Department of Mathematical Sciences
UNIVERSITY OF COPENHAGEN

Magnus Baunsgaard Kristensen

Orthogonality Axioms in Type Theory

Advisor
Jesper Michael Møller

February 1, 2019

Abstract

In this thesis we give a presentation of dependent type theory, and discuss how to model it in presheaf categories. The discussion is centered around the intensional Martin-Löf type theory, touching on both the classic connection to proof theory and the more recent advances in relating such theories to homotopy theory known as homotopy type theory. The presheaf semantics are based on [8], in which the model is presented in the more general setting of categories with families; the primary focus here is hence giving presheaf categories the structure of a category with families. We extend this theory, and consequently the model of this theory, in three ways through out the thesis.

First we will discuss how to add a fix point combinator to dependent type theory without introducing inconsistencies into the theory, and how to encode the coinductive type of streams in this theory. This idea is based on guarded recursion as proposed by [13], and a variant of if it is put into programmatic practice in [1]. We provide a model for this theory with a clock irrelevance axiom in a certain presheaf category and show that clock irrelevance can be modelled as orthogonality to an object of clocks, following [5].

We then consider an implementation of an intensional identity type in cubical type theory. The presentation of cubical type theory follows [7], and in particular we implement path types as a function space from an abstract interval; the interval is here assumed to have the structure of a de Morgan algebra. Path types are given the structure of an intensional identity type using composition operations on types. The theory is modelled in the category of cubical sets, based on a presentation of cubical sets compatible with the de Morgan structure on the syntactic interval. Cubical sets support a notion of Kan fibrations, a notion known from the setting of simplicial sets, which we modify and relate to the composition structure in the theory.

Finally, we show that guarded dependent type theory is consistent with the addition of intensional identity types implemented as path types. We do this by presenting a hybrid of the two other theories, rephrasing the clock irrelevance axiom presented in [5] as a path equality of terms. We model this theory in a certain presheaf category based on the previously presented models. In particular, the modified clock irrelevance axiom is modelled using an orthogonality condition.

Contents

Introduction

Modern mathematics are formulated in terms of formulas and sets. If we want to show that some set of formulas is consistent, we would try to construct a model of the theory. Modern mathematics is generally formalized within the theory of ZFC, and we posit the existence of models of this theory in which we can construct our objects of study.

Type theory as a mathematical formalism offers an alternative to this situation. Instead of formulas and sets as the object of study, and proofs carried out in a combination of metatheory dictated by some variant of classical logic, we would consider types, terms of types and proofs that are objects of the theory. That the proofs are objects of the theory is a nice property, since it allows us to keep our reasoning in a single theory, instead of jumping between several modes of proof; this property is called proof relevance. There are reasons beyond the foundational to consider type theory. There are many variants of type theory; in this thesis we will study type theories in the style of Martin-Löf, which are also called constructive or intuitionistic type theories. Apart from the features shared by all type theory, Martin-Löf style type theory allows for so called dependent types. A basic type theory such as the λ-calculus is constructed by considering some number of base types (for instance the natural numbers), and allowing for construction such as function types, $A \to B$, and product types $A \times B$. In dependent type theory, we consider the dependent analogues of these, the Π types, or the dependent product types, and the Σ types, or the dependent sum types. We will discuss these types in detail, and see how they recover the non-dependent types of λ-calculus. The most important part of dependent type theory is the identity type, which allows for a method of identifying equal terms within the theory, as opposed to the syntactical judgemental equalities. From now on, when we say type theory, it will refer to these dependent type theories.

As with conventional mathematics, we wish to construct models of this theory, and there is a standard construction of models of basic dependent type theory. This model is an interpretation of the theory into a certain type of category, namely a category with families; we will see how to give presheaf categories the structure of a category with families and will discuss the standard interpretation of type theories in this context. Having such a model shows consistency of the theory, and in particular, if we add assumptions to our type theory, we can check whether the model is still valid to see if the new theory is consistent.

There are two important interpretations of type theory other than the interpretation as a foundation for mathematics. Type theory has a deep connection to computer science; indeed, any functional programming language can be thought of as a type theoretic construct. In particular type theory is used in the implementation of proof assistants, such as Coq and Agda, which have recently found their purpose. Among others, such assistants were used in the verification of the four-color theorem and the odd-order theorem, which have both been verified in Coq. Functional programming languages place an emphasis on the safety of the programs, and the developments of type theory can be directly implemented in such languages.

A recent development in type theory is the relation to homotopy theory. The mathematical view on type theory has the weakness that it is hard to give a definitive answer to the question of what a type actually is. We think of it informally as a collection akin to a set, but this is not terribly precise. Under the homotopy interpretation it is very easy to answer this question; types are spaces, and terms of types are points in those spaces. It is important to note that this analogy is only to the homotopic information of spaces, not the instantiation of these as either topological spaces or simplicial sets.

This interpretation becomes very natural once one realizes that the basic structures of type theory have the same structure that is present in higher category theory. For this reason, type theory is thought to be especially well suited as a foundation for homotopy theory. The homotopy theory developed in homotopy type theory is called synthetic homotopy theory, as opposed to the usual analytic homotopy theory developed in models such as topological spaces or simplicial sets.

The primary achievement of homotopy type theory is the development of a number of extensionality principles. As we discussed before, dependent type theory comes with a notion of identity types, and this can be used to make sense of identifications of types. There is a weaker notion of type equivalence, which is easier to work with, and in models restrict to isomorphisms. The univalence axiom can be added to type theory to expand the notion of identification of types to be equivalent to equivalence of types. In addition, we can show that many types of identifications can be split into smaller parts. For instance, to show that two functions are equal, it will in homotopy type theory be sufficient to show that they produce equal outputs on every input.

The soundness of this interpretation was verified by Voevodsky, when he gave a model of this type theory in simplicial sets, the model is covered in [12]. The formulation of the important univalence axiom is also due to Voevodsky.

In the first chapter we will review some basic category theory, and make explicit part of the connection between type theory, programming languages and formal logic. The second chapter of the thesis is dedicated to giving an introduction to dependent type theory as a formal language, and understanding its standard model in a presheaf category. The discussion of the theory follows [14], while the model is based on [8]. We will then, in the third chapter, give two examples of extensions of the basic theory. First we will discuss a variant of guarded recursion with multiple clocks. The goal of the discussion of this theory is to be able to add recursive definition to type theory; doing so without the concept of guarded recursion is, as we will see, unsafe, and leads to an inconsistent theory. This theory has a model in the category of covariant presheaves on a category of time objects with a close connection to the idea of multiple clocks. In particular one we will need a clock irrelevance axiom, and the corresponding notion in the model is that of orthogonality to the object of clocks. This discussion follows [5].

The second example is that of cubical type theory. Cubical type theory is an attempt at giving a constructive version of the simplicial interpretation due to Voevodsky. It adds a primitive syntactic interval to the theory, and the identity type is then explicitly identified with the type of paths, or certain maps from the interval. The proof that the simplicial model is valid is not constructive, but the verification that cubical type theory can be modelled in cubical sets *is* [7]. In addition to this, cubical type theory has new proofs of the extensionality principles posited in homotopy type theory. We will discuss how a certain technical feature of this theory relates to a Kan-like lifting condition.

In the fourth chapter we will propose a mix of the two theories, with both guarded recursion and path types. In particular, we will investigate a lifting condition similar to the orthogonality condition presented in chapter three. This is not a model that has been proposed before; we give some theorems and some conjectures. In particular, we show that this newly proposed lifting condition corresponds to the clock irrelevance axiom stated in terms of path equality.

CHAPTER 1
Preliminaries

1.1 Category theory

We will assume familiarity with basic constructions of category theory, and in particular categories, throughout this thesis. In this section we review the most important definitions and theorems for our purposes, providing full proofs only for some lesser known results.

1.1.1 Cartesian closed categories

In this subsection we review a notion which is categorically equivalent to the formalism which we will later introduce as the setting for modelling type theory categorically, namely that of locally Cartesian closed categories. We will also review some classical results of this setting. The main utility of this subsection is, that results about categorical features of the model can be shown in either setting, and are often easier to understand an work with in this one.

Definition 1.1.1
A category $\mathcal{C}$ is called Cartesian closed if it has finite products and exponential objects, i.e., if all $X, Y \in \mathcal{C}$ there exists an object Y^X, equipped with an evaluation map $\mathrm{ev} : X^Y \times Y \to X$. The evaluation map is universal in the sense that, given $Z \in \mathrm{ob}\,\mathcal{C}$ and a map $e : Z \times Y \to X$, there exists a unique map $u : Z \to X^Y$ such that

$$\begin{tikzcd}
Z \times Y \arrow[r, "u\times\mathrm{id}_Y"] \arrow[dr, "e"'] & X^Y \times Y \arrow[d, "\mathrm{ev}"] \\
 & X.
\end{tikzcd}$$

We will use the shorthand CCC for a Cartesian closed category, and will occasionally call the exponential objects internal Hom objects. ○

It is well known that exponentiation extends to a functor, and that this functor is right adjoint to the product functor. Indeed, the existence of adjoints to each product functor $X \times (-)$ is equivalent to the definition we gave. We also note that CCC's are the preferred setting for modelling non-dependent type theories. These theories are simply obtained from dependent ones in which the dependent product is replaced by the non-dependent function space, and the dependent sum is replaced by the non-dependent product. Here we can see that we can straightforwardly model function space types as exponential objects and products of types as the products of their corresponding interpretations. The fact that these theories can be modelled in CCC's hint at the fact that dependent type theory might be modelled in a refinement of these, which is true up to equivalence of categories.

Definition 1.1.2
A category $\mathcal{C}$ with a terminal object is locally Cartesian closed if for each object of $\mathcal{C}$, X, the slice category $\mathcal{C}_{/X}$ is CCC. ○

Note that any LCCC is automatically a CCC, since $\mathcal{C}$ is equivalent to the slice over the terminal object. One can leave out this requirement from the definition, which would mean that this implication fails in general.

Now we consider an equivalent characterization of LCCC's in terms of the pullback functor, to which we first commit a remark.

Remark 1.1.3 (Change of base functor)
Given a morphism $f : A \to B$ in a category with pullbacks, $\mathcal{C}$, we get a functor $\mathcal{C}_{/B} \to \mathcal{C}_{/A}$ given simply by taking the pullback along f on objects, which we denote by f^*X. Since the pullback is only defined up to isomorphism, we assume that a choice of pullback is made for each pullback diagram.

A morphism $h : (X, p) \to (Y, p')$ in $\mathcal{C}_{/B}$ gives us a diagram

$$\begin{tikzcd}
f^*X \arrow[r, "j"] \arrow[dd, bend right, "g"'] & X \arrow[d, "h"] \\
f^*Y \arrow[r] \arrow[d] & Y \arrow[d, "p'"] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

in which the outer square is a pullback, since the leftmost map is equal to p. The maps g and hj are then compatible maps into Y and A, which induces a map $f^*X \to f^*Y$, which is equal to the map g when postcomposed by the projection $f^*Y \to A$, or in other words, a map $f^*X \to f^*Y$ over A, so a morphism in the category $\mathcal{C}_{/A}$. As is convention, we denote the functor by f^* on both objects and morphisms.

We note here that the pullback functor has a left adjoint, given by composition. More specifically the functor taking $X \to A$ to $X \to A \to B$, and given by the identity on morphisms. To see that this functor is indeed left adjoint to the pullback functor, note that maps in $\mathcal{C}_{/B}$ from an object (X, p), where p factors over A as $f\hat{p}$ for some $\hat{p} : X \to A$, to (Y, p') we have a commuting diagram

$$\begin{tikzcd}
X \arrow[r] \arrow[d] & Y \arrow[d] \\
A \arrow[r] & B
\end{tikzcd}$$

which induces a unique map $X \to f^*Y$ compatible with the respective projections to A by construction. Of course this also works in the other direction; a map $X \to f^*Y$ compatible with the given projection $X \to A$ yields a map $X \to Y$ over B. These operations are clearly natural, so this gives us the required natural isomorphism of Hom-sets. ○

As we saw in the remark, the pullback functor always has a left adjoint.

Lemma 1.1.4
Let $\mathcal{C}$ be a category with a terminal object. Then $\mathcal{C}$ is LCCC if and only if $\mathcal{C}$ has finite limits, and for each morphism $f : A \to B$, the change of base functor $f^* : \mathcal{C}_{/B} \to \mathcal{C}_{/A}$ has a right adjoint $\Pi_f : \mathcal{C}_{/A} \to \mathcal{C}_{/B}$.

Proof. Suppose that $\mathcal{C}_{/X}$ is CCC for each X in the category. We need to show that for each $f : A \to B$ in $\mathcal{C}$ the the change of base functor $f^* : \mathcal{C}_B \to \mathcal{C}_A$ has a right adjoint $\Pi_f : \mathcal{C}_{/A} \to \mathcal{C}_{/B}$, and that $\mathcal{C}$ has finite limits.

The category $\mathcal{C}$ has a terminal object, so it remains to prove the existence of pullbacks. Since $\mathcal{C}_{/A}$ is CCC it in particular has products, but products in $\mathcal{C}_{/A}$ are pullbacks in $\mathcal{C}$.

It remains to show that we can define a functor Π_f which is right adjoint to f^*. Consider $f : A \to B$ as a morphism in $\mathcal{C}$, and consider the moprhisms $x : X \to A$ and $y : Y \to B$ in $\mathcal{C}$ as objects of $\mathcal{C}_{/A}$ and $\mathcal{C}_{/B}$ respectively. We define Π_f on objects by taking the pullback over B:

$$\begin{tikzcd}
\Pi_f X \arrow[r] \arrow[d, "\Pi_f x"] & X^f \arrow[d, "x^f"] \arrow[rdd, bend left, "(f\circ x)^f"] & \\
B \arrow[r, "\iota"] \arrow[rrd, bend right, "\mathrm{id}_B"] & A^f \arrow[rd, "f^f"] & \\
& & B
\end{tikzcd}$$

where $\iota : \mathrm{id}_B \to f^f$ is the exponential transpose $\mathrm{id}_f : f \to f$ in $\mathcal{C}_{/B}$. Now define $\Pi_f x = \iota^* x^f$, with transposes taken in $\mathcal{C}_{/B}$ and the pullback ι^* computed in $\mathcal{C}$. This application, being an iterated application of functors, is functorial in x, hence Π_f defines a functor $\mathcal{C}_{/A} \to \mathcal{C}_{/B}$. Note that we overload the notation slightly as is often the case in over categories, denoting both the action on the object X over A and the morphism x by Π_f.

It remains to show that we have the desired adjunction; specifically, we want an isomorphism,

$$\mathrm{Hom}_{\mathcal{C}_{/A}}(f^* y, x) \cong \mathrm{Hom}_{\mathcal{C}_{/B}}(y, \Pi_f x),$$

natural in x and y. By the universal property of the pullback, morphisms $h : y \to \Pi_f x$ in $\mathcal{C}_{/B}$ correspond naturally with pairs of morphisms

$$h_1 : (y : Y \to B) \to ((f \circ x)^f : X^f \to B) \quad \text{and} \quad h_2 : (y : Y \to B) \to (\mathrm{id}_B : B \to B),$$

such that $x^f \circ h_1 = \iota \circ h_2$. Note here that this forces the second morphism, h_2, to be y, since id_B is terminal in $\mathcal{C}_{/B}$.

Using the adjunction product exponential adjunction, the morphism h_2 corresponds to the projection $f^* y : f^* Y \to A$, so the morphism $h : y \to \Pi_f x$ corresponds to the morphisms $k : f^* Y \to X$, such that $x \circ k = f^* y$. Diagrammatically. this can be expressed as

$$\begin{tikzcd}
f^* Y \arrow[r, "k"] \arrow[d, "f^* y"] & X \arrow[d, "x"] \arrow[rdd, bend left, "f\circ x"] & \\
A \arrow[r, "\mathrm{id}_A"] \arrow[rrd, bend right, "f"] & A \arrow[rd, "f"] & \\
& & B.
\end{tikzcd}$$

But such a morphism k is precisely a morphism $f^* y \to x$ in $\mathcal{C}_A$, which establishes the required natural bijection, so Π_f is right adjoint to f^*, as required.

Suppose that the existence of Π_f for each f, which is right adjoint to f^*, and the existence of all finite limits in $\mathcal{C}$. We need to prove that $\mathcal{C}_{/A}$ is a CCC. First note that $\mathcal{C}_{/A}$ has id_A as a terminal object. Given objects $x : X \to A$ and $y : Y \to A$, their product in $\mathcal{C}_{/A}$ exists, because it is the pullback in $\mathcal{C}$ which has all finite limits by assumption. For all $x : X \to A$, the product functor $(-) \times x : \mathcal{C}_{/A} \to \mathcal{C}_{/A}$ is precisely the composition

$$\Sigma_x x^* : \mathcal{C}_{/A} \xrightarrow{x^*} \mathcal{C}_{/X} \xrightarrow{\Sigma_x} \mathcal{C}_{/A},$$

but $\Sigma_x \dashv x^* \dashv \Pi_x$, and hence $\Sigma_x x^* \dashv \Pi_x x^*$, so we can define the exponentials as $y^x = \Pi_x x^* y$ for all $y : Y \to A$ in $\mathcal{C}_{/A}$. □

One might wonder if there is always a right adjoint to base change functor. This is not the case, and below we provide a simple counterexample.

Lemma 1.1.5
Cat *is a CCC, but not a LCCC.*

Proof. That Cat is a CCC is essentially well known. The exponential objects are the functor categories, wherein morphisms are natural transformations, so it remains only to show that we have a universal evaluation map. We leave out this verification here.

We now consider an example from [11]. Consider the following categories $\mathcal{C}$ and $\mathcal{D}$, given as

$$0 \xrightarrow{f} 2 \qquad 0 \xrightarrow{g} 1 \xrightarrow{h} 2.$$

respectively. Consider $\mathbb{F} \cong \mathcal{C} + \mathcal{C}$,

$$a \xrightarrow{k} b \qquad c \xrightarrow{l} d$$

Let $F : \mathcal{C} \to \mathcal{D}$ be given as $f \mapsto h \circ g$, and let $G : \mathbb{F} \to \mathcal{D}$ be given as $k \mapsto g$ and $l \mapsto h$. Now define the discrete category $\mathcal{K}$ of 0, 1, and 2, and functors $U, V : \mathcal{K} \to \mathbb{F}$, as

$$U(0) = a, U(1) = b, U(2) = d \quad \text{and} \quad V(0) = a, V(1) = c, V(2) = d.$$

It is clear that G is the coequalizer of U and V, but F^*G is not a coequalizer, hence F^* does not preserve colimits, hence it can't have a right adjoint by the adjoint functor theorem. □

The goal of this thesis is to model certain features of type theories by orthogonality in CCC's, which we now define.

Definition 1.1.6
Consider two morphisms, $p : X \to Y$ and $r : Z \to W$, in a category. These morphisms are said to be orthogonal if for every square we have a unique filler:

$$\begin{tikzcd} X \arrow[r] \arrow[d, "p"] & Z \arrow[d, "r"] \\ Y \arrow[r] \arrow[ur, dashed] & W \end{tikzcd}$$

○

We are interested not in this relation on morphisms, but in the following derived notion.

Definition 1.1.7
A morphism $p : X \to Y$ is orthogonal to an object Z, if it is orthogonal to each projection of the form $\pi_W : Z \times W \to W$. Diagrammatically, we require a unique diagonal filler in each square of the following form:

$$\begin{tikzcd} Z \times W \arrow[r] \arrow[d, "\pi_W"] & X \arrow[d, "p"] \\ W \arrow[r] \arrow[ur, dashed] & Y \end{tikzcd}$$

○

Example 1.1.8
There are two important examples of how to obtain morphisms orthogonal to some object.

- All isomorphisms are orthogonal to any object. To see this, note that the desired lift can be obtained by composition with the inverse of the isomorphism.
- If $p : A \to B$ and $q : B \to C$ are both orthogonal to X, then so is qp. To see this, note that we can obtain the desired lift iteratively, first obtaining the dashed arrow, and then the dotted in the following diagram, with uniqueness of the lift following from uniqueness at each stage.

$$\begin{tikzcd}
W \times X \arrow[r] \arrow[dd, "\pi_W"'] & A \arrow[d, "p"] \\
 & B \arrow[d, "q"] \\
W \arrow[r] \arrow[ur, dashed] \arrow[uur, dotted] & C
\end{tikzcd}$$

○

There are two results on orthogonality specific to the setting of CCC's.

Lemma 1.1.9
Let $\mathcal{C}$ be a CCC, and consider some morphism, $p : A \to B$, which is orthogonal to some collection of objects, X_i. Then p is also orthogonal to any connected colimit of these objects.

Proof. Let $X \cong \lim_i X_i$ be a connected colimit. Since the product in a CCC has a right adjoint, it will preserve colimits, so that $X \times Z \cong (\lim_i X_i \times Z)$, and a map $X \times Z \to A$ is a compatible collection of maps $Z \times X_i \to A$. We get a unique lift in each diagram of the form

$$\begin{tikzcd}
Z \times X_i \arrow[r] \arrow[d, "\pi_W"] & A \arrow[d, "p"] \\
Z \arrow[r] \arrow[ur, dashed] & B
\end{tikzcd}$$

by assumption, although it may differ in i a priori. We consider first the case of X_i and X_j that are connected directly by a morphism $X_i \to X_j$. Here we get a diagram

$$\begin{tikzcd}
Z \times X_j \arrow[rrd, bend left] \arrow[dd, bend right] & & \\
 & Z \times X_i \arrow[ul] \arrow[r] \arrow[d, "\pi_W"] & A \arrow[d, "p"] \\
 & Z \arrow[r] \arrow[ur, dashed] & B
\end{tikzcd}$$

And since the triangles commute, the lift obtained for X_j is also a lift for X_i, and by uniqueness of of the lifts they must coincide. This single lift along each for each of the X_i's, let us denote it by h, is thus a strong candidate for a lift in the full diagram.

$$\begin{tikzcd}
Z \times X \arrow[r] \arrow[d, "\pi_W"] & A \arrow[d, "p"] \\
Z \arrow[r] \arrow[ur, "h"] & B
\end{tikzcd}$$

The only thing to check here is whether the upper triangle commutes, and since our colimit commutes with product, the projection is the colimits of level-wise projections. But we have level-wise commutativity of the upper triangle by assumption, so the upper triangle commutes when the map $Z \times X \to A$ is the colimit of the maps described above, and hence it commutes. □

Lemma 1.1.10

Consider a CCC, an object X in this category, and a morphism, $p : A \to B$. Then p is orthogonal to X if and only if the following square is a pullback

$$\begin{tikzcd}
A \arrow[r, "c_A"] \arrow[d, "p"] & A^X \arrow[d, "p^X"] \\
B \arrow[r, "c_B"] & B^X
\end{tikzcd}$$

wherein the morphisms c_A and c_B are given by the exponential transposes of the projections $A \times X \to A$ and $B \times X \to B$.

Proof. Assume that p is orthogonal to X, and consider a pair of maps $h_1 : W \to B$ and $h_2 : W \to A^X$ such that $c_B h_1 = p^X h_2$. We can transpose this commuting square to obtain

$$\begin{tikzcd}
W \times X \arrow[r, "\overline{h_2}"] \arrow[d, "h_1 \times \mathrm{id}_X"] & A \arrow[d, "p"] \\
B \times X \arrow[r, "\pi_B"] & B
\end{tikzcd}$$

noting furthermore that $\pi_B(h_1, \mathrm{id}_X) = h_1 \pi_W$, where $\pi_B : B \times X \to B$ is the projection. This means that we get a unique lift $W \to A$, which makes the whole diagram commute. To summarize, we have a commuting diagram as on the left, and the only remaining thing to check is whether the top triangle on the right commutes

$$\begin{tikzcd}
W \times X \arrow[r, "\overline{h_2}"] \arrow[d, "\pi_W"] & A \arrow[d, "p"] \\
W \arrow[r, "h_1"] \arrow[ur, "h"] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
W \arrow[rrd, bend left, "h_2"] \arrow[rd, "h"] \arrow[rdd, bend right] & & \\
& A \arrow[r, "c_A"] \arrow[d, "p"] & A^X \arrow[d] \\
& B \arrow[r] & B^X
\end{tikzcd}$$

To see this we use the same trick as before, noting that $\overline{h_2} = h\pi_W = \pi_A(h, \mathrm{id})$. This map transposes simply to $c_A h$, implying in turn that $h_2 = c_A h$ as desired.

Since the second part of the proof is similar, I will leave out the diagrams. Assume that the square is a pullback, and consider some commuting diagram

$$\begin{tikzcd}
X \times Z \arrow[r, "f"] \arrow[d, "\pi_Z"] & A \arrow[d, "p"] \\
Z \arrow[r, "g"] & B
\end{tikzcd}$$

As above, we factor $g\pi_Z$ as $\pi_B(g, \mathrm{id}_X)$, and transpose the diagram thus obtained. This gives us a map $Z \to A$ obtained from the universal property of pullbacks, which also serves as the lift in the original square. □

The property of being orthogonal to an object is closed under various operations on morphisms. We have already seen that morphisms orthogonal to X are closed under composition. Now we will show that they are closed under dependent product and pullback, provided that the category is a LCCC.

Proposition 1.1.11
Let $\mathcal{C}$ be a LCCC and let X be an object of $\mathcal{C}$. If $p : A \to B$ is orthogonal to X, then so is the dependent product of p by any $f : B \to C$, and the pullback of p for any $f : C \to B$.

Proof. To see that orthogonality is stable under pullbacks, let $f : C \to B$, and consider the diagram

$$\begin{tikzcd}
f^*A \arrow[r] \arrow[d] & A \arrow[r, "c_A"] \arrow[d, "p"] & A^X \arrow[d, "p^X"] \\
C \arrow[r, "f"] & B \arrow[r, "c_B"] & B^X
\end{tikzcd}$$

Noting that the outer square is a pullback square by the pullback lemma and the fact that each of the inner squares are pullbacks. By the naturality of $c_{(-)}$, which is a consequence of the proof of 1.1.10, the outer square of the above diagram is the same as the outer square of the following:

$$\begin{tikzcd}
f^*A \arrow[r, "c_{f^*A}"] \arrow[d] & (f^*A)^X \arrow[r, "\pi_A^X"] \arrow[d, "p"] & A^X \arrow[d, "p^X"] \\
C \arrow[r, "c_C"] & C^X \arrow[r, "f^X"] & B^X
\end{tikzcd}$$

We can conclude that the outer square is a pullback, meaning that the pullback of p along f is also orthogonal to X by 1.1.10.

Consider now $\Pi_f p$, for some $f : B \to C$, and a diagram as follows:

$$\begin{tikzcd}
X \times Y \arrow[r] \arrow[d] & \Pi_f A \arrow[d] \\
Y \arrow[r] & C
\end{tikzcd}$$

Note first that be the universal property of the dependent product, fillers in this diagram are in one-to-one correspondence with fillers of the diagram

$$\begin{tikzcd}
(X \times Y) \times_C B \arrow[r] \arrow[d, "\pi_Y \times_C B"] & A \arrow[d] \\
Y \times_C B \arrow[r, "\pi_B"] & B
\end{tikzcd}$$

Here the top horizontal mp is given by transposing the top horizontal map of the original square, and the bottom horizontal map is given by pullback along f. Here we will need a small lemma, which we prove afterwards, namely that we have an isomorphism $(X \times Y) \times_C B \simeq X \times (Y \times_C B)$. This means that orthogonality of p to X that we have a unique lift in this new diagram, and hence also in the old one. □

We now prove the lemma, with a slight extension that we used implicitly in the above proof. The reason to lift this minor detail to a lemma is that we will need it again towards the end of the thesis.

Lemma 1.1.12
We have an isomorphism $(X \times Y) \times_C B \simeq X \times (Y \times_C B)$. Furthermore, the projection onto $Y \times_C B$ from the right hand side is equal to the pullback by B over C functor applied to the projection $X \times Y \to Y$.

Proof. We get the isomorphism by the pullback lemma and considering the following diagram:

$$\begin{tikzcd}
X \times (Y \times_C B) \arrow[r, "\pi_Y \times \mathrm{id}_X"] \arrow[d, "\pi_{Y \times_C B}"] & X \times Y \arrow[d, "\pi_Y"] \\
Y \times_C B \arrow[r, "\pi_Y"] \arrow[d, "\pi_B"] & Y \arrow[d] \\
B \arrow[r] & C
\end{tikzcd}$$

We are being a little relaxed with the notation for our projections here. The projection in the top map is the composition of projecting off the X, and then using the projection from the pullback onto Y. Each of the inner squares here are pullbacks, hence the outer one is as well.

The equality of the maps follows from writing out the diagram which represents the map $(X \times Y) \times_C B \to X \times (Y \times_C B)$. □

1.1.2 Presheaf categories

In this section we will review some results on presheaf categories. Most will be well-known, and stated without proof, but some lesser known ones are presented with proof.

The collection of sets and maps of sets form a category, which we denote by Set. For any two categories, $\mathcal{C}$ and $\mathcal{D}$, we can form the functor category, $\mathcal{C}^{\mathcal{D}}$, in which the morphisms are natural transformations. From now on we assume that $\mathcal{C}$ is locally small. The specific functor category $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ is called the category of presheaves on $\mathcal{C}$, with the category $\mathsf{Set}^{\mathcal{C}}$ being the category of covariant presheaves. In either case, we will often call the image of a morphism in $\mathcal{C}$ a restriction map. Two important kinds of presheaves are the representable and corepresentable presheaves, also known as the Hom-functors. These take an object to $\mathrm{Hom}(-, c) = y^c$ or $\mathrm{Hom}(c, -) = y_c$ respectively, and take morphisms to post- and precomposition respectively. Since Hom is functorial in both variables, $y_{(-)}$ and $y^{(-)}$ form functors. Each of these are embeddings of categories, and we will refer to $y_{(-)} : \mathcal{C} \to \mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ as the Yoneda embedding and $y_{(-)}$ as the covariant Yoneda embedding. Note that both Yoneda embeddings are actually embeddings of categories. This justifies the definition of the presheaf categories as the free cocompletion of $\mathcal{C}^{\mathsf{op}}$ and $\mathcal{C}$ respectively.

Perhaps the single most important result of category theory is a statement about these functors known as the Yoneda lemma.

Theorem 1.1.13 (The Yoneda lemma)
Consider a presheaf on some category $X \in \mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$, and an object $c \in \mathcal{C}$. We then have an isomorphism

$$\mathrm{Nat}(y_c, X) \cong X(c)$$

which is natural in both X and c.

We have only stated the Yoneda lemma for contravariant presheaves, but of course the formally dual covariant Yoneda lemma also holds.

The reason we are interested in presheaves in this thesis, is that they are examples of both CCC's and LCCC's, which we now show.

Proposition 1.1.14
For any locally small category $\mathcal{C}$, the category $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ is Cartesian closed.

Proof. Using the Yoneda lemma and the product/exponentiation adjunction, we get the description

$$X^Y(c) \cong \mathrm{Hom}(\mathrm{Hom}(-, c), X^Y) \cong \mathrm{Hom}(Y \times \mathrm{Hom}(-, c), X).$$

It suffices to check that X^Y defined here is right adjoint to the product in Y, which we leave out here. □

As mentioned, presheaf categories are also locally Cartesian closed, but to see this we will first review the category of elements construction. Since this construction arises naturally at a later point in the thesis, we end this discussion simply by stating the result.

Proposition 1.1.15
For any small category, $\mathcal{C}$, the category of presheaves, $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$ is an LCCC.

Later we will see how this property can be used to reason about models of type theory.

1.2 Decidability and implementability

In this section we briefly discuss the concept of decidability, and the correspondence between type theory and programming in general.

Decidability is a property that a logical expression can have relative to a satisfaction predicate. In general we can talk of external or internal decidability of a statement φ. If φ is decidable externally, this means that we have either a proof of φ or a proof of $\neg\varphi$. Internal decidability means that $\varphi \vee \neg\varphi$ is derivable in the system. In classical logic, every statement is internally decidable since this is simply the law of the excluded middle. It is not, however, the case that every statement of classical logic are externally decidable, and this is exactly the statement of the Gödel incompleteness theorems. The critical distinction between the two is whether decidability is a statement in the metatheory or the theory itself.

The case we are interested in is external decidability relative to intuitionistic or constructive logic. Here we know that the law of excluded middle does not hold in general, so that there must be some statements that are not decidable. Intuitionistic predicate logic has the disjunction property in the empty context, meaning that a proof of $p \vee q$ is either a proof of p or a proof of q when we do not make uncancelled assumptions. Therefore, external and internal decidability of such statements are equivalent. This does not however extend to every statement of constructive theories, so there is still a distinction. From now on, when we say decidable we mean externally decidable in a constructive system.

There is another notion of decidability, and as we shall see it is equivalent to the one described above. To a computer scientist, decidable statements, or more generally decidable functions where statements are in correspondence with functions into $\{0, 1\}$, are those that can be computed by some Turing machine. There is a strong connection between constructive proof theory and type theory, and among such connective results we have the Curry-Howard isomorphism described in [10]. We will see this correspondence in the language of this thesis at a later point. This isomorphism gives us a correspondence between correct proofs of statements, and certain terminating programs specified as inhabitants of some type. We can directly think of terms of types as programs; indeed this is the idea upon which functional programming languages are built. There are examples functional, Turing complete languages, so we have a direct connection between constructively valid proofs and terminating programs computing some function.

Programming languages such as Agda and Coq are in a sense made for implementing this isomorphism, as proof assistants. Each of them implement a dependent type theory and thus provide an instantiation of a proof relevant theory, in which we can reason about programs and their specifications, and the proofs that these specifications are correct in the same system.

In this thesis we will discuss three extensions of usual type theory, with one extension adding extensionality principles which will ease the reasoning in the system, one adding a refinement of the fix point combinator which is employed by languages such as Haskell to achieve Turing completeness, and one combination of the two, which will enjoy some of the properties from each of the others.

CHAPTER 2

Type theory

This chapter will cover the syntax of a basic dependent type theory and a model in the framework of categories with families. We will also make explicit how to give presheaf categories the structure of a category with families. In the syntactical part we provide motivation and interpretations for each type former that we make use of in the thesis. For further discussion of the syntactical side of type theory, see [14], on which the discussion here is based.

The semantics that we consider here were pioneered in [8]. For this thesis, the most important part of this chapter is to see that we can restrict this model in a certain way.

2.1 Syntax

Type theory is a deductive system, in which we arrive at *judgements* via some collection of *inference rules*. Before describing type theory, we recall some proof theory, to first see the concepts of type theory in a more familiar setting. Most mathematicians are at least somewhat familiar with the deductive system of propositional logic, although they tend not to think of its formal presentation as a deductive system. In propositional logic we can ask whether a certain formula, φ, is true under some assumptions $\Gamma = \varphi_0, \varphi_1, \ldots$, and if the answer is positive and we have a proof, we can make the judgement that $\Gamma \vdash \varphi$. To arrive at such a judgement we have to give a proof, and to give a proof we must construct the deductive system in which it can be expressed. Here we present one inference rule of the natural deduction system:

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

This inference rule is called the *introduction rule* for $\wedge$ expresses the fact that if φ follows from Γ, and ψ follows from Γ, then also $\varphi \wedge \psi$ follows from Γ. There are also two dual rules to this one, expressing the fact that from $\varphi \wedge \psi$ we can conclude either φ or ψ alone, and these are called the *elimination rules*. One should think of $\varphi \wedge \psi$ containing some amount of information; the introduction rule allows us to package all the information in a single expression, and the elimination rules allow us to extract each piece of information from the expression at a later time. Implicit in this are of course the judgements that at each stage everything we are working with are well formed formulas, or propositions, and this is another judgement of propositional logic; that a certain expression is a proposition.

The above system is a syntactical description of propositional logic. In the rest of this section, we will give a syntactical description of type theory, which will enable us to answer some questions of type theory, much in the same way that proof theory allows us to answer some questions about propositional logic [1].

Type theory is very similar to the proof theory described above, although the objects of study and the judgements that we can make relating them are slightly different. In type theory the objects we concern ourselves with are *types*, which we denote by capital letters

[1] By the soundness theorem and Gödel's completeness theorems, all questions of first-order logic are answerable both in this proof theoretic setting, and in the model theoretic setting.

$A, B, C, \ldots$. These can play either the role of propositions or sets (and later on, spaces), and each perspective is important. For now we will think of types as collection of things, which are all the same kind of thing. For example we might have the type of natural numbers $\mathbb{N}$, and this collection should then contain exactly the familiar natural numbers (or some equivalent description of them). Syntactically types are inhabited by *terms*, which we denote by lowercase letters, $t, s, \ldots$. Terms specify certain object of the type in the familiar ways; for instance if $f : A \to B$ (read as f has the type of a function from A to B), and $t : A$, we can construct a term of B by function application: $f(t) : B$. The two things missing are now the contexts, and the variables. If we have a type, we might declare a variable of that type, in language as "let x be a natural number...", and in type theory as $x : \mathbb{N}$. A context consists of a finite list of variable declarations $x_0 : A_0, x_1 : A_1, \ldots, x_n : A_n$, where the variables do not occur more than once, and each A_i is a type, at least in the presence of all the previous variables.

So now that we know the objects of study, let us consider the judgements that can be made. Type theory has following kinds of judgements:

- Judgements of the form $\Gamma \vdash$, indicating that Γ is a valid context.
- Judgements of the form $\Gamma \vdash A\,\mathbf{type}$ indicating that A is a valid type in the context Γ.
- Judgements of them form $\Gamma \vdash t : A$ indicating that a term t is of the type A in the context Γ.

Note that the two last judgements rely on some already established judgements for them to make sense. To consider judgement $\Gamma \vdash A\,\mathbf{type}$, and whether we can make this judgement or not, we first have to have made the judgement $\Gamma \vdash$. In other words; before we discuss the question of whether something is a type in a certain context, we have to first establish the fact that we have a valid context in which to discuss the question. Furthermore, these judgements allow for the question of typehood to depend on the context, so that some assumptions might be needed for A to be a type. Allowing this dependence is the basis for dependent type theory, and what separates us from the animals... errr, i mean... what separates dependent type theory from type theories like the simply type lambda calculus.

We are still missing three kinds of judgements, each concerned with the definitional equality of certain objects.

- We can make judgements of the form $\vdash \Gamma \equiv \Delta$, expressing the fact that Γ and Δ are definitionally equal contexts.
- We can make judgements of the form $\Gamma \vdash A \equiv B$, expressing the fact that A and B are definitionally equal types in the context Γ.
- We can make judgements of the form $\Gamma \vdash t \equiv t' : A$, expressing the fact that t and t' are definitionally equal terms of type A.

Now that we know the kinds of judgements that can be made, let us consider some examples of inference rules. We gave before an informal description of contexts as lists of variables, where each variable has a type which may depend on the context up to that point. Formally, we are defining context inductively by the following two rules:

$$\frac{}{.\vdash} \qquad \frac{\Gamma \vdash A\,\mathbf{type} \qquad x \notin \Gamma}{\Gamma, x : A \vdash}$$

Here the left rule expresses the fact that the empty list is a context. The right rule says that whenever we have a context Γ, a type in that context A, and a fresh variable name x, we may form a new context declaring the variable x of type A after all the other declaration of Γ, forming the context $\Gamma, x : A$. Since we will not (for the time being) give more inference rules for forming contexts, we now see that every context is precisely of the form we described earlier.

Right now we are able to make judgements of definitional equalities, but there is nothing that tells us how it might act, or how to obtain them. An obvious trio of rules to consider is then the rules that enforce reflexivity, transitivity, and symmetry of definitional equality as a relation. We will of course want these rules for all of types, context, and terms, but we render only the rules for contexts, leaving the rest for the reader to work out.

$$\frac{}{\vdash \Gamma \equiv \Gamma} \qquad \frac{\vdash \Gamma \equiv \Delta}{\vdash \Delta \equiv \Gamma} \qquad \frac{\vdash \Gamma \equiv \Gamma' \qquad \vdash \Gamma' \equiv \Gamma''}{\vdash \Gamma \equiv \Gamma''}$$

We will encounter another large class of inference rules for definitional equalities once we start introducing type formers. These will be constructed in such a way that we can always be certain that definitional equality is also a congruence. As with the above, these rules will follow an apparent general pattern, and so will be left out in most cases, although we will give some of them explicitly in the case of the dependent product.

Before moving on, we need to review the notion of capture-free substitution and α-equivalence classes of expressions.

If we have some expression φ in which some variable x occurs we could consider the φ but with t in each place where x occurs. This is fine if we are substituting a specific number for x in the expression x^2, but we quickly run into problems when considering formulas of the shape $\forall x \varphi(x)$. Specifically, we might have a formula $\forall x : f(x) < y$. If we naively replace x by y in this expression, we end up with $\forall y : f(y) < y$, which is clearly a very different statement. The problem here is, that we of want to maintain the relation between φ and the variable bound be the $\forall$ quantifier. This is where the notion of α equivalence comes in. We want to consider the formulas $\forall x \varphi(x)$ and $\forall y \varphi(y)$ to be equivalent given that y is not bound in $\varphi(x)$, and this is what we call an α-equivalence of formulas. We now define the substitution of y for x in the expression t, $t[y/x]$, be considering an expression that is α-equivalent to t in which y is not bound, and then substituting naively. Note also that in each case that we are considering in type theory we cannot bind the same variable twice. Had this been possible, we might have to add something to the effect of y being bound exactly when x is in the above.

We are now ready to start introducing the type formers. To set the stage for this, we will first consider the general recipe for a type. This discussion, and many of the concrete type formers, follows A.2 in [14]. To define a type, we should give five pieces of data, which we list below.

- A *formation rule*, which specifies in which contexts we can consider the type.
- Some number of *introduction rules*, which tells us how to obtain inhabitants of the type. These will also be called constructors.
- Some number of *elimination rules*, telling us what information can be extracted from elements of the type. These will also be called eliminators.
- Some number of β-laws, which are also called computation rules, specifying the action of the eliminiator on the result of a constructor in judgemental equalities.

- Some number of η-laws, that govern how judgementally determined an inhabitant of the type is by the results of eliminators applied to them.

To give some intuition for what these kinds of rules are, we will consider the product type as an example. In this example we will also give a few of the congruence rules to get a feel for the form of such rules. When we develop the theory we will see another way to define the product type, and from then on that will be the definition which we use.

Example 2.1.1 (Product Types)
Given two types, we would like to form the product type of these two types. As a formation rule this is expressed as

$$\frac{\Gamma \vdash A\,\mathbf{type} \qquad \Gamma \vdash B\,\mathbf{type}}{\Gamma \vdash A \times B\,\mathbf{type}}$$

The basic elements of this type are the pairs, which are obtained by pairing up something from A with something from B. As an introduction rule, this is expressed as

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

If we wanted to be more formal, we could define this as a pair operation, and write $\mathrm{Pair}(a, b)$, but we will avoid this when the operation is clear.

Given an element of the product $A \times B$, we an extract an element of A and an element of B by respectively the first and second projections. As inference rules, this can be expressed as

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \mathsf{pr}_1 x : A} \qquad \frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \mathsf{pr}_2 x : B}$$

Now we know what kinds of terms inhabit the type and we know what information can be obtained from an inhabitant. This is sufficient to tell us the type of a term involving the pairing and projection operations. This kind of information is crucial in type theory, and it is part of what ensures the good computational properties of the theories we consider. At a later point, we will discuss the issue of decidability of type-checking[2]

We are now ready to consider how the constructors and eliminators interact. The thing we want, is for the first, respectively second, projection of a pair, (a, b), to just be a, respectively b. As inference rules, this is expressed as

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{pr}_1(a, b) \equiv a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{pr}_2(a, b) \equiv b : B}$$

A shorter way that one could express the above rule is $\mathsf{pr}_1(a, b) \equiv a$, leaving out typing and context assumptions. We will not do this in definitions of types, but it will be used outside definitions. This way of stating the definitional equalities will occasionally require extra assumptions, which must be stated separately.

We also want to specify to what degree an element of a pair is determined by its projections. To express this as an inference rule, we need to the pairing of the projections applied to some element of the product is the same as just that element:

[2] Type-checking is a check of what type a term has, but can also be used in other contexts. We can for example ask if a certain term "type-checks". In the product type example, we could say that $\mathsf{pr}_1 x$ type-checks when x is a pair. On the other hand, it would not type-check if x was not a pair. Instead of asking whether a given term type-checks, we can also ask if it is well-typed, which means the same thing.

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t \equiv (\mathsf{pr}_1 t, \mathsf{pr}_2 t) : A \times B}$$

This brings us to the congruence rules. First off, we should say that the type formation is congruent, meaning that we have the following inference rule:

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma \vdash B \equiv B'}{A \times B \equiv A' \times B'}$$

Similarly we want the pairing operation on equivalent terms to produce equivalent pairs, and projections of equivalent pairs to produce equivalent terms. We give only the first of these rules:

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

This concludes the formal definition of non-dependent product types. ○

Before we move into the types which make up most dependent type theories we will extend the notion of substitution in types and terms to substitutions of contexts. These can be seen as morphisms between contexts, and indeed there is a syntactic category which has as objects the contexts of a given type theory and as morphisms the context substitutions. We add the judgement form $\rho : \Gamma \to \Delta$ to our theory, signifying the fact that ρ is a substitution from Γ to Δ. We obtain the substitutions inductively from the rules

$$\frac{}{[\cdot] : \Gamma \to \cdot} \qquad \frac{\rho : \Gamma \to \Delta \qquad \Gamma \vdash t : A\rho}{\rho[x \mapsto t] : \Gamma \to \Delta, x : A}$$

Here $A\rho$ is the type A with the substitution ρ applied to it. Essentially, a context substitution, $\Gamma \to \Delta$ is an interpretation of Δ in the context Γ, giving a term in context of Γ for each variable declared in Δ which plays the role of the corresponding variable. Once we see how to identify types and propositions, the assumption $\Gamma \vdash t : A\rho$ will make sense, as it is interpreted as the constructive evidence for the context Γ being consistent with the assumption of A's truth. There are two immediate consequences of the existence of a substitution $\Gamma \to \Delta$.

- If we have $\Delta \vdash A\,\mathbf{type}$ we also have $\Gamma \vdash A\rho\,\mathbf{type}$.
- If we have $\Delta \vdash t : A$ we also have $\Gamma \vdash t\rho : A\rho$.

We will not be concerning ourselves much with syntactic context substitutions in the rest of the thesis and state the definition here only for completeness.

2.1.1 Dependent Product

When we have two sets, we can consider the set of functions between them, and the same is true for types. In dependent type theory, which is what we will study in this thesis, we have instead the more general notion of a dependent function space.

When introducing the types, we will generally rely on some pre-existing intuition, which will inform our choice of rules. Since we do not assume familiarity with type theory, and we will not be motivating it from scratch, this intuition will be the one obtained in set theory.

A dependent function in set theory is a function $f : A \to \Pi_{a \in A} B_a$, such that $f(a) \in B_a$. Its easy to see that this recovers the usual space of functions $A \to B$ in the case where the family

B_a is constant and equal to B. We will see that something very similar happens in the type theoretic case.

This will be our first fully formal definition of a type, so we will be very explicit. After this, many of the inference rules will be left implicit, and we will trust that the reader could produce them on their own.

Definition 2.1.2
If we have a type B in the context $\Gamma, x : A$ we can form the type of dependent functions $\Pi(x : A).B$. As an inference rule:

$$\frac{\Gamma, x : A \vdash B\,\mathbf{type}}{\Gamma \vdash \Pi(x : A).B\,\mathbf{type}}$$

It is important to note that x is bound in $\Pi(x : A).B$ in the same way that v is bound in the formula $\forall v\varphi$.

Elements of the dependent function type are introduced using lambda abstraction and eliminated using function application.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x : a).B} \qquad \frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash fu : B[u/x]}$$

Note that the typing judgement $\Gamma \vdash fu : B[u/x]$ is sensible, since we have by assumption that $\Gamma, x : A \vdash B\,\mathbf{type}$. Again it is the case that x becomes bound in $\lambda x.t$. There is a slight difference from before, since we are now binding a variable in a term, as opposed to binding it in a type, but these are treated the same. We skip the congruence rules for each of the above rules.

Computing a function obtained by lambda abstraction amounts to substitution in the terms; as an inference rule, this β rule can be expressed as

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \lambda x.t(u) \equiv t[u/x] : B[u/x]}$$

There is also an η rule, which can be stated as

$$\frac{\Gamma \vdash f : \Pi(x : A).B}{\Gamma \vdash f \equiv \lambda x.(fx) : \Pi(x : A).B}$$

which would in other words mean that λ abstraction recovers any function.

This concludes the definition of the dependent product; dependent products are also called Π types. ○

If λ abstraction seems confusing, recall the general discussion of substitution in terms. That discussion was set in the context of formulas with quantifiers, but the

Remark 2.1.3
There is an alternative way to state the η law for the dependent product; we will present it and show equivalence to the given presentation here, since variations of this rule will occur later on. The rule in question is

$$\frac{\Gamma \vdash f, g : \Pi(x : A).B \qquad \Gamma, y : A \vdash fy \equiv gy}{\Gamma \vdash f \equiv g : \Pi(x : A).B}$$

In other words, if, when evaluated at each point, two functions are judgementally equal, they are judgementally equal. This follows from the usual η rule by observing that $f \equiv \lambda x.(fx) \equiv \lambda x.(gx) \equiv g$, where the second judgemental equality follows from the assumption that $fx \equiv gx$, and congruence for λ abstraction. On the other hand, if we have the above rule, we want to show that $f \equiv \lambda x.(fx)$, but by the β law we have that $fx \equiv \lambda x.(fx)x$, so that the two are judgementally equal at each point, as desired.

The rule that we have given adheres closer to the intuition of a syntactic reduction rule, giving the justification for using the simpler term f instead of $\lambda x.(fx)$. This equivalent rule is closer to a uniqueness principle; a function can be defined by defining it at each point. Both perspectives are important in the wider theory, but we will only truly be appealing to the second.

We will soon encounter a different notion of equality, propositional equality, and one can formulate the above rule for that notion. This will turn out not to follow from the η law where we replace judgemental equality by propositional equality. The rule obtained from this substitution is called function extensionality, and it plays an important role in dependent type theory.

We will, as noted, encounter this rule at a later point, where we consider two types that will behave very much like the dependent product types, except the domain will not be a type, but instead a primitive of the theory. ◦

Instead of introducing function types separately, we recover them as a special case of the dependent product. If we replace the requirement $\Gamma, x : A \vdash B\,\mathbf{type}$ by the stronger assumption that $\Gamma \vdash B\,\mathbf{type}$, so that the type B does not depend on $x : A$, we will denote $\Pi(x : A.B$ simply by $A \to B$. Inhabitants of this function are simply functions from A to B, and the type of the output does not depend on the input. This follows from the fact that for $f : A \to B$ and $a : A$, we have that $fa : B[a/x]$, but since B is a type in the context $\Gamma \vdash$ (in which x does not occur), $B[a/x]$ is simply B.

2.1.2 Dependent Sum

The second dependent type former is the dependent sum. As for the dependent product, we first consider what it would like like in set theory. There it would be the coproduct $\Sigma_{a \in A} B_a$, where B_a is again a family indexed by A. Usually, one can get away with not being specific about the form of the elements in this disjoint union, but it will inform our next move in this case; an element $\Sigma_{a \in A} B_a$ is a pair (a, b) with $a \in A$ and $b \in B_a$. Similarly to the above, we see that when B_a is a constant family, this definition coincides with the product. With this we are ready for the definition of the dependent sum.

Definition 2.1.4
If we have a type B in the context $\Gamma, x : A$ we can form the dependent sum type, $\Sigma(x : A).B$. As an inference rule:

$$\frac{\Gamma, x : A \vdash B\,\mathbf{type}}{\Gamma \vdash \Sigma(x : A).B\,\mathbf{type}}$$

As with the dependent product, the variable x is bound in $\Sigma(x : A).B$.

Elements of the dependent sum are obtained by pairing of $a : A$ and $b : B[a/x]$, and eliminated by projection. As inference rules:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash (t, u) : \Sigma(x : A).B}$$

$$\frac{\Gamma \vdash x : \Sigma(x : A).B}{\Gamma \vdash \mathsf{pr}_1 x : A} \qquad \frac{\Gamma \vdash x : \Sigma(x : A).B}{\Gamma \vdash \mathsf{pr}_2 x : B[\mathsf{pr}_1/x]}$$

The only thing to note among these is the fact that the typing of the second projection will depend on the value of the first projection. Apart from this, these are the same rules as we saw in 2.1.1. This will also be the case for the β and η rules:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{pr}_1(a, b) \equiv a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{pr}_2(a, b) \equiv b : B[a/x]}$$

Again, we also want to specify to what degree an element of a pair is determined by its projections:

$$\frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash t \equiv (\mathsf{pr}_1 t, \mathsf{pr}_2 t) : \Sigma(x : A).B}$$

This concludes the definition of the dependent sum; dependent sums are also called Σ types.

○

In the formal language, the product type which we have visited before is defined as a special case of the dependent sum. As for the function spaces, we require instead that $\Gamma \vdash B\,\textbf{type}$, so that B is the same as $B[a/x]$, and thus the type of the second projection of a pair does not depend on the first projection.

2.1.3 Natural Numbers

This example will return in almost any type theory, and by its very nature, type theory will often revolve around this type. The characteristic property of the natural numbers in almost any system they are defined is the ability to do induction over them, and type theory is no exception to this rule. Since the natural numbers as a set is familiar, we do not expand on it here.

We will give the definition formally, and them discuss the implications afterwards.

Definition 2.1.5
The natural numbers are a type in any context, so the type former is very simple. The introduction rules will be familiar to most mathematicians; we have a zero element and a successor operation. In the case of the natural numbers, the formation rules can be stated more succinctly by skipping the usual requirement that $\mathbb{N}$ is a type in the relevant context.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbb{N}\,\textbf{type}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash s(n) : \mathbb{N}}$$

The remaining rules implement induction over the natural numbers. The elimination rule gives us the induction rule itself, and then we have two elimination rules to ensure that the induction gives us the desired result.

$$\frac{\Gamma, x : \mathbb{N} \vdash C\,\textbf{type} \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x : \mathbb{N}, y : C \vdash c_s : C[s(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, n) : C[n/x]}$$

Note that we decorate the induction element with an $\mathbb{N}$; this is so that we may reuse this same notation for other inductive types. The computation rules will look quite similar to the elimination rule itself; this happens often.

$$\frac{\Gamma, x:\mathbb{N} \vdash C\,\mathbf{type} \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x:\mathbb{N}, y:C \vdash c_s : C[s(x)/x]}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, 0) \equiv c_0 : C[0/x]}$$

$$\frac{\Gamma, x:\mathbb{N} \vdash C\,\mathbf{type} \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x:\mathbb{N}, y:C \vdash c_s : C[s(x)/x] \qquad \Gamma \vdash n:\mathbb{N}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, s(n)) \equiv c_s[(n, \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, n))/(n, y)] : C[s(n)/x]}$$

○

The natural numbers are an example of a closed type. This simply means that there are no freely occurring variable types in the definition of the type, and it is the reason for the slightly more succinct introduction rule.

The elimination rule for the natural numbers and the associated computation rules are complicated to look at, but they have natural interpretations, just as the ones for Π and Σ types. Before we tackle the general case, we consider the example of defining a sequence by recursion.

Example 2.1.6
We will here consider the special case of the elimination rule for the natural numbers where C is $\mathbb{N}$ itself. This is the same as defining a sequence of natural numbers by recursion.

The first assumption of the rule is simply typehood of $\mathbb{N}$, which is free, and since $\mathbb{N}$ is closed, we have that $C[0/x] \equiv C[n/x] \equiv C$ for any n. We want to leave the n free, so it remains to specify a base case, c_0, and a function, c_s, which takes as input two natural numbers. We think of c_s as containing the recursion procedure, or "next step date". Strictly speaking, c_s is not a function, but simply a term with two free variables. We can think of it as a function where function evaluation is given by substituting specific values, or we can make this intuition explicit by defining $\lambda.a(\lambda.b(c_s[(a,b)/(x,y)]))$.

Now that we have a sense of what the elimination rule takes as input, let us define the sequence $0, 1, 2, \ldots$ by recursion. Clearly our base case is 0, so it only remains to define c_s, and we can take c_s to be simply $s(y)$. Now we want to verify that this indeed gives us the correct outputs, and this is where the computation rules find their use. This is exactly the point of the computation rules; we want recursion to not just define some arbitrary function, but rather the specific function which starts at c_0 and steps as c_s. The first computation rule gives us directly that $\mathsf{ind}_{\mathbb{N}}(\mathbb{N}, 0, s(y), 0) \equiv 0$. The second computation rule is not quite as useful, since it does not end up with a number. We have that $\mathsf{ind}_{\mathbb{N}}(\mathbb{N}, 0, s(y), s(n)) \equiv s(\mathsf{ind}_{\mathbb{N}}(\mathbb{N}, 0, s(y), n))$. It is easy to verify that this has the desired output for any $n:\mathbb{N}$, but if we want a formal proof we need a little more theory. Specifically, we would need identity types to even formulate the question. ○

The above example is just about the simplest example of recursion there is, apart from the vacuous case of defining constant functions by recursion. The classical example of an application of induction principle for natural numbers is the definition of addition.

Example 2.1.7
Addition is an operation on two natural numbers, so we can define it as a function $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. The first assumption of the elimination, or induction, principle is typehood of this expression, which is clear. Once again all types involved are closed, so we can ignore any substitutions made in the types themselves (although again we rely crucially on the substitutions in the terms).

In this case, the base case c_0 should tell us what adding zero looks like, which is easy; it should do nothing. Note that the type of c_0 here is simply $\mathbb{N} \to \mathbb{N}$, and we define it by $\lambda n.n$.

The step function should tell us how to deal with an expression of the form $s(n)+m$, which we clearly would like to simply be the successor of the sum, i.e., $s(n+m)$. We do have to be careful about typing here. Now the input for c_s is a number, as before, and a function $\mathbb{N} \to \mathbb{N}$. So we define $c_s \equiv \lambda(x,g).s(g(n))$, with $x : \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, implicitly using the fact that we can consider an iterated function as a function on a product. Here the g is going to be addition by some number k when we are doing the recursion.

By the elimination principle for natural numbers, we now have some element $\mathsf{ind}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, c_0, c_s, n) : \mathbb{N} \to \mathbb{N}$, which we can think of as the function $m \mapsto m+n$. Again we can only verify the most basic properties of this operation which we have defined, and that it gives the correct output on fully specified input, e.g., $2+2 \equiv 4$. Addition by zero is again zero, and by construction $s(m)+n \equiv s(m+n)$. We might want addition to be associative and commutative, but once again these questions cannot even be asked in the theory yet. ○

These preceding examples have mostly been for familiarizing the reader with the application of induction rules in type theory in general and for $\mathbb{N}$ in particular. We will return to the natural numbers later, to consider how one might give the proofs tat are as of now not possible, and to give a more application oriented example.

2.1.4 Identity Types

When we have a certain type, A, and two (not necessarily distinct) elements of this type, say $a, a' : A$, we can form their identity type, $a =_A a'$. We will leave out the subscript when it is irrelevant or clear from the context. The way to think of this type differs wildly depending on whether a certain rule is included in the theory, so we will discuss interpretations of this type after giving the definition.

Definition 2.1.8
Given a type A and $a, a' : A$, there is a type $a =_A a'$, which we will call the identity type. The formation inference rule is

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash a' : A}{\Gamma \vdash a =_A a' \,\mathbf{type}}$$

To obtain elements of this type we should know that the two inhabitants it is formed over are indeed the same, so our introduction rule will simply give us elements of $a =_A a$ for each inhabitant of the type, a. As an inference rule this is expressed as follows:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_a : a =_A a}$$

We call the element refl_a the reflector for a.

Now we need the elimination rule for the identity type. There are two equivalent presentations called path induction and based path induction; we only give path induction here, and refer the interested reader to [14] for a discussion of the equivalence and differences of the two rules.

We want our eliminator for the identity type to, given a family of types indexed by two inhabitants of a type and an inhabitant of this family at each reflector, give an element of each type in the family. The inference rule corresponding to this is

$$\frac{\Gamma, a : A, a' : A, p : a =_A a' \vdash C \,\mathbf{type} \qquad \Gamma, a'' : A \vdash c(a'') : C[(a'', a'', \mathrm{refl}_{a''})/(a, a', p)]}{\Gamma, a : A, a' : A, p : a =_A a' \vdash f(a, a', p) : C}$$

There is a clear candidate for the β law in this case. The family f should restrict to the input family c on the reflectors:

$$\frac{\Gamma, a : A, a' : A, p : a =_A a' \vdash C\,\mathbf{type} \qquad \Gamma, a'' : A \vdash c(a'') : C[(a'', a'', \mathrm{refl}_{a''})/(a, a', p)]}{\Gamma, a'' : A \vdash f(a'', a'', \mathrm{refl}_{a''}) \equiv c(a'') : C[(a'', a'', \mathrm{refl}_{a''})/(a, a', p)]}$$

This means that we can think of f as an extension of the family c. Note also that as with the other rules, this f will be unique; the rule does not state abstract existence, but rather the possibility of a certain construction. There is no judgemental η law for the identity type, but below we derive a propositional uniqueness principle. ○

In the above definition one should note that since types are stable under substitution by definitionally equal terms (by the omitted congruence rules), we also have $\mathrm{refl}_x : x = y$ whenever we know that $x \equiv y$.

As a note on the language, we remark that the type $x = y$ being inhabited should be seen as x being equal to y (indeed, that is the entire point of introducing this type). Since we already have the notion of judgemental equality, we will call this new concept propositional equality. True to the proof relevant philosophy of type theory, an inhabitant of the type $x =_A y$ can be seen as a *proof* of the *proposition* that x and y are equal.

We now express and prove the propositional uniqueness principle for identity types.

Lemma 2.1.9
The family of identity types over a fixed type A is generated by the elements refl_x *for $x : A$. In other words, we have a dependent function*

$$f : \Pi_{x,y:A}\Pi_{p:x=y}(x, y, p) =_{\Sigma_{x,y}} (x, x, \mathrm{refl}_x),$$

which is to say that we have elements of $(x, y, p) =_{\Sigma(x,y:A).(x=y)} (x, x, \mathrm{refl}_x)$ for each (x, y, p).

Proof. This is a straightforward application of path induction. Inducting on p, we can assume that $x \equiv y$ and that $p \equiv \mathrm{refl}_x$, and in this case we have that $\mathrm{refl}_{(x,x,\mathrm{refl}_x)} : (x, y, p) =_{\Sigma(x,y:A).(x=y)} (x, x, \mathrm{refl}_x)$. This style of proof is often what we will resort to, but since this is the first proof of its kind, we give the formal argument below.

We have not yet introduced universes; for now, just think of the judgement $A : \mathcal{U}$ as being equivalent to $A\,\mathbf{type}$, with the additional property that this can be stated as an assumption in the context. To apply the path induction inference rule, we must show each of its assumptions. We start by considering the family defined by $C(x, y, p) \equiv (x, y, p) =_{\Sigma(x,y:A).(x=y)} (x, x, \mathrm{refl}_x)$ in the context $\Gamma, A : \mathcal{U}, x : A, y : A, p : x =_A y \vdash$, so that the type A is an arbitrary type.

We should argue that this corresponds to the first assumption of path induction, or more specifically, that $\Gamma, A : \mathcal{U}, x : A, y : A, p : x =_A y \vdash (x, y, p) =_{\Sigma(x,y:A).(x=y)} (x, x, \mathrm{refl}_x)\,\mathbf{type}$. It will be sufficient to argue that $\Gamma, A : \mathcal{U}, x : A, y : A, p : x =_A y \vdash \Sigma(x, y : A).(x = y)\,\mathbf{type}$, which follows from the fact that we have $x = y\,\mathbf{type}$ in this context. In fact we have that $\Gamma, A : \mathcal{U} \vdash \Sigma(x, y : A).(x = y)\,\mathbf{type}$, but then we just need to weaken this statement to obtain the desired judgement.

As noted above we can easily give a partial element of this family, $c(x) : C(x, x, \mathrm{refl}_x)$, simply by declaring that $c(x) \equiv \mathrm{refl}_{(x,x,\mathrm{refl}_x)}$, which is the second assumption of path induction. Now path induction gives us an f of the type $C(x, y, p)$ which is exactly what we set out to construct. □

The first proof strategy seems deceptively simple, as if we are getting more than we give, while the second is cumbersome, and seems to have many intricacies. In fact, the arguments are completely equivalent, and what we have discovered mirrors the set theoretic case, where it is often intractible to give the full argument directly from the axioms, even when the abstract proof is trivial. One has to justify the verbiage of "assuming for induction" that $x \equiv x$ and $p \equiv \mathrm{refl}_x$, but allowing for this style of proof is exactly the strength of the path induction rule (or any other rule). We can replace the hard problem of showing a particular instance of the conclusion of path induction by the much easier problem of showing the assumptions of the rule. Then we just have to package this in words, instead of the cumbersome formal notation, to obtain a human-readable proof.

Remark 2.1.10 (Extensional and intensional identity types)
A rule that one can add to the type theory is a reflection rule for propositional equality. It is the principle that one can, from the existence a proof that a is equal to b, conclude that a and b are indeed propositionally equal. As an inference rule it is expressed as

$$\frac{\Gamma \vdash p : a =_A b}{\Gamma \vdash a \equiv b : A}$$

Adding this rule takes us into *extensional* type theory. The name derives from the extensional behaviour of type-checking terms within the system. As we have seen many times, if two types are judgementally the same, a term of one is a term of the other. In addition to the above rule, that would mean that checking whether $t : B$ for some term t which we know to be of type A, we would need to consider all possible proofs that $A = B$. An important consequence of the reflection rule is the uniqueness of identity proofs, i.e., the fact that types of the form $x = y$ contain at most one element. To see this, assume that $p : x = y$, and claim that there is a $q : p =_{(x=y)} \mathrm{refl}_x$, noting that $\mathrm{refl}_x : x = y$ by the reflection rule so that this expression is well typed. Towards an application of path induction, we assume that $x \equiv y$, and $p \equiv \mathrm{refl}_x$, in which case we can define $q \equiv \mathrm{refl}_{\mathrm{refl}_x}$. Path induction then yields, for any x, y, p, an inhabitant $q : p =_{(x=y)} \mathrm{refl}_x$, and a final application of the reflection rule lets us conclude that indeed $p \equiv \mathrm{refl}_x$ for any path. The main advantage of the extensional type theories are the extensionality properties they enjoy. As we discussed for the dependent function space, judgemental equality at each point means judgemental equality of the function, and in an extensional type theory this just means that functions are equal exactly when they are equal at each point. This is a strong proof technique that one might take for granted, and its employ in a certain theory is not easy to establish.

If we leave out this rule, we end up with intensional type theory. Since the only way to obtain judgemental equality of terms in this model is by the structural rules in the type theory, type-checking is a decidable procedure. Apart from this very desirable computational property, intensional type theories are the ones with connections to topology and higher category theory. The lack of extensionality is then remedied by adding other axioms to the theory, most famously the univalence axiom. We have not yet developed the notion of type equivalence, and hence we cannot state the univalence axiom, but we can mention another extensionality principle; function extensionality. This is the axiom which allows us to conclude that $f = g$ in the propositional sense, from the fact that $fx = gx$ for each x. We will return to this point in a later chapter, where we discuss cubical type theory. ○

2.1.5 Types as propositions

So far we have mostly considered types as being an alternative to sets, thinking of them mostly as collections of objects, with the additional constraint that these objects do not make sense in the absence of a type to which they belong. A different way to think of types is as propositions. The basic example is that of identity types, which we already alluded to. The identity type $a = b$ can be thought of as the proposition that a and b are equal, and inhabitants of the identity type can be thought of as proofs of this statement. This subsection is a brief overview of how we extend this to more complicated propositions.

We need to define several new types; the empty type, the unit type, the negation type and the coproduct type.

Definition 2.1.11
We give here the types needed to implement first order logic in type theory.

- The empty type is a closed type with no elements, and we denote it by $\mathbf{0}$. It can be formed in any context and does not have an introduction rule. The elimination rule corresponds to a kind of proof by contradiction, with the extra condition that the inhabitant of $\mathbf{0}$ becomes bound in this inhabitant. This will be a natural requirement to those familiar with the natural deduction proof system for intuitionistic logic, as it corresponds to the reductio ad absurdum rule. As an inference rule:

$$\frac{\Gamma, x : \mathbf{0} \vdash C\,\mathbf{type} \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathsf{ind}_{\mathbf{0}}(a) : C[a/x]}$$

  Note again that a is bound in the term $\mathsf{ind}_{\mathbf{0}}(a)$. We think of this type as the false proposition, or bottom.

- Now that we have the empty type we can define negation of A simply as the function type $A \to \mathbf{0}$, and we denote it by $\neg A$. There are no new rules to add here, so we simply note that the type is a way to construct an inhabitant of $\mathbf{0}$, so inhabitants of this type prove that A is not inhabited.

- The unit type, denoted $\mathbf{1}$, is the true proposition or top. It is again a closed type valid in any context, but here we have every rule except an η rule, although we leave out formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \mathbf{1}} \qquad \frac{\Gamma, x : \mathbf{1} \vdash C\,\mathbf{type} \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathsf{ind}_{\mathbf{1}}(C, c, a) : C[a/x]}$$

  Essentially the sum of these rules express the fact that if you have a valid construction with a variable of type $\mathbf{1}$ in the context, that same construction can be made without this variable. This reflects the fact that if you have a valid proof assuming a true statement, you can drop that assumption and still have a valid proof.

- The last type we need is the coproduct type, which corresponds to the logical disjunction. The formation is similar to the product; if we have two types, A and B, we can form the coproduct of these types, denoted $A + B$. We overload the $+$ slightly here, but it is always clear from the context whether one is taking the coproduct of types or the sum of numbers. As you would maybe expect from a disjunction, introducing an element is done by imputing either an element of A or B, So that the coproduct corresponds to disjoint union.

$$\frac{\Gamma \vdash A\,\mathbf{type} \qquad \Gamma \vdash B\,\mathbf{type}}{\Gamma \vdash A + B\,\mathbf{type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathbf{type}}{\Gamma \vdash \mathsf{inl}(a) : A + B}$$
$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathbf{type}}{\Gamma \vdash \mathsf{inr}(b) : A + B}$$

There is a handedness to these rules, since inl imputes a value on the left, and vice versa, and we can read them as "input left" and "input right" respectively. The elimination rule for coproducts might seem overly complicated, but it will make sense when cast in the types as propositions setting. Say that we know some disjunction $\varphi \vee \psi$ to be true, and we want to prove χ. If we can show χ from both φ and ψ, then we have a proof, since we know that at least one of these are true. With this in mind, we present the inference rule:

$$\frac{\Gamma, z : A + B \vdash C\,\mathbf{type} \qquad \Gamma, x : A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathsf{inr}(y)/z] \qquad \Gamma \vdash e : A + B}{\Gamma \vdash \mathsf{ind}_{+}(C, c, d, e) : C[e/z]}$$

We leave out the computation rules for the coproduct type; they simply state that if E is of the form inl(a) or $\mathsf{inr}(b)$, the induction rule does not construct a new element of $C[\mathsf{inl}(a)/e]$, but rather just returns $c[a/x]$, which is part of the input to the rule.

○

With this, the only propositional constructor needed for propositional logic without an interpretation in type theory is implication, which is modelled by the non-dependent function space. We can extend our interpretation to predicate logic by declaring Π types to be the interpretation of universal quantification, $\forall$, and Σ types to be existential quantification, $\exists$.

To illustrate how one applies this in practice, we sketch the proof of addition being commutative.

Proposition 2.1.12
Addition on the type $\mathbb{N}$ as defined in 2.1.7 is commutative.

Proof. We want an inhabitant of $x + y = y + x$ which has x and y as free variables, which is the same as an inhabitant of $\Pi(x : \mathbb{N}).\Pi(y : \mathbb{N}).x + y = y + x$. The key observation that we have not yet made, is that the type family C in the elimination rule for the natural numbers could be a type corresponding to some proposition, for instance $x + y = y + x$, which is a perfectly fine type in the context of two natural numbers x and y. What we need now is to show the statement in the case where x is 0, and under the assumption that we have $x + y = y + x$ show $s(x) + y = y + s(x)$. □

Before moving on we will introduce the notions of transport. The idea here is that if we have some type, or proposition, P, with a free variable x, and we know that $P[a/x]$ and $a = b$ are each inhabited, we want to conclude that $P[b/x]$ is inhabited. In other words, if we know some proposition P to be true for a, and we know that a and b are equal, then we also know P to be true of b. Formally this is achieved via transport, which is simply a dependent function of type $\Pi(y, z : A).\Pi(p : x = y).P[y/x] \to P[z/x]$ with the assumption that $\Gamma, x : A \vdash P\,\mathbf{type}$. Clearly we then also have $\Gamma, y : A, z : A, p : (y = z) \vdash P[y/x] \to P[z/x]\,\mathbf{type}$, and by the congruence rules for judgemental equality we have $P[y/x] \equiv P[z/x]$ whenever $y \equiv z$. Now the identity function on $P[y/x]$ gives us a partial element at each refl, and the induction principle for the identity type gives us a full element of $\Pi(y, z : A).\Pi p : (x = y).P[y/x] \to P[z/x]$, and we denote this element by p_*. This operation mapping p to p_* is called transport, and can be thought of as a function $\Pi p : (x = y).P(x) \to P(y)$, and we denote this function simply by $\mathsf{transport}^P(p, -)$.

We also note here that our work in type theory so far has been proof relevant. This means that every time we have shown something, it has been by constructing a term of some type, and each of these terms can be reasoned about within the very theory they were constructed.

2.1.6 Universes

It is tempting in type theory to naively say that there is some type of types, or a universe of types. This corresponds to the existence of a set containing all sets, leading immediately to a variant of Russels paradox. Precisely as in set theory, we can resolve this by considering instead a sequence of universes or similar. We want to replace judgements of the form $\Gamma \vdash A\,\mathbf{type}$ with judgements $\Gamma \vdash A : \mathcal{U}$, so that instead of declaring something to be a type, we declare it to be an inhabitant of the universe of types.

Instead of constructing such a $\mathcal{U}$ directly, we consider a sequence of types $\mathcal{U}_n$, indexed by natural numbers of the meta theory, such that $\mathcal{U}_i : \mathcal{U}_{i1}$. We require moreover that these types are cumulative, so that $A : \mathcal{U}_{i+1}$ whenever $A : \mathcal{U}_i$. In all other parts of the type theory we have presented so far, typing has been unique up to judgemental equivalence, in the sense that if we had some term $t : A$ which had another type $t : B$, we would have known that $A \equiv B$. This style of universe introduction breaks this property, although only for inhabitants of the universe types. We say that a type is small if it is contained in this universe of types. There are two main ways that one can formalize this, known as universes á la Russel and universes á la Tarski. We will take an informal approach to universes, and only note that there are issues, which are not within the scope of this thesis.

With a universe of types we are now able to consider types as variables. In particular we can define a family of types $C : \Pi(x : A).\mathcal{U}$, which gives us a type $C[a/x]$ for each $a : A$. Unravelling the definition of the family, we see that we have already encountered examples of this, namely when we have written $\Gamma, x : A \vdash B\,\mathbf{type}$ for the introductions for the dependent types. Indeed, we might usually give the presentation the other way around so as to frame these type formation rules in the sleeker language of type families, as opposed to the atomic judgements we have been using so far.

Now that we can treat types as terms, we have some added flexibility in the language, and in particular we can now form the identity type of two types, i.e., we can consider the type

$$A =_{\mathcal{U}} B.$$

So now we have a looser notion of equality of types than the judgemental one, just as we have for terms of types. For our purposes, this is the main reason to introduce universes to the theory, although they of course have other purposes generally.

2.1.7 Types as spaces

A recent development in type theory is the analogy between types and spaces. In the previous section we saw how types are like propositions, and inhabitants of types are proofs of these propositions. The analogous analogy in the types as spaces paradigm is between spaces and types, with inhabitants of types being points in the space. Under this interpretation, the identity types are given a special status, as they are the paths between points. The function types are interpreted as the collections of continuous maps, and the pointwise equalities, or more specifically inhabitants of the type $\Pi x.fx = gx$ for two parallel functions f and g, are the homotopies of functions. The primary focus of the types as spaces interpretation is the study of identity types, and in particular higher identity types.

We have already encountered identity types of identity types, which are interpreted as homotopies of paths, and we can iterate to obtain homotopies of homotopies and so on. We can define composition and inversion of paths, and we already have the the constant paths at each point, since these are exactly the refl_a's. In [16] this idea is developed to its conclusion, which is that types have the structure of weak ω-groupoids.

One definition which is particularly clear in this context is that of an equivalence of types. In topology, we say that two spaces are homotopy equivalent if there are functions in each direction and both compositions are homotopic to the identity. We can define the composition of two function $f : A \to B$ and $g : B \to C$ as $\lambda x.gfx$, which allows for the desired comparisons. We can translate the homotopy equivalence definition directly to type theory:

Definition 2.1.13
Two types $\Gamma \vdash A\,\mathbf{type}$ and $\Gamma \vdash B\,\mathbf{type}$ are equivalent if there are functions $f : A \to B$ and $g : B \to A$ and homotopies $h : \Pi x.f \circ gx = x$ and $h' : \Pi x.g \circ fx = x$. In this situation f and g are called equivalences, and g is the pseudoinverse of f and vice versa. ○

The existence of two pseudoinverse functions is one of many equivalent definitions. We refer the reader to chapter four of [14] for a detailed discussion of type equivalences. We recall here that there is a type of equivalences $A \simeq B$, defined simply as the type of functions with a pseudoinverse. Terms of this type are then a function $A \to B$, a function $B \to A$, and proofs that the compositions are homotopic to the identity. Here proof is understood in the sense of the types as propositions analogy, so simply as an inhabitant term of some appropriate type.

Having introduced the concept of type equivalence, we can now state the univalence axiom. We can always obtain an equivalence of A and B given an equality of the two; by induction on the equality type we can assume that $A \equiv B$ and that the equality is refl_A, and in this case $\text{id}_A : A \to B$ is its own pseudoinverse. We denote this function by $\mathsf{idtoequiv}$.

On the other hand there is no way to obtain an equality from an equivalence in the theory so far. The univalence axiom is exactly this, or more specifically, the univalence axiom states that $\mathsf{idtoequiv}$ is an equivalence. Univalence is a property of the universe that one has added to the type theory, since the type of types is exactly what allows for the definition of $A = B$ in the theory.

2.2 Categorical semantics

In this section we will see how one can model type theory in presheaf categories, and hint at the model in the more general setting of categories with families. Before we begin, we will comment on some foundational issues. Type theory is supposed to give us a foundation for mathematics, and in the end replace set theory in this regard. Even so, many of the semantic notions that we will present in this chapter relies on sets and therefore set theory in some capacity. It is convention to say that we work in an ambient set theory strong enough to have the operations that we use, but not as strong as the general ZFC theory to which mathematicians are accustomed.

Definition 2.2.1
The category Fam has as objects pairs $(A, (B_a)_{a\in A})$ where A is a sets, and $(B_a)_{a\in A}$ is a family of sets indexed by A. Morphisms $(A, (B_a)_{a\in A}) \to (A', (B'_a)_{a\in A'})$ are functions on the index sets $f : A \to A'$ and families of functions $g_a : B_a \to B_{f(a)}$. Identities are the identity on the index set together with identities along each B_a, and similarly we define composition as composition of the maps on the indexes and composition of the maps on the B_a's. ○

We are now ready to define categories with families. The definition comes with many specifications of language, designed to make obvious the interpretation of each part as a model of type theory.

Definition 2.2.2
A category with families (from here on out abbreviated CwF) is a category $\mathcal{C}$ with a terminal object, together with a functor $\mathbb{F} : \mathcal{C}^{\mathsf{op}} \to \mathsf{Fam}$. We will call the objects of $\mathcal{C}$ *contexts*, and in particular the terminal object will be the *empty context.* We will call the morphisms substitutions. When there is a possibility of confusion, we will call them semantic substitutions, as opposed to the syntactic substitution defined in the previous section.

For a context $\Gamma \in \mathcal{C}$, we denote the indexing family of $\mathbb{F}(\Gamma)$ by $\mathrm{Ty}_{\mathbb{F}}(\Gamma)$ and the family by $(\mathrm{Term}_{\mathbb{F}}(\Gamma; A) | A \in \mathrm{Ty}_{\mathbb{F}}(\Gamma))$. To make this fit with our type theoretic notations, and to suggest the interpretation of the theory in the model, we will also write $\mathcal{E}(\Gamma \vdash A)$ (short for the syntactic judgement $\Gamma \vdash A\,\mathbf{type}$) for $A \in \mathrm{Ty}_{\mathbb{F}}(\Gamma)$ and $\mathcal{E}(\Gamma \vdash a : A)$ for $a \in \mathrm{Term}_{\mathbb{F}}(\Gamma; A)$. We follow convention and guard the semantic expression with an $\mathcal{E}$, so as to avoid confusing semantic statement with syntactic ones. For substitutions $\sigma : \Delta \to \Gamma$ the action of the substitution on $\mathrm{Ty}_F(\Gamma)$ and $\mathrm{Term}_{\mathbb{F}}(\Gamma; A)$ will be written as in the syntactic case, i.e., as $A\sigma$ and $a\sigma$ respectively. The functoriality of $\mathbb{F}$ means that, as for syntactic substitutions,

$$A\,\mathrm{id} = A, \qquad (A\sigma)\tau = A(\sigma\tau), \qquad a\,\mathrm{id} = a, \quad \text{and} \quad (a\sigma)\tau = a(\sigma\tau).$$

We also require that the category has specified *context extension* operations, corresponding to the syntactic extension of a context by a variable. If $\Gamma \in \mathcal{C}$ and $\mathcal{E}(\Gamma \vdash A)$ there is an object $\Gamma.A$, a substitution $p : \Gamma.A \to \Gamma$ and a term $\mathcal{E}(\Gamma \vdash q : Ap)$. These satisfy the universal property that for each substitution $\sigma : \Gamma \to \Delta$ and term $\mathcal{E}(\Delta \vdash a : A\sigma)$, we have a substitution $(\sigma, a) : \Delta \to \Gamma.A$ satisfying the following equalities:

$$p(\sigma, a) = \sigma, \qquad q(\sigma, a) = a, \qquad (\sigma, a)\tau = (\sigma\tau, a\tau) \quad \text{and} \quad (p, q) = \mathrm{id}\,.$$

○

The universal property that context extensions enjoy is in [8] called a comprehension for Γ; the notation $\mathcal{E}(\Gamma \vdash A)$ is following Dybjer. These context extension operations are what will allow us to build each context inductively inside the model, with the universal property ensuring that we remain true to the underlying type theory at each stage. This means that we could at this point produce a model of a type theory with dependent typing, in the sense that we could construct families of types within the model. This is not quite enough for our purposes; we wish to model both dependent products, dependent sums and identity types, all of which require extra structure on top of the CwF structure. For each of these type formers, we will define the notion of a CwF which supports them. This is done by requiring stability of the sets $\mathrm{Ty}_{\mathbb{F}}(\Gamma)$ under the very same operations we saw in the formation rules for these types, subject to the relevant β and η laws. For example, for a CwF to support Π-types, we need it to satisfy the following conditions:

- Whenever we have $\mathcal{E}(\Gamma \vdash A)$ and $y \in \mathcal{E}(\Gamma, a : A \vdash B(a))$, there is an element $\Pi(x : A).B \in \mathrm{Ty}_{\mathbb{F}}(\Gamma)$, corresponding to the type former.
- Whenever we have $\mathcal{E}(\Gamma, a : A \vdash f : B(a))$, we have an element $\mathcal{E}(\Gamma \vdash \lambda x.f : \Pi(x : A).B)$, corresponding to λ abstraction, the introduction rule for Π types.
- Whenever we have $\mathcal{E}(\Gamma, x : A \vdash f : B(x))$ and $\mathcal{E}(\Gamma \vdash u : A)$ we have an element $\mathcal{E}(\Gamma \vdash fu : B(u))$, corresponding to function application, the elimination rule for Π types.
- We have the β law $\lambda x.f(a) = f(a)$ (where we implicitly identify f with an actual function).
- We have the η law $\lambda x.(f(x)) = f$.
- All this structure is compatible with the morphisms of the category, so that

$$\begin{aligned}(\Pi(x : A).B)\sigma &= \Pi(x : A\sigma).(B(\sigma)\\ (\lambda(x : A).f)\sigma &= \lambda(x : A\sigma).f(\sigma p, q)\\ (fu)\sigma &= (f\sigma)(u(\sigma, p))\end{aligned}$$

  where p and q are as in the CwF definition.

This follows the clear pattern of translating the five classes of syntactic rules needed for defining a type and the requirement ensuring compatibility with substitutions directly into certain classes of maps. To avoid repetition, we leave out the complete definitions of what it means for a CwF to support Σ types and identity types. We are now ready to present an example which collects the set theoretic intuitions given at explanations and justifications for each of the type formers. Apart from showing that the next example supports Π types we will also construct Σ types and identity types, which will motivate the definition of a CwF supporting each of these types.

Example 2.2.3
We will now give the category Set the structure of a CwF supporting each of the dependent sum and dependent product in addition to a not quite satisfactory identity type. To do this, we must first provide a functor $\mathsf{Set}^{\mathsf{op}} \to \mathsf{Fam}$. To a set Γ we assign as indexing family the set of small sets indexed by Γ, and the types over Γ in a context to be simply the sections of the index projection, or collections of $a_\gamma \in A_\gamma$ indexed by Γ. To be more explicit, the indexing set is any possible collection of small sets indexing by Γ. Another way to package this information is to regard the set Γ as a discrete category, and A as a presheaf on this category. This is the which we will generalize to obtain the model in presheaf categories.

We can see the terms of type $(A_\gamma)_{\gamma\in\Gamma}$ as dependent functions $t : (\gamma : \Gamma) \to A_\gamma$. To a map of sets $\sigma : \Gamma \to \Delta$ we must associate a map of families in the other direction, which we do simply by mapping a family $(A_\delta)_{\delta\in\Delta}$ to the family $A\sigma = (A_{f(\gamma)})_{\delta\in\Gamma}$. Similarly we map a term, t, of type A in context Δ to $t(\sigma(\gamma))$, which is clearly a term of type $A\sigma$.

Now we must specify the context extension operation. We define it by taking the dependent sum as described for sets (or disjoint union), letting

$$\Gamma.A = \{(\gamma, a) \mid \gamma \in \Gamma, a \in A_\gamma\}.$$

In this case the projections p and q are just given by the actual projections so that, e.g., $p(\gamma, a) = \gamma$.

The terminal object in Set is of course $\{*\}$.

This gives Set the structure of a CwF, which means that it models type dependencies. We would like to extend this to the types we considered in the syntax section, so that Set supports dependent functions, dependent sums, natural numbers and identity types. Before we do this, we note that there is another way to view the information given above. A family indexed by Γ can be seen simply as a set with a function $A \to \Gamma$, or in other words an element of $\mathsf{Set}_{/\Gamma}$. Under this view, a substitution (or map of sets) induces a functor on slice categories by pullback, and this induced map coincides with the one defined above, which is the same story we saw in the preliminary category theory.

We would like for this model to support Π types. By spelling out the definitions, we need to give a family of sets $\Pi(x : A).B$ in the context of Γ, so a type over Γ, under the assumption that we have a type over $\Gamma.A$. Perhaps the most instructive way to construct this type is to consider the data that goes into constructing a term of the type. If we take some $\mathcal{E}(\Gamma, x : a \vdash t : B)$, we have a section of a map of sets $B \to \Gamma.A$. This is the same as a collection of maps $A_\gamma \to B(a, \gamma)$, which is exactly a dependent function from A to B in the context of Γ. So we can define the dependent product type to consist of such all set theoretic dependent functions. It is clear that we have our introduction rule for the type, and the fact that our terms split into functions $A_\gamma \to B_{(\gamma,a)}$ means that this set of possible dependent functions make up a family over Γ.

This is also an appropriate place to highlight what is going on with expressions like $B(u)$ in the model. If we have some type $\Gamma.A.B \to \Gamma.A$, and some section, or term, $u : \Gamma \to \Gamma.A$, we want to construct a family $B(u)$ indexed by Γ. We do this by letting $B(u)_\gamma = B_{(\gamma, u(\gamma))}$. Similarly we can consider $t(u)$ if we have some $\mathcal{E}(\Gamma x : A \vdash B)$, and terms of type A in context Γ, u, and of type B in context $\Gamma.A$, t. A term of $B(u)$ corresponds to a section of the composite projection $\Gamma.A.B \to \Gamma$ which factors over u.

The elimination rule is most clearly seen when considering the fact that for a term of A over Γ is a section to the map $u : \Gamma.A \to \Gamma$, and a term of type $\Pi(x : A).B$ is a section to the map $\Gamma.A.B \to \Gamma.A$. The composition of these section gives us a section of the composite $\Gamma.A.B \to \Gamma$, i.e., a term of type $B(u)$ in the context of Γ and a specific term of A. Under this view, the η rule becomes almost vacuous; if t has type $\Pi(x : A).B$, it is already a section of the projection $\Gamma.A.B \to \Gamma.A$, and λ abstraction just returns this same section. The β rule is understood using similar arguments.

We leave out the full verification of compatibility with substitutions, noting simply that they follow from the fact that substitutions must preserve sections and context projections.

The Σ types can be constructed in an equally intuitive way. We have the same assumptions for the type former, i.e., we have a Γ indexed set A, and a $\Gamma.A$ indexed set B, so we should first describe the elements. These are pairs of terms of A and B, with the second term depending on the first. In the model, this is a pair of sections $\Gamma \to \Gamma.A$ and $\Gamma \to \Gamma.A.B$. Over a specific $\gamma \in \Gamma$ this becomes just a pair (a, b), where b is indexed by (γ, a).

The introduction and elimination rules are pairing and projection, which just correspond to pairing and projection of Cartesian products of sets. There is no η rule, so we just have to check that the projections applied to the pairing of two terms returns the respective original terms, which is clearly true. ○

We can give a definition of supporting Σ types which is almost a direct translation of the syntactical definition of the type, by simply wrapping each statement of the definition in $\mathcal{E}(-)$. The only thing missing here is the compatibility with morphisms of the category. This is done in a similar fashion as we saw for Π types.

So far, we have not actually shown that type theory can be modelled in a structure as we have defined it, although we have hinted at the interpretation of the objects of type theory in a CwF. We will leave out the general interpretation of type theory in a CwF, and as a consequence, we will have to defer the main results to after we have introduced the particular presheaf model that will be our focus for the rest of the thesis.

2.2.1 Presheaf categories as Categories with Families

In a certain sense this subsection will just be a repetition of the Set example, but since presheaf semantics for type theory are central in the rest of the thesis we take the time to spell out the example, as well as its connection to some of the results presented in the preliminaries.

Let $\mathcal{C}$ be a small category. We will now present the presheaf semantics for type theory in $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$. To begin with, note that the terminal object is the constant singleton presheaf. We need to give a functor $(\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}})^{\mathsf{op}} \to \mathsf{Fam}$. We associate to each presheaf X its category of elements, $\int X$, and in turn the (small) presheaves on this category $\mathsf{Set}^{\int(X)^{\mathsf{op}}}$. It is slightly more convenient to state these definitions for covariant presheaves.

Definition 2.2.4
Let $\mathcal{C}$ be a small category, and consider a covariant presheaf $X \in \mathsf{Set}^{\mathcal{C}}$. The *category of elements of* X has as objects pairs (c, x), where $c \in \mathcal{C}$ and $x \in X(c)$. Morphisms $(c, x) \to (c', x')$ are morphisms $f : c \to c'$ such that $X(f)(x) = x'$. We denote this category by $\int_{\mathcal{C}} X$. ○

It is worthwhile to spell out what it means to have a presheaf on the category of elements for some covariant presheaf X on a small category $\mathcal{C}$. We need for each object in the category $c \in \mathcal{C}$ and element in $x \in X(c)$ a set $A(c, x)$. For each morphism $f : c \to c'$ in $\mathcal{C}$, we need maps $A(c, x) \to A(c', X(f)(c)$, and this assignment is functorial.

The reason we emphasize this, is the following lemma.

Proposition 2.2.5
Let $\mathcal{C}$ be a locally small category small category, and consider a presheaf $X \in \mathsf{Set}^{\mathcal{C}}$. Then we have an equivalence of categories

$$\mathsf{Set}^{\int_{\mathcal{C}} X} \cong \mathsf{Set}^{\mathcal{C}}_{/X}$$

Proof. We define functors in each direction. First, we note that there is a functor $F : \mathsf{Set}^{\int_{\mathcal{C}} X} \to \mathsf{Set}^{\mathcal{C}}_{/X}$ given by disjoint union. Specifically, $F(A)(c) = \amalg_{x \in A(c)} A(c, x)$, with the restriction maps being the glueing of these maps. The map $A \to X$ is given at $c \in \mathcal{C}$ by mapping the set $A(c, x)$ to $x \in X(c)$. We must check that this map is natural, but this is ensured by the definition of morphisms in $\int_{\mathcal{C}} X$. We extend the map to moprhisms by noting that natural transformations $A \to A'$ must be compatible with the indexing, and so these are sent to natural transformations over the object X.

To go in the other direction, consider some $(B, p) \in \mathsf{Set}^{\mathcal{C}}_{/X}$. We define $G(B)(c, x)$ to be $(p(c))^{-1}(x)$, restriction are again inherited, and we extend the map to morphisms by reusing the same functions.

Each composition is clearly naturally isomorphic to the appropriate identity functor, so we have our equivalence of categories. □

As a corollary to this proposition we obtain the fact that every presheaf category on a small category is LCCC, since every over category is equivalent to a presheaf category on some other category, which we know to be CCC.

So now we are ready to provide the functor $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$. To each presheaf X we associate the set of small presheaves on $\int_{\mathcal{C}^{\mathsf{op}}} X$, which will play the roles of types. So we have replaced the families of sets indexed by some base set by a presheaf indexed by another presheaf. A term is again a collection of elements, again indexed by elements of the context, which are now pairs (c, x), and we write $a_{(c,x)} \in A(c, x)$. Now we have the extra requirement that these elements be compatible with the presheaf restrictions, or more explicitly

$$A(f)(a_{(c,x)}) = a_{(f(c),X(f)(x))}$$

Just as before, the easiest way to package this information is as a section to the indexing map.

Context extension is defined similarly to before; if we have some $A \in \mathsf{Set}^{\int_{\mathcal{C}^{\mathsf{op}}} X}$, we define $\Gamma.A$ to be the level-wise disjoint union

$$\Gamma.A(c) = \{(\gamma, a) \mid \gamma \in \Gamma(c), a \in A(\gamma, c)\}.$$

The restrictions are simply the coordinate-wise restrictions, which are compatible by assumption. Once again, this definition has the correct universal property.

This is sufficient to give any presheaf category on a small category the structure of a CwF, but we need this model to support Σ types, Π types and natural numbers. The easiest one of these is clearly the natural numbers. We want every natural number to be constructible in any context, so the natural numbers must simply be interpreted by the product $\Gamma \times \mathbb{N}$ in context Γ with the context projection being the usual projection. As a presheaf on the category of elements of Γ this is simply the constant $\mathbb{N}$ presheaf, with each restriction map being the identity. This extends to a general statement: closed types are always modelled by constant presheaves. For us to have a complete interpretation of the natural numbers we still need to define the 0 and successor function, which are each defined as the level-wise 0 and successors in the set $\mathbb{N}$.

For the Σ types, say of types $\mathcal{E}(\Gamma \vdash A)$ and $\mathcal{E}(\Gamma, x : A \vdash B)$, we follow the analogue of the construction in Set. We want a term of the type to be a pair of a term of type A and one of type B, with typing of the second term being determined by the first. In the presheaf model, this translates into a pair (a, b) where $\mathcal{E}(\Gamma \vdash a : A)$ and $\mathcal{E}(\Gamma \vdash b : B[a/x])$, and we define $\mathcal{E}(\Gamma \vdash \Sigma(x : A).B)$ to be the collection of such pairs. The restriction of a pair is simply the coordinate-wise restriction. Since this is essentially the same construction we saw in the Set model, we will not further discuss this.

We consider again the situation from before, but now we want to consider the dependent product. We define $\Pi(x : A).B$ at $I \in \mathcal{C}$ over a specific $x \in \Gamma$ by collecting all families $(f_\sigma)_{(\sigma:J \to I)}$, such that $f_\sigma \in \Pi_{a \in A(I,x)} B(I, x, a)$ and for any $\tau : K \to J$ we have that the action of τ on $f_\sigma(u)$ equals $f_{\tau\sigma}(u)$.

Any presheaf category will also support a certain identity type. If we have two terms of the same type $\mathcal{E}(\Gamma \vdash a : A)$ and $\mathcal{E}(\Gamma \vdash a' : A$, we can ask at each (I, x) whether they are equal. This information we can use to define $\mathcal{E}(\Gamma \vdash a = a')$ by $\{\star \,|\, a(I, x) = a(I, x)\}$, i.e, the singleton set whenever a and a' are equal and the empty set whenever they are not. Because this set has at most one element, the theory of the model satisfies uniqueness of identity proofs. Hence the type is the extensional identity type. Therefore we cannot at this point conclude that the intensional identity types are different from the extensional ones; we might simply have that the rules presented for the identity type imply the reflection rule. This is luckily not the case, as we shall see in the next section

Another thing which can be added to the theory at this point is an interpretation of universes. There is a standard construction which, given a set theoretic notion of universes, supplies a interpretation for the universes we discussed earlier. This construction is the Hofmann-Streicher construction, and is covered in [9]. Essentially, we consider presheaves with image completely contained "small", and let types, which are presheaves, be considered "small" in the sense of universes in the type theory, if the presheaf is small in this sense.

Now we are ready to state the results that make the whole thing come together; the soundness of the interpretation, and the substitution lemma.

Part of the reason to give the presentation of the model as done here, is that if one were to prove the substitution lemma, it would have to be done alongside defining the interpretation. Since we leave out the proofs of these foundational results, we have not been careful in this process, but it is important to note that this is the proof strategy.

We have already given the interpretations for types and terms, and the interpretation for contexts as presheaves is obtained by context extensions. The only thing still missing is interpretations of substitutions, which are, like the syntactic counterpart, not central to this thesis. The fact that substitutions can be interpreted into the model is important however, so we include the definition for completeness. The interpretation of the substitution $[\,\cdot\,] : \Gamma \to \cdot$ is the unique map from the interpretation of Γ to the terminal presheaf. Now we only need to define the interpretation of $\sigma[x \mapsto t]$, given the interpretation of $\sigma : \Gamma \to \Delta$, with the assumption that $\Gamma \vdash t : A\sigma$. But recall that this is exactly the the case in which we can apply the universal property of the context extension. So now we are ready to give the substitution lemma.

Theorem 2.2.6
Let $\sigma : \Gamma \to \Delta$ be a substitution of contexts. Then we have that

- *the interpretation of σ acts on interpretations of types, say $\mathcal{E}(\Delta \vdash A)$, is the same as the interpretation of $A\sigma$;*
- *the interpretation of σ acts on interpretations of terms, say $\mathcal{E}(\Delta \vdash a : A)$, is the same as the interpretation of $a\sigma$.*

Now that we have the full interpretation of our fragment of type theory in presheaf categories, we make a small note regarding notation. It is standard to denote the notation of syntax by $[\![-]\!]$; in our case, this means that

- we denote the interpretation of a context $\Gamma \vdash$ by $[\![\Gamma]\!]$;
- we denote the interpretation f a type $\Gamma \vdash A\,\mathbf{type}$ by $[\![\Gamma \vdash A]\!]$, or simply $[\![A]\!]$ if the context is clear from the context;

- we denote the interpretation of a term $\Gamma \vdash t : A$ by $[\![\Gamma \vdash t : A]\!]$, or simply $[\![t]\!]$ when the context and typing are not important;
- we denote the interpretation of a substitution $\sigma : \Gamma \to \Delta$ by $[\![\sigma]\!]$.

The strength of this notation is that we can give very short statements of theorems concerning the interaction between judgements of type theory and features of the model. Note that as the types are presheaves on the category of elements of their contexts, we will sometimes write $[\![A]\!](c, \gamma)$ for their evaluations. We will also sometimes write $\gamma(t)$ for $[\![t]\!](c, \gamma)$.

As a salient example, we give the statement of the soundness theorem:

Theorem 2.2.7
The interpretation of type theory is sound. Informally, this means that any judgement of type theory is validated in the model. More specifically, we have translations of typing judgements:

- *if we have a context, $\Gamma \vdash$, we have a presheaf $[\![\Gamma]\!]$;*
- *if we have a type, $\Gamma \vdash A\,\mathbf{type}$, we have a presheaf $[\![\Gamma \vdash A]\!]$ with a context projection to $[\![\Gamma]\!]$;*
- *if we have a term, $\Gamma \vdash a : A$, we have a section of the context projection $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$.*

In addition, any judgemental equality in type theory is modelled as equality in the model, so that

- *if we have the judgement $\vdash \Gamma \equiv \Delta$, we have that $[\![\Gamma]\!] = [\![\Delta]\!]$;*
- *if we have the judgement $\Gamma \vdash A \equiv B$, we have that $[\![A]\!] = [\![B]\!]$, and in particular the context projection coincide;*
- *if we have the judgement $\Gamma \vdash t \equiv u : A$, we have that $[\![t]\!] = [\![u]\!]$.*

Of course the meat of the theorem is the second part, since we have already given the first part. The theorem suggests the general pattern for adding features to a model of type theory; give an interpretation, and show that the posited judgemental equalities are respected y this interpretation as equalities. Moreover, we should show that these interpretations are compatible with substitutions and their interpretations to get the substitution lemma.

Before moving on we will discuss the restriction of presheaf models. First we give the explicit connection between the category of elements framework and the over category framework.

Lemma 2.2.8
Assume that $\mathcal{E}(\Gamma \vdash A)$ and $\mathcal{E}(\Gamma, x : A \vdash B)$, and let $p : [\![A]\!] \to [\![\Gamma]\!]$ be the context projection. In the model presented above, under the equivalence from 2.2.5

- *the interpretation of the dependent sum, $\Sigma(x : A).B$, corresponds to the left adjoint of the pullback along the p;*
- *the interpretation of the dependent product, $\Pi(x : A).B$, corresponds to the right adjoint of the pullback along p.*

This lemma justifies the fact that we have been using the same notation and words for these seemingly distinct concept; any confusion up until this point should now be resolved.

One possibility is that, for certain presheaf models, the interpretation given above can only ever produce types such that the context projection has some extra structure. This could happen as a consequence of the definition of the category on which we consider presheaves, but for us it will be more deliberate. It will be the case that certain axiom in the type theory, which are in a sense just restriction on which types we consider, will give us extra structure in the model.

Instead of reproving the soundness of the interpretation and the substitution lemma in such cases, we want to say that it simply follows from the theorems we stated in this section. The key ingredient will be 2.2.8, which implies the following lemma.

Lemma 2.2.9
If we have some class of morphisms W, in $\mathsf{Set}^{\mathcal{C}^{\mathsf{op}}}$, which contains all isomorphisms and is closed under compositions, arbitrary dependent product and arbitrary pullbacks, we obtain an interpretation of type theory in the types such that the context projection belongs to W. This interpretation will be sound and satisfy the substitution lemma.

This lemma will be very useful in the following sections. In particular, if we want to show that every context projection of the model has some property, it is sufficient to show that the property is closed under the operations mentioned in the above lemma.

CHAPTER 3

Two Examples

In this chapter we will consider two recent extensions of type theory. Each of these will be based on the fragment of the theory presented in the previous chapter with just natural numbers, dependent product and dependent sum. We will give presheaf semantics for each of the theories, which we then restrict to presheaves that have certain properties; for guarded dependent type theory we consider an orthogonality condition, and for cubical type theory we consider a condition similar to the Kan fibration condition on simplicial sets.

First we consider the guarded dependent type theory, which is a way to safely add recursion to type theory. We will see how to encode an example coinductive type in a syntax with recursion guarded by multiple clocks. Additionally, we discuss how to specialize the semantics for the theory with multiple clocks to the single clock and clock free cases, and see that this recovers the models we already know from the previous section.

Secondly we consider cubical type theory. The motivation for this type theory is to have a constructive implementation of a type theory with univalence and intensional identity types. The types theory presented in the previous chapter, with the addition of the univalence axiom and universes, can be given a model in the category of simplicial sets. This is due to Voevodsky, and the construction is given in [12]. The proof that this models the theory soundly is, however, not constructively valid, and s we turn to cubical type theory. Cubical type theory offers an implementation of the identity types via path types, which are constructed as mapping spaces from an abstract interval. We will see how this path type fits into the mold of an identity type, and discuss the presheaf semantics in the category of cubical sets. In particular the correspondence between a syntactical composition structure on types is equivalent to a fibration structure on the presheaf modelling that type is discussed.

3.1 Guarded dependent type theory

Guarded dependent type theory, which we from now on shorten to GDTT occasionally, is a way to add recursion to type theory. There are two things we need to explain at this point; firstly what kind of recursion is the type theory missing at this point and secondly, how exactly does guarded recursion extend possible recursion in the theory.

We have already seen a definition by recursion on the natural numbers as a special case of the inductive principle for the natural numbers. The kind of recursion we are missing is recursive definitions of types. Allowing unrestricted self reference in type formation would allow for a statement of Russells paradox within type theory, which means that we would have defined an inconsistent logical system. On the other hand, certain real life examples of types needed for programming seem to only be definable by recursion. The primary example for this is the type of streams, which are sequences of natural numbers. The way we would like to define this is to simply state that the type Str is equivalent to $\mathbb{N} \times \mathsf{Str}$, and we would likewise want to define terms of this type by recursive formulas. Streams provide a motivation in expanding the computational capabilities of type theory, but the problem that we need to solve is fundamentally mathematical in nature.

One proposed solution for this problem of adding recursion to type theory is guarded recursion as pioneered in [13]. The idea here is to allow for recursive calls that are *guarded by a* $\triangleright$, where the symbol $\triangleright$ is read as "later". More specifically, we way to reason about types that are at the next stage of computation, which is the programmatic interpretation of $\triangleright A$ for some type A. We obtain inhabitants of $\triangleright A$ by pushing inhabitants of A one timestep ahead, with the constructor next. Recursive definition is done by requiring the existence of fixed points for functions $\triangleright A \to A$. To see why there is a problem with unrestricted recursion, we consider the fact that each type has an identity function (defined by $\lambda x.x$), and a fixpoint for this function gives us inhabitant of any type. Under the types as propositions interpretation, this would mean that any proposition is true. On the other hand guarded recursion gives us a natural way to define guarded streams, simply as $\mathsf{Str}^g \simeq \mathbb{N} \times \triangleright \mathsf{Str}^g$.

In general, guarded recursion refers to the above idea, but in this thesis we want to work with an extension of this idea, based on the discussion in [5]. One can think of the above case as having a single clock along which we can step in time, and we refer to it as the single clock case of guarded recursion. The extension we are interested in here is one with multiple such clocks, which we will refer to simply as guarded recursion. This means that we index each operator, $\triangleright$, next and so on, by a κ. This idea was introduced in [1], where it is treated from the programming point of view. The reason to go to the case with multiple clocks is to encode coinductive types via clock quantification. We still have a notion of guarded streams, and now we need to further specify a clock along which the stream is guarded, i.e., $\mathsf{Str}^\kappa \equiv \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa$. We would then define the coinductive type of streams as $\mathsf{Str} \equiv \forall \kappa . \mathsf{Str}^\kappa$.

For this definition to be sensible, we would like an isomorphism $\forall \kappa . \mathsf{Str}^\kappa = \forall \kappa . \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa \cong \mathbb{N} \times \forall \kappa . \mathsf{Str}^\kappa$, but this is not trivial to achieve. The first step is to see that $\forall \kappa . (-)$ distributes over products, so that we only need isomorphisms $\mathbb{N} \simeq \forall \kappa . \mathbb{N}$ and $\forall \kappa . \triangleright^\kappa \mathsf{Str}^\kappa \simeq \forall \kappa . \mathsf{Str}^\kappa$.

We start by giving a presentation of the syntax required to add guarded recursion to a basic type theory with dependent types and natural numbers. Along the way we will relate back to the example of streams as a coinductive type given above.

Then we give semantics for this theory in presheaves on a certain category, following in the model presented earlier in general presheaves, with an additional orthogonality axiom. We will show that if we have not added any clocks, we recover the Set model presented earlier, and with a single clock we recover the topos of trees model presented in [3].

3.1.1 Syntax for guarded dependent type theory

The basis of GDTT is the two dependent types, so Π and Σ types, together a notion of clocks. We do not for now add identity types to the theory, but we return to this point when we have considered the semantics for the theory.

The clocks will be implemented as a primitive of the theory. What this means is simply that we cannot apply our type formers to this primitive and that terms of the primitive cannot use other terms. Below we give as one definition the syntax for this primitive and three derived notions, namely clock quantification, the later modality and a fixpoint operator.

We denote the primitive of clocks by clock. A context $\Gamma \vdash$ can be extended by a clock, $\kappa : \mathsf{clock}$, provided that κ does not occur in $\Gamma \vdash$. In the same way, if a clock is already declared in a context, we have a variable rule.

$$\frac{\Gamma \vdash \qquad \kappa \notin \Gamma}{\Gamma, \kappa : \mathsf{clock} \vdash} \qquad\qquad \frac{\kappa : \mathsf{clock} \in \Gamma \vdash}{\Gamma \vdash \kappa : \mathsf{clock}}$$

The purpose of these rules is to have judgements of the theory which can replace the question of whether a certain variable occurs in some context. This changes what a context is, so it will also change what a context substitution is. Since the context substitution definition was given in an inductive way on the context construction, it suffices to give the inductive step for the clock primitive. We simply need to add the rule

$$\frac{\rho : \Gamma \to \Delta \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\rho[\kappa' \to \kappa] : \Gamma \to \Delta, \kappa : \mathsf{clock}}$$

Now that we have the clock introduction rules, we can introduce the derived notions. Given a type, we can consider the type one time step from now on the clock κ, where κ is a declared clock in the context. The primary way to obtain inhabitants of this type is to push inhabitants of A one step ahead.

$$\frac{\Gamma \vdash A\,\mathbf{type} \qquad \Gamma \vdash \kappa : \mathsf{clock}}{\Gamma \vdash \triangleright^\kappa A\,\mathbf{type}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \kappa : \mathsf{clock}}{\Gamma \vdash \mathsf{next}^\kappa t : \triangleright^\kappa A}$$

So far we have not done any recursion, since we need the fixpoint operator to actually get recursively defined terms. The fixpoint operator is simply a function $\mathsf{fix}^\kappa : (\triangleright^\kappa A \to A) \to A$, which produces a fixpoint. Again $\triangleright^\kappa A$ and A are different, so it does not make sense to ask for a fixpoint of the input function, so we require that the output of $\mathsf{fix}^\kappa(f)$, for some $f : \triangleright A \to A$, is a fixpoint for the composition $A \overset{\mathsf{next}^\kappa}{\to} \triangleright^\kappa A \overset{f}{\to} A$. We express these criteria as inference rules:

$$\frac{\Gamma, x : \triangleright^\kappa A \vdash t : A}{\Gamma \vdash \mathsf{fix}^\kappa x.t : A} \qquad \frac{\Gamma, x : \triangleright^\kappa A \vdash t : A}{\Gamma \vdash \mathsf{fix}^\kappa x.t \equiv t[\mathsf{next}^\kappa \mathsf{fix}^\kappa t/x]}$$

Now we can actually give some examples of recursive definitions. We define $\mathsf{Str}^\kappa \equiv \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa$, noting that we get maps in both direction from the properties of the product. We have $(\mathsf{head}, \mathsf{tail})$ corresponding to the two projections, allowing us to deconstruct a stream into its head (the first number, which is available now) and the tail (the rest of the stream, available one time-step from now). But we can also use the pairing operation to obtain a stream from a natural number and a stream available one time-step from now, and we denote this operation by fold (here we leave out the κ in the notation, although it is technically needed). Now we can consider the stream $\mathsf{fix}^\kappa \lambda s.\mathsf{fold}(0, s)$, which is the stream of all zeroes, defined recursively. Of course, these kinds of streams are not very interesting, and could have been defined with less work with more syntactic checks of productivity. The advantage of this approach is more in the realm of utilizing several streams at once, for example alternating between two streams or similar.

The goal of guarded recursion is to encode coinductive types via clock quantification. Clock quantification is essentially the same as a Π type over clock, in that terms are introduced similarly to λ abstraction, and terms are eliminated by application. The only difference comes from the fact that there are no terms of clock; since it is a primitive of the theory, we can only apply our terms to simple clocks.

Definition 3.1.1

For any type in the context of some clock, we can bind this clock in the type by quantifying over it. Given a term of a type in the context of a clock, we can abstract this clock away in the term to obtain a term of the clock quantification.

$$\frac{\Gamma, \kappa : \mathsf{clock} \vdash A\,\mathbf{type}}{\Gamma \vdash \forall \kappa.A\,\mathbf{type}} \qquad \frac{\Gamma, \kappa : \mathsf{clock} \vdash t : A}{\Gamma \vdash \Lambda_{\mathsf{clock}}\kappa.t : \forall \kappa.A}$$

Elimination is done as for Π types, and we have similar β and η laws.

$$\frac{\Gamma \vdash t : \forall\kappa.A \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\Gamma \vdash tu : A[\kappa'/\kappa]} \qquad \frac{\Gamma, \kappa : \mathsf{clock} \vdash t : A \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\Gamma \vdash \Lambda_{\mathsf{clock}}\kappa.t\kappa' \equiv t[\kappa'/\kappa]}$$

$$\frac{\Gamma \vdash t : \forall\kappa.A}{\Gamma \vdash t \equiv \Lambda_{\mathsf{clock}}\kappa.t : \forall\kappa.A}$$

In addition to these function like laws, we have a restricted inverse of next^κ, the $\mathsf{prev}\kappa$ operation, which binds κ, with an equality:

$$\frac{\Gamma, \kappa : \mathsf{clock} \vdash t : \rhd^\kappa A}{\Gamma \vdash \mathsf{prev}\kappa.t : \forall\kappa.A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \kappa : \mathsf{clock}}{\Gamma \vdash \Lambda_{\mathsf{clock}}\kappa.t \equiv \mathsf{prev}\kappa.\mathsf{next}^\kappa t : \forall\kappa.A}$$

○

Already we have the ingredients for part of the desired isomorphism of types from the introduction to this section.

Lemma 3.1.2
Clock quantification distributes over binary products.

Proof. We show the slightly more general statement that dependent product distributes over binary product. Note first that $\Pi(x : A).B \times C$ is a type in any context in which $(\Pi(x : A).B) \times (\Pi(x : A).C)$ is. We want to define mutually inverse maps on these types.

To get a map $\Pi(x : A).B \times C \to (\Pi(x : A).B) \times (\Pi(x : A).C)$, we simply define $f \equiv \lambda t.(\mathsf{pr}_1 t, \mathsf{pr}_2 t)$, where $\mathsf{pr}_i t$ is understood as the composition of t with the i'th projection of the product $B \times C$. Conversely, we define a map in the other direction by $g \equiv \lambda u.(\mathsf{pr}_1 u, \mathsf{pr}_2 u)$, where the projections are now of the product $(\Pi(x : A).B) \times (\Pi(x : A).C)$.

To see that these maps are mutually inverse is simply an application of the η laws for the product and the dependent product for each composition. □

Lemma 3.1.3
We have a map $\mathsf{force}^\kappa : \forall\kappa. \rhd^\kappa A \to \forall\kappa.A$ *which is inverse to the map induced by* next^κ.

Proof. Firstly, we should spell out exactly what we want force^κ to be inverse to. If we have some $t : \forall\kappa.A$ we can consider the term $\Lambda_{\mathsf{clock}}\kappa.(\mathsf{next}^\kappa t)$. The claim is that we can construct an inverse to this map.

But this is more or less the way we defined the prev operation. We required that $\mathsf{prev}\kappa.\mathsf{next}^\kappa t \equiv \Lambda_{\mathsf{clock}}\kappa.t$, which ensures that mapping $u : \forall\kappa. \rhd^\kappa A$ to $\Lambda_{\mathsf{clock}}\kappa.(\mathsf{prev}\kappa.u)[\kappa]$ is an inverse to force^κ. □

From the above lemmas we can conclude that we have isomorphisms $\forall\kappa.(\mathbb{N} \times \rhd^\kappa \mathsf{Str}^\kappa) \cong (\forall\kappa.\mathbb{N}) \times (\forall\kappa.\mathsf{Str}^\kappa)$. We note also that the proof can be used to reconstruct prev from force, which will be useful to know when we consider the semantics of the theory.

At this point, the only thing missing from our theory is the clock irrelevance axiom.

Definition 3.1.4
The clock irrelevance axiom is the inference rule

$$\frac{\Gamma \vdash t : \forall\kappa.A \qquad \Gamma \vdash A\,\mathbf{type} \qquad \Gamma \vdash \kappa' : \mathsf{clock} \qquad \Gamma \vdash \kappa'' : \mathsf{clock}}{\Gamma \vdash t\kappa' \equiv t\kappa''}$$

If for some A, this rule is valid for every t, we say that A is clock irrelevant. ◦

One thing to note about this rule is the assumption $\Gamma \vdash A\,\mathbf{type}$, which ensures that κ does not occur in A. Keeping with the intuition that clock quantification is a like function type, we could say that clock irrelevance assures that any map into A from the primitive of clocks is constant. Here we are of course only considering non-dependent functions, whereas clock quantification by a clock which *does* occur in will of course have non-constant elements. This is the idea we will implement directly in the presheaf semantics in the next section.

Now we have the final ingredient for the desired isomorphism, namely the proof of $\mathbb{N} \cong \forall\kappa.\mathbb{N}$. This will also formalize the idea that functions $\mathsf{clock} \to A$ must be constant.

Proposition 3.1.5
For any type, $\Gamma \vdash A\,\mathbf{type}$ *with* $\Gamma, \kappa : \mathsf{clock} \vdash$ *we have an isomorphism* $A \cong \forall\kappa.A$.

Proof. We assume that some clock, κ_0 is available in the context Γ, which can be achieved by adding a clock constant to the language.

There is a map $A \to \forall\kappa.A$ given by taking the constant map, as a term: $f \equiv \Lambda_{\mathsf{clock}}\kappa.a$ for $a : A$, so we need to define a map in the other direction. This is where the clock constant comes in, so that we can define $g \equiv \lambda t.(t[\kappa_0/\kappa])$.

Clearly the constant a map evaluated at κ_0 is judgementally equal to a, so it remains to check that any map t is judgementally equal to the constant map at $t[\kappa_0/\kappa]$. By the η law for inhabitants of the clock quantification type, it suffices to show that $t[\kappa_0/\kappa] = t[\kappa'/\kappa]$ for each $\Gamma \vdash \kappa' : \mathsf{clock}$. But this is exactly the clock irrelevance axiom. □

3.1.2 Presheaf semantics for guarded dependent type theory

The base of the semantics for guarded dependent type theory is the a certain category of clocks and remaining ticks on these, which we will call the time category and denote by $\mathbb{T}$. The theory will then be modelled in the category of covariant presheaves on this category.

We fix for the rest of the thesis a countable set of clocks which we denote $\mathcal{N}_{\mathrm{Cl}}$. For each finite $\mathcal{E} \subset \mathcal{N}_{\mathrm{Cl}}$ we pick a $\lambda_{\mathcal{E}} \notin \mathcal{E}$, and we will frequently write $\mathcal{E}, \lambda$ as shorthand for $\{\mathcal{E}\} \cup \{\lambda\}$. If we have a map $\delta : \mathcal{E} \to \mathbb{N}$ we denote the extension to $\mathcal{E}, \lambda$ which sends λ to n by $\delta[\lambda \mapsto n]$.

Definition 3.1.6
The category $\mathbb{T}$ has as objects finite subsets of $\mathcal{N}_{\mathrm{Cl}}$, $\mathcal{E}$, together with a map to the natural numbers, $\delta : \mathcal{E} \to \mathbb{N}$. A morphism $f : (\mathcal{E}, \delta) \to (\mathcal{E}', \delta')$ is a map $\mathcal{E} \to \mathcal{E}'$ such that $\delta'(f(\lambda) \leq \delta(\lambda)$ for all $\lambda \in \mathcal{E}$. Composition and identities are simply given by composition of the underlying maps and the identity on the underlying set respectively. ◦

It is clear that the above defines a category, which is furthermore both locally small and small. We will identify the morphism and the underlying map of sets in our notation, denoting both by $f, g, h, \ldots$. The inequality requirement reflects the fact that we cannot add time to a clock, we can only ever decrease the number of ticks on it.

The most important morphism of this category is the tick on a clock, which simply decreases the number of ticks on a certain clock by one. It is the morphism $\mathsf{tick}^{\kappa} : (\mathcal{E}, \delta) \to (\mathcal{E}, \delta')$ which is the identity on $\mathcal{E}$, with $\delta'(\kappa) = \delta(\kappa) - 1$ and $\delta' = \delta$ for all other elements of $\mathcal{E}$.

There are essentially only three other morphisms to consider, the clock introduction morphisms, the clock renamings and the clock synchronization morphisms.

- A *clock introduction* is a morphism $(\mathcal{E}, \delta) \to (\mathcal{E}, \lambda, \delta[\lambda \mapsto n])$, which is the inclusion on the underlying sets.
- A *clock renaming* is a morphism $f : (\mathcal{E}, \delta) \to (\mathcal{E}', \delta')$, which is a bijection on the underlying set, and $\delta(\lambda) = \delta'(f(\lambda))$.
- A *clock synchronization* is a morphism $f : (\mathcal{E}, \delta) \to (\mathcal{E}', \delta')$ where f identifies exactly two clocks, say $\lambda, \lambda' \in \mathcal{E}$ is identified so that $f(\lambda) = f(\lambda')$. The map δ' is then defined to be the map δ where it is defined, and $\delta'(f(\lambda)) = \min \delta(\lambda), \delta(\lambda')$.

Any morphism of the time category can be written as a composition of these four kinds of morphisms.

In the presheaf category $\mathsf{Set}^{\mathbb{T}}$ we have an object which is simply the first projection, i.e, the map $(\mathcal{E}, \delta) \mapsto \mathcal{E}$. We call this presheaf the clock object and denote it by Cl.

Lemma 3.1.7
The presheaf Cl *is isomorphic to the colimit of the diagram*

$$\begin{tikzcd}
y_{(\{\lambda\},0)} & y_{(\{\lambda\},1)} \arrow[l, "\mathsf{tick}_\lambda"'] & y_{(\{\lambda\},2)} \arrow[l, "\mathsf{tick}_\lambda"'] & \cdots \arrow[l, "\mathsf{tick}_\lambda"']
\end{tikzcd}$$

Proof. Colimits are computed level-wise, so consider some $(\mathcal{E}, \delta)$. Here we have by the Yoneda lemma that $y(\lambda, n)(\mathcal{E}, \delta)$ simply contains those $\lambda' \in \mathcal{E}$ for which $\delta(\lambda') \leq n$, so the colimit is those $\lambda' \in \mathcal{E}$ such that $\delta(\lambda') \leq n$ for some n, which is of course all of them. □

Being a presheaf category, the category $\mathsf{Set}^{\mathbb{T}}$ supports the concept of orthogonality of morphisms and orthogonality of a morphism to an object. The primary reason the present category is well suited to modelling GDTT is a combination of the results shown in the section on orthogonality in the preliminaries.

We say that a morphism in $\mathsf{Set}^{\mathbb{T}}$ is invariant under clock introduction if it is orthogonal to each object of the form $y_{(\{\lambda\},n)}$. In particular, we say that a type $\Gamma \vdash A\,\mathbf{type}$ is invariant under clock introduction if the context projection is invariant under clock introduction. Note that by 1.1.9, any object invariant under clock introduction is thus orthogonal to the object of clocks.

Modelling the theory

Since we are in an extension of the theory we know how to model in any presheaf category, we only have to give interpretations for the things which we have added to the theory. This means that we have to give interpretations of

- the object of clocks, clock variables and clock quantification;
- the $\triangleright^{\kappa}$ modalities and the next^{κ} constructors;
- the $\mathsf{prev}\kappa.(-)$ operations, and the fix point combinator fix^{κ}.

We have already hinted at the interpretation of the clock primitive and variables hereof, they are simply given by the presheaf Cl and sections of the context projection (which are just the projection from the categorical product).

The modelling of $\triangleright^{\kappa}$ is what informs our intuition about the model. A type in GDTT can vary along a number of clocks, so we think of $[\![A]\!](\mathcal{E}, \delta, \gamma)$ as what type would look like if we had the clocks in $\mathcal{E}$ with the time prescribed by δ left. In other words, we consider how much of the type can be computed using the time left on the clocks declared. With this in mind, pushing a type one step into the future, which is the intuition for what $\rhd^\kappa$ does, should simply increment the type, so that for each $\gamma \in [\![\Gamma]\!]$ we have

$$[\![\rhd^\kappa A]\!](\mathcal{E}, \delta, \gamma) = \begin{cases} \{\star\} & \text{if } \delta(\gamma(\kappa)) = 0 \\ [\![A]\!](\mathcal{E}, \delta^{-\gamma(\kappa)}, \mathsf{tick}^{\gamma(\kappa)}\gamma) & \text{otherwise} \end{cases}$$

The constructor for this type is defined in much the same way, for the same reasons; a term in context $(\mathcal{E}, \delta)$ can be thought of as what that term is after computing a number of steps given in δ on the clocks in $\mathcal{E}$. Formally,

$$[\![\mathsf{next}^\kappa t]\!](\mathcal{E}, \delta) = \begin{cases} \{\star\} & \text{if } \delta(\gamma(\kappa)) = 0 \\ [\![t]\!](\mathcal{E}, \delta^{-\gamma(\kappa)}, \mathsf{tick}^{\gamma(\kappa)}\gamma) & \text{otherwise} \end{cases}$$

Clock quantification is modelled as we did the dependent product, with the presheaf of clocks playing the role of the domain type. We will leave out the modelling of prev and fix, since we will not deal with them further in this thesis. Their models and verifications of soundness et cetera can be found in [5].

The culmination of this section is to show that the model validates the clock irrelevance axiom, which would show its logical consistency. To do that we first need the following lemma:

Lemma 3.1.8
*If $\Gamma \vdash A$ **type** is a judgement of GDTT, the interpretation of A is invariant under clock introduction.*

Proof. We prove this by induction on the construction of A, noting first that invariance under clock introduction is orthogonality to the Yoneda embeddings of single clocks with any amount of time left, i.e., the objects of the form $y(\lambda, n)$.

We already know that orthogonality to specific objects is closed under composition, pullback and dependent product, which also implies that $\forall\kappa.A$ is invariant under clock introduction. we only need to show that the interpretation of $\rhd^\kappa A$ is invariant under clock introduction.

We use that the context projection is invariant under clock introduction if and only if the action of the inclusion $\iota : (\mathcal{E},) \to (\mathcal{E}, \lambda_\mathcal{E}, \delta[\lambda \mapsto n])$, where $\lambda_\mathcal{E} \notin \mathcal{E}$, is an isomorphism for each n. Assuming this, we are almost done since the restriction on $[\![\rhd^\kappa A]\!]$ are inherited from $[\![A]\!]$.

Since we have the assumption $\Gamma \vdash \kappa : \mathsf{clock}$, we know that $(\iota\gamma)(\kappa) = \gamma(\kappa)$. Either $\delta(\gamma(\kappa)) = 0$, in which case the action of the inclusion is the unique map $\{\star\} \to \{\star\}$, or $\delta(\gamma(\kappa)) = 0$ in which case the action of the inclusion is an isomorphism inherited from A. □

We now show the lemma we applied before.

Lemma 3.1.9
*Assume that $\Gamma \vdash A$ **type**. The context projection is invariant under clock introduction if and only if the action of the inclusion $\iota : (\mathcal{E}, \delta) \to (\mathcal{E}, \lambda_\mathcal{E}, \delta[\lambda \mapsto n])$, where $\lambda \notin \mathcal{E}$, is an isomorphism for each n.*

Proof. Recall that since any presheaf category is an LCCC, we have orthogonality of a morphism p, to an object X, if and only if a certain square is a pullback, from 1.1.10.

Note that $(\mathcal{E}, \lambda, \delta[\lambda \mapsto n])$ splits as the coproduct of $(\mathcal{E}, \delta)$ and $(\lambda_{\mathcal{E}}, n)$ in $\mathbb{T}$, and hence as a product in $\mathbb{T}^{\mathsf{op}}$. Since the Yoneda embedding $\mathbb{T}^{\mathsf{op}} \to \mathsf{Set}^{\mathbb{T}}$ preserves any products of $\mathbb{T}^{\mathsf{op}}$, we have by applying the definition of exponentials in presheaf categories and the Yoneda lemma that

$$\begin{aligned} X^{y(\lambda,n)}(\mathcal{E}, \delta) &\simeq \mathrm{Hom}(y(\mathcal{E}, \delta) \times y(\lambda, n), X) \\ &\simeq \mathrm{Hom}(y(\mathcal{E}, \lambda, \delta[\lambda \mapsto n]), x) \\ &\simeq X(\mathcal{E}, \lambda, \delta[\lambda \mapsto n]) \end{aligned}$$

We now need to check that the isomorphism we obtained here precomposed with c_A is the same as the action of the inclusion. This follows from the fact that the projection $y(\mathcal{E}, \delta) \times y(\lambda, n) \to y(\mathcal{E}, \delta)$ in $\mathsf{Set}^{\mathbb{T}}$ corresponds to the inclusion $\iota : (\mathcal{E}, \delta) \to (\mathcal{E}, \lambda, \delta[\lambda \mapsto n])$. □

Now we are ready to establish the validation of the clock irrelevance axiom.

Theorem 3.1.10
Assume that $\Gamma \vdash A\,\mathbf{type}$*,* $\Gamma \vdash \kappa : \mathsf{clock}$ *and consider some* $\Gamma \vdash t : \forall\kappa.A$*. For any two* $\Gamma \vdash \kappa', \kappa'' : \mathsf{clock}$*, we have that* $[\![t[\kappa'/\kappa]]\!] = [\![t[\kappa''/\kappa]]\!]$*.*

Proof. We note first that the interpretation of A and t can be written out as a commuting diagram

$$\begin{tikzcd} {[\![\Gamma]\!] \times \mathrm{Cl}} \arrow[r, "{[\![t]\!]}"] \arrow[d, "p"] & {[\![A]\!]} \arrow[d, "p_A"] \\ {[\![\Gamma]\!]} \arrow[r, "\mathrm{id}"] & {[\![\Gamma]\!]} \end{tikzcd}$$

Furthermore, the interpretations of κ' and κ'' are sections of p, meaning that in particular $[\![t]\!] \circ [\![\kappa']\!]$ and $[\![t]\!] \circ [\![\kappa']\!]$ are both lifts in the diagram. To conclude this, we need to check that each of the triangles commute; the upper triangle commutes since the interpretations of the κ's are sections, the lower triangles commute since $[\![t]\!][\![\kappa']\!] = [\![t[\kappa'/\kappa]]\!]$, which means in particular that the proposed lifts are terms of A in context Γ, i.e., sections of p_A. But since A is invariant under clock introduction, it is orthogonal to Cl, and hence there is a unique filler in the square above. This means in particular that $[\![t]\!][\![\kappa']\!] = [\![t[\kappa'/\kappa]]\!]$ and $[\![t]\!][\![\kappa']\!] = [\![t[\kappa''/\kappa]]\!]$ must coincide. □

Now that we have a model of GDTT in a presheaf category, we can consider the identity type question posed in the very beginning. It would further require that we show in the model that these identity types were invariant under clock introduction, which is done in section five of [5]. We have some sense of the desirability of intensional identity types in the theory from our investigations of these in the earlier section, so what we might want is that we could add such identity types to our theory. This is not possible without some extra work; any presheaf model supports an identity type which is simply interpreted by equality in the model, which is to say the presheaf of singletons whenever we have equality and empty set whenever we do not. So this model we are building does not justify the addition of an intentional identity type to GDTT.

Because these presheaves have at most one inhabitant at each object, they correspond in the theory to identity types with uniqueness of identity proofs. We saw earlier that this implies that the identity types are extensional and not intensional. Furthermore, we know that this means that the theory does not support decidable type checking. We do have function extensionality, since in this context it is simply equivalent to the η rule of the dependent function types.

We end this section by stating the soundness theorem and substitution lemma for our model.

Theorem 3.1.11
The interpretation of GDTT into $\mathsf{Set}^{\mathbb{T}}$ *is sound and satisfies the substitution lemma.*

Proof. The soundness is simply a matter of collecting results from this section; we need only to show that any judgemental equality of the theory is modelled as an equality in the model, which we showed along the way.

For the substitution lemma, we note that we do not need to reprove it for clock quantification, since this follows the scheme of the dependent product. We leave out the rest of the proofs for this, referring to [5] for details. □

In the above proof we left out the prev and fix operations since we have not discussed these in detail. For a proof that these can be included in the theory we refer to [5].

Submodels

We now show that the model presented here recovers the Set model under the assumption that there are no clocks in the context, and the topos of trees model in the case of a single clock.

Proposition 3.1.12
The full subcategory of $\mathsf{Set}^{\mathbb{T}}$ *of the presheaves such that the map to the terminal presheaf is orthogonal to the clock object is equivalent to* Set*.*

Proof. We wish to produce an equivalence of categories between the full subcategory of covariant presheaves on $\mathbb{T}$ that are orthogonal to the clock object, and the category of sets. Note first that the presheaves that are orthogonal to the clock object are precisely those in which every restriction map is an isomorphism. Consider the assignment of a presheaf X, to the set $X(\emptyset, !)$, and to each morphism of presheaves the map at this component, and note that this is functorial. It would suffice to show that this functor is fully faithful and essentially surjective. Since the constant presheaves (in which the restriction maps are just identities) are invariant under clock introduction this functor is clearly full and essentially surjective.

It only remains to show that it is faithful, which in this case means that any morphism of presheaves orthogonal to the clock object should be completely determined by the action at $(\emptyset, !)$, so let $\varphi : X \to Y$ be such a morphism. Since morphisms of presheaves are natural transformations, we know in particular that the following square commutes:

$$\begin{tikzcd}
X(\mathcal{E},\delta) \arrow[r, "\varphi_{(\mathcal{E},\delta)}"] & Y(\mathcal{E},\delta) \\
X(\emptyset,!) \arrow[u, "\iota"] \arrow[r, "\varphi_{(\emptyset,!)}"] & Y(\emptyset,!) \arrow[u, "\iota"]
\end{tikzcd}$$

But since ι is an isomorphism, we have that $\varphi_{\mathcal{E},\delta} = \iota \circ \varphi_{(\emptyset,!)} \circ \iota^{-1}$. This shows that φ is determined by its value along the empty set, and hence that the functor is injective on Hom-sets, i.e., faithful. □

Proposition 3.1.13

The full subcategory of $\mathsf{Set}^{\mathbb{T}}_{\mathrm{Cl}}$ of maps which are orthogonal to the clock object is equivalent to the topos of trees, or the presheaf category $\mathsf{Set}^{\omega^{\mathrm{op}}}$.

Proof. Invariance under clock introduction is still defined as the map to the terminal object being orthogonal to the object of clocks, but now the terminal object is $(\mathcal{C}, \mathrm{id})$. So where before we had constant presheaves, owing to the fact that the inclusion maps fit into pullback squares

$$\begin{tikzcd}
X(\emptyset, !) \arrow[r, "\iota"] \arrow[d] & X(\mathcal{E}, \delta) \arrow[d] \\
\star \arrow[r] & \star
\end{tikzcd}$$

and hence is the pullback of an isomorphism; now we know that each square of the form

$$\begin{tikzcd}
X(\lambda, \lambda \mapsto n) \arrow[r, "\iota"] \arrow[d] & X(\mathcal{E}, \delta) \arrow[d] \\
\{\lambda\} \arrow[r] & \mathcal{E}
\end{tikzcd}$$

is a pullback. What this tells us, is that the fibre over some λ of the map to the clock object $X(\mathcal{E}, \delta) \to \mathcal{E}$ can depend only on $\delta(\lambda)$. Note also that any morphism $\mathsf{Set}^{\mathbb{T}}/\mathcal{C}$ must preserve the fibres, so that a morphism of such presheaves is completely determined by the restrictions along maps of the form $(\lambda, \lambda \mapsto n) \to (\lambda', \lambda' \mapsto k)$, which exist precisely when $n \geq k$.

We are now ready to define a functor into the topos of trees. Consider the assignment of presheaves $X \in \mathsf{Set}^{\mathbb{T}}/\mathcal{C}$ which are invariant under clock introduction to the presheaf $\hat{X} \in \mathsf{Set}^{\omega^{\mathrm{op}}}$ defined on numbers as $\hat{X}(n) = X(\lambda_0, n)$ and on maps as $\hat{X}(n \geq k) = X((\lambda_0, \lambda_0 \mapsto n) \to (\lambda_0, \lambda_0 \mapsto k))$ for some specific λ_0. This assignment is well defined by the above argument, and clearly functorial.

In the other direction, we can for some X in the topos of trees, consider the functor defined by $\tilde{X}(\mathcal{E}, \delta) = \amalg_{\lambda \in \mathcal{E}} X(\delta(\lambda))$, with the maps defined fibrewise. To check that this is well defined, we need to check that the following square is a pullback:

$$\begin{tikzcd}
\tilde{X}(\mathcal{E}, \delta) \arrow[r, "\iota"] \arrow[d, "p"] & \tilde{X}((\mathcal{E}, \lambda), \delta[\lambda \mapsto n]) \arrow[d] \\
\mathcal{E} \arrow[r] & \mathcal{E}, \lambda
\end{tikzcd}$$

where the vertical maps are the projections onto the correct respective fibres. This is more or less immediate, since a map into $\tilde{X}((\mathcal{E}, \lambda), \delta[\lambda \to n])$ which is compatible with the inclusion $\mathcal{E} \to \mathcal{E}, \lambda$ cannot hit the factor corresponding to λ, and hence lifts to a map into $\tilde{X}(\mathcal{E}, \delta)$.

To see that these two functors constitute an equivalence of categories, we need to provide natural isomorphisms to the identity functors from each composition. Starting with an element of the topos of trees, we get $\hat{\tilde{X}}(n) = \{(\lambda_0, x) | x \in X(n)\}$, so we can define a natural transformation to the identity simply by projecting off the λ_0. It's clear that this will be a fibrewise isomorphism, and hence that it will assemble into a natural isomorphism. For the other composition, we note that we can recover the the universal property of the coproduct by letting λ vary over $\mathcal{E}$ in the square

$$\begin{tikzcd}
X(\lambda, \lambda \mapsto n) \arrow[r, "\iota"] \arrow[d] & X(\mathcal{E}, \delta) \arrow[d] \\
\{\lambda\} \arrow[r] & \mathcal{E}
\end{tikzcd}$$

and hence we have an isomorphism $X(\mathcal{E}, \delta) \simeq \amalg_{\lambda \in \mathcal{E}} X(\lambda, \delta|_\lambda)$, which is natural in $\mathcal{E}$, which is exactly what we wanted. □

This concludes our discussion of the semantics for guarded dependent type theory.

3.2 Cubical type theory

We have already briefly encountered some notions of homotopy type theory.

The discussion here will follow [7]. The purpose of this section is both to gain some insight in to the particulars of the cubical type theory and in particular to understand the utility of uniform lifting conditions in models of type theory. The syntax and semantics of cubical type theory interact heavily, so before presenting either we will discuss some algebraic structures.

We will need the concepts of distributive lattices and de Morgan algebras in the following, so we quickly recall their definitions here.

Definition 3.2.1
A *distributive lattice* is a set A with two binary operations, meet and join, denoted by $\wedge$ and $\vee$ respectively, a top elements, 1_A and a bottom element 0_A. These must satisfy the following:

$$1_A \wedge r = r, \qquad 1_A \vee r = 1_A, \qquad 0_A \wedge r = 0, \qquad 0_A \vee r = r \quad \text{and} \quad r \wedge (s \vee t) = (r \wedge s) \vee (r \wedge t).$$

○

The definition we have given above is often referred t as a bounded distributive lattice, since we have specified maximum and minimum elements. Generally, the lattice structures of interest in logic will be bounded, since the logical language requires a true and a false proposition, which becomes the top and bottom elements of the representing lattice. We can enhance a distributive lattice with an involution, and if this involution satisfies the de Morgan laws, we call it a de Morgan algebra. The definition is written out below for completeness.

Definition 3.2.2
A *de Morgan algebra* is a set A with two binary operations, meet and join, denoted by $\wedge$ and $\vee$ respectively, a top elements, 1_A, a bottom element 0_A, and an involution, $r \mapsto 1_A - r$[1]. We require that $(A, 1_A, 0_A, \wedge, \vee)$ is a distributive lattice. Furthermore, the involution must satisfy the following:

$$1_A - 0_A = 1_A, \qquad 1_A - 1_A = 0_A, \qquad 1_A - (r \vee s) = (1_A - r) \wedge (1_A - s)$$
$$\text{and} \quad 1_A - (r \wedge s) = (1_A - r) \vee (1_A - s).$$

A morphism of de Morgan algebras is a map of sets between de Morgan algebra which preserves the signatures. ○

[1] Being perfectly formal would require us to define this as some operation $\neg(-)$. Instead, we use this notation which is suggestive of the role that the involution plays in the interval, since this will be the most important example.

Whenever it is clear from the context which de Morgan algebra we are considering, we will drop the subscript on the top and bottom elements, denoting them simply by 1 and 0 respectively.

Remark 3.2.3 (Freely generated de Morgan algebras)
The de Morgan algebras which we will encounter are all freely generated on some set; first in the syntax we consider the free de Morgan algebra on a countable set of names, and in the semantics we will consider free de Morgan algebras on finite sets.

The free de Morgan algebra on a set X is denoted $\mathsf{dM}(X)$, and is obtained by adjoining a top and bottom element while imposing no relations on the joins, meets and involutions of elements in X apart from the ones required in the definition of de Morgan algebras.

The free de Morgan algebra on a set has a universal property. If we have a map of sets $f : X \to A$, and A is a de Morgan algebra, there is a unique morphism of de Morgan algebras $\overline{f} : \mathsf{dM}(X) \to A$, given simply by freely extending the original morphism such that it preserves the signature. Another way to obtain the results that follow from this is to extend the free de Morgan algebra functor (which is extended by this universal property to morphisms) to a monad, which is the approach taken in [7].

We will also make use of freely generated distributive lattices, although here the morphisms will not play a role. Constructing these follow the same pattern as for de Morgan algebras. ○

We fix throughout this section a countable set of names, $\mathcal{N}$, assumed not to contain the symbols 1 and 0. We denote the elements of this set by lower-case letters, $i, j, k, \ldots$, and we denote finite subsets of the names by capital letters, $I, J, K, \ldots$. The *interval*, denoted $\mathbb{I}$, is defined as the free de Morgan algebra on $\mathcal{N}$.

We have already encountered function extensionality and univalence as examples of extensionality principles. In this section we will also see an extensionality principle for the dependent sum. The extensionality principle we are looking for should express the fact that path equality of two pairs can be obtained from path equality of the projections.

3.2.1 Syntax for cubical type theory

We again take for granted the usual dependent type theory, meaning Π types, Σ types and the natural numbers, as part of cubical type theory. The crucial addition from cubical type theory is the model of an intensional identity type as the path space, which we will construct as a mapping space from a syntactic interval. As with the clocks in GDTT, this interval will be regarded as a primitive, so that we cannot apply our typical type formers to it. We denote the interval again by $\mathbb{I}$, overloading this notation as we overload the word.

Given a context, $\Gamma \vdash$, and a name, $i \in \mathcal{N}$, which does not occur in Γ, we can declare that name in the context to get $\Gamma, i : \mathbb{I} \vdash$. This is the exact same situation as we saw with the clocks, but with the interval we can furthermore give a sort of typing judgement $\Gamma \vdash r : \mathbb{I}$. Whenever the r depends only on names declared in Γ, or more specifically, when r can be obtained from the names declared in Γ via meets, joins and involutions. In this way, we introduce the signature of the de Morgan algebra $\mathbb{I}$, into our formal language. In particular, we need substitution on terms of the syntactic interval and terms depending on these. Not only do we want to be able to write $t[r/i]$, were i is a term occurring in t, and r is some term of type $\mathbb{I}$. Since the entire signature of de Morgan algebras, and in particular $\mathbb{I}$, is in our formal language, we also have the face substitutions; $t[0/i]$ and $t[1/i]$. There is a variation on the usual substitution notation which we will will sometimes employ for these face substitutions; instead of writing $A[0/i]$ or $t[1/i]$, we will write $A(i0)$ and $t(i1)$. This will simplify some later definitions involving the iteration of several face substitutions. We also note that the equalities of elements of $\mathbb{I}$ as a de Morgan algebra propagates into the theory as judgemental equalities.

In this way, the interval is much more interactive than the clocks; we can form terms which depend on previous terms, and as a consequence the substitutions in terms make sense. In the case of clocks we thought of types in the context of a clock along which time could tick. For the interval, we think of the declared variables as dimensions along which the type can vary from 0 to 1. Under this analogy, we can think of a type in the context of no interval variables as a point, a type in the context of a single interval variable is a line and so forth. This is the idea which we use to construct the presheaf semantics for cubical type theory.

As for the syntactic clocks, we have to extend our notion of context substitution to include substitutions on $\mathbb{I}$. This follows the same pattern as the definition for types and clocks, so we leave out the statement, and will not mention this fact in the future when we encounter variations on type theory.

We are now ready to define the path type.

Definition 3.2.4
To form a path type, we need two terms of a single type. As an inference rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash a' : A}{\Gamma \vdash \mathsf{Path}_A(a)(a')\,\mathbf{type}}$$

We introduce and eliminate paths in the same way we would dependent functions or inhabitants of $\forall\kappa.A$. We give the elimination and introduction rules simultaneously, since the elimination rule is need for typing in the introduction rule:

$$\frac{\Gamma \vdash t : \mathsf{Path}_A(a)(a') \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash tr : A} \qquad \frac{\Gamma \vdash A\,\mathbf{type} \qquad \Gamma, i : \mathbb{I} \vdash f : A}{\Gamma \vdash \Lambda_{\mathbb{I}} i.t : \mathsf{Path}_A(t0)(t1)}$$

We want to enforce the endpoints of the paths at the judgemental level:

$$\frac{\Gamma \vdash t : \mathsf{Path}_A(a)(a')}{\Gamma \vdash t0 \equiv a : A} \qquad \frac{\Gamma \vdash t : \mathsf{Path}_A(a)(a')}{\Gamma \vdash t1 \equiv a' : A}$$

As with the function types we want β and η laws:

$$\frac{\Gamma, i : \mathbb{I} \vdash t : A \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \Lambda_{\mathbb{I}} i.t(r) \equiv t[r/i] : A[r/i]} \qquad \frac{\Gamma \vdash t : \mathsf{Path}_A(a)(a')}{\Gamma \vdash t \equiv \Lambda_{\mathbb{I}} i.(ti) : \mathsf{Path}_A(a)(a')}$$

Here we again note that there is a different statement of the η law, given by judgemental extensionality for paths, as was the case for dependent function, but we do not give the rule here. ○

The motivation for introducing the path type is to add an identity type to the theory, and so we have some justification to do.

Example 3.2.5
Before we discuss the elimination rule for the path type as an equality type, we give some examples of its use as an identity type.

- There is a constant path at each term, $a : A$. This corresponds to the existence of refl_t in the case of the identity types. It is constructed simply as $\Lambda_{\mathbb{I}} i.a$, and we note that this means that the path type has the same introduction rule as the identity type, since the only assumption needed was $\Gamma \vdash a : A$. We denote this constant path at a by 1_a.

- In a similar spirit, we can define the inversion of a path, simply by $\Lambda_{\mathbb{I}} i.(t[i/1-i])$. This shows symmetry of the identity relation. To see that this path inversion indeed gives us symmetry, it suffices to check that the path has the correct endpoints. As noted before, the equalities of $\mathbb{I}$ as a de Morgan algebra is also true in the theory, and therefore we have that $\Lambda_{\mathbb{I}} i.(t[i/1-i])0 \equiv t[1-i/i][0/i]$ which is syntactically the same as $t[1/i]$, and vice versa for the other endpoint, which is what we wanted. We denote the inversion of a path p by p^{-1}.

- For any function, the images of terms with a path between them, has a path between them, witnessing the fact that equal terms have equal images. Since the path type is only defined within a single type, we can only write out this term for non-dependent functions. We write it out as an inference rule:

  $$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash a' : A \qquad \Gamma \vdash f : A \to B \qquad \Gamma \vdash p : \mathsf{Path}_A(a)(b)}{\Gamma \vdash \Lambda_{\mathbb{I}} i.(fpi) : \mathsf{Path}_B(fa)(fa')}$$

  Again it is clear from just plugging in the face substitutions that this path indeed has the correct endpoint.

- We can show 2.1.9 directly. Specifically, we want to show that there is a path from (a, a', p), where p is a path from a to a', to the triple $(a, a, 1_a)$. For convenience we define the path in the other direction, but we already know that this does not matter since we can invert paths. We can define this path as

  $$\Lambda_{\mathbb{I}} i.(a, pi, \Lambda_{\mathbb{I}} jp(i \wedge j)).$$

  Unfolding this term in words, we want the first term in the triple to remain constant. On the second term we apply the path p, so that we know it will start as a and end at a'. The third coordinate is is a truncated path; we run along p, until we reach i, at which point we stop. This means that $p(i \wedge j)$ is a path from a to pi, which is exactly what we wanted.

○

Now that we are slightly more familiar with the path type we might think that it makes a very good approximation of the Martin-Löf identity type. We are only missing a composition of paths in the above for it to be an equivalence relation, and it is very nice to have function extensionality as a provable feature. It turns out that path composition is more complicated to get to, and to introduce it, we need a more general composition operation, and a discussion of systems and the face lattice. Before moving on to this, we motivate the choice of path types as identity types by presenting function extensionality in this type theory.

Example 3.2.6 (Function extensionality)
We have a term given as the conclusion of

$$\frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash g : \Pi(x : A).B \qquad \Gamma \vdash p : \Pi(x : A).\mathsf{Path}_B(fx)(gx)}{\Gamma \vdash \Lambda_{\mathbb{I}} i.\lambda x.pxi : \mathsf{Path}_{\Pi x:A.B}(f)(g)}$$

where we are simply constructing a function by evaluating the paths from fx to gx. It is clear that this term has the correct faces, i.e., that when restricted to $i = 0$ we get f, and when we restrict to $i = 1$ we get g. ◦

To obtain the extensionality principle for Σ types we need the transport operation, so we will present it towards the end of this subsection.

We define the face lattice $\mathbb{F}$ as the free distributive lattice on the symbols $(i = 1)$ and $(i = 0)$, where i ranges over $\mathcal{N}$, modulo the identification $(i = 0) \wedge (i = 1) = 0$. The reason for introducing this concept is to develop a way to specify sub cubes, in the sense that we leave out some faces, internally in the theory. This method allows us to think of these cubes with some faces left out as simply compatible unions of cubes. We can declare elements of $\mathbb{F}$ in a context as long as all the variables in it are declared. For example we have $i : \mathbb{I}, j : \mathbb{I} \vdash (i = 1) \vee (j = 0) : \mathbb{F}$, but we do not have $i : \mathbb{I} \vdash (i = 1) \vee (j = 0) : \mathbb{F}$. These examples are of course also valid in a broader context, i.e, if the context contains more types. Furthermore, we can add a formula to a context if it is valid in that context by the rule $\dfrac{\Gamma \vdash \varphi : \mathbb{F}}{\Gamma, \varphi \vdash}$. We think of adding a formula φ to the context as restricting that context in a way which depends on φ.

As discussed earlier, types in cubical type theory can be thought of as hypercubes, varying along each dimension declared in the context. The generators of $\mathbb{F}$ can be thought of as prescribing that the "coordinate" along a certain dimension is either 0 or 1. So a generator of $\mathbb{F}$ is a face of a cube, while conjunctions are iterated faces and disjunctions are unions of faces of varying dimensions. We can for instance consider the boundary of some set of names I, which is $\bigvee_{i \in I}(i = 0) \vee (i = 1)$, so the union of every face of the I cubes. Because this formula for the boundary contains only the symbols on I it will be the largest element of $\mathbb{F}$ depending only on symbols in I, but it can of course not actually be the top element, seeing as it will be incomparable to elements depending on symbols not in I.

Example 3.2.7
Now that we understand the inhabitants of $\mathbb{F}$, we can more clearly describe the restriction of contexts. We think of a type in the context as a hypercube in dimensions I, and if we add φ to the context, this will give a restriction of that cube. So in the context of i and j a type will be a cube, and here adding the formula $(i = 0) \vee (j = 1)$ means that a type will be instead the union of the top of the cube and the right side of the cube. We illustrate this restriction of a context Γ with the type A in the diagram below. We will allow ourselves the slightly simplified notation for substitutions of interval variables, writing, e.g., $A(i0)$ for the substitution $A[0/i]$.

$$\begin{tikzcd} A(i0)(j1) \arrow[r, dash, "A(j1)"] \arrow[d, dash, "A(i0)"'] \arrow[dr, phantom, "A(i0 \vee j1)"] & A(i1)(j1) \\ A(i0)(j0) & \end{tikzcd} \qquad \begin{tikzcd} A(i0)(j1) \arrow[r, dash, "A(j1)"] \arrow[d, dash, "A(i0)"'] \arrow[dr, phantom, "A"] & A(i1)(j1) \arrow[d, dash, "A(i1)"] \\ A(i0)(j0) \arrow[r, dash, "A(j0)"'] & A(i1)(j0) \end{tikzcd}$$

In diagrams such as the above, we will slightly abuse notation and write $A(\varphi)$ for $\Gamma, \varphi \vdash A$. At this point we already have the goal in sight. We want to complete types restricted to open box shapes, such as $(i = 0) \vee (j = 0) \vee (j = 1)$, into full types, or types in the context Γ. ◦

Another way to think of $\mathbb{F}$ is by considering the congruences induced by its elements. A generator of $\mathbb{F}$, say $(i = 0)$, can be mapped to the congruence on $\mathbb{I}$ identifying i with 0. The conjunction of independent generators $\varphi = \bigwedge_{i \in I, r_i \in \{0,1\}} (i = r_i)$ is the disjunction of the congruences, or in other words the union of the induced relations on $\mathbb{I}$. The disjunction of two formulas are sent to the meet of the congruences induced by each formula, i.e., $r = s : \mathbb{I}$ mod $\varphi \vee \psi$ if $r = s : \mathbb{I}$ mod φ and $r = s : \mathbb{I}$ mod ψ. Concretely, if we have $\Gamma, (i = 0) \vdash r \equiv s : \mathbb{I}$, this means that r and s define the same element of I when we replace the name i by 0.

Up to this point we have only analysed the case of adding a single formula of $\mathbb{F}$. We need to specify what happens when we add several elements of $\mathbb{F}$ to a context. In this case, we say that $\Gamma, \varphi, \psi \vdash$ generates the congruence of $\varphi \wedge \psi$.

Definition 3.2.8

A *system* is a collection of types in some context Γ with a variable restriction. We introduce the judgement $\Gamma \vdash [\varphi_1 A_1, \ldots, \varphi_n A_n]$, which is read as $[\varphi_1 A_1, \ldots, \varphi_n A_n]$ is a system in the context of Γ.

We will assume that we have formulas $\varphi_1, \ldots, \varphi_n : \mathbb{F}$ in the context Γ, and $\bigvee \varphi_i = 1$. A system represents compatible types via subtypes in given restriction, and is introduced as follows:

$$\frac{\Gamma, \varphi_i \vdash A_i \,\textbf{type} \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash A_i \equiv A_j \qquad 1 \leq i, j \leq n}{\Gamma \vdash [\varphi_1 A_1, \ldots, \varphi_n A_n]}$$

Just as we can have a system of compatible types, we can have a system of compatible terms, sharing the same type. This type must be unrestricted, so that we can think of the system of terms being completely contained in a single type.

$$\frac{\Gamma \vdash A \,\textbf{type} \qquad \Gamma, \varphi_i \vdash t_i : A \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash t_i \equiv t_j : A \qquad 1 \leq i, j \leq n}{\Gamma \vdash [\varphi_1 t_1, \ldots, \varphi_n t_n] : A}$$

If we can derive some judgement in the context of each φ_i, we can conclude the it holds in the context of Γ.

$$\frac{\Gamma, \varphi_i \vdash J \qquad 1 \leq i \leq n}{\Gamma \vdash J}$$

For both systems on types and terms, if a single formula is true, which is to say that $\varphi_k = 1$ for some $k \leq n$, the system collapses to just the i'th component.

$$\frac{\Gamma \vdash [\varphi_1 A_1, \ldots, \varphi_n A_n] \qquad \Gamma \vdash \varphi_k = 1 : \mathbb{F}}{\Gamma \vdash [\varphi_1 A_1, \ldots, \varphi_n A_n] \equiv A_k} \qquad \frac{\Gamma \vdash [\varphi_1 t_1, \ldots, \varphi_n t_n] : A \qquad \Gamma \vdash \varphi_k = 1 : \mathbb{F}}{\Gamma \vdash [\varphi_1 t_1, \ldots, \varphi_n t_n] \equiv t_k : A}$$

We allow for the degenerate case where $n = 0$ in the rules. ○

Because we allow for the case $n = 0$ in the above definition we get an empty system in any context (both a type system and a term system). In particular we have $\Gamma \vdash []$ and $\Gamma \vdash [] : A$ for any context such that $\Gamma \vdash 0 = 1 : \mathbb{F}$.

The primary reason to introduce the face lattice and systems was to introduce partial elements of a type, which we did, since these correspond to judgements of the form $\Gamma, \varphi \vdash u : A$. We now want to define the notion of such a partial element being extensible, or having a completion as an element of the full type. If we have an inhabitant of the full type, $\Gamma \vdash a : A$, and this term agrees with our partial one, formally we write $\Gamma, \varphi \vdash u \equiv a : A$, we write $\Gamma \vdash a : A[\varphi \mapsto u]$. In such a case, we say that $\Gamma \vdash a : A$ is a completion of $\Gamma, \varphi \vdash u : A$. If we consider the situation in 3.2.7 with the additional assumption that $\Gamma, \varphi \vdash u : A$, if we had $\Gamma \vdash a : A[\varphi \mapsto u]$, this would be a witness to the fact that u could be extended to a full square.

Note that this is more information than simply the faces, which should be clear from the fact that $(i = 0) \vee (i = 1) \vee (j = 0) \vee (j = 1) \neq 0$. We can generalize this notion to systems, so that we can write $\Gamma \vdash a : A[\varphi_1 \mapsto u_1, \ldots, \varphi_n \mapsto u_n]$. This we take to mean that $\Gamma \vdash a : A$ as before, with the condition that $\Gamma, \varphi_i \vdash a \equiv u_i : A$ for each $i \leq n$.

Before we give the composition operation and the rule for it, we will consider the example of composing paths. We need this operation to justify the transitivity of the path identifications.

Example 3.2.9
Consider the case where $\Gamma \vdash a, b, c : A$, and we have paths $\Gamma \vdash p : \mathsf{Path}_A(a)(b)$ and $\Gamma \vdash p' : \mathsf{Path}_A(b)(c)$. We can view the composition of these paths as a lid on the box:

$$\begin{tikzcd}
a \arrow[r, dashed] & c \\
a \arrow[u, "1_a"] \arrow[r, "p"] & b \arrow[u, "p'"']
\end{tikzcd}$$

since we identify paths with functions from the interval. If we index the vertical dimension by i and the horizontal dimension by j, we think of this case as having two elements of $\Gamma, i : \mathbb{I}, j : \mathbb{I} \vdash A\,\mathbf{type}$, specifically $1_a : A(i0)$ and $p' : A(i1)$, where if we further restrict to $A(j0)$, these terms are connected. In the notation we developed previously, we write $p : A(j0)[(i = 0) \mapsto a, (i = 1) \mapsto b]$, where we note that $a \equiv 1_a(j0)0$ and $b \equiv p'(j0)1$. We would like to conclude that we have some $p'' : A(j1)[(i = 0) \mapsto a, (i = 1) \mapsto c]$, so that the extensibility of the system of a and b at $j = 0$ to a path between the two implies the extensibility of the system of a and c to a path between them. The crucial assumption here is that the system we want to extend is obtained as the endpoint of a path which starts at an extensible system. Since the extension thus obtained will vary with i, the variable i should be bound in this term. $\circ$

We want to give an operation which formalizes the existence of the composition given above. It can be given as the rule

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A\,\mathbf{type} \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A(i0)[\varphi \mapsto u(i0)]}{\Gamma \vdash \mathsf{comp}^i A[\varphi \mapsto u]a_0 : A(i1)[\varphi \mapsto u(i1)]}$$

Here u is a system in context φ. The operation comp^i binds i, since as we saw in 3.2.9 we want the composition to act as a function in $\mathbb{I}$.

The composition operation is required to be compatible with substitutions, so consider a substitution $\sigma : \Gamma \to \Delta$, and j is not a declared interval variable in Δ. Define σ' to be the substitution σ which additionally substitutes j for i; in this case we have that

$$(\mathsf{comp}^i A[\varphi \mapsto u]a_0)\sigma \equiv \mathsf{comp}^j A\sigma'[\varphi\sigma \mapsto u\sigma']a_0\sigma.$$

At this point we notice a problem; the composition structure is supposed to give us *the* composition, whereas in the example we gave in 3.2.9 we made no effort towards any sort of uniqueness. The problem is that we have only specified the faces of the cube in 3.2.9, and we need a way to compute the filling of the cube. First we note that with the composition operation we obtain the path composition as

$$\frac{\Gamma \vdash p : \mathsf{Path}_A(a)(b) \qquad \Gamma \vdash p' : \mathsf{Path}_A(b)(c)}{\Gamma \vdash \mathsf{comp}^i A[(i = 0) \mapsto 1_a j, (i = 1) \mapsto p' j]pi : \mathsf{Path}_A(a)(c)}$$

where we again view a as the constant path at a in the variable j. We also get compositions for each truncated path, i.e, the paths that are stopped at a certain j. These truncated compositions can then be assembled into a filler for the square, so that we have a path from the assumed extension to the one obtained via comp. We call these fillers Kan fillers, and we can formally define them using the substitutions $[i \wedge j/i]$ and comp operations as follows:

$$\Gamma, i : \mathbb{I} \vdash \mathsf{fill}^i A[\varphi \mapsto u]a_0 \equiv \mathsf{comp}^j A[i \wedge j/i][\varphi \mapsto u[i \wedge j/i], (i = 0) \mapsto a_0]a_0 : A$$

Here we assume j to not be declared in Γ, so that comp^j makes sense.

We need to verify that these proposed fillers actually serve the desired purpose, so we need to check three equalities. We let define v as $\mathsf{fill}^i A[\varphi \mapsto u]a_0 : A$ in the following.

- We need the restriction of the filler to the already specified face to be the same as this specification, so that $v(i0) \equiv a_0 : A(i0)$.
- The restriction to the path of systems needs to return the same system, i.e., we need that $\Gamma, \varphi, i : \mathbb{I} \vdash v \equiv u$.
- The restriction to the unspecified face, $i1$, must be the some as what we obtain via comp. This holds by definition.

So if we have a composition operation, we automatically have a Kan filling operation. We want to give the definition of the composition operation by induction of the type A. Recall that we are in a theory which has only dependent sum, dependent product, natural numbers and path types. If we can define the composition structure on each of these given composition structures on the constituents, we will have defined it on every A such that $\Gamma \vdash A\,\mathbf{type}$ is a valid judgement of cubical type theory. We also need to use the filling operation for this definition, but as we just saw this does not require an extra assumption.

Definition 3.2.10
Assume that $\Gamma \vdash A\,\mathbf{type}$, $\Gamma, x : A \vdash B\,\mathbf{type}$, and that we have composition structures on A and B in their respective contexts. In each case we assume that $\Gamma, \varphi, i : \mathbb{I} \vdash u : C$ and that we have an extension at $i0$, $\Gamma \vdash c : C(i0)[\varphi \mapsto u(i0)]$.

- Consider $C \equiv \Sigma(x : A).B$. First we use the fill operation to define the term which will determine the a for which we use the composition structure on $B[a/x]$ induces by the one we have on B. Then we can simply define the composition as the coordinate-wise composition, first defining:

  $$\Gamma, i : \mathbb{I} \vdash a \equiv \mathsf{fill}^i A[\varphi \mapsto \mathsf{pr}_1 u]\mathsf{pr}_1 c : A, \qquad \Gamma \vdash c_1 \equiv \mathsf{comp}^i A[\varphi \mapsto \mathsf{pr}_1 u]\mathsf{pr}_1 c : A(i0)$$
  $$\text{and } \Gamma \vdash c_2 \equiv \mathsf{comp}^i B[a/x][\varphi \mapsto \mathsf{pr}_2 u]\mathsf{pr}_2 c : B[a/x]$$

  and we can then define $\mathsf{comp}^i C[\varphi \mapsto u]c \equiv (c_1, c_2) : C(i1)$.

- Consider $C \equiv \Pi(x : A).B$. Since the composition will be a dependent function, it suffices to define it on each input, so let $\Gamma \vdash a_1 : A(i1)$. For us to apply the function we already have, we need to move the a_1 "down" to $A(i0)$, so to speak. Towards this, we define

  $$\Gamma, i : \mathbb{I} \vdash a \equiv \mathsf{fill}^i\,[\,]\,a_1 : A[1 - i/i], \qquad \text{and} \qquad a\Gamma i, \mathbb{I} \vdash a' \equiv a[1 - i/i] : A$$

So now we want to define the compositions value on a_1, which we do using the composition structure for B:

$$\Gamma \vdash (\mathsf{comp}^i C[\varphi \mapsto u]c)a_1 \equiv \mathsf{comp}^i B[a_1/x]\,[\varphi \mapsto ua'](ca'(i0)) : B[a'/x](i0)$$

- Consider $C \equiv \mathbb{N}$, noting that we are still in the context Γ. We can simply give the definition via the recursion principle for the naturals, so that we only need to specify a composition on 0 and $s(n)$ given the structure on composition on n. Here we can simply define

$$\begin{aligned}&\Gamma \vdash \mathsf{comp}^i\mathbb{N}[\varphi \mapsto 0]0 \equiv 0 : \mathbb{N}\\ &\Gamma \vdash \mathsf{comp}^i\mathbb{N}[\varphi \mapsto s(n)](s(n_0)) \equiv s(\mathsf{comp}^i\mathbb{N}[\varphi \mapsto s(n)]n_0] : \mathbb{N}\end{aligned}$$

- Assume further that $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$, so that $\Gamma \vdash \mathsf{Path}_A(a)(a')\,\textbf{type}$, and let $C \equiv \mathsf{Path}_A(a)(a')$. Here we can define the composition directly by

$$\Gamma \vdash \mathsf{comp}^i C[\varphi \mapsto u]c \equiv \Lambda_{\mathbb{I}}j.\mathsf{comp}^i[\varphi \mapsto pj, (j = 0) \mapsto a, (j = 1) \mapsto a']u : C(i1)$$

Note that in each case, compatibility of the original composition structures with substitutions implies compatibility of the newly defined ones. ◦

Now we have transitivity of the path type, so we know that it defines an equivalence relation on terms. The elimination rule for the identity type we presented in the previous section on general type theory expresses in some sense that it is the smallest such equivalence relation. To obtain this for path types, we will need to define transport for path types, in analogy to transport for the identity type.

Example 3.2.11
Recall that transport ask for a map $P[a/x] \to P[a'/x]$ whenever we have a path $p : \mathsf{Path}_A(a)(a')$, all in the context Γ. Consider the empty transport, corresponding to the empty system:

$$\Gamma \vdash \mathsf{trans}^P(p, -) \equiv \mathsf{comp}^i A\,[\,]\,a : P(i1)$$

The point here is that we can think of P as a type in context $\Gamma, i : \mathbb{I} \vdash$, and the path is then a system which is extensible at 0, since this just means that the type $P[a/x]$ is inhabited. The composition than transports this inhabitant to $P(i1)$, or the type $P[a'/x]$. When the type family in which the transport happens is not important or is clear, we will write p_* for $\mathsf{trans}^P(p, -)$. ◦

Now that we have transport, we can justify the identity type elimination rule for the path type. Specifically, we want to produce a full element of a type family $\Gamma, x : a, y : a, p : \mathsf{Path}_A(x)(y) \vdash C$ given a partial element at each $a, a, 1_a$. But as we have seen, any such triple (a, b, p) is path equal to $(a, a, 1_a)$, and so we can transport the partial element along these paths to obtain a full element.

This does not justify the computation rules for identity types, so we still have some work left. The β rule for identity types would follow if we knew that transport along the constant path did not change the input, even judgementally. This does not hold in general, so the identity type which we have presented is only a weak identity type, which means an identity type without the judgemental computation rule, but with a propositional (relative to itself) computation rule. A fix for this is the one proposed by Swan, who constructs a type equivalent to the identity type using the weak identity type in [15].

The final thing we will see in the theory of cubical type theory is the extensionality for Σ types. Three is a standard construction which gives an equivalence of the identity type in Σ types and a Σ type of identity types. What we give here is a simpler construction, which merely shows logical equivalence and not type equivalence of the statements. As a consequence, the proposition below cannot be considered fully proof relevant.

Proposition 3.2.12
Assume that $w, w' : \Sigma(x : A).B$. We can construct maps

$$\mathsf{Path}_{\Sigma(x:A).B}(w)(w') \to \Sigma(p : \mathsf{Path}_A(\mathsf{pr}_1 w)(\mathsf{pr}_1'))\mathsf{.Path}_{B[\mathsf{pr}_1 w/x]}(p_*(\mathsf{pr}_2 w))(\mathsf{pr}_2 w')$$

and

$$\Sigma(p : \mathsf{Path}_A(\mathsf{pr}_1 w)(\mathsf{pr}_1'))\mathsf{.Path}_{B[\mathsf{pr}_2 w/x]}(p_*(\mathsf{pr}_2 w))(\mathsf{pr}_2 w') \to \mathsf{Path}_{\Sigma(x:A).B}(w)(w').$$

In particular, we can deduce equality of a pair by providing a pair of equalities, since the propositional equivalent of having two such functions is exactly a biimplication.

Proof. Consider some $q : \mathsf{Path}_\Sigma(w)(w')$. We obtain a path in A by the first projection, i.e., by $\Lambda_{\mathbb{I}} i.\mathsf{pr}_1(qi)$, so it remains only to construct a path in $\mathsf{Path}_{B[\mathsf{pr}_1 w/x]}(q_*(\mathsf{pr}_2 w))(\mathsf{pr}_2 w')$. For this we transport along the truncated path; at each i, we have some $\mathsf{pr}_2(qi) : B[\mathsf{pr}_1 qi/x]$. This can be transported to $B[\mathsf{pr}_2 w'/x]$ via the path $\Lambda_{\mathbb{I}} j.q(i \vee j)$. This gives us a path which goes from $q_*(\mathsf{pr}_2 w)$ to $(1_{\mathsf{pr}_2 w'})_*(\mathsf{pr}_2 w')$, which we can postcompose with the inverse of the Kan filler associated to transport.

To get a map in the other direction, assume that we have some

$$t : \Sigma(p : \mathsf{Path}_A(\mathsf{pr}_1 w)(\mathsf{pr}_1'))\mathsf{.Path}_{B[\mathsf{pr}_1 w/x]}(p_*(\mathsf{pr}_2 w))(\mathsf{pr}_2 w').$$

We wish to produce a path from w to w', which means to specify some term in $\Sigma(x : A).B$ to each i with the zero term being w and the 1 term being w'. It would be sufficient to specify (r, r') with i free in both r and r', such that r is a path from $\mathsf{pr}_1 w$ to $\mathsf{pr}_1 w'$, and $r'0 \equiv \mathsf{pr}_2 w$, $r'1 \equiv \mathsf{pr}_2 w'$ with $r'i : B[ri/x]$. To obtain this path, we consider each coordinate of t and do path induction on the first projection, i.e., we assume that $\mathsf{pr}_1 w \equiv \mathsf{pr}_1 w'$ and that $\mathsf{pr}_1 t \equiv 1_{\mathsf{pr}_1 w}$. This means that we already have on part of the pair, since we can use the constant path at $\mathsf{pr}_1 w$. Furthermore, the r' is just a path of type $\mathsf{Path}_{B[\mathsf{pr}_1 w/x]}(p_*(\mathsf{pr}_2 w))(\mathsf{pr}_2 w')$, which is exactly the second projection of t. □

3.2.2 Presheaf semantics for cubical type theory

In this section we will discuss the semantics of cubical type theory in the category of cubical sets. We will first need to define cubical sets, since there are many presentations of this category in the literature, and we then move on to giving some homotopical structure in the model.

The category of cubical sets is a presheaf category on a cube category which we now define.

Definition 3.2.13
The category Cube has as objects finite subsets of $\mathcal{N}$. A morphism $f : I \to J$ is a map of sets $J \to \mathsf{dM}$. ○

We will need to check that the above defines a category; we take as identities the inclusion $I \hookrightarrow \mathsf{dM}$. Composition of two morphisms $f : K \to \mathsf{J}$ and $g : J \to \mathsf{I}$ is given by postcomposing with the unique extension of g to $\overline{g} : \mathsf{dM}(J) \to \mathsf{dM}(I)$.

Definition 3.2.14
A *cubical set* is a presheaf $X \in \mathsf{Set}^{\mathsf{Cube}^{\mathsf{op}}}$. As usual for presheaf categories, a morphism of cubical sets is a natural transformation. ◦

As in the guarded recursion example, we immediately get an interpretation of the usual constructs of dependent type theory in cubical sets, simply be specializing the general presheaf model. We shall soon see how to add the interval to this model, and then the path types will be constructed as we did the dependent functions and the clock quantifications. Before then, we will make some comments on the homotopy theory that can be implemented in cubical sets.

For readers familiar with simplicial sets, the setting of cubical sets should come naturally. As for simplicial sets, there is a canonical way to realize a cubical set as a CW-complex, and the category of cubical sets can be endowed with a model structure which makes this canonical realization a Quillen equivalence. For an investigation of questions of cubical sets as a model for topological spaces and classical homotopy theory see [6]. The reason to call this similarity to the readers attention at this point, is to recall the well-known Kan lifting condition, usually presented in the context of simplicial sets. There are certain maps of simplicial sets called horn inclusions, and we say that the morphism is a Kan fibration. If the morphism to the terminal simplicial set from some other simplicial set X, we say that X is a Kan complex, or simply that it is Kan. We will now review a similar notion for cubical sets. We saw in the previous section how to specify an open box shape on an I-cube, and we will reuse the notion here.

Definition 3.2.15
We define the *free-standing I-cube* to be the Yoneda embedding of I. In particular, the free-standing cube on any singleton set is the interval, which we denote $\mathbb{I}$. ◦

Since the interval is important, we will dedicate a small lemma to its representation as a cubical set.

Lemma 3.2.16
The cubical set $\mathbb{I}$ is evaluated on objects as $\mathbb{I}(I) = \mathsf{dM}(I)$, and is the Yoneda embedding of a singleton.

We can now give the interpretation of the interval and path types in the model. We define $[\![\mathbb{I}]\!] = \mathbb{I}$, so that the presheaf interpreting the syntactic interval is the semantic interval described above. Because the interval is syntactically a primitive, it can never depend on the context, so the context extension $\Gamma.\mathbb{I}$ is simply the product. Path types are obtained in the same way dependent functions were, with a small modification. The path type of CTT remember the endpoints, and this is information that needs to be encoded in the model. To do this we need to utilize the constant presheaf 2, which assigns to each I the two element de Morgan algebra. It is a trivial observation that two sections to a projection $A \to \Gamma$ can be packaged as a map $\Gamma \times 2 \to A$, and we note furthermore that 2 is a subfunctor of $\mathbb{I}$, with the map $2(I) \to \mathbb{I}(I)$ simply given by including $0, 1$ in the de Morgan algebra $\mathbb{I}(I)$.

Definition 3.2.17

Assume that we have $\Gamma \vdash A\,\mathbf{type}$, and $\Gamma \vdash t, t' : A$, and we wish to give the interpretation of a term of the type $\mathsf{Path}_A(t)(t')$. Given this data, we obtain the following diagram in the model:

$$\begin{tikzcd}
\llbracket\Gamma\rrbracket \times 2 \arrow[r, "(\llbracket t\rrbracket{,}\llbracket t'\rrbracket)"] \arrow[d, hook] & \llbracket A\rrbracket \arrow[d] \\
\llbracket\Gamma\rrbracket \times \mathbb{I} \arrow[r] & \llbracket\Gamma\rrbracket
\end{tikzcd}$$

This diagram commutes simply because each composition is the projection, so there is no information there. We have established that a path in A should be interpreted as a map $\llbracket\Gamma\rrbracket \times \mathbb{I} \to \llbracket A\rrbracket$, and so we to require that the triangles in the diagram above commute. That the lower triangle commutes witnesses the fact that the path type is constructed in the context Γ, and the upper triangle commuting witnesses the fact the endpoints are as specified.

The interpretation of the path type is then simply the presheaf of such terms; there is an explicit definition given in the same format as the one we gave for the dependent product. ○

Now we know how to interpret path types, and the only thing missing is the interpretation of a composition structure. To obtain this definition, we will discuss the uniform Kan condition for cubical sets.

Note now that if we have a face formula φ, on generators among I, we get an inclusion of cubical sets, $\sqcup_\varphi \to \square^I$. The cubical set $\sqcup_\varphi$ is simply generated by the faces declared in φ, together with the compatibilities required. Each $\gamma \in \Gamma(I)$ can be identified with a map $\square^I \to \Gamma$ by the Yoneda lemma, and hence a term of A at I is a lift of this map

$$\begin{tikzcd}
 & A \arrow[d] \\
\square^I \arrow[r] \arrow[ur, dashed] & \Gamma
\end{tikzcd}$$

Furthermore, partial elements of a type in the context of the systems introduced in the previous section can be visualized as

$$\begin{tikzcd}
\sqcup_\varphi \arrow[r] \arrow[d, hook] & A \arrow[d] \\
\square^I \arrow[r] & \Gamma
\end{tikzcd}$$

We are using the $\sqcup_\varphi$ notation without justification here. For our purposes we need a slight restriction so that $\sqcup_\varphi$ is actually what is called an open box shape on I. We will require φ to be the disjunction of every face

With this in mind, we describe the concept of a uniform Kan fibration.

Definition 3.2.18
Let $p : A \to \Gamma$ be a morphism of cubical sets. If we have, for each open box inclusion $\sqcup_S \to \square^I$, we have fillers in the diagram

$$\begin{tikzcd}
\sqcup_S \arrow[r] \arrow[d, hook] & A \arrow[d] \\
\square^I \arrow[r] \arrow[ur, dashed] & \Gamma
\end{tikzcd}$$

and moreover, we can give a uniform choice of such fillers, we say that p is a uniform Kan fibration. Here uniformity expresses the fact that if we have a map of open box inclusions, we get a diagram

$$\begin{tikzcd}
\sqcup_S \arrow[r] \arrow[d, hook] \arrow[rr, bend left] & A \arrow[d] & \sqcup_R \arrow[l] \arrow[d, hook] \\
\square^I \arrow[r] \arrow[ur, dashed] \arrow[rr, bend right] & \Gamma & \square^K \arrow[l] \arrow[ul, dashed]
\end{tikzcd}$$

and we require that this diagram commutes. This ensures that the lift in the left diagram coincides with the one we obtain by composing the bottom map with the filler in the right diagram, hence uniformity. ○

The composition operation can be applied in a wider range of scenarios. We want to apply the Kan lifting condition in a case where we have a system on φ which is extensible at $i0$, and be able to conclude that it is also extensible at $i1$. Without going into detail on this extension matter, we note that it can be obtained by iteratively applying the lifting condition presented above.

We mentioned earlier that the cube category is a strict test category, and hence the category of cubical sets has a test model structure and it is Quillen equivalent to the category of simplicial sets with this model structure. One could ask whether the uniform Kan fibrations fit into a model structure on cubical sets. This seems to be the case, although I could not find a reference for this fact apart from a discussion in the HoTT Google group[2]. On the other hand we have a weaker theorem. It can be shown that the full subcategory of fibrant cubical sets form a path category, which is a slight strengthening of the concept of categories of fibrant objects described in [2]. A path category needs a notion of fibration, this role is played by uniform Kan fibrations, path objects, which can be chosen to be the path objects we defined above, and weak equivalences. With the path objects and the fibrations we can construct the homotopy relation on maps, and hence the notion of a homotopy equivalence of cubical sets, following the well known construction of these relations in model categories. Since any path category is saturated, which means simply that homotopy equivalences and weak equivalences must coincide), this is sufficient to give fibrant cubical sets a path category structure.

The results mentioned above are mostly proven for different presentations of cubical sets. One can verify in each case that the results generalize to this de Morgan algebraic representation of cubes.

The point of introducing the concept of uniformly Kan cubical sets is that a uniform choice of fillers is equivalent to a composition structure compatible with substitutions. We can then restrict the model of general dependent type theory with an interval primitive to those types, or maps of presheaves $A \to \Gamma$, which are uniform Kan fibrations. Interpreting the composition structure is done by directly translating its definition, as we saw for both Σ and Π types. Since this interpretation requires a choice of fillers, it is convention to consider instead of types, fibrant types, which are simply pairs of types and a composition structure. Here type is taken in the CwF sense, i.e., we are considering composition structures on maps $A \to \Gamma$.

[2] Discussion of the model structure on a certain presentation of cubical sets can be found here`https://groups.google.com/forum/#!topic/homotopytypetheory/RQkLWZ_83kQ`

Proposition 3.2.19
A composition structure is equivalent to Kan lifting condition.

We have already covered the connection between these concepts, so we leave out the verification here.

Proposition 3.2.20
All isomorphisms are uniformly Kan, and uniform Kan fibrations are closed under composition, pullback and dependent product.

Proof. For isomorphisms the inverses provide unique, hence uniform, lifts, and for compositions we can lift along each map one at a time, with uniformity following from uniformity of each map. For pullbacks, consider a diagram

$$\begin{tikzcd}
\sqcup_S \arrow[r] \arrow[d, hook] & A \arrow[r] \arrow[d] & B \arrow[d] \\
\square^I \arrow[r] & C \arrow[r] & D
\end{tikzcd}$$

In which the right square is a pullback and the map $B \to D$ is a uniform Kan fibration. We the have a lift in the outer square, i.e., a map $\square^I \to B$, and since this is a lift, we now have compatible maps $\square^I \to C$ and $\square^I \to B$, which induce a map $\square^I \to A$ by the pullback property. Since the lift was obtained via a universal property, uniformity of the lifts thus produced follow from the uniformity condition on $B \to D$.

We leave out the dependent product in this proof. □

Theorem 3.2.21
The interpretation of CTT into the category of cubical sets is sound, and satisfies the substitution lemma.

We leave out the soundness proof for cubical type theory. For the substitution lemma we note again that path types follow the scheme of the dependent product, so it will be sufficient to show that the interpretation of the composition operation is compatible with substitutions. But this is exactly the reason we defined it as we did, and the reason that it corresponds to a uniform filling condition in the model. Indeed, uniformity of the lifting condition is equivalent to the substitution lemma for the composition operation.

The primary motivation for cubical type theory is that it provides a constructively valid proof of soundness of the univalence axiom. There are two missing pieces from the presentation given here that one needs to show that cubical type theory supports the univalence axiom, namely the glueing operation and the universes. We will not discuss these operations, and will be content to have an implementation of an intensional identity type, although if we wanted the full power of the theory presented here, and the one which we will present in the next chapter, this would be the direction to go.

Furthermore, we note that a significant part of the motivation for pursuing theories with an intensional identity type was the decidability of type checking. Type checking in cubical type theory has an implementation in Haskell [7], which instils some confidence that we have decidable type checking. So far we have treated decidable type checking as an obvious consequence of intensionality of the identity type, but this is not the case. The only thing we have show so far is that there is no hope for decidable type checking when this is not the case, but there is significant work involved in showing the positive statement.

CHAPTER 4

Path orthogonality

As we saw in the previous chapter, it is possible to implement coinductive types in type theory via guarded recursion. The specific implementation we gave had two problems, both related to the fact that the theory was extensional. Firstly, the theory does not have decidable type-checking. If this was the only issue, it would maybe not warrant an investigation into alternatives; decidable type-checking means that you can be sure that type-checking terminates, but there is no way to bound the time it takes. The other problem is one internal to the theory. We saw some examples of extensionality principles in the section on cubical type theory, and these are missing from the guarded recursion section.

In addition to these already established extensionality principles, the interaction between cubical and guarded dependent type theory allow us to construct extensionality principles related to the later modality. This has been investigated in the single clock case in [4]. Our goal will be to instead give a presentation of a guarded dependent type theory with multiple clocks and an interval type.

In this chapter we will present a theory which is a mix of the cubical and guarded dependent theories. We retrace some of the steps we took in each of these theories to verify that we have retained the properties that we are interested in. In particular we restate the clock irrelevance axiom in this new setting.

We will then give a foundation for denotational semantics for this theory in the presheaf category $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}}$. In particular we show that the clock irrelevance axiom in this setting can be modelled as an orthogonality condition up to a path.

4.1 Clock irrelevance revisited

We consider now a theory which supports each type we have seen so far. It is easiest to think of it as being constructed in steps. We take the same basic dependent type theory we have been in the previous sections, with just Π types, Σ types and natural numbers. To this we add the constructs of GDTT, in particular we have

- a clock primitive clock, variables of this primitive $\kappa : \mathsf{clock}$;
- the later type former $\triangleright^\kappa$, and a constructor for this type, next^κ;
- the clock quantification type former $\forall\kappa$, an introduction rule for this type given by $\Lambda_{\mathsf{clock}}\kappa.$ and a limited inverse to next^κ given by $\mathsf{prev}\kappa.$;
- a fixpoint combinator fix^κ.

We require that the above to satisfy the judgemental equalities from GDTT, except for the clock irrelevance axiom. We then add the CTT constructs:

- The interval primitive $\mathbb{I}$, variables of this primitive $i : \mathbb{I}$ and formulas of the face lattice $\varphi : \mathbb{F}$ and systems based on such formulas;

- the path type, with lambda abstraction and path application as introduction and elimination rules respectively;
- A composition structure inductively defined on each of the types *except* $\triangleright^\kappa A$.

Again we require all the same judgemental equalities to hold. Right now there are two things missing from the theory, namely a composition structure on types of the form $\triangleright^\kappa A$ and a clock irrelevance axiom. Note that the composition structure on $\forall\kappa.A$ is obtained in the same way as we did for Π types, and that it would not be sensible to require a composition structure on clock, as it is a primitive.

The central axiom of our theory is the pathed clock irrelevance axiom, and the primary goal of the thesis is to obtain a model for the theory in which this rule is modelled by an orthogonality condition.

Definition 4.1.1
The pathed clock irrelevance axiom (henceforth the PCI axiom) is the following inference rule:

$$\frac{\Gamma \vdash t : \forall\kappa.A \qquad \Gamma \vdash A\,\mathbf{type} \qquad \Gamma \vdash \kappa' : \mathsf{clock} \qquad \Gamma \vdash \kappa'' : \mathsf{clock}}{\Gamma \vdash \mathsf{cirr}_A(t) : \mathsf{Path}_A(t\kappa')(t\kappa'')}$$

We furthermore require that $\mathsf{cirr}_A(t)$ is compatible with substitutions, in the sense that $\mathsf{cirr}_A(t)\sigma \equiv \mathsf{cirr}_{A\sigma}(t\sigma)$ for any substitution σ. As before, we say that the type, A, satisfies the PCI axiom if such a $\mathsf{cirr}_A(t)$ exists for each $t : \forall\kappa.A$. ○

The primary motivation for studying variants of guarded type theories remains a desire to define coinductive types, in particular streams, and so we want an equivalence as in the following proposition.

Proposition 4.1.2
Assume that $\Gamma \vdash A\,\mathbf{type}$, and $\Gamma \vdash \kappa : \mathsf{clock}$, noting that this means that κ does not occur in A. Then there is an equivalence of A and $\forall\kappa.A$, i.e., a non-dependent map $f : A \to \forall\kappa A$, which has an inverse.

Proof. We can define f by $\lambda a.(\Lambda_{\mathsf{clock}}\kappa.a)$, or in words, a gets mapped to the constant function at a. The inverse will be evaluation at a specific clock, say some κ_0, which is again added to the language as a constant. We denote this function by g and define it formally as $\lambda t.(t\kappa_0)$.

First we note that $gfa \equiv (\Lambda_{\mathsf{clock}}\kappa.a)(\kappa_0) \equiv a[\kappa/\kappa_0] \equiv a$, where the last equality follows from the fact that κ does not occur in the term a. This means that gf is pointwise judgementally equal to the identity function, and is hence itself judgementally equal to the identity function.

It remains to show that we have some $p : \mathsf{Path}_{\forall\kappa.A}(fg)(\mathrm{id})$. We want to employ essentially the same strategy as above, but in this case judgemental equalities are too much to ask for. Instead we have by PCI for A a path at each point, which we lift to a path of functions by function extensionality for path types. Specifically we can evaluate $fgt \equiv \Lambda_{\mathsf{clock}}\kappa.(t\kappa_0)$. What we want at this point is, for each $\kappa' \notin \Gamma$ a path from $t\kappa_0$ to $t\kappa'$. But this is exactly what PCI ensures the existence of, and so we are done. □

The above proof is one example in which we can more or less directly translate from the GDTT setting. The difference was that we used function extensionality instead of the η rule for functions. The two other lemmas we needed to give the definition of streams are still valid on the nose, since they used only judgemental equalities which the present theory inherited from GDTT.

We are now ready to present the extensionality principle specific to the GDTT setting.

Proposition 4.1.3
We have an equivalence of types between $\mathsf{Path}_{\rhd^\kappa A}(\mathsf{next}^\kappa a)(\mathsf{next}^\kappa a')$ *and* $\rhd^\kappa \mathsf{Path}_A(a)(a')$.

The analogous result in the single clock case is proved in [4]. We will not be able to give a proof for this, as it requires a composition structure on $\rhd^\kappa A$. We have not spent a great deal of time on the fix^κ combinator, but note here that the theory may allow for a relaxation of the requirement on the unfolding of fixpoints, as is conjectures in [4].

4.2 Orthogonality up to a path

The semantics for this theory will, as the syntax was, be a mix of our two main examples. Concretely, we will consider the presheaf category $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathsf{op}}}$ of two dimensional presheaves. Of course one of these presheaves takes as input iterated pairs, but we just write $X(\mathcal{E},\delta,I)$ for evaluation. Before we start giving definitions of the model itself, we will consider a few special presheaves. Essentially, these are recycled definitions from the previous section, but we reiterate their validity in this new setting.

Definition 4.2.1
We briefly cover the presheaves found in the other models.

- The interval is the presheaf $\mathbb{I} \in \mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathsf{op}}}$ defined by $\mathbb{I}(\mathcal{E},\delta,I) = \mathsf{dM}(I)$
- The clock object is the presheaf $\mathrm{Cl} \in \mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathsf{op}}}$ defined by $\mathrm{Cl}(\mathcal{E},\delta,I) = \mathcal{E}$.

○

As before, the interpretation of the syntactic clock primitive is the object of clocks, and the interpretation of the syntactic interval is the interval presheaf. Consequently, we can model clock quantification and path types as we do dependent function spaces, in the same way that we presented in the respective models earlier.

We can also reuse the interpretation of $\rhd^\kappa$ and next^κ simply by ignoring the cubical dimension, i.e., we define

$$[\![\rhd^\kappa A]\!](\mathcal{E},\delta,I,\gamma) = \begin{cases} \{\star\} & \text{if } \delta(\gamma(\kappa)) = 0 \\ [\![A]\!](\mathcal{E},\delta^{-\gamma(\kappa)},I,\mathsf{tick}^{\gamma(\kappa)}\gamma) & \text{otherwise} \end{cases}$$

with tick defined as before. The interpretation of next is again analogous.

We can now give the orthogonality condition which will model the PCI axiom.

Definition 4.2.2
Let $p : A \to \Gamma$ be a morphism in $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cubeop}}$. We say that p is path orthogonal to X, if there is a uniform choice of lifts in diagrams of the shape

$$\begin{tikzcd}
\Delta \times 2 \arrow[r, "{(a,b)}"] \arrow[d, hook] & \Delta \times X \arrow[r, "t"] \arrow[d] & A \arrow[d] \\
\Delta \times \mathbb{I} \arrow[r] \arrow[urr, dashed] & \Delta \arrow[r, "\sigma"] & \Gamma
\end{tikzcd}$$

where the unlabelled morphisms are projections. Uniformity here means that for any $\tau : \Delta \to \Theta$, the lift produced in Δ is the same as the one obtained from the fact that a lift for Θ precomposed by τ is a lift for Δ. ∘

The uniformity condition is the semantic counterpart to compatibility with substitutions, as we saw in the case of cubical type theory.

The goal is to obtain a model for our type theory as a subcategory of $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathrm{op}}}$. The following lemma means that the only way that this is possible, is if we restrict at least to presheaves that are path orthogonal to the object of clocks.

Lemma 4.2.3

Assume that $\Gamma \vdash A\,\mathbf{type}$, $\Gamma \vdash \kappa : \mathsf{clock}$ *and consider some* $\Gamma \vdash t : \forall\kappa.A$. *The PCI axiom for A implies the existence of a lift in the following diagram:*

$$\begin{tikzcd}
\llbracket\Gamma\rrbracket \times 2 \arrow[r, "\llbracket\kappa'\rrbracket\times\llbracket\kappa''\rrbracket"] \arrow[d, hook] & \llbracket\Gamma\rrbracket \times \mathrm{Cl} \arrow[r, "\llbracket t\rrbracket"] & \llbracket A\rrbracket \arrow[d, "p_A"] \\
\llbracket\Gamma\rrbracket \times \mathbb{I} \arrow[rr] \arrow[urr, dashed] & & \llbracket\Gamma\rrbracket
\end{tikzcd}$$

with the diagram constructed using the interpretations of the assumed terms and types.

Proof. The interpretation of a path in A as a type is a map $\llbracket\Gamma\rrbracket \times \mathbb{I} \to \llbracket A\rrbracket$ over $\llbracket\Gamma\rrbracket$ in the model, so it only remains to check that it has the correct endpoints. But since the square

$$\begin{tikzcd}
\llbracket\Gamma\rrbracket \times 2 \arrow[r, "\llbracket\kappa'\rrbracket\ \llbracket\kappa''\rrbracket"] \arrow[d, hook] & \llbracket\Gamma\rrbracket \times \mathrm{Cl} \arrow[d, "\llbracket t\rrbracket"] \\
\llbracket\Gamma\rrbracket \times \mathbb{I} \arrow[r] & \llbracket A\rrbracket
\end{tikzcd}$$

commutes, the endpoints of the path are determined and equal to the desired points in A. Here we are using the same characterization of the path types given in the section on cubical type theory, which remains valid in this new model. □

This hints at the existence of a model of the theory in $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathrm{op}}}$ among those types for which the context projections are path orthogonal to the object of clocks. In the section on GDTT we did this the other way around; first we showed that the context projections were orthogonal to the clock object, and then we used this to show the soundness of the model. Here we are assuming without roof that we can model our type theory in a way which ensures that the context projections are path orthogonal to the object of clocks, and as we saw above, this would be sufficient to see that the PCI axiom is valid in the model. The first step towards providing the model we want is to reprove the closure lemma for path orthogonality. The proof is similar to the one for orthogonality, but we review it in this new setting for completeness.

Lemma 4.2.4

Any isomorphism is path orthogonal to any other object. Furthermore, path orthogonality to X is closed under composition with maps also path orthogonal to X, any dependent product and any pullback.

Proof. For the isomorphisms, note that the inverse can be used to construct the desired lift. For the composition, we construct the lift in stages.

Consider some map $p : A \to B$ which is path orthogonal to X, and some morphism $f : C \to B$, and denote the pullback of A along f by f^*A. Assume that we have a commuting diagram

$$\begin{tikzcd}
\Delta \times 2 \arrow[r] \arrow[d] & \Delta \times X \arrow[r] \arrow[d] & f^*A \arrow[d] \\
\Delta \times \mathbb{I} \arrow[r] & \Delta \arrow[r] & C
\end{tikzcd}$$

By the pullback property it is enough to produce compatible map $\Delta \times \mathbb{I} \to C$ and $\Delta \times \mathbb{I} \to A$. Extending the diagram to include p allows us to apply path orthogonality to obtain the map $\Delta \times \mathbb{I} \to A$, and the map $\Delta \times \mathbb{I} \to C$ is given by postcomposing the bottom row by f. These maps are compatible, and the uniformity of the lifts obtained this way follows from the uniformity condition on p.

Assume now that we have some $f : B \to C$. We want to show that $\Pi_f(p)$ is path orthogonal to X. As for the non-pathed case, we first apply 1.1.12, to see that lifts in the above diagram correspond naturally to lifts in the following:

$$\begin{tikzcd}
(\Delta \times_C B) \times 2 \arrow[r] \arrow[d] & (\Delta \times_C B) \times \mathrm{Cl} \arrow[r] \arrow[d] & A \arrow[d] \\
(\Delta \times_C B) \times \mathbb{I} \arrow[r] & \Delta \times_C B \arrow[r] & B
\end{tikzcd}$$

wherein again we need similar arguments to last time to see that the morphisms have the correct form. □

We have not yet shown that the interpretation of $\rhd^\kappa A$ is path orthogonal to the object of clocks. in the case of GDTT, we used the lemma which establishes a connection between orthogonality and a certain square being a pullback, 1.1.10. A step towards this proof might therefore be to modify this lemma so that it holds in this setting.

We have not at this point given the interpretation for the composition operation. The direct translation of the syntactical requirements might still work, but there is no longer a clear connection to the uniform Kan condition on morphisms of presheaves. To remedy this, we would need to study in greater detail the connection between the composition structures and the Kan condition in terms of path categories, or a similar homotopical formalism. With an interpretation based on this, we might be able to rediscover the connection simply by giving the category $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathrm{op}}}$ a path category structure. If we had a clear description of the model structure on cubical sets, we could consider for instance the projective model structure on $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathrm{op}}}$, since we have an equivalence of categories $\mathsf{Set}^{\mathbb{T}\times\mathsf{Cube}^{\mathrm{op}}} \simeq \mathsf{Fun}\left(\mathbb{T}, \mathsf{Set}^{\mathsf{Cube}^{\mathrm{op}}}\right)$.

Conclusions and further work

In this thesis we saw how to construct a basic type theory. We related it to both the propositional interpretations of type theory via the Curry-Howard isomorphism, and we briefly covered some construction of homotopy type theory. We presented a model of our type theory in a presheaf category, and discussed why this model was insufficient for modelling intensional type theory.

We saw two extensions of type theory; one focused on the concept of recursion and one focused on implementing a version of the homotopy type theory discussed earlier. In the section on guarded dependent type theory, we saw how one could implement recursion in a type theory with a $\triangleright^{\kappa}$ modality and multiple clocks. Furthermore, we demonstrated how one could in principle reason about coinductive types via the guarded recursion and clock quantification notions of this theory, by considering the example of guarded streams. We discussed the denotational semantics for this theory in a certain presheaf category.

In the discussion of cubical type theory, we saw how one could implement a path type as a modification of the function spaces with codomain a syntactic interval. A composition structure on the types of the theory was used to show that the path type was a suitable replacement for the abstract identity type of the standard intensional Martin-Löf type theory. Cubical sets were introduced and used to model the theory; in particular we related the syntactical composition operation to the uniform Kan lifting condition.

In the final section we saw a lifting condition with relations to the orthogonality principle discussed in relation to the guarded dependent type theory. We saw how this lifting condition could be used to model a weakening of the clock irrelevance axiom to a statement of path types from one of judgemental equalities. We discussed possible extensionality principles, and part of a proof that the theory could be soundly modelled in an iterated presheaf category. This model can be thought of as an extension of the model presented in [5] to include the constructs of cubical type theory, and a partial answer to the question posed at the end of [4] of whether one can construct a model with multiple clocks and path types.

The first step in any further work along this line would be to study the model presented in the last chapter in more detail. In particular, we do not have a composition structure on $\triangleright^{\kappa} A$ at the moment, and this is needed for the theory to be sound. This would also be the first step towards the extensionality principles we proposed; we would need a clearer picture of the interaction between the time and cube dimensions of both the model and the theory before we could give a proof.

It would be instructive to study the model on its own terms in further details. In particular, the Kan lifting condition of simplicial sets fits into a model structure on simplicial sets, and the model of Voevodsky is compatible with this model structure. We are considering an iterated presheaf category, so assuming a model structure on cubical sets based on the uniform Kan condition, we can consider at least the projective and injective model structures on this category. If we are once again interested in fibrant types, the projective model structure might be the more suitable one, given that the fibrations of this model structure are easily understood in terms of the input model structure. We might then ask whether one could find a model of the theory discussed in the last chapter among the projectively fibrant types which are also path orthogonal to the object of clocks.

Once the above is finished, one could ask whether we could generalize the construction in the guarded dependent type theory, i.e., we could ask the question of which theories could be enhanced with the cubical concepts as we proposed to do for the guarded theory. This could be a powerful tool for adding intensional identity types to a variety of theories. If we furthermore extended the work done in this thesis to include univalence and glueing from cubical type theory, we could try to give a general strategy fr including most of the ideas of homotopy type theory in many other theories soundly.

Acknowledgements

I would like to thank my informal supervisor, Rasmus Ejlers Møgelberg, without whom this project would not have been possible. He answered my many questions, and provided me with all the support I needed during the project. I would also like to thank Jesper Michael Møller for his faith in me, and for backing this project and in particular for taking care of the formal procedures.

Bibliography

[1] Robert Atkey and Conor McBride. "Productive Coprogramming with Guarded Recursion". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 197–208. ISSN: 0362-1340. DOI: 10.1145/2544174.2500597. URL: http://doi.acm.org/10.1145/2544174.2500597.

[2] I. Moerdijk B. van den Berg. "Exact completion of path categories and algebraic set theory – Part I: Exact completion of path categories". In: (2016).

[3] Lars Birkedal et al. "First steps in synthetic guarded domain theory: step-indexing in the topos of trees". In: *Logical Methods in Computer Science* 8.4 (2012). DOI: 10.2168/LMCS-8(4:1)2012. URL: https://doi.org/10.2168/LMCS-8(4:1)2012.

[4] Lars Birkedal et al. "Guarded Cubical Type Theory: Path Equality for Guarded Recursion". In: *CoRR* abs/1606.05223 (2016). arXiv: 1606.05223. URL: http://arxiv.org/abs/1606.05223.

[5] Ales Bizjak and Rasmus Ejlers Møgelberg. "Denotational semantics for guarded dependent type theory". In: *CoRR* abs/1802.03744 (2018). arXiv: 1802.03744. URL: http://arxiv.org/abs/1802.03744.

[6] Ulrik Buchholtz and Edward Morehouse. "Varieties of cubical sets". In: *International Conference on Relational and Algebraic Methods in Computer Science*. Springer. 2017, pp. 77–92.

[7] Cyril Cohen et al. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: *CoRR* abs/1611.02108 (2016). arXiv: 1611.02108. URL: http://arxiv.org/abs/1611.02108.

[8] Martin Hofmann. "Syntax and semantics of dependent types". In: *Extensional Constructs in Intensional Type Theory*. Springer, 1997, pp. 13–54.

[9] Martin Hofmann, Thomas Streicher, et al. "Lifting grothendieck universes". In: ().

[10] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[11] Peter T Johnstone. *Sketches of an elephant: a Topos theory compendium*. Oxford logic guides. Oxford Univ. Press, 2002.

[12] Chris Kapulkin and Peter LeFanu Lumsdaine. "The simplicial model of univalent foundations (after Voevodsky)". In: *arXiv preprint arXiv:1211.2851* (2012).

[13] Hiroshi Nakano. "A Modality for Recursion". In: *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000)*. Santa Barbara, CA, USA: IEEE Computer Society Press, 2000, pp. 255–266.

[14] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.

[15] Andrew Swan. "An algebraic weak factorisation system on 01-substitution sets: A constructive proof". In: *arXiv preprint arXiv:1409.1829* (2014).

[16] Benno Van Den Berg and Richard Garner. "Types are weak ω-groupoids". In: *Proceedings of the London Mathematical Society* 102.2 (2011), pp. 370–394.