$

$$W\mathrm{E}\ \frac{\Gamma, w : W_{x:A} B(x) \vdash C(w) : \textsc{Type} \qquad \Gamma, x : A, f : B(x) \to W_{x:A} B(x), g : \prod_{y:B(x)} C(f(y)) \vdash c(x,f,g) : C(\mathrm{sup}(x,y))}{\Gamma, w : W_{x:A} B(x) \vdash \mathrm{wrec}(w,c) : C(w)}$$

$$\mathrm{wrec}(\mathrm{sup}(a,f),c) = c(a,f,\lambda y.\,\mathrm{wrec}(u(y),c))$$

$$\rightsquigarrow\mathrm{F}\ \frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x \rightsquigarrow y : \textsc{Type}}$$

$$\rightsquigarrow\mathrm{I}\ \frac{\Gamma \vdash x : A}{\Gamma \vdash \mathrm{refl}_x : x \rightsquigarrow x}$$

$$\rightsquigarrow\mathrm{E}\ \frac{\Gamma, p : x \rightsquigarrow y \vdash C(x,y,p) : \textsc{Type} \qquad \Gamma, x : A \vdash t(x) : C(x,x,\mathrm{refl}_x)}{\Gamma, p : x \rightsquigarrow y \vdash J(t,x,y,p) : C(x,y,p)}$$

$$J(t,x,x,\mathrm{refl}_x) = t$$